Мэтт
Фрисби

JavaScript

для
профессиональных
веб-разработчиков

4-е международное
издание

PROFESSIONAL

JavaScript® for Web Developers

Matt Frisbie

A Wiley Brand

Мэтт Фрисби

JavaScript

для профессиональных веб-разработчиков

4-е международное издание

ПИТЕР®

Санкт-Петербург • Москва • Минск

2022

ББК 32.988.02-018.1
УДК 004.738.5
Ф89

Фрисби М.

Ф89 JavaScript для профессиональных веб-разработчиков. 4-е международное изд. — СПб.: Питер, 2022. — 1168 с.: ил. — (Серия «Для профессионалов»).

ISBN 978-5-4461-1740-6

Самое полное руководство по современному JavaScript.

Как максимально прокачать свои навыки и стать топовым JS-программистом? Четвертое издание «JavaScript для профессиональных веб-разработчиков» идеально подойдет тем, кто уже имеет базовые знания и опыт разработки на JavaScript. Автор сразу переходит к техническим деталям, которые сделают ваш код чистым и переведут вас с уровня рядового кодера на высоту продвинутого разработчика.

Рост мобильного трафика увеличивает потребность в адаптивном динамическом веб-дизайне, а изменения в JS-движках происходят постоянно, так что каждый веб-разработчик должен постоянно обновлять свои навыки работы с JavaScript.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.988.02-018.1
УДК 004.738.5

ISBN 978-1119366447 англ.
ISBN 978-5-4461-1740-6

Краткое содержание

Оглавление

Джордан за ее непоколебимую поддержку,
несмотря на все мои «уже почти готово».

Об авторе

МЭТТ ФРИСБИ (MATT FRISBIE) занимается разработкой веб-приложений более десяти лет. За это время он побывал соучредителем стартапа, инженером в технологической компании «Большой четверки» и первым инженером стартапа Y Combinator, который в итоге превратился в бизнес на миллиард долларов. Как инженер-программист Google Мэтт работал на платформах AdSense и Accelerated Mobile Pages (AMP); его код работает на большинстве веб-браузеров планеты. До этого он был первым инженером в DoorDash, где заложил основу для планирования обновлений их драйверов, управления меню и инфраструктуры распределения заказов. Мэтт написал две книги и выпустил две серии видео для O'Reilly и Packt. Он выступает на встречах и вебкастах фронтендщиков, и к тому же является первоклассным сомелье. В Twitter Мэтта можно найти под ником @mattfriz.

О научных редакторах

ХАИМ КРАУЗЕ (CHAIM KRAUSE) — любитель компьютеров, электроники, животных и электронной музыки. Больше всего он радуется, когда удается объединить два или более этих интересов в одном проекте. Почти всему Хаим обучился самостоятельно. Он в шутку говорит всем, что единственная разница между его домашними занятиями и работой — это используемый вход в систему. Как вечного студента, его часто раздражают технические ошибки в документации, которые отнимают драгоценное время и вызывают пустую досаду. Одна из причин, по которой он работает научным редактором технических книг — помочь другим избежать тех же ошибок.

МАРСИЯ УИЛБУР (MARCIA WILBUR) — технический писатель. Она дает консультации в области полупроводников и специализируется на индустриальном интернете вещей (IIoT) и ИИ. Марсия имеет ученые степени в области информатики, технических коммуникаций и информационных технологий. В качестве президента Copper Linux User Group, она активно участвует в сообществе разработчиков: возглавляет West Side Linux + Pi и East Valleyи ведет регулярные проекты Raspberry Pi, Beaglebone, Banana Pi/Pro и ESP8266. Эти проекты включают в себя домашнюю автоматизацию, игровые приставки, видеонаблюдение, сеть, мультимедиа и другие «Pi-развлечения».

Помимо работы, она является волонтером в различных организациях, использующих Pi-модули и Linux, чтобы обеспечить доступ к образовательному контенту для школ в сельских, недостаточно обслуживаемых и пострадавших от стихийных бедствий районах. Ради интереса она служит сообществу в роли ведущего разработчика Debian для Linux Respin, инструмента для резервного копирования и настройки дистрибутива.

Благодарности

Спасибо Wiley, что позволили мне заняться этим проектом. Написание четвертого издания «JavaScript для професиональных веб-разработчиков» было одним из наиболее сложных, но в то же время интересных проектов, над которыми я когда-либо работал. Эта книга не вышла бы в свет без терпения и поддержки от Wiley. Спасибо сотрудникам Wiley, в частности Джиму Минателю (Jim Minatel), который передал этот проект в мои руки и довел его до конца.

Хотел бы поблагодарить Николаса Закаса (Nicholas Zakas), автора первых трех изданий, за всю работу, которую он проделал до моего участия. Книга не получилась бы настолько удачной без железного фундамента, заложенного им. Я желаю ему скорейшего выздоровления.

Отдельное спасибо Адаоби Оби Тултон (Adaobi Obi Tulton) за ее наставничество. Ее участие было бесценным на протяжении всего процесса, и я не справился бы без ее терпения и опыта.

Я также хотел бы поблагодарить всех, кто вычитывал рукопись: Сэмюэля Каллнера (Samuel Kallner), Хаима Краузе (Chaim Krause), Марсию Уилбур (Marcia Wilbur), Нэнси Рапопорт (Nancy Rapoport), Атияппан Лалит Кумар (Athiyappan Lalith Kumar) и Эвелин Веллборн (Evelyn Wellborn). Книга вроде этой просто развалилась бы без вашего участия.

Наконец, хотел бы поблагодарить Зака Тратара (Zach Tratar) за предисловие. Мне посчастливилось встретиться с Заком в тот же день, когда я переехал в Сан-Франциско. За прошедшие годы он проявил себя как безумно эрудированный и чрезвычайно приятный человек, не говоря уже о том, насколько он хороший инженер программного обеспечения. Я считаю большой честью то, что он согласился внести свой вклад в эту книгу.

Предисловие

Промышленная революция родилась из производства стали, а интернет-революция — из JavaScript. Выкованное и закаленное за счет постоянных итераций в течение последних 25 лет, доминирование JavaScript в разработке приложений в настоящее время трудно поставить под сомнение, но так было не всегда.

Брендану Эйху (Brendan Eich) потребовалось всего десять дней, чтобы создать первую версию JavaScript. Она казалась хрупкой, но, как показывает история, первое впечатление обманчиво. Сегодня каждый аспект JavaScript — каждая деталь, о которой вы узнаете в этой книге, — является результатом долгих дискуссий. Не каждое решение идеально — в конце концов идеального языка программирования не существует. Но если судить только по одному его повсеместному распространению, JavaScript может приблизиться к идеалу. Это единственный язык, который можно применить везде: серверы, настольные браузеры, мобильные веб-браузеры и даже собственные мобильные приложения.

JavaScript теперь используется разработчиками всех направлений и уровней опыта: от тех, кто заботится о хорошо спроектированном, элегантном ПО, до тех, кому просто нужно по-быстрому собрать рабочий вариант для достижения бизнес-целей.

Как вы будете его использовать, зависит только от вас. Эта сила ваша.

За последние 15 лет разработки ПО инструменты JavaScript и его лучшие методики сильно изменились. Мой опыт работы с языком начался в 2004 г., когда доминировали Geocities, Yahoo Groups и Macromedia Flash player. JavaScript был похож на игрушку, и я играл в некоторых из популярных в то время песочниц: RSS и MySpace Profile Pages. Помощь другим в изменении и настройке личных сайтов показалась мне настоящим Диким Западом и в итоге зацепила меня.

Когда я запускал свою первую компанию, настройка хоста для базы данных занимала несколько дней, и JavaScript был встроен в HTML. Не было никаких «приложений» внешнего интерфейса — все это в основном представляло из себя бессистемные функции. По мере того как Ajax, возглавляемый jQuery, становился все более популярным, развивался новый мир с более надежными приложениями. Это движение набрало головокружительную скорость, а затем неожиданно были выпущены мощные фреймворки. Модели внешнего интерфейса! Привязка данных! Управление маршрутами! Реактивное отображение! Именно во время этой фронтенд-революции я переехал в Кремниевую долину, чтобы помочь запустить

компанию, основанную Леди Гагой, и вскоре миллионы пользователей начали использовать мой код. Находясь в Кремниевой долине достаточно долго, я внес свой вклад в проекты с открытым исходным кодом, обучил больше разработчиков, чем могу сосчитать, и поймал удачу за хвост. Моя последняя компания была приобретена Stripe в 2018 г., где я сейчас работаю над созданием экономической инфраструктуры для интернета.

Я имел удовольствие встретиться с Мэттом в тот день, когда он впервые вылетел в Пало-Альто, чтобы возглавить инжиниринг при небольшом стартапе Claco: в то время я только присоединился к нему в качестве советника. Энергия и страсть Мэтта к отличному ПО были очевидны, и молодая компания быстро выпустила прекрасный продукт. Как было принято в Кремниевой долине со времен HP, этот стартап зародился у кого-то дома. Но это не был обычный дом. Это был «хакерский дом», где постоянно жили десять или около того блестящих разработчиков ПО. Хотя это была не жизнь высшего класса — двухъярусные кровати и стулья, выброшенные кем-то и затем подобранные на улице, были обычным явлением, — количество и качество ежедневно пишущегося там кода поражало воображение. В нерабочее время большинство разработчиков просто переключали внимание и занимались своими сторонними проектами еще пару часов. Новички, не умевшие писать код, часто вдохновлялись: у них возникало желание учиться и за несколько недель они осваивали программирование.

Мэтт был движущей силой этого сосредоточия продуктивности. Он был самым опытным инженером-программистом в доме, а также оказался самым неиспорченным и профессиональным. Диплом в сфере компьютерной инженерии тогда не был чем-то привычным, поэтому при виде алгоритмов, расчетов производительности и кода, написанных на окнах или на доске, можно было догадаться, что в этот момент Мэтт работает над следующим большим проектом. Со временем мы стали близкими друзьями. Его ум, любовь к наставничеству и способность переводить большинство вещей в шутку — качества, которыми я восхищался.

Мэтт невероятно талантливый инженер-программист и лидер; именно его уникальный набор опыта и знаний делает его одним из самых квалифицированных людей в мире для написания этой книги.

Он не просто проводил время, обучая других, — он сам работал.

В Claco он разработал множество полноценных продуктов, чтобы помочь учителям улучшить учебный процесс в своих классах. В DoorDash, будучи первым инженером в компании, он создал надежную сеть логистики и доставки, которая достигла небывалого роста и теперь стоит свыше двенадцати миллиардов долларов. Наконец, программное обеспечение Мэтта, применяемое в Google, используют миллиарды людей по всему миру.

Огромное владение, огромный рост и огромный масштаб. Большинству разработчиков за всю карьеру удается достичь только чего-то одного, если им повезет. Мэтт не только добился всего этого, но и стал автором бестселлера, в «свободное

время» написав две другие книги по JavaScript и Angular[1]. Честно говоря, надеюсь, что в его следующей книге будут даны чертежи машины времени, которые он явно скрывает от нас.

Эта книга — надежный инструмент, наполненный знаниями по JavaScript и реальными перспективами. Я рад, что вы продолжаете учиться и воплощать свои идеи. Делите книгу на части, делайте заметки и не забывайте открывать редактор кода — в конце концов, интернет-революция еще только начинается!

Зак Тратар,
инженер-программист в Stripe,
бывший соучредитель и генеральный директор Jobstart

[1] *Мэтт Фрисби*. Angular. Сборник рецептов. Вильямс, 2018 г.

Введение

Техлид в Google однажды поделился со мной убедительным взглядом на JavaScript: это *не совсем* связный язык программирования — по крайней мере, в формальном смысле. Спецификация ECMA-262 *определяет* JavaScript, но единственной истинной *реализации* его не существует. Более того, язык далеко не герметичен. Он плавает в настоящем океане смежных спецификаций, которые управляют API всего, что касается JavaScript: DOM, сетевых запросов, системного оборудования, хранилища, событий, файлов, криптографии и *сотен* других. Веб-браузеры и их различные JS-движки реализуют эти спецификации по своему усмотрению. Chrome использует Blink/V8, Firefox — Gecko/SpiderMoney, а Safari — WebKit/JavaScriptCore. Браузеры будут запускать *почти* весь JavaScript таким образом, который соответствует спецификациям, но в интернете полно примеров идиосинкразий каждого браузера. Поэтому JavaScript более точно характеризуется как совокупность реализаций браузера.

Хотя веб-пуристы могут настаивать на том, что JavaScript не должен быть неотъемлемым компонентом веб-страниц, они должны признать, что современный веб сильно бы без него сократился. Нелишне говорить, что JavaScript практически неизбежен: в телефоны, компьютеры, планшеты, телевизоры, игровые приставки, умные часы, холодильники и даже в автомобили теперь встроены веб-браузеры, поддерживающие JavaScript. Почти три миллиарда человек сейчас используют смартфон с веб-браузером. Живое сообщество языка создает целый поток высококачественных проектов с открытым исходным кодом. Браузеры теперь имеют первоклассную поддержку API, эмулирующих нативные мобильные приложения. В опросе разработчиков Stack Overflow за 2019 г. JavaScript был признан самым популярным языком программирования седьмой год подряд.

Возрождение JavaScript близко.

Эта книга полностью описывает эволюцию JavaScript, начиная с его реализаций в ранних браузерах Netscape и заканчивая современными возможностями, включающими поддержку головокружительного спектра браузерных технологий. Книга охватывает большое количество продвинутых тем в мельчайших деталях, но при этом гарантирует, что читатель будет понимать, как использовать эти темы и где они уместны. Используя полученные значения, вы сможете решать бизнес-задачи, с которыми каждый день сталкиваются многие веб-разработчики по всему миру.

ЦЕЛЕВАЯ АУДИТОРИЯ

Эта книга ориентирована на три группы читателей:

- опытных разработчиков, разбирающихся в объектно-ориентированном программировании и желающих изучить JavaScript в контексте традиционных объектно-ориентированных языков, таких как Java и C++;
- разработчиков веб-приложений, которым нужно сделать свои веб-сайты и веб-приложения более удобными в использовании;
- начинающих разработчиков на JavaScript, желающих лучше понять этот язык.

Кроме того, книга может заинтересовать вас, если вы используете что-то из этого:

- Java;
- PHP;
- Python;
- Ruby;
- Golang;
- HTML;
- CSS.

Эта книга не подойдет вам, если вы не владеете базовыми навыками программирования или всего лишь хотите добавить на веб-сайт простые средства взаимодействия с пользователями. Если вы узнали себя в этом описании, вам лучше обратиться к пятому изданию книги *Beginning JavaScript* (Wiley, 2015).

ТЕМЫ, РАССМАТРИВАЕМЫЕ В КНИГЕ

Книга, которую вы держите в руках, объединяет введение в JavaScript для разработчиков и описание более сложных и полезных возможностей этого языка.

Сначала мы рассмотрим историю и эволюцию JavaScript, после чего подробно обсудим компоненты языка, уделив особое внимание стандартам, таким как ECMAScript и Document Object Model (DOM).

Взяв эту информацию за основу, мы рассмотрим базовые концепции JavaScript, в том числе классы, промисы, итераторы и прокси. Затем перейдем к углубленному изучению обнаружения клиентов, событий, анимации, форм, ошибок и JSON.

Последняя часть книги посвящена новейшим и наиболее важным спецификациям, появившимся за последние несколько лет. Они включают в себя Fetch, модули, веб-работники, рабочие потоки служб и набор новых API.

СТРУКТУРА

В книге 28 глав:

1. **Что такое JavaScript?** Эта глава содержит общие сведения о JavaScript: вы узнаете, как появился этот язык, как он развивался и что представляет собой сегодня. Мы обсудим, как JavaScript соотносится с ECMAScript, объектной моделью документа (DOM) и объектной моделью браузера (BOM). Кроме того, вы ознакомитесь с соответствующими стандартами от Европейской ассоциации производителей вычислительной техники (ECMA) и консорциума World Wide Web (W3C).
2. **JavaScript в HTML.** Описано применение JavaScript в сочетании с HTML для создания динамических веб-страниц. Также рассмотрены различные способы внедрения JS-кода в страницу, типы JavaScript-контента и их использование с элементом `<script>`.
3. **Основы языка.** Рассмотрены базовые концепции языка, в том числе его синтаксис и управляющие инструкции. Указаны сходства и различия JavaScript и других C-подобных языков, а также описано приведение типов в связи со встроенными операторами. Охватывает все языковые примитивы, включая тип `Symbol`.
4. **Переменные, область видимости и память.** Здесь рассказано о переменных, которые в JavaScript являются слабо типизированными. Глава содержит сведения о различиях между примитивными и ссылочными значениями и контексте выполнения в связи с переменными. Вы также узнаете о том, как работает сборщик мусора и как память возвращается среде, когда переменные покидают область видимости.
5. **Ссылочные типы.** Эта глава посвящена встроенным в JavaScript ссылочным типам, таким как `Date`, `Regexp`, примитивы и оболочки примитивов. Для каждого ссылочного типа, определенного в ECMA-262, приведены как теоретические сведения, так и подробности его реализации в браузерах.
6. **Ссылочные типы коллекций.** Продолжает рассмотрение встроенных ссылочных типов с `Object`, `Array`, `Map`, `WeakMap`, `Set` и `WeakSet`.
7. **Итераторы и генераторы.** Представляет две новые фундаментальные концепции из недавней версии ECMAScript: итераторы и генераторы. Каждая из них обсуждается как относительно ее фундаментального поведения, так и относительно ее использования в существующих языковых конструкциях.
8. **Объекты, классы и объектно-ориентированное программирование.** В этой главе рассмотрены приемы использования классов и объектно-ориентированного программирования на JavaScript. Она начинается с углубленного изучения типа `Object` JavaScript и продолжается рассмотрением прототипного наследования. Далее следует полное объяснение классов ES6 и того, почему они являются близкими родственниками прототипного наследования.

9. **Прокси и Reflect.** Данная глава представляет две тесно связанные между собой концепции: Proxy и Reflect API. Они могут использоваться для перехвата и добавления дополнительного поведения в основные операции в языке.

10. **Функции.** Функции-выражения относятся к наиболее мощным аспектам применения JavaScript. В этой главе описываются замыкания, подробности функционирования объекта `this`, паттерн Модуль, создание закрытых членов объектов, стрелочные функции, параметры по умолчанию и операторы расширения.

11. **Промисы и асинхронное программирование.** В этой главе рассмотрены две новые тесно связанные конструкции асинхронного программирования: тип `Promise` и `async/await`. Глава начинается с обсуждения парадигмы асинхронного JavaScript и продолжается обзором использования промисов и их отношений к асинхронным функциям.

12. **Объектная модель браузера.** В этой главе описана объектная модель браузера (BOM), которая предоставляет объекты для взаимодействия с браузером. Вы ознакомитесь со всеми BOM-объектами, включая `window`, `document`, `location`, `navigator` и `screen`.

13. **Распознавание клиента.** В этой главе рассмотрены способы распознавания клиентского браузера и поддерживаемых им функциональных возможностей. Вы узнаете о распознавании возможностей, анализе строки пользовательского агента, о достоинствах и недостатках каждого подхода и о том, какой подход оптимален в той или иной ситуации.

14. **Объектная модель документа.** В этой главе описаны объекты, определенные в спецификации DOM Level 1. После ознакомления с XML в контексте DOM вы сможете подробно изучить модель DOM и предоставляемые ею возможности по манипулированию содержимым страницы.

15. **Расширения DOM.** Глава содержит сведения о том, как API и сами браузеры расширяют функционал DOM. В число рассматриваемых тем входят Selectors, Element Traversal API и расширения HTML5.

16. **DOM Level 2 и 3.** В этой главе, основанной на двух предыдущих главах, рассказано о том, как спецификации DOM Level 2 и 3 расширяют DOM дополнительными свойствами, методами и объектами. Включает охват дополнений DOM4, таких как наблюдатели за изменениями.

17. **События.** Из этой главы вы узнаете о природе JavaScript-событий, их генерировании и о том, как события переопределены в DOM.

18. **Анимация и рисование на холсте.** Эта глава посвящена тегу `<canvas>` и его использованию для динамичного создания графики. Рассмотрены двумерный контекст и контекст WebGL (трехмерный), что поможет приступить к созданию анимаций и игр. Включает обзор WebGL1 и WebGL2.

19. **Работа с формами.** В этой главе рассказывается, как с помощью JavaScript улучшить взаимодействие с формами и обойти ограничения браузера. Особое

внимание уделено работе с элементами форм, такими как текстовые поля и списки, а также проверке и обработке данных.

20. **API в JavaScript.** Рассматривает широкий ассортимент JavaScript API, включая Atomics, Encoding, File, Blob, Notifications, Streams, Timing, Web Components и Web Cryptography.

21. **Обработка ошибок и отладка.** В этой главе рассмотрены способы обработки ошибок в JS-коде. Также описаны инструменты и приемы отладки для каждого браузера и приведены рекомендации по упрощению процесса отладки.

22. **XML в JavaScript.** В этой главе рассмотрены возможности JavaScript, используемые для чтения XML-данных и манипулирования ими. Описаны различия возможностей и объектов в разных веб-браузерах и приведены советы по написанию кроссбраузерного кода. Также в главе приведены сведения об использовании XSLT-преобразований для трансформации XML-данных на клиентских системах.

23. **JSON.** В этой главе представлен формат JSON — альтернатива XML. Описаны возможности синтаксического анализа и сериализации JSON и приведены сведения о том, как обеспечить безопасность при использовании JSON.

24. **Сетевые запросы и удаленные ресурсы.** Данная глава исследует все наиболее распространенные способы запроса данных и ресурсов браузером. Включает в себя рассмотрение унаследованного объекта `XMLHttpRequest`, а также современного Fetch API.

25. **Клиентское хранилище.** В этой главе рассказано о том, как определить, что приложение работает в автономном режиме, и описаны различные методики сохранения данных на клиентском компьютере. В главе рассмотрены как традиционные файлы cookie, так и более новые возможности, такие как вебхранилище и база данных IndexedDB.

26. **Модули.** Здесь обсуждается шаблон Модуль и его влияние на кодовые базы. Затем рассматриваются загрузчики модулей до ES6, такие как CommonJS, AMD и UMD, заканчивается глава подробным описанием нового шаблона ES6 — Модуль — и его правильного использования.

27. **Рабочие потоки.** Эта глава в целом охватывает выделенные рабочие потоки, общие рабочие потоки и рабочие потоки служб. Включает обсуждение того, как рабочие потоки ведут себя на уровне операционной системы и на уровне браузера, а также стратегии оптимального использования различных типов потоков.

28. **Лучшие практики.** Эта глава посвящена использованию JavaScript в корпоративной среде. В ней описаны приемы обслуживания кода, в том числе методики написания и форматирования кода и общие приемы программирования. Также приведены советы по оптимизации и повышению быстродействия кода. Наконец, рассмотрены вопросы развертывания приложений, включая реализацию процесса сборки.

ЧТО НУЖНО ДЛЯ ЭФФЕКТИВНОЙ РАБОТЫ С КНИГОЙ

Для выполнения примеров из книги вам потребуется следующее:

- любая современная операционная система вроде Windows, Linux, MacOS, Android или iOS;
- любой современный браузер вроде IE11+, Edge 12+, Firefox 26+, Chrome 39+, Safari 10+, Opera 26+, or iOS Safari 10+.

Полный исходный код можно загрузить с сайта https://www.wiley.com/en-us/Professional+JavaScript+for+Web+Developers%2C+4th+Edition-p-9781119366447.

ОТ ИЗДАТЕЛЬСТВА

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете приложения из «JavaScript для профессиональных веб-разработчиков», а также подробную информацию о наших книгах.

1

Что такое JavaScript?

- История JavaScript
- Общие сведения о JavaScript
- JavaScript как реализация ECMAScript
- Разные версии JavaScript

Загрузки с сайта wrox.com для этой главы

Обратите внимание, что все примеры кода для этой главы доступны на веб-сайте книги по ссылке www.wrox.com/go/projavascript4e на вкладке Downloads.

Когда в 1995 г. появился JavaScript, его основным назначением была проверка вводимых пользователем данных, что прежде выполняли такие серверные языки, как Perl. Раньше, чтобы определить, не пропущено ли обязательное поле и допустимы ли введенные в форму значения, требовалось обращение к серверу. В Netscape Navigator с помощью JavaScript была предпринята попытка изменить ситуацию. Во времена коммутируемого доступа к интернету возможность выполнять простую проверку на стороне клиента была воспринята с неподдельным энтузиазмом. Из-за низкой скорости подключения каждое обращение к серверу становилось настоящим испытанием терпения пользователей.

За прошедшее время JavaScript стал важным компонентом каждого популярного веб-браузера. Задачи JavaScript больше не ограничиваются простой проверкой данных: теперь он отвечает за взаимодействие почти всех составляющих окна браузера и его контента. JavaScript стал полноценным языком программирования, поддерживающим сложные вычисления и конструкции, включая замыкания, анонимные (лямбда) функции и даже метапрограммирование. JavaScript превратился в такую важную часть Сети, что его поддерживают даже альтернативные браузеры,

в том числе браузеры для мобильных устройств и пользователей с ограниченными возможностями. Даже Microsoft использует собственную реализацию JavaScript в браузере Internet Explorer (с самых ранних версий), несмотря на наличие собственного клиентского языка сценариев VBScript.

Предугадать превращение JavaScript из простого инструмента для проверки вводимых данных в мощный язык программирования было невозможно. Он одновременно и прост, и сложен. Изучить его синтаксис можно за считаные дни, но чтобы научиться применять язык мастерски, требуются многие годы. Чтобы раскрыть полный потенциал JavaScript, важно понимать его природу, историю и ограничения.

КРАТКАЯ ИСТОРИЯ JAVASCRIPT

По мере роста популярности интернета обозначилась потребность в языках сценариев для клиентской стороны. Хотя большинство пользователей подключалось к интернету с помощью модемов на скорости 28,8 Кбит/с, размер и сложность вебстраниц постоянно росли. Хуже того: даже для простой проверки форм требовалось несколько раз обращаться к серверу. Только представьте, каково было заполнить форму, щелкнуть на кнопке отправки, подождать 30 секунд, пока информация будет обработана, и получить сообщение о том, что при вводе данных было пропущено обязательное поле. В компании Netscape, бывшей тогда на рубеже инноваций, начали всерьез задумываться о разработке языка сценариев для простой обработки данных на клиентской стороне.

Брендан Эйх (Brendan Eich), работавший тогда в Netscape, в 1995 г. начал создавать язык сценариев Mocha (позднее переименованный в LiveScript) для браузера Netscape Navigator 2. Предполагалось, что этот язык будет использоваться и в браузере, и на сервере (под названием LiveWire). Чтобы успеть завершить реализацию LiveScript до выпуска браузера, Netscape объединила усилия с Sun Microsystems. Незадолго до выхода Netscape Navigator 2 в компании решили переименовать LiveScript в JavaScript, чтобы попытаться извлечь выгоду из популярности Java.

JavaScript 1.0 оказался очень успешным, и Netscape выпустила его версию 1.1 в составе Netscape Navigator 3. Популярность интернета стремительно росла, и Netscape заслуженно занимала ведущее место на этом рынке. Тем временем в Microsoft решили выделить больше ресурсов на разработку конкурирующего браузера Internet Explorer. Вскоре после выхода Netscape Navigator 3 корпорация Microsoft представила Internet Explorer 3 со своей реализацией JavaScript под названием JScript (чтобы избежать проблем с Netscape, связанных с лицензированием). Вторжение Microsoft в мир веб-браузеров в августе 1996 г. оказалось началом конца Netscape, но в то же время ускорило развитие JavaScript.

То, что JavaScript был реализован в Microsoft, означало, что появилось две версии языка: JavaScript (Netscape Navigator) и JScript (Internet Explorer). В отличие от C и многих других языков программирования, на тот момент не было никаких

стандартов JavaScript, определяющих его синтаксис и функциональность, и существование разных версий языка только подчеркивало эту проблему. Чтобы развеять опасения представителей отрасли, было решено стандартизировать язык.

В 1997 г. спецификация JavaScript 1.1 была принята Европейской ассоциацией производителей вычислительной техники (European Computer Manufacturers Association, Ecma). Был организован Технический комитет № 39 (Technical Committee, TC39), перед которым стояла задача «стандартизировать синтаксис и семантику кроссплатформенного независимого языка сценариев общего назначения» (www.ecma-international.org/memento/TC39.htm). Комитет TC39 объединил программистов Netscape, Sun, Microsoft, Borland, NOMBAS и других компаний, проявляющих интерес к будущему языков сценариев, и за несколько месяцев разработал стандарт ECMA-262, определивший новый язык сценариев с названием ECMAScript.

В следующем году Международная организация по стандартизации (International Organization for Standardization, ISO) и Международная электротехническая комиссия (International Electrotechnical Commission, IEC) также приняли ECMAScript в качестве стандарта (ISO/IEC-16262). С тех пор разработчики браузеров с переменным успехом используют ECMAScript как основу для реализации своих версий JavaScript.

РЕАЛИЗАЦИИ JAVASCRIPT

Хотя названия JavaScript и ECMAScript часто используются как синонимы, JavaScript — это гораздо больше, чем стандарт ECMA-262. Полная реализация JavaScript состоит из трех частей (рис. 1.1):

- ядро (ECMAScript);
- объектная модель документа (Document Object Model, DOM);
- объектная модель браузера (Browser Object Model, BOM).

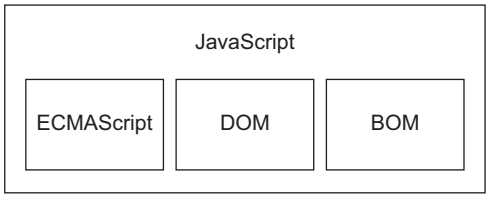

Рис. 1.1

ECMAScript

Сфера применения *ECMAScript* — языка, определенного в ECMA-262, никак не связана с веб-браузерами. На самом деле в нем даже нет методов ввода и вывода данных. Стандарт ECMA-262 определяет этот язык как основу для создания полноценных языков сценариев. Веб-браузеры — это всего лишь одна из *сред выполнения* (host environment), в которых может работать ECMAScript-реализация.

Среда выполнения содержит базовую ECMAScript-реализацию и ее расширения, разработанные для взаимодействия с самой средой. Среди других сред выполнения можно отметить NodeJS (серверная JavaScript-платформа) и все больше и больше устаревающий Adobe Flash.

Что же определяет стандарт ECMA-262, если в нем не фигурируют веб-браузеры? На базовом уровне он определяет следующие части языка:

- синтаксис;
- типы;
- инструкции;
- ключевые слова;
- зарезервированные слова;
- операторы;
- глобальные объекты.

ECMAScript — это просто описание языка, в котором реализованы все аспекты спецификации, а JavaScript — это реализация ECMAScript, но и Adobe ActionScript — тоже реализация ECMAScript.

Редакции ECMAScript

Версии ECMAScript называют *редакциями* (в соответствии с «номером» ECMA-262, где написана конкретная реализация). Последняя (седьмая) редакция ECMA-262 вышла в 2016 г. Первая редакция ECMA-262 была почти такой же, как и JavaScript 1.1 от Netscape, но из нее были удалены все ссылки на код, специфичный для браузеров. Кроме того, в нее были внесены небольшие изменения: ECMA-262 должна была поддерживать стандарт Юникод (для использования других языков) и независимость объектов от платформы (в Netscape JavaScript 1.1 встречались реализации объектов, например `Date`, зависящие от платформы). Таким образом, реализации JavaScript 1.1 и 1.2 не соответствовали первой редакции ECMA-262.

Вторая редакция ECMA-262 появилась из бюрократических соображений. Стандарт был обновлен для согласования с ISO/IEC-16262 и не содержал никаких изменений. В реализациях ECMAScript вторая редакция обычно не используется для оценки соответствия стандарту.

Третья редакция ECMA-262 стала первым реальным усовершенствованием стандарта. В ней были обновлены спецификации обработки строк, определения ошибок и вывода чисел, добавлены регулярные выражения, некоторые управляющие инструкции и обработка исключений с помощью блоков `try-catch`. Также в нее были внесены небольшие изменения, позволяющие подготовить стандарт к интернационализации. Именно с третьей редакции многие начали воспринимать ECMAScript как настоящий язык программирования.

В четвертой редакции ECMA-262 язык был полностью переработан. Привлеченные популярностью JavaScript в интернете, разработчики начали адаптировать ECMAScript к растущим требованиям пользователей со всех уголков мира. В связи с этим снова был созван комитет TC39, чтобы определиться с будущим языка. Итоговая спецификация описывала практически новый язык, созданный на базе третьей редакции. Четвертая редакция включала строго типизированные переменные, новые инструкции и структуры данных, полноценные классы, классическое наследование и новые способы взаимодействия с данными.

В качестве альтернативного предложения подкомитет TC39 разработал спецификацию ECMAScript 3.1, которая не так сильно отличалась от третьей редакции. Она определяла дополнения к ECMAScript, которые могли быть реализованы поверх существующих модулей JavaScript. В итоге подкомитет ES3.1 склонил на свою сторону участников TC39, и работа над четвертой редакцией ECMA-262 была приостановлена.

Спецификация ECMAScript 3.1 стала пятой редакцией ECMA-262 и была официально опубликована 3 декабря 2009 г. Она проясняет неоднозначные места третьей редакции и вводит новую функциональность, в том числе встроенный объект `JSON` для синтаксического анализа и сериализации данных в формате JSON, методы наследования и расширенного определения свойств и новый строгий режим, немного расширяющий возможности интерпретации и выполнения кода модулями ECMAScript. Пятое издание было пересмотрено в июне 2011 г. только для исправления в спецификации, никакие новые функции языка или библиотеки не были введены.

Шестое издание ECMA-262, неофициально называемое ES6, ES2015 или ES Harmony, было опубликовано в июне 2015 г. и является, пожалуй, самым важным сборником улучшений спецификации с момента ее создания. ES6 добавляет формальную поддержку классов, модулей, итераторов, генераторов, стрелочных функций, промисов, отражения, прокси и множества новых типов данных. Седьмое издание ECMA-262, получившее название ES7 или ES2016, было опубликовано в июне 2016 г. Эта версия включала в себя только несколько синтаксических дополнений, таких как *`Array.prototype.include`* и оператор возведения в степень.

Восьмая редакция ECMA-262, получившая название ES8 или ES2017, была завершена в январе 2017 г. Эта редакция добавила асинхронные итерации, операторы остатка и расширения, набор новых функций регулярных выражений, обработчик перехвата *`Promise finally()`* и изменения шаблонных строк.

Девятая редакция ECMA-262 все еще находится в стадии доработки, но она уже имеет большое количество функций на этапе 3. Ее наиболее значительным дополнением, вероятно, будет динамический импорт модулей ES6.

Что означает «соответствие спецификации ECMAScript»?

ECMA-262 проверяет соответствие спецификации ECMAScript. Чтобы считаться ECMAScript-реализацией, язык должен:

- поддерживать все «типы, значения, объекты, свойства, функции, а также синтаксис и семантику программ» согласно их описанию в ECMA-262;
- поддерживать стандарт символов Юникода.

Кроме того, реализация, соответствующая требованиям, может:

- содержать «дополнительные типы, значения, объекты, свойства и функции», не указанные в ECMA-262 (дополнительные элементы описаны преимущественно как новые объекты или новые свойства объектов, которых нет в спецификации);
- поддерживать «синтаксис программ и регулярных выражений», не определенный в ECMA-262 (то есть встроенные средства поддержки регулярных выражений можно изменять и расширять).

Данные критерии позволяют разработчикам создавать новые языки на основе спецификации ECMAScript, чем частично и объясняется ее популярность.

Поддержка ECMAScript в веб-браузерах

Браузер Netscape Navigator 3 с поддержкой JavaScript 1.1 был выпущен в 1996 г. Эта же спецификация JavaScript 1.1 затем была отправлена в Ecma как предложение нового стандарта ECMA-262. Наблюдая за взрывным ростом популярности JavaScript, в Netscape с радостью приступили к разработке версии 1.2. Но была проблема: в Ecma еще не приняли предложение Netscape.

Вскоре после выпуска Netscape Navigator 3 корпорация Microsoft представила Internet Explorer 3. Эта версия IE включала язык JScript 1.0, который должен был стать аналогом JavaScript 1.1, однако из-за недокументированных и неточно воспроизведенных функций оказался гораздо менее популярным.

Браузер Netscape Navigator 4 с JavaScript 1.2 вышел в 1997 г., опередив принятую в том же году первую редакцию ECMA-262. В результате оказалось, что JavaScript 1.2 не соответствует первой редакции ECMAScript, которая должна была базироваться на JavaScript 1.1.

Следующим обновлением JScript стал JScript 3.0, входящий в состав Internet Explorer 4 (версия 2.0 была в составе Microsoft Internet Information Server 3.0, но никогда не включалась в браузеры). Microsoft выпустила пресс-релиз, расхваливающий JScript 3.0 как первый по-настоящему совместимый с Ecma язык сценариев. Окончательная версия ECMA-262 все еще не была принята, так что JScript 3.0 постигла та же судьба, что и JavaScript 1.2, — оказалось, что он не соответствует ECMAScript.

Netscape Navigator 4.06 вышел с обновленной версией JavaScript 1.3, которая была полностью совместимой с первой редакцией ECMA-262. Netscape добавила поддержку стандарта Юникод и сделала все объекты платформенно-независимыми, сохранив при этом возможности, представленные в JavaScript 1.2.

Когда Netscape открыла исходный код браузера в рамках проекта Mozilla, предполагалось, что JavaScript 1.4 войдет в Netscape Navigator 5, однако радикальное решение полностью переработать код браузера поставило крест на этих ожиданиях. JavaScript 1.4 был выпущен только как серверный язык для Netscape Enterprise Server и никогда не использовался в браузерах.

К 2008 г. все пять основных веб-браузеров (Internet Explorer, Firefox, Safari, Chrome и Opera) соответствовали третьей редакции ECMA-262. Internet Explorer 8 стал первым браузером, в котором была начата реализация пятой редакции ECMA-262, а в Internet Explorer 9 она была реализована полностью. Вскоре к IE присоединился Firefox 4. Сведения о поддержке ECMAScript в наиболее популярных веб-браузерах представлены в следующей таблице.

БРАУЗЕР	СООТВЕТСТВИЕ СПЕЦИФИКАЦИИ ECMASCRIPT
Netscape Navigator 2	–
Netscape Navigator 3	–
Netscape Navigator 4–4.05	–
Netscape Navigator 4.06–4.79	Редакция 1
Netscape 6+ (Mozilla 0.6.0+)	Редакция 3
Internet Explorer 3	–
Internet Explorer 4	–
Internet Explorer 5	Редакция 1
Internet Explorer 5.5–8	Редакция 3
Internet Explorer 9	Редакция 5*
Internet Explorer 10-11	Редакция 5
Edge 12+	Редакция 6
Opera 6–7.1	Редакция 2
Opera 7.2+	Редакция 3
Opera 15–28	Редакция 5
Opera 29–35	Редакция 6*
Opera 36+	Редакция 6
Safari 1–2.0.x	Редакция 3*
Safari 3.1–5.1	Редакция 5*
Safari 6–8	Редакция 5
Safari 9+	Редакция 6
iOS Safari 3.2–5.1	Редакция 5*

БРАУЗЕР	СООТВЕТСТВИЕ СПЕЦИФИКАЦИИ ECMASCRIPT
iOS Safari 6–8.4	Редакция 5
iOS Safari 9.2+	Редакция 6
Chrome 1–3	Редакция 3
Chrome 4–22	Редакция 5*
Chrome 23+	Редакция 5
Chrome 42–48	Редакция 6*
Chrome 49+	Редакция 6
Firefox 1–2	Редакция 3
Firefox 3.0.x–20	Редакция 5*
Firefox 21–44	Редакция 5
Firefox 45+	Редакция 6

* Неполная реализация

Объектная модель документа

Объектная модель документа (DOM) — это прикладной программный интерфейс (Application Programming Interface, API) для XML, применение которого было расширено на HTML. В DOM вся страница представляется как иерархия узлов. Каждый элемент HTML- или XML-страницы является узлом определенного типа, содержащим те или иные данные. Рассмотрим следующую HTML-страницу:

```
<html>p
    <head>
        <title>Sample Page</title>
    </head>
    <body>
        <p>Hello World!</p>
    </body>
</html>
```

Этот код можно изобразить с помощью DOM как иерархию узлов (рис. 1.2).

Представляя документ в виде дерева, DOM предоставляет разработчикам беспрецедентный контроль над его контентом и структурой. Используя DOM API, можно с легкостью удалять узлы, добавлять, заменять и изменять их.

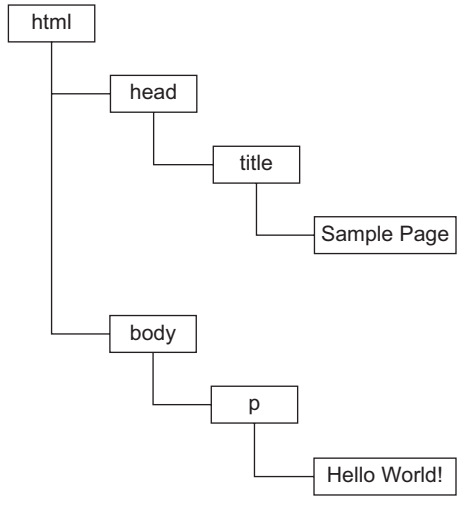

Рис. 1.2

Почему необходима модель DOM

Благодаря реализации динамического HTML (Dynamic HTML, DHTML), в Internet Explorer 4 и Netscape Navigator 4 разработчики впервые смогли изменять вид и контент веб-страниц без их перезагрузки. Это стало важным этапом развития веб-технологий, но возникла серьезная проблема. Netscape и Microsoft избрали разные пути развития DHTML, что завершило период, когда можно было писать HTML-страницы, не задумываясь о веб-браузере.

Стало ясно, что для сохранения кроссплатформенности нужно что-то делать, ведь из-за потенциальных разногласий Netscape и Microsoft Сеть могла разделиться на две части, каждая из которых была бы доступна только пользователям конкретного браузера. Комитет по стандартизации веб-коммуникаций консорциума World Wide Web (W3C) приступил к работе над DOM.

Уровни DOM

В октябре 1998 г. спецификация DOM Level 1 получила статус рекомендации W3C. Она состояла из двух модулей: DOM Core (ядро DOM), определяющий способ представления структуры XML-документа и обеспечивающий удобный доступ к любым частям документа и выполнения операций над ними, и DOM HTML — расширение ядра DOM, оределяющее объекты и методы, специфичные для HTML.

> **ПРИМЕЧАНИЕ** Имейте в виду, что модель DOM не является специфичной для JavaScript и реализована во многих других языках. Однако для веб-браузеров DOM реализована с использованием ECMAScript и теперь является значимой частью JavaScript.

Спецификация DOM Level 1 была предназначена для представления структуры документа, а DOM Level 2 охватывала гораздо больше областей. В оригинальную модель DOM была добавлена поддержка мыши и событий пользовательского интерфейса (давно поддерживавшихся в DHTML), диапазонов, способов обхода DOM-элементов, а также поддержка каскадных таблиц стилей (Cascading Style Sheets, CSS) с помощью объектных интерфейсов. Ядро DOM, представленное в DOM Level 1, также было расширено поддержкой пространств имен XML. Для работы с новыми интерфейсами в DOM Level 2 были представлены новые модули:

- **DOM views** — интерфейсы для отслеживания различных представлений документа (например, документ до и после применения стилей CSS);
- **DOM events** — интерфейсы для событий и обработки событий;
- **DOM style** — интерфейсы для работы со стилизацией элементов с помощью CSS;
- **DOM traversal and range** — интерфейсы для обхода элементов дерева документа и выполнения операций над ними.

DOM Level 3 дополнила DOM унифицированными методами загрузки и сохранения документов (содержатся в новом модуле DOM Load and Save) и методами проверки документа (DOM Validation). В ядре DOM была реализована поддержка всей спецификации XML 1.0, включая XML Infoset, XPath и XML Base.

В настоящее время W3C поддерживает DOM не как набор уровней, а скорее как DOM Living Standard, снимки которого называются DOM4. Среди его введений — добавление наблюдателей за изменениями вместо событий изменений.

ПРИМЕЧАНИЕ Изучая DOM, вы можете встретить упоминания DOM Level 0. Имейте в виду, что стандарта DOM Level 0 не существует, это просто стартовая точка истории DOM. За DOM Level 0 принимают оригинальный DHTML, поддержка которого была реализована в Internet Explorer 4.0 и Netscape Navigator 4.0.

Другие DOM

Кроме интерфейсов DOM Core и DOM HTML, отдельные стандарты DOM опубликованы для нескольких других языков. Так, следующие языки основаны на XML и для каждого из них DOM добавляет уникальные методы и интерфейсы:

- Scalable Vector Graphics (SVG) 1.0;
- Mathematical Markup Language (MathML) 1.0;
- Synchronized Multimedia Integration Language (SMIL).

Некоторые языки, такие как XML User Interface Language (XUL) от Mozilla, содержат собственные DOM-реализации, но только языки из приведенного списка являются стандартными рекомендациями W3C.

Поддержка DOM в веб-браузерах

Спустя некоторое время стандарт DOM начали реализовывать в браузерах. В Internet Explorer начало было положено в версии 5, но серьезной поддержки DOM в IE не было до версии 5.5, в которой была реализована основная часть DOM Level 1. В Internet Explorer 6 и 7 новые функции DOM не добавлялись, а в версии 8 были исправлены некоторые ошибки.

Netscape Navigator не поддерживал DOM до версии Netscape 6 (Mozilla 0.6.0), а после Netscape 7 разработчики из Mozilla сосредоточились на браузере Firefox. Firefox 3+ полностью поддерживает DOM Level 1, почти полностью Level 2 и частично Level 3 (в Mozilla поставлена цель создать браузер, полностью совместимый со стандартами, и усилия себя оправдывают).

Поддержка DOM стала одним из приоритетов для производителей браузеров, которые с каждым выпуском делают ее более полной. В следующей таблице представлены сведения о поддержке DOM в популярных браузерах.

БРАУЗЕР	СООТВЕТСТВИЕ СПЕЦИФИКАЦИИ DOM
Netscape Navigator 1.–4.x	–
Netscape 6+ (Mozilla 0.6.0+)	Level 1, 2 (почти полностью), 3 (частично)
Internet Explorer 2–4.x	–
Internet Explorer 5	Level 1 (минимально)
Internet Explorer 5.5–8	Level 1 (почти полностью)
Internet Explorer 9+	Level 1, 2, 3
Edge	Level 1, 2, 3
Opera 1–6	–
Opera 7–8.x	Level 1 (почти полностью), 2 (частично)
Opera 9–9.9	Level 1, 2 (почти полностью), 3 (частично)
Opera 10+	Level 1, 2, 3 (частично)
Safari 1.0.x	Level 1
Safari 2+	Level 1, 2 (частично), 3 (частично)
iOS Safari 3.2+	Level 1, 2 (частично), 3 (частично)
Chrome 1+	Level 1, 2 (частично), 3 (частично)
Firefox 1+	Level 1, 2 (почти полностью), 3 (частично)

ПРИМЕЧАНИЕ Содержимое данной таблицы совместимости постоянно меняется и должно использоваться только в качестве исторической справки.

Объектная модель браузера

В Internet Explorer 3 и Netscape Navigator 3 была представлена *объектная модель браузера* (BOM), которая обеспечивает доступ к окну браузера и позволяет манипулировать его элементами. Используя BOM, можно взаимодействовать с браузером вне контекста отображаемой страницы. До недавних пор BOM была единственной частью реализации JavaScript, не имеющей стандарта, из-за чего при работе с ней часто возникали проблемы. Формализация многих элементов BOM в HTML5 изменила ситуацию к лучшему, прояснив многие неясные аспекты модели.

BOM регламентирует работу с окном и фреймами браузера, но любое специфичное для браузера JavaScript-расширение тоже обычно считается частью BOM. Вот некоторые такие расширения:

- функция отображения всплывающих окон в браузере;
- возможность перемещать, закрывать и изменять размеры окна браузера;
- объект `navigator`, предоставляющий подробные сведения о браузере;

- объект `location`, предоставляющий подробные сведения о странице, загруженной в браузере;
- объект `screen`, предоставляющий подробные сведения о разрешении экрана;
- поддержка cookie-файлов;
- объект `performance`, который предоставляет подробную информацию о потреблении памяти браузером, поведении навигации и статистике синхронизации;
- пользовательские объекты, включая `XMLHttpRequest`, а также `ActiveXObject` в Internet Explorer.

Стандартов BOM долго не было, поэтому в каждом браузере она реализована по-своему. *Де-факто* некоторые стандарты существуют, например поддержка объектов `window` и `navigator`, но каждый браузер определяет для этих и других объектов свои методы и свойства. Ожидается, что благодаря HTML5 реализации BOM будут развиваться более согласованно. Подробнее тема BOM обсуждается в главе 12 «Объектная модель браузера».

ВЕРСИИ JAVASCRIPT

Как преемник Netscape, фонд Mozilla является единственным производителем браузеров, который продолжает оригинальную нумерацию версий JavaScript. Когда был создан проект Mozilla с открытым исходным кодом, новейшей версией JavaScript в браузерах была версия 1.3 (как уже отмечалось, версия 1.4 использовалась только на сервере). По мере добавления в язык новых возможностей, ключевых слов и элементов синтаксиса номера версий увеличивались. В приведенной ниже таблице показано, как изменялись номера версий JavaScript в браузерах Netscape/Mozilla.

БРАУЗЕР	ВЕРСИЯ JAVASCRIPT
Netscape Navigator 2	1.0
Netscape Navigator 3	1.1
Netscape Navigator 4	1.2
Netscape Navigator 4.06	1.3
Netscape 6+ (Mozilla 0.6.0+)	1.5
Firefox 1	1.5
Firefox 1.5	1.6
Firefox 2	1.7
Firefox 3	1.8
Firefox 3.5	1.8.1
Firefox 3.6	1.8.2
Firefox 4	1.8.5

Предполагалось, что в Firefox 4 будет использоваться JavaScript 2.0, и каждое увеличение номера версии указывает, насколько близка реализация JavaScript к спецификации 2.0. Однако эволюция JavaScript пошла по иному пути, нарушив первоначальный план. В настоящее время у Mozilla нет планов выпуска конкретного продукта с JavaScript 2.0, и этот стиль управления версиями был упразднен после выпуска Firefox 4.

ПРИМЕЧАНИЕ Важно отметить, что этой схеме нумерации соответствовали только браузеры Netscape/Mozilla. Например, в Internet Explorer для нумерации версий JScript применяется совершенно другая схема. Более того, в большинстве браузеров поддержку JavaScript оценивают по соответствию спецификации ECMAScript и поддержке DOM.

ИТОГИ

JavaScript — это язык сценариев, разработанный для взаимодействия с веб-страницами и состоящий из трех основных компонентов:

- язык ECMAScript, определенный в стандарте ECMA-262, обеспечивает базовую JavaScript-функциональность;
- объектная модель документа предоставляет методы и интерфейсы для работы с контентом веб-страницы;
- объектная модель браузера предоставляет методы и интерфейсы для взаимодействия с браузером.

Пять основных веб-браузеров (Internet Explorer, Firefox, Chrome, Safari и Opera) различаются по поддержке компонентов JavaScript. Все производители браузеров в целом адекватно реализовали спецификацию ECMAScript 5 и постепенно расширяют поддержку ECMAScript 6 и 7. Степень реализации DOM варьируется в широких пределах, но соответствие Level 3 становится все более нормативным. Поддержка модели BOM, формализованной в HTML5, также зависит от браузера, хотя некоторые ее элементы универсальны.

2

JavaScript в HTML

- Использование элемента `<script>`
- Сравнение встроенных и внешних сценариев
- Режимы документа в контексте JavaScript
- Показ веб-страниц без JavaScript

С появлением JavaScript немедленно возник вопрос его интеграции с HTML, главным языком веб-страниц. Еще при разработке JavaScript специалисты Netscape думали над тем, как использовать JavaScript в HTML-страницах так, чтобы не нарушить их визуализацию в других браузерах. После многочисленных проб, ошибок и споров удалось прийти к некоторым решениям по универсальной поддержке сценариев в Сети. Многое из того, что было изобретено на заре развития веб-технологий, было формализовано в спецификации HTML и используется по сей день.

ЭЛЕМЕНТ <SCRIPT>

Для вставки JS-кода в HTML-страницу обычно используют элемент `<script>`, который впервые появился в браузере Netscape Navigator 2, а позднее вошел в формальную спецификацию HTML. Он имеет шесть атрибутов.

- `async` (необязательный). Указывает, что нужно немедленно начать загрузку сценария с сервера и сразу же перейти к выполнению других действий на странице, таких как загрузка ресурсов или других сценариев. Действителен только для внешних файлов сценариев.
- `charset` (необязательный). Определяет кодировку сценария, указанного с помощью атрибута `src`. Этот атрибут используется редко, и большинство браузеров его игнорирует.

- `crossorigin` (необязательный). Настраивает параметры CORS для соответствующего запроса; по умолчанию CORS вообще не используется. `crossorigin="anonymous"` настроит запрос для файла с отсутствием установленного флага учетных данных. `crossorigin="use-credentials"` установит флаг учетных данных, это означает, что исходящий запрос будет включать учетные данные.
- `defer` (необязательный). Указывает, что выполнение сценария можно безопасно отложить, пока не будут полностью закончены синтаксический анализ и визуализация контента документа. Действителен только для внешних сценариев. В Internet Explorer 7 и более ранних версиях этот атрибут можно использовать во встроенных сценариях.
- `integrity` (необязательный). Позволяет проверить целостность подресурса (SRI) путем проверки извлеченного ресурса по предоставленной криптографической подписи. Если подпись полученного ресурса не совпадает с подписью, указанной в этом атрибуте, страница выдаст ошибку и скрипт не будет выполнен. Это позволяет гарантировать, что сеть доставки контента (CDN) не обслуживает вредоносные данные.
- `language` (устарел). Определял язык сценариев, используемый в блоке кода (например, `"JavaScript"`, `"JavaScript1.2"` или `"VBScript"`). Большинство браузеров игнорируют этот атрибут, поэтому использовать его не следует.
- `src` (необязательный). Указывает внешний файл с кодом, который нужно выполнить.
- `type` (необязательный). Заменяет атрибут `language`. Указывает тип контента (MIME-тип) языка сценариев, который используется в блоке кода. Традиционно этот атрибут имел значение `"text/javascript"`, но оба значения, и `"text/javascript"`, и `"text/ecmascript"`, устарели. JS-файлы обычно возвращаются сервером с MIME-типом `"application/x-javascript"`, хотя присвоение этого значения атрибуту `type` может привести к игнорированию сценария. В браузерах, отличных от Internet Explorer, также поддерживаются `"application/javascript"` и `"application/ecmascript"`. По традиции и ради обеспечения совместимости этому атрибуту обычно присваивают значение `"text/javascript"`. Если значение атрибута — `module`, код обрабатывается как модуль ES6, и только тогда он может использовать ключевые слова `import` и `export`.

Есть два способа применения элемента `<script>`: можно внедрить JS-код непосредственно в страницу или включить в нее сценарий из внешнего файла.

Чтобы встроить JS-код непосредственно в страницу, поместите его прямо в элемент `<script>`:

```
<script>
    function sayHi() {
        console.log("Hi!");
    }
</script>
```

JS-код в элементе `<script>` обрабатывается сверху вниз. В приведенном примере определение функции интерпретируется и сохраняется в среде интерпретатора.

Остальной контент страницы не загружается и не отображается, пока не будет выполнен весь код в элементе `<script>`.

При написании встроенного JS-кода помните, что использовать строку `"</script>"` в своем коде нельзя. Например, при загрузке следующего кода в браузере возникнет ошибка:

```
<script>
    function sayScript() {
        console.log("</script>");
    }
</script>
```

При синтаксическом анализе встроенного в страницу сценария браузер интерпретирует строку `"</script>"` как закрывающий тег `</script>`. Чтобы решить эту проблему, экранируйте знак /, как показано в этом фрагменте:

```
<script>
    function sayScript() {
        console.log("<\/script>");
    }
</script>
```

Это изменение устраняет ошибку, делая код понятным для браузеров.

Чтобы включить в страницу JS-код из внешнего файла, нужно использовать атрибут `src`. Его значением должен быть URL-адрес файла со сценарием, например:

```
<script src="example.js"></script>
```

В этом примере в страницу загружается внешний файл с именем `example.js`. Сам файл должен содержать только JS-код, который иначе располагался бы между тегами `<script>` и `</script>`. Как и в случае встроенного JS-кода, на время интерпретации внешнего файла обработка страницы приостанавливается (также требуется некоторое время на загрузку файла). В XHTML-документах закрывающий тег можно опускать, например:

```
<script src="example.js" />
```

Не используйте такой синтаксис в HTML-документах, потому что он нарушает правила HTML и неправильно обрабатывается некоторыми браузерами, в частности Internet Explorer.

Элемент `<script>` с атрибутом `src` не может содержать дополнительный JS-код между тегами `<script>` и `</script>`, в противном случае, хотя сценарий загружается и выполняется, встроенный код игнорируется.

Одной из наиболее мощных и противоречивых особенностей элемента `<script>` является возможность включать JS-файлы из внешних доменов. По аналогии с элементом `<img>` атрибуту `src` элемента `<script>` можно назначить полный URL-адрес, не относящийся к домену текущей HTML-страницы, например:

```
<script src="http://www.somewhere.com/afile.js"></script>
```

ПРИМЕЧАНИЕ Внешним JS-файлам обычно назначают расширение .js, но это не обязательно, поскольку браузеры не проверяют расширения включаемых файлов. Это позволяет динамически генерировать JS-код с помощью языка сценариев на стороне сервера или внутрибраузерного переноса в JavaScript из языка расширения JavaScript, такого как TypeScript или React's JSX. Но помните, что серверы часто используют расширение файла для определения правильного MIME-типа, назначаемого ответу. Если вы не применяете расширение .js, убедитесь, что ваш сервер возвращает правильный MIME-тип.

Когда браузер переходит к разрешению этого ресурса, он отправляет `GET`-запрос по пути, указанному в атрибуте `src`, для получения ресурса — предположительно, файла JavaScript. Этот первоначальный запрос не подпадает под ограничения браузера для разных источников, но это не касается любого возвращенного и выполняемого кода JavaScript. Конечно, этот запрос все еще подчиняется протоколу HTTP/HTTPS родительской страницы.

Код из внешнего домена загружается и интерпретируется как часть страницы, в которую он загружается. Это позволяет при необходимости получать JS-сценарии из других доменов, но будьте осторожны, загружая сценарий с сервера, который вы не контролируете: злоумышленник может в любой момент заменить загружаемый файл. Включайте JS-файлы из других доменов, только если эти домены принадлежат вам или если вы доверяете их владельцам. Атрибут `integrity` тега `<script>` предоставляет инструмент для защиты от этого, однако он имеет ограниченную поддержку браузерами.

Независимо от того, как код включается в HTML-страницу, элементы `<script>` интерпретируются в том порядке, в котором они расположены, при условии, что у них нет атрибутов `defer` и `async`. Код первого элемента `<script>` должен быть полностью интерпретирован, чтобы можно было приступить ко второму элементу `<script>`, второй элемент должен быть полностью обработан перед третьим и т. д.

Расположение тегов

Все элементы `<script>` в коде страницы традиционно размещались внутри элемента `<head>`, как в следующем примере HTML-страницы:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Example HTML Page</title>
    <script src="example1.js"></script>
    <script src="example2.js"></script>
  </head>
  <body>
    <!-- содержимое -->
  </body>
</html>
```

Такой формат использовался во многом для того, чтобы ссылки на внешние CSS- и JS-файлы находились в одном месте. Однако подобный формат имеет существенный недостаток: загрузка, синтаксический анализ и интерпретация всех сценариев должны быть завершены до начала визуализации страницы (визуализация начинается, когда браузер получает открывающий тег `<body>`). Если на странице используется много JS-кода, это может приводить к длительным задержкам, в течение которых пользователь видит пустое окно браузера. Поэтому в современных веб-приложениях все ссылки на JS-сценарии обычно указываются в элементе `<body>` после контента страницы:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Example HTML Page</title>
  </head>
  <body>
    <!-- содержимое -->
    <script src="example1.js"></script>
    <script src="example2.js"></script>
  </body>
</html>
```

При таком подходе страница полностью визуализируется в браузере до обработки JS-кода. Для пользователей это предпочтительнее, потому что контент отображается раньше.

Отложенные сценарии

В HTML 4.01 для элемента `<script>` определен атрибут `defer`, который указывает, что сценарий не будет изменять структуру страницы, а потому его можно безопасно выполнить после синтаксического анализа всей страницы. Атрибут `defer` сигнализирует браузеру, что загрузку сценария можно начать немедленно, но его выполнение следует отложить:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Example HTML Page</title>
    <script defer src="example1.js"></script>
    <script defer src="example2.js"></script>
  </head>
  <body>
    <!-- содержимое -->
  </body>
</html>
```

Несмотря на то что в этом примере элементы `<script>` включены в элемент `<head>` документа, выполнение сценариев не начнется, пока браузер не получит закрывающий тег `</html>`. В спецификации HTML5 указано, что сценарии интерпретируются по порядку, так что первый отложенный сценарий выполняется перед вторым, а оба они будут выполнены до события `DOMContentLoaded`

(см. главу 17 «События»). Однако в реальности требования спецификации не всегда соблюдаются, поэтому по возможности лучше включать в страницу только один отложенный сценарий.

Как уже отмечалось, атрибут `defer` поддерживается только для внешних файлов сценариев. Это уточнение было добавлено в HTML5, так что браузеры, поддерживающие HTML5 (включая Internet Explorer 8 и более поздних версий), игнорируют атрибут `defer`, если он задан для встроенного сценария. Браузеры Internet Explorer 4–7 работают по-старому.

Поддержка атрибута `defer` появилась в браузерах Internet Explorer 4, Firefox 3.5, Safari 5 и Chrome 7. Более старые браузеры просто игнорируют этот атрибут и обрабатывают сценарии с ним обычным образом, поэтому отложенные сценарии лучше располагать в конце страницы.

> **ПРИМЕЧАНИЕ** В XHTML-документах указывайте атрибут defer как defer="defer".

Асинхронные сценарии

В HTML5 для элемента `<script>` представлен атрибут `async`, который похож на атрибут `defer` в том смысле, что он тоже изменяет способ обработки сценария. Он также применяется только к внешним сценариям и указывает браузеру немедленно начать загрузку файла, но для сценариев с атрибутом `async` не гарантируется выполнение в порядке их добавления, например:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Example HTML Page</title>
    <script async src="example1.js"></script>
    <script async src="example2.js"></script>
  </head>
  <body>
    <!-- содержимое -->
  </body>
</html>
```

Здесь второй сценарий может быть выполнен перед первым, поэтому важно, чтобы между ними не было зависимостей. Атрибут `async` используется, если нужно разрешить браузеру продолжить загрузку страницы, не дожидаясь завершения загрузки и выполнения сценария. По этой причине рекомендуется не изменять в асинхронных сценариях DOM-элементы.

Асинхронные сценарии гарантированно выполняются до события `load` страницы, но могут выполняться до или после события `DOMContentLoaded` (см. главу 17 «События»). Они поддерживаются в браузерах Firefox 3.6, Safari 5 и Chrome 7. При использовании асинхронных сценариев страница также неявно предполагает, что вы не намерены использовать `document.write`, тем более что наилучшие методики веб-разработки требуют в любом случае его не использовать.

ПРИМЕЧАНИЕ В XHTML-документах указывайте атрибут async как async="async".

Динамическая загрузка сценариев

Вы не ограничены использованием статических тегов `<script>` для извлечения ресурсов. Поскольку JavaScript может использовать DOM API, можно добавлять элементы сценария, которые, в свою очередь, загрузят указанные ресурсы. Для этого нужно создать элементы сценария и прикрепить их к DOM:

```
let script = document.createElement('script');
script.src = 'gibberish.js';
document.head.appendChild(script);
```

Конечно, этот запрос не будет сгенерирован до тех пор, пока HTMLElement не будет присоединен к DOM, и, следовательно, до тех пор, пока сам скрипт не будет запущен. По умолчанию сценарии, созданные таким образом, помечаются как `async`. Однако это может быть проблематично, поскольку все браузеры поддерживают `createElement`, но не все поддерживают запросы `async` сценариев. Поэтому, чтобы унифицировать поведение динамической загрузки скрипта, можно явно пометить тег как синхронный:

```
let script = document.createElement('script');
script.src = 'gibberish.js';
script.async = false;
document.head.appendChild(script);
```

Полученные таким образом ресурсы будут скрыты от предварительных загрузчиков браузера. Это серьезно понизит их приоритет в очереди выборки ресурса. В зависимости от того, как работает приложение и как оно используется, такой подход может серьезно снизить производительность. Чтобы сообщить предварительным загрузчикам о существовании этих динамически запрашиваемых файлов, их нужно явно объявить в заголовке документа:

```
<link rel="subresource" href="gibberish.js">
```

Изменения в XHTML

Расширяемый язык гипертекстовой разметки (Extensible HyperText Markup Language, XHTML) переопределяет HTML как разновидность XML. В отличие от HTML, где атрибут `type` не нужен при использовании JavaScript, в XHTML элемент `<script>` требует указания атрибута `type` как `text/javascript`.

Правила написания XHTML-кода строже в сравнении с HTML, это касается и элементов `<script>` с внедренным JS-кодом. Например, следующий код допустим в HTML, но недопустим в XHTML:

```
<script type="text/javascript">
    function compare(a, b) {
```

```
        if (a < b) {
            console.log("A is less than B");
        } else if (a > b) {
            console.log("A is greater than B");
        } else {
            console.log("A is equal to B");
        }
    }
</script>
```

В HTML есть специальные правила синтаксического анализа содержимого элемента `<script>`, которые в XHTML не применяются. Из-за этого символ «меньше» (`<`) в инструкции `a < b` интерпретируется как начало тега, что приводит к синтаксической ошибке (за символом «меньше» не может следовать пробел).

Есть два способа исправить эту синтаксическую ошибку в XHTML-коде. Первый — заменить все экземпляры символа «меньше» (`<`) его HTML-мнемоникой (`<`):

```
<script type="text/javascript">
    function compare(a, b) {
        if (a &lt; b) {
            console.log("A is less than B");
        } else if (a > b) {
            console.log("A is greater than B");
        } else {
            console.log("A is equal to B");
        }
    }
</script>
```

Хотя этот код будет работать на XHTML-странице, читать его сложнее. К счастью, есть другой способ: поместить JS-код в раздел `CDATA`. В XHTML (и XML) разделы `CDATA` применяются для указания областей документа с произвольным текстом, который исключается из синтаксического анализа. Это позволяет использовать любые знаки, включая символ «меньше», и ошибок не возникает:

```
<script type="text/javascript"><![CDATA[
    function compare(a, b) {
        if (a < b) {
            console.log("A is less than B");
        } else if (a > b) {
            console.log("A is greater than B");
        } else {
            console.log("A is equal to B");
        }
    }
]]></script>
```

В веб-браузерах, совместимых с XHTML, это устраняет проблему, однако многие браузеры все еще не поддерживают XHTML и раздел `CDATA`. Для обходного решения этой проблемы нужно добавить перед разметкой `CDATA` JavaScript-комментарий:

```
<script type="text/javascript">
//<![CDATA[
    function compare(a, b) {
        if (a < b) {
            console.log("A is less than B");
        } else if (a > b) {
            console.log("A is greater than B");
        } else {
            console.log("A is equal to B");
        }
    }
//]]>
</script>
```

Данный способ работает во всех современных браузерах. Конечно, это нестандартный прием, но он соответствует требованиям XHTML и не вызывает проблем в браузерах, не поддерживающих XHTML.

> **ПРИМЕЧАНИЕ** Режим XHTML включается, если для страницы задан MIME-тип "application/xhtml+xml". Не все браузеры официально поддерживают такой XHTML-код.

Устаревший синтаксис

С момента появления Netscape 2 в 1995 г. все браузеры стали использовать JavaScript в качестве языка программирования по умолчанию. Атрибут `type` использует строку типа MIME для идентификации содержимого `<script>`, но типы MIME не стандартизированы в разных браузерах. Несмотря на то что браузеры используют JavaScript по умолчанию, в некоторых случаях недопустимое или нераспознанное значение MIME-типа для атрибута `type` приведет к тому, что некоторые браузеры пропустят выполнение связанного кода. Поэтому, если вы не используете XHTML или тег `<script>` не запрашивает или не использует JavaScript, лучше не указывать атрибут `type` вообще.

С появлением элемента `<script>` потребовалось обновить традиционный способ синтаксического анализа HTML-кода. К содержимому элемента нужно было применять специальные правила, из-за чего возникали проблемы в браузерах, не поддерживающих JavaScript, таких как Mosaic. Эти браузеры просто выводили содержимое элементов `<script>` на страницах, что портило все впечатление.

Компании Netscape и Mosaic совместно разработали решение, скрывающее встроенный JS-код от браузеров, которые его не поддерживали. Было предложено заключать код сценариев в HTML-комментарии:

```
<script><!--
    function sayHi() {
        console.log("Hi!");
    }
//--></script>
```

В результате браузеры вроде Mosaic безопасно игнорировали содержимое тегов `<script>`, а браузеры, поддерживающие JavaScript, распознавали этот шаблон, извлекая код из комментариев.

Хотя этот формат все еще распознается и правильно интерпретируется всеми веб-браузерами, он больше не нужен, поэтому использовать его не следует. Кроме того, в режиме XHTML такой код оказывается скрытым в правильных XML-комментариях и поэтому игнорируется.

ВСТРОЕННЫЙ КОД ИЛИ ВНЕШНИЕ ФАЙЛЫ?

Хотя JS-сценарии можно внедрять непосредственно в HTML-страницы, считается, что лучше включать JS-код из внешних файлов. Хотя это не является незыблемым принципом, здесь приведены некоторые аргументы в пользу такого подхода.

- **Удобство сопровождения.** JS-код, разбросанный по многим HTML-страницам, трудно сопровождать. Гораздо проще создать один каталог для всех JS-файлов, чтобы разработчики могли редактировать код независимо от разметки, в которой он используется.
- **Кеширование.** Браузеры кешируют все связанные внешние JS-файлы, и если в двух страницах используется один файл, он загружается только один раз, что ускоряет загрузку страниц.
- **Готовность к будущему.** При включении сценариев из внешних файлов не нужно использовать упомянутые ранее приемы с комментариями и XHTML-кодом. Синтаксис включения в HTML и XHTML внешних файлов одинаков.

При настройке способа запроса внешних файлов следует обратить внимание на их влияние на пропускную способность запроса. При использовании SPDY/HTTP2 расходы на запрос существенно сокращаются, поскольку может быть выгодно доставлять сценарии клиенту в виде легких независимых компонентов JavaScript.

Например, на первой странице может быть указано следующее:

```
<script src="mainA.js"></script>
<script src="component1.js"></script>
<script src="component2.js"></script>
<script src="component3.js"></script>
...
```

На следующей загруженной странице может быть указано это:

```
<script src="mainB.js"></script>
<script src="component3.js"></script>
<script src="component4.js"></script>
<script src="component5.js"></script>
...
```

Если браузер поддерживает SPDY/HTTP2, то по первому запросу он сможет эффективно извлекать несколько файлов из одной и той же конечной точки и добавлять их

в кеш отдельно для каждого файла. С точки зрения браузера поиск этих отдельных ресурсов через SPDY/HTTP2 должен иметь примерно ту же задержку, что и при получении монолитной полезной нагрузки JavaScript.

По запросу второй страницы, поскольку мы разделили приложение на легкие кешируемые файлы, некоторые компоненты, от которых также зависит вторая страница, уже находятся в кеше.

Конечно, это предполагает, что браузер поддерживает SPDY/HTTP2, что является только предполагаемым допущением для современных браузеров. Монолитная полезная нагрузка может подойти, если ваша цель — поддержка старых браузеров.

РЕЖИМЫ ДОКУМЕНТА

В Internet Explorer 5.5 были представлены режимы документа, выбираемые путем переключения типов документа. Первыми двумя режимами были *режим совместимости* (quirks mode), в котором Internet Explorer работал как версия 5 (с несколькими нестандартными функциональными возможностями), и *стандартный режим* (standards mode), в котором Internet Explorer работал согласно стандартам. Хотя основные различия этих режимов связаны с использованием CSS, косвенно они затронули и JavaScript. Эти побочные эффекты обсуждаются во многих местах книги.

Вслед за Internet Explorer режимы документа были реализованы и в других браузерах, при этом появился третий режим, названный *почти стандартным* (almost standards mode). Он во многом напоминает стандартный режим, но менее строг. Основное отличие проявляется в том, как обрабатываются интервалы вокруг изображений, что наиболее заметно, если изображения находятся в таблицах.

Режим совместимости включается во всех браузерах, если в начале документа не указан его тип. Это считается плохой практикой, потому что реализация режима совместимости сильно различается в браузерах и добиться их согласованной работы в этом режиме очень непросто.

Стандартный режим включается для следующих типов документов:

```
<!-- HTML 4.01 Strict -->
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">

<!-- XHTML 1.0 Strict -->
<!DOCTYPE html PUBLIC
"-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<!-- HTML5 -->
<!DOCTYPE html>
```

Почти стандартный режим включается при указании в качестве типов документов `transitional` и `frameset`:

```
<!-- HTML 4.01 Transitional -->
<!DOCTYPE HTML PUBLIC
"-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">

<!-- HTML 4.01 Frameset -->
<!DOCTYPE HTML PUBLIC
"-//W3C//DTD HTML 4.01 Frameset//EN"
"http://www.w3.org/TR/html4/frameset.dtd">

<!-- XHTML 1.0 Transitional -->
<!DOCTYPE html PUBLIC
"-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<!-- XHTML 1.0 Frameset -->
<!DOCTYPE html PUBLIC
"-//W3C//DTD XHTML 1.0 Frameset//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-frameset.dtd">
```

Из-за того что стандартный и почти стандартный режимы так похожи, обычно их не различают. Когда говорят о «стандартном режиме», речь может идти о любом из них, и определение типа документа (которое мы обсудим позже) также не проводит различие между ними. В этой книге под *стандартным режимом* (standards mode) понимается любой режим, кроме режима совместимости.

ЭЛЕМЕНТ <NOSCRIPT>

В ранних браузерах, не поддерживающих JavaScript, очень важно было элегантно переключаться на упрощенные версии страниц. Для этого был создан элемент `<noscript>`, позволяющий предоставлять альтернативный контент таким браузерам. Хотя фактически 100% браузеров теперь поддерживают JavaScript, этот элемент по-прежнему является полезным для браузеров, явно отключающих JavaScript.

Элемент `<noscript>` может содержать любые HTML-элементы (кроме `<script>`), которые могут быть добавлены в элемент `<body>` документа. Любое содержимое элемента `<noscript>` выводится на экран только в двух следующих ситуациях:

- браузер не поддерживает сценарии;
- поддержка сценариев в браузере отключена.

Если одно из этих условий выполнено, содержимое элемента `<noscript>` визуализируется, в противном случае он пропускается.

Рассмотрим простой пример:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Example HTML Page</title>
    <script defer="defer" src="example1.js"></script>
```

```
    <script defer="defer" src="example2.js"></script>
  </head>
  <body>
    <noscript>
      <p> This page requires a JavaScript-enabled browser.</p>
    </noscript>
  </body>
</html>
```

Если поддержка сценариев недоступна, этот код выводит на экран следующее сообщение, извещающее, что для вывода страницы нужен браузер с поддержкой JavaScript:

```
This page requires a JavaScript-enabled browser
```

В браузерах с поддержкой сценариев сообщение не выводится, хотя и является частью страницы.

ИТОГИ

JS-код вставляется в HTML-страницы с помощью элемента `<script>`. Он позволяет встраивать сценарии непосредственно в разметку HTML-страницы или использовать сценарии из внешних файлов.

- Чтобы включить в страницу внешний JS-файл, назначьте атрибуту `src` URL-адрес файла, который может находиться как на сервере со страницей-контейнером, так и в другом домене.
- Все элементы `<script>` интерпретируются в том порядке, в котором они расположены на странице. Если не используются атрибуты `defer` и `async`, браузер должен полностью интерпретировать содержимое одного элемента `<script>`, прежде чем сможет перейти к следующему.
- В случае неотложенных сценариев браузер должен завершить интерпретацию кода внутри элемента `<script>` перед продолжением визуализации остальной части страницы. По этой причине элементы `<script>` обычно располагают ближе к концу страницы: после основного контента и непосредственно перед закрывающим тегом `</body>`.
- С помощью атрибута `defer` можно отложить выполнение сценария до завершения визуализации страницы. Отложенные сценарии всегда выполняются в том порядке, в котором они указаны.
- С помощью атрибута `async` можно указать, что сценарий не должен дожидаться других сценариев и блокировать визуализацию документа. Асинхронные сценарии могут выполняться не в том порядке, в котором они расположены на странице.

Элемент `<noscript>` позволяет задать контент, выводимый на экран только в том случае, если браузер не поддерживает сценарии. Если сценарии в браузере поддерживаются, содержимое элемента `<noscript>` не визуализируется.

3

Основы языка

- Обзор синтаксиса
- Типы данных
- Операторы управления потоком
- Функции

Загрузки с сайта wrox.com для этой главы

Обратите внимание, что все примеры кода для этой главы доступны на веб-сайте книги по ссылке www.wrox.com/go/projavascript4e на вкладке Download Code.

В сердце каждого языка лежит базовое описание принципов его работы. Как правило, в нем приводятся синтаксис, операторы, типы данных и встроенный функционал, на основе которых можно создавать сложные решения. В стандарте ECMA-262 все эти элементы определены для JavaScript в форме псевдоязыка, который называется ECMAScript.

В большинстве веб-браузеров реализована версия ECMAScript из пятой редакции ECMA-262. На очереди шестая редакция, которая на конец 2017 г. почти или полностью реализована во всех основных браузерах. Сведения в этой главе основаны преимущественно на шестой редакции ECMAScript.

СИНТАКСИС

Синтаксис ECMAScript во многом похож на C и другие C-подобные языки, такие как Java и Perl. Если вы знакомы с ними, вам будет легко привыкнуть к более свободному синтаксису ECMAScript.

Чувствительность к регистру

В ECMAScript все элементы, включая имена переменных, функций и операторов, чувствительны к регистру. Например, переменные `test` и `Test` различны, а ключевое слово `typeof` не может быть именем функции, тогда как `typeOf` — нормальное имя.

Идентификаторы

Идентификатор (identifier) — это имя переменной, функции, свойства или аргумента функции. Идентификаторы могут состоять из одного или нескольких знаков, удовлетворяющих двум условиям:

- первый знак должен быть буквой, знаком подчеркивания (`_`) или знаком доллара (`$`);
- все остальные знаки могут быть буквами, знаками подчеркивания, знаками доллара или цифрами.

В идентификаторах можно использовать буквы из расширенного набора ASCII или из Юникода, такие как À и Æ, но это не рекомендуется.

В ECMAScript-идентификаторах применяется верблюжья нотация. Это означает, что первая буква является строчной, а первые буквы всех последующих слов — прописными, например:

```
firstSecond
myCar
doSomethingImportant
```

Хотя это не является требованием, рекомендуется следовать этому правилу, чтобы не отступать от формата встроенных функций и объектов ECMAScript.

> **ПРИМЕЧАНИЕ** Ключевые слова, зарезервированные слова и значения true, false и null не могут быть идентификаторами (см. далее раздел «Ключевые и зарезервированные слова»).

Комментарии

ECMAScript поддерживает однострочные и блочные комментарии в стиле C. Для ввода однострочного комментария используются две косые черты:

```
// однострочный комментарий
```

Блочный комментарий начинается с косой черты и звездочки (`/*`), а заканчивается ими же в обратном порядке (`*/`):

```
/*
 * Это многострочный
 * комментарий
 */
```

Строгий режим

В ECMAScript 5 представлена концепция *строгого режима* (strict mode) — особой модели синтаксического анализа и выполнения JS-кода, в которой исправлены некоторые аспекты работы ECMAScript и генерируются ошибки при небезопасных действиях. Чтобы включить строгий режим для всего сценария, добавьте в начало файла следующую команду:

```
"use strict"
```

Хотя она выглядит как строка, которую забыли присвоить переменной, на самом деле это директива, переводящая JavaScript в строгий режим. Такой синтаксис был выбран специально, чтобы исключить конфликты с ECMAScript 3.

Строгий режим можно включить и для отдельной функции, добавив эту директиву в начало тела функции:

```
function doSomething() {
    "use strict";
    // тело функции
}
```

В строгом режиме выполнение JS-кода заметно меняется, и мы не раз с этим столкнемся. Строгий режим поддерживается во всех современных браузерах.

Инструкции

Инструкции в ECMAScript завершаются точками с запятой, хотя синтаксический анализатор сам способен определить конец инструкции, например:

```
let sum = a + b   // правильно даже без точки с запятой, но не рекомендуется
let diff = a – b; // правильно и рекомендуется
```

Хотя точки с запятой в конце инструкций необязательны, нужно всегда добавлять их. Это предотвращает некоторые ошибки, например незавершенный ввод, и позволяет сжимать ECMAScript-код за счет удаления пустых мест (без точек с запятой это приводит к синтаксическим ошибкам). Кроме того, это препятствует снижению быстродействия, потому что синтаксические анализаторы пытаются исправлять предполагаемые ошибки, добавляя недостающие точки с запятой.

Как и в C, при помощи фигурных скобок ({}) несколько инструкций можно объединить в блок кода:

```
if (test) {
    test = false;
    console.log(test);
}
```

В управляющих инструкциях вроде `if` блоки требуются, только если инструкций несколько, но на практике рекомендуется создавать блок даже для одной инструкции:

```
// допустимо, но чревато ошибками и не рекомендуется
if (test)
    console.log(test);

// предпочтительный способ
if (test) {
    console.log(test);
}
```

Использование блоков кода с управляющими инструкциями проясняет намерения программиста и предотвращает ошибки при изменении кода.

КЛЮЧЕВЫЕ И ЗАРЕЗЕРВИРОВАННЫЕ СЛОВА

Стандарт ECMA-262 определяет набор зарезервированных *ключевых слов* (keywords), служащих для решения специализированных задач, таких как указание начала или конца управляющей инструкции или выполнение специфической операции. Ключевые слова нельзя использовать как идентификаторы или имена свойств. Вот их полный список для шестой редакции ECMA-262:

```
break       do          in          typeof
case        else        instanceof  var
catch       export      new         void
class       extends     return      while
const       finally     super       with
continue    for         switch      yield
debugger    function    this
default     if          throw
delete      import      try
```

Кроме того, ECMA-262 содержит набор *будущих зарезервированных слов* (future reserved words), которые также нельзя использовать как идентификаторы или имена свойств. Хотя эти слова не имеют специфического применения в языке, они зарезервированы на будущее как потенциальные ключевые слова.

Вот полный список будущих зарезервированных слов из шестой редакции ECMA-262:

```
Всегда зарезервированы:

enum

Зарезервированы в строгом режиме:

implements    package     public
interface     protected   static
let           private

Зарезервированы в модульном коде:

await
```

Данные слова также могут не быть идентификаторами, но теперь их разрешено использовать как имена свойств в объектах. В общем, для обеспечения совместимости

с прошлыми и будущими редакциями ECMAScript лучше не использовать ключевые и зарезервированные слова как идентификаторы и имена свойств.

ПЕРЕМЕННЫЕ

ECMAScript-переменные типизированы слабо, то есть могут содержать данные любого типа. Каждая переменная — это просто именованный заполнитель для значения. Для объявления переменной можно использовать три ключевых слова: `var`, которое доступно во всех версиях ECMAScript, `const` и `let`, которые были введены в ECMAScript 6.

Ключевое слово var

Для определения переменной используется оператор `var` (заметьте, что это одно из ключевых слов), после которого указывается имя (идентификатор) переменной, например:

```
var message;
```

Здесь определяется переменная с именем `message`, которая может содержать любое значение (без инициализации она содержит специальное значение `undefined`, описанное в следующем разделе). ECMAScript поддерживает инициализацию переменных, то есть можно одновременно определить переменную и присвоить ей значение, например:

```
var message = "hi";
```

Здесь определяется переменная `message` для хранения строки `"hi"`. Инициализация не превращает переменную в строковую, она просто присваивает ей значение. После инициализации можно не только изменить хранящееся в переменной значение, но и тип этого значения, например:

```
var message = "hi";
message = 100;           // допустимо, но не рекомендуется
```

В этом примере переменная `message` сначала определяется как строковое значение `"hi"`, а затем перезаписывается числовым значением 100. Хотя изменять тип данных, содержащихся в переменной, не рекомендуется, в ECMAScript это возможно.

Область объявления var

Важно отметить, что при определении переменной с помощью оператора `var` она становится локальной в текущей области видимости. Например, если определить переменную с оператором `var` внутри функции, она будет уничтожена при выходе из функции:

```
function test() {
    var message = "hi";    // локальная переменная
```

```
}
test();
console.log(message);      // ошибка!
```

Здесь переменная `message` определяется с помощью оператора `var` в функции `test()`. При создании переменной ей присваивается значение, но сразу же после этого она уничтожается, из-за чего в последней строке возникает ошибка. Однако переменную можно определить глобально, просто опустив оператор `var`:

```
function test() {
    message = "hi";        // глобальная переменная
}
test();
console.log(message);      // "hi"
```

Теперь переменная `message` определена как глобальная. При вызове функции `test()` она инициализируется и становится доступна вне функции.

> **ПРИМЕЧАНИЕ** Определять глобальные переменные, опуская оператор var, не рекомендуется. Код с глобальными переменными, определенными локально, трудно разбирать и сопровождать, потому что непонятно, пропущен оператор var намеренно или случайно. В строгом режиме при попытке присвоить значение необъявленной переменной возникает ошибка ReferenceError.

В одной инструкции можно определить сразу несколько переменных, разделив их (с инициализацией или без нее) запятыми:

```
var message = "hi",
    found = false,
    age = 29;
```

Здесь объявляются и инициализируются три переменные. Поскольку ECMAScript типизирован слабо, в одной инструкции переменные можно инициализировать значениями разных типов. Чтобы облегчить чтение кода, можно разделить строку на несколько и добавить отступы, но это не требуется.

В строгом режиме определить переменную с именем `eval` или `arguments` нельзя. Попытка сделать это приведет к синтаксической ошибке.

Поднятие объявлений var

При использовании `var` возможно следующее, потому что переменные, объявленные с использованием этого ключевого слова, поднимаются в верхнюю часть области видимости функции:

```
function foo() {
    console.log(age);
    var age = 26;
}
foo(); // undefined
```

Ошибки не происходит, потому что среда выполнения ECMAScript технически обрабатывает код следующим образом:

```
function foo() {
    var age;
    console.log(age);
    age = 26;
}
foo(); // undefined
```

Это называется поднятием, когда интерпретатор вытягивает все объявления переменных в верхнюю часть своей области видимости. Это также позволяет использовать избыточные объявления `var` без наказания за это:

```
function foo() {
    var age = 16;
    var age = 26;
    var age = 36;
    console.log(age);
}
foo(); // 36
```

Объявления let

`let` работает почти так же, как `var`, но с некоторыми существенными отличиями. Наиболее примечательным является то, что `let` подчиняется области видимости блока, а `var` — области видимости функции.

```
if (true) {
    var name = 'Matt';
    console.log(name);      // Matt
}
console.log(name);          // Matt

if (true) {
    let age = 26;
    console.log(age);       // 26
}
console.log(age);           // ReferenceError: age is not defined
```

Здесь на переменную `age` нельзя ссылаться вне блока `if`, поскольку ее область видимости не выходит за пределы блока. Область видимости блока является строго подмножеством области действия функции, поэтому любые ограничения области видимости, которые применяются к объявлениям `var`, также будут применяться к объявлениям `let`. Объявление `let` также не допускает никаких избыточных объявлений в пределах блока. Это приведет к ошибке:

```
var name;
var name;

let age;
let age;    // SyntaxError; identifier 'age' has already been declared
```

Конечно, движок JavaScript будет отслеживать идентификаторы, используемые для объявлений переменных, и область видимости блока, в которой они были объявлены, поэтому вложение с использованием идентичных идентификаторов ведет себя так, как и следовало ожидать, без ошибок, поскольку переопределения не происходит:

```
var name = 'Nicholas';
console.log(name);         // 'Nicholas'
if (true) {
    var name = 'Matt';
    console.log(name);     // 'Matt'
}

let age = 30;
console.log(age);          // 30
if (true) {
    let age = 26;
    console.log(age);      // 26
}
```

Ошибки избыточности объявления не являются порядковой функцией и не затрагиваются, если `let` смешивается с `var`. Различные ключевые слова не объявляют разные типы переменных — они просто указывают на то, как переменные существуют внутри соответствующей области видимости.

```
var name;
let name;    // SyntaxError

let age;
var age;    // SyntaxError
```

Временная мертвая зона

Другое важное поведение `let`, отличающее его от `var`, заключается в том, что объявления `let` не могут использоваться способом, который предполагает подъем:

```
// name поднимается
console.log(name);    // undefined
var name = 'Matt';

// age не поднимается
console.log(age);     // ReferenceError: age is not defined
let age = 26;
```

При синтаксическом анализе кода механизмы JavaScript все равно будут знать об объявлениях `let`, которые появляются позже в блоке, но на эти переменные невозможно будет ссылаться каким-либо образом до того, как произойдет фактическое объявление. Сегмент выполнения, который происходит перед объявлением, называется временной мертвой зоной, и любые попытки ссылаться на эти переменные будут вызывать `ReferenceError`.

Глобальные объявления

В отличие от ключевого слова `var`, при объявлении переменных с использованием `let` в глобальном контексте переменные не будут присоединяться к объекту `window`, как это происходит с `var`.

```
var name = 'Matt';
console.log(window.name);    // 'Matt'

let age = 26;
console.log(window.age);     // undefined
```

Тем не менее объявления `let` будут по-прежнему происходить внутри глобальной области видимости блока, которая сохраняется в течение всего срока жизни страницы. Следовательно, нужно убедиться, что страница не пытается дублировать объявления, чтобы избежать появления `SyntaxError`.

Условное объявление

При использовании `var` для объявления переменных, поскольку объявление поднимается, движок JavaScript с радостью объединит избыточные объявления в одно в верхней части области видимости. Поскольку объявления `let` ограничены блоками, невозможно проверить, была ли ранее объявлена переменная `let`, и условно объявить ее, только если это не было уже сделано с ней.

```
<script>
    var name = 'Nicholas';
    let age = 26;
</script>

<script>
    // Предположим, что этот скрипт не уверен в том, что уже было объявлено
    // на странице. Он предполагает, что переменные не были объявлены.

    var name = 'Matt';
    // Здесь нет проблем, так как это будет обрабатываться как одно поднятое
    // объявление. Нет необходимости проверять, была ли переменная объявлена ранее.

    let age = 36;
    // Здесь произойдет ошибка, потому что 'age уже была объявлена.
</script>
```

Использование оператора `try/catch` или оператора `typeof` не решит проблему, поскольку объявление `let` внутри условного блока будет ограничено этим блоком.

```
<script>
    let name = 'Nicholas';
    let age = 36;
</script>

<script>
    // Предположим, что этот скрипт не уверен в том, что уже было объявлено
    // на странице. Он предполагает, что переменные не были объявлены.

    if (typeof name !== 'undefined') {
```

```
        let name;
    }
    // 'name' ограничен областью действия блока if {},
    // так что это присвоение будет действовать как глобальное присвоение
    name = 'Matt';

    try (age) {
        // Если возраст не указан, это приведет к ошибке
    }
    catch(error) {
        let age;
    }
    // 'age' ограничен областью действия блока catch {},
    // так что это присвоение будет действовать как глобальное присвоение
    age = 26;
</script>
```

Из-за этого нельзя полагаться на шаблон условного объявления с этим новым ключевым словом объявления ES6.

> **ПРИМЕЧАНИЕ** Невозможность использования `let` для условного объявления – это хорошо, так как условное объявление считается плохим шаблоном в кодовой базе. Оно усложняет понимание потока программы. Если вы обнаружите, что код приводит к этому паттерну, то очень велики шансы, что существует способ переписать его получше.

Объявление let в циклах for

До появления `let` определение цикла `for` включало в себя использование переменной-итератора, определение которой выходило за пределы тела цикла:

```
for (var i = 0; i < 5; ++i) {
    // тело цикла
}
console.log(i); // 5
```

При переходе на объявления `let` это перестало быть проблемой, поскольку переменная-итератор будет видна только блоку `for`:

```
for (let i = 0; i < 5; ++i) {
    // тело цикла
}
console.log(i); // ReferenceError: i is not defined
```

При использовании `var` часто встречалась проблема, связанная с объявлением и модификацией переменной-итератора:

```
for (var i = 0; i < 5; ++i) {
    setTimeout(() => console.log(i), 0)
}
// console.log, как ожидается, должен выводить 0, 1, 2, 3, 4
// на самом деле console.log выведет 5, 5, 5, 5, 5
```

Это происходит потому, что цикл завершается с переменной-итератором, все еще установленной на значении, из-за которого цикл завершился: 5. Тайм-ауты, которые выполняются позже, ссылаются на эту же переменную и, следовательно, `console.log` — ее окончательное значение.

При использовании `let` для объявления итератора цикла, за кулисами движок JavaScript фактически объявляет новую переменную-итератор при каждой итерации цикла. Каждый `setTimeout` ссылается на этот отдельный экземпляр, и поэтому `console.log` будет выводить ожидаемое значение: значение переменной-итератора при выполнении данной итерации цикла.

```
for (let i = 0; i < 5; ++i) {
    setTimeout(() => console.log(i), 0)
}
// console.log-и выведут 0, 1, 2, 3, 4
```

Такое декларативное поведение для каждой итерации применимо для всех стилей циклов for, включая циклы `for-in` и `for-of`.

Объявления 'const'

`const` ведет себя так же, как и `let`, но с одним важным отличием — он должен быть инициализирован значением, и это значение не может быть переопределено после объявления. Попытка изменить переменную `const` приведет к ошибке во время выполнения.

```
const age = 26;
age = 36; // TypeError: assignment to a constant

// const по-прежнему запрещает избыточное объявление
const name = 'Matt';
const name = 'Nicholas'; // SyntaxError

// const все еще принадлежит области видимости блока
const name = 'Matt';
if (true) {
    const name = 'Nicholas';
}
console.log(name); // Matt
```

Объявление `const` применяется только в отношении ссылки на переменную, на которую оно указывает. Если переменная `const` ссылается на объект, она не нарушает ограничения `const` для изменения свойств внутри этого объекта.

```
const person = {};
person.name = 'Matt'; // допустимо
```

Несмотря на то что движок JavaScript создает новые экземпляры переменных-итераторов `let` для циклов `for`, и то, что `const`-переменные ведут себя аналогично переменным `let`, нельзя использовать `const` для объявления итераторов цикла:

```
for (const i = 0; i < 10; ++i) {} // TypeError: assignment to constant variable
```

Однако если объявить переменную цикла `for`, которая не будет изменяться, `const` допустим — именно потому, что для каждой итерации объявляется новая переменная. Это особенно актуально в случае циклов `for-of` и `for-in`:

```
let i = 0;
for (const j = 7; i < 5; ++i) {
    console.log(j);
}
// 7, 7, 7, 7, 7

for (const key in {a: 1, b: 2}) {
    console.log(key);
}
// a, b

for (const value of [1,2,3,4,5]) {
    console.log(value);
}
// 1, 2, 3, 4, 5
```

Стили объявлений и наилучшие методики

Введение `let` и `const` в ECMAScript 6 предоставляет объективно лучший инструментарий для языка в форме повышенной точности области видимости объявлений и семантики. Ни для кого не секрет, что причудливое поведение объявлений `var` заставляло сообщество JavaScript долгие годы рвать на себе волосы из-за всех проблем, которые оно вызывало. После ввода этих новых ключевых слов стали появляться все более распространенные шаблоны, которые могут улучшить качество кода.

Не используйте var

С применением `let` и `const` большинство разработчиков обнаружат, что им больше не нужно нигде использовать `var` в кодовой базе. Шаблоны, возникающие при ограничении объявления только `let`- и `const`-переменных, будут служить для обеспечения более высокого качества кодовой базы благодаря тщательному управлению областью видимости переменных, локальностью объявления и правильностью `const`.

const предпочтительнее, чем let

Использование объявлений `const` позволяет браузеру во время выполнения принудительно задавать постоянные переменные, а также позволяет инструментам статического анализа кода предвидеть незаконные операции переназначения. Поэтому многие разработчики предпочитают по умолчанию объявлять переменные как `const`, при условии, что в какой-то момент не потребуется переназначать их значения. Это позволяет разработчикам более конкретно рассуждать о значениях, которые, как они знают, никогда не будут изменены, а также быстро обнаруживать непредвиденное поведение в тех случаях, когда код пытается выполнить непредвиденное переназначение.

ТИПЫ ДАННЫХ

В ECMAScript есть шесть простых типов данных, также называемых *примитивными типами* (primitive types): неопределенный (undefined), нулевой (null), логический (boolean), числовой (number), строковый (string) и символьный (symbol). Символьный тип был недавно введен в ECMAScript 6. Есть также один сложный тип данных (object), который представляет собой неупорядоченный список пар имен и значений. Поскольку определить собственные типы данных в ECMAScript нельзя, все значения представляются с помощью одного из этих семи типов. Может показаться, что этого недостаточно, но у типов данных в ECMAScript есть динамические аспекты, благодаря которым каждый из них работает сразу за нескольких.

Оператор typeof

Поскольку ECMAScript типизирован слабо, для работы с ним необходим какой-то способ определения типа данных переменной. Для получения этой информации можно применить к значению оператор `typeof`, который возвращает одну из следующих строк:

- `"undefined"`, если значение не определено;
- `"boolean"`, если значение имеет логический тип;
- `"string"`, если значение является строкой;
- `"number"`, если значение является числом;
- `"object"`, если значение является объектом (отличным от функции) или значением `null`;
- `"function"`, если значение является функцией;
- `"symbol"`, если значение является символом.

Оператор `typeof` вызывается следующим образом:

```
var message = "some string";
alert(typeof message);     // "string"
alert(typeof(message));    // "string"
alert(typeof 95);          // "number"
```

Здесь оператору `typeof` передаются переменная (`message`) и числовой литерал. Поскольку `typeof` — это оператор, а не функция, заключать операнды в скобки не требуется (хотя можно).

Иногда `typeof` возвращает странные, но технически правильные значения. Так, `typeof null` возвращает строку `"object"`, потому что специальное значение `null` считается ссылкой на пустой объект.

> **ПРИМЕЧАНИЕ** Технически функции считаются в ECMAScript объектами, а не отдельным типом данных. Однако они имеют некоторые специальные свойства, вследствие чего оператор `typeof` отличает их от других объектов.

Тип Undefined

Неопределенный тип (undefined) содержит единственное специальное значение `undefined`. Такое значение имеет переменная, объявленная с помощью оператора `var` или `let`, но не инициализированная:

```
let message;
console.log(message == undefined);    // true
```

Здесь переменная `message` объявляется без инициализации. Сравнение с литеральным значением `undefined` показывает, что они равны. Этот пример идентичен следующему:

```
let message = undefined;
console.log(message == undefined);    // true
```

Теперь переменная `message` явно инициализируется значением `undefined`, но это не требуется, потому что по умолчанию любая переменная без инициализации получает значение `undefined`.

> **ПРИМЕЧАНИЕ** Вообще говоря, переменным не следует явно присваивать значение `undefined`. Оно предоставляется преимущественно для сравнения и было добавлено только в третьей редакции ECMA-262, чтобы формализовать различие между указателем на пустой объект (`null`) и неинициализированной переменной.

Имейте в виду, что переменная со значением `undefined` отличается от переменной, которая вообще не определена. Рассмотрим пример:

```
let message;       // переменная объявляется, но имеет значение undefined

// следующая переменная не объявляется
// let age

console.log(message);    // "undefined"
console.log(age);        // ошибка
```

В этом примере в первом оповещении выводится значение переменной `message`, то есть `"undefined"`. Во втором случае в функцию `console.log()` передается необъявленная переменная `age`, что приводит к ошибке. Для необъявленной переменной возможна только одна полезная операция: вызов оператора `typeof` (вызов `delete` для необъявленной переменной не приведет к ошибке в нестрогом режиме, но пользы от этого никакой).

И для неинициализированной, и для необъявленной переменных оператор `typeof` возвращает значение `"undefined"`, что немного запутывает. Взгляните на следующий пример:

```
let message;    // переменная объявляется, но имеет значение undefined

// следующая переменная не объявляется
// let age
```

```
console.log(typeof message);  // "undefined"
console.log(typeof age);      // "undefined"
```

В обоих случаях `typeof` возвращает строку `"undefined"`. Определенный смысл в этом есть, потому что никаких реальных операций нельзя выполнить ни для одной из переменных, хотя технически они совершенно разные.

> **ПРИМЕЧАНИЕ** Несмотря на то что неинициализированные переменные автоматически получают значение `undefined`, рекомендуется всегда выполнять инициализацию. Так вы будете знать, что оператор `typeof` возвращает строку `"undefined"` из-за того, что переменная не была объявлена, а не потому, что она просто не инициализирована.

Значение `undefined` является ложным; таким образом, вы можете более кратко проверить его там, где необходимо. Имейте в виду, однако, что многие другие возможные значения также ложны, поэтому будьте осторожны в сценариях, где нужно проверить точное значение `undefined`, а не просто ложное значение:

```
let message; // this variable is declared but has a value of undefined
// 'age' не объявлена

if (message) {
    // Данный блок не выполнится
}

if (!message) {
    // Данный блок выполнится
}

if (age) {
    // Здесь код вернет ошибку
}
```

Тип Null

Нулевой тип (Null) также содержит единственный элемент — специальное значение `null`. Логически `null` — это указатель на пустой объект, поэтому оператор `typeof` возвращает для него строку `"object"`:

```
let car = null;
console.log(typeof car);    // "object"
```

При определении переменной, которая позднее будет содержать объект, рекомендуется инициализировать ее именно значением `null`. Это позволяет явно проверять, была ли назначена переменной ссылка на объект, например:

```
if (car != null) {
    // какие-то действия с car
}
```

Значение undefined является производным от null, так что в ECMA-262 они определены как нестрого равные:

```
console.log(null == undefined);  // true
```

При сравнении значений null и undefined с помощью оператора == всегда возвращается true, но помните, что этот оператор преобразует свои операнды (см. далее).

Несмотря на то что значения null и undefined связаны, используются они поразному. Как уже отмечалось, никогда не следует явно присваивать переменной значение undefined, но к null это не относится. Каждый раз, когда нужный объект недоступен, вместо него следует использовать null. Это отражает тот факт, что значение null было введено как указатель на пустой объект, и подчеркивает его отличие от undefined.

Тип null является ложным; таким образом, можно более кратко проверить его там, где необходимо. Имейте в виду, однако, что многие другие возможные значения также ложны, поэтому будьте осторожны в сценариях, где нужно проверить точное значение null, а не просто ложное значение:

```
let message = null;
let age;

if (message) {
    // Данный блок не выполнится
}

if (!message) {
    // Данный блок выполнится
}

if (age) {
    // Данный блок не выполнится
}

if (!age) {
    // Данный блок выполнится
}
```

Тип Boolean

Логический тип (Boolean) — один из наиболее часто используемых в ECMAScript типов данных и имеет только два литеральных значения: true и false. Они отличаются от числовых значений: true не равно 1, а false не равно 0. Присвоить логические значения переменным можно следующим образом:

```
let found = true;
let lost = false;
```

Имейте в виду, что литералы true и false чувствительны к регистру, так что True и False (и эти же слова с другими сочетаниями прописных и строчных букв) являются допустимыми идентификаторами, но не логическими значениями.

Хотя литеральных логических значений всего два, в ECMAScript логические эквиваленты есть у всех значений. Для преобразования значения в его логический эквивалент используется специальная функция приведения типов `Boolean()`:

```
let message = "Hello world!";
let messageAsBoolean = Boolean(message);
```

В этом примере строка `message` преобразуется в логическое значение и сохраняется в переменной `messageAsBoolean`. Функция `Boolean()` может принимать и данные других типов, но всегда возвращает логическое значение. Правила преобразования значения в `true` или `false` зависят как от самого значения, так и от его типа (см. таблицу).

ТИП ДАННЫХ	ЗНАЧЕНИЯ, ПРЕОБРАЗУЕМЫЕ В TRUE	ЗНАЧЕНИЯ, ПРЕОБРАЗУЕМЫЕ В FALSE
`boolean`	`true`	`false`
`string`	Любая непустая строка	"" (пустая строка)
`number`	Любое ненулевое число (включая бесконечность)	`0`, `NaN` (см. раздел «NaN» далее в этой главе)
`object`	Любой объект	`null`
`undefined`	—	`undefined`

Важно понимать эти преобразования, потому что управляющие инструкции вроде `if` выполняют их автоматически, например:

```
let message = "Hello world!";
if (message) {
    console.log("Value is true");
}
```

В этом примере результат console.log выводится на экран, потому что строка `message` автоматически преобразуется в логический эквивалент (`true`). Внимательно следите за тем, какие переменные используются в управляющих инструкциях. Ошибочное указание объекта вместо логического значения может радикально изменить логику приложения.

Тип Number

Пожалуй, наиболее интересным типом данных в ECMAScript является числовой (Number). Он служит для представления целых чисел и чисел с плавающей точкой (которые в ряде языков называются числами с двойной точностью) в формате IEEE-754. Для поддержки чисел разных типов предусмотрено несколько разных форматов числовых литералов.

Самый простой из них — формат десятичного числа, которое можно ввести непосредственно:

```
let intNum = 55;    // целое число
```

Целые числа также можно представлять как восьмеричные или шестнадцатеричные литералы. В восьмеричном литерале первой цифрой является нуль (0), за которым следует последовательность восьмеричных цифр (от 0 до 7). Если в литерале обнаруживается цифра не из этого диапазона, начальный нуль игнорируется и число интерпретируется как десятичное, например:

```
let octalNum1 = 070;  // 56 в восьмеричном формате
let octalNum2 = 079;  // недопустимое восьмеричное значение –
                      // интерпретируется как 79
let octalNum3 = 08;   // недопустимое восьмеричное значение –
                      // интерпретируется как 8
```

В строгом режиме попытка использовать восьмеричные литералы приведет к синтаксической ошибке.

Шестнадцатеричный литерал содержит знаки 0x в первых двух позициях (не чувствительны к регистру), а затем — любое количество шестнадцатеричных цифр (от 0 до 9 и от A до F). Буквы могут быть и строчными, и прописными, например:

```
let hexNum1 = 0xA;     // 10 в шестнадцатеричном формате
let hexNum2 = 0x1f;    // 31 в шестнадцатеричном формате
```

Числа, созданные в восьмеричном или шестнадцатеричном формате, во всех арифметических операциях используются как десятичные.

> **ПРИМЕЧАНИЕ** Способ хранения чисел в JavaScript позволяет представить положительный нуль (+0) и отрицательный нуль (-0). Они всегда эквивалентны, но помечаются в книге знаками для ясности.

Значения с плавающей точкой

Чтобы определить значение с плавающей точкой, необходимо ввести десятичную точку и как минимум одну цифру после нее. Нуль перед десятичной точкой необязателен, но лучше его указывать. Вот некоторые примеры:

```
let floatNum1 = 1.1;
let floatNum2 = 0.1;
let floatNum3 = .1;    // допустимо, но не рекомендуется
```

Из-за того что для хранения значений с плавающей точкой требуется вдвое больше памяти, чем для целых чисел, ECMAScript по возможности преобразует значения в целые числа. Если после десятичной точки нет разрядов, число становится целым. Если значение не имеет дробной части (например, 1.0), оно также преобразуется в целое число, например:

```
let floatNum1 = 1.;    // нет разрядов после десятичной точки –
                       // интерпретируется как целое число 1
let floatNum2 = 10.0;  // нет дробной части –
                       // интерпретируется как целое число 10
```

Очень большие и очень малые числа с плавающей точкой можно представлять в *экспоненциальном формате* (e-notation), в котором значения умножаются на 10 в соответствующей степени. В ECMAScript значение в экспоненциальном формате состоит из числа (целого или с плавающей точкой), прописной или строчной буквы E и показателя степени числа 10, например:

```
var floatNum = 3.125e7  // 31250000
```

В этом примере в экспоненциальном формате записано число 31 250 000. Можете понимать эту запись как умножение числа 3.125 на 10^7.

Экспоненциальный формат можно также использовать для представления очень малых чисел, таких как 0.00000000000000003, что можно сокращенно записать как 3e-17. По умолчанию ECMAScript переводит в экспоненциальный формат любые значения с плавающей точкой, содержащие как минимум шесть нулей после точки (например, 0.0000003 преобразуется в 3e-7).

Значения с плавающей точкой представляются с точностью до 17-го десятичного разряда, но все равно они гораздо менее точны в арифметических вычислениях, чем целые числа. Например, сложение 0.1 и 0.2 дает в результате 0.30000000000000004 вместо 0.3. Такие небольшие ошибки округления затрудняют проверку конкретных значений с плавающей точкой, например:

```
if (a + b == 0.3) {      // не делайте так!
    console.log("You got 0.3.");
}
```

Здесь сумма двух чисел проверяется на равенство 0.3. Этот код правильно сработает, если сложить 0.05 и 0.25 или 0.15 и 0.15, но для чисел 0.1 и 0.2 будет получен чуть больший результат, и оповещение не появится. Никогда не проверяйте, равно ли значение с плавающей точкой конкретному числу.

> **ПРИМЕЧАНИЕ** Ошибки округления – это побочный эффект арифметических операций над числами с плавающей точкой формата IEEE-754, а не особенность ECMAScript. В других языках, в которых используется этот формат, наблюдаются такие же проблемы.

Диапазон значений

Из-за ограничений памяти в ECMAScript используется не весь диапазон чисел. Наименьшее и наибольшее числа, которые могут быть представлены в ECMAScript, хранятся в свойствах `Number.MIN_VALUE` и `Number.MAX_VALUE` и равны в большинстве браузеров 5e-324 и 1.7976931348623157e+308 соответственно. Результат вычисления, не попадающий в диапазон чисел JavaScript, автоматически приравнивается к специальному значению `Infinity`. Любое отрицательное число, которое не может быть представлено, считается отрицательной бесконечностью (`-Infinity`), а положительное — положительной бесконечностью (`Infinity`).

Если вычисление возвращает одну из бесконечностей, это значение не может использоваться ни в каких дальнейших вычислениях, потому что `Infinity` не имеет числового представления. Чтобы определить, конечно ли значение, можно задействовать функцию `isFinite()`. Она возвращает `true`, только если ее аргумент находится между минимальным и максимальным значениями:

```
let result = Number.MAX_VALUE + Number.MAX_VALUE;
console.log(isFinite(result));    // false
```

Хотя значения, не попадающие в диапазон конечных чисел, применяются в вычислениях редко, это все же случается. Работая с очень большими или очень малыми числами, будьте внимательны.

> **ПРИМЕЧАНИЕ** Положительную и отрицательную бесконечности можно получить с помощью свойств `Number.NEGATIVE_INFINITY` и `Number.POSITIVE_INFINITY`, которые возвращают значения `-Infinity` и `Infinity` соответственно.

NaN

Специальное числовое значение `NaN`, то есть *не число* (Not a Number), указывает, что операция, которая должна была возвратить число, не сделала этого (но и не сгенерировала ошибку). Например, в большинстве других языков деление на 0 приводит к критической ошибке, но в ECMAScript вместо этого возвращается значение `NaN`, что позволяет продолжить работу.

Значение `NaN` имеет два уникальных свойства. Во-первых, любая операция с `NaN` (например, `NaN/10`) всегда возвращает `NaN`, что может быть проблемой, если вычисление выполняется в несколько этапов. Во-вторых, `NaN` не равно никакому значению, в том числе и другому `NaN`. Например, следующее выражение возвращает `false`:

```
console.log(NaN == NaN);    // false
```

По этой причине ECMAScript предоставляет функцию `isNaN()`, которая принимает один аргумент любого типа и определяет, является ли он «не числом». При передаче значения в `isNaN()` предпринимается попытка преобразовать его в число. Некоторые нечисловые значения, например строка `"10"` или логическое значение, без проблем преобразуются в числа, а для аргументов, которые не могут быть преобразованы в числа, эта функция возвращает `true`, например:

```
console.log(isNaN(NaN));      // true
console.log(isNaN(10));       // false – 10 является числом
console.log(isNaN("10"));     // false – может быть преобразовано в число 10
console.log(isNaN("blue"));   // true  – не может быть преобразовано в число
console.log(isNaN(true));     // false – может быть преобразовано в число 1
```

В этом примере проверяются пять разных значений. Первая проверка выполняется для самого значения `NaN` и, конечно, возвращает `true`. В следующих двух выражениях проверяются число 10 и строка `"10"`, и в обоих случаях возвращается `false`, потому

что оба значения эквивалентны 10. Строка `"blue"` не может быть преобразована в число, поэтому следующий вызов возвращает `true`. Наконец, логическое значение `true` может быть преобразовано в число 1, поэтому последний вызов возвращает `false`.

ПРИМЕЧАНИЕ Функцию `isNaN()` можно вызывать и для объектов, хотя обычно так не делают. В этом случае сначала вызывается метод `valueOf()`, чтобы определить, может ли возвращенное значение быть преобразовано в число. Если нет, вызывается метод `toString()` и проверяется возвращенное им значение. Таков типичный способ работы встроенных в ECMAScript функций и операторов (подробности см. в разделе «Операторы»).

Преобразование чисел

Есть три функции преобразования нечисловых значений в числа: `Number()`, `parseInt()` и `parseFloat()`. Функцию приведения типов `Number()` можно использовать с любым типом данных, а две другие функции служат для преобразования строк в числа. Каждая из них обрабатывает один и тот же ввод по-своему.

Функция `Number()`преобразует значения по следующим правилам.

- Логические значения `true` и `false` преобразуются в 1 и 0 соответственно.
- Числа возвращаются без изменений.
- Значение `null` преобразуется в 0.
- Значение `undefined` преобразуется в `NaN`.
- Для строк действуют особые правила:
  - если строка содержит только числовые символы с начальным знаком «плюс» или «минус» либо без знака, она всегда преобразуется в десятичное число. Так, `Number("1")` преобразуется в 1, `Number("123")` — в 123, а `Number("011")` — в 11 (начальные нули игнорируются);
  - если строка содержит значение с плавающей точкой в правильном формате, такое как `"1.1"`, она преобразуется в соответствующее число с плавающей точкой (начальные нули также игнорируются);
  - если строка содержит шестнадцатеричное значение в правильном формате, такое как `"0xf"`, она преобразуется в соответствующее целое число;
  - если строка пуста (не содержит знаков), она преобразуется в 0;
  - если строка содержит что-то, отличное от предыдущих вариантов, она преобразуется в `NaN`.
- Для объектов вызывается метод `valueOf()`, а возвращенное им значение преобразуется по предыдущим правилам. Если это преобразование дает результат `NaN`, вызывается метод `toString()` и применяются правила преобразования строк.

Уже по количеству правил для функции `Number()` ясно, что преобразование различных типов данных в числа — непростая задача. Вот несколько примеров:

```
let num1 = Number("Hello world!");  // NaN
let num2 = Number("");              // 0
let num3 = Number("000011");        // 11
let num4 = Number(true);            // 1
```

Строка `"Hello world!"` преобразуется в `NaN`, потому что ей не соответствует никакое числовое значение, а для пустой строки возвращается 0. Из строки `"000011"` получается число 11, потому что начальные нули игнорируются, а из значения `true` — число 1.

> **ПРИМЕЧАНИЕ** Унарный оператор «плюс», описываемый далее в разделе «Операторы», работает так же, как функция `Number()`.

Из-за сложных правил преобразования строк функцией `Number()` для получения целых чисел из строк лучше использовать функцию `parseInt()`. Она тщательнее проверяет строку, выясняя, соответствует ли она числовому шаблону. Начальные пробелы в строке игнорируются до первого символа, отличного от пробельного. Если этот первый символ не является числом, знаком «минус» или «плюс», функция `parseInt()` всегда возвращает `NaN` (в том числе для пустой строки, в отличие от функции `Number()`, которая возвращает 0). Если же первым символом является число, знак «плюс» или «минус», функция переходит ко второму символу, третьему и т. д., вплоть до конца строки или до нечислового символа. Например, строка `"1234blue"` преобразуется в 1234, потому что слово `"blue"` игнорируется. Аналогично `"22.5"` преобразуется в 22, потому что десятичная точка не используется в целых числах.

Если первый знак в строке — цифра, функция `parseInt()` также распознает различные форматы целых чисел (десятичный, восьмеричный и шестнадцатеричный). Так, строка, которая начинается с префикса `"0x"`, интерпретируется как шестнадцатеричное целое число, а если строка начинается с префикса `"0"`, после чего следует цифра, значение обрабатывается как восьмеричное.

Вот несколько примеров преобразования, поясняющих, что происходит:

```
let num1 = parseInt("1234blue");   // 1234
let num2 = parseInt("");           // NaN
let num3 = parseInt("0xA");        // 10 в шестнадцатеричном формате
let num4 = parseInt(22.5);         // 22
let num5 = parseInt("70");         // 70 в десятичном формате
let num6 = parseInt("0xf");        // 15 в шестнадцатеричном формате
```

Чтобы было проще следить за числовыми форматами, можно использовать с функцией `parseInt()` второй аргумент: основание системы счисления (иначе говоря, количество цифр в ней). Если вы знаете, что число является шестнадцатеричным, можете передать в функцию вместе с ним значение 16, чтобы число наверняка было обработано правильно:

```
let num = parseInt("0xAF", 16);    // 175
```

В этом случае можно опустить начальные знаки "0x":

```
let num1 = parseInt("AF", 16);     // 175
let num2 = parseInt("AF");         // NaN
```

Здесь первое преобразование выполняется правильно, а второе нет, потому что первая функция parseInt() получает указание, что строка содержит шестнадцатеричное число. Вторая же функция выясняет, что первый знак не является цифрой, и автоматически прекращает работу.

Передача основания системы счисления может радикально изменить результат преобразования, например:

```
let num1 = parseInt("10", 2);     // двоичное число 2
let num2 = parseInt("10", 8);     // восьмеричное число 8
let num3 = parseInt("10", 10);    // десятичное число 10
let num4 = parseInt("10", 16);    // шестнадцатеричное число 16
```

Поскольку при единственном аргументе функция parseInt() сама решает, как его интерпретировать, рекомендуется всегда передавать ей основание системы счисления во избежание ошибок.

ПРИМЕЧАНИЕ Чаще всего вы будете передавать в функцию parseInt() десятичные числа, указывая 10 в качестве второго аргумента.

Функция parseFloat() работает подобно parseInt(), считывая каждый знак с нулевой позиции до конца строки или знака, которого не может быть в числе с плавающей точкой. Это означает, что одна десятичная точка допустима, но при обнаружении второй точки остальная часть строки игнорируется. Например, строка "22.34.5" преобразуется в 22.34.

Другое отличие функции parseFloat() в том, что она всегда пропускает начальные нули. Она распознает любые форматы чисел с плавающей точкой, описанные ранее, а также десятичный формат (начальные нули всегда игнорируются). Шестнадцатеричные числа всегда преобразуются в 0. Поскольку функция parseFloat() анализирует только десятичные значения, основание системы счисления она не принимает. Наконец, если строка содержит значение без десятичной точки или с нулем после нее, функция parseFloat() возвращает целое число. Вот несколько примеров:

```
let num1 = parseFloat("1234blue");    // 1234 (целое число)
let num2 = parseFloat("0xA");         // 0
let num3 = parseFloat("22.5");        // 22.5
let num4 = parseFloat("22.34.5");     // 22.34
let num5 = parseFloat("0908.5");      // 908.5
let num6 = parseFloat("3.125e7");     // 31250000
```

Тип String

Строковый тип (String) — это последовательности 16-разрядных знаков Юникода (в том числе пустые). Строки могут быть заключены в двойные (`"`), одинарные (`'`) или обратные (`` ` ``) кавычки:

```
let firstName = "John";
let lastName = 'Jacob';
let lastName = `Jingleheimerschmidt`
```

В отличие от некоторых языков, в которых интерпретация строки зависит от типа кавычек, в ECMAScript все эти варианты синтаксиса одинаковы, но кавычки в начале и конце строки не должны различаться. Например, такое выражение вызовет синтаксическую ошибку:

```
let firstName = 'Nicholas";    // синтаксическая ошибка — разные кавычки
```

Символьные литералы

Строковый тип данных поддерживает несколько литералов, представляющих полезные символы, в том числе непечатаемые. Они указаны в таблице.

ЛИТЕРАЛ	ЗНАЧЕНИЕ
`\n`	Перевод строки
`\t`	Табуляция
`\b`	Возврат на одну позицию
`\r`	Возврат каретки
`\f`	Перевод страницы
`\\`	Обратная косая черта (\)
`\'`	Одинарная кавычка (`'`). Используется, если строка заключена в одинарные кавычки, например: `'He said, \'hey.\''`
`\"`	Двойная кавычка (`"`). Используется, если строка заключена в двойные кавычки, например: `"He said, \"hey.\""`
`\``	Обратная кавычка (`` ` ``). Используется, если строка заключена в обратные кавычки, например: `` `He said, \`hey.\`` ``
`\xnn`	Знак с шестнадцатеричным кодом *nn* (где *n* — шестнадцатеричный знак 0-F). Пример: \x41 is equivalent to «A»
`\unnnn`	Знак Юникода с шестнадцатеричным кодом *nnnn* (где *n* — шестнадцатеричный знак 0-F). Пример: \u03a3 is equivalent to the Greek character Σ

Символьные литералы могут находиться в любом месте строки и интерпретируются как один символ, например:

```
let text = "This is the letter sigma: \u03a3.";
```

В этом примере переменная `text` содержит 28 символов, хотя одна только экранирующая последовательность состоит из 6 символов. Однако она представляет один символ и считается одним символом.

Длину любой строки можно узнать с помощью свойства `length`:

```
console.log(text.length);    // 28
```

Это свойство возвращает количество 16-разрядных символов в строке.

> **ПРИМЕЧАНИЕ** Если строка содержит символы удвоенной разрядности, свойство `length` может возвратить неправильный результат. Стратегии смягчения последствий этого подробно описаны в главе 5 «Ссылочные типы».

Природа строк

После создания строки в ECMAScript изменить ее значение невозможно. Для изменения строковой переменной первоначальная строка уничтожается, а затем переменной присваивается другая строка с новым значением:

```
let lang = "Java";
lang = lang + "Script";
```

Здесь переменная `lang` определяется со значением `"Java"`. В следующей строке она переопределяется как конкатенация строк `"Java"` и `"Script"` и получает значение `"JavaScript"`. Для этого создается новая строка с местом, достаточным для хранения 10 символов, и затем оно заполняется фрагментами `"Java"` и `"Script"`. Наконец, строки `"Java"` и `"Script"` уничтожаются, потому что они больше не нужны. Все это происходит за кулисами, и именно поэтому старые браузеры (например, Firefox до версии 1.0 и Internet Explorer 6.0) очень медленно выполняли конкатенацию строк. Более поздние версии этих браузеров обрабатывают строки эффективнее.

Преобразование значения в строку

Преобразовать значение в строку можно двумя способами. Первый — использовать метод `toString()`, который есть почти у любого значения. Он просто возвращает строковый эквивалент значения, например:

```
let age = 11;
let ageAsString = age.toString();      // строка "11"
let found = true;
let foundAsString = found.toString();  // строка "true"
```

Метод `toString()` доступен для чисел, логических значений, объектов и строк (да, у каждой строки есть метод `toString()`, который просто возвращает копию строки). Для значений `null` и `undefined` вызвать его нельзя.

В большинстве случаев у метода `toString()` нет аргументов, но при использовании с числами он может принимать один аргумент: основание целевой системы счисления. По умолчанию метод `toString()` всегда возвращает строковое представление числа в десятичном формате, но если задано основание, он может вывести значение с двоичным, восьмеричным, шестнадцатеричным и любым другим допустимым основанием, например:

```
let num = 10;
console.log(num.toString());    // "10"
console.log(num.toString(2));   // "1010"
console.log(num.toString(8));   // "12"
console.log(num.toString(10));  // "10"
console.log(num.toString(16));  // "a"
```

Этот пример показывает, как изменяется вывод метода `toString()` для чисел при изменении основания. Значение 10 может выводиться в самых разных числовых форматах, но по умолчанию (без аргумента) используется основание 10.

Если переменная способна принимать значение `null` или `undefined`, вы можете использовать функцию приведения типов `String()`, которая всегда возвращает строку независимо от полученного значения. Она работает следующим образом:

- если у значения есть метод `toString()`, он вызывается (без аргументов), а затем возвращается результат;
- для значения `null` возвращается строка `"null"`;
- для значения `undefined` возвращается строка `"undefined"`.

Рассмотрим следующий пример:

```
let value1 = 10;
let value2 = true;
let value3 = null;
let value4;

console.log(String(value1));    // "10"
console.log(String(value2));    // "true"
console.log(String(value3));    // "null"
console.log(String(value4));    // "undefined"
```

Здесь в строки преобразуются число, логическое значение, значения `null` и `undefined`. Для числа и логического значения возвращается такой же результат, как если бы был вызван метод `toString()`. Поскольку для значений `"null"` и `"undefined"` он недоступен, для них метод `String()` просто возвращает текстовые литералы.

ПРИМЕЧАНИЕ Также можно преобразовать значение в строку, добавив к нему пустую строку (`""`) с помощью оператора «плюс» (см. далее раздел «Операторы»).

Шаблонные строки

В ECMAScript 6 появилась возможность определять строки с использованием шаблонных строк. В отличие от одинарных и двойных кавычек, шаблонные строки учитывают символы новой строки и могут быть определены в нескольких строках:

```
let myMultiLineString = 'first line\nsecond line';
let myMultiLineTemplateLiteral = `first line
second line`;

console.log(myMultiLineString);
// first line
// second line"

console.log(myMultiLineTemplateLiteral);
// first line
// second line

console.log(myMultiLineString === myMultiLinetemplateLiteral); // true
```

Как следует из названия, шаблонные строки особенно полезны при определении шаблонов, таких как HTML:

```
let pageHTML = `
<div>
    <a href="#">
        <span>Jake</span>
    </a>
</div>`;
```

Поскольку шаблонные строки будут в точности учитывать пробелы внутри обратных кавычек, при их определении необходимо будет соблюдать особую осторожность. Правильно отформатированная шаблонная строка может иметь неправильный отступ:

```
// Эта шаблонная строка содержит 25 пробелов за символом переноса строки
let myTemplateLiteral = `first line
                         second line`;
console.log(myTemplateLiteral.length); // 47

// Эта шаблонная строка начинается с символа переноса строки
let secondTemplateLiteral = `
first line
second line`;
console.log(secondTemplateLiteral[0] === '\n'); // true

// В этой шаблонной строке нет неожиданных пробельных символов
let thirdTemplateLiteral = `first line
second line`;
console.log(thirdTemplateLiteral[0]);
// first line
// second line
```

Интерполяция

Одной из наиболее полезных функций шаблонных строк является их поддержка интерполяции, которая позволяет вставлять значения в одном или нескольких

местах внутри одного непрерывного определения. Технически шаблонные строки не являются строками, они представляют собой специальные синтаксические выражения JavaScript, которые преобразуются в строки.

Шаблонные строки вычисляются сразу же после определения и преобразования в экземпляр строки, и любые интерполированные переменные будут извлечены из их непосредственной области видимости.

Это можно сделать с помощью выражения JavaScript внутри `${}`:

```
let value = 5;
let exponent = 'second';

// Ранее интерполяция осуществлялась следующим образом:
let interpolatedString =
    value + ' to the ' + exponent + ' power is ' + (value * value);

// То же самое с использованием шаблонных строк:
let interpolatedTemplateLiteral =
    `${ value } to the ${ exponent } power is ${ value * value }`;

console.log(interpolatedString); // 5 to the second power is 25
console.log(interpolatedTemplateLiteral); // 5 to the second power is 25
```

Интерполируемое значение в конечном итоге будет приведено к строке с помощью `toString()`, но любое выражение JavaScript можно безопасно интерполировать. Вложенные шаблонные строки безопасны без необходимости экранирования:

```
console.log(`Hello, ${ `World` }!`); // Hello, World!
```

`toString()` вызывается для приведения результата выражения к строке:

```
let foo = { toString: () => 'World' };
console.log(`Hello, ${ foo }!`); // Hello, World!
```

Допускается вызов функций и методов внутри интерполированных выражений:

```
function capitalize(word) {
    return `${ word[0].toUpperCase() }${ word.slice(1) }`;
}
console.log(`${ capitalize('hello') }, ${ capitalize('world') }!`); // Hello,
World!
```

Кроме того, шаблоны могут безопасно интерполировать свои предыдущие значения:

```
let value = '';
function append() {
    value = `${value}abc`
    console.log(value);
}
append(); // abc
append(); // abcabc
append(); // abcabcabc
```

Теговые функции шаблонных строк

Шаблонные строки также поддерживают возможность определения *теговых функций*, которые могут определять пользовательское поведение интерполяции. Теговая функция передает отдельные фрагменты после разбиения шаблона токеном интерполяции и после вычисления выражений.

Теговая функция определяется как обычная функция и применяется к шаблонной строке путем добавления к ней префикса, как показано в следующем коде. Теговая функция будет передавать шаблонную строку, разбитую на части: первый аргумент — это массив простых строк, а остальные аргументы — это результаты вычисленных выражений. Возвращаемое значение этой функции будет строкой, вычисленной из шаблонной строки.

Лучше всего продемонстрировать это на примере:

```
let a = 6;
let b = 9;
function simpleTag(strings, aValExpression, bValExpression, sumExpression) {
    console.log(strings);
    console.log(aValExpression);
    console.log(bValExpression);
    console.log(sumExpression);

    return 'foobar';
}

let untaggedResult = `${ a } + ${ b } = ${ a + b }`;
let taggedResult = simpleTag`${ a } + ${ b } = ${ a + b }`;
// ["", " + ", " = ", ""]
// 6
// 9
// 15

console.log(untaggedResult); // "6 + 9 = 15"
console.log(taggedResult); // "foobar"
```

Поскольку в данном случае количество аргументов выражения переменно, обычно целесообразно использовать оператор распространения для объединения их в одну коллекцию:

```
let a = 6;
let b = 9;

function simpleTag(strings, ...expressions) {
    console.log(strings);
    for(const expression of expressions) {
        console.log(expression);
    }

    return 'foobar';
}
let taggedResult = simpleTag`${ a } + ${ b } = ${ a + b }`;
// ["", " + ", " = ", ""]
// 6
// 9
```

```
// 15

console.log(taggedResult); // "foobar"
```

Для шаблонной строки с *n* интерполированными значениями число аргументов выражения для теговой функции всегда будет *n*, а количество частей строки в первом аргументе всегда будет ровно *n + 1*. Поэтому, если нужно «сжать» строки и вычисленные выражения вместе в возвращаемую строку по умолчанию, можно сделать это следующим образом:

```
let a = 6;
let b = 9;

function zipTag(strings, ...expressions) {
    return strings[0] +
        expressions.map((e, i) => `${e}${strings[i + 1]}`)
                   .join('');
}

let untaggedResult = `${ a } + ${ b } = ${ a + b }`;
let taggedResult = zipTag`${ a } + ${ b } = ${ a + b }`;

console.log(untaggedResult); // "6 + 9 = 15"
console.log(taggedResult); // "6 + 9 = 15"
```

Простые строки

Шаблонные строки также можно использовать, чтобы предоставить доступ к простому содержимому шаблонной строки, не преобразуя его в реальные символьные представления, такие как новая строка или символ Unicode.

Это можно сделать с помощью теговой функции `String.raw`, доступной по умолчанию.

```
// Пример Unicode
// \u00A9 — знак охраны авторского права
console.log(`\u00A9`); // ©
console.log(String.raw`\u00A9`); // \u00A9

// Пример с символом переноса строки
console.log(`first line\nsecond line`);
// first line
// second line

console.log(String.raw`first line\nsecond line`); // "first line\nsecond line"

// Это не работает для реальных символов переноса строки: они не пройдут
// преобразование из открытых экранированных эквивалентов
console.log(`first line
second line`);
// first line
// second line

console.log(String.raw`first line
second line`);
// first line
// second line
```

Простые значения также доступны как свойство для каждого элемента в коллекции частей строки внутри теговой функции:

```
function printRaw(strings) {
    console.log('Actual characters:');
    for (const string of strings) {
        console.log(string);
    }

    console.log('Escaped characters;');
        for (const rawString of strings.raw) {
            console.log(rawString);
        }
}

printRaw`\u00A9${ 'and' }\n`;
// Actual characters:
// ©
// (newline)
// Escaped characters:
// \u00A9
// \n
```

Тип Symbol

Тип данных Symbol является новым в ECMAScript 6. Символы — примитивные значения, а экземпляры символов уникальны и неизменны. Цель символа — быть гарантированным уникальным идентификатором для свойств объекта без риска столкновения свойств.

Хотя может показаться, что символы имеют некоторые общие черты с частными свойствами, они не предназначены для обеспечения поведения частных свойств (особенно потому, что Object API предлагает методы, позволяющие легко обнаруживать свойства символов). Вместо этого символы предназначены для использования в качестве уникальных токенов, которые можно использовать для обозначения специальных свойств чем-то отличным от строки.

Основное использование символов

Символы создаются с помощью функции `Symbol`. Поскольку это собственный примитивный тип, оператор `typeof` идентифицирует символ как `symbol`.

```
let sym = Symbol();
console.log(typeof sym);    // symbol
```

При вызове функции можно указать необязательную строку, которая может использоваться для идентификации экземпляра символа при отладке. Предоставленная вами строка полностью отделена от определения или идентичности символа:

```
let genericSymbol = Symbol();
let otherGenericSymbol = Symbol();
```

```
let fooSymbol = Symbol('foo');
let otherFooSymbol = Symbol('foo');

console.log(genericSymbol == otherGenericSymbol);    // false
console.log(fooSymbol == otherFooSymbol);            // false
```

Символы не имеют синтаксиса шаблонных строк, и это определяет ключевое значение для их целевого использования. Спецификация, регулирующая работу символов, позволяет создавать новый экземпляр `Symbol` и использовать его для ввода нового свойства объекта с гарантией того, что существующее свойство объекта не будет перезаписано — независимо от того, использует ли он строку или символ как ключ.

```
let genericSymbol = Symbol();
console.log(genericSymbol);    // Symbol()

let fooSymbol = Symbol('foo');
console.log(fooSymbol);        // Symbol(foo);
```

Важно отметить, что функция `Symbol` не может использоваться с ключевым словом `new`. Это сделано для того, чтобы избежать оборачивания объекта символа, как это возможно с `Boolean`, `String` и `Number`, которые поддерживают поведение конструктора и создают экземпляр примитивного объекта-обертки:

```
let myBoolean = new Boolean();
console.log(typeof myBoolean);    // "object"

let myString = new String();
console.log(typeof myString);     // "object"

let myNumber = new Number();
console.log(typeof myNumber);     // "object"

let mySymbol = new Symbol();      // TypeError: Symbol is not a constructor
```

Если необходимо использовать объект-обертку, запустите функцию `Object()`:

```
let mySymbol = Symbol();
let myWrappedSymbol = Object(mySymbol);
console.log(typeof myWrappedSymbol); // "object"
```

Использование глобального реестра символов

В сценариях, где различные части среды выполнения хотели бы совместно и повторно использовать экземпляр символа, можно создавать и повторно использовать символы в глобальном реестре символов со строковыми ключами.

Такое поведение может быть достигнуто с использованием `Symbol.for()`:

```
let fooGlobalSymbol = Symbol.for('foo');
console.log(typeof fooGlobalSymbol); // "object"
```

`Symbol.for()` является идемпотентной операцией для каждого строкового ключа. При первом вызове с заданной строкой он проверит глобальный реестр выполнения,

обнаружит, что символа не существует, сгенерирует новый экземпляр символа и добавит его в реестр. Дополнительные вызовы с тем же строковым ключом проверят глобальный реестр выполнения, обнаружат, что символ существует для этой строки, и вместо этого вернут этот экземпляр символа.

```
let fooGlobalSymbol = Symbol.for('foo');          // создание нового символа
let otherFooGlobalSymbol = Symbol.for('foo');     // переиспользование существующего
                                                  // символа

console.log(fooGlobalSymbol === otherFooGlobalSymbol); // true
```

Символы, определенные в глобальном реестре, полностью отличаются от символов, созданных с помощью `Symbol()`, даже если они имеют общее описание:

```
let localSymbol = Symbol('foo');
let globalSymbol = Symbol.for('foo');

console.log(localSymbol === globalSymbol); // false
```

Глобальный реестр требует использования строковых ключей, поэтому все, что было предоставлено в качестве аргумента `Symbol.for()`, будет преобразовано в строку. Кроме того, ключ, используемый для реестра, также будет использоваться в качестве описания символа.

```
let emptyGlobalSymbol = Symbol.for();
console.log(emptyGlobalSymbol);    // Symbol(undefined)
```

Можно проверить нахождение символа в глобальном реестре, используя `Symbol.keyFor()`, который принимает символ и возвращает ключ глобальной строки для этого глобального символа, или `undefined`, если символ не является глобальным.

```
// Создаем глобальный символ
let s = Symbol.for('foo');
console.log(Symbol.keyFor(s));     // foo

// Создаем обычный символ
let s2 = Symbol('bar');
console.log(Symbol.keyFor(s2));    // undefined
```

Использование `Symbol.keyFor()` с не-символом вернет `TypeError`.

```
Symbol.keyFor(123); // TypeError: 123 is not a symbol
```

Использование символов в качестве свойств

Везде, где обычно можно использовать свойство строки или числа, также можно использовать символ. Это включает в себя литеральные свойства объекта и `Object.defineProperty()/Object.defineProperties()`. Объектный литерал может использовать символ только как свойство внутри синтаксиса вычисляемого свойства.

```
let s1 = Symbol('foo'),
    s2 = Symbol('bar'),
    s3 = Symbol('baz'),
    s4 = Symbol('qux');
```

```
let o = {
    [s1]: 'foo val'
};
// Также верно: o[s1] = 'foo val';

console.log(o);
// {Symbol{foo}: foo val}

Object.defineProperty(o, s2, {value: 'bar val'});

console.log(o);
// {Symbol{foo}: foo val, Symbol(bar): bar val}

Object.defineProperties(o, {
    [s3]: {value: 'baz val'},
    [s4]: {value: 'qux val'}
});

console.log(o);
// {Symbol{foo}: foo val, Symbol(bar): bar val,
// Symbol{baz}: baz val, Symbol(qux): qux val}
```

Как `Object.getOwnPropertyNames()` возвращает массив обычных свойств для экземпляра объекта, так и `Object.getOwnPropertySymbols()` возвращает массив свойств символа для экземпляра объекта. Возвращаемые значения этих двух методов являются взаимоисключающими. `Object.getOwnPropertyDescriptors()` вернет объект, содержащий как обычные, так и символьные дескрипторы свойств. `Reflect.ownKeys()` вернет оба типа ключей:

```
let s1 = Symbol('foo'),
    s2 = Symbol('bar');

let o = {
    [s1]: 'foo val',
    [s2]: 'bar val',
    baz: 'baz val',
    qux: 'qux val'
};

console.log(Object.getOwnPropertySymbols(o));
// [Symbol(foo), Symbol(bar)]

console.log(Object.getOwnPropertyNames(o));
// ["baz", "qux"]

console.log(Object.getOwnPropertyDescriptors(o));
// {baz: {...}, qux: {...}, Symbol(foo): {...}, Symbol(bar): {...}}

console.log(Reflect.ownKeys(o));
// ["baz", "qux", Symbol(foo), Symbol(bar)]
```

Поскольку свойство считается ссылкой на этот символ в памяти, символы не теряются при непосредственном создании и использовании в качестве свойств. Однако отказ от сохранения явной ссылки на свойство означает, что для восстановления ключа свойства потребуется пройти через все свойства символа объекта:

```
let o = {
    [Symbol('foo')]: 'foo val',
    [Symbol('bar')]: 'bar val'
};

console.log(o);
// {Symbol(foo): "foo val", Symbol(bar): "bar val"}

let barSymbol = Object.getOwnPropertySymbols(o)
              .find((symbol) => symbol.toString().match(/bar/));

console.log(barSymbol);
// Symbol(bar)
```

Общеизвестные символы

Наряду с добавлением символов ECMAScript 6 также представил коллекцию *общеизвестных символов*, которые будут использоваться по всему языку для представления внутреннего поведения языка для прямого доступа, переопределения или эмуляции. Эти общеизвестные символы существуют как строковые свойства в фабричной функции `Symbol`.

Один из основных способов использования этих общеизвестных символов — переопределение их для изменения поведения конструкций родного языка. Например, поскольку известно, как цикл `for-of` будет использовать свойство `Symbol.iterator` для любого объекта, предоставленного ему, можно предоставить настраиваемое определение значения `Symbol.iterator` в настраиваемом объекте, чтобы управлять поведением `for-of`, когда этот объект передается в `Symbol.iterator`.

В общеизвестных символах нет ничего особенного, они являются обычными строковыми свойствами в глобальном Symbol, которые задают экземпляр символа. Каждое четко определенное свойство символа является свойством только для чтения, неперечислимым и неконфигурируемым.

> **ПРИМЕЧАНИЕ** В дискуссиях о спецификации ECMAScript вы часто будете видеть эти символы, на которые ссылаются их имена спецификации, с префиксом @@. Например, @@iterator ссылается на `Symbol.iterator`.

Symbol.asyncIterator

Согласно спецификации ECMAScript, этот символ используется как свойство для «метода, который возвращает AsyncIterator по умолчанию для объекта. Вызывается семантикой выражения `for-await-of`». Он используется для идентификации функции, которая реализует API асинхронного итератора.

Языковые конструкции, такие как цикл `for-await-of`, используют эту функцию для выполнения асинхронной итерации. Они будут вызывать функцию с ключом

`Symbol.asyncIterator` и ожидать, что она вернет объект, который реализует Iterator API. Во многих случаях он будет принимать форму AsyncGenerator, объекта, который реализует этот API:

```
class Foo {
    async *[Symbol.asyncIterator]() {}
}

let f = new Foo();

console.log(f[Symbol.asyncIterator]());
// AsyncGenerator {<suspended>}
```

В частности, объект, созданный функцией `Symbol.asyncIterator`, должен последовательно создавать экземпляры `Promise` с помощью метода `next()`. Это может выражаться через явное определение метода `next()` или неявно через функцию асинхронного генератора:

```
class Emitter {
    constructor(max) {
        this.max = max;
        this.asyncIdx = 0;
    }

    async *[Symbol.asyncIterator]() {
        while(this.asyncIdx < this.max) {
            yield new Promise((resolve) => resolve(this.asyncIdx++));
        }
    }
}

async function asyncCount() {
    let emitter = new Emitter(5);

    for await(const x of emitter) {
        console.log(x);
    }
}

asyncCount();
// 0
// 1
// 2
// 3
// 4
```

ПРИМЕЧАНИЕ `Symbol.asyncIterator` является частью спецификации ES2018, поэтому его поддерживают только очень современные версии браузеров. Более подробную информацию об асинхронной итерации и цикле `for-await-of` можно найти в приложении А.

Symbol.hasInstance

Согласно спецификации ECMAScript, этот символ используется как свойство для «метода, который определяет, распознает ли объект конструктора объект как один из экземпляров конструктора. Вызывается семантикой оператора `instanceof`». Оператор `instanceof` предоставляет способ определения того, имеет ли экземпляр объекта прототип в своей цепочке прототипов. Типичное использование `instanceof` заключается в следующем:

```
function Foo() {}
let f = new Foo();
console.log(f instanceof Foo);    // true

class Bar {}
let b = new Bar();
console.log(b instanceof Bar);    // true
```

В ES6 оператор `instanceof` использует функцию `Symbol.hasInstance` для оценки этой взаимосвязи. `Symbol.hasInstance` включает функцию, которая повторяет то же поведение, но с операндами в обратном порядке:

```
function Foo() {}
let f = new Foo();
console.log(Foo[Symbol.hasInstance](f));    // true

class Bar {}
let b = new Bar();
console.log(Bar[Symbol.hasInstance](b));    // true
```

Это свойство определено в прототипе `Function`, и поэтому оно автоматически доступно по умолчанию для всех определений функций и классов. Поскольку оператор `instanceof` будет искать определение свойства в цепочке прототипов, как и любое другое свойство, можно переопределить функцию унаследованного класса как статический метод:

```
class Bar {}
class Baz extends Bar {
    static [Symbol.hasInstance]() {
        return false;
    }
}

let b = new Baz();
console.log(Bar[Symbol.hasInstance](b)); // true
console.log(b instanceof Bar); // true
console.log(Baz[Symbol.hasInstance](b)); // false
console.log(b instanceof Baz); // false
```

Symbol.isConcatSpreadable

Согласно спецификации ECMAScript, этот символ используется как свойство для «булевого значения свойства, которое, если оно истинно, указывает, что объект

должен быть сведен к элементам массива с помощью `Array.prototype.concat()`». Метод `Array.prototype.concat` в ES6 выберет способ присоединения объекта, подобного массиву, к экземпляру массива в зависимости от типа передаваемого объекта. Значение `Symbol.isConcatSpreadable` позволяет переопределить это поведение.

Объекты массива по умолчанию будут сведены в существующий массив; значение `false`, или ложное, добавит весь объект в массив. Подобные массиву объекты по умолчанию будут добавлены в массив; значение `true`, или истинное, будет компилировать массивоподобный объект в экземпляр массива. Другие объекты, не похожие на массивы, будут игнорироваться, если для `Symbol.isConcatSpreadable` задано значение `true`.

```
let initial = ['foo'];

let array = ['bar'];
console.log(array[Symbol.isConcatSpreadable]);    // undefined
console.log(initial.concat(array));               // ['foo', 'bar']
array[Symbol.isConcatSpreadable] = false;
console.log(initial.concat(array));               // ['foo', Array(1)]

let arrayLikeObject = { length: 1, 0: 'baz' };
console.log(arrayLikeObject[Symbol.isConcatSpreadable]);    // undefined
console.log(initial.concat(arrayLikeObject));               // ['foo', {...}]
arrayLikeObject[Symbol.isConcatSpreadable] = true;
console.log(initial.concat(arrayLikeObject));               // ['foo', 'baz']

let otherObject = new Set().add('qux');
console.log(otherObject[Symbol.isConcatSpreadable]);    // undefined
console.log(initial.concat(otherObject));               // ['foo', Set(1)]
otherObject[Symbol.isConcatSpreadable] = true;
console.log(initial.concat(otherObject));               // ['foo']
```

Symbol.iterator

Согласно спецификации ECMAScript, этот символ используется как свойство для «метода, который возвращает итератор по умолчанию для объекта. Вызывается семантикой выражений `for-of`». Он используется для идентификации функции, которая реализует API итератора.

Языковые конструкции, такие как цикл `for-of`, используют эту функцию для выполнения итерации. Они будут вызывать функцию с ключом `Symbol.iterator` и ожидать, что она возвратит объект, который реализует Iterator API. Во многих случаях результат примет форму генератора, объекта, который реализует этот API:

```
class Foo {
    *[Symbol.iterator]() {}
}

let f = new Foo();

console.log(f[Symbol.iterator]());
// Generator {<условный>}
```

В частности, объект, созданный функцией `Symbol.iterator`, должен последовательно генерировать значения с помощью метода `next()`. Это может быть через явное определение метода `next()` или неявное, через функцию генератора:

```
class Emitter {
    constructor(max) {
        this.max = max;
        this.idx = 0;
    }

    *[Symbol.iterator]() {
        while(this.idx < this.max) {
            yield this.idx++;
        }
    }
}

function count() {
    let emitter = new Emitter(5);

    for (const x of emitter) {
        console.log(x);
    }
}

count();
// 0
// 1
// 2
// 3
// 4
```

ПРИМЕЧАНИЕ Определение итератора дано в главе 7 «Итераторы и генераторы».

Symbol.match

Согласно спецификации ECMAScript, этот символ используется как свойство для «метода регулярного выражения, который сопоставляет регулярное выражение со строкой. Вызывается методом `String.prototype.match()`». Метод `String.prototype.match()` будет использовать функцию, обозначенную `Symbol.match`, для оценки выражения. В прототипе регулярного выражения эта функция определена по умолчанию, и поэтому все экземпляры регулярного выражения являются допустимыми параметрами для метода `String` по умолчанию:

```
console.log(RegExp.prototype[Symbol.match]);
// ƒ [Symbol.match]() { [встроенный код] }

console.log('foobar'.match(/bar/));
// ["bar", index: 3, input: "foobar", groups: undefined]
```

Передача в этот метод чего-либо, отличного от регулярного выражения, приведет к его преобразованию в объект `RegExp`. Если вам это не нужно и вы хотите сделать так, чтобы метод использовал параметр напрямую, можно передать что-то, отличное от экземпляра регулярного выражения, в метод `match()`, определив функцию `Symbol.match`, чтобы заменить поведение, которое в противном случае было бы вызвано регулярным выражением. Эта функция имеет единственный параметр, который является экземпляром строки, для которой вызывается `match()`. Возвращаемое значение не ограничено:

```
class FooMatcher {
    static [Symbol.match](target) {
        return target.includes('foo');
    }
}

console.log('foobar'.match(FooMatcher)); // true
console.log('barbaz'.match(FooMatcher)); // false

class StringMatcher {
    constructor(str) {
        this.str = str;
    }

    [Symbol.match](target) {
        return target.includes(this.str);
    }
}
console.log('foobar'.match(new StringMatcher('foo'))); // true
console.log('barbaz'.match(new StringMatcher('qux'))); // false
```

Symbol.replace

Согласно спецификации ECMAScript, этот символ используется как свойство для «метода регулярного выражения, который заменяет совпадающие подстроки строки. Вызывается методом `String.prototype.replace()`». Метод `String.prototype.replace()` будет использовать функцию, обозначенную `Symbol.replace`, для оценки выражения. В прототипе регулярного выражения эта функция определена по умолчанию, и поэтому все экземпляры регулярного выражения являются допустимыми параметрами для метода `String` по умолчанию:

```
console.log(RegExp.prototype[Symbol.replace]);
// ƒ [Symbol.replace]() { [встроенный код] }

console.log('foobarbaz'.replace(/bar/, 'qux'));
// 'fooquxbaz'
```

Передача в этот метод чего-либо, отличного от регулярного выражения, приведет к его преобразованию в объект `RegExp`. Если вы хотите обойти это поведение и сделать так, чтобы метод использовал параметр напрямую, можно передать что-то, отличное от экземпляра регулярного выражения, в метод `replace()`, определив функцию `Symbol.replace`, чтобы заменить поведение, которое в противном случае

было бы вызвано регулярным выражением. Эта функция имеет два параметра, первый из которых является экземпляром строки, для которой вызывается `replace()`, а второй — строкой для замены. Возвращаемое значение не ограничено:

```
class FooReplacer {
    static [Symbol.replace](target, replacement) {
        return target.split('foo').join(replacement);
    }
}

console.log('barfoobaz'.replace(FooReplacer, 'qux'));
// "barquxbaz"

class StringReplacer {
    constructor(str) {
        this.str = str;
    }

    [Symbol.replace](target, replacement) {
        return target.split(this.str).join(replacement);
    }
}

console.log('barfoobaz'.replace(new StringReplacer('foo'), 'qux'));
// "barquxbaz"
```

Symbol.search

Согласно спецификации ECMAScript, этот символ используется как свойство для «метода регулярного выражения, который возвращает индекс строки, соответствующий регулярному выражению. Вызывается методом `String.prototype.search()`». Метод `String.prototype.search()` будет использовать функцию, обозначенную `Symbol.search`, для оценки выражения. В прототипе регулярного выражения эта функция определена по умолчанию, и поэтому все экземпляры регулярного выражения являются допустимыми параметрами для метода `String` по умолчанию:

```
console.log(RegExp.prototype[Symbol.search]);
// ƒ [Symbol.search]() { [встроенный код] }

console.log('foobar'.search(/bar/));
// 3
```

Передача в этот метод чего-либо, отличного от регулярного выражения, приведет к его преобразованию в объект `RegExp`. Если вы хотите обойти это поведение и сделать так, чтобы метод использовал параметр напрямую, можно передать что-то, отличное от экземпляра регулярного выражения, в метод `search()`, определив функцию `Symbol.search`, чтобы заменить поведение, которое в противном случае было бы вызвано регулярным выражением. Эта функция имеет единственный параметр, который является экземпляром строки, для которой вызывается `search()`. Возвращаемое значение не ограничено:

```
class FooSearcher {
    static [Symbol.search](target) {
        return target.indexOf('foo');
    }
}

console.log('foobar'.search(FooSearcher)); // 0
console.log('barfoo'.search(FooSearcher)); // 3
console.log('barbaz'.search(FooSearcher)); // -1

class StringSearcher {
    constructor(str) {
        this.str = str;
    }

    [Symbol.search](target) {
        return target.indexOf(this.str);
    }
}

console.log('foobar'.search(new StringSearcher('foo'))); // 0
console.log('barfoo'.search(new StringSearcher('foo'))); // 3
console.log('barbaz'.search(new StringSearcher('qux'))); // -1
```

Symbol.species

В соответствии со спецификацией ECMAScript, этот символ используется в качестве свойства для «свойства со значением функции, которое является функцией конструктора, используемой для создания производных объектов». Он чаще всего применяется для встроенных типов, которые предоставляют методы, создающие экземпляры производных объектов для возвращаемого значения метода экземпляра. Определение статического метода получения свойств с помощью `Symbol.species` позволяет переопределить прототип для вновь созданного экземпляра:

```
class Bar extends Array {}
class Baz extends Array {
    static get [Symbol.species]() {
        return Array;
    }
}

let bar = new Bar();
console.log(bar instanceof Array); // true
console.log(bar instanceof Bar);   // true
bar = bar.concat('bar');
console.log(bar instanceof Array); // true
console.log(bar instanceof Bar);   // true

let baz = new Baz();
console.log(baz instanceof Array); // true
console.log(baz instanceof Baz);   // true
baz = baz.concat('baz');
console.log(baz instanceof Array); // true
console.log(baz instanceof Baz);   // false
```

Symbol.split

Согласно спецификации ECMAScript, этот символ используется как свойство для «метода регулярного выражения, который разбивает строку по индексам, соответствующим регулярному выражению. Вызывается методом `String.prototype.split()`». Метод `String.prototype.split()` будет использовать функцию, обозначенную `Symbol.split`, для оценки выражения. В прототипе регулярного выражения эта функция определена по умолчанию, и поэтому все экземпляры регулярного выражения являются допустимыми параметрами для метода `String` по умолчанию:

```
console.log(RegExp.prototype[Symbol.split]);
// ƒ [Symbol.split]() { [втсроенный код] }

console.log('foobarbaz'.split(/bar/));
// ['foo', 'baz']
```

Передача в этот метод чего-либо, отличного от регулярного выражения, приведет к его преобразованию в объект `RegExp`. Если вы хотите обойти это поведение и сделать так, чтобы метод использовал параметр напрямую, можно передать что-то, отличное от экземпляра регулярного выражения, в метод `split()`, определив функцию `Symbol.split`, чтобы заменить поведение, которое в противном случае было бы вызвано регулярным выражением. Эта функция имеет единственный параметр, который является экземпляром строки, для которой вызывается `split()`. Возвращаемое значение не ограничено:

```
class FooSplitter {
    static [Symbol.split](target) {
        return target.split('foo');
    }
}

console.log('barfoobaz'.split(FooSplitter));
// ["bar", "baz"]

class StringSplitter {
    constructor(str) {
        this.str = str;
    }

    [Symbol.split](target) {
        return target.split(this.str);
    }
}

console.log('barfoobaz'.split(new StringSplitter('foo')));
// ["bar", "baz"]
```

Symbol.toPrimitive

Согласно спецификации ECMAScript, этот символ используется как свойство для «метода, который преобразует объект в соответствующее примитивное значение. Вызывается абстрактной операцией `ToPrimitive`». Существует ряд встроенных

операций, которые пытаются привести объект к примитивному значению: строке, числу или неопределенному типу примитива. Для пользовательского экземпляра объекта это поведение можно изменить, определив функцию в свойстве экземпляра `Symbol.toPrimitive`.

Основываясь на строковом параметре, переданном в функцию (`string`, `number` или `default`), можно управлять возвращаемым примитивом:

```
class Foo {}
let foo = new Foo();

console.log(3 + foo); // "3[object Object]"
console.log(3 - foo); // NaN
console.log(String(foo)); // "[object Object]"

class Bar {
    constructor() {
        this[Symbol.toPrimitive] = function(hint) {
            switch (hint) {
                case 'number':
                    return 3;
                case 'string':
                    return 'string bar';
                case 'default':
                default:
                    return 'default bar';
            }
        }
    }
}
let bar = new Bar();

console.log(3 + bar); // "3default bar"
console.log(3 - bar); // 0
console.log(String(bar)); // "string bar"
```

Symbol.toStringTag

Согласно спецификации ECMAScript, этот символ используется для «строкового свойства, используемого при создании строкового описания объекта по умолчанию. Доступ осуществляется при помощи встроенного метода `Object.prototype.toString()`».

Идентификация объекта с помощью метода `toString()` извлекает идентификатор экземпляра, заданный `Symbol.toStringTag`, по умолчанию имеющий значение `Object`. Для встроенных типов это значение уже указано, но экземпляры пользовательских классов требуют явного определения:

```
let s = new Set();

console.log(s); // Set(0) {}
console.log(s.toString()); // [object Set]
console.log(s[Symbol.toStringTag]); // Set
```

```
class Foo {}
let foo = new Foo();

console.log(foo); // Foo {}
console.log(foo.toString()); // [object Object]
console.log(foo[Symbol.toStringTag]); // undefined

class Bar {
    constructor() {
        this[Symbol.toStringTag] = 'Bar';
    }
}
let bar = new Bar();

console.log(bar); // Bar {}
console.log(bar.toString()); // [object Bar]
console.log(bar[Symbol.toStringTag]); // Bar
```

Symbol.unscopables

В соответствии со спецификацией ECMAScript, этот символ используется для «свойства объекта, чьи собственные и унаследованные имена свойств являются именами свойств, исключенных из привязок среды связанного объекта». Установка этого символа таким образом, чтобы он назначил объекту, сопоставляющему соответствующее свойство, значение `true`, предотвратит привязку среды к `with`, как показано здесь:

```
let o = { foo: 'bar' };

with (o) {
    console.log(foo); // bar
}

o[Symbol.unscopables] = {
    foo: true
};

with (o) {
    console.log(foo); // ReferenceError
}
```

> **ПРИМЕЧАНИЕ** Использование `with` не рекомендуется, поэтому не рекомендуется и использование `Symbol.unscopables`.

Тип Object

В ECMAScript объекты создаются как неспецифические сочетания данных и функциональности. Чтобы добавить в программу объект, нужно ввести оператор `new` и указать тип объекта. Для создания собственных объектов разработчики обычно создают экземпляры типа `Object`, а затем добавляют к ним свойства и (или) методы, например:

```
let o = new Object();
```

Этот синтаксис похож на Java, хотя в ECMAScript скобки нужны только при передаче аргументов в конструктор. Если аргументов нет, скобки можно опускать (однако это не рекомендуется):

```
let o = new Object;    // допустимо, но не рекомендуется
```

Сами по себе экземпляры типа `Object` не очень полезны, но важно понимать основы их работы, потому что, подобно типу `java.lang.Object` в Java, тип `Object` в ECMAScript является родительским для всех остальных объектов. Все его свойства и методы есть у других, более специфичных объектов.

Каждый экземпляр `Object` имеет свойства и методы из приведенного списка.

- `constructor` — функция, которая была использована для создания объекта. В предыдущем примере это функция `Object()`.
- `hasOwnProperty(имяСвойства)` — указывает, есть ли у объекта (не у прототипа) данное свойство. Имя свойства должно быть указано как строка (например, `o.hasOwnProperty("name")`).
- `isPrototypeOf(объект)` — определяет, является ли объект прототипом другого объекта (прототипы обсуждаются в главе 5).
- `propertyIsEnumerable(имяСвойства)` — указывает, можно ли перебирать данное свойство в инструкции `for-in` (см. далее). Как и в случае метода `hasOwnProperty()`, имя свойства должно быть строкой.
- `toLocaleString()` — возвращает строковое представление объекта в соответствии с региональными настройками среды выполнения.
- `toString()` — возвращает строковое представление объекта.
- `valueOf()` — возвращает строковый, численный или логический эквивалент объекта, часто совпадающий с результатом вызова `toString()`.

Поскольку тип `Object` является родительским для всех объектов в ECMAScript, эти базовые свойства и методы есть у каждого объекта. Подробные сведения об этом см. в главах 5 и 6.

> **ПРИМЕЧАНИЕ** Технически принципы работы объектов в ECMA-262 относятся не ко всем объектам в JavaScript. Объекты в среде браузера, например BOM- и DOM-объекты, предоставляются и определяются средой. На них не распространяются требования ECMA-262, а потому они могут не наследоваться от типа `Object`.

ОПЕРАТОРЫ

Для работы с данными ECMA-262 предоставляет набор *операторов* (operators), которые варьируются от математических (таких как сложение и вычитание) и поразрядных до операторов отношения и сравнения. ECMAScript-операторы уникальны в том смысле, что их можно задействовать со многими разными значениями,

включая строки, числа, логические значения и даже объекты. Если операторы используются с объектами, для получения операндов обычно вызывается метод `valueOf()` и (или) `toString()`.

Унарные операторы

Операторы, работающие с единственным значением, называются *унарными* (unary operators). Это самые простые операторы в ECMAScript.

Инкремент и декремент

Операторы инкремента и декремента взяты непосредственно из C и имеют две версии: префиксную и постфиксную. Префиксные версии указываются перед операндами, постфиксные — после. Чтобы добавить 1 к числовой переменной с помощью оператора префиксного инкремента, введите перед именем переменной два знака «плюс» (++):

```
let age = 29;
++age;
```

В этом примере значение `age` изменяется на 30 (к предыдущему значению 29 добавляется 1), что эквивалентно следующему коду:

```
let age = 29;
age = age + 1;
```

Префиксный декремент работает похоже, вычитая 1 из числа. Чтобы использовать префиксный декремент, введите перед именем переменной два знака «минус» (--):

```
let age = 29;
--age;
```

Здесь переменная `age` уменьшается до 28 (из 29 вычитается 1).

При использовании префиксного инкремента или декремента значение переменной изменяется до вычисления выражения — в программировании это обычно называют *побочным эффектом* (side effect). Взгляните на следующий код:

```
let age = 29;
let anotherAge = --age + 2;

console.log(age);         // 28
console.log(anotherAge);  // 30
```

В этом примере переменная `anotherAge` инициализируется суммой уменьшенного на единицу значения `age` и числа 2. Поскольку декремент выполняется первым, `age` получает значение 28, а затем к нему добавляется 2, что дает в итоге 30.

Префиксный инкремент и декремент имеют в инструкциях одинаковый приоритет и выполняются слева направо, например:

```
let num1 = 2;
let num2 = 20;
let num3 = --num1 + num2;
let num4 = num1 + num2;
console.log(num3); // 21
console.log(num4); // 21
```

Переменная num3 равна 21, потому что значение num1 уменьшается на 1 перед сложением. Переменная num4 также равна 21, потому что суммируются уже измененные значения.

Постфиксные операторы инкремента и декремента имеют такой же синтаксис (соответственно ++ и --), но указываются после переменной, а не перед ней. Они отличаются от префиксных версий тем, что выполняются после вычисления инструкции, которая их содержит. Иногда это не имеет значения, например:

```
let age = 29;
age++;
```

Преобразование префиксного инкремента в постфиксный ничего не изменило, потому что эти инструкции больше ничего не делают. Однако если добавить другие операции, разница становится очевидной:

```
let num1 = 2;
let num2 = 20;
let num3 = num1-- + num2;
let num4 = num1 + num2;
console.log(num3); // 22
console.log(num4); // 21
```

Как видите, аналогичное преобразование привело к изменению результата. В примере с префиксной версией переменные num3 и num4 были равны 21, тогда как здесь num3 имеет значение 22, а num4 — 21. Дело в том, что при вычислении num3 слагаемое num1 имеет первоначальное значение (2), а в инструкции с num4 используется уменьшенное значение (1).

Все операторы инкремента и декремента работают не только с целыми числами, но и с любыми другими значениями: строками, логическими значениями, числами с плавающей точкой и объектами. При этом применяются такие правила.

- Если операнд — строка, представляющая допустимое число, она преобразуется в число, к которому применяется оператор. Строковая переменная становится числовой.
- Если операнд — строка, которая не является допустимым числом, переменная получает значение NaN и становится числовой.
- Если операнд — логическое значение false, оно преобразуется в 0, а затем применяется оператор. Логическая переменная становится числовой.
- Если операнд — логическое значение true, оно преобразуется в 1, а затем применяется оператор. Логическая переменная становится числовой.

- Если операнд — число с плавающей точкой, оно увеличивается или уменьшается на 1.
- Если операнд — объект, вызывается его метод `valueOf()` (см. главу 5) для получения допустимого операнда, после чего применяются другие правила. Если в результате получается `NaN`, вызывается метод `toString()` и снова применяются другие правила. Объект становится числовой переменной.

Некоторые из этих правил продемонстрированы в следующем примере:

```
let s1 = "2";
let s2 = "z";
let b = false;
let f = 1.1;
let o = {
    valueOf() {
        return -1;
    }
};

s1++;  // результат — число 3
s2++;  // результат - NaN
b++;   // результат — число 1
f--;   // результат — 0.10000000000000009 (из-за неточностей округления)
o--;   // результат — число -2
```

Унарные плюс и минус

Унарные операторы «плюс» и «минус» известны всем со школы и работают в ECMAScript точно так же. Унарный плюс (`+`) указывается перед переменной, и если это число, он ничего не делает:

```
let num = 25;
num = +num
console.log(num); // 25
```

Если унарный плюс применяется к нечисловому значению, оно преобразуется так же, как и при вызове функции приведения типов `Number()`: логические значения `false` и `true` изменяются на 0 и 1, строковые значения анализируются согласно набору правил, а для объектов вызываются их методы `valueOf()` и (или) `toString()`.

Следующий пример поясняет применение унарного оператора «плюс» к разным типам данных:

```
let s1 = "01";
let s2 = "1.1";
let s3 = "z";
let b = false;
let f = 1.1;
let o = {
    valueOf() {
        return -1;
    }
```

```
};

s1 = +s1;  // результат – число 1
s2 = +s2;  // результат – число 1.1
s3 = +s3;  // результат – NaN
b = +b;    // результат – число 0
f = +f;    // число не изменяется (1.1)
o = +o;    // результат – число -1
```

Унарный минус чаще всего используется для изменения знака числа, например:

```
let num = 25;
num = -num;
console.log(num); // -25
```

При использовании с числовым значением унарный минус просто изменяет его знак, как в этом примере. Если значение не является числом, применяются те же правила, что и для унарного оператора «плюс», а затем меняется знак результата:

```
let s1 = "01";
let s2 = "1.1";
let s3 = "z";
let b = false;
let f =  1.1;
let o = {
    valueOf() {
        return -1;
    }
};

s1 = -s1;  // результат – число -1
s2 = -s2;  // результат – число -1.1
s3 = -s3;  // результат – NaN
b = -b;    // результат – число 0
f = -f;    // результат – число -1.1
o = -o;    // результат – число 1
```

Унарные операторы «плюс» и «минус» обычно применяются в простых арифметических операциях, но, как показывают примеры, могут использоваться и для преобразований.

Поразрядные операторы

Операторы из этого раздела работают с числами на самом низком уровне — уровне отдельных битов. Все числа в ECMAScript хранятся в 64-разрядном формате IEEE-754, но поразрядные операции не работают непосредственно с этим представлением. Вместо этого значение преобразуется в 32-разрядное число, для него выполняется нужная операция, а затем результат преобразуется обратно в 64-разрядный формат. Поскольку 64-разрядный формат прозрачен, для разработчика все выглядит так, как если бы существовали только 32-разрядные целые числа, которые мы сейчас и обсудим.

В целых числах со знаком все биты, кроме 32-го, представляют само значение, тогда как 32-й бит определяет знак числа: 0 для положительных чисел и 1 для отрицательных. Значение этого бита, называемого *знаковым* (sign bit), определяет формат остальной части числа. Положительные числа хранятся в настоящем двоичном формате, в котором все биты, кроме знакового, представляют степени двойки: первый бит (бит 0) соответствует 2^0, второй — 2^1 и т. д. Если какие-либо биты не используются, они считаются равными нулю и, по сути, игнорируются. Например, число 18 представляется как 00000000000000000000000000010010, или, сокращенно, как 10010. Эти пять значимых битов и определяют фактическое значение числа (рис. 3.1).

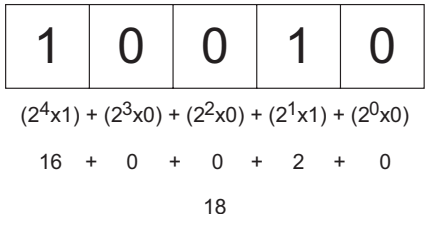

Рис. 3.1

Отрицательные числа также хранятся в двоичном коде, но в формате, который называется *дополнительным кодом* (two's complement). Он вычисляется в три этапа.

1. Определяется двоичное представление абсолютного значения числа (например, для числа –18 сначала определяется двоичное представление 18).
2. Находится обратный код числа. Это означает, что каждый ноль заменяется единицей и наоборот.
3. К результату добавляется 1.

Определим двоичное представление числа –18. Начнем с абсолютного значения (18):

```
0000 0000 0000 0000 0000 0000 0001 0010
```

Далее определим обратный код:

```
1111 1111 1111 1111 1111 1111 1110 1101
```

Наконец, добавим 1 к обратному коду числа:

```
1111 1111 1111 1111 1111 1111 1110 1101
                   1
-------------------------------------
1111 1111 1111 1111 1111 1111 1110 1110
```

Итак, двоичным эквивалентом –18 является 11111111111111111111111111101110. Помните, что при работе с целыми числами со знаком бит 31 недоступен.

ECMAScript делает все возможное, чтобы скрыть от вас всю эту «кухню». Например, при выводе отрицательного числа в виде двоичной строки вы получаете двоичный код абсолютного значения со знаком «минус»:

```
let num = -18;
console.log(num.toString(2));    // "-10010"
```

При преобразовании числа –18 в двоичную строку в результате получается –10010, поскольку предполагается, что этот формат более понятен, чем дополнительный код.

> **ПРИМЕЧАНИЕ** По умолчанию все целые числа представляются в ECMAScript как числа со знаком. Существуют, однако, и целые числа без знака, в которых 32-й бит не указывает знак. Все такие числа положительные, но благодаря дополнительному биту они могут быть больше, чем целые числа со знаком.

Когда поразрядные операторы применяются к числам, 64-разрядные числа преобразуются в 32-разрядные, выполняется операция, а затем 32-разрядный результат снова сохраняется в 64-разрядном числе. Это создает иллюзию, что вы работаете с настоящими 32-разрядными числами, из-за чего двоичные операции в ECMAScript похожи на свои аналоги в других языках. У преобразований есть также любопытный побочный эффект: специальные значения `NaN` и `Infinity` в поразрядных операциях интерпретируются как 0.

Если поразрядный оператор применяется к нечисловому значению, оно автоматически преобразуется в число с помощью функции `Number()` и только затем выполняется операция. Ее результатом является число.

Поразрядное НЕ

Поразрядное НЕ (`~`) просто возвращает обратный код числа. Это один из немногих ECMAScript-операторов для двоичной математики. Рассмотрим пример:

```
let num1 = 25;     // двоичное число 00000000000000000000000000011001
let num2 = ~num1;  // двоичное число 11111111111111111111111111100110
console.log(num2);       // -26
```

Здесь поразрядное НЕ применяется к числу 25, что дает в результате –26. По сути, число просто инвертируется, а затем новое значение уменьшается на 1. Тот же результат можно получить следующим образом:

```
let num1 = 25;
let num2 = -num1 – 1;
console.log(num2);      // "-26"
```

Хотя результаты одинаковы, поразрядное НЕ выполняется гораздо быстрее, потому что оно работает на самом низком уровне.

Поразрядное И

Поразрядный оператор И (`&`) сопоставляет биты двух значений, а затем выполняет операцию И для каждой пары битов в одной позиции по правилам из следующей таблицы истинности:

БИТ ПЕРВОГО ЧИСЛА	БИТ ВТОРОГО ЧИСЛА	РЕЗУЛЬТАТ
1	1	1
1	0	0
0	1	0
0	0	0

Поразрядное И возвращает 1, только если биты обоих операндов в этой позиции равны 1. Если хотя бы один из них равен 0, то и бит результата равен 0.

В следующем примере поразрядное И выполняется для чисел 25 и 3:

```
let result = 25 & 3;
console.log(result);      // 1
```

В результате получается 1. Почему? Взгляните сами:

```
 25 = 0000 0000 0000 0000 0000 0000 0001 1001
  3 = 0000 0000 0000 0000 0000 0000 0000 0011
--------------------------------------------
AND = 0000 0000 0000 0000 0000 0000 0000 0001
```

Как видите, только в одной позиции биты обоих операндов равны 1. Из-за этого все остальные биты результата обнуляются, что в итоге дает 1.

Поразрядное ИЛИ

Поразрядный оператор ИЛИ (|) также работает с двумя операндами, при этом применяются правила из следующей таблицы истинности:

БИТ ПЕРВОГО ЧИСЛА	БИТ ВТОРОГО ЧИСЛА	РЕЗУЛЬТАТ
1	1	1
1	0	1
0	1	1
0	0	0

Поразрядное ИЛИ возвращает 1, если хотя бы один бит равен 1, и 0, если оба бита равны 0.

Рассмотрим выполнение поразрядного ИЛИ для чисел из предыдущего примера:

```
let result = 25 | 3;
console.log(result);      // 27
```

Результат 27 получается следующим образом:

```
 25 = 0000 0000 0000 0000 0000 0000 0001 1001
  3 = 0000 0000 0000 0000 0000 0000 0000 0011
--------------------------------------------
 OR = 0000 0000 0000 0000 0000 0000 0001 1011
```

Каждый единичный бит любого из операндов переходит в результат. Двоичный код 11011 соответствует числу 27.

Поразрядное исключающее ИЛИ

Поразрядное исключающее ИЛИ (^) выполняется для двух операндов по правилам из следующей таблицы:

БИТ ПЕРВОГО ЧИСЛА	БИТ ВТОРОГО ЧИСЛА	РЕЗУЛЬТАТ
1	1	0
1	0	1
0	1	1
0	0	0

Оно отличается от обычного поразрядного ИЛИ тем, что возвращает 1, только если один бит равен 1 (если оба бита равны 1, возвращается 0).

Выполним исключающее ИЛИ для тех же чисел, 25 и 3:

```
let result = 25 ^ 3;
console.log(result);      // 26
```

В результате получается 26:

```
 25 = 0000 0000 0000 0000 0000 0000 0001 1001
  3 = 0000 0000 0000 0000 0000 0000 0000 0011
--------------------------------------------
XOR = 0000 0000 0000 0000 0000 0000 0001 1010
```

Этот пример отличается от предыдущего только тем, что первый бит результата обнуляется, поскольку в обоих операндах он равен 1. Все остальные единичные биты переходят в результат, потому что у них нет пары во втором операнде. Двоичному коду 11010 соответствует число 26 (заметьте, что оно на единицу меньше, чем результат поразрядного ИЛИ).

Сдвиг влево

Оператор сдвига влево (<<) сдвигает все биты числа влево на указанное количество позиций. Например, если сдвинуть число 2 (10 в двоичном формате) на 5 битов влево, получится 64 (1000000 в двоичном формате):

```
let oldValue = 2;              // 10 в двоичном формате
let newValue = oldValue << 5;  // 1000000 в двоичном формате (десятичное 64)
```

В этом примере при сдвиге числа влево справа остаются пять пустых битов, которые заполняются нулями, чтобы в результате получилось полное 32-разрядное число (рис. 3.2).

Рис. 3.2

Имейте в виду, что при сдвиге влево знак числа не меняется. Например, если сдвинуть –2 на пять битов влево, получится число –64, а не 64.

Сдвиг вправо с сохранением знака

Оператор сдвига вправо с сохранением знака (>>) в точности противоположен сдвигу влево. Например, если сдвинуть 64 вправо на 5 битов с сохранением знака, получится 2:

```
let oldValue = 64;              // 1000000 в двоичном формате
let newValue = oldValue >> 5;  // 10 в двоичном формате или 2 в десятичном
```

При сдвиге вправо пустые биты появляются слева от числа, но справа от знакового бита (рис. 3.3). Чтобы получилось полное число, в них копируется знаковый бит.

Рис. 3.3

Сдвиг вправо с заполнением нулями

Для положительных чисел оператор сдвига вправо с заполнением нулями (>>>) эквивалентен сдвигу вправо с сохранением знака. Если сдвинуть 64 вправо на 5 битов с заполнением нулями, получится 2, как и в предыдущем примере:

```
let oldVallue = 64;              // 1000000 в двоичном формате
let newValue = oldValue >>> 5;  // 10 в двоичном формате или 2 в десятичном
```

Отрицательные числа — совсем другое дело. В отличие от сдвига вправо с сохранением знака, теперь пустые биты заполняются нулями независимо от знака числа, что в случае отрицательных чисел дает другой результат. При сдвиге вправо с заполнением нулями отрицательное число в двоичном формате обрабатывается как положительное число. Поскольку отрицательное число является дополнением его абсолютного значения до двух, число становится очень большим, например:

```
let oldValue = -64;       // двоичное число 11111111111111111111111111000000
let newValue = oldValue >>> 5;  // 134217726 в десятичном формате
```

Как видите, при сдвиге числа –64 вправо на 5 битов с заполнением нулями получается 134 217 726. Это происходит потому, что в двоичном формате –64 записывается как 11111111111111111111111111000000, которое в данном случае считается положительным числом 4 294 967 232. После сдвига мы получаем 0000011111111 11111111111111110, или десятичное число 134 217 726.

Логические операторы

Логические операторы относятся к важнейшим элементам любого языка программирования. Если бы нельзя было проверить, как соотносятся два значения, условные инструкции и циклы были бы бесполезны. Логических операторов три: НЕ, И и ИЛИ.

Логическое НЕ

Логический оператор НЕ (`!`) в ECMAScript можно применять к любым значениям. Он преобразует операнд в логическое значение, выполняет его отрицание и возвращает логическое значение, при этом действуют следующие правила:

- Если операнд — объект, возвращается `false`.
- Если операнд — пустая строка, возвращается `true`.
- Если операнд — непустая строка, возвращается `false`.
- Если операнд — число 0, возвращается `true`.
- Если операнд — любое число, отличное от 0 (включая `Infinity`), возвращается `false`.
- Если операнд — значение `null`, возвращается `true`.
- Если операнд — значение `NaN`, возвращается `true`.
- Если операнд — значение `undefined`, возвращается `true`.

Вот некоторые примеры:

```
console.log(!false);    // true
console.log(!"blue");   // false
console.log(!0);        // true
console.log(!NaN);      // true
console.log(!"");       // true
console.log(!12345);    // false
```

С помощью двух логических НЕ можно также преобразовать значение в его логический эквивалент, что имитирует функцию приведения типов `Boolean()`. Первый оператор НЕ возвращает логическое значение независимо от типа операнда, а второй отрицает это значение, предоставляя логический аналог первоначальной переменной. Это дает тот же результат, что и вызов функции `Boolean()`:

```
console.log(!!"blue");  // true
console.log(!!0);       // false
console.log(!!NaN);     // false
console.log(!!"");      // false
console.log(!!12345);   // true
```

Логическое И

Логический оператор И (`&&`) применяется к двум значениям, например:

```
let result = true && false;
```

Он работает согласно следующей таблице истинности:

ОПЕРАНД 1	ОПЕРАНД 2	РЕЗУЛЬТАТ
`true`	`true`	`true`
`true`	`false`	`false`
`false`	`true`	`false`
`false`	`false`	`false`

Логическое И действует с операндами любых типов. Если один из операндов не является примитивным логическим значением, логическое И не всегда возвращает логическое значение. Вместо этого применяются такие правила:

- Если первый операнд — объект, всегда возвращается второй операнд.
- Если второй операнд — объект, а первый эквивалентен значению `true`, возвращается этот объект.
- Если оба операнда — объекты, возвращается второй операнд.
- Если хотя бы один из операндов — значение `null`, возвращается `null`.
- Если хотя бы один из операндов — значение `NaN`, возвращается `NaN`.
- Если хотя бы один из операндов — значение `undefined`, возвращается `undefined`.

Логическое И поддерживает сокращенное вычисление: если первого операнда достаточно для определения результата, второй операнд не оценивается. Так, если первый операнд — `false`, то каким бы ни был второй операнд, результатом не может быть `true`. Рассмотрим пример:

```
let found = true;
let result = (found && someUndeclaredVariable);    // ошибка
console.log(result);    // эта строка никогда не выполняется
```

При выполнении логического И в этом коде возникает ошибка, потому что переменная `someUndeclaredVariable` не объявлена. Значение `found` равно `true`, поэтому интерпретатор переходит к оценке переменной `someUndeclaredVariable`, которая не объявлена, а потому не может использоваться в логическом И. Если изменить значение `found` на `false`, ошибка не возникает:

```
let found = false;
let result = (found && someUndeclaredVariable);    // ошибки нет
console.log(result);    // все работает
```

В этом случае оповещение выводится в консоль. Переменная `someUndeclaredVariable` здесь тоже не определена, но она никогда не оценивается, потому что первый операнд равен `false`, и следовательно, результатом операции тоже может быть только `false`. Используя логическое И, не забывайте о сокращенной схеме его вычисления.

Логическое ИЛИ

Логический оператор ИЛИ (`||`) используется в ECMAScript следующим образом:

```
let result = true || false;
```

Он работает согласно следующей таблице истинности:

ОПЕРАНД 1	ОПЕРАНД 2	РЕЗУЛЬТАТ
`true`	`true`	`true`
`true`	`false`	`true`
`false`	`true`	`true`
`false`	`false`	`false`

Если один из операндов не является логическим, логическое ИЛИ не всегда возвращает логическое значение. Вместо этого применяются такие правила:

- Если первый операнд — объект, возвращается он.
- Если первый операнд эквивалентен значению `false`, возвращается второй операнд.
- Если оба операнда — объекты, возвращается первый операнд.
- Если оба операнда — значения `null`, возвращается `null`.
- Если оба операнда — значения `NaN`, возвращается `NaN`.
- Если оба операнда — значения `undefined`, возвращается `undefined`.

Как и логическое И, логическое ИЛИ поддерживает сокращенные вычисления. В этом случае второй операнд не оценивается, если первый эквивалентен значению `true`, например:

```
let found = true;
let result = (found || someUndeclaredVariable);    // ошибки нет
console.log(result);    // все работает
```

Как и в предыдущем примере, переменная `someUndeclaredVariable` не определена, но благодаря тому что переменная `found` равна `true`, значение `someUndeclaredVariable` никогда не оценивается и код выводит `"true"`. Если значение `found` изменить на `false`, возникнет ошибка:

```
let found = false;
let result = (found || someUndeclaredVariable);    // ошибка
console.log(result);    // эта строка никогда не выполняется
```

Используя эту схему, можно предотвратить присваивание переменной значения `null` или `undefined`, например:

```
let myObject = preferredObject || backupObject;
```

Здесь переменной `myObject` присваивается одно из двух значений. Переменная `preferredObject` содержит предпочтительное значение, но на тот случай, если оно окажется недоступным, предоставляется также резервная переменная `backupObject`. Если значение `preferredObject` не равно `null`, оно присваивается переменной `myObject`, но если оно равно `null`, переменная получает значение `backupObject`. Этот прием очень часто используется в ECMAScript, и вы еще не раз встретите его в книге.

Мультипликативные операторы

В ECMAScript к группе мультипликативных относятся три оператора: умножение, деление и деление по модулю. Они во многом похожи на свои аналоги в таких языках, как Java, C и Perl, но автоматически преобразуют некоторые нечисловые значения. Если какой-либо из операндов этих операторов не является числом, он за кулисами преобразуется в число с помощью функции приведения типов `Number()`. Это означает, например, что пустая строка интерпретируется как 0, а логическое значение `true` — как 1.

Умножение

Как нетрудно догадаться, оператор умножения (`*`) умножает два числа. Его синтаксис такой же, как в C:

```
let result = 34 * 56;
```

Умножение специальных значений приводит к ряду уникальных режимов работы.

- Если множители — числа, выполняется обычное арифметическое умножение, при этом умножение двух положительных или двух отрицательных значений дает положительный результат, а умножение операндов с разными знаками — отрицательный. Если результат не может быть представлен в ECMAScript, возвращается значение `Infinity` или `-Infinity`.
- Если какой-либо из множителей — значение `NaN`, в результате получается `NaN`.
- Если `Infinity` умножается на 0, получается `NaN`.

- Если `Infinity` умножается на любое конечное число, отличное от 0, в результате получается или `Infinity`, или `-Infinity`, в зависимости от знака второго множителя.
- Если `Infinity` умножается на `Infinity`, получается `Infinity`.
- Если какой-либо из множителей не является числом, он преобразуется в число с помощью функции `Number()`, затем применяются другие правила.

Деление

Оператор деления (`/`) делит первый операнд на второй:

```
let result = 66 / 11;
```

Как и умножение, деление специальных значений имеет ряд особенностей.

- Если операнды — числа, выполняется обычное арифметическое деление, при этом в случае одинаковых знаков операндов получается положительный результат, а в случае разных — отрицательный. Если результат не может быть представлен в ECMAScript, возвращается значение `Infinity` или `-Infinity`.
- Если какой-либо из операндов — значение `NaN`, в результате получается `NaN`.
- Если `Infinity` делится на `Infinity`, получается `NaN`.
- Если 0 делится на 0, получается `NaN`.
- Если ненулевое конечное число делится на 0, в результате получается `Infinity` или `-Infinity`, в зависимости от знака первого операнда.
- Если `Infinity` делится на любое число, в результате получается `Infinity` или `-Infinity`, в зависимости от знака второго операнда.
- Если какой-либо из операндов не является числом, он преобразуется в число с помощью функции `Number()`, затем применяются другие правила.

Деление по модулю

Оператор деления по модулю, или взятия остатка (`%`), используется следующим образом:

```
let result = 26 % 5;    // 1
```

Он также имеет особенности, когда используется со специальными значениями.

- Если операнды — числа, выполняется обычное арифметическое деление и возвращается остаток.
- Если бесконечное число делится на конечное, в результате получается `NaN`.
- Если конечное число делится на 0, получается `NaN`.
- Если `Infinity` делится на `Infinity`, получается `NaN`.
- Если конечное число делится на бесконечное, возвращается делимое.
- Если 0 делится на число, не равное 0, в результате получается 0.

- Если какой-либо из операндов не является числом, он преобразуется в число с помощью функции `Number()`, затем применяются другие правила.

Оператор возведения в степень

Появившийся в ECMAScript 7 `Math.pow()` теперь получает свой собственный оператор `**`, который ведет себя идентично.

```
console.log(Math.pow(3, 2);    // 9
console.log(3 ** 2);           // 9

console.log(Math.pow(16, 0.5); // 4
console.log(16** 0.5);         // 4
```

Более того, оператор также получает собственный оператор присваивания возведения в степень `**=`, который выполняет возведение в степень и последующее присваивание результата:

```
let squared = 3;
squared **= 2;
console.log(squared);    // 9

let sqrt = 16;
sqrt **= 0.5;
console.log(sqrt);       // 4
```

Операторы сложения и вычитания

Сложение и вычитание — одни из простейших математических операций в языках программирования, но в ECMAScript у них есть много нюансов. При сложении и вычитании также выполняется неявное преобразование типов данных, но его правила не так просты, как для умножения и деления.

Сложение

Оператор сложения (`+`) используется обычным образом:

```
let result = 1 + 2;
```

Если оба слагаемых являются числами, он выполняет арифметическое сложение и возвращает результат по указанным правилам.

- Если какое-либо из слагаемых — значение `NaN`, в результате получается `NaN`.
- Если суммируются значения `Infinity` и `Infinity`, получается `Infinity`.
- Если суммируются `-Infinity` и `-Infinity`, получается `-Infinity`.
- Если суммируются `Infinity` и `-Infinity`, получается `NaN`.
- Если суммируются +0 и +0, получается +0.
- Если суммируются –0 и +0, получается +0.
- Если суммируются –0 и –0, получается –0.

Если один из операндов является строкой, применяются другие правила.

- Если оба операнда — строки, вторая строка присоединяется к первой.
- Если только один операнд — строка, другой операнд преобразуется в строку и выполняется конкатенация строк.

Если какой-либо из операндов является объектом, числом или логическим значением, вызывается его метод `toString()` для получения строкового значения, а затем применяются правила для строк. Для значений `undefined` и `null` вызывается функция `String()`, которая возвращает `"undefined"` и `"null"` соответственно.

Рассмотрим следующий пример:

```
let result1 = 5 + 5;      // два числа
console.log(result1);     // 10
let result2 = 5 + "5";    // число и строка
console.log(result2);     // "55"
```

Этот код поясняет различие между двумя режимами оператора сложения. В обычной ситуации 5 + 5 равно 10 (числовое значение), что подтверждают первые две строки кода. Однако если изменить один из операндов на строку `"5"`, результат изменяется на `"55"` (примитивное строковое значение), потому что другой операнд также преобразуется в строку `"5"`.

Невнимание к типам слагаемых часто приводит в ECMAScript к ошибкам, например:

```
let num1 = 5;
let num2 = 10;
let message = "The sum of 5 and 10 is " + num1 + num2;
console.log(message);  // выводится сообщение о том, что сумма 5 и 10 равна 510
```

В этом примере переменной `message` присваивается строка, включающая результат двух операций сложения. Предполагается, что ею должна быть строка `"The sum of 5 and 10 is 15"` (сумма 5 и 10 равна 15), но на экран выводится сообщение `"The sum of 5 and 10 is 510"` (сумма 5 и 10 равна 510). Это происходит из-за того, что каждое сложение выполняется отдельно. В первый раз строка складывается с числом 5, что дает в результате строку. Далее к ней добавляется число 10, и опять получается строка. Чтобы сложить числа и присоединить результат к строке, просто добавьте скобки:

```
let num1 = 5;
let num2 = 10;
let message = "The sum of 5 and 10 is " + (num1 + num2);
console.log (message);    // Сумма 5 и 10 равна 15
```

Теперь интерпретатор сначала вычисляет сумму числовых переменных в скобках, а затем присоединяет ее к строке. Итоговая строка: `"The sum of 5 and 10 is 15"` (сумма 5 и 10 равна 15).

Вычитание

Оператор вычитания (`-`) также используется весьма часто. Вот пример с ним:

```
let result = 2 – 1;
```

Как и при сложении, при вычитании в ECMAScript действуют специальные правила преобразования типов.

- Если оба операнда — числа, выполняется арифметическое вычитание и возвращается результат.
- Если какой-либо из операндов — `NaN`, в результате получается `NaN`.
- Если `Infinity` вычитается из `Infinity`, получается `NaN`.
- Если `-Infinity` вычитается из `-Infinity`, получается `NaN`.
- Если `-Infinity` вычитается из `Infinity`, получается `Infinity`.
- Если `Infinity` вычитается из `-Infinity`, получается `-Infinity`.
- Если +0 вычитается из +0, получается +0.
- Если –0 вычитается из +0, получается –0.
- Если –0 вычитается из –0, получается +0.
- Если какой-либо из операндов — строка, логическое значение, `null` или `undefined`, он преобразуется в число с помощью функции `Number()`, а затем выполняется арифметическое вычитание по описанным правилам. Если операнд преобразуется в `NaN`, результат вычитания — `NaN`.
- Если какой-либо из операндов является объектом, вызывается его метод `valueOf()` для получения числового значения. Если это значение `NaN`, результат вычитания — `NaN`. Если для объекта не определен метод `valueOf()`, вызывается метод `toString()`, а полученная строка преобразуется в число.

Вот несколько примеров использования этих правил:

```
let result1 = 5 – true;    // 4, потому что true преобразуется в 1
let result2 = NaN – 1;     // NaN
let result3 = 5 – 3;       // 2
let result4 = 5 – "";      // 5, потому что "" преобразуется в 0
let result5 = 5 – "2";     // 3, потому что "2" преобразуется в 2
let result6 = 5 – null;    // 5, потому что null преобразуется в 0
```

Операторы отношений

Операторы отношений «меньше» (`<`), «больше» (`>`), «меньше или равно» (`<=`) и «больше или равно» (`>=`) сравнивают значения так же, как в школьной математике. Каждый из них возвращает логическое значение, например:

```
let result1 = 5 > 3;    // true
let result2 = 5 < 3;    // false
```

Как и в других ECMAScript-операциях, при сравнении некоторые типы данных преобразуются по особым правилам.

- Если операнды — числа, выполняется числовое сравнение.
- Если операнды — строки, сравниваются коды знаков в одинаковых позициях.
- Если один операнд — число, другой операнд преобразуется в число и выполняется числовое сравнение.

- Если операнд — объект, вызывается метод `valueOf()` и его результат сравнивается с другим операндом по предыдущим правилам. Если метод `valueOf()` недоступен, вызывается метод `toString()` и полученное значение сравнивается по предыдущим правилам.
- Если операнд — логическое значение, он преобразуется в число и выполняется сравнение.

Если оператор отношения применяется к двум строкам, происходит кое-что интересное. Обычно предполагают, что «меньше» означает «ближе к началу алфавита», а «больше» — «ближе к концу алфавита», но это не так. Для строк каждый числовой код знака первой строки сравнивается с соответствующим кодом знака второй строки, после чего возвращается логическое значение. Проблема в том, что коды прописных букв меньше, чем строчных, из-за чего получается следующее:

```
let result = "Brick" < "alphabet";    // true
```

В этом примере строка `"Brick"` считается меньше, чем строка `"alphabet"`, потому что буква *B* имеет код 66, а буква *a* — 97. Чтобы сравнить строки по алфавиту, необходимо перед сравнением преобразовать оба операнда в один регистр (неважно, верхний или нижний):

```
let result = "Brick".toLowerCase() < "alphabet".toLowerCase();  // false
```

После преобразования в нижний регистр слово `"alphabet"` правильно распознается как находящееся ближе к началу алфавита, чем `"Brick"`.

Другая непростая ситуация возникает при сравнении чисел в строковой форме, например:

```
let result = "23" < "3";  // true
```

Здесь утверждается, что `"23"` меньше, чем `"3"`. Почему? Потому, что строковые операнды сравниваются по кодам знаков (код знака `"2"` — 50; код знака `"3"` — 51). Если один из операндов изменить на число, результат больше не удивляет:

```
let result = "23" < 3;    // false
```

В этом примере строка `"23"` перед сравнением преобразуется в число 23, что обеспечивает правильный результат. Когда строка сравнивается с числом, она преобразуется в число и выполняется числовое сравнение. Это хорошо работает в ситуациях вроде предыдущей, но что, если строку невозможно преобразовать в число? Взгляните на следующий пример:

```
let result = "a" < 3;    // false, потому что "a" преобразуется в NaN
```

Буква `"a"` не может быть осмысленно преобразована в число, поэтому она становится значением `NaN`. Как правило, в результате любой операции отношения с операндом `NaN` получается `false`, что имеет интересные следствия:

```
let result1 = NaN < 3;    // false
let result2 = NaN >=3;    // false
```

Обычно считается, что если одно значение не меньше другого, то оно должно быть больше или равно ему, но для значения `NaN` оба сравнения возвращают `false`.

Операторы эквивалентности

Определение эквивалентности двух переменных — одна из самых важных операций в программировании. Это довольно просто при работе со строками, числами и логическими значениями, но задача усложняется, когда дело доходит до объектов. Первоначально операторы равенства и неравенства в ECMAScript преобразовывали операнды перед сравнением в похожие типы, но затем был поставлен вопрос, а правильно ли это. В итоге в ECMAScript определили два набора операций: *равенство* (equal) и *неравенство* (not equal), которые преобразуют данные перед сравнением, и *строгое равенство* (identically equal) и *строгое неравенство* (not identically equal), которые выполняют сравнение без преобразования.

Равенство и неравенство

Оператор равенства (`==`) возвращает `true`, если операнды равны. Оператор неравенства (`!=`) возвращает `true`, если операнды не равны. Для определения равенства операндов оба оператора при необходимости преобразуют их типы, что часто называют *приведением типов* (type coercion).

При преобразовании типов для операторов равенства и неравенства применяются свои правила.

- Если операнд — логическое значение, перед проверкой на равенство оно преобразуется в число. Значение `false` преобразуется в 0, а `true` — в 1.
- Если операнды — строка и число, перед сравнением предпринимается попытка преобразовать строку в число.
- Если один из операндов — объект, для него вызывается метод `valueOf()`, чтобы получить примитивное значение, которое затем сравнивается по предыдущим правилам.

При сравнении применяются свои правила.

- Значения `null` и `undefined` равны.
- Значения `null` и `undefined` не преобразуются для сравнения ни в какие другие значения.
- Если одним из операндов является значение `NaN`, оператор равенства возвращает `false`, а оператор неравенства — `true`. Даже если оба операнда — значения `NaN`, оператор равенства возвращает `false`, потому что по правилам `NaN` не равно `NaN`.
- Если оба операнда — объекты, они сравниваются, чтобы выяснить, один ли это объект. Если да, возвращается `true`, иначе — `false`.

Некоторые специальные сравнения и результаты приведены в таблице.

ВЫРАЖЕНИЕ	ЗНАЧЕНИЕ
`null == undefined`	`true`
`"NaN" == NaN`	`false`
`5 == NaN`	`false`
`NaN == NaN`	`false`
`NaN != NaN`	`true`
`false == 0`	`true`
`true == 1`	`true`
`true == 2`	`false`
`undefined == 0`	`false`
`null == 0`	`false`
`"5" == 5`	`true`

Строгое равенство и строгое неравенство

Операторы строгих равенства и неравенства делают то же самое, что и обычные операторы равенства и неравенства, но не преобразуют операнды перед сравнением. Оператор строгого равенства (`===`) возвращает `true`, только если операнды равны без преобразования, например:

```
let result1 = ("55" == 55);    // true — равно благодаря преобразованию
let result2 = ("55" === 55);   // false — не равно из-за разных типов данных
```

В первом случае здесь с помощью оператора равенства сравниваются строка `"55"` и число 55, что дает в результате `true`. Как уже отмечалось, это происходит потому, что строка `"55"` преобразуется в число 55, которое затем сравнивается с другим числом 55. Во втором случае строка и число сравниваются без преобразования, и конечно, возвращается `false`, потому что строка не равна числу.

Оператор строгого неравенства (`!==`) возвращает `true`, если без преобразования операнды не равны, например:

```
let result1 = ("55" != 55);    // false — равно благодаря преобразованию
let result2 = ("55" !== 55);   // true — не равно из-за разных типов данных
```

Здесь в первом сравнении используется оператор равенства, который преобразует строку `"55"` в число 55. Оно равно второму операнду, поэтому выражение получает значение `false`. Во втором сравнении используется оператор строгого неравенства, который возвращает `true`, потому что строка `"55"` отличается от числа 55.

Запомните, что выражение `null == undefined` истинно, потому что значения равны, но `null === undefined` ложно, потому что типы этих значений разные.

ПРИМЕЧАНИЕ Из-за проблем преобразования типов, возникающих при работе с операторами равенства и неравенства, рекомендуется использовать вместо них операторы строгих равенства и неравенства. Это помогает поддерживать целостность типов данных в коде.

Условный оператор

Условный оператор используется в ECMAScript в самых разных ситуациях и работает так же, как и в Java:

```
переменная = логическое_выражение ? значение_если_true : значение_если_false
```

С помощью этого оператора можно присваивать переменной разные значения в зависимости от логического выражения. Если оно истинно, переменной присваивается первое значение, иначе — второе:

```
let max = (num1 > num2) ? num1 : num2
```

В этом примере переменной `max` присваивается большее число. Если `num1` больше, чем `num2`, то переменная `max` получает значение `num1`. Если же выражение в скобках ложно (то есть `num1` меньше или равно `num2`), она становится равной `num2`.

Операторы присваивания

Простое присваивание выполняется с помощью знака равенства (`=`), при этом значение справа от него просто присваивается переменной слева, например:

```
let num = 10;
```

Составное присваивание выполняется с помощью одного из арифметических операторов или операторов сдвига, за которым следует знак равенства. Это сокращает код в некоторых популярных сценариях, например:

```
let num = 10;
num = num + 10;
```

Этот код эквивалентен следующему:

```
let num = 10;
num += 10;
```

Составные операторы присваивания есть для всех основных математических операций и для нескольких других. Вот они:

- умножение с присваиванием (`*=`);
- деление с присваиванием (`/=`);
- деление по модулю с присваиванием (`%=`);

- сложение с присваиванием (+=);
- вычитание с присваиванием (-=);
- сдвиг влево с присваиванием (<<=);
- сдвиг вправо с сохранением знака и присваиванием (>>=);
- сдвиг вправо с заполнением нулями и присваиванием (>>>=).

Эти операторы только сокращают объем кода, но не увеличивают его быстродействие.

Оператор «запятая»

Оператор «запятая» позволяет выполнить в одной инструкции более одной операции:

```
let num1 = 1, num2 = 2, num3 = 3;
```

Чаще всего он используется в объявлениях переменных, но с его помощью можно также присваивать значения. В этом случае он всегда возвращает последний элемент выражения, например:

```
let num = {5, 1, 4, 8, 0};  // num равно 0
```

В этом примере переменная `num` получает значение 0, потому что это последний элемент выражения. Так запятые используются редко, однако полезно знать, что это возможно.

ИНСТРУКЦИИ

ECMA-262 включает несколько инструкций, называемых также *управляющими инструкциями* (flow-control statements), которые составляют основную часть синтаксиса ECMAScript и обычно решают специфическую задачу с помощью одного или нескольких ключевых слов. Сложность инструкций варьируется в широких пределах — от тривиального выхода из функции до блоков многократно выполняемых команд.

Инструкция if

Инструкция `if` часто используется почти во всех языках программирования. Она имеет следующий синтаксис:

```
if (условие) инструкция1 else инструкция2
```

Условие может быть любым выражением. Оно даже может не относиться к логическому типу, потому что ECMAScript автоматически преобразует результат выражения в логическое значение, вызывая для него функцию `Boolean()`. Если условие

эквивалентно true, выполняется инструкция 1, в противном случае — инструкция 2. Любая из инструкций может быть одной строкой или блоком кода (группой строк в фигурных скобках), например:

```
if (i > 25)
    console.log("Greater than 25.");             // однострочная инструкция
else {
    console.log("Less than or equal to 25.");    // блочная инструкция
}
```

Рекомендуется всегда использовать блочные инструкции, даже если нужно выполнить всего одну строку кода. Это ясно показывает, что должно быть выполнено в каждом случае.

Инструкции if можно сцеплять друг с другом:

```
if (условие1) инструкция1 else if (условие2) инструкция2 else инструкция3
```

Вот пример:

```
if (i > 25) {
    console.log("Greater than 25.");
} else if (i < 0) {
    console.log("Less than 0.");
} else {
    console.log("Between 0 and 25, inclusive.");
}
```

Инструкция do-while

Инструкция do-while создает цикл с постусловием, в котором условие выхода из цикла проверяется только после выполнения кода внутри него. Тело цикла выполняется как минимум один раз перед оценкой выражения. Вот синтаксис цикла:

```
do {
    инструкция
} while (выражение);
```

А вот пример его использования:

```
let i = 0;
do {
    i +=2;
} while (i < 10);
```

Этот цикл продолжается, пока переменная i меньше 10. Она равна 0 в начале цикла и увеличивается на 2 на каждой итерации.

ПРИМЕЧАНИЕ Циклы с постусловием чаще всего используются, если тело цикла должно быть выполнено хотя бы один раз.

Инструкция while

Инструкция `while` создает цикл с предусловием. Это означает, что условие выхода из цикла проверяется перед выполнением кода внутри него. Возможно, что тело цикла не будет выполнено ни разу. Синтаксис этого цикла таков:

```
while(выражение) инструкция
```

А вот пример его использования:

```
let i = 0;
while (i < 10) {
    i += 2;
}
```

Переменная `i` равна 0 перед началом цикла и увеличивается на 2 на каждой итерации. Пока она меньше 10, цикл продолжается.

Инструкция for

Инструкция `for` — это вариант цикла с предусловием, позволяющий инициализировать переменную перед началом цикла и указать код, выполняемый после цикла. Она имеет следующий синтаксис:

```
for (инициализация; выражение; выражение после цикла) инструкция
```

Пример цикла `for`:

```
let count = 10;
for (let i=0; i < count; i++) {
    console.log(i);
}
```

В этом фрагменте определяется переменная `i` с нулевым значением. Цикл `for` начинается, только если результатом условного выражения (`i < count`) является значение `true`, то есть тело цикла может быть не выполнено ни разу. Если тело цикла выполняется, вслед за ним в выражении после цикла увеличивается значение `i`. Этот цикл `for` эквивалентен следующему:

```
let count = 10;
let i = 0;
while (i < count) {
    console.log(i);
    i++;
}
```

С помощью инструкции `for` нельзя сделать ничего такого, что невозможно было бы выполнить в цикле `while`, она просто группирует связанный с циклом код в одном месте.

Важно отметить, что нет необходимости использовать ключевое слово объявления переменной внутри инициализации цикла `for`. Однако в подавляющем большинстве

случаев переменная итератора оказывается бесполезной после завершения цикла. В этих случаях наиболее чистой реализацией является использование объявления `let` внутри инициализации цикла для объявления переменной итератора, поскольку ее область действия будет ограничена только самим циклом.

Инициализация, управляющее выражение и выражение после цикла не обязательны. Если опустить все три части, получится бесконечный цикл:

```
for (;;) {          // бесконечный цикл
    doSomething();
}
```

Добавив только управляющее выражение, можно преобразовать цикл `for` в цикл `while`:

```
let count = 10;
let i = 0;
for (; i < count; ) {
    console.log(i);
    i++;
}
```

Благодаря такой гибкости инструкция `for` в ECMAScript является одной из наиболее востребованных.

Инструкция for-in

Инструкция `for-in` используется для перебора несимвольных ключевых свойств объектов и имеет следующий синтаксис:

```
for (свойство in выражение) инструкция
```

Пример с инструкцией `for-in`:

```
for (const propName in window) {
     document.write(propName);
}
```

Здесь инструкция `for-in` используется для вывода из объектной модели браузера на экран всех свойств объекта `windows`. При каждой итерации цикла переменной `propName` присваивается имя очередного свойства. Это продолжается, пока не будут перебраны все доступные свойства. Как и в цикле `for`, оператор `const` в управляющем выражении здесь не обязателен, но рекомендуется использовать его, чтобы переменная была локальной и не была изменена позже.

Свойства объектов в ECMAScript не упорядочены, поэтому порядок возврата их имен в цикле `for-in` предсказать нельзя. Все перечисленные свойства будут возвращены, но порядок их вывода может зависеть от браузера.

Если переменная, представляющая перебираемый объект, равна `null` или `undefined`, инструкция `for-in` просто пропустит выполнение кода в теле цикла.

Инструкция for-of

Инструкция `for-of` — строго итеративная. Она используется для циклического прохождения элементов в итерируемом объекте и имеет следующий синтаксис:

```
for (свойство of выражение) инструкция
```

Пример с инструкцией `for-of`:

```
for (const el in [2,4,6,8) {
    document.write(el);
}
```

Здесь инструкция `for-of` используется для отображения всех элементов внутри четырехэлементного массива. Это продолжается до тех пор, пока цикл не пройдет по каждому элементу в массиве. Как и в инструкции `for`, оператор `const` в инструкции управления не является обязательным, но рекомендуется для обеспечения использования локальной переменной, которая не будет изменена.

Цикл `for-of` будет выполнять итерацию в том порядке, в котором итерируемый метод создает значения через метод `next()`. Это подробно рассматривается в главе 7 «Итераторы и генераторы».

Обратите внимание, что инструкция `for-of` вернет ошибку, если объект, который она пытается перебрать, не поддерживает итерацию.

> **ПРИМЕЧАНИЕ** В ES2018 инструкция `for-of` расширена как цикл `for-await-of` для поддержки асинхронных итераций, которые производят промисы. Подробнее об этом в приложении А.

Метки инструкций

Инструкции можно помечать, чтобы затем ссылаться на них. Синтаксис меток таков:

```
метка: инструкция
```

Вот пример кода с меткой:

```
start: for (let i=0; i < count; i++) {
    console.log(i);
}
```

В этом примере на метку `start` можно ссылаться позднее в инструкции `break` или `continue`. Помеченные инструкции обычно используются с вложенными циклами.

Инструкции break и continue

Инструкции `break` и `continue` обеспечивают более точный контроль над выполнением кода в цикле. Инструкция `break` немедленно завершает цикл, передавая управление

следующей инструкции после цикла, а `continue` завершает только текущую итерацию цикла, начиная новую. Рассмотрим пример:

```
let num = 0;

for (let i=1; i < 10; i++) {
    if (i % 5 == 0) {
        break;
    }
    num++;
}

console.log(num);    // 4
```

В цикле `for` переменная `i` увеличивается с 1 до 10. В теле цикла инструкция `if` с помощью оператора деления по модулю проверяет, делится ли значение `i` без остатка на 5. Если да, выполняется инструкция `break` и цикл завершается. Переменная `num` подсчитывает количество итераций цикла. После `break` в консоли выводится сообщение со значением 4. Цикл выполняется 4 раза потому, что когда `i` равно 5, инструкция `break` завершает цикл до очередного увеличения значения `num`. Если изменить `break` на `continue`, получится другой результат:

```
let num = 0;

for (let i=1; i < 10; i++) {
    if (i % 5 == 0) {
        continue;
    }
    num++;
}

console.log(num);    // 8
```

В этот раз итоговое количество итераций равно 8. Когда `i` достигает значения 5, итерация цикла завершается до увеличения переменной `num`, но цикл продолжается со следующей итерации со значением `i`, равным 6. Затем цикл выполняется до естественного завершения при значении `i`, равном 10. Окончательное значение `num` равно 8, а не 9, потому что одна операция инкремента пропускается из-за инструкции `continue`.

И `break`, и `continue` можно использовать вместе с помеченными инструкциями для возврата к конкретному месту в коде. Обычно это делается во вложенных циклах, например:

```
let num = 0;

outermost:
for (let i=0; i < 10; i++) {
    for (let j=0; j < 10; j++) {
        if (i == 5 && j == 5) {
            break outermost;
        }
```

```
        num++;
    }
}

console.log(num);    // 55
```

В этом примере для первой инструкции `for` добавлена метка `outermost`. Каждый цикл включает 10 итераций, то есть инструкция `num++` предположительно должна быть выполнена 100 раз, после чего переменная `num` должна быть равна 100. Инструкция `break` получает здесь в качестве аргумента метку для перехода, вследствие чего она завершает не только внутренний цикл `for` (с переменной `j`), но и внешний (с переменной `i`). Окончательное значение `num` равно 55, потому что циклы завершаются, когда `i` и `j` равны 5. Инструкция `continue` используется аналогично:

```
let num = 0;

outermost:
for (let i=0; i < 10; i++) {
    for (let j=0; j < 10; j++) {
        if (i == 5 && j == 5) {
             continue outermost;
        }
        num++;
    }
}

console.log(num);    //95
```

В этом случае инструкция `continue` завершает выполнение внутреннего цикла, начиная новую итерацию внешнего. Она выполняется, когда `j` равно 5, то есть пропускаются пять итераций внутреннего цикла, из-за чего `num` в итоге имеет значение 95.

Использование помеченных инструкций вместе с `break` и `continue` — очень эффективный прием, но не злоупотребляйте им, иначе будет трудно отлаживать код. Всегда назначайте меткам описательные имена и не создавайте циклов с большим количеством уровней вложенности.

Инструкция with

Инструкция `with` делает областью видимости кода конкретный объект. Вот ее синтаксис:

```
with (выражение) инструкция;
```

Инструкция `with` была создана ради удобства для тех случаев, когда имя одного объекта приходится вводить снова и снова, например:

```
let qs = location.search.substring(1);
let hostName = location.hostname;
let url = location.href;
```

Чтобы не указывать в каждой строке объект `location`, этот код можно переписать следующим образом:

```
with(location) {
    let qs = search.substring(1);
    let hostName = hostname;
    let url = href;
}
```

В этой версии кода, где инструкция `with` используется с объектом `location`, каждая переменная внутри блока сначала считается локальной. Если выясняется, что она не является локальной, выполняется поиск свойства с тем же именем в объекте `location`. Если оно обнаруживается, переменная интерпретируется как свойство объекта `location`.

В строгом режиме инструкция `with` не поддерживается. Попытка использовать ее приведет к синтаксической ошибке.

> **ПРИМЕЧАНИЕ** Использование инструкции with в окончательном коде считается плохой практикой, потому что это снижает быстродействие и затрудняет отладку кода.

Инструкция switch

С `if` тесно связана управляющая инструкция `switch`, также заимствованная из других языков. Синтаксис `switch` в ECMAScript напоминает аналоги в других C-подобных языках программирования:

```
switch (выражение) {
    case значение1: инструкция
        break;
    case значение2: инструкция
        break;
    case значение3: инструкция
        break;
    case значение4: инструкция
        break;
    default: инструкция
}
```

Если выражение равно конкретному значению в блоке `switch`, выполняется соответствующая инструкция. Ключевое слово `break` вызывает выход из блока `switch`, в противном случае просто выполнялась бы следующая инструкция в списке. Ключевое слово `default` указывает код, который выполняется, если выражение не равно ни одному значению (по сути, оно аналогично `else`).

Инструкция `switch` избавляет от необходимости писать код вроде этого:

```
if (i == 25) {
    console.log("25");
} else if (i == 35) {
    console.log("35");
```

```
} else if (i == 45) {
    console.log("45");
} else {
    console.log("Other");
}
```

Этот фрагмент эквивалентен следующему:

```
switch (i) {
    case 25:
        console.log("25");
        break;
    case 35:
        console.log("35");
        break;
    case 45:
        console.log("45");
        break;
    default:
        console.log("Other");
}
```

Лучше всегда добавлять слово `break` в конце каждого раздела `case`, чтобы управление не «проваливалось» в следующий раздел. Если же именно это и нужно, добавьте комментарий, поясняющий, что инструкция `break` опущена умышленно:

```
switch (i) {
    case 25:
         /* переход к следующему разделу */
    case 35:
        console.log("25 или 35");
        break;
    case 45:
        console.log("45");
        break;
    default:
        console.log("Other");
}
```

Хотя инструкция `switch` позаимствована из других языков, в ECMAScript она имеет некоторые особенности. Во-первых, она работает со всеми типами данных (во многих языках только с числами), так что ее можно использовать со строками и даже с объектами. Во-вторых, значения для сравнения с выражением могут быть не только константами, но и переменными, и даже выражениями. Рассмотрим пример:

```
switch ("hello world!") {
    case "hello" + " world!":
        console.log("Greeting was found.");
        break;
    case "goodbye":
        console.log("Closing was found.");
        break;
    default:
        console.log("Unexpected message was found.");
}
```

В этом примере в инструкции `switch` используется строковое значение, которое в первом же разделе сравнивается с выражением — результатом конкатенации строк. Поскольку объединенная строка равна аргументу `switch`, выводится сообщение `"Greeting was found."` (это приветствие). Возможность использовать выражения как селекторы разделов `case` позволяет писать такой код:

```
let num = 25;
switch (true) {
    case num < 0:
        console.log("Less than 0.");
        break;
    case num >=0 && num <= 10:
        console.log("Between 0 and 10.");
        break;
    case num > 10 && num <= 20:
        console.log("Between 10 and 20.");
        break;
    default:
        console.log("More than 20.");
}
```

Здесь переменная `num` определена вне инструкции `switch`. В `switch` передается значение `true`, которое сравнивается с логическими значениями — результатами вычисления условий `case`. Каждое значение оценивается по порядку, пока не будет обнаружено совпадение или пока не встретится инструкция `default` (как в этом примере).

> **ПРИМЕЧАНИЕ** В инструкции `switch` значения сравниваются с помощью оператора строгого равенства, то есть без преобразования типов (например, строка «*10*» не равна числу 10).

ФУНКЦИИ

Функции — это базовые элементы любого языка, с помощью которых можно инкапсулировать группы инструкций и в любое время выполнять их в других местах кода. В ECMAScript для объявления функции используется ключевое слово `function`, за которым следуют аргументы и тело функции.

> **ПРИМЕЧАНИЕ** Подробное описание функций можно найти в главе 10 «Функции».

Вот базовый синтаксис функции:

```
function имяФункции(арг0, арг1, ..., аргN) {
    инструкции
}
```

Пример функции:

```
function sayHi(name, message) {
    console.log("Hello " + name + ", " + message);
}
```

После определения функции ее можно вызывать по имени, указывая в скобках после него аргументы, разделенные запятыми. Например, функция `sayHi()` вызывается так:

```
sayHi("Nicholas", "how are you today?");
```

В результате в окне оповещения будет выведена строка `"Hello Nicholas, how are you today?"` (Здравствуй, Николас, как дела?), составленная из приветствия и аргументов `name` и `message`.

Для функций в ECMAScript можно не указывать, возвращают ли они значения. Из любой функции можно возвратить значение в любое время с помощью инструкции `return`, например:

```
function sum(num1, num2) {
    return num1 + num2;
}
```

Эта функция суммирует два значения и возвращает результат. Обратите внимание, что кроме инструкции `return` нет никакого специального объявления о том, что функция возвращает значение. Эту функцию можно вызвать следующим образом:

```
Const result = sum(5, 10);
```

Помните, что при достижении инструкции `return` функция сразу завершается. Никакой код после `return` никогда не выполняется, например:

```
function sum(num1, num2) {
    return num1 + num2;
    console.log("Hello world!");  // никогда не выполняется
}
```

В этом примере сообщение в консоли никогда не появится, потому что вызов `console.log` расположен после `return`.

Функция может содержать несколько инструкций `return`:

```
function diff(num1, num2) {
    if (num1 < num2) {
        return num2 - num1;
    } else {
        return num1 - num2;
    }
}
```

Эта функция находит разность двух чисел. Если первое число меньше, оно вычитается из второго, в противном случае второе число вычитается из первого. Каждая ветвь кода содержит отдельную инструкцию `return` с правильным выражением.

Инструкцию `return` можно также использовать без указания возвращаемого значения. В этом случае функция немедленно завершается, возвращая значение `undefined`. Это обычно делается для преждевременного завершения функций, которые не возвращают значение. В следующем примере сообщение в консоли не выводится:

```
function sayHi(name, message) {
    return;
    console.log("Hello " + name + ", " + message);  // никогда не вызывается
}
```

> **ПРИМЕЧАНИЕ** В соответствии с наилучшими методиками рекомендуется либо возвращать значение из функции всегда, либо не возвращать никогда. Функцию, которая возвращает значение иногда, труднее понять, особенно на этапе отладки.

В строгом режиме на функции налагаются следующие ограничения:

- функции не могут называться `eval` или `arguments`;
- именованные параметры не могут называться `eval` или `arguments`;
- именованные параметры не могут иметь одинаковые имена.

Если какое-либо из этих условий нарушено, возникает синтаксическая ошибка и код не выполняется.

ИТОГИ

Фундаментальные возможности JavaScript определены в ECMA-262 как псевдоязык с именем ECMAScript. Он содержит все основные синтаксические элементы, операторы, типы данных и объекты, необходимые для выполнения базовых вычислительных задач, но не предоставляет средств ввода или вывода данных. Знание ECMAScript и его особенностей важно для полного понимания реализаций JavaScript в веб-браузерах. Перечислим некоторые основные элементы этого языка.

- К базовым ECMAScript-типам данных относятся неопределенный (undefined), нулевой (null), логический (boolean), числовой (number), строковый (string) и символьный (symbol) типы.
- В отличие от других языков, в ECMAScript нет отдельных типов данных для целых чисел и чисел с плавающей точкой. Все числа представляются числовым типом.
- Также имеется сложный тип данных Object, который является базовым для всех объектов в языке.
- В строгом режиме налагаются ограничения на некоторые возможности языка, часто приводящие к ошибкам.

- ECMAScript содержит много базовых операторов, доступных в C и других C-подобных языках, в том числе арифметические и логические операторы, операторы отношений, эквивалентности и присваивания.
- В языке доступны управляющие инструкции, типичные для многих других языков, такие как `if`, `for` и `switch`.

Функции в ECMAScript работают иначе, чем функции в других языках. Вот некоторые их особенности.

- Указывать возвращаемое значение функции не требуется, поскольку любая функция может возвратить любое значение в любое время.
- Функции без возвращаемого значения на самом деле возвращают специальное значение `undefined`.

4

Переменные, область видимости и память

- Примитивные и ссылочные значения
- Контекст выполнения
- Сборка мусора

Загрузки с сайта wrox.com для этой главы

Обратите внимание, что все примеры кода для этой главы доступны на веб-сайте книги по ссылке www.wrox.com/go/projavascript4e на вкладке Downloads.

Переменные в спецификации ECMA-262 во многом отличаются от переменных в других языках. Будучи слабо типизированными, они просто определяют имена для конкретных значений в конкретный момент времени. Поскольку никакие правила не определяют тип данных переменной, ее значение и тип со временем могут меняться. Кроме этой интересной, полезной и иногда спорной особенности у переменных есть много других нетривиальных аспектов.

ПРИМИТИВНЫЕ И ССЫЛОЧНЫЕ ЗНАЧЕНИЯ

ECMAScript-переменные могут содержать значения двух видов: примитивные и ссылочные. *Примитивные значения* (primitive values) — это просто атомарные элементы данных, в то время как *ссылочные значения* (reference values) — это объекты, которые могут состоять из нескольких значений.

Когда значение присваивается переменной, интерпретатор JavaScript должен определить, примитивное оно или ссылочное. Шесть примитивных типов были

рассмотрены в предыдущей главе: неопределенный (undefined), нулевой (null), логический (boolean), числовой (number), строковый (string) и символьный (symbol). Доступ к переменным этих типов осуществляется *по значению* (by value), то есть вы работаете с фактическим значением, хранящимся в переменной.

Ссылочные значения — это объекты, хранящиеся в памяти. В отличие от других языков, в JavaScript невозможен прямой доступ к памяти, в том числе занимаемой объектами. Вместо этого при выполнении каких-либо действий над объектом вы на самом деле работаете не с самим объектом, а со *ссылкой* (reference) на него. Говорят, что доступ к таким значениям осуществляется *по ссылке*.

> **ПРИМЕЧАНИЕ** Во многих языках строки представляются объектами и соответственно считаются ссылочными типами. ECMAScript не следует этой традиции.

Динамические свойства

Примитивные и ссылочные значения определяются схожим образом: вы создаете переменную и присваиваете ей значение. Однако действия, которые можно выполнять со значениями переменных, различаются. При работе со ссылочными значениями можно в любое время добавлять, изменять и удалять их свойства и методы, например:

```
let person = new Object();
person.name = "Nicholas";
console.log(person.name);    // "Nicholas"
```

Здесь мы создаем объект и сохраняем его в переменной `person`. Далее к объекту добавляется свойство `name`, которому присваивается строковое значение `"Nicholas"`. Новое свойство можно использовать, пока объект не будет уничтожен или пока свойство не будет явно удалено.

К примитивным значениям добавить свойства нельзя, хотя это и не является ошибкой:

```
let name = "Nicholas";
name.age = 27;
console.log(name.age);    // undefined
```

Здесь для строки `name` определяется свойство `age`, которому затем присваивается значение 27, но уже в третьей инструкции свойство недоступно. Динамически добавленные свойства остаются доступны только у ссылочных значений.

Обратите внимание, что создание экземпляра примитивного типа может быть выполнено с использованием только примитивной литеральной формы. Если бы вы использовали ключевое слово `new`, JavaScript создал бы тип Object, но такой, который ведет себя как примитив. Вот пример, чтобы научиться их различать:

```
let name1 = "Nicholas";
let name2 = new String("Matt");
```

```
name1.age = 27;
name2.age = 26;
console.log(name1.age); // undefined
console.log(name2.age); // 26
console.log(typeof name1); // string
console.log(typeof name2); // object
```

Копирование значений

Примитивные и ссылочные значения не только хранятся, но и копируются поразному. Когда одна переменная с примитивным значением присваивается другой, создается копия значения, хранящегося в объекте переменных, а затем она записывается по адресу новой переменной, например:

```
let num1 = 5;
let num2 = num1;
```

Здесь `num1` содержит значение 5. Когда переменная `num2` инициализируется значением `num1`, она также получает значение 5. Она никак не связана с `num1`, потому что содержит копию значения. Затем эти переменные можно использовать по отдельности без побочных эффектов (рис. 4.1).

Рис. 4.1

Когда ссылочное значение одной переменной присваивается другой, значение в объекте переменных также копируется в расположение новой переменной, но в этот раз оно является указателем на объект в куче. После копирования обе переменные указывают на один объект, поэтому изменения одной из них отражаются на другой:

```
let obj1 = new Object();
let obj2 = obj1;
obj1.name = "Nicholas";
console.log(obj2.name);    // "Nicholas"
```

В этом примере переменной `obj1` присваивается новый объект, после чего значение переменной копируется в `obj2`. Теперь обе переменные указывают на один объект, и когда для `obj1` задается свойство `name`, оно становится доступным через `obj2`. На рис. 4.2 показаны отношения между переменными в объекте переменных и объектом в куче.

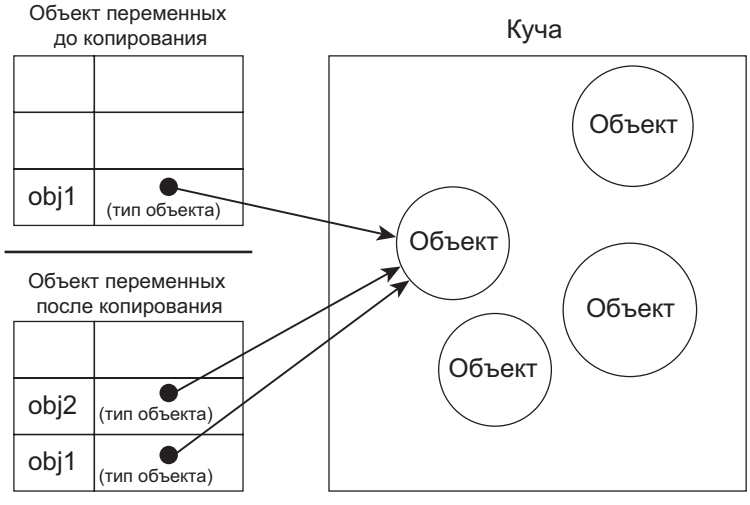

Рис. 4.2

Передача аргументов

Все аргументы функций в ECMAScript передаются по значению. Это означает, что значения извне функции копируются в аргументы внутри функции так же, как копируются значения переменных: примитивные — по одному сценарию, ссылочные — по другому. В то время как доступ к переменным возможен и по значению, и по ссылке, многие разработчики плохо понимают, что аргументы передаются только по значению.

Когда аргумент передается по значению, его значение просто копируется в локальную переменную (которой в ECMAScript соответствуют именованный аргумент и ячейка объекта `arguments`). При передаче аргумента по ссылке в локальной переменной сохраняется расположение его значения в памяти. Это означает, что изменения локальной переменной отражаются вне функции (в ECMAScript это невозможно). Рассмотрим следующий пример:

```
function addTen(num) {
    num +=10;
    return num;
}

let count = 20;
let result = addTen(count);
console.log(count);     // 20 — без изменений
console.log(result);    // 30
```

Здесь определяется функция `addTen()` с аргументом `num`, который, по сути, является локальной переменной. При вызове функции ей передается переменная `count` со значением 20, которая копируется в аргумент `num` для использования в функции `addTen()`. В функции к значению `num` прибавляется 10, но это не изменяет значение `count` вне функции. Аргумент `num` и переменная `count` не знают друг о друге, они просто имеют одинаковые значения. Если бы значение `num` было передано по ссылке, переменная `count` также стала бы равной 30 согласно изменению внутри функции. Все это очевидно при работе с примитивными значениями вроде чисел, но не с объектами. Взгляните на этот пример:

```
function setName(obj) {
    obj.name = "Nicholas";
}

let person = new Object();
setName(person);
console.log(person.name);    // "Nicholas"
```

В этом коде переменной `person` назначается созданный объект, который затем передается в функцию `setName`, при этом он копируется в локальную переменную `obj`. Внутри функции переменные `obj` и `person` указывают на один и тот же объект, вследствие чего доступ к объекту с помощью `obj` выполняется по ссылке, хотя он и был передан в функцию по значению. При задании свойства `name` в функции это изменение отражается вне функции, потому что объект, на который указывает `obj`, находится в куче и доступен глобально. Многие разработчики ошибаются, когда думают, что если локальное изменение объекта проявляется глобально, значит, аргумент был передан по ссылке. Взгляните на измененный код, позволяющий убедиться, что объекты передаются по значению:

```
function setName(obj) {
    obj.name = "Nicholas";
    obj = new Object();
    obj.name = "Greg";
}

let person = new Object();
setName(person);
console.log(person.name);    // "Nicholas"
```

Функция `setName()` содержит две дополнительные строки кода, которые переопределяют `obj` как новый объект с другим именем. Когда в функцию `setName()` передается объект `person`, его свойству `name` присваивается значение `"Nicholas"`. Затем переменной `obj` назначается новый объект, а его свойству `name` присваивается значение `"Greg"`. Если бы переменная `person` была передана по ссылке, она автоматически стала бы указывать на объект со свойством `"Greg"`. Однако при выводе свойства `person.name` снова отображается значение `"Nicholas"`, то есть первоначальная ссылка остается нетронутой, хотя значение аргумента внутри функции изменилось. При перезаписи переменной `obj` в функции она становится указателем на локальный объект, который уничтожается сразу по завершении функции.

ПРИМЕЧАНИЕ Аргументы функций в ECMAScript можно считать обычными локальными переменными.

Проверка типа

Чтобы определить, относится ли переменная к одному из примитивных типов, лучше всего использовать оператор `typeof`, упомянутый в предыдущей главе. Точнее говоря, это наилучший способ определить, является ли переменная строкой, числом, логическим значением или значением `undefined`. Для объектов и значения `null` оператор `typeof` возвращает `"object"`:

```
let s = "Nicholas";
let b = true;
let i = 22;
let u;
let n = null;
let o = new Object();

console.log(typeof s);    // string
console.log(typeof i);    // number
console.log(typeof b);    // boolean
console.log(typeof u);    // undefined
console.log(typeof n);    // object
console.log(typeof o);    // object
```

Хотя оператор `typeof` вполне подходит для примитивных значений, при работе со ссылочными значениями от него мало пользы. Обычно вас не интересует, является ли значение объектом, — вам нужно знать, какого объект типа. Чтобы помочь с этим, ECMAScript предоставляет оператор `instanceof` со следующим синтаксисом:

```
результат = переменная instanceof конструктор
```

ПРИМЕЧАНИЕ Оператор `typeof` также возвращает значение `"function"` для функций. Кроме того, в Safari до версии 5 включительно и Chrome до версии 7 включительно он из-за особенностей реализации возвращает `"function"` для регулярных выражений. В ECMA-262 указано, что оператор `typeof` должен возвращать значение `"function"` для любых объектов, реализующих внутренний метод `[[Call]]`. Поскольку регулярные выражения реализуют его в этих браузерах, `typeof` возвращает `"function"`. В Internet Explorer и Firefox оператор `typeof` возвращает для регулярных выражений значение `"object"`.

Оператор `instanceof` возвращает `true`, если переменная — экземпляр конкретного ссылочного типа, определяемого по его цепочке прототипов (см. главу 8 «Объекты, классы и объектно-ориентированное программирование»):

```
console.log(person instanceof Object);   // относится ли person к типу Object?
console.log(colors instanceof Array);    // относится ли colors к типу Array?
console.log(pattern instanceof RegExp);  // относится ли pattern к типу RegExp?
```

Все ссылочные значения по определению являются экземплярами типа `Object`, так что для ссылочных значений и конструктора `Object` оператор `instanceof` всегда возвращает `true`. Соответственно, для примитивных значений он всегда возвращает `false`, потому что примитивные значения не объекты.

КОНТЕКСТ ВЫПОЛНЕНИЯ И ОБЛАСТЬ ВИДИМОСТИ

Концепция контекста выполнения, или просто *контекста* (context), очень важна в JavaScript. От контекста выполнения переменной или функции зависит, какие другие данные ей доступны и как она должна работать. С каждым контекстом выполнения связан *объект переменных* (variable object), содержащий все переменные и функции контекста. Он недоступен в коде, но используется за кулисами для обработки данных.

Наиболее общим является глобальный контекст выполнения. В зависимости от среды выполнения для ECMAScript-реализации он может представляться разными объектами. В веб-браузерах глобальным контекстом считается контекст объекта `window` (см. главу 12 «Объектная модель браузера»), так что все глобальные переменные и функции, объявленные с помощью ключевого слова `var`, создаются как свойства и методы объекта `window`. Объявления, использующие `let` и `const` на верхнем уровне, не определены в глобальном контексте, но они разрешаются идентично в цепочке областей действия. Когда весь код в контексте выполнен, он уничтожается вместе со всеми определенными в нем переменными и функциями (глобальный контекст существует вплоть до завершения приложения, например до закрытия веб-страницы или веб-браузера).

У каждого вызова функции имеется свой контекст выполнения. При передаче управления в функцию ее контекст помещается в стек контекста, а при выходе из функции он извлекается из стека, при этом управление возвращается в прежний контекст. Так осуществляется контроль над последовательностью операций в ECMAScript-программе.

При выполнении кода в контексте создается *цепочка областей видимости* (scope chain) объектов переменных, которая обеспечивает упорядоченный доступ ко всем переменным и функциям, доступным в контексте выполнения. Первым звеном цепочки областей видимости всегда является объект переменных контекста, код которого выполнятся. Если контекст — функция, в качестве объекта переменных используется *объект активации* (activation object), который начинает существование с единственной переменной `arguments` (у глобального контекста ее нет). Каждый последующий объект переменных в цепочке относится к все более внешнему контексту, пока не достигается глобальный контекст. Объект переменных глобального контекста всегда последний в цепочке областей видимости.

При разрешении идентификатора его имя ищется в цепочке областей видимости. Поиск всегда осуществляется в направлении от первого звена к последнему, пока не обнаруживается идентификатор (если найти его не удается, обычно возникает ошибка).

Рассмотрим следующий код:

```
var color = "blue";

function changeColor() {
    if (color === "blue") {
        color = "red";
    } else {
        color = "blue";
    }
}

changeColor();
```

В этом простом примере функция `changeColor()` имеет цепочку областей видимости с двумя объектами: собственным объектом переменных (в котором определен объект `arguments`) и объектом переменных глобального контекста. Переменная `color` доступна в функции, потому что она имеется в цепочке областей видимости.

В локальном контексте вместе с глобальными переменными можно использовать локальные, например:

```
var color = "blue";

function changeColor() {
    let anotherColor = "red";

    function swapColors() {
        let tempColor = anotherColor;
        anotherColor = color;
        color = tempColor;

        // здесь доступны переменные color, anotherColor и tempColor
    }

    // здесь доступны переменные color и anotherColor, но не tempColor
    swapColors();
}

// здесь доступна только переменная color
changeColor();
```

В этом коде три контекста выполнения: глобальный контекст, локальный контекст функции `changeColor()` и локальный контекст функции `swapColors()`. В глобальном контексте определены переменная `color` и функция `changeColor()`. В локальном контексте `changeColor()` определены переменная `anotherColor` и функция `swapColors()`, но в нем также доступна переменная `color` из глобального контекста. Локальный контекст `swapColors()` содержит одну переменную `tempColor`, которая доступна только в этой функции, но не в глобальном контексте и не в локальном контексте `changeColor()`. В контексте `swapColors()` также полностью доступны переменные из двух других контекстов выполнения, потому что они являются родительскими по отношению к нему. Цепочка областей видимости для этого примера представлена на рис. 4.3.

На этом рисунке разные контексты выполнения показаны прямоугольниками. Внутреннему контексту по цепочке областей видимости доступно все из обоих внешних контекстов, но тем во внутреннем контексте недоступно ничего. Связь между контекстами линейна и упорядочена. Каждый контекст может искать переменные и функции в цепочке областей видимости по направлению наружу, но не внутрь. В цепочке областей видимости локального контекста `swapColors()` три объекта: объекты переменных функций `swapColors()` и `changeColor()` и глобальный объект переменных. При обращении к переменной или функции в локальном контексте `swapColors()` начинается поиск ее имени в локальном объекте переменных. Если найти имя не удается, выполняется поиск в следующем объекте цепочки. Цепочка областей видимости контекста `changeColor()` содержит его собственный объект переменных и глобальный объект переменных, а контекст `swapColors()` в нем недоступен.

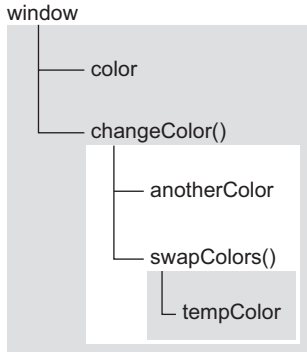

Рис. 4.3

ПРИМЕЧАНИЕ Аргументы функций считаются переменными и подчиняются тем же правилам доступа, что и любые другие переменные в контексте выполнения.

Приращение цепочки областей видимости

Основных типов контекстов выполнения два, глобальный контекст и контекст функции (есть также третий внутри вызова `eval()`), но существуют и другие способы приращения цепочки областей видимости. Некоторые инструкции временно добавляют области видимости к началу цепочки, которые затем удаляются. Это происходит в двух случаях:

- при входе в блок `catch` инструкции `try-catch`;
- при выполнении инструкции `with`.

Обе эти инструкции добавляют объект переменных к началу цепочки областей видимости. В случае инструкции `with` к цепочке областей видимости добавляется переданный инструкции объект, а в случае блока `catch` создается новый объект переменных с объявлением объекта ошибки. Рассмотрим пример:

```
function buildUrl() {
    let qs = "?debug=true";

    with(location) {
        let url = href + qs;
    }

    return url;
}
```

В этом примере инструкция `with` применяется к объекту `location`, который добавляется к цепочке областей видимости. В функции `buildUrl()` определена единственная

переменная qs. При обращении к переменной href на самом деле используется переменная location.href, которая относится к отдельному объекту переменных. При ссылке на qs используется переменная, определенная в функции buildUrl() и относящаяся к объекту переменных контекста buildUrl(). Также в инструкции with объявляется переменная url, которая становится частью контекста функции и потому может быть возвращена как значение функции.

ПРИМЕЧАНИЕ В Internet Explorer до версии 8 включительно ошибка, перехваченная в блоке catch, добавляется в объект переменных контекста выполнения, а не блока catch, из-за чего она доступна даже вне блока catch. Это отклонение от правила устранено в Internet Explorer 9.

Объявление переменной

После введения ES6 JavaScript претерпел явное преобразование в отношении того, как переменные объявляются в языке. В ECMAScript 5.1 ключевым словом, подходящим для всего на свете, было var. ES6 не только вводит два новых ключевых слова let и const, но и делает эти новые ключевые слова в подавляющем большинстве предпочтительными объявлениями по сравнению с var.

Объявление области действия функции с использованием var

При объявлении переменной с помощью ключевого слова var она автоматически добавляется в текущий контекст. В функции им является локальный контекст, а в инструкции with — контекст функции. Если переменная инициализируется без предварительного объявления, она автоматически добавляется в глобальный контекст. Рассмотрим следующий пример:

```
function add(num1, num2) {
    var sum = num1 + num2;
    return sum;
}

let result = add(10, 20);  // 30
console.log(sum);          // ошибка, переменная sum недействительна
```

В функции add() определяется локальная переменная sum, которой присваивается результат сложения. Затем это значение возвращается из функции, но сама переменная sum недоступна вне функции. Если опустить ключевое слово var, переменная sum станет доступна после вызова add():

```
function add(num1, num2) {
    sum = num1 + num2;
    return sum;
}

let result = add(10, 20);  // 30
console.log(sum);          // 30
```

Здесь переменная `sum` инициализируется без объявления. При вызове `add()` она создается в глобальном контексте и продолжает существовать даже после завершения функции.

> **ПРИМЕЧАНИЕ** Инициализация JS-переменных без их объявления часто ошибочна. Во избежание проблем рекомендуется всегда объявлять переменные до их инициализации. В строгом режиме инициализация необъявленной переменной считается ошибкой.

Объявление `var` будет выведено в начало функции или глобальной области видимости и перед любым существующим кодом внутри нее. Это называется поднятием, оно позволяет безопасно использовать перемещенную переменную в любом месте в одной и той же области видимости без учета того, была ли она объявлена или нет. Однако на практике это может привести к допустимому, но причудливому коду, в котором переменная используется до ее объявления. Вот пример двух эквивалентных фрагментов кода в глобальной области видимости:

```
var name = "Jake";

// Это эквивалентно следующему:

name = 'Jake';
var name;
```

Ниже пример двух эквивалентных функций:

```
function fn1() {
    var name = 'Jake';
}

// Это эквивалентно следующему:
function fn2() {
    var name;
    name = 'Jake';
}
```

Вы можете доказать, что переменная поднимается, проверив ее перед объявлением. Поднятие объявления означает, что вместо `ReferenceError` появится `undefined`:

```
console.log(name); // undefined
var name = 'Jake';
function() {
    console.log(name); // undefined
    var name = 'Jake';
}
```

Объявление области действия блока с помощью let

Совершенно новый для ES6, `let` работает во многом так же, как и `var`, но он ограничен на уровне блоков — новая концепция в JavaScript. Область видимости блока

определяется как ближайший набор фигурных скобок {}. Это означает, что блоки if, блоки while, блоки function и даже отдельные блоки будут охватывать любую переменную, объявленную с помощью let.

```
if (true) {
    let a;
}
console.log(a);    // ReferenceError: a is not defined

while (true) {
    let b;
}
console.log(b);    // ReferenceError: b is not defined

function foo() {
    let c;
}
console.log(c);    // ReferenceError: c is not defined
                   // Это не должно быть удивительно, поскольку
                   // объявление var также выдает ошибку

// Это не объектный литерал, а отдельный блок.
// Интерпретатор JavaScript будет идентифицировать его как таковой на основе его
// содержимого.
{
    let d;
}
console.log(d);    // ReferenceError: d is not defined
```

В аналогичном отклонении от поведения var, let не может быть объявлена дважды внутри одной и той же области видимости блока. Повторяющиеся объявления var просто игнорируются; повторяющиеся объявления let генерируют ошибку SyntaxError.

```
var a;
var a;
// Ошибок нет

{
    let b;
    let b;
}
// SyntaxError: Identifier 'b' has already been declared
```

Поведение let особенно полезно при использовании итераторов внутри циклов. Объявления итераторов, использующие var, будут выходить за пределы цикла после его завершения, что часто является очень нежелательным поведением. Рассмотрим эти два примера:

```
for (var i = 0; i < 10; ++i) {}
console.log(i); // 10

for (let j = 0; j < 10; ++j) {}
console.log(j); // ReferenceError: j is not defined
```

Технически `let` используется во время выполнения JavaScript, но из-за «временной мертвой зоны» запрещено использовать переменную выше ее фактического объявления. Следовательно, для целей написания JavaScript `let` не поднимается так же, как `var`.

Объявление констант с помощью const

ES6 также вводит `const` в качестве компаньона для `let`. Переменная, объявленная с использованием `const`, должна быть инициализирована некоторым значением. После объявления значение не может быть переопределено новым в любой момент его жизни.

```
const a; // SyntaxError: Missing initializer in const declaration

const b = 3;
console.log(b); // 3
b = 4; // TypeError: Assignment to a constant variable
```

Помимо применения правил `const`, переменные `const` ведут себя одинаково с переменными `let`:

```
if (true) {
    const a = 0;
}
console.log(a); // ReferenceError: a is not defined

while (true) {
    const b = 1;
}
console.log(b); // ReferenceError: b is not defined

function foo() {
    const c = 2;
}
console.log(c); // ReferenceError: c is not defined

{
    const d = 3;
}
console.log(d); // ReferenceError: d is not defined
```

Объявление `const` применяется только к примитиву или объекту верхнего уровня. Другими словами, переменная `const`, назначенная объекту, не может быть переназначена другому ссылочному значению, но при этом ключи внутри объекта не защищены.

```
const o1 = {};
o1 = {}; // TypeError: Assignment to a constant variable;

const o2 = {};
o2.name = 'Jake';
console.log(o2.name); // 'Jake'
```

Если вы хотите сделать весь объект неизменным, можно использовать `Object.freeze()`, хотя попытка присвоения свойства не приведет к ошибкам, а лишь молча потерпит неудачу:

```
const o3 = Object.freeze({});
o3.name = 'Jake';
console.log(o3.name); // undefined
```

Поскольку объявления `const` подразумевают, что значение имеет один тип и является неизменным, JavaScript-компилятор может заменить все его экземпляры фактическим значением вместо выполнения поиска переменной через таблицу поиска. Движок Google Chrome V8 выполняет подобную оптимизацию.

ПРИМЕЧАНИЕ Согласно наилучшим методикам, предпочтительно использовать `const` как можно чаще, если только вам действительно не требуется переменная, которая может быть переопределена, и это не окажет существенного влияния на процесс разработки. Так вы можете найти целую золотую жилу ошибок переопределения гораздо раньше, чем обычно.

Поиск идентификатора

При доступе к идентификатору для чтения или записи в конкретном контексте необходимо выяснить, какую переменную он представляет. Для этого начинается поиск идентификатора с конкретным именем с начала цепочки областей видимости. Если имя идентификатора обнаруживается в локальном контексте, поиск прекращается и переменная используется. Если имя переменной в локальном контексте отсутствует, запускается поиск в следующем элементе цепочки (имейте в виду, что у объектов в цепочке областей видимости также есть цепочка прототипов, так что поиск может охватывать и цепочку прототипов каждого объекта). Это продолжается, пока не будет достигнут объект переменных глобального контекста. Если и в нем нет идентификатора, значит, он не был объявлен.

Рассмотрим следующий пример:

```
var color = "blue";

function getColor() {
    return color;
}

console.log(getColor());  // "blue"
```

Здесь при вызове функции `getColor()` выполняется обращение к переменной `color`, при этом начинается поиск, включающий два этапа. Сначала идентификатор `color` ищется в объекте переменных функции `getColor()`. Если найти его не удается, выполняется поиск в следующем объекте переменных (из глобального контекста). Поскольку переменная `color` определена в нем, поиск завершается.

Таким образом, доступ к локальной переменной автоматически блокирует продолжение поиска в другом объекте переменных. Это означает, что идентификатор в родительском контексте не используется, если в локальном контексте есть идентификатор с тем же именем, например:

```
var color = "blue";

function getColor() {
    let color = "red";
    return color;
}

console.log(getColor());  // "red"
```

Использование блочных объявлений не меняет процесс поиска, но может добавить дополнительные уровни к лексической иерархии:

```
var color = 'blue';

function getColor() {
    let color = 'red';
    {
        let color = 'green';
        return color;
    }
}

console.log(getColor()); // 'green'
```

В этом измененном коде при вызове функции `getColor()` объявляется локальная переменная `color`. При выполнении второй строки функции начинается поиск в локальном контексте, в котором обнаруживается переменная `color` со значением `"green"`. На этом поиск прекращается, и значение локальной переменной возвращается из функции. В коде после объявления локальной переменной `color` глобальная переменная `color` недоступна без квалификатора области видимости (`window.color`).

> **ПРИМЕЧАНИЕ** Поиск переменных имеет свою цену. Доступ к локальным переменным выполняется быстрее, чем к глобальным, потому что для этого не нужно просматривать цепочку областей видимости. С другой стороны, алгоритмы поиска идентификаторов становятся в интерпретаторах JavaScript все эффективнее, так что это различие может со временем стать несущественным.

СБОРКА МУСОРА

JavaScript — это язык со сборкой мусора, то есть за управление памятью при работе сценариев отвечает среда выполнения. В языках вроде C и C++ слежение за использованием памяти относится к важнейшим задачам и является источником

многих проблем. JavaScript освобождает разработчиков от забот по управлению памятью, автоматически выделяя сценариям нужную память и возвращая в среду память, которая больше не используется. Идея сборки мусора проста: нужно выяснить, какие переменные больше не потребуются, и освободить связанную с ними память. Сборщик мусора запускается периодически с заданной частотой или в предопределенные моменты выполнения кода. Процесс сборки мусора является приблизительным и несовершенным решением, потому что общая проблема понимания, нужен ли какой-то фрагмент памяти, неразрешима (undecidable), то есть не может быть решена алгоритмом.

При нормальном жизненном цикле локальной переменной она создается в ходе выполнения функции. В этот момент для нее выделяется память в стеке (и, возможно, в куче). Далее переменная используется в функции, и рано или поздно функция завершается. После этого переменная больше не нужна, поэтому ее память может быть освобождена для повторного использования. В описанном случае ясно, что переменная стала ненужной, но не все ситуации так очевидны. Сборщик мусора должен отслеживать, какие переменные могут потребоваться, а какие нет, чтобы можно было выявить те из них, которые допускают освобождение памяти. Способ выявления неиспользуемых переменных зависит от реализации, но в браузерах традиционно применяются две стратегии: отслеживание и очистка и подсчет ссылок.

Отслеживание и очистка

Наиболее популярным способом сборки мусора в JavaScript является *отслеживание и очистка* (mark-and-sweep). Когда переменная появляется в контексте, например при объявлении внутри функции, она помечается как находящаяся в контексте. Память переменных в контексте не должна освобождаться, потому что они могут потребоваться, пока продолжается выполнение кода в этом контексте. Когда переменная покидает контекст, она помечается как находящаяся вне контекста.

Переменные могут помечаться разными способами — например, с помощью бита-переключателя или списков переменных «в контексте» и «вне контекста». Реализация пометок не имеет значения, важен принцип.

При запуске сборщика мусора он тем или иным способом маркирует все переменные, находящиеся в памяти, а затем снимает пометки со всех переменных в контексте и переменных, на которые они ссылаются. Переменные, оставшиеся помеченными после этого, считаются готовыми к удалению, потому что они недостижимы ни для каких переменных, находящихся в контексте. Затем сборщик мусора очищает память, уничтожая все помеченные переменные и возвращая связанную с ними память в среду.

На 2008 г. в Internet Explorer, Firefox, Opera, Chrome и Safari для сборки мусора использовался метод отслеживания и очистки (или его вариации), но время сборки мусора определялось в них по-разному.

Подсчет ссылок

Еще одним способом сборки мусора, который оказался не столь востребованным, является *подсчет ссылок* (reference counting). Его идея в том, что каждое значение отслеживает количество ссылок на него. Когда переменная объявляется и ей присваивается ссылочное значение, счетчик ссылок равен единице. Если это же значение присваивается другой переменной, счетчик ссылок увеличивается, а если переменная со ссылкой на значение перезаписывается другим значением, он уменьшается. Когда счетчик ссылок на значение достигает нуля, оно становится недоступным и занимаемую им память можно безопасно освободить, что и делает сборщик мусора при следующем запуске.

Подсчет ссылок первоначально использовался в Netscape Navigator 3.0, но при этом сразу возникла проблема циклических ссылок. *Циклическая ссылка* (circular reference) имеет место, если объект A указывает на объект B, а B — на A, например:

```
function problem() {
    let objectA = new Object();
    let objectB = new Object();

    objectA.someOtherObject = objectB;
    objectB.anotherObject = objectA;
}
```

В этом примере `objectA` и `objectB` ссылаются друг на друга посредством своих свойств, и у каждого из них счетчик ссылок равен двум. В системе с отслеживанием и очисткой это не было бы проблемой, потому что оба объекта покинули бы область видимости при завершении функции. Однако в системе с подсчетом ссылок оба объекта продолжают существовать после выполнения функции, потому что их счетчики ссылок никогда не достигают нуля. Если эта функция вызывается многократно, большие объекты памяти так и не освобождаются. По этой причине в Netscape Navigator 4.0 сборка мусора с подсчетом ссылок была заменена алгоритмом отслеживания и очистки, но проблемы подсчета ссылок на этом не закончились.

Не все объекты в Internet Explorer 8 и более ранних версий являются естественными объектами JavaScript. BOM- и DOM-объекты реализованы как объекты COM (Component Object Model — объектная модель программных компонентов) на C++ и используют для сборки мусора подсчет ссылок. Несмотря на то что интерпретатор JavaScript в Internet Explorer использует отслеживание и очистку, из-за подсчета ссылок для COM-объектов может возникать знакомая проблема циклических ссылок, например:

```
let element = document.getElementById("some_element");
let myObject = new Object();
myObject.element = element;
element.someObject = myObject;
```

В этом примере формируется циклическая ссылка между DOM-элементом `element` и JavaScript-объектом `myObject`. Свойство `element` переменной `myObject` указывает на DOM-элемент, а его свойство `someObject`, в свою очередь, указывает на `myObject`.

Из-за этой циклической ссылки память, занимаемая DOM-элементом, не освобождается, даже если он удаляется со страницы.

Для предотвращения подобных проблем с циклическими ссылками следует разрывать связи между естественными JavaScript-объектами и DOM-элементами по завершении работы с ними. Например, в следующем коде циклическая ссылка из предыдущего примера уничтожается:

```
myObject.element = null;
element.someObject = null;
```

Присвоение переменной значения `null` разрывает связь между ней и значением, на которое она ссылалась до этого. При следующем запуске сборщика мусора он может удалить это значение и возвратить память в среду.

Чтобы решить некоторые из этих проблем, в Internet Explorer 9 BOM- и DOM-объекты были сделаны настоящими JavaScript-объектами. Это унифицировало алгоритмы сборки мусора и устранило частые причины утечек памяти.

ПРИМЕЧАНИЕ Циклические ссылки могут возникать в ряде других сценариев, и их мы рассмотрим в следующих главах.

Производительность

Периодическая сборка мусора может требовать много ресурсов, если в памяти находится большое количество переменных, поэтому важно подбирать для нее подходящее время. В частности, на мобильных устройствах с ограниченной системной памятью сборка мусора может заметно снизить скорость и частоту кадров при рендеринге. Вы не можете заранее знать, когда начнется сборка мусора, поэтому лучшая стратегия состоит в организации кода так, чтобы он позволял сборщику мусора выполнять свою работу наилучшим образом в любое запланированное время.

Современные сборщики мусора решают, когда начинать работу, основываясь на наборе эвристик, которые измеряются из среды выполнения JavaScript. Эти эвристики будут варьироваться в зависимости от движка, но все они будут приблизительно основаны на размере и количестве объектов, которые были распределены. Например, из сообщения в блоге V8 в 2016 г.: «В конце полной сборки мусора стратегия наращивания кучи V8 определяет, когда произойдет следующая сборка мусора, на основе количества живых объектов с некоторой дополнительной погрешностью».

Когда-то в Internet Explorer это было серьезной проблемой. Сборка мусора запускалась в этом браузере согласно количеству выделенных фрагментов памяти, а именно — по достижении 256 переменных, 4096 литералов объектов/массивов и ячеек массивов или 64 Кбайт строковых данных. Проблема заключалась в том, что если для решения некой задачи в сценарии требовалось такое большое количество переменных, обычно эти пределы затем достигались снова и снова, что каждый раз

запускало сборку мусора. Это существенно снижало быстродействие и вынудило изменить алгоритм сборки мусора в Internet Explorer 7.

В сборщике мусора Internet Explorer 7 была реализована динамическая настройка порогов выделения памяти для переменных, литералов и ячеек массивов. Сначала в Internet Explorer 7 действуют те же ограничения, что и в Internet Explorer 6. Если при сборке мусора освобождается менее 15 % выделенных фрагментов памяти, порог для переменных, литералов и ячеек массивов удваивается. Если освобождается более 85 % выделенных фрагментов, восстанавливаются пороговые значения, предлагаемые по умолчанию. Это простое изменение значительно повысило быстродействие браузера при обработке веб-страниц с большим объемом JS-кода.

ПРИМЕЧАНИЕ В некоторых браузерах можно инициировать сборку мусора вручную, но делать это не рекомендуется. В Internet Explorer для немедленного запуска сборки мусора используется метод `window.CollectGarbage()`, а в Opera 7 и более поздних версий – метод `window.opera.collect()`.

Управление памятью

В средах программирования со сборкой мусора разработчикам обычно не приходится волноваться об управлении памятью. Однако интерпретаторы JavaScript работают в среде, уникальной в плане управления памятью и сборки мусора. Как правило, объем памяти, доступной в веб-браузерах, гораздо меньше, чем в обычных приложениях, и даже еще меньше, чем в мобильных браузерах. Он ограничивается в основном ради безопасности, чтобы сценарии JavaScript в веб-страницах не могли вызвать сбой операционной системы, израсходовав всю системную память. Ограничения памяти влияют не только на выделение памяти для переменных, но и на стек вызовов и количество инструкций, выполняемых в одном потоке.

Экономия памяти в JS-сценариях ускоряет обработку страниц, а наилучший способ оптимизировать использование памяти — это хранить в коде только те данные, которые требуются для его выполнения. Если данные больше не нужны, лучше всего присвоить соответствующей переменной значение `null`, чтобы разорвать ее связь с данными. Этот совет относится преимущественно к глобальным переменным и свойствам глобальных объектов. В случае локальных переменных ссылки на данные удаляются автоматически, когда переменные покидают контекст, например:

```
function createPerson(name) {
    let localPerson = new Object();
    localPerson.name = name;
    return localPerson;
}

let globalPerson = createPerson("Nicholas");

// какие-то действия с globalPerson

globalPerson = null;
```

В этом коде переменной `globalPerson` присваивается значение, возвращенное функцией `createPerson()`. Внутри функции создается объект `localPerson`, к которому добавляется свойство `name`. Затем этот объект возвращается из функции и назначается переменной `globalPerson`. Переменная `localPerson` покидает контекст при завершении функции `createPerson()`, так что разрывать ее связь со значением не требуется. Что касается `globalPerson`, то это глобальная переменная, поэтому когда она больше не требуется, ей присваивается значение `null`.

Помните, что разрыв связи переменной со значением не приводит к автоматическому освобождению занятой им памяти. Значение просто выводится из контекста, что делает возможным возвращение памяти при следующей сборке мусора.

Повышение производительности благодаря объявлениям const и let

Введение этих ключевых слов в ES6 пойдет на благо не только вашему стилю кода, но и процессу сборки мусора. Поскольку `const` и `let` видимы в пределах блока, а не функции, в зависимости от того, как организован код, это может сигнализировать сборщику мусора о том, что выделенная переменная подходит для очистки гораздо раньше, чем это было бы при использовании `var`. Это может произойти в ситуациях, когда область действия блока заканчивается намного раньше, чем область действия функции.

Скрытые классы и операция delete

В зависимости от того, где ожидается запуск JavaScript, иногда стоит рассмотреть различные факторы производительности, основанные на том, какой движок JavaScript использует браузер. По состоянию на 2017 г. самым популярным веб-браузером является Google Chrome, который использует движок JavaScript V8. Этот механизм использует скрытые классы при компиляции интерпретированного кода JavaScript в реальный машинный код, и это может быть важно для вас, если вы пишете код, чувствительный к производительности.

Во время выполнения V8 будет ассоциировать скрытые классы с каждым созданным объектом для отслеживания формы его свойств. Объекты, которые могут совместно использовать один и тот же скрытый класс, будут иметь лучшую производительность, и V8 оптимизирует их, насколько это будет возможно. Рассмотрим следующий фрагмент кода:

```
function Article() {
    this.title = 'Inauguration Ceremony Features Kazoo Band';
}

let a1 = new Article();
let a2 = new Article();
```

За кулисами V8 настроит два экземпляра класса для совместного использования одного и того же скрытого класса. Это имеет смысл, потому что они имеют общий

конструктор и прототип. Предположим, что вы добавили следующую строку в конец вышеизложенного кода:

```
a2.author = 'Jake';
```

Теперь два экземпляра `Article` будут иметь две различные реализации скрытых классов. В зависимости от частоты этой операции и размера скрытых классов это может оказать существенное влияние на производительность.

Решение, конечно же, состоит в том, чтобы избежать динамического назначения свойства JavaScript и вместо этого объявить все свойства внутри конструктора, как показано здесь:

```
function Article(opt_author) {
    this.title = 'Inauguration Ceremony Features Kazoo Band';
    this.author = opt_author;
}

let a1 = new Article();
let a2 = new Article('Jake');
```

Теперь два экземпляра будут вести себя по существу одинаково (не считая возвращаемых значений `hasOwnProperty`), и они также будут использовать скрытый класс, что потенциально приведет к повышению производительности. Имейте в виду, что использование ключевого слова `delete` может привести к той же фрагментации скрытого класса. Это поведение демонстрируется в примере ниже:

```
function Article() {
    this.title = 'Inauguration Ceremony Features Kazoo Band';
    this.author = 'Jake';
}

let a1 = new Article();
let a2 = new Article();

delete a1.author;
```

В конце этого фрагмента два экземпляра больше не будут использовать скрытый класс, даже если они используют унифицированный конструктор. Динамическое удаление свойства даст тот же эффект, что и динамическое добавление. Наилучшие методики диктуют, что нежелательные свойства должны быть заданы как `null`. Это позволит скрытым классам оставаться нетронутыми и общими и также повлияет на удаление ссылок в интересах сборщика мусора.

```
function Article() {
    this.title = 'Inauguration Ceremony Features Kazoo Band';
    this.author = 'Jake';
}

let a1 = new Article();
let a2 = new Article();

a1.author = null;
```

Утечки памяти

Плохо написанный JS-код может привести к некоторым хитрым и коварным утечкам памяти. На устройствах с ограниченной памятью или в контексте функций, которые вызываются много раз, это может вызвать большие проблемы. В подавляющем большинстве случаев утечки памяти в JavaScript вызваны нежелательными ссылками.

Одна из наиболее распространенных и легко исправляемых утечек памяти — случайное объявление глобальных переменных. В следующем коде переменная не имеет префикса с ключевым словом объявления:

```
function setName() {
    name = 'Jake';
}
```

В этом примере интерпретатор будет обрабатывать это как `window.name = 'Jake'`, и, конечно, свойства, установленные для объекта `window`, никогда не будут очищены при условии, что сам объект `window` не очищен. Это легко исправить, добавив к объявлению префикс `var`, `let` или `const`, который выйдет из области видимости в конце выполнения функции.

Интервальные таймеры также могут незаметно вызывать утечки памяти. В следующем примере код устанавливает интервал, который ссылается на переменную, предоставленную посредством замыкания:

```
let name = 'Jake';
setInterval(() => {
    console.log(name);
}, 100);
```

Пока данный интервальный таймер работает, функция-обработчик, содержащая ссылку на `name`, остается выделенной. Сборщик мусора распознает это и поэтому не может очистить внешнюю переменную.

Замыкания в JavaScript — весьма распространенный способ утечки памяти, который не так просто заметить. Рассмотрим следующий пример:

```
let outer = function() {
    let name = 'Jake';
    return function() {
        return name;
    };
};
```

Это приводит к утечке памяти, выделенной для `name`. Код создает внутреннее замыкание, поэтому, пока существует функция `outer`, переменная `name` не может быть очищена, поскольку в этом замыкании будет постоянная ссылка на нее. Если бы содержимое переменной `name` было слишком большим, а не просто короткой строкой, это могло бы привести к серьезным проблемам.

Статическое распределение и объектные пулы

В самом конце спектра производительности JavaScript вы можете захотеть выжать из браузера самую последнюю каплю производительности. Чтобы достичь этого, нужно сосредоточиться на минимизации количества операций по сборке мусора, выполняемых браузером. Поскольку вы непосредственно не контролируете, когда происходит сборка мусора, можно вместо этого оптимизировать эвристику, которую браузеры используют при планировании сборки мусора. Теоретически, если вы будете ответственно использовать выделенную память и в то же время избавитесь от лишних сборок мусора, то сможете добиться прироста производительности, который в противном случае был бы потерян при освобождении памяти.

Одной из важных метрик, измеряемых браузером при принятии решения, когда планировать сборку мусора, является скорость оттока объектов. Если множество объектов создается, а затем выходит из области видимости, браузер будет более агрессивно планировать сборку мусора, что, конечно, замедлит работу приложения. Рассмотрим следующий пример — функцию сложения двухмерного вектора:

```
function addVector(a, b) {
    let resultant = new Vector();
    resultant.x = a.x + b.x;
    resultant.y = a.y + b.y;
    return resultant;
}
```

При вызове эта функция создает новый объект в куче, изменяет его и возвращает вызывающей стороне. Если время жизни данного векторного объекта короткое, он скоро потеряет все свои ссылки и получит право на сборку мусора. Если функция сложения векторов вызывается часто, планировщик сборки мусора заметит высокий уровень оттока объектов и сборка мусора будет планироваться чаще.

Предположим, что вместо создания динамического вектора, вы изменили метод для использования существующего векторного объекта:

```
function addVector(a, b, resultant) {
    resultant.x = a.x + b.x;
    resultant.y = a.y + b.y;
    return resultant;
}
```

Конечно, для этого требуется, чтобы аргумент вектора `resultant` был свежим и создавался где-то еще, но поведение этой функции остается неизменным. Где тогда создать вектор, избегая при этом вовлечения эвристики сборщика мусора?

Одна стратегия заключается в использовании пула объектов. В какой-то момент инициализации вы создадите пул объектов, который управляет коллекцией перерабатываемых объектов. Ваше приложение может запросить объект из этого пула, установить его свойства, использовать его и вернуть в пул при завершении обработки. Поскольку создания новых экземпляров не происходит, эвристика сборки мусора не будет измерять увеличение оттока объектов и сборка мусора будет происходить реже. Псевдореализация пула объектов может выглядеть примерно так:

```
// vectorPool – это существующий пул объектов
let v1 = vectorPool.allocate();
let v2 = vectorPool.allocate();
let v3 = vectorPool.allocate();

v1.x = 10;
v1.y = 5;
v2.x = -3;
v2.y = -6;

addVector(v1, v2, v3);

console.log([v3.x, v3.y]); // [7, -1]

vectorPool.free(v1);
vectorPool.free(v2);
vectorPool.free(v3);

// Если у объектов есть свойства, ссылающиеся на другие объекты,
// то для них здесь также должно быть установлено значение null
v1 = null;
v2 = null;
v3 = null;
```

Если пул объектов выделяет векторы только по мере необходимости (то есть создает новые, когда они не существуют, и повторно использует уже существующие), эта реализация по сути будет жадным алгоритмом, который имеет монотонно увеличивающуюся, но статическую память. Этот пул должен поддерживать коллекцию, используя некоторую структуру, и хорошим выбором для этого является массив. Однако реализация, использующая массив, должна быть тщательно спроектирована, чтобы не вызывать дополнительную сборку мусора. Рассмотрим следующий пример:

```
let vectorList = new Array(100);
let vector = new Vector();
vectorList.push(vector);
```

Поскольку JavaScript использует динамически изменяемые массивы, механизм удалит массив размером 100 и создаст новый массив размером 200. Сборщик мусора заметит это удаление, и потому его можно будет запускать быстрее. Этого динамического распределения можно избежать, создавая массив соответствующего размера при инициализации, что позволит избавиться от вышеупомянутой операции изменения размера. Однако потребуется представление о том, насколько большим должен быть этот массив.

ПРИМЕЧАНИЕ Статическое распределение – крайняя форма оптимизации. Оно приведет к повышению производительности в момент, когда производительность вашего приложения будет ограничена из-за издержек при сборке мусора, но это будет происходить очень редко. В большинстве случаев это будет всего лишь недопустимой формой преждевременной оптимизации.

ИТОГИ

JavaScript-переменные могут содержать значения двух типов: примитивные и ссылочные. Примитивные значения относятся к одному из шести примитивных типов данных: неопределенному (undefined), нулевому (null), логическому (boolean), числовому (number), строковому (string) и символьному (symbol). Перечислим некоторые характеристики JavaScript-значений.

- Примитивные значения имеют фиксированный размер и хранятся в памяти в стеке.
- При копировании примитивного значения из одной переменной в другую создается его копия.
- Ссылочные значения — это объекты, которые хранятся в памяти в куче.
- Переменная со ссылочным значением на самом деле содержит не объект, а лишь указатель на него.
- При копировании ссылочного значения копируется только указатель, так что в результате обе переменные ссылаются на один объект.
- Для определения типа примитивных и ссылочных значений используются операторы `typeof` и `instanceof` соответственно.

И примитивные, и ссылочные переменные существуют в том или ином контексте выполнения (называемом также областью видимости), который определяет время жизни переменной, а также те части кода, в которых она доступна.

- Контекст выполнения может быть глобальным или локальным. В последнем случае он ограничен функцией или блоком.
- При входе в каждый новый контекст выполнения создается цепочка областей видимости, служащая для поиска переменных и функций.
- В локальном контексте функции или блока доступны переменные не только из текущей области видимости, но и из всех внешних контекстов, включая глобальный.
- В глобальном контексте доступны только переменные и функции из глобальной области видимости, а непосредственного доступа к каким-либо данным в локальных контекстах нет.
- Контексты выполнения переменных помогают управлять освобождением памяти.

В среде программирования JavaScript применяется сборка мусора, то есть разработчику не нужно беспокоиться о выделении или освобождении памяти. Механизм сборки мусора в JavaScript работает следующим образом.

- Значения, покидающие область видимости, автоматически помечаются как подлежащие удалению и удаляются во время сборки мусора.

- Наиболее популярный алгоритм сборки мусора — отслеживание и очистка. Он помечает неиспользуемые значения, а затем освобождает занимаемую ими память.
- В алгоритме подсчета ссылок отслеживается количество ссылок на конкретное значение. В интерпретаторах JavaScript он больше не применяется, но Internet Explorer использует его при доступе к сторонним объектам (таким как DOM-элементы).
- Подсчет ссылок приводит к проблемам при наличии циклических ссылок в коде.
- Разрыв связей переменных со значениями не только помогает при наличии циклических ссылок, но и повышает эффективность сборки мусора в целом. Когда глобальные объекты и их свойства становятся ненужными, присваивайте им значение `null`, чтобы оптимизировать возвращение памяти системе.

5

Ссылочные типы

- Работа с объектами
- Базовые типы данных в JavaScript
- Работа с примитивными типами и оболочками примитивных типов

Загрузки с сайта wrox.com для этой главы

Обратите внимание, что все примеры кода для этой главы доступны на веб-сайте книги по ссылке www.wrox.com/go/projavascript4e на вкладке Downloads.

Ссылочное значение (объект) — это экземпляр конкретного *ссылочного типа* (reference type). В ECMAScript ссылочные типы представляют собой структуры, которые используют для группировки данных и функций и часто ошибочно называют *классами* (classes). Хотя технически ECMAScript относится к объектно-ориентированным языкам, в нем нет некоторых традиционных базовых конструкций объектно-ориентированного программирования, в том числе классов и интерфейсов. Ссылочные типы также иногда называют *определениями объектов* (object definitions), потому что они описывают свойства и методы, которые должны быть у объектов.

> **ПРИМЕЧАНИЕ** Хотя ссылочные типы похожи на классы, это разные вещи. Во избежание путаницы термин «класс» далее в главе не используется.

Объекты представляют собой *экземпляры* (instances) конкретного ссылочного типа и создаются с помощью оператора `new`, за которым следует *конструктор* (constructor). Конструктор — это просто функция, служащая для создания объектов, например:

```
let now = new Date();
```

Эта инструкция создает экземпляр ссылочного типа `Date` и сохраняет его в переменной `now`. Конструктор `Date()` создает простой объект, содержащий лишь свойства и методы, предлагаемые по умолчанию. ECMAScrit предоставляет ряд встроенных ссылочных типов, таких как `Date`, для решения типичных задач программирования.

> **ПРИМЕЧАНИЕ** Функции также являются ссылочным типом, но это слишком широкая тема для данной главы, и поэтому им посвящена целая глава 10 «Функции».

ТИП DATE

Тип `Date` в ECMAScript основан на ранней версии `java.util.Date`. Даты хранятся в нем как количество миллисекунд, прошедших с полуночи 1 января 1970 г., согласно универсальному временному коду (Universal Time Code, UTC). Благодаря такому формату с помощью типа `Date` можно точно представлять даты, отстоящие от 1 января 1970 г. на 285 616 лет.

Чтобы создать объект `Date`, используйте оператор `new` с конструктором `Date`:

```
let now = new Date();
```

Если конструктор `Date` вызывается без аргументов, создается объект с текущими значениями даты и времени. Чтобы создать объект `Date` с другой датой или временем, нужно передать в конструктор значение даты в миллисекундах, прошедших с полуночи 1 января 1970 г. эпохи Unix, согласно UTC. Чтобы упростить решение этой задачи, можно использовать методы `Date.parse()` и `Date.UTC()`.

Метод `Date.parse()` принимает строковое представление даты и пытается преобразовать его в дату в миллисекундах. В пятой редакции ECMA-262 сказано, что метод `Date.parse()` должен поддерживать следующие форматы данных:

- месяц/день/год (например, 5/23/2019);
- название_месяца день, год (например, May 23, 2019);
- день_недели название_месяца день год часы:минуты:секунды часовой_пояс (например, Tue May 23 2019 00:00:00 GMT-0700);
- ГГГГ-ММ-ДДТЧЧ:мм:сс.cccZ (например, 2019-05-23T00:00:00) — этот расширенный формат ISO 8601 работает только в реализациях, совместимых с ECMAScript 5.

Например, создать объект `Date` для 23 мая 2019 г. можно следующим образом:

```
let someDate = new Date(Date.parse("May 23, 2019"));
```

Если строка, переданная в метод `Date.parse()`, не представляет дату, он возвращает значение `NaN`. Если строка даты передается непосредственно в конструктор `Date`, он неявно вызывает метод `Date.parse()`, поэтому приведенный пример можно переписать так:

```
let someDate = new Date("May 23, 2019");
```

Эта инструкция идентична предыдущей.

ПРИМЕЧАНИЕ Реализации типа Date в разных браузерах имеют много особенностей. Часто значения дат, не попадающие в допустимые диапазоны, заменяются правильными аналогами (например, некоторые браузеры интерпретируют строку «`January 32, 2019`» как «`February 1, 2019`»), тогда как Opera обычно подставляет текущий день текущего месяца, возвращая «`January текущий_день, 2019`». Иначе говоря, при обработке указанной даты 21 сентября будет возвращена строка «`January 21, 2019`».

Метод `Date.UTC()` также возвращает представление даты в миллисекундах, но создает его на основе других данных. Аргументами `Date.UTC()` являются год, месяц с отсчетом от нуля (январь — 0, февраль — 1 и т. д.), день месяца (от 1 до 31), часы (от 0 до 23), минуты, секунды и миллисекунды. Обязательны только первые два аргумента (год и месяц). Если не указан день месяца, предполагается, что он равен 1, тогда как все остальные опущенные аргументы считаются равными 0. Вот два примера использования метода `Date.UTC()`:

```
// 1 января 2000 г., полночь (GMT)
let y2k = new Date(Date.UTC(2000, 0));

// 5 мая 2005 г., 17:55:55 (GMT)
let allFives = new Date(Date.UTC(2005, 4, 5, 17, 55, 55));
```

В этих примерах создаются две даты. Первая — полночь по Гринвичу 1 января 2000 г., чему соответствуют год 2000 и месяц 0 (январь). Поскольку вместо остальных аргументов подставляются значения, предлагаемые по умолчанию (1 как день месяца и нули вместо всего остального), в результате получается полночь первого дня месяца. Вторая дата представляет 5 мая 2005 г., 17:55:55 по Гринвичу. Обратите внимание, что месяц задан числом 4, потому что месяцы отсчитываются от нуля. В остальных аргументах нет ничего необычного.

Конструктор `Date` поддерживает также формат метода `Date.UTC()`, но с одним важным отличием: он создает дату и время в локальном часовом поясе, а не по Гринвичу. Аргументы в него передаются такие же, что и в метод `Date.UTC()`. Если первым аргументом является число, конструктор `Date` предполагает, что это год даты, второй аргумент — месяц и т. д. Два предыдущих примера можно переписать следующим образом:

```
// 1 января 2000 г., полночь по локальному времени
let y2k = new Date(2000, 0);

// 5 мая 2005 г., 17:55:55 по локальному времени
let allFives = new Date(2005, 4, 5, 17, 55, 55);
```

Здесь создаются те же даты, что и в предыдущих примерах, но на этот раз они относятся к локальному часовому поясу, заданному в системных параметрах.

В ECMAScript также предложен метод `Date.now()`, который возвращает дату и время его выполнения в миллисекундах. Это позволяет использовать объекты `Date` для профилирования кода:

```
// получение времени начала
let start = Date.now();

// вызов функции
doSomething();

// получение времени окончания
let stop = Date.now(),
    result = stop – start;
```

Унаследованные методы

Как и другие ссылочные типы, тип `Date` переопределяет методы `toLocaleString()`, `toString()` и `valueOf()`, но у него эти методы возвращают разные значения. Метод `toLocaleString()` типа `Date` возвращает дату и время в региональном формате, заданном для браузера. Например, формат может включать обозначение AM или PM для часов и не содержать никаких сведений о часовом поясе (точный формат зависит от браузера). Метод `toString()` обычно возвращает дату и время со сведениями о часовом поясе, при этом время обычно указывается в 24-часовом формате (от 0 до 23). Далее показано, какое значение возвращают методы `toLocaleString()` и `toString()` для полуночи 1 февраля 2019 г. по стандартному тихоокеанскому времени в региональном формате «en-US»:

```
toLocaleString() – 2/1/2019 12:00:00 AM
toString() – Thu Feb 1 2019 00:00:00 GMT-0800 (Pacific Standard Time)
```

Современные браузеры объединились для вывода одинаковых строк для этих двух методов. При использовании устаревших браузеров форматы значений, возвращаемых этими методами в разных браузерах, различаются. По этой причине методы `toLocaleString()` и `toString()` полезны только для отладки, но не для вывода данных.

Метод `valueOf()` в типе `Date` переопределен и возвращает не строку, а представление даты в миллисекундах, чтобы операторы (такие как «меньше» и «больше») правильно работали с датами. Рассмотрим пример:

```
let date1 = new Date(2019, 0, 1);    // 1 января 2019 г.
let date2 = new Date(2019, 1, 1);    // 1 февраля 2019 г.

console.log(date1 < date2);    // true
console.log(date1 > date2);    // false
```

1 января 2019 г. наступило раньше, чем 1 февраля 2019 г., поэтому вполне можно сказать, что первая дата меньше второй. Поскольку в миллисекундах 1 января 2019 г. меньше, чем 1 февраля 2019 г., оператор «меньше» возвращает при их сравнении `true`, что позволяет легко упорядочивать даты.

Методы форматирования дат

Тип `Date` включает несколько методов, с помощью которых можно форматировать даты как строки.

- `toDateString()` — выводит день недели, месяц, день месяца и год в формате, зависящем от реализации;
- `toTimeString()` — выводит часы, минуты, секунды и часовой пояс в формате, зависящем от реализации;
- `toLocaleDateString()` — выводит день недели, месяц, день месяца и год в формате, зависящем от реализации и региональных параметров;
- `toLocaleTimeString()` — выводит часы, минуты и секунды в формате, зависящем от реализации;
- `toUTCString()` — выводит полную UTC-дату в формате, зависящем от реализации.

Как и в случае методов `toLocaleString()` и `toString()`, выводимые этими методами данные во многом зависят от браузера, поэтому их не следует использовать для вывода дат в пользовательском интерфейсе.

> **ПРИМЕЧАНИЕ** Есть также метод `toGMTString()`, который эквивалентен методу `toUTCString()` и предоставляется ради обратной совместимости. В спецификации рекомендуется использовать в новом коде только метод `toUTCString()`.

Методы для работы с компонентами даты/времени

Остальные методы типа `Date`, приведенные в таблице, получают и задают отдельные части даты и времени. Если в таблице упоминается стандарт UTC, это означает, что дата интерпретируется без смещения часового пояса (преобразуется в дату по Гринвичу).

МЕТОД	ОПИСАНИЕ
`getTime()`	Возвращает представление даты в миллисекундах; то же, что и `valueOf()`
`setTime` *(миллисекунды)*	Задает представление даты в миллисекундах, изменяя тем самым всю дату
`getFullYear()`	Возвращает четырехсимвольное значение года (2019, а не просто 19)
`getUTCFullYear()`	Возвращает четырехсимвольное значение года даты в формате UTC
`setFullYear` *(год)*	Задает год даты. Значение года должно содержать четыре цифры (2019, а не просто 19)

МЕТОД	ОПИСАНИЕ
`setUTCFullYear` *(год)*	Задает год даты в формате UTC. Значение года должно содержать четыре цифры (2019, а не просто 19)
`getMonth()`	Возвращает месяц даты (0 представляет январь, а 11 – декабрь)
`getUTCMonth()`	Возвращает месяц даты в формате UTC (0 представляет январь, а 11 – декабрь)
`setMonth` *(месяц)*	Задает месяц даты с отсчетом от 0. Указание числа больше 11 приводит к увеличению значения года
`setUTCMonth` *(месяц)*	Задает месяц даты в формате UTC с отсчетом от 0. Указание числа больше 11 приводит к увеличению значения года
`getDate()`	Возвращает день месяца даты (от 1 до 31)
`getUTCDate()`	Возвращает день месяца даты в формате UTC (от 1 до 31)
`setDate` *(дата)*	Задает день месяца для даты. Если значение даты больше, чем количество дней в месяце, значение месяца увеличивается
`setUTCDate` *(дата)*	Задает день месяца для даты в формате UTC. Если значение даты больше, чем количество дней в месяце, значение месяца увеличивается
`getDay()`	Возвращает день недели даты как число (0 представляет воскресенье, 6 – субботу)
`getUTCDay()`	Возвращает день недели даты в формате UTC как число (0 представляет воскресенье, 6 – субботу)
`getHours()`	Возвращает час даты как число от 0 до 23
`getUTCHours()`	Возвращает час даты в формате UTC как число от 0 до 23
`setHours` *(часы)*	Задает час даты. Указание числа больше 23 увеличивает значение дня месяца
`setUTCHours` *(часы)*	Задает час даты в формате UTC. Указание числа больше 23 увеличивает значение дня месяца
`getMinutes()`	Возвращает минуты даты как число от 0 до 59
`getUTCMinutes()`	Возвращает минуты даты в формате UTC как число от 0 до 59
`setMinutes` *(минуты)*	Задает минуты даты. Указание числа больше 59 увеличивает значение часа
`setUTCMinutes` *(минуты)*	Задает минуты даты в формате UTC. Указание числа больше 59 увеличивает значение часа
`getSeconds()`	Возвращает секунды даты как число от 0 до 59
`getUTCSeconds()`	Возвращает секунды даты в формате UTC как число от 0 до 59
`setSeconds` *(секунды)*	Задает секунды даты. Указание числа больше 59 увеличивает значение минут
`setUTCSeconds` *(секунды)*	Задает секунды даты в формате UTC. Указание числа больше 59 увеличивает значение минут

МЕТОД	ОПИСАНИЕ
`getMilliseconds()`	Возвращает миллисекунды даты
`getUTCMilliseconds()`	Возвращает миллисекунды даты в формате UTC
`setMilliseconds` *(миллисекунды)*	Задает миллисекунды даты
`setUTCMilliseconds` *(миллисекунды)*	Задает миллисекунды даты в формате UTC
`getTimeZoneOffset()`	Возвращает количество минут, на которое локальный часовой пояс отстоит от UTC. Например, для восточного стандартного времени возвращается число 300. При переходе региона на летнее время это значение меняется

ТИП REGEXP

Тип `RegExp` реализует в ECMAScript регулярные выражения, которые можно с легкостью создавать, используя синтаксис, похожий на Perl:

```
let выражение = /шаблон/флаги;
```

Шаблоном может быть регулярное выражение любой сложности, включающее классы символов, квантификаторы, группировки, предпросмотр и обратные ссылки. Каждое выражение может иметь или не иметь флаги, указывающие режим сопоставления. Поддерживаются следующие флаги:

- `g` — включает глобальный режим, в котором шаблон применяется ко всей строке, то есть поиск не прекращается после обнаружения первого совпадения;
- `i` — включает режим без учета регистра, в котором при поиске совпадений регистры шаблона и строки игнорируются;
- `m` — включает многострочный режим, в котором поиск совпадений продолжается после достижения конца одной строки текста;
- `y` — включает режим закрепления, то есть поиск совпадений учитывает только содержимое строки, начиная с `lastIndex`;
- `u` — включает режим Юникода.

Регулярное выражение создается путем объединения шаблона и флагов, например:

```
// Поиск всех экземпляров "at" в строке.
let pattern1 = /at/g;

// Поиск первого экземпляра "bat" или "cat" без учета регистра.
let pattern2 = /[bc]at/i;

/ Поиск всех трехсимвольных сочетаний, заканчивающихся на "at",
 * независимо от регистра.
 */
let pattern3 = /.at/gi;
```

Как и в других языках, все *метасимволы* (metacharacters) в шаблоне регулярного выражения нужно экранировать. Доступны следующие метасимволы:

```
(   [   {   \   ^   $   |   )   ]   }   ?   *   +   .
```

Метасимволы могут использоваться в регулярных выражениях несколькими способами, поэтому если требуется сопоставить метасимвол со строкой, его нужно экранировать обратной косой чертой, например:

```
// Поиск первого экземпляра "bat" или "cat" без учета регистра.
let pattern1 = /[bc]at/i;

// Поиск первого экземпляра "[bc]at" без учета регистра.
let pattern2 = /\[bc\]at/i;

/*
 * Поиск всех трехсимвольных сочетаний, заканчивающихся на "at",
 * без учета регистра.
 */
let pattern3 = /.at/gi;

// Поиск всех экземпляров ".at" без учета регистра.
let pattern4 = /\.at/gi;
```

В этом коде `pattern1` сопоставляется со всеми экземплярами `"bat"` и `"cat"` независимо от регистра. Чтобы выполнить сопоставление со строкой `"[bc]at"`, нужно экранировать обе квадратные скобки, как сделано в шаблоне `pattern2`. В шаблоне `pattern3` точка указывает, что символам `"at"` может предшествовать любой символ. Если требуется найти подстроку `".at"`, нужно экранировать точку, как в шаблоне `pattern4`.

Во всех предыдущих примерах регулярные выражения определены как литералы, но их также можно создавать с помощью конструктора `RegExp`. Он принимает два аргумента: строковый шаблон для сопоставления и необязательную строку флагов. Любое регулярное выражение, которое можно определить, используя синтаксис литералов, можно также создать с помощью конструктора, например:

```
// Поиск первого экземпляра "bat" или "cat" без учета регистра.
let pattern1 = /[bc]at/i;

// То же, что и pattern1, но с использованием конструктора.
let pattern2 = new RegExp("[bc]at", "i");
```

Здесь `pattern1` и `pattern2` определяют эквивалентные регулярные выражения. Обратите внимание, что оба аргумента конструктора `RegExp` являются строками (литералы регулярных выражений не следует передавать в конструктор `RegExp`). Поскольку шаблон передается в конструктор `RegExp` как строка, иногда его знаки могут требовать двойного экранирования. Это относится ко всем метасимволам, а также к знакам, которые уже экранированы, таким как `\n` (знак `\`, который обычно экранируется в строках как `\\`, повторяется в строке регулярного выражения четырежды: `\\\\`). В следующей таблице приведены некоторые шаблоны в форме литералов и эквивалентные строки для передачи в конструктор `RegExp`:

ЛИТЕРАЛ ШАБЛОНА	СТРОКОВЫЙ ЭКВИВАЛЕНТ
/\[bc\]at/	"\\[bc\\]at"
/\.at/	"\\.at"
/name\/age/	"name\\/age"
/\d.\d{1,2}/	"\\d.\\d{1,2}"
/\w\\hello\\123/	"\\w\\\\hello\\\\123"

Помните, что определить регулярное выражение на основе литерала и создать его с помощью конструктора `RegExp` — это не совсем одно и то же. В ECMAScript литералы регулярных выражений всегда относятся к одному экземпляру `RegExp`, тогда как вызов конструктора типа `RegExp` каждый раз создает его новый экземпляр. Рассмотрим следующий пример:

```
let re = null;
for (let i=0; i < 10; i++) {
    re = /cat/g;
    re.test("catastrophe");
}

for (let i=0; i < 10; i++) {
    re = new RegExp("cat", "g");
    re.test("catastrophe");
}
```

В первом цикле создается только один экземпляр `RegExp` для шаблона `/cat/`, несмотря на то что он указан в теле цикла. Свойства экземпляра (см. следующий раздел) не сбрасываются, из-за чего метод `test()` каждый второй раз не может обнаружить шаблон `/cat/` в строке. При первом вызове `test()` шаблон обнаруживается, но в следующей итерации поиск начинается с индекса 3 (конец первого совпадения) и завершается ничем. После достижения конца строки и перехода к следующей итерации метод `test()` снова начинает поиск с начала строки.

Во втором цикле используется конструктор `RegExp`, который на каждой итерации создает новое регулярное выражение, поэтому каждый вызов метода `test()` возвращает `true`.

В ECMAScript 5 для литералов регулярных выражений создаются новые экземпляры `RegExp`, как если бы непосредственно вызывался конструктор `RegExp`.

Также можно скопировать существующие экземпляры регулярного выражения и при необходимости изменить их флаги с помощью конструктора:

```
const re1 = /cat/g;
console.log(re1); // "/cat/g"

const re2 = new RegExp(re1);
console.log(re2); // "/cat/g"

const re3 = new RegExp(re1, "i");
console.log(re3); // "/cat/i"
```

Свойства экземпляра RegExp

У каждого экземпляра `RegExp` есть следующие свойства, позволяющие получить сведения о шаблоне:

- `global` — логическое значение, указывающее, задан ли флаг `g`;
- `ignoreCase` — логическое значение, указывающее, задан ли флаг `i`;
- `unicode` — логическое значение, указывающее, задан ли флаг `u`;
- `sticky` — логическое значение, указывающее, задан ли флаг `y`;
- `lastIndex` — целое число, указывающее позицию в исходной строке, где сопоставление будет выполнено в следующий раз (это значение всегда первоначально равно 0);
- `multiline` — логическое значение, указывающее, задан ли флаг `m`;
- `source` — исходная строка регулярного выражения, которая всегда возвращается в форме литерала (без открывающей и закрывающей косых черт), а не как строковый шаблон, переданный в конструктор;
- `flags` — строковые флаги регулярного выражения. Данное свойство всегда возвращается, как если бы оно было указано в буквальной форме (без открывающей и закрывающей косой черты), а не в виде строкового шаблона, переданного в конструктор.

С помощью этих свойств можно получить полезную информацию о регулярном выражении, но они используются редко, потому что эта информация доступна в объявлении шаблона, например:

```
let pattern1 = /\[bc\]at/i;

console.log(pattern1.global);        // false
console.log(pattern1.ignoreCase);    // true
console.log(pattern1.multiline);     // false
console.log(pattern1.lastIndex);     // 0
console.log(pattern1.source);        // "\[bc\]at"
console.log(pattern1.flags);         // "i"

let pattern2 = new RegExp("\\[bc\\]at", "i");

console.log(pattern2.global);        // false
console.log(pattern2.ignoreCase);    // true
console.log(pattern2.multiline);     // false
console.log(pattern2.lastIndex);     // 0
console.log(pattern2.source);        // "\[bc\]at"
console.log(pattern2.flags);         // "i"
```

Заметьте, что значения свойства `source` и `flags` у обоих шаблонов одинаковы, хотя первый представлен в формате литерала, а второй был передан в конструктор `RegExp`. Свойства `source` и `flags` форматируют строку как литерал.

Методы экземпляра RegExp

Главный метод объекта `RegExp` называется `exec()` и предназначен для работы с группами захвата. Он принимает в качестве единственного аргумента строку, к которой нужно применить шаблон, и возвращает массив со сведениями о первом совпадении или значение `null`, если совпадения отсутствуют. Возвращенный массив является экземпляром `Array`, но содержит два дополнительных свойства: `index` — место в строке, где было зарегистрировано совпадение с шаблоном, и `input` — исходная строка для сопоставления с шаблоном. Первым элементом массива является строка, соответствующая всему шаблону, а любые дополнительные элементы представляют захваченные группы в выражении (если в шаблоне нет групп захвата, массив содержит только один элемент). Рассмотрим следующий пример:

```
let text = "mom and dad and baby";
let pattern = /mom( and dad( and baby)?)?/gi;

let matches = pattern.exec(text);
console.log(matches.index);    // 0
console.log(matches.input);    // "mom and dad and baby"
console.log(matches[0]);       // "mom and dad and baby"
console.log(matches[1]);       // " and dad and baby"
console.log(matches[2]);       // " and baby"
```

В этом примере у шаблона две группы захвата. Внутренняя сопоставляется со строкой `" and baby"`, а охватывающая ее — со строкой `" and dad"` или `" and dad and baby"`. При вызове метода `exec()` для строки обнаруживается совпадение. Поскольку шаблону соответствует вся строка, свойству `index` массива `matches` присваивается значение 0. В первом элементе массива сохраняется вся сопоставленная строка, во втором — содержимое первой группы захвата, в третьем — содержимое второй группы захвата.

Метод `exec()` возвращает сведения об одном совпадении за раз, даже если шаблон глобален. Если флаг глобального поиска не указан, многократные вызовы метода `exec()` для одной и той же строки всегда возвращают сведения о первом совпадении.

```
let text = "cat, bat, sat, fat";
let pattern = /.at/;

let matches = pattern1.exec(text);
console.log(matches.index);         // 0
console.log(matches[0]);            // cat
console.log(pattern.lastIndex);     // 0

matches = pattern.exec(text);
console.log(matches.index);         // 0
console.log(matches[0]);            // cat
console.log(pattern.lastIndex);     // 0
```

Шаблон в этом примере не является глобальным, поэтому каждый вызов `exec()` возвращает только первое совпадение (`"cat"`). `lastIndex` остается неизменным в неглобальном режиме.

При установленном глобальном флаге `g` в шаблоне каждый вызов `exec()` перемещается дальше в строку в поисках совпадений, как в этом примере:

```
let text = "cat, bat, sat, fat";
let pattern = /.at/g;
let matches = pattern.exec(text);
console.log(matches.index);            // 0
console.log(matches[0]);               // cat
console.log(pattern.lastIndex);        // 3

matches = pattern.exec(text);
console.log(matches.index);            // 5
console.log(matches[0]);               // bat
console.log(pattern.lastIndex);        // 8

matches = pattern.exec(text);
console.log(matches.index);            // 10
console.log(matches[0]);               // sat
console.log(pattern.lastIndex);        // 13
```

Этот шаблон глобален, так что при каждом вызове `exec()` возвращается следующее совпадение в строке, пока она не заканчивается. Выбор режима также влияет на свойство `lastIndex` шаблона. В режиме глобального сопоставления значение `lastIndex` увеличивается после каждого вызова `exec()`. `lastIndex` отслеживает индекс символа, который сразу появляется справа от последнего совпадения.

Если на шаблоне установлен флаг закрепления `y`, каждый вызов `exec()` будет искать совпадение в строке только в `lastIndex` — и больше нигде. Флаг закрепления переопределяет глобальный флаг.

```
let text = "cat, bat, sat, fat";
let pattern = /.at/y;

let matches = pattern.exec(text);
console.log(matches.index);           // 0
console.log(matches[0]);              // cat
console.log(pattern.lastIndex);       // 3

// Нет совпадений, начиная с символьного индекса 3, поэтому exec() вернет null
// exec() при отсутствии совпадений сбрасывает lastIndex на 0
matches = pattern.exec(text);
console.log(matches);                 // null
console.log(pattern.lastIndex);       // 0

// Продвижение lastIndex позволит закрепленному регулярному выражению exec() найти
// следующее соответствие:
pattern.lastIndex = 5;
matches = pattern.exec(text);
console.log(matches.index);          // 5
console.log(matches[0]);             // bat
console.log(pattern.lastIndex);      // 8
```

Если требуется выяснить, соответствует ли строка шаблону, но текст совпадения не нужен, можно использовать метод `test()`. Он принимает строковый аргумент

и возвращает `true`, если шаблон соответствует аргументу, или `false` в противном случае. Метод `test()` часто используется в инструкциях `if`, например:

```
let text = "000-00-0000";
let pattern = /\d{3}-\d{2}-\d{4}/;

if (pattern.test(text)) {
    console.log("The pattern was matched.");  // Обнаружено соответствие шаблону
}
```

В этом примере регулярное выражение сопоставляется с последовательностью цифр. Если входной текст соответствует шаблону, выводится сообщение. Подобный код часто применяется для проверки введенных пользователем данных, когда вас интересует только то, допустимы эти данные или нет.

Унаследованные методы `toLocaleString()` и `toString()` возвращают литерал регулярного выражения независимо от того, как оно было создано, например:

```
let pattern = new RegExp("\\[bc\\]at", "gi");
console.log(pattern.toString());          // /\[bc\]at/gi
console.log(pattern.toLocaleString());    // /\[bc\]at/gi
```

Хотя шаблон в этом примере создается с помощью конструктора `RegExp`, методы `toLocaleString()` и `toString()` возвращают его, как если бы он был задан в формате литерала.

> **ПРИМЕЧАНИЕ** Метод `valueOf()` типа `RegExp` возвращает само регулярное выражение.

Свойства конструктора RegExp

У функции конструктора `RegExp` есть несколько свойств, указанных в приведенной далее таблице (в других языках они были бы статическими). Они применяются ко всем регулярным выражениям в области видимости и изменяются согласно последней операции с регулярным выражением. Эти свойства уникальны еще тем, что есть два способа доступа к ним: по полному и по сокращенному имени.

ПОЛНОЕ ИМЯ	СОКРАЩЕННОЕ ИМЯ	ОПИСАНИЕ
`input`	`$_`	Последняя строка, для которой выполнялось сопоставление
`lastMatch`	`$&`	Последний совпавший текст
`lastParen`	`$+`	Последняя совпавшая группа захвата
`leftContext`	`$\``	Текст в строке `input` перед `lastMatch`
`rightContext`	`$'`	Текст в строке `input` после `lastMatch`

Эти свойства можно использовать для извлечения сведений об операции, выполненной методом `exec()` или `test()`, например:

```
let text = "this has been a short summer";
let pattern = /(.)hort/g;

if (pattern.test(text)) {
    console.log(RegExp.input);          // this has been a short summer
    console.log(RegExp.leftContext);    // this has been a
    console.log(RegExp.rightContext);   // summer
    console.log(RegExp.lastMatch);      // short
    console.log(RegExp.lastParen);      // s
    console.log(RegExp.multiline);      // false
}
```

В этом коде создается шаблон, который ищет любой знак, предшествующий строке `"hort"`, и определяет группу захвата для первой буквы. Со свойствами при этом происходит следующее:

- свойство `input` содержит исходную строку;
- свойство `leftContext` содержит символы строки до слова `"short"`, а свойство `rightContext` — после слова `"short"`;
- свойство `lastMatch` содержит последнюю строку, которая соответствует всему регулярному выражению, или `"short"`;
- свойство `lastParen` содержит последнюю совпавшую группу захвата, или `"s"` в данном случае.

Полные имена свойств можно заменить сокращенными, но для доступа к ним нужно использовать квадратные скобки, потому что большинство из них не являются допустимыми ECMAScript-идентификаторами:

```
let text = "this has been a short summer";
let pattern = /(.)hort/g;

/*
 * Примечание: Opera не поддерживает сокращенные имена свойств.
 * Internet Explorer не поддерживает свойство multiline.
 */
if (pattern.test(text)) {
    console.log(RegExp.$_);         // this has been a short summer
    console.log(RegExp["$`"]);      // this has been a
    console.log(RegExp["$'"]);      //  summer
    console.log(RegExp["$&"]);      // short
    console.log(RegExp["$+"]);      // s
    console.log(RegExp["$*"]);      // false
}
```

У конструктора также есть свойства, хранящие до девяти совпадений с группами захвата. Они имеют имена с `RegExp.$1` (первое совпадение) по `RegExp.$9` (девятое совпадение). Эти свойства заполняются при вызове метода `exec()` или `test()`, что позволяет написать такой код:

```
let text = "this has been a short summer";
let pattern = /(..)or(.)/g;

if (pattern.test(text)) {
    console.log(RegExp.$1);    // sh
    console.log(RegExp.$2);    // t
}
```

В этом примере создается шаблон с двумя группами сопоставления, который затем применяется к строке. Хотя метод `test()` просто возвращает логическое значение, при этом также заполняются свойства `$1` и `$2` конструктора `RegExp`.

> **ПРИМЕЧАНИЕ** Все эти свойства конструктора `RegExp` не являются частью какого-либо веб-стандарта; избегайте использования в любом реальном приложении.

Ограничения шаблонов

В целом поддержка регулярных выражений в языке ECMAScript очень хороша, но в нем нет некоторых нетривиальных возможностей, доступных в таких языках, как Perl. Например, следующие функциональные возможности в ECMAScript не поддерживаются (дополнительные сведения см. на сайте www.regularexpressions.info):

- якоря `\A` и `\Z` (начало и конец строки соответственно);
- обратный просмотр;
- классы объединения и пересечения;
- атомарные группировки;
- поддержка Юникода (исключая сопоставление с одним символом за раз);
- именованные группы захвата;
- режимы сопоставления `s` (однострочный) и `x` (с пробельными символами);
- условия;
- комментарии в регулярных выражениях.

Несмотря на эти ограничения, имеющихся в ECMAScript возможностей достаточно для решения большинства задач сопоставления с шаблонами.

ОБОЛОЧКИ ПРИМИТИВНЫХ ТИПОВ

Специальные ссылочные типы `Boolean`, `Number` и `String` упрощают работу с соответствующими примитивными значениями. Они поддерживают стандартные возможности ссылочных типов, но имеют также специальные формы поведения, связанные с их примитивными аналогами. Каждый раз при чтении примитивного значения неявно создается соответствующий объект оболочки примитивного

типа, обеспечивающий доступ к методам манипулирования данными. Рассмотрим следующий пример:

```
let s1 = "some text";    // какой-то текст
let s2 = s1.substring(2);
```

Здесь переменной `s1` присваивается примитивное строковое значение, а затем для нее вызывается метод `substring()` и результат сохраняется в `s2`. Примитивные значения не являются объектами, поэтому, по идее, у них не должно быть методов, но этот код работает, как предполагалось. Чтобы это стало возможным, запускаются некоторые скрытые процессы. Доступ к `s1` во второй строке осуществляется в режиме чтения, то есть это значение читается из памяти. Всякий раз, когда строковое значение используется в режиме чтения, происходит трехэтапная процедура:

1. Создание экземпляра типа `String`.
2. Вызов указанного метода для экземпляра.
3. Уничтожение экземпляра.

Эти действия можно представить следующими тремя строками ECMAScript-кода:

```
let s1 = new String("some text");
let s2 = s1.substring(2);
s1 = null;
```

Благодаря такому поведению примитивное строковое значение может работать как объект. Те же три этапа выполняются для логических и числовых значений с типами `Boolean` и `Number` соответственно.

Главное различие между ссылочными типами и оболочками примитивных типов — время жизни объекта. Экземпляр ссылочного типа, созданный с помощью оператора `new`, остается в памяти до тех пор, пока не выходит из области видимости, тогда как автоматически создаваемые оболочки примитивных типов существуют только одну строку кода, после чего уничтожаются. Это означает, что к ним невозможно добавить свойства и методы во время выполнения. Взгляните на этот пример:

```
let s1 = "some text;
s1.color = "red";
console.log(s1.color);    // undefined
```

Здесь во второй строке предпринимается попытка добавить к строке `s1` свойство `color`, но уже в следующей строке оно недоступно. Это происходит потому, что объект `String`, созданный во второй строке, уничтожается к моменту выполнения третьей строки. В третьей строке создается другой объект `String`, у которого нет свойства `color`.

Оболочки примитивных типов можно создавать явно с помощью конструкторов `Boolean`, `Number` и `String`, но делать это без необходимости не следует, потому что иначе многим разработчикам будет не совсем понятно, с каким значением они имеют дело, примитивным или ссылочным. Все оболочки примитивных типов преобразуются в логическое значение `true`, а вызов оператора `typeof` для них возвращает `"object"`.

Конструктор `Object` может работать как фабричный метод, возвращая экземпляр оболочки примитивного типа на основе типа полученного значения, например:

```
let obj = new Object("some text");
console.log(obj instanceof String);  // true
```

При передаче строки в конструктор `Object` создается экземпляр `String`, для числового аргумента возвращается экземпляр `Number`, а для логического — экземпляр `Boolean`.

Имейте в виду, что вызов конструктора оболочки примитивного типа с помощью оператора `new` — это не то же самое, что вызов функции приведения типов с тем же именем:

```
let value = "25";
let number = Number(value);     // функция приведения типов
console.log(typeof number);     // "number"

let obj = new Number(value);    // конструктор
console.log(typeof obj);        // "object"
```

В этом примере в переменной `number` сохраняется примитивное числовое значение 25, а переменной `obj` присваивается экземпляр `Number`.

Хотя явно создавать оболочки примитивных типов не рекомендуется, при работе с примитивными значениями они играют важную роль. У оболочки каждого типа есть методы, которые упрощают работу с данными.

Тип Boolean

`Boolean` — это ссылочный тип, соответствующий булевым значениям. Чтобы создать объект `Boolean`, вызовите конструктор `Boolean`, передав ему значение `true` или `false`, например:

```
let booleanObject = new Boolean(true);
```

Экземпляры `Boolean` переопределяют метод `valueOf()`, чтобы он возвращал примитивное значение `true` или `false`. Метод `toString()` также переопределяется и возвращает строку `"true"` или `"false"`. К сожалению, объекты `Boolean` не только практически бесполезны в ECMAScript, но и затрудняют понимание кода. Проблемы обычно возникают при использовании объектов `Boolean` в логических выражениях, например:

```
let falseObject = new Boolean(false);
let result = falseObject && true;
console.log(result);    // true

let falseValue = false;
result = falseValue && true;
console.log(result);    // false
```

В этом коде создается объект `Boolean` со значением `false`, а затем для него и примитивного значения `true` выполняется операция И. В булевой математике результатом

операции false И true является false, однако в этой строке кода оценивается объект falseObject, а не его значение (false). Как уже отмечалось, в логических выражениях все объекты автоматически преобразуются в true, так что вместо falseObject на самом деле используется значение true. Применение оператора И к двум значениям true дает в результате true.

Есть и другие различия между примитивным и ссылочным логическими типами. Так, оператор typeof возвращает "boolean" для примитивного типа, но "object" для ссылочного. Кроме того, объект Boolean является экземпляром типа Boolean, поэтому оператор instanceof возвращает для него true, а для примитивного значения — false:

```
console.log(typeof falseObject);  // object
console.log(typeof falseValue);   // boolean
console.log(falseObject instanceof Boolean);  // true
console.log(falseValue instanceof Boolean);   // false
```

Важно понимать различия между примитивным логическим значением и объектом Boolean и по возможности не использовать последний.

Тип Number

Тип Number — это ссылочный тип для числовых значений. Чтобы создать объект Number, вызовите конструктор Number, передав в него любое число, например:

```
let numberObject = new Number(10);
```

Как и Boolean, тип Number переопределяет методы valueOf(), toLocaleString() и toString(). Метод valueOf() возвращает примитивное числовое значение, представленное объектом, а другие два метода возвращают число как строку. Метод toString() может принимать аргумент, указывающий основание системы счисления:

```
let num = 10;
console.log(num.toString());     // "10"
console.log(num.toString(2));    // "1010"
console.log(num.toString(8));    // "12"
console.log(num.toString(10));   // "10"
console.log(num.toString(16));   // "a"
```

Кроме унаследованных методов у типа Number есть несколько дополнительных методов, позволяющих форматировать числа как строки.

Метод toFixed() возвращает число как строку с указанным количеством знаков после точки:

```
let num = 10;
console.log(num.toFixed(2));    // "10.00"
```

Здесь методу toFixed() передается аргумент 2, поэтому он возвращает строку "10.00", заменяя отсутствующую дробную часть двумя нулями. Если число содержит после точки больше знаков, чем указывает аргумент, результат округляется до ближайшего разряда, например:

```
let num = 10.005;
console.log(num.toFixed(2));    // "10.01"
```

Округление с помощью метода `toFixed()` может быть полезно в приложениях, работающих с денежными суммами, но важно отметить, что арифметические операции между несколькими значениями с плавающей запятой могут не давать точных результатов, например, `0,1 + 0,2 = 0,30000000000000004`.

> **ПРИМЕЧАНИЕ** Метод `toFixed()` может представлять числа, содержащие от 0 до 20 разрядов после точки. Этот диапазон типичен для разных реализаций, хотя в некоторых браузерах он может быть шире.

С форматированием чисел связан также метод `toExponential()`, который возвращает строку с числом в экспоненциальной записи. Как и предыдущий метод, он принимает один аргумент — количество выводимых знаков после точки:

```
let num = 10;
console.log(num.toExponential(1));  // "1.0e+1"
```

Этот код выводит строку `"1.0e+1"`, хотя обычно для столь малых чисел экспоненциальная запись не используется. Если вам нужна более уместная форма записи, следует задействовать метод `toPrecision()`.

Метод `toPrecision()` возвращает строковое представление числа с фиксированным количеством знаков или в экспоненциальной записи в зависимости от того, в чем больше смысла. В качестве единственного аргумента он принимает общее количество разрядов итогового числа (не включая степень), например:

```
let num = 99;
console.log(num.toPrecision(1));  // "1e+2"
console.log(num.toPrecision(2));  // "99"
console.log(num.toPrecision(3));  // "99.0"
```

В этом примере в первой строке число 99 выводится с одним разрядом как `"1e+2"`, или 100. Поскольку 99 не может быть точно представлено с помощью одного разряда, оно округляется до 100. С двумя разрядами число 99 выводится как `"99"`, а с тремя — как `"99.0"`. По сути, метод `toPrecision()` на основе полученного числового значения выбирает и вызывает метод `toFixed()` или `toExponential()`; все три метода округляют значение вверх или вниз для точного представления числа с правильным количеством разрядов.

> **ПРИМЕЧАНИЕ** Метод `toPrecision()` может представлять числа, содержащие от 1 до 21 десятичных разрядов. Этот диапазон типичен для разных реализаций, хотя в некоторых браузерах он может быть шире.

Подобно `Boolean`, объекты `Number` обеспечивают важный функционал для работы с числовыми значениями, но их не следует создавать непосредственно из-за тех же

потенциальных проблем. Операторы `typeof` и `instanceof` работают с примитивными и ссылочными числами по-разному:

```
let numberObject = new Number(10);
let numberValue = 10;
console.log(typeof numberObject);       // "object"
console.log(typeof numberValue);        // "number"
console.log(numberObject instanceof Number);  // true
console.log(numberValue instanceof Number);   // false
```

Для примитивных чисел оператор `typeof` всегда возвращает `"number"`, а для объектов `Number` — `"object"`. Соответственно, объект `Number` является экземпляром типа `Number`, а примитивное число — нет.

Метод isInteger() и безопасные целые числа

Недавно введенный в ES6 метод `Number.isInteger()` определяет, хранится ли числовое значение как целое число или нет. Это полезно, когда конечный десятичный 0 может вводить в заблуждение — сохраняется ли число в формате с плавающей запятой или нет:

```
console.log(Number.isInteger(1));       // true
console.log(Number.isInteger(1.00));    // true
console.log(Number.isInteger(1.01));    // false
```

Числовой формат IEEE 754 имеет отдельный числовой диапазон, внутри которого двоичное значение может представлять ровно одно целочисленное значение. Этот числовой диапазон простирается от `Number.MIN_SAFE_INTEGER` или -2 ^ 53 + 1 до `Number.MAX_SAFE_INTEGER` или 2 ^ 53 - 1. За пределами этого диапазона вы можете попытаться сохранить целое число, но формат кодирования IEEE 754 означает, что это двоичное значение также может иметь псевдоним совершенно другого числа. Чтобы определить, находится ли целое число внутри этого диапазона, используйте метод `Number.isSafeInteger()`:

```
console.log(Number.isSafeInteger(-1 * (2 ** 53)));        // false
console.log(Number.isSafeInteger(-1 * (2 ** 53) + 1));    // true

console.log(Number.isSafeInteger(2 ** 53));               // false
console.log(Number.isSafeInteger((2 ** 53) - 1));         // true
```

Тип String

Тип `String` — это ссылочный аналог строк. Объекты этого типа создаются с помощью конструктора `String`:

```
let stringObject = new String("hello world");
```

Методы объекта `String` доступны для всех строковых примитивов. Все три унаследованных метода — `valueOf()`, `toLocaleString()` и `toString()` — возвращают примитивное строковое значение объекта.

Каждый экземпляр `String` содержит единственное свойство `length`, в котором хранится количество знаков в строке, например:

```
let stringValue = "hello world";
console.log(stringValue.length);    // "11"
```

В этом примере выводится строка `"11"`, отражающая количество знаков в строке `"hello world"`. Имейте в виду, что даже если строка содержит двухбайтовые символы, а не только однобайтовые ASCII-символы, каждый символ все равно считается за один.

Тип `String` содержит целый ряд методов, помогающих выполнять самые разные действия со строками.

Символ в JavaScript

Строки в JavaScript состоят из 16-битных единиц кода. Для большинства символов каждая 16-битная единица кода будет соответствовать одному символу. Свойство `length` указывает на количество 16-битных кодовых единиц внутри строки:

```
let message = "abcde";

console.log(message.length); // 5
```

Кроме того, `charAt()` возвращает символ по заданному индексу, указанному целочисленным аргументом метода. В частности, этот метод находит 16-битную единицу кода по указанному индексу и возвращает символ, соответствующий этой единице кода:

```
let message = "abcde";

console.log(message.charAt(2)); // "c"
```

Строки в JavaScript используют гибрид двух кодировок Юникода: UCS-2 и UTF-16. Для символов, которые могут быть закодированы 16 битами (от U+0000 до U+FFFF), эти два кодирования фактически идентичны.

> **ПРИМЕЧАНИЕ** Для всестороннего освещения кодировки символов проверьте превосходное сообщение в блоге Джоэла Спольски: https://www.joelonsoftware.com/2003/10/08/theabsolute-minimum-every-software-developer-absolutely-positively-mustknow-about-unicode-and-character-sets-no-excuses/.
>
> Другим хорошим ресурсом является сообщение в блоге Матиаса Биненса: https://mathiasbynens.be/notes/javascript-encoding.

Можно проверить кодировку символов данного блока кода с помощью метода `charCodeAt()`. Этот метод возвращает значение единицы кода по заданному индексу (целочисленному аргументу метода). Вот пример:

```
let message = "abcde";

// "Латинская строчная буква C" в Юникоде — U + 0063
console.log(message.charCodeAt(2)); // 99
```

```
// Десятичное 99 === шестнадцатеричное 63
console.log(99 === 0x63); // true
```

Метод `fromCharCode()` используется для создания символов в строке из их представления в качестве единиц кода в UTF-16. Этот метод принимает любое количество чисел и возвращает их символьные эквиваленты, объединенные в строку:

```
// "Латинская маленькая буква A" в Юникоде — U+0061
// "Латинская маленькая буква B" в Юникоде — U+0062
// "Латинская маленькая буква C" в Юникоде — U+0063
// "Латинская маленькая буква D" в Юникоде — U+0064
// "Латинская маленькая буква E" в Юникоде — U+0065

console.log(String.fromCharCode(0x61, 0x62, 0x63, 0x64, 0x65)); // "abcde"

// 0x0061 === 97
// 0x0062 === 98
// 0x0063 === 99
// 0x0064 === 100
// 0x0065 === 101

console.log(String.fromCharCode(97, 98, 99, 100, 101)); // "abcde"
```

Для символов в диапазоне от U+0000 до U+FFFF `length`, `charAt()`, `charCodeAt()` и `fromCharCode()` ведут себя точно так, как ожидается, потому, что каждый символ представлен ровно 16 битами и каждый из этих методов работает на 16-битных единицах кода. Пока есть четность между размером кодировки символов и размером кодовой единицы, эти методы будут вести себя ожидаемо.

Этот паритет нарушается при расширении в область дополнительных символьных плоскостей Юникода. Идея этой концепции относительно проста: 16 битов могут однозначно представлять только 65 536 символов. Этого достаточно, чтобы охватить большинство языковых наборов символов, и данный набор называется *Базовой многоязычной плоскостью* (Basic Multilingual Plane, BMP). Чтобы ввести еще больше символов, Юникод определил стратегию, которая использовала дополнительные 16 бит на символ для выбора *дополнительной*, или *астральной*, *плоскости*. Использование двух 16-битных кодовых единиц на символ называется *суррогатной парой*.

С введением этого соглашения ранее обсужденные строковые методы начинают разрушаться. Рассмотрим следующий пример, в котором используется смайлик — символ, закодированный с помощью суррогатной пары:

```
// Код смайла "улыбающееся лицо с глазами" — U+1F60A
// 0x1F60A === 128522
let message = "ab😊de";

console.log(message.length); // 6

console.log(message.charAt(1)); // b
console.log(message.charAt(2)); // <?>
console.log(message.charAt(3)); // <?>
console.log(message.charAt(4)); // d
```

```
console.log(message.charCodeAt(1)); // 98
console.log(message.charCodeAt(2)); // 55357
console.log(message.charCodeAt(3)); // 56842
console.log(message.charCodeAt(4)); // 100

console.log(String.fromCharCode(0x1F60A)); // ☺

console.log(String.fromCharCode(97, 98, 55357, 56842, 100, 101)); // ab☺de
```

Эти методы по-прежнему обрабатывают каждую 16-битную единицу кода как отдельный символ, когда фактически единицы кода в индексах 2 и 3 необходимо рассматривать вместе как одну суррогатную пару, чтобы сформировать один символ. Метод `fromCharCode()` по-прежнему работает правильно, используя две отдельные единицы кода, потому что этот метод буквально собирает строку из предоставленного двоичного представления. Браузер может правильно анализировать суррогатную пару (которая была собрана как две отдельные единицы кода) и правильно интерпретировать ее как один символ Юникода со смайликом.

Чтобы правильно проанализировать строку, содержащую как единичный код, так и символы суррогатной пары, можно использовать метод `codePointAt()` вместо подверженного ошибкам `charAt()`. Как и в случае с `charAt()`, данный метод принимает 16-битный индекс единицы кода и возвращает кодовую точку с этим индексом. *Кодовая точка* относится к полному идентификатору Юникода для одного символа. Кодовая точка для «c» — 0x0063. Кодовая точка для «☺» — 0x1F60A. Для полного представления кодовых точек может потребоваться 16 или 32 бита, а метод `codePointAt()` идентифицирует полную кодовую точку, начиная с указанной единицы кода.

```
let message = "ab☺de";

console.log(message.codePointAt(1)); // 98
console.log(message.codePointAt(2)); // 128522
console.log(message.codePointAt(3)); // 56842
console.log(message.codePointAt(4)); // 100
```

Обратите внимание, что в этом примере кодовая точка может быть неверно идентифицирована, если она нацелена на индекс единицы кода, не являющийся началом суррогатной пары. Это проблематично только для одноразовой проверки символов, и это можно предотвратить, обойдя строку слева направо и увеличив правильное количество единиц кода на итератор. Итератор для строки достаточно умен, чтобы идентифицировать кодовые точки суррогатной пары:

```
console.log([..."ab☺de"]); // ["a", "b", "☺", "d", "e"]
```

Подобно тому как `charAt()` имеет аналог `codePointAt()`, у `fromCharCode()` есть аналог `fromCodePoint()`. Этот метод принимает любое количество номеров кодовых точек и возвращает их символьные эквиваленты, объединенные в строку:

```
console.log(String.fromCharCode(97, 98, 55357, 56842, 100, 101)); // ab☺de
console.log(String.fromCodePoint(97, 98, 128522, 100, 101)); // ab☺de
```

Метод normalize()

Некоторые символы Юникода могут быть закодированы несколькими способами. Иногда символ может быть представлен либо одним символом BMP, либо суррогатной парой. Для примера рассмотрим следующее:

```
// U+00C5: Латинская заглавная буква A с кольцом над ней
console.log(String.fromCharCode(0x00C5)); // Å

// U+212B: Знак ангстрема
console.log(String.fromCharCode(0x212B)); // Å

// U+0041: Латинская заглавная буква A
// U+030A: Комбинированное кольцо над ней
console.log(String.fromCharCode(0x0041, 0x030A)); // Å
```

Операторы сравнения не заботятся о внешнем виде символов, и поэтому три символа ниже будут считаться различными:

```
let a1 = String.fromCharCode(0x00C5),
    a2 = String.fromCharCode(0x212B),
    a3 = String.fromCharCode(0x0041, 0x030A);

console.log(a1, a2, a3); // Å, Å, Å

console.log(a1 === a2); // false
console.log(a1 === a3); // false
console.log(a2 === a3); // false
```

Юникод объясняет это, предлагая четыре формы нормализации, с помощью которых символы, подобные этому, можно нормализовать в согласованный формат независимо от происхождения их кодов символов. Эти четыре формы нормализации — форма нормализации D (NFD), форма нормализации C (NFC), форма нормализации KD (NFKD) — и форма нормализации KC (NFKC), могут быть применены к строке с помощью метода `normalize()`. Этот метод должен быть снабжен строковым идентификатором, чтобы указать, какую форму нормализации применять: `NFD`, `NFC`, `NFKD` или `NFKC`.

> **ПРИМЕЧАНИЕ** Специфика каждой из этих нормальных форм выходит за рамки этого текста. Обратитесь к http://unicode.org/reports/tr15/ для получения дополнительной информации.

Можно определить, нормализована ли уже строка, проверив ее по возвращаемому значению `normalize()`:

```
let a1 = String.fromCharCode(0x00C5),
    a2 = String.fromCharCode(0x212B),
    a3 = String.fromCharCode(0x0041, 0x030A);

// U+00C5 – нормализованная форма NFC/NFKC для 0+212B
console.log(a1 === a1.normalize("NFD"));     // false
```

```
console.log(a1 === a1.normalize("NFC"));     // true
console.log(a1 === a1.normalize("NFKD"));    // false
console.log(a1 === a1.normalize("NFKC"));    // true

// U+212B не нормализован
console.log(a2 === a2.normalize("NFD"));     // false
console.log(a2 === a2.normalize("NFC"));     // false
console.log(a2 === a2.normalize("NFKD"));    // false
console.log(a2 === a2.normalize("NFKC"));    // false

// U+0041/U+030A — нормализованная форма NFD/NFKD для 0+212B
console.log(a3 === a3.normalize("NFD"));     // true
console.log(a3 === a3.normalize("NFC"));     // false
console.log(a3 === a3.normalize("NFKD"));    // true
console.log(a3 === a3.normalize("NFKC"));    // false
```

Выбор нормальной формы позволит оператору сравнения вести себя ожидаемо при проверке одинаковых символов:

```
let a1 = String.fromCharCode(0x00C5),
    a2 = String.fromCharCode(0x212B),
    a3 = String.fromCharCode(0x0041, 0x030A);

console.log(a1.normalize("NFD") === a2.normalize("NFD"));      // true
console.log(a2.normalize("NFKC") === a3.normalize("NFKC"));    // true
console.log(a1.normalize("NFC") === a3.normalize("NFC"));      // true
```

Методы манипулирования строками

Для работы со строками можно использовать несколько методов. Первый из них, `concat()`, присоединяет одну или несколько строк к другой и возвращает объединенную строку:

```
let stringValue = "hello ";
let result = stringValue.concat("world");

console.log(result);         // "hello world"
console.log(stringValue);    // "hello"
```

В этом примере вызов метода `concat()` для `stringValue` возвращает строку `"hello world"`, а значение `stringValue` остается без изменений. Метод `concat()` принимает любое количество аргументов, что позволяет составить строку из любого количества других строк:

```
let stringValue = "hello ";
let result = stringValue.concat("world", "!");

console.log(result);         // "hello world!"
console.log(stringValue);    // "hello"
```

В этом измененном примере к строке `"hello"` присоединяются строки `"world"` и `"!"`. Хотя для конкатенации строк предоставляется метод `concat()`, оператор сложения (`+`) используется чаще и в большинстве случаев работает быстрее, даже если строк несколько.

Для создания строковых значений из подстрок в ECMAScript применяются методы `slice()`, `substr()` и `substring()`. Все они возвращают подстроку строки, для которой были вызваны, и принимают один или два аргумента. Первым аргументом является позиция, с которой начинается захват подстроки, а второй, если он используется, указывает, когда нужно остановиться. У методов `slice()` и `substring()` второй аргумент определяет позицию, перед которой операция завершается, а у метода `substr()` — количество возвращаемых символов. Если второй аргумент опущен в каком-либо методе, операция выполняется до конца строки. Как и метод `concat()`, методы `slice()`, `substr()` и `substring()` не изменяют саму строку — они просто возвращают примитивное строковое значение, оставляя оригинал неизменным. Рассмотрим следующий пример:

```
let stringValue = "hello world";
console.log(stringValue.slice(3));        // "lo world"
console.log(stringValue.substring(3));    // "lo world"
console.log(stringValue.substr(3));       // "lo world"
console.log(stringValue.slice(3, 7));     // "lo w"
console.log(stringValue.substring(3,7));  // "lo w"
console.log(stringValue.substr(3, 7));    // "lo worl"
```

В этом примере методы `slice()`, `substr()` и `substring()` используются аналогичным образом и в основном возвращают одинаковые значения. При единственном аргументе `3` все они возвращают строку `"lo world"`, потому что в позиции 3 находится вторая буква `"l"` в слове `"hello"`. При двух аргументах `3` и `7` методы `slice()` и `substring()` возвращают `"lo w"` (буква `"o"` в слове `"world"` находится в позиции 7 и не включается в результат), а метод `substr()` — `"lo worl"`, потому что в его случае второй аргумент указывает количество возвращаемых символов.

Если аргументом является отрицательное число, эти методы работают иначе. Метод `slice()` вычитает его из длины строки. Метод `substr()` вычитает первый отрицательный аргумент из длины строки, а второй преобразует в 0. Метод `substring()` оба отрицательных аргумента преобразует в 0, например:

```
let stringValue = "hello world";
console.log(stringValue.slice(-3));         // "rld"
console.log(stringValue.substring(-3));     // "hello world"
console.log(stringValue.substr(-3));        // "rld"
console.log(stringValue.slice(3, -4));      // "lo w"
console.log(stringValue.substring(3, -4));  // "hel"
console.log(stringValue.substr(3, -4));     // "" (пустая строка)
```

Этот пример ясно демонстрирует различия между тремя методами. Когда методы `slice()` и `substr()` вызываются с единственным отрицательным аргументом, они работают одинаково, потому что –3 преобразуется в 7 (длина строки плюс аргумент). По сути, это преобразует вызовы в `slice(7)` и `substr(7)`. Что касается метода `substring()`, то он возвращает всю строку, потому что –3 преобразуется в 0.

Если отрицателен второй аргумент, эти три метода работают по-разному. Метод `slice()` преобразует второй аргумент в 7, по сути, изменяя вызов на `slice(3, 7)`, и возвращает `"lo w"`. Для метода `substring()` второй аргумент преобразуется в 0,

превращая вызов метода в `substring(3, 0)`, который на самом деле эквивалентен `substring(0, 3)`, потому что этот метод интерпретирует меньшее число как начальную позицию, а большее — как конечную. Для метода `substr()` второй аргумент также преобразуется в 0; это означает, что в возвращенной строке должно быть 0 знаков, поэтому возвращается пустая строка.

Методы поиска строк

Есть два метода поиска подстрок в других строках: `indexOf()` и `lastIndexOf()`. Оба метода ищут в строке конкретную подстроку и возвращают ее позицию (или –1, если найти ее не удается). Разница между ними в том, что метод `indexOf()` начинает искать подстроку с начала строки, а `lastIndexOf()` — с конца. Рассмотрим пример:

```
let stringValue = "hello world";
console.log(stringValue.indexOf("o"));      // 4
console.log(stringValue.lastIndexOf("o"));  // 7
```

Здесь первая подстрока `"o"` встречается в позиции 4, это буква `"o"` в слове `"hello"`. Последнее вхождение подстроки `"o"` в строку имеет место в позиции 7 в слове `"world"`. Если бы в строке содержалась только одна буква `"o"`, методы `indexOf()` и `lastIndexOf()` вернули бы одно и то же значение.

Оба метода могут принимать необязательный второй аргумент, указывающий начальную позицию поиска в строке. Иначе говоря, метод `indexOf()` выполняет поиск с этой позиции до конца строки, пропуская все символы перед начальной позицией, а метод `lastIndexOf()` начинает поиск с указанной позиции и продвигается к началу строки, пропуская символы между указанной позицией и концом строки:

```
let stringValue = "hello world";
console.log(stringValue.indexOf("o", 6));        // 7
console.log(stringValue.lastIndexOf("o", 6));    // 4
```

Как видите, если в каждый метод передать второй аргумент 6, возвращаются результаты, противоположные предыдущим. На этот раз метод `indexOf()` возвращает 7, потому что он начинает поиск подстроки с позиции 6 (буква `"w"`) и обнаруживает букву `"o"` в позиции 7. Метод `lastIndexOf()` возвращает 4, потому что он начинает поиск с позиции 6 и продвигается к началу строки, пока не встречает букву `"o"` в слове `"hello"`. С помощью второго аргумента можно найти все экземпляры подстроки в строке, циклически вызывая метод `indexOf()` или `lastIndexOf()`:

```
let stringValue = "Lorem ipsum dolor sit amet, consectetur adipisicing...";
let positions = new Array();
let pos = stringValue.indexOf("e");

while(pos > -1) {
    positions.push(pos);
    pos = stringValue.indexOf("e", pos + 1);
}

console.log(positions);    // "3,24,32,35"
```

Этот фрагмент обрабатывает строку, постоянно увеличивая позицию, в которой метод `indexOf()` начинает поиск. Сначала определяется позиция первой подстроки `"e"` в строке, а затем запускается цикл, в котором в метод `indexOf()` каждый раз передается позиция последней обнаруженной буквы `"e"`, увеличенная на 1. Благодаря этому поиск продолжается после обнаружения каждой подстроки. Позиции подстрок сохраняются в массиве `positions`, чтобы эти данные можно было использовать позже.

Методы включения строк

В ECMAScript 6 добавлены три дополнительных метода для определения того, включена ли строка в другую строку: `startWith()`, `endWith()` и `includes()`. Все методы ищут строку для данной подстроки и возвращают логическое значение, указывающее, включено ли оно или нет. Разница между ними заключается в том, что `beginWith()` проверяет совпадение, начинающееся с индекса 0, `endWith()` проверяет совпадение, начинающееся с индекса `(string.length - substring.length)`, и `includes()` проверяет всю строку.

```
let message = "foobarbaz";

console.log(message.startsWith("foo"));    // true
console.log(message.startsWith("bar"));    // false

console.log(message.endsWith("baz"));      // true
console.log(message.endsWith("bar"));      // false

console.log(message.includes("bar"));      // true
console.log(message.includes("qux"));      // false
```

Методы `startWith()` и `includes()` принимают необязательный второй аргумент, который указывает позицию для начала поиска внутри строки. Это означает, что методы начнут поиск с этой позиции и пойдут к концу строки, игнорируя все до начальной позиции. Вот пример:

```
let message = "foobarbaz";

console.log(message.startsWith("foo"));       // true
console.log(message.startsWith("foo", 1));    // false

console.log(message.includes("bar"));         // true
console.log(message.includes("bar", 4));      // false
```

Метод `endWith()` принимает необязательный второй аргумент, который указывает позицию, которая должна рассматриваться как конец строки. Если это значение не указано, длина строки используется по умолчанию. Если указан второй аргумент, метод будет обрабатывать строку так, как если бы она имела столько символов:

```
let message = "foobarbaz";

console.log(message.endsWith("bar"));       // false
console.log(message.endsWith("bar", 6));    // true
```

Метод trim()

В ECMAScript введен метод `trim()`, который создает копию строки, удаляет все начальные и конечные пробельные символы, а затем возвращает результат, например:

```
let stringValue = " hello world ";
let trimmedStringValue = stringValue.trim();
console.log(stringValue);         // " hello world "
console.log(trimmedStringValue);  // "hello world"
```

Также доступны методы `trimLeft()` и `trimRight()`, которые удаляют только начальные и только конечные пробельные символы соответственно.

Метод repeat()

В ECMAScript существует метод `repeat()` для всех строк. Метод `repeat()` принимает один целочисленный аргумент `count`, копирует строки `count` раз и объединяет все копии.

```
let stringValue = "na ";
console.log(stringValue.repeat(16) + "batman");
// na na na na na na na na na na na na na na na na batman
```

Методы padStart() и padEnd()

Методы `padStart()` и `padEnd()` копируют строку и, если длина строки меньше указанной длины, добавляют отступы с любой стороны строки, чтобы расширить ее до определенной длины. Первый аргумент — желаемая длина, а второй — необязательная строка, добавляемая в качестве заполнителя. Если не указано иное, будет использоваться символ пробела U + 0020.

```
let stringValue = "foo";

console.log(stringValue.padStart(6));         // "  foo"
console.log(stringValue.padStart(9, "."));    // "......foo"

console.log(stringValue.padEnd(6));           // "foo  "
console.log(stringValue.padEnd(9, "."));      // "foo......"
```

Необязательный аргумент не ограничивается одним символом. Если предоставлена многосимвольная строка, метод будет использовать конкатенированные отступы и урезать их до точной длины. Кроме того, если длина меньше либо равна длине строки, операция, по сути, запрещена.

```
let stringValue = "foo";

console.log(stringValue.padStart(8, "bar"));    // "barbafoo"
console.log(stringValue.padStart(2));           // "foo"

console.log(stringValue.padEnd(8, "bar"));      // "foobarba"
console.log(stringValue.padEnd(2));             // "foo"
```

Строковые итераторы и деструктурирование

Прототип строки предоставляет метод `@@iterator` для каждой строки, позволяющий выполнять итерацию по отдельным символам. Ручное использование итератора работает следующим образом:

```
let message = "abc";
let stringIterator = message[Symbol.iterator]();

console.log(stringIterator.next());    // {value: "a", done: false}
console.log(stringIterator.next());    // {value: "b", done: false}
console.log(stringIterator.next());    // {value: "c", done: false}
console.log(stringIterator.next());    // {value: undefined, done: true}
```

При использовании в цикле `for` цикл будет использовать этот итератор для посещения каждого символа в следующем порядке:

```
for (const c of "abcde") {
    console.log(c);
}

// a
// b
// c
// d
// e
```

Строковый итератор становится особенно полезным, поскольку он допускает взаимодействие с оператором деструктурирования. Это позволяет легко разделить строку по символам:

```
let message = "abcde";

console.log([...message]); // ["a", "b", "c", "d", "e"]
```

Методы изменения регистра символов

Для изменения регистра символов можно использовать методы `toLowerCase()`, `toLocaleLowerCase()`, `toUpperCase()` и `toLocaleUpperCase()`. Методы `toLowerCase()` и `toUpperCase()` созданы по образцу аналогичных Java-методов (`java.lang.String`), а методы `LocaleLowerCase()` и `toLocaleUpperCase()`, по идее, должны быть реализованы на основе конкретного регионального стандарта. Во многих региональных стандартах эти методы не отличаются от универсальных, но в некоторых языках, например в турецком, действуют специальные правила преобразования регистра символов Юникода, что требует использования специфичных методов. Вот некоторые примеры:

```
let stringValue = "hello world";
console.log(stringValue.toLocaleUpperCase());  // "HELLO WORLD"
console.log(stringValue.toUpperCase());        // "HELLO WORLD"
console.log(stringValue.toLocaleLowerCase());  // "hello world"
console.log(stringValue.toLowerCase());        // "hello world"
```

Здесь методы `toLocaleUpperCase()` и `toUpperCase()` выводят строку `"HELLO WORLD"`, а методы `toLocaleLowerCase()` и `toLowerCase()` — `"hello world"`. Если вы не знаете, в какой языковой среде будет выполняться код, безопаснее использовать методы, специфичные для регионального стандарта.

Методы сопоставления строк с шаблонами

Тип `String` содержит несколько методов для сопоставления строк с шаблонами. Первый из них, `match()`, аналогичен методу `exec()` объекта `RegExp`. В качестве единственного аргумента он принимает или строку регулярного выражения, или объект `RegExp`, например:

```
let text = "cat, bat, sat, fat";
let pattern = /.at/;

// то же, что и pattern.exec(text)
let matches = text.match(pattern);
console.log(matches.index);        // 0
console.log(matches[0]);           // "cat"
console.log(pattern.lastIndex);    // 0
```

Метод `match()` возвращает такой же массив, что и метод `exec()` объекта `RegExp`, когда в него передается строка. Первым элементом массива является строка, которая соответствует всему шаблону, а все остальные элементы (если они есть) представляют группы захвата в выражении.

Другой метод поиска шаблонов называется `search()` и принимает такой же аргумент, что и метод `match()`, а именно — регулярное выражение в форме строки или объекта `RegExp`. Метод `search()` возвращает индекс первого вхождения шаблона в строку или –1, если найти его не удается. Поиск шаблона ведется с начала строки. Вот пример:

```
let text = "cat, bat, sat, fat";
let pos = text.search(/at/);
console.log(pos);    // 1
```

Здесь вызов `search(/at/)` возвращает 1 — позицию первого вхождения подстроки `"at"` в строку.

Чтобы упростить замену подстрок, ECMAScript предоставляет метод `replace()`, который принимает два аргумента. Первым может быть объект `RegExp` или строка (она не преобразуется в регулярное выражение), вторым — строка или функция. Если первым аргументом является строка, заменяется только первое вхождение подстроки в строку. Чтобы заменить все экземпляры подстроки, необходимо передать в метод регулярное выражение с глобальным флагом, например:

```
let text = "cat, bat, sat, fat";
let result = text.replace("at", "ond");
console.log(result);    // "cond, bat, sat, fat"

result = text.replace(/at/g, "ond");
console.log(result);    // "cond, bond, sond, fond"
```

В этом примере строка "at" сначала передается в метод replace() с текстом для замены "ond". В результате слово "cat" изменяется на "cond", но остальная часть строки остается неизменной. После замены первого аргумента регулярным выражением с глобальным флагом каждое вхождение "at" заменяется строкой "ond".

Если второй аргумент является строкой, для вставки значений можно использовать несколько специальных последовательностей символов. Доступные в ECMA-262 последовательности представлены в таблице.

ПОСЛЕДОВАТЕЛЬНОСТЬ	ТЕКСТ ДЛЯ ЗАМЕНЫ
$$	$
$&	Подстрока, совпадающая со всем шаблоном. То же, что RegExp.lastMatch
$'	Часть строки перед совпавшей подстрокой. То же, что RegExp.rightContext
$`	Часть строки после совпавшей подстроки. То же, что RegExp.leftContext
$n	*n*-ная группа захвата, где *n* – значение от 0 до 9. Например, $1 – это первая группа захвата, $2 – вторая и т. д. Если захвата нет, используется пустая строка
$nn	*nn*-ная группа захвата, где *nn* – значение от 01 до 99. Например, $01 – это первая группа захвата, $02 – вторая и т. д. Если захвата нет, используется пустая строка

С помощью этих специальных последовательностей можно заменять подстроки, используя сведения о последнем совпадении, например:

```
let text = "cat, bat, sat, fat";
result = text.replace(/(.at)/g, "word ($1)");
console.log(result);    // word (cat), word (bat), word (sat), word (fat)
```

Здесь с помощью последовательности $1 каждое слово, оканчивающееся на "at", заменяется словом "word", за которым в скобках следует исходное слово.

Второй аргумент метода replace() может быть функцией. При наличии одного совпадения в нее передаются три аргумента: совпадение, позиция совпадения в строке и вся строка. Если групп захвата несколько, каждая совпавшая строка передается в функцию как аргумент, при этом двумя последними аргументами являются позиция совпадения с шаблоном в строке и оригинальная строка. Функция должна возвращать строку, указывающую, чем следует заменить совпадение. Использование функции в качестве второго аргумента обеспечивает более детальный контроль над текстом для замены, например:

```
function htmlEscape(text) {
    return text.replace(/[<>"&]/g, function(match, pos, originalText) {
```

```
        switch(match) {
            case "<":
                return "&lt;";
            case ">":
                return "&gt;";
            case "&":
                return "&amp;";
            case "\"":
                return "&quot;";
        }
    });
}
console.log(htmlEscape("<p class=\"greeting\">Hello world!</p>"));
// "&lt;p class=&quot;greeting&quot;&gt;Hello world!&lt;/p&gt";
```

Здесь определяется функция `htmlEscape()`, которая обрабатывает знаки «меньше», «больше», амперсанды и двойные кавычки, подготавливая их к вставке в HTML-код. Для этого выполняется поиск этих знаков с помощью регулярного выражения, а затем определяется функция, которая возвращает специфические HTML-сущности для каждого совпавшего знака.

Последний метод для работы с шаблонами, `split()`, разбивает строку на массив подстрок по разделителю, которым может быть строка или объект `RegExp` (в этом методе строка не считается регулярным выражением). Необязательный второй аргумент, ограничение массива, гарантирует, что возвращенный массив не будет превышать определенный размер. Рассмотрим пример:

```
let colorText = "red,blue,green,yellow";
let colors1 = colorText.split(",");      // ["red", "blue", "green", "yellow"]
let colors2 = colorText.split(",", 2);   // ["red", "blue"]
let colors3 = colorText.split(/[^\,]+/); // ["", ",", ",", ",", ""]
```

В этом примере строка `colorText` содержит список цветов, разделенных запятыми. Вызов `split(",")` возвращает массив этих цветов, разделяя строку по запятым. Далее метод вызывается со вторым аргументом, равным двум, при этом результат ограничивается двумя элементами. Наконец, с помощью регулярного выражения можно получить массив запятых. Обратите внимание, что в последнем вызове `split()` возвращенный массив содержит пустую строку перед запятыми и после. Это происходит потому, что разделитель, указанный с помощью регулярного выражения, имеется и в начале строки (подстрока `"red"`), и в конце (подстрока `"yellow"`).

Метод localeCompare()

Метод `localeCompare()` сравнивает одну строку с другой и возвращает одно из трех значений.

- Если строка должна располагаться по алфавиту перед строковым аргументом, возвращается отрицательное число (обычно –1, но вообще значение может зависеть от реализации).

- Если строка равна строковому аргументу, возвращается 0.
- Если строка должна располагаться по алфавиту после строкового аргумента, возвращается положительное число (обычно 1, но значение может зависеть от реализации).

Рассмотрим пример:

```
let stringValue = "yellow";
console.log(stringValue.localeCompare("brick"));     // 1
console.log(stringValue.localeCompare("yellow"));    // 0
console.log(stringValue.localeCompare("zoo"));       // -1
```

В этом коде строка `"yellow"` сравнивается со значениями `"brick"`, `"yellow"` и `"zoo"`. Поскольку строка `"brick"` предшествует по алфавиту строке `"yellow"`, метод `localeCompare()` возвращает 1. Строка `"yellow"` равна самой себе, поэтому во втором случае метод `localeCompare()` возвращает 0. Наконец, строка `"zoo"` должна располагаться после строки `"yellow"`, поэтому в третий раз возвращается значение -1. Так как значения зависят от реализации, лучше использовать метод `localeCompare()` следующим образом:

```
function determineOrder(value) {
    let result = stringValue.localeCompare(value);
    if (result < 0) {
        console.log("The string 'yellow' comes before the string ' ${value}'.");
    } else if (result > 0) {
        console.log("The string 'yellow' comes after the string '${value}'.");
    } else {
        console.log("The string 'yellow' is equal to the string '${value}'.");
    }
}

determineOrder("brick");
determineOrder("yellow");
determineOrder("zoo");
```

Используя такую конструкцию, вы можете быть уверены, что код будет работать правильно во всех реализациях.

Уникальность метода `localeCompare()` заключается в том, что региональный стандарт реализации (страна и язык) точно указывает, как метод должен работать. В США, где стандартным языком ECMAScript-реализаций является английский, метод `localeCompare()` чувствителен к регистру, то есть прописные буквы предшествуют по алфавиту строчным, но в других региональных стандартах это может быть не так.

Методы для форматирования HTML-кода

Производители веб-браузеров быстро осознали потенциал динамического форматирования HTML-кода средствами JavaScript и добавили в спецификацию несколько предназначенных для этого методов, которые указаны в приведенной таблице. Однако, имейте в виду, что эти методы используются редко, потому что они генерируют несемантическую разметку.

МЕТОД	ВЫВОД
anchor(*имя*)	*строка*
big()	<big>*строка*</big>
bold()	*строка*
fixed()	<tt>*строка*</tt>
fontcolor(*цвет*)	*строка*
fontsize(*размер*)	*строка*
italics()	*строка*
link(url)	*строка*
small()	*строка*
strike()	<strike>*строка*</strike>
sub()	_{*строка*}
sup()	^{*строка*}

ВСТРОЕННЫЕ ОДИНОЧНЫЕ ОБЪЕКТЫ

ECMA-262 определяет встроенный объект как «любой объект, предоставляемый ECMAScript-реализацией, не зависящий от среды выполнения и присутствующий в начале выполнения ECMAScript-программы». Это означает, что разработчикам не нужно явно создавать встроенные объекты, они уже созданы. Вы уже знаете большинство встроенных объектов, таких как `Object`, `Array` и `String`. В ECMA-262 также определены два встроенных одиночных объекта: `Global` и `Math`.

Объект Global

Объект `Global` является самым необычным объектом в ECMAScript, потому что он недоступен явно. В ECMA-262 он описывается как некое подобие склада для свойств и методов, у которых без `Global` не было бы объекта-владельца. В действительности глобальных переменных или глобальных функций не существует — все переменные и функции, определенные глобально, становятся свойствами объекта `Global`. Функции, описанные ранее в этой книге, такие как `isNaN()`, `isFinite()`, `parseInt()` и `parseFloat()`, на самом деле являются методами объекта `Global`. Кроме них у объекта `Global` есть несколько других методов.

Методы кодирования URI

Методы `encodeURI()` и `encodeURIComponent()` используются для кодирования универсальных идентификаторов ресурса (Uniform Resource Identifier, URI), передаваемых браузеру. Допустимые URI не могут содержать определенные знаки,

например пробелы. Чтобы браузер все же мог принимать и понимать их, методы кодирования URI заменяют все недопустимые знаки специальными кодами в кодировке UTF-8.

Метод `encodeURI()` работает с целыми URI (такими как `www.wrox.com/illegal value.js`), а метод `encodeURIComponent()` — исключительно с их сегментами (например, `illegal value.js` из предыдущего URI). Основное различие между двумя методами заключается в том, что `encodeURI()` не кодирует специальные знаки, входящие в URI, такие как точка, косая черта, вопросительный знак и знак решетки, а `encodeURIComponent()` кодирует любой нестандартный знак, который обнаруживает, например:

```
let uri = "http://www.wrox.com/illegal value.js#start";

// "http://www.wrox.com/illegal%20value.js#start"
console.log(encodeURI(uri));

// "http%3A%2F%2Fwww.wrox.com%2Fillegal%20value.js%23start"
console.log(encodeURIComponent(uri));
```

Метод `encodeURI()` оставил значение почти нетронутым, заменив только пробел кодом `%20`, в то время как метод `encodeURIComponent()` заменил все не алфавитно-цифровые символы их закодированными эквивалентами. По этой причине метод `encodeURI()` можно использовать с полными URI, а `encodeURIComponent()` — только со строками, которые добавляются в конец существующих URI.

> **ПРИМЕЧАНИЕ** Метод `encodeURIComponent()` используется гораздо чаще, чем `encodeURI()`, потому что аргументы строк запросов обычно кодируются отдельно от базового URI.

У этих методов есть обратные методы, `decodeURI()` и `decodeURIComponent()`. Метод `decodeURI()` декодирует только те коды, которые генерирует метод `encodeURI()`. Например, код `%20` заменяется пробелом, а код `%23` не заменяется, потому что он представляет знак решетки (`#`), который метод `encodeURI()` пропускает. Соответственно, метод `decodeURIComponent()` декодирует все коды, создаваемые методом `encodeURIComponent()`; по сути, это означает, что он декодирует все специальные значения. Рассмотрим пример:

```
let uri = "http%3A%2F%2Fwww.wrox.com%2Fillegal%20value.js%23start";

// http%3A%2F%2Fwww.wrox.com%2Fillegal value.js%23start
console.log(decodeURI(uri));

// http://www.wrox.com/illegal value.js#start
console.log(decodeURIComponent(uri));
```

Переменная `uri` содержит строку, закодированную с помощью метода `encodeURIComponent()`. Первое закомментированное значение является результатом вызова метода `decodeURI()`, который лишь заменил код `%20` пробелом. Второе значение

возвращено методом decodeURIComponent(), который заменяет все специальные коды соответствующими знаками (эта строка не является допустимым URI).

ПРИМЕЧАНИЕ Методы encodeURI(), encodeURIComponent(), decodeURI() и decodeURIComponent() заменяют методы escape() и unescape(), которые признаны устаревшими в третьей редакции ECMA-262. Новые методы предпочтительнее во всех ситуациях, потому что они кодируют все символы Юникода, а старые правильно кодируют только ASCII-символы. Не используйте методы escape() и unescape() в окончательной версии кода.

Метод eval()

Метод eval() является, наверное, самым мощным во всем языке ECMAScript. Он подобен целому интерпретатору ECMAScript и принимает один аргумент, строку ECMAScript-кода (или JS-кода), которую нужно выполнить, например:

```
eval("console.log('hi')");
```

Этот вызов функционально эквивалентен следующему:

```
console.log("hi");
```

Когда интерпретатор обнаруживает вызов eval(), он преобразует его аргумент в фактические инструкции ECMAScript и заменяет ими аргумент. Код, выполненный с помощью eval(), считается частью контекста выполнения, в котором был вызван метод, и имеет ту же цепочку областей видимости, что и этот контекст. Это означает, что на переменные, определенные во внешнем контексте, можно ссылаться при вызове eval(), например:

```
let msg = "hello world!";
eval("console.log(msg)");    // "hello world!"
```

Переменная msg определена вне контекста вызова eval(), но при вызове console.log() все же выводится текст "Hello world!", потому что вторая строка заменяется реальной строкой кода. Аналогично можно определить функцию или переменные при вызове eval() и ссылаться на них во внешнем коде:

```
eval("function sayHi() { console.log('hi'); }");
sayHi();
```

Здесь функция sayHi() определена внутри вызова eval(), но поскольку он заменяется фактической функцией, можно вызвать sayHi() в следующей строке. Для переменных этот механизма работает так же:

```
eval("let msg = 'hello world!';");
console.log(msg);    // "hello world!"
```

Любые переменные или функции, созданные внутри eval(), не поднимаются, потому что при синтаксическом анализе кода содержатся в строке. Они создаются только во время выполнения метода eval().

В строгом режиме переменные и функции, созданные внутри `eval()`, недоступны снаружи, так что в этих двух примерах возникли бы ошибки. В строгом режиме попытка присвоить значение переменной `eval` также вызывает ошибку:

```
"use strict";
eval = "hi";  // ошибка
```

ПРИМЕЧАНИЕ Возможность интерпретировать строки кода очень эффективна, но и очень опасна. Будьте крайне осторожны с методом `eval()`, особенно при передаче в него данных, введенных пользователем, так как этот метод предоставляет большую возможную поверхность атаки для XSS-уязвимостей. Злоумышленник может попытаться ввести значения, нарушающие безопасность сайта или приложения (это называется инъекцией кода).

Свойства объекта Global

С некоторыми свойствами объекта `Global` мы уже встречались. Например, в их число входят специальные значения `undefined`, `NaN` и `Infinity`. Конструкторы встроенных ссылочных типов, таких как `Object` и `Function`, также являются свойствами объекта `Global`. Все его свойства указаны в таблице.

СВОЙСТВО	ОПИСАНИЕ
`undefined`	Специальное значение `undefined`
`NaN`	Специальное значение `NaN`
`Infinity`	Специальное значение `Infinity`
`Object`	Конструктор `Object`
`Array`	Конструктор `Array`
`Function`	Конструктор `Function`
`Boolean`	Конструктор `Boolean`
`String`	Конструктор `String`
`Number`	Конструктор `Number`
`Date`	Конструктор `Date`
`RegExp`	Конструктор `RegExp`
`Symbol`	Псевдоконструктор `Symbol`
`Error`	Конструктор `Error`
`EvalError`	Конструктор `EvalError`
`RangeError`	Конструктор `RangeError`
`ReferenceError`	Конструктор `ReferenceError`
`SyntaxError`	Конструктор `SyntaxError`
`TypeError`	Конструктор `TypeError`
`URIError`	Конструктор `URIError`

Объект Window

Хотя в ECMA-262 не указан способ непосредственного доступа к объекту `Global`, в веб-браузерах он реализуется с помощью делегата — объекта `window`. Это означает, что все переменные и функции, объявленные в глобальной области видимости, становятся свойствами `window`. Рассмотрим такой пример:

```
let color = "red";

function sayColor() {
    console.log(window.color);
}

window.sayColor();  // "red"
```

Здесь определяются глобальная переменная `color` и глобальная функция `sayColor()`, внутри которой переменная используется как `window.color`, чтобы показать, что она стала свойством `window`. Затем функция вызывается непосредственно для объекта `window` как `window.sayColor()`, в результате чего выводится сообщение в консоли.

> **ПРИМЕЧАНИЕ** Объект `window` в JavaScript используется при решении самых разных задач, а не только как реализация ECMAScript-объекта `Global`. Подробно объект `window` обсуждается в главе 12 «Объектная модель браузера».

Другой способ получить объект `Global` — использовать следующий код:

```
let global = function() {
    return this;
}();
```

В этом коде создается и сразу же вызывается функция-выражение, которая возвращает значение `this`. Как уже отмечалось, если при вызове функции значение `this` не указано явно (то есть функция вызывается не как метод объекта и не с помощью методов `call()`/`apply()`), оно эквивалентно объекту `Global`. Таким образом, вызов функции, которая просто возвращает `this`, обеспечивает согласованный способ получения объекта `Global` в любой среде выполнения.

Объект Math

В ECMAScript-объекте `Math` реализованы математические формулы, константы и вычисления. Объект `Math` предоставляет свойства и методы для проведения вычислений.

> **ПРИМЕЧАНИЕ** Вычисления, доступные на объекте `Math`, выполняются быстрее, чем в аналогичном JS-коде, потому что вычисления на объекте `Math` используют более эффективные реализации в механизме JavaScript и инструкциях процессора. Побочным эффектом этого является то, что точность этих операций может варьироваться в зависимости от браузера, операционной системы, набора команд и аппаратного обеспечения.

Свойства объекта Math

Объект `Math` имеет несколько свойств, которые в основном представляют специальные математические значения. Они приведены в таблице.

СВОЙСТВО	ОПИСАНИЕ
`Math.E`	Значение e, основание натурального логарифма
`Math.LN10`	Натуральный логарифм 10
`Math.LN2`	Натуральный логарифм 2
`Math.LOG2E`	Двоичный логарифм e
`Math.LOG10E`	Десятичный логарифм e
`Math.PI`	Число π
`Math.SQRT1_2`	Квадратный корень из ½
`Math.SQRT2`	Квадратный корень из 2

Смысл и способы использования этих значений мы рассматривать не будем, но вам следует знать, что они описаны в спецификации ECMAScript.

Методы min() и max()

Объект `Math` содержит множество методов, предназначенных для выполнения математических вычислений разной сложности.

Методы `min()` и `max()` определяют наименьшее и наибольшее числа в группе чисел. Они принимают любое количество параметров, например:

```
let max = Math.max(3, 54, 32, 16);
console.log(max);    // 54

let min = Math.min(3, 54, 32, 16);
console.log(min);    // 3
```

Для группы чисел 3, 54, 32 и 16 методы `Math.max()` и `Math.min()` возвращают 54 и 3 соответственно. С их помощью можно определять максимальные и минимальные значения в группах чисел без циклов и условных инструкций.

Для нахождения максимального или минимального значения в массиве можно использовать оператор распространения:

```
let values = [1, 2, 3, 4, 5, 6, 7, 8];
let max = Math.max.apply(...Math, values);
```

Методы округления

В эту группу входят методы `Math.ceil()`, `Math.floor()`, `Math.round()` и `Math.fround()`, которые округляют дробные значения до целых. Все они выполняют округление по-разному.

- Метод `Math.ceil()` всегда округляет числа вверх до ближайшего целого.
- Метод `Math.floor()` всегда округляет числа вниз до ближайшего целого.
- Метод `Math.round()` выполняет стандартное округление (вверх, если дробная часть равна 0.5 или больше, и вниз в противном случае). Это обычный школьный способ округления.
- Метод `Math.fround()` возвращает представление числа с плавающей точкой ближайшей одинарной точности (32 бита).

Следующий пример поясняет работу этих методов:

```
console.log(Math.ceil(25.9));     // 26
console.log(Math.ceil(25.5));     // 26
console.log(Math.ceil(25.1));     // 26

console.log(Math.round(25.9));    // 26
console.log(Math.round(25.5));    // 26
console.log(Math.round(25.1));    // 25

console.log(Math.fround(0.4));    // 0.4000000059604645
console.log(Math.fround(0.5));    // 0.5
console.log(Math.fround(25.9));   // 25.899999618530273

console.log(Math.floor(25.9));    // 25
console.log(Math.floor(25.5));    // 25
console.log(Math.floor(25.1));    // 25
```

Для всех значений между 25 (не включая) и 26 метод `Math.ceil()` возвращает 26, потому что он округляет число вверх. Метод `Math.round()` возвращает 26, только если число равно 25.5 или больше, в противном случае возвращается 25. Наконец, метод `Math.floor()` возвращает 25 для всех чисел между 25 и 26 (не включая).

Метод random()

Метод `Math.random()` возвращает случайное число от 0 до 1, не включая 0 и 1. Это позволяет, например, выводить случайные цитаты или факты, когда пользователь заходит на веб-сайт. С помощью метода `Math.random()` можно получать случайные числа из определенного целочисленного диапазона по следующей формуле:

```
number = Math.floor(Math.random() * количество_вариантов + первое_возможное_
значение)
```

Метод `Math.floor()` используется здесь потому, что `Math.random()` всегда возвращает дробное значение, а значит, после умножения этого значения на число и добавления другого числа также получается дробное значение. Если нужно получить число от 1 до 10, можно использовать такой код:

```
let num = Math.floor(Math.random() * 10 + 1);
```

Обратите внимание, что как первое возможное значение указана единица. Если требуется число от 2 до 10, код будет таким:

```
let num = Math.floor(Math.random() * 9 + 2);
```

В интервале от 2 до 10 только 9 целых чисел, поэтому общее количество вариантов равно 9, а первым возможным значением является 2. При желании также можно использовать следующую функцию, которая сама определяет общее количество вариантов и первое возможное значение, например:

```
function selectFrom(lowerValue, upperValue) {
    let choices = upperValue - lowerValue + 1;
    return Math.floor(Math.random() * choices + lowerValue);
}

let num = selectFrom(2,10);
console.log(num);    // число от 2 до 10 включительно
```

Функция `selectFrom()` принимает два аргумента: наименьшее и наибольшее значения, которые могут быть возвращены. Для определения количества вариантов функция вычисляет разность этих значений и увеличивает ее на единицу, после чего это значение используется в формуле. Вызов `selectFrom(2,10)` возвращает случайное число от 2 до 10 (включительно). Используя эту функцию, можно легко выбрать случайный элемент массива:

```
let colors = ["red", "green", "blue", "yellow", "black", "purple", "brown"];
let color = colors[selectFrom(0, colors.length-1)];
```

В этом примере вторым аргументом `selectFrom()` является длина массива за вычетом 1, что соответствует последней позиции в массиве.

> **ПРИМЕЧАНИЕ** Метод `Math.random()` подходит для целей, продемонстрированных здесь. Если вам необходимо использовать генерацию случайных чисел для криптографических целей (что требует более высокой энтропии во входах генератора), вместо этого предпочтительнее использовать `window.crypto.getRandomValues()`.

Другие методы

Многие методы объекта `Math` служат для выполнения стандартных математических операций. Мы не будем обсуждать их достоинства и недостатки или сценарии их применения, а ограничимся таблицей с кратким описанием.

МЕТОД	ОПИСАНИЕ
`Math.abs(x)`	Возвращает абсолютное значение `x`
`Math.exp(x)`	Возвращает `Math.E` в степени, заданной `x`
`Math.expm1(x)`	Эквивалентна `Math.exp(x) - 1`
`Math.log(x)`	Возвращает натуральный логарифм `x`
`Math.log1p(x)`	Эквивалентна `1 + Math.log(x)`
`Math.pow(x, степень)`	Возвращает `x` в указанной степени

МЕТОД	ОПИСАНИЕ
`Math.pow(...nums)`	Возвращает квадратный корень из суммы квадратов всех чисел из `nums`
`Math.clz32(x)`	Возвращает количество нулей слева 32-разрядного целого числа `x`
`Math.sign(x)`	Возвращает `1`, `0`, `-0` или `-1`, обозначая знак `x`
`Math.trunc(x)`	Возвращает целочисленную составляющую `x`, удаляя все десятичные дроби
`Math.sqrt(x)`	Возвращает квадратный корень из `x`
`Math.cbrt(x)`	Возвращает кубический корень из `x`
`Math.acos(x)`	Возвращает арккосинус `x`
`Math.acosh(x)`	Возвращает гиперболический арккосинус `x`
`Math.asin(x)`	Возвращает арксинус `x`
`Math.asinh(x)`	Возвращает гиперболический арксинус `x`
`Math.atan(x)`	Возвращает арктангенс `x`
`Math.atanh(x)`	Возвращает гиперболический арктангенс `x`
`Math.atan2(y, x)`	Возвращает арктангенс `y/x`
`Math.cos(x)`	Возвращает косинус `x`
`Math.sin(x)`	Возвращает синус `x`
`Math.tan(x)`	Возвращает тангенс `x`

Хотя эти методы определены в ECMA-262, точность вычисления синусов, косинусов и тангенсов зависит от реализации, потому что вычислить их можно многими разными способами.

ИТОГИ

Объекты в JavaScript являются ссылочными значениями. Для создания специфических объектов можно использовать встроенные ссылочные типы.

- Ссылочные типы похожи на классы в традиционном объектно-ориентированном программировании, но реализованы иначе.
- Тип `Date` служит для работы с датами и значениями времени (в том числе с текущими датой и временем).
- Тип `RegExp` обеспечивает поддержку регулярных выражений, реализуя в основном базовую и отчасти нетривиальную функциональность.

Один из уникальных аспектов JavaScript заключается в том, что функции на самом деле являются экземплярами типа `Function`, то есть объектами. Соответственно, у функций есть методы, которые можно использовать для расширения их поведения.

С помощью оболочек примитивных типов можно работать с примитивными типами в JavaScript так, как если бы они были объектами. Есть три оболочки примитивных типов: `Boolean`, `Number` и `String`. Все они имеют ряд общих характеристик.

- Каждый из типов-оболочек соответствует примитивному типу с тем же именем.
- При чтении примитивного значения создается экземпляр его оболочки, используемый для работы со значением.
- После выполнения инструкции с примитивным значением объект-оболочка уничтожается.

В начале выполнения кода уже существуют встроенные объекты `Global` и `Math`. Объект `Global` недоступен в большинстве ECMAScript-реализаций, но в веб-браузерах он представлен объектом `window`. Все глобальные переменные и функции являются свойствами объекта `Global`. Объект `Math` содержит свойства и методы, помогающие выполнять сложные математические вычисления.

6

Ссылочные типы коллекций

- Работа с объектами
- Работа с массивами и типизированными массивами
- Работа с типами Map, WeakMap, Set и WeakSet

Загрузки с сайта wrox.com для этой главы

Обратите внимание, что все примеры кода для этой главы доступны на веб-сайте книги по ссылке www.wrox.com/go/projavascript4e на вкладке Downloads.

ТИП OBJECT

До сих пор в большинстве примеров со ссылочными значениями фигурировал тип `Object` — один из наиболее востребованных ECMAScript-типов. Хотя экземпляры `Object` не могут похвастать широкими возможностями, они идеально подходят для хранения и передачи данных в приложениях.

Есть два способа явного создания экземпляров `Object`. Первый — использовать оператор `new` с конструктором `Object`:

```
let person = new Object();
person.name = "Nicholas";
person.age = 29;
```

Второй способ — использовать *литерал объекта* (object literal). Так называется сокращенная форма определения объекта, которая упрощает создание объектов с большим количеством свойств. С ее помощью объект `person` из предыдущего примера можно определить так:

```
let person = {
    name : "Nicholas",
    age : 29
};
```

В этом примере левая фигурная скобка { указывает начало литерала объекта, потому что мы имеем дело с *контекстом выражения* (expression context). Так в ECMAScript называется контекст, в котором ожидается значение (выражение). Справа от оператора присваивания должно находиться значение, поэтому левая фигурная скобка после знака равенства определяет начало выражения. Та же левая фигурная скобка в *контексте инструкции* (statement context), например после условия в инструкции `if`, определяет начало блочной инструкции.

После скобки указано свойство `name`, а за ним — двоеточие и значение свойства. В литерале объекта свойства разделяются запятыми. В нашем случае запятая указывается после строки `"Nicholas"`, но не после значения 29, потому что `age` — это последнее свойство объекта. Добавление запятой после значения последнего свойства вызывает ошибку в очень старых браузерах, но все новые версии поддерживают это.

В литералах объектов имена свойств могут быть строками и числами, например:

```
let person = {
    "name" : "Nicholas",
    "age" : 29,
    5: true
};
```

В этом примере создается объект со свойствами `name`, `age` и `"5"`. Имейте в виду, что числовые имена свойств автоматически преобразуются в строки.

Используя нотацию литералов объектов, можно создавать объекты, содержащие только свойства и методы, предлагаемые по умолчанию. Для этого следует оставить место в фигурных скобках пустым, например:

```
let person = {};      // то же, что new Object()
person.name = "Nicholas";
person.age = 29;
```

Этот код эквивалентен первому примеру в этом разделе, хотя и выглядит немного странно. Ради ясности лучше использовать литералы объектов, только если вы собираетесь задавать в них свойства.

> **ПРИМЕЧАНИЕ** При определении объекта с помощью литерала конструктор `Object` на самом деле не вызывается.

Хотя приемлем любой способ создания экземпляров `Object`, разработчики обычно предпочитают нотацию литералов объектов, потому что она более лаконична и визуально группирует все связанные данные. Кроме того, она часто применяется, если нужно передать в функцию много необязательных аргументов:

```
function displayInfo(args) {
    let output = "";

    if (typeof args.name == "string") {
        output += "Name: " + args.name + "\n";
    }

    if (typeof args.age == "number") {
        output += "Age: " + args.age + "\n";
    }

    alert(output);
}

displayInfo({
    name: "Nicholas",
    age: 29
});

displayInfo({
    name: "Greg"
});
```

Функция `displayInfo()` принимает единственный аргумент `args`, который может содержать только одно из свойств `name` и `age` или ни одного. Функция проверяет существование свойств с помощью оператора `typeof` и составляет сообщение с имеющимися значениями. Она дважды вызывается с разными данными, которые передаются ей как литералы объектов, и оба раза выводится правильный результат.

> **ПРИМЕЧАНИЕ** Этот способ передачи аргументов в функции хорош при большом количестве необязательных аргументов. Вообще говоря, с именованными аргументами работать проще, но если их много, код становится громоздким. Лучше всего передавать обязательные значения как именованные аргументы и использовать литерал объекта для передачи многочисленных необязательных значений.

Для доступа к свойствам объектов обычно применяется *точечная нотация* (dot notation), типичная для многих объектно-ориентированных языков, но можно также использовать *скобочную нотацию* (bracket notation). В этом случае имя свойства указывается как строка в квадратных скобках, например:

```
alert(person["name"]);    // "Nicholas"
alert(person.name);       // "Nicholas"
```

Функционально эти подходы одинаковы. Главное преимущество скобочной нотации в том, что она позволяет использовать переменные для доступа к свойствам:

```
let propertyName = "name";
alert(person[propertyName]);  // "Nicholas"
```

Скобочная нотация также полезна, если имя свойства содержит ключевое или зарезервированное слово либо знак, вызывающий синтаксическую ошибку:

```
person["first name"] = "Nicholas";
```

Поскольку имя `"first name"` содержит пробел, вы не можете использовать для доступа к нему точечную нотацию. Благодаря скобочной нотации возможен доступ к свойствам, имена которых содержат не только алфавитно-цифровые знаки.

Если же доступ к свойствам по именам с помощью переменных не требуется, обычно точечная нотация предпочтительнее.

> **ПРИМЕЧАНИЕ** Тип `Object` подробно рассматривается в главе 8 «Объекты, классы и объектно-ориентированное программирование».

ТИП ARRAY

Пожалуй, второе место по популярности после типа `Object` в ECMAScript занимает тип `Array` (массив). ECMAScript-массивы сильно отличаются от массивов в большинстве других языков программирования. Как и в других языках, ECMAScript-массивы — это упорядоченные списки данных, но в отличие от других языков они могут содержать значения разных типов. Это означает, что можно создать массив со строкой в первой позиции, числом во второй, объектом в третьей и т. д. ECMAScript-массивы имеют динамические размеры, автоматически увеличиваясь при добавлении данных.

Создание массивов

Создать массив можно несколькими способами. Первый — использовать конструктор `Array`:

```
let colors = new Array();
```

Если известно количество элементов массива, можно передать его в конструктор, и тогда автоматически будет создано свойство `length` (длина массива) с этим значением. Например, следующий код создает массив со значением `length`, равным 20:

```
let colors = new Array(20);
```

В конструктор `Array` можно также передать элементы, которые нужно добавить в массив. Например, здесь создается массив с тремя строками:

```
let colors = new Array("red", "blue", "green");
```

Если в конструктор передается одно значение, все немного сложнее. При передаче числового аргумента всегда создается массив с указанным количеством элементов, но если аргумент имеет другой тип, создается массив с этим единственным элементом, например:

```
let colors = new Array(3);     // массив с тремя элементами
let names = new Array("Greg"); // массив с одним элементом, строкой "Greg"
```

При вызове конструктора `Array` можно опустить оператор `new` без изменения результата:

```
let colors = Array(3);         // массив с тремя элементами
let names = Array("Greg");     // массив с одним элементом, строкой "Greg"
```

Второй способ создать массив — использовать нотацию *литерала массива* (array literal), который представляет собой список разделенных запятыми элементов в квадратных скобках, например:

```
let colors = ["red", "blue", "green"];  // массив с тремя строками
let names = [];                         // пустой массив
let values = [1,2,];                    // массив с двумя элементами
```

Здесь в первой строке создается массив с тремя строковыми переменными. Во второй строке с помощью квадратных скобок создается пустой массив. Следующая инструкция демонстрирует поведение, которое возникает, если ввести лишнюю запятую вслед за последним значением в литерале массива: в массив `values` будут добавлены два элемента со значениями 1 и 2.

В четвертом примере показана другая разновидность этой ошибки: в Internet Explorer 9+, Firefox, Opera, Safari и Chrome эта инструкция создает массив с пятью элементами, а в Internet Explorer 8 и более ранних версий — с шестью. Если значения между запятыми отсутствуют, каждый элемент получает значение `undefined`, что логически эквивалентно вызову конструктора `Array` и передаче в него соответствующего количества элементов. Однако из-за того что литералы массивов некорректно реализованы в ранних версиях Internet Explorer, использовать этот синтаксис не рекомендуется.

> **ПРИМЕЧАНИЕ** Как и в случае объектов, при создании массива с помощью литерала конструктор `Array` не вызывается.

Конструктор `Array` также имеет два дополнительных статических метода, введенных в ES6 для создания массивов: `from()` и `of()`. `from()` используется для преобразования массивоподобных конструкций в экземпляр массива, тогда как `of()` используется для преобразования набора аргументов в экземпляр массива.

Первым аргументом `Array.from()` является «массивоподобный» объект, который представляет собой все, что является итеративным или имеет свойство `length` и проиндексированные элементы. Этот тип может использоваться самыми разными способами:

```
// Строки будут разбиты на массив из отдельных символов
alert(Array.from("Matt")); // ["M", "a", "t", "t"]

// Set и Map могут быть преобразованы в новый экземпляр массива с помощью from()
const m = new Map().set(1, 2)
                   .set(3, 4);
const s = new Set().add(1)
                   .add(2)
                   .add(3)
                   .add(4);

alert(Array.from(m)); // [[1, 2], [3, 4]]
alert(Array.from(s)); // [1, 2, 3, 4]

// Array.from() выполняет поверхностное копирование существующего массива
const a1 = [1, 2, 3, 4];
const a2 = Array.from(a1);

alert(a1);           // [1, 2, 3, 4]
alert(a1 === a2);    // false

// Может быть использован любой итерируемый объект
const iter = {
    *[Symbol.iterator]() {
      yield 1;
      yield 2;
      yield 3;
      yield 4;

    }
};
alert(Array.from(iter));    // [1, 2, 3, 4]

// Объект arguments теперь может быть легко преобразован в массив:
function getArgsArray() {
    return Array.from(arguments);
}
alert(getArgsArray(1, 2, 3, 4));    // [1, 2, 3, 4]

// from() будет использовать пользовательский объект с необходимыми свойствами
const arrayLikeObject = {
    0: 1,
    1: 2,
    2: 3,
    3: 4,
    length: 4
};
alert(Array.from(arrayLikeObject)); // [1, 2, 3, 4]
```

`Array.from()` также принимает второй необязательный аргумент функции `map()`. Это позволяет увеличивать значения нового массива без предварительного создания промежуточного массива, что имеет место, если то же самое было выполнено с помощью `Array.from().map()`. Третий необязательный аргумент указывает значение `this` внутри функции `map()`. Переопределенное значение `this` не применяется внутри стрелочной функции:

```
const a1 = [1, 2, 3, 4];
const a2 = Array.from(a1, x => x**2);
const a3 = Array.from(a1, function(x) {return x**this.exponent}, {exponent: 2});
alert(a2); // [1, 4, 9, 16]
alert(a3); // [1, 4, 9, 16]
```

`Array.of()` преобразует список аргументов в массив. Это служит для замены распространенного до ES6 метода преобразования объекта `arguments` в массив с использованием исключительно громоздких `Array.prototype.slice.call(arguments)`:

```
alert(Array.of(1, 2, 3, 4));    // [1, 2, 3, 4]
alert(Array.of(undefined));     // [undefined]
```

Дыры в массивах

Инициализация массива с помощью литерала массива позволяет создавать «дыры», используя последовательные запятые. ECMAScript будет обрабатывать значение в индексе между запятыми как дыру, а спецификация ES6 уточняет, как обрабатываются эти дыры.

Массив дыр может быть создан следующим образом:

```
const options = [,,,,,];    // Создание массива с 5 элементами
alert(options.length);      // 5
alert(options);             // [,,,,,]
```

Методы и итераторы, представленные в ES6, ведут себя иначе, чем методы, представленные в более ранних версиях ECMAScript. Дополнения ES6 повсеместно будут обрабатывать дыры как существующий элемент со значением `undefined`:

```
const options = [1,,,,5];
for (const option of options) {
    alert(option === undefined);
}
// false
// true
// true
// true
// false

const a = Array.from([,,,]); // Массив из 3 дыр, созданный
                             // с помощью Array.from() в ES6
for (const val of a) {
    alert(val === undefined);
}
// true
// true
// true

alert(Array.of(...[,,,])); // [undefined, undefined, undefined]
for (const [index, value] of options.entries()) {
    alert(value);
}
```

```
// 1
// undefined
// undefined
// undefined
// 5
```

И наоборот, методы, доступные до ES6, будут иметь тенденцию игнорировать дыры, хотя точное поведение может немного отличаться между методами:

```
const options = [1,,,,5];

// map() пропустит дыры целиком
alert(options.map(() => 6)); // [6, undefined, undefined, undefined, 6]

// join() воспринимает дыры как пустые строки
alert(options.join('-')); // "1----5"
```

> **ПРИМЕЧАНИЕ** Из-за их странного поведения и проблем с производительностью избегайте использования дыр в массивах в коде. Предпочтительно использовать явный `undefined` вместо дыры.

Индексирование в массивы

Чтобы получить и установить значения массива, необходимо использовать квадратные скобки и предоставить начальный числовой индекс значения, как показано ниже:

```
let colors = ["red", "blue", "green"];    // объявить массив строк
alert(colors[0]);                         // вывести первый элемент
colors[2] = "black";                      // изменить третий элемент
colors[3] = "brown";                      // добавить четвертый элемент
```

Указатель в квадратных скобках указывает на доступ к значению. Если он меньше количества элементов в массиве, возвращается значение соответствующего элемента: например, для `colors[0]` в приведенном примере выводится строка `"red"`. Задаются значения точно так же, при этом значение в указанной позиции заменяется новым. Если при задании значения указан индекс за пределами массива, как в случае `colors[3]`, длина массива автоматически изменяется на значение этого индекса, увеличенное на единицу (для индекса 3 из примера она увеличивается до 4).

Количество элементов в массиве хранится в свойстве `length`, которое всегда возвращает неотрицательное число:

```
let colors = ["red", "blue", "green"];    // массив с тремя строками
let names = [];                           // пустой массив

alert(colors.length);    // 3
alert(names.length);     // 0
```

Свойство `length` уникально тем, что оно доступно не только для чтения. С его помощью можно легко удалять или добавлять конечные элементы массива.

```
let colors = ["red", "blue", "green"];    // массив с тремя строками
colors.length = 2;
alert(colors[2]);                         // undefined
```

Массив `colors` сначала содержит три значения. Когда свойству `length` присваивается значение 2, последний элемент (в позиции 2) удаляется и становится недоступен как `colors[2]`. Если присвоить свойству `length` значение, превышающее количество элементов в массиве, новые элементы получат значения `undefined`:

```
let colors = ["red", "blue", "green"];    // массив с тремя строками
colors.length = 4;
alert(colors[3]);      // undefined
```

В этом коде свойству `length` массива `colors` присваивается значение 4, хотя массив содержит только три элемента. Позиции 3 в массиве нет, поэтому при обращении к этому значению возвращается специальное значение `undefined`.

Свойство `length` также полезно при добавлении элементов в конец массива:

```
let colors = ["red", "blue", "green"];  // массив с тремя строками
colors[colors.length] = "black";        // добавление черного цвета в позиции 3
colors[colors.length] = "brown";        // добавление коричневого цвета в позиции 4
```

Последний элемент массива всегда находится в позиции `length` — `1`, так что индекс следующей ячейки равен `length`. Каждый раз при добавлении элемента за последним элементом массива свойство `length` автоматически обновляется. Это означает, что инструкция `colors[colors.length]` задает значение в позиции 3 во второй строке примера и в позиции 4 в последней строке. При добавлении элемента в позиции за пределами массива автоматически вычисляется его новая длина, для чего к позиции прибавляется 1, например:

```
let colors = ["red", "blue", "green"];    // массив с тремя строками
colors[99] = "black";    // добавление черного цвета в позиции 99
alert(colors.length);    // 100
```

В этом коде в массив `colors` добавляется значение в позиции 99, при этом длина становится равна 100 (99 + 1). Элементы с индексами от 3 до 98 не существуют, и при доступе к ним возвращается значение `undefined`.

ПРИМЕЧАНИЕ Массивы могут содержать не более 4 294 967 295 элементов, это достаточно для решения почти любых задач. Если попытаться добавить в массив еще больше элементов, возникнет исключение. Попытка создать массив с первоначальным размером, близким к этому максимуму, может вызвать ошибку из-за длительного выполнения сценария.

Идентификация массивов

Одной из классических проблем ECMAScript считается определение того, является ли конкретный объект массивом. При работе с одной веб-страницей (а значит, и с одной областью видимости) для этого можно использовать оператор `instanceof`:

```
if (value instanceof Array) {
    // какие-то действия с массивом
}
```

К сожалению, оператор `instanceof` предполагает, что имеется единственный глобальный контекст выполнения. Если веб-страница содержит несколько фреймов, то глобальных контекстов выполнения тоже несколько и конструктор `Array` имеет как минимум две версии. Если бы нужно было передать массив из одного фрейма в другой, он имел бы иную функцию конструктора, чем массив, изначально созданный во втором фрейме.

Для обходного решения этой проблемы в ECMAScript имеется метод `Array.isArray()`, позволяющий наверняка узнать, является ли конкретное значение массивом, независимо от того, в каком глобальном контексте выполнения оно было создано:

```
if (Array.isArray(value)) {
    // какие-то действия с массивом
}
```

Методы итераторов

В ES6 были добавлены три новых метода в прототип `Array`, которые позволяют проверять содержимое массива: `keys()`, `values()` и `entry()`. `keys()` возвращает итератор индексов массива, `values()` возвращает итератор элементов массива, а `entry()` возвращает итератор пар индекс–значение:

```
const a = ["foo", "bar", "baz", "qux"];

// Поскольку эти методы возвращают итераторы, можно перенаправить их содержимое
// в экземпляры массива с помощью Array.from()
const aKeys = Array.from(a.keys());
const aValues = Array.from(a.values());
const aEntries = Array.from(a.entries());

alert(aKeys);       // [0, 1, 2, 3]
alert(aValues);     // ["foo", "bar", "baz", "qux"]
alert(aEntries);    // [[0, "foo"], [1, "bar"], [2, "baz"], [3, "qux"]]
```

Деструктурирование в ES6 означает, что теперь стало очень легко разделить пары ключ–значение внутри цикла:

```
const a = ["foo", "bar", "baz", "qux"];

for (const [idx, element] of a.entries()) [
    alert(idx);
```

```
    alert(element);
}

// 0
// foo
// 1
// bar
// 2
// baz
// 3
// qux
```

ПРИМЕЧАНИЕ Хотя они и включены в спецификацию ES6, по состоянию на конец 2017 г. в некоторых современных браузерах эти методы до конца не реализованы.

Методы копирования и заполнения

Новыми в ES6 являются два метода, `fill()` и `copyWithin()`, которые соответственно позволяют выполнять пакетное заполнение и копирование внутри массива. Оба метода имеют одинаковую сигнатуру функции в том смысле, что они позволяют указать диапазон в существующем экземпляре массива, используя начальный (включительно) и конечный (не включительно) индексы. Массивы, использующие этот метод, никогда не будут изменены.

Метод `fill()` позволяет вставить одно и то же значение во весь существующий массив или его часть. Добавление необязательного индекса начала указывает на то, что заполнение начнется с этого индекса и будет продолжаться до конца массива, если не указан индекс конца. Отрицательные индексы интерпретируются с конца массива; другой способ их интерпретации состоит в том, что к отрицательным индексам добавляется длина массива для вычисления положительного индекса:

```
const zeroes = [0, 0, 0, 0, 0];

// Весь массив заполняется пятерками
zeroes.fill(5);
alert(zeroes); // [5, 5, 5, 5, 5]
zeroes.fill(0); // сброс

// Заполнение массива начиная с индекса 3 шестерками
zeroes.fill(6, 3);
alert(zeroes); // [0, 0, 0, 6, 6]
zeroes.fill(0); // сброс

// Заполнение массива начиная с индекса 1 и заканчивая индексом 3 семерками
zeroes.fill(7, 1, 3);
alert(zeroes); // [0, 7, 7, 0, 0];
zeroes.fill(0); // сброс
```

```
// Заполнение массива, начиная с индекса 1 и заканчивая индексом 4 восьмерками
// (-4 + zeroes.length = 1)
// (-1 + zeroes.length = 4)
zeroes.fill(8, -4, -1);
alert(zeroes); // [0, 8, 8, 8, 0];
```

`fill()` игнорирует диапазоны, которые выходят за границы массива, имеют нулевую длину или идут в обратном направлении:

```
const zeroes = [0, 0, 0, 0, 0];

// Заполнение со слишком низких индексов недопустимо
zeroes.fill(1, -10, -6);
alert(zeroes); // [0, 0, 0, 0, 0]

// Заполнение со слишком высоких индексов недопустимо
zeroes.fill(1, 10, 15);
alert(zeroes); // [0, 0, 0, 0, 0]

// Заполнение с обратным порядком индексов недопустимо
zeroes.fill(2, 4, 2);
alert(zeroes); // [0, 0, 0, 0, 0]

// Лучшая попытка — это заполнение с частичным перекрытием индексов
zeroes.fill(4, 3, 10)
alert(zeroes); // [0, 0, 0, 4, 4]
```

В отличие от `fill()`, `copyWithin()` выполняет итеративное поверхностное копирование некоторых массивов и перезаписывает существующие значения, начиная с предоставленного индекса. Тем не менее он использует те же соглашения в отношении начального и конечного индексов:

```
let ints,
    reset = () => ints = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
reset();

// Копирует содержимое ints, начиная с индекса 0, до значений, начинающихся
// с индекса 5.
// Останавливается при достижении конца массива в источнике
// индексов или назначенных индексах.
ints.copyWithin(5);
alert(ints); // [0, 1, 2, 3, 4, 0, 1, 2, 3, 4]
reset();

// Копирует содержимое ints, начиная с индекса 5, до значений, начинающихся с
// индекса 0.
ints.copyWithin(0, 5);
alert(ints); // [5, 6, 7, 8, 9, 5, 6, 7, 8, 9]
reset();

// Копирует содержимое ints, начиная с индекса 0 и заканчивая индексом 3 в
// значениях, начинающихся с индекса 4.
ints.copyWithin(4, 0, 3);
alert(ints); // [0, 1, 2, 3, 0, 1, 2, 7, 8, 9]
reset();
```

```
// Движок JS выполнит полное копирование диапазона значений перед вставкой,
// поэтому исчезает опасность перезаписи во время копирования.
ints.copyWithin(2, 0, 6);
alert(ints); // [0, 1, 0, 1, 2, 3, 4, 5, 8, 9]
reset();

// Поддержка отрицательной индексации ведет себя идентично fill() –
// отрицательные индексы считаются с конца массива
ints.copyWithin(-4, -7, -3);
alert(ints); // [0, 1, 2, 3, 4, 5, 3, 4, 5, 6]
```

`fill()` игнорирует диапазоны, которые выходят за границы массива, имеют нулевую длину или идут в обратном направлении:

```
let ints,
    reset = () => ints = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
reset();

// Копирование со слишком низких индексов недопустимо
ints.copyWithin(1, -15, -12);
alert(ints); // [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
reset()

// Копирование со слишком высоких индексов недопустимо
ints.copyWithin(1, 12, 15);
alert(ints); // [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
reset();

// Копирование с обратным порядком индексов недопустимо
ints.copyWithin(2, 4, 2);
alert(ints); // [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
reset();

// Лучшая попытка – это копирование с частичным перекрытием индексов
ints.copyWithin(4, 7, 10)
alert(ints); // [0, 1, 2, 3, 7, 8, 9, 7, 8, 9];
```

Методы преобразования массивов

Как уже было сказано, у всех объектов есть методы `toLocaleString()`, `toString()` и `valueOf()`. Методы `toString()` и `valueOf()`, будучи вызванными для массива, возвращают один и тот же результат, а именно — строку из строковых эквивалентов значений массива, разделенных запятыми. При составлении итоговой строки метод `toString()` вызывается для каждого элемента массива. Рассмотрим пример:

```
let colors = ["red", "blue", "green"];  // массив с тремя строками
alert(colors.toString());   // red,blue,green
alert(colors.valueOf());    // red,blue,green
alert(colors);              // red,blue,green
```

В этом коде методы `toString()` и `valueOf()` сначала вызываются явно, возвращая строковое представление массива, которое содержит его отдельные значения, разделенные запятыми. Затем в метод `alert()` передается сам массив. Поскольку метод

`alert()` ожидает строку, при этом неявно вызывается метод `toString()`, благодаря чему выводится тот же результат, что и при непосредственном вызове `toString()`.

Метод `toLocaleString()` не всегда возвращает то же значение, что и методы `toString()` и `valueOf()`. Если вызвать его для массива, он также составляет строку значений массива, разделенных запятыми, но в отличие от двух других методов, вызывает для получения значений метод `toLocaleString()` каждого элемента массива, а не метод `toString()`. Взгляните на следующий пример:

```
let person1 = {
    toLocaleString() {
        return "Nikolaos";
    },

    toString : function() {
        return "Nicholas";
    }
};

let person2 = {
    toLocaleString() {
        return "Grigorios";
    },

    toString() {
        return "Greg";
    }
};

let people = [person1, person2];
alert(people);                      // Nicholas,Greg
alert(people.toString());           // Nicholas,Greg
alert(people.toLocaleString());     // Nikolaos,Grigorios
```

Здесь определяются объекты `person1` и `person2` с методами `toString()` и `toLocaleString()`, которые возвращают разные значения. После этого создается массив `people`, содержащий оба объекта. При передаче массива в метод `alert()` выводится строка `"Nicholas,Greg"`, потому что для каждого элемента массива вызывается метод `toString()` (как и при явном вызове `toString()` в следующей строке кода). Когда для массива вызывается метод `toLocaleString()`, выводится результат `"Nikolaos,Grigorios"`, потому что в этом случае для каждого элемента массива вызывается метод `toLocaleString()`.

Каждый из унаследованных методов, `toLocaleString()`, `toString()` и `valueOf()`, возвращает элементы массива как строку значений, разделенных запятыми. При желании можно составить строку с другим разделителем, используя метод `join()`. Он принимает разделитель как единственный аргумент и возвращает строку, содержащую все элементы массива:

```
let colors = ["red", "green", "blue"];
alert(colors.join(","));     // red,green,blue
alert(colors.join("||"));    // red||green||blue
```

При вызове с запятой в качестве аргумента метод `join()` повторяет вывод метода `toString()`, возвращая список значений массива `colors`, разделенных запятыми. После этого в него передаются две вертикальные черты, в результате чего выводится строка `"red||green||blue"`. Если в метод `join()` передать значение `undefined` или не передать ничего, в качестве разделителя используется запятая.

> **ПРИМЕЧАНИЕ** Если элемент массива имеет значение `null` или `undefined`, в результатах методов `join()`, `toLocaleString()`, `toString()` и `valueOf()` он представляется пустой строкой.

Методы для работы с массивом как со стеком

Одной из интересных особенностей ECMAScript-массивов является то, что их можно использовать как другие структуры данных. Например, массив может работать как стек — одна из структур данных, которые ограничивают возможности добавления и удаления элементов. Стек работает по принципу *LIFO* (last-in-first-out — последним вошел, первым вышел), то есть последний добавленный элемент извлекается первым. Добавление (push) элементов в стек и *извлечение* (pop) их из стека выполняются только в одном месте: на его вершине. Специально для этих операций в ECMAScript-массивах реализованы методы `push()` и `pop()`.

Метод `push()` принимает любое количество аргументов и добавляет их в конец массива, возвращая его новую длину. Метод `pop()` извлекает последний элемент массива, уменьшает длину массива на 1 и возвращает элемент. Рассмотрим пример:

```
let colors = new Array();                  // создание массива
let count = colors.push("red", "green");   // включение двух элементов
alert(count);                              // 2

count = colors.push("black");              // включение еще одного элемента
alert(count);                              // 3

let item = colors.pop();                   // извлечение последнего элемента
alert(item);                               // "black"
alert(colors.length);                      // 2
```

В этом фрагменте создается массив, который затем используется как стек (обратите внимание, что специальный код для этого не требуется; `push()` и `pop()` — методы, по умолчанию доступные для массива). Сначала две строки добавляются в конец массива с помощью метода `push()`, а новая длина массива (2) присваивается переменной `count`. Затем в стек добавляется еще один элемент, а переменная `count` увеличивается до трех. Поскольку теперь в стеке три элемента, метод `push()` возвращает 3. Далее вызывается метод `pop()`, который возвращает последний элемент массива, строку `"black"`. После этого стек снова содержит два элемента.

Методы стека можно использовать вместе с любыми другими методами массива, например:

```
let colors = ["red", "blue"];
colors.push("brown");          // добавление элемента
colors[3] = "black";           // добавление элемента
alert(colors.length);          // 4

let item = colors.pop();       // получение последнего элемента
alert(item);                   // "black"
```

Здесь массив инициализируется двумя значениями. Затем в него добавляются еще два значения: третье — с помощью метода `push()`, а четвертое — путем непосредственного присваивания строки элементу с индексом 3. При вызове `pop()` возвращается строка `"black"`, которая была добавлена в массив последней.

Методы для работы с массивом как с очередью

Очереди ограничивают доступ к элементам порядком *FIFO* (first-in-first-out — первым вошел, первым вышел). Элементы добавляются в конец очереди и извлекаются из ее начала. Для добавления элементов в конец массива можно использовать метод `push()`, поэтому все, что нужно для имитации очереди, это способ получения первого элемента массива. Этот метод называется `shift()`. Он удаляет первый элемент массива, возвращая его и уменьшая длину массива на 1. Используя метод `shift()` вместе с `push()`, можно работать с массивом как с очередью:

```
let colors = new Array();                    // создание массива
let count = colors.push("red", "green");     // добавление двух элементов
alert(count);                                // 2

count = colors.push("black");    // добавление еще одного элемента
alert(count);                    // 3

let item = colors.shift();       // извлечение первого элемента
alert(item);                     // "red"
alert(colors.length);            // 2
```

В этом примере с помощью метода `push()` создается массив из трех цветов. В выделенной строке метод `shift()` служит для получения первого элемента массива — строки `"red"`. После его удаления в массиве остаются два элемента, при этом первое место занимает элемент `"green"`, а второе — `"black"`.

В ECMAScript у массивов есть также метод `unshift()`, обратный методу `shift()`: он добавляет любое количество элементов в начало массива и возвращает его новую длину. Используя `unshift()` вместе с `pop()`, можно имитировать обратную очередь, в которой значения добавляются в начало массива, а извлекаются с конца, например:

```
let colors = new Array();                       // создание массива
let count = colors.unshift("red", "green");     // добавление элементов
alert(count);                                   // 2

count = colors.unshift("black");    // добавление еще одного элемента
alert(count);                       // 3
```

```
let item = colors.pop();    // извлечение элемента
alert(item);                // "green"
alert(colors.length);       // 2
```

В этом фрагменте созданный массив заполняется с помощью метода `unshift()`. Сначала в массив добавляются строки `"red"` и `"green"`, а затем `"black"`, в результате элементы располагаются в порядке `"black"`, `"red"`, `"green"`. При вызове метода `pop()` последний элемент (`"green"`) удаляется из массива и возвращается.

Методы изменения порядка следования элементов

Для изменения порядка следования элементов, уже находящихся в массиве, используются методы `reverse()` и `sort()`. Метод `reverse()` просто изменяет порядок следования элементов в массиве на обратный, например:

```
let values = [1, 2, 3, 4, 5];
values.reverse();
alert(values);    // 5,4,3,2,1
```

Массив `values` первоначально содержит значения 1, 2, 3, 4 и 5 в данном порядке. Вызов метода `reverse()` для массива изменяет порядок на 5, 4, 3, 2, 1. Этот метод прост, но не гибок, поэтому вам также может пригодиться метод `sort()`.

По умолчанию метод `sort()` располагает элементы по возрастанию: наименьшее значение первым, а наибольшее последним. Чтобы отсортировать массив, он вызывает функцию приведения типов `String()` для каждого элемента, а затем сравнивает возвращенные строки. Это происходит, даже если массив содержит только числа, например:

```
let values = [0, 1, 5, 10, 15];
values.sort();
alert(values);    // 0,1,10,15,5
```

Хотя значения в этом примере сразу расположены в правильном числовом порядке, метод `sort()` сортирует их как строки. Например, строка `"10"` располагается в итоговом массиве раньше, чем `"5"`, хотя число 10 больше. Ясно, что во многих случаях требуется совсем не это, поэтому в метод `sort()` можно передать *функцию сравнения* (comparison function), которая упорядочивает два значения.

Функция сравнения принимает два аргумента и возвращает отрицательное число, если первый аргумент должен предшествовать второму, нуль, если аргументы равны, и положительное число, если первый аргумент должен следовать за вторым. Вот пример простой функции сравнения:

```
function compare(value1, value2) {
    if (value1 < value2) {
        return -1;
    } else if (value1 > value2) {
        return 1;
    } else {
```

```
        return 0;
    }
}
```

Эта функция сравнения работает с большинством типов данных и может использоваться как аргумент метода `sort()`, например:

```
let values = [0, 1, 5, 10, 15];
values.sort(compare);
alert(values);    // 0,1,5,10,15
```

При вызове метода `sort()` с функцией сравнения в качестве аргумента числа остаются в правильном порядке. Если поменять в ней местами возвращаемые значения, массив будет отсортирован по убыванию:

```
function compare(value1, value2) {
    if (value1 < value2) {
        return 1;
    } else if (value1 > value2) {
        return -1;
    } else {
        return 0;
    }
}

let values = [0, 1, 5, 10, 15];
values.sort(compare);
alert(values);    //15,10,5,1,0
```

Кроме того, функция `compare` может быть сокращена и определена как однострочная стрелочная функция:

```
let values = [0, 1, 5, 10, 15];
values.sort((a, b) => a < b ? a > b ? -1 : 0);
alert(values); // 15,10,5,1,0
```

В отличие от предыдущего примера, здесь функция сравнения возвращает 1, если первое значение меньше второго, и –1, если оно больше. Это означает, что в итоговом массиве бóльшие значения будут предшествовать меньшим, и массив будет отсортирован по убыванию. Конечно, если нужно просто изменить порядок следования элементов на обратный, метод `reverse()` намного эффективнее сортировки.

> **ПРИМЕЧАНИЕ** Методы `reverse()` и `sort()` возвращают ссылку на массив, для которого они были вызваны.

Для сортировки чисел и объектов, метод `valueOf()` которых возвращает числа (например, объектов `Date`), в функции сравнения можно просто вычесть одно значение из другого:

```
function compare(value1, value2) {
    return value2 – value1;
}
```

Возвращение разности двух аргументов полностью соответствует спецификации функции сравнения.

Методы манипулирования элементами

Над элементами массивов можно выполнять различные операции. Например, метод `concat()` позволяет создать новый массив на основе текущего. Сначала он создает копию массива, а затем добавляет аргументы в его конец и возвращает новый массив. Если метод `concat()` вызван без аргументов, он просто возвращает копию массива. Если передать в метод `concat()` один или несколько массивов, все их элементы будут добавлены в конец результата. Значения, которые не являются массивами, просто добавляются в конец итогового массива. Рассмотрим пример:

```
let colors = ["red", "green", "blue"];
let colors2 = colors.concat("yellow", ["black", "brown"]);
alert(colors);     // red,green,blue
alert(colors2);    // red,green,blue,yellow,black,brown
```

Сначала массив `colors` содержит три значения. Далее для него вызывается метод `concat()`, которому передаются строка `"yellow"` и массив со значениями `"black"` и `"brown"`. В результате в массив `colors2` записываются строки `"red"`, `"green"`, `"blue"`, `"yellow"`, `"black"` и `"brown"`, а исходный массив `colors` остается неизменным.

Можно переопределить это поведение принудительного выравнивания по умолчанию, указав специальный символ в экземпляре массива аргументов `Symbol.isConcatSpreadable`. Это предотвратит объединение результата методом `concat()`. И наоборот, установка значения в `true` приведет к объединению массивоподобных объектов:

```
let colors = ["red", "green", "blue"];
let newColors = ["black", "brown"];
let moreNewColors = {
    [Symbol.isConcatSpreadable]: true,
    length: 2,
    0: "pink",
    1: "cyan"
};

newColors[Symbol.isConcatSpreadable] = false;

// Отмена объединения массива
let colors2 = colors.concat("yellow", newColors);

// Объединение массивоподобного объекта
let colors3 = colors.concat(moreNewColors);

alert(colors);     // ["red", "green","blue"]
alert(colors2);    // ["red", "green", "blue", "yellow", ["black", "brown"]]
alert(colors3);    // ["red", "green", "blue", "pink, "cyan"]
```

Метод `slice()` создает массив с одним или более элементами, уже содержащимися в массиве. Он принимает один или два аргумента: начальную и конечную позиции

элементов, которые нужно возвратить. Если аргумент только один, метод возвращает все элементы с этой позиции до конца массива. Если аргументов два, метод возвращает все элементы между начальной и конечной позициями, не включая конечный элемент. Эта операция никак не влияет на исходный массив. Рассмотрим пример:

```
let colors = ["red", "green", "blue", "yellow", "purple"];
let colors2 = colors.slice(1);
let colors3 = colors.slice(1,4);

alert(colors2);    // green,blue,yellow,purple
alert(colors3);    // green,blue,yellow
```

Здесь массив `colors` содержит 5 элементов. Метод `slice()` с аргументом 1 возвращает массив с четырьмя элементами без элемента `"red"`, потому что копирование начинается с позиции 1, или строки `"green"`. Итоговый массив `colors2` содержит строки `"green"`, `"blue"`, `"yellow"` и `"purple"`. Далее метод `slice()` вызывается с аргументами 1 и 4, то есть копируются элементы в позициях 1–3. В результате массив `colors3` содержит строки `"green"`, `"blue"` и `"yellow"`.

> **ПРИМЕЧАНИЕ** Если начальная или конечная позиция в `slice()` является отрицательным числом, оно вычитается из длины массива. Например, вызов `slice(-2, -1)` для массива с пятью элементами эквивалентен вызову `slice(3, 4)`. Если конечная позиция меньше начальной, возвращается пустой массив.

Пожалуй, самым мощным методом для работы с массивами является `splice()`. Он используется в основном для вставки элементов в середину массива, но есть и два других способа его применения.

- **Удаление.** Из массива можно удалить любое количество элементов, указав позицию первого элемента, подлежащего удалению, и количество удаляемых элементов. Например, вызов `splice(0, 2)` удаляет первые два элемента.
- **Вставка.** Элементы можно вставить в массив в конкретной позиции, указав три или более аргументов: начальную позицию, 0 (количество удаляемых элементов) и элемент, который нужно вставить. С помощью четвертого, пятого и т. д. параметров можно вставить дополнительные элементы. Например, вызов `splice(2, 0, "red", "green»)` вставляет в массив строки `"red"` и `"green"`, начиная с позиции 2.
- **Замена.** При вставке элементов в конкретной позиции можно одновременно удалить элементы, которые уже есть в массиве. Для этого нужно указать три или более аргументов: начальную позицию, количество удаляемых элементов и любое количество вставляемых элементов. Вставляемых элементов может быть больше или меньше, чем удаляемых. Например, вызов `splice(2, 1, "red", "green")` удаляет один элемент в позиции 2, а затем вставляет в этой же позиции строки `"red"` и `"green"`.

Метод `splice()` всегда возвращает массив, содержащий удаленные элементы (или пустой массив, если элементы не удалялись). Три способа его применения показаны в следующем примере:

```
let colors = ["red", "green", "blue"];
let removed = colors.splice(0,1); // удаление первого элемента
alert(colors);                    // green,blue
alert(removed);                   // red — массив с одним элементом

removed = colors.splice(1, 0, "yellow", "orange"); // вставка двух элементов
                                                   // в позиции 1
alert(colors);                                     // green,yellow,orange,blue
alert(removed);                                    // пустой массив

removed = colors.splice(1, 1, "red", "purple");    // вставка двух значений
                                                   // и удаление одного
alert(colors);                                  // green,red,purple,orange,blue
alert(removed);                                 // yellow — массив с одним элементом
```

В начале примера массив `colors` содержит три элемента. Первый вызов `splice` удаляет первый элемент, оставляя в массиве `colors` строки `"green"` и `"blue"`. Второй вызов `splice()` вставляет два элемента в позиции 1, в результате получается массив со строками `"green"`, `"yellow"`, `"orange"` и `"blue"`. Никакие элементы при этом не удаляются, так что возвращается пустой массив. В последнем вызове `splice()` удаляется один элемент в позиции 1, а вместо него вставляются элементы `"red"` и `"purple"`. После выполнения всего кода массив `colors` будет содержать строки `"green"`, `"red"`, `"purple"`, `"orange"` и `"blue"`.

Методы поиска элементов

ECMAScript предлагает две стратегии поиска элементов в экземплярах массивов: поиск по строгой эквивалентности и поиск с помощью функции предиката.

Строгая эквивалентность

В ECMAScript представлены три метода поиска по строгой эквивалентности — `indexOf()` и `lastIndexOf()`, которые доступны во всех версиях ECMAScript и `includes()`, который был представлен в спецификации ECMAScript 7. Все они принимают два аргумента: искомый элемент и необязательный индекс, с которого начинается поиск. Методы `indexOf()` и `includes()` ищут элемент с начала массива (индекс 0) до конца, а метод `lastIndexOf()` — в обратном порядке.

Методы `indexOf()` и `lastIndexOf()` возвращают позицию элемента в массиве или –1, если найти элемент не удалось. `includes()` возвращает логическое значение, указывающее на то, соответствует ли хотя бы один элемент в массиве поиска указанному элементу. При сравнении первого аргумента с каждым элементом массива проверяется их идентичность, то есть они должны быть строго равны, как при использовании оператора `===`. Вот несколько примеров:

```
let numbers = [1, 2, 3, 4, 5, 4, 3, 2, 1];

alert(numbers.indexOf(4));              // 3
alert(numbers.lastIndexOf(4));          // 5
```

```
alert(numbers.includes(4));            // true
alert(numbers.indexOf(4, 4));          // 5
alert(numbers.lastIndexOf(4, 4));      // 3
alert(numbers.includes(4, 7));         // false

let person = { name: "Nicholas" };
let people = [{ name: "Nicholas" }];
let morePeople = [person];

alert(people.indexOf(person));         // -1
alert(morePeople.indexOf(person));     // 0
alert(people.includes(person));        // false
alert(morePeople.includes(person));    // true
```

Методы `indexOf()` и `lastIndexOf()` позволяют легко найти конкретные элементы в массиве. Они поддерживаются в Internet Explorer 9+, Firefox 2+, Safari 3+, Opera 9.5+ и Chrome.

Поиск с помощью функции предиката

ECMAScript также позволяет определять функцию `predicate`, которая будет вызываться для каждого индекса. Возвращаемое значение функции определяет, считается ли элемент по этому индексу совпадением.

Функция предиката принимает вид `predicate(element, index, array)`, где `element` — это текущий элемент в проверяемом массиве, `index` — это индекс элемента внутри массива, а `array` — это экземпляр массива. Истинное возвращаемое значение указывает на совпадение.

Эти два метода используют `find()` и `findIndex()`. Оба начинают поиск с самого низкого индекса в массиве; `find()` возвращает первый соответствующий элемент, а `findIndex()` возвращает индекс первого соответствующего элемента. Оба метода также принимают второй необязательный параметр, с помощью которого можно указать значение `this` внутри предиката.

```
const people = [
    {
        name: "Matt",
        age: 27
    },
    {
        name: "Nicholas",
        age: 29
    }
];

alert(people.find((element, index, array) => element.age < 28));
// {name: "Matt", age: 27}

alert(people.findIndex((element, index, array) => element.age < 28));
// 0
```

Все методы прекращают поиск, как только будет найдено совпадение.

```
const evens = [2, 4, 6];

// Последний элемент массива никогда не будет проверен после того,
// как было найдено совпадение
evens.find((element, index, array) => {
    alert(element);
    alert(index);
    alert(array);
    return element === 4;
});
// 2
// 0
// [2, 4, 6]
// 4
// 1
// [2, 4, 6]
```

Методы перебора элементов

В ECMAScript определены пять методов перебора элементов массивов. Каждый из них принимает два аргумента: функцию, выполняемую для каждого элемента, и необязательный объект области, в которой будет выполняться функция (зависит от значения `this`). Функция, передаваемая в эти методы, принимает три аргумента: значение элемента массива, позицию элемента в массиве и сам объект массива. В зависимости от метода результаты выполнения этой функции могут не влиять на возвращаемое методом значение. Вот эти пять методов:

- `every()` — выполняет полученную функцию для каждого элемента массива и возвращает `true`, если она возвратила `true` для каждого элемента;
- `filter()` — выполняет полученную функцию для каждого элемента массива и возвращает массив всех элементов, для которых она возвратила `true`;
- `forEach()` — выполняет полученную функцию для каждого элемента массива, но не возвращает никакого значения;
- `map()` — выполняет полученную функцию для каждого элемента массива и возвращает массив с результатами каждого вызова функции;
- `some()` — выполняет полученную функцию для каждого элемента массива и возвращает `true`, если она возвратила `true` хотя бы для одного элемента.

Эти методы не изменяют значения, содержащиеся в массиве.

Из пяти методов наиболее похожи `every()` и `some()`, которые проверяют, соответствуют ли элементы массива некоторым условиям. Метод `every()` возвращает `true`, если переданная ему функция возвратила `true` для каждого элемента массива, в противном случае возвращается `false`. Метод `some()` возвращает `true`, если переданная ему функция возвратила `true` хотя бы для одного элемента, например:

```
let numbers = [1,2,3,4,5,4,3,2,1];

let everyResult = numbers.every((item, index, array) => item > 2);
alert(everyResult);    // false

let someResult = numbers.some((item, index, array) => item > 2);
alert(someResult);     // true
```

В этом коде в методы `every()` и `some()` передается функция, которая возвращает `true`, если элемент больше 2. Метод `every()` возвращает `false`, потому что не все элементы соответствуют условию. Метод `some()` возвращает `true`, потому что как минимум один из элементов больше 2.

Метод `filter()` использует полученную функцию для определения элементов, которые нужно включить в возвращаемый массив. Например, следующий код возвращает массив всех чисел больше 2:

```
let numbers = [1,2,3,4,5,4,3,2,1];

let filterResult = numbers.filter((item, index, array) => item > 2);
alert(filterResult);    // [3,4,5,4,3]
```

Здесь метод `filter()` создает и возвращает массив с элементами 3, 4, 5, 4 и 3, потому что переданная в него функция возвращает `true` для каждого из них. Этот метод полезен, если нужно запросить все элементы массива, соответствующие некоторым условиям.

Метод `map()` возвращает массив, каждый элемент которого является результатом выполнения полученной функции для соответствующего элемента исходного массива. Например, с его помощью можно умножить каждый элемент массива на 2:

```
let numbers = [1,2,3,4,5,4,3,2,1];

let mapResult = numbers.map((item, index, array) => item * 2);

alert(mapResult);    // [2,4,6,8,10,8,6,4,2]
```

В этом примере возвращается массив удвоенных чисел. Метод `map()` полезен при создании массивов, элементы которых соответствуют друг другу.

Пятый метод, `forEach()`, просто выполняет полученную функцию для каждого элемента массива. Он ничего не возвращает и, по сути, аналогичен перебору массива с помощью цикла `for`, например:

```
let numbers = [1,2,3,4,5,4,3,2,1];

numbers.forEach((item, index, array) => {
    // какие-то действия
});
```

Методы перебора упрощают обработку массивов с помощью различных операций.

Методы редукции массивов

В ECMAScript представлены два метода редукции массивов: `reduce()` и `reduceRight()`. Оба они перебирают все элементы в массиве, формируя на их основе единственное возвращаемое значение. Метод `reduce()` делает это в направлении от первого элемента к последнему, а метод `reduceRight()` — в обратном порядке.

Оба метода принимают два аргумента: функцию, вызываемую для каждого элемента, и необязательное первоначальное значение результата редукции. Функция, передаваемая в методы `reduce()` и `reduceRight()`, принимает четыре аргумента: предыдущее значение, текущее значение, индекс элемента и объект массива. Любое значение, возвращаемое из этой функции, автоматически передается как первый аргумент в вызов функции для следующего элемента. Первая итерация выполняется для второго элемента массива, так что первым аргументом функции при этом является первый элемент массива.

Метод `reduce()` можно использовать для выполнения таких операций, как сложение всех чисел в массиве:

```
let values = [1,2,3,4,5];
let sum = values.reduce((prev, cur, index, array) => prev + cur);

alert(sum);    // 15
```

Когда функция обратного вызова выполняется в первый раз, аргументы `prev` и `cur` равны 1 и 2 соответственно. При второй итерации оба они равны 3 (`prev` как сумма 1 и 2, а `cur` как третий элемент массива). Этот процесс продолжается, пока не будут обработаны все элементы.

Метод `reduceRight()` делает то же самое, но в обратном направлении:

```
let values = [1,2,3,4,5];
let sum = values.reduceRight(function(prev, cur, index, array) {
    return prev + cur;
});
alert(sum);    // 15
```

В этой версии при первом вызове функции аргументы `prev` и `cur` равны 5 и 4. Поскольку она просто складывает элементы массива, результат получается таким же, как и в предыдущем примере.

Выбор метода `reduce()` или `reduceRight()` зависит исключительно от того, в каком направлении нужно обработать элементы массива. В остальном эти методы одинаковы.

ТИПИЗИРОВАННЫЕ МАССИВЫ

Появившийся в ECMAScript 6, типизированный массив — это конструкция, предназначенная для эффективной передачи двоичных данных во встроенные

библиотеки. В JavaScript нет фактического типа TypedArray, скорее этот термин относится к коллекции специализированных массивов, которые содержат числовые типы. Чтобы понять, как использовать типизированный массив, полезно сначала понять его предназначение.

История

По мере того как веб-браузеры приобретали популярность, было несложно предвидеть, что возможность запуска сложных 3D-приложений внутри них будет пользоваться популярностью. Еще в 2006 г. поставщики браузеров, в том числе Mozilla и Opera, начали экспериментировать с платформой программирования для рендеринга графических приложений внутри браузера, не требующих запуска плагинов. Цель состояла в том, чтобы разработать JavaScript API, который мог бы использовать API для трехмерной графики, и ускорение графического процессора, чтобы обеспечить визуализацию сложной графики на элементе `<canvas>`.

WebGL

Окончательный JavaScript API был основан на спецификации OpenGL для встроенных систем (OpenGL ES) 2.0, подмножестве OpenGL, которое специализируется на компьютерной графике 2D и 3D. Новый API, названный Web Graphics Library (WebGL), был выпущен в марте 2011 г. С его помощью разработчики смогли написать код приложения с графическим интерфейсом, который может быть интерпретирован любым веб-браузером, совместимым с WebGL.

В начальных версиях WebGL существенное несоответствие между массивами JavaScript и встроенными массивами вызывало проблемы с производительностью. API графических драйверов часто не хотят, чтобы числа передавались им в стандартном формате JavaScript с двойной плавающей точкой. Кроме того, API графических драйверов ожидали, что массивы чисел будут переданы им в двоичном формате, что, конечно, не похоже на формат массива JavaScript в памяти. Поэтому каждый раз, когда массив передавался между WebGL и средой выполнения JavaScript, привязка WebGL выполняла бы дорогостоящую операцию выделения нового массива в среде назначения, итерации по массиву в его текущем формате и преобразования числа в соответствующий формат в новом массиве.

Возникновение типизированных массивов

Это, конечно, было недопустимо, и Mozilla решила эту проблему, реализовав `CanvasFloatArray`, массив чисел с плавающей точкой в стиле C, который предлагает интерфейс JavaScript. Использование этого типа позволило среде выполнения JavaScript выделять, читать и записывать массив, который можно было передавать напрямую в API графического драйвера и из него. `CanvasFloatArray` в конечном итоге будет преобразован в `Float32Array`, который был первым типом, доступным для типизированных массивов, как они существуют и в настоящее время.

Использование типа ArrayBuffer

`Float32Array` — фактически один тип представления, который позволяет среде выполнения JavaScript получать доступ к блоку выделенной памяти, называемому `ArrayBuffer`. `ArrayBuffer` является фундаментальной единицей, на которую ссылаются все типизированные массивы и представления.

> **ПРИМЕЧАНИЕ** `TypedArrayBuffer` – это вариант `ArrayBuffer`, который можно передавать между контекстами выполнения без копирования. Обратитесь к главе 27 «Рабочие потоки» за информацией об этом типе.

`ArrayBuffer` — это обычный JavaScript-конструктор, который можно использовать для выделения определенного количества байтов в памяти.

```
const buf = new ArrayBuffer(16);     // Выделение 16 байтов в памяти
alert(buf.byteLength);               // 16
```

Размер `ArrayBuffer` никогда не может быть изменен. Однако существует возможность скопировать весь или часть существующего `ArrayBuffer` в новый экземпляр, используя `slice()`:

```
const buf1 = new ArrayBuffer(16);
const buf2 = buf1.slice(4, 12);
alert(buf2.byteLength);    // 8
```

`ArrayBuffer` в чем-то похож на `malloc()` в C ++, за некоторыми заметными исключениями.

- Когда `malloc()` не удается выделить память, он возвращает нулевой указатель. Если выделение `ArrayBuffer` завершается неудачно, выдается ошибка.
- Вызов `malloc()` может использовать преимущества виртуальной памяти, поэтому максимальный размер выделения ограничен только адресуемой системной памятью. Выделение `ArrayBuffer` не может превышать `Number.MAX_SAFE_INTEGER` (2 ^ 53) байтов.
- Успешный вызов `malloc()` не выполняет инициализацию реальных адресов. Объявление `ArrayBuffer` инициализирует все биты в 0.
- Память кучи, выделенная функцией `malloc()`, не может использоваться системой, пока не будет вызван метод `free()` или программа не завершит работу. Память кучи, выделенная объявлением `ArrayBuffer`, по-прежнему собирается как мусор — ручное управление памятью не требуется.

Содержимое `ArrayBuffer` не может быть прочитано или записано только со ссылкой на экземпляр буфера. Чтобы прочитать или записать данные, нужно сделать это с представлением. Существуют разные типы представлений, но все они ссылаются на двоичные данные, хранящиеся в `ArrayBuffer`.

Тип DataView

Первый тип представления, позволяющий читать и записывать `ArrayBuffer`, — это `DataView`. Это представление предназначено для файлового ввода-вывода и сетевого ввода-вывода; API обеспечивает высокую степень контроля при работе с буферными данными, но в результате обеспечивает более низкую производительность по сравнению с различными типами представлений. `DataView` не предполагает ничего о содержимом буфера и не является итеративным.

`DataView` должен быть создан для чтения и записи в уже существующий `ArrayBuffer`. Он может использовать весь буфер или только его часть и поддерживает ссылку на экземпляр буфера и место, где в буфере начинается представление.

```
const buf = new ArrayBuffer(16);

// По умолчанию DataView использует весь ArrayBuffer
const fullDataView = new DataView(buf);
alert(fullDataView.byteOffset);              // 0
alert(fullDataView.byteLength);              // 16
alert(fullDataView.buffer === buf);          // true

// Конструктор принимает необязательное смещение байта и длину байта
//    byteOffset=0 начинает просмотр с начала буфера
//    byteLength=8 ограничивает просмотр первыми 8 байтами
const firstHalfDataView = new DataView(buf, 0, 8);
alert(firstHalfDataView.byteOffset);         // 0
alert(firstHalfDataView.byteLength);         // 8
alert(firstHalfDataView.buffer === buf);     // true

// DataView будет использовать оставшуюся часть буфера, если не указано иное
//    byteOffset=8 начинает просмотр с 9-го байта буфера
//    byteLength по умолчанию — остаток буфера
const secondHalfDataView = new DataView(buf, 8);
alert(secondHalfDataView.byteOffset);        // 8
alert(secondHalfDataView.byteLength);        // 8
alert(secondHalfDataView.buffer === buf);    // true
```

Для чтения и записи в буфер через `DataView` вам потребуется использовать несколько компонентов:

- смещение байта, при котором требуется читать или записывать. Его можно рассматривать как своего рода «адрес» в `DataView`;
- `ElementType`, который `DataView` должен использовать для преобразования между типом `Number` во время выполнения JavaScript и двоичным форматом в буфере;
- порядок значения в памяти. По умолчанию используется для байтов.

ElementType

`DataView` не делает никаких предположений о том, какой тип данных хранится в буфере. Предоставляемый им API вынуждает вас указывать `ElementType` при чтении

или записи, а `DataView` покорно выполняет преобразование для выполнения этого чтения или записи.

ECMAScript 6 поддерживает восемь различных `ElementTypes`:

ELEMENTTYPE	БАЙТЫ	ОПИСАНИЕ	ЭКВИВАЛЕНТ В C	ДИАПАЗОН ЗНАЧЕНИЙ
Int8	1	8-разрядное целое число со знаком	`signed char`	от –128 до 127
Uint8	1	8-разрядное целое число без знака	`unsigned char`	от 0 до 255
Int16	2	16-разрядное целое число со знаком	`short`	от –32768 до 32767
Uint16	2	16-разрядное целое число без знака	`unsigned short`	от 0 до 65535
Int32	4	32-разрядное целое число со знаком	`int`	от –2147483648 до 2147483647
Uint32	4	32-разрядное целое число без знака	`unsigned int`	от 0 до 4294967295
Float32	4	32-разрядное число IEEE-754 с плавающей точкой	`float`	от –3.4E+38 до +3.4E+38
Float64	8	64-разрядное число IEEE-754 с плавающей точкой	`double`	от –1.7E+308 до +1.7E+308

`DataView` предоставляет методы `get` и `set` для каждого из этих типов, которые используют `byteOffset` для адресации в буфер для чтения и записи значений. Типы могут использоваться взаимозаменяемо:

```
// Выделение двух байтов памяти и объявление DataView
const buf = new ArrayBuffer(2);
const view = new DataView(buf);

// Демонстрация того, что весь буфер заполнен нулями
// Проверка первого и второго байта
alert(view.getInt8(0));    // 0
alert(view.getInt8(1));    // 0
// Проверка всего буфера
alert(view.getInt16(0));   // 0

// Заполнение буфера единицами
// 255 в двоичной системе – 11111111 (2^8 – 1)
view.setUint8(0, 255);

// DataView автоматически приводит значения к назначенному ElementType
// 255 в восьмеричной системе – 0xFF
```

```
view.setUint8(1, 0xFF);

// Буфер заполнен единицами, что при чтении как
// целое число со знаком дополнения до двух будет прочитано как -1
alert(view.getInt16(0));    // -1
```

Порядок байтов от старшего к младшему и наоборот

Байты буфера в предыдущем примере были намеренно идентичны, чтобы избежать проблемы порядка байтов. «Порядок байтов» относится к соглашению порядка следования байтов, поддерживаемому вычислительной системой. Для целей `DataViews` поддерживаются только два соглашения: порядок байтов от старшего к младшему и от младшего к старшему.

Порядок от старшего к младшему (прямой), также называемый порядком байтов в сети, означает, что самый старший байт содержится в первом байте, а младший байт содержится в последнем байте. Порядок от младшего к старшему (обратный) означает, что младший значащий байт содержится в первом байте, а старший значащий байт содержится в последнем байте.

Собственный порядок системы, запускающей среду выполнения JavaScript, будет определять способ чтения и записи байтов, но `DataView` не подчиняется этому соглашению. `DataView` — это беспристрастный интерфейс для сегмента памяти, который будет следовать любой указанной вами последовательности. Все методы API `DataView` по умолчанию соответствуют соглашению с прямым порядком байтов, но принимают необязательный конечный логический аргумент, который позволяет включить соглашение с обратным порядком байтов, установив для него значение `true`.

```
// Выделение двух байтов памяти и объявление DataView
const buf = new ArrayBuffer(2);
const view = new DataView(buf);

// Заполнение буфера таким образом, что первый и последний байты равны 1
view.setUint8(0, 0x80);    // Установка самого левого байта в 1
view.setUint8(1, 0x01);    // Установка самого правого байта в 1

// Содержимое буфера (разделено пробелами для читабельности):
// 0x8 0x0 0x0 0x1
// 1000 0000 0000 0001

// Чтение в прямом порядке Uint16
// 0x80 — старший байт, 0x01 — младший байт
// 0x8001 = 2^15 + 2^0 = 32768 + 1 = 32769
alert(view.getUint16(0));    // 32769

// Чтение в обратном порядке Uint16
// 0x01 — старший байт, 0x80 — младший байт
// 0x0180 = 2^8 + 2^7 = 256 + 128 = 384
alert(view.getUint16(0, true));    // 384

// Запись в прямом порядке Uint16
view.setUint16(0, 0x0004);
```

```
// Содержимое буфера (разделено пробелами для читабельности):
// 0x0 0x0 0x0 0x4
// 0000 0000 0000 0100
alert(view.getUint8(0));    // 0
alert(view.getUint8(1));    // 4

// Запись в обратном порядке Uint16
view.setUint16(0, 0x0002, true);

// Содержимое буфера (разделено пробелами для читабельности):
// 0x0 0x2 0x0 0x0
// 0000 0010 0000 0000
alert(view.getUint8(0));    // 2
alert(view.getUint8(1));    // 0
```

Граничные случаи

`DataView` завершит чтение или запись, только если для этого будет достаточно места в буфере; в противном случае он выдает `RangeError`:

```
const buf = new ArrayBuffer(6);
const view = new DataView(buf);

// Попытка получить значение, которое частично выходит за пределы буфера
view.getInt32(4);
// RangeError

// Попытка получить значение за пределами буфера
view.getInt32(8);
// RangeError

// Попытка получить значение за пределами буфера
view.getInt32(-1);
// RangeError

// Попытка установить значение за пределами буфера
view.setInt32(4, 123);
// RangeError
```

`DataView` приложит все усилия, чтобы привести значение к соответствующему типу при записи в буфер, возвращаясь к 0. Если это невозможно, он выдаст ошибку:

```
const buf = new ArrayBuffer(1);
const view = new DataView(buf);

view.setInt8(0, 1.5);
alert(view.getInt8(0));    // 1

view.setInt8(0, [4]);
alert(view.getInt8(0));    // 4

view.setInt8(0, 'f');
alert(view.getInt8(0));    // 0

view.setInt8(0, Symbol());
// TypeError
```

Типизированные массивы

Типизированные массивы — это еще одна форма представления `ArrayBuffer`. Хотя он по своей сути похож на `DataView`, типизированный массив отличается тем, что он поддерживает один `ElementType` и подчиняется системному порядку байтов. Взамен он предлагает гораздо более широкий API и улучшенную производительность. Типизированные массивы предназначены для эффективного обмена двоичными данными со встроенными библиотеками, такими как WebGL. Поскольку двоичное представление типизированных массивов находится в легкоусваиваемом формате для собственной операционной системы, механизмы JavaScript способны значительно оптимизировать арифметические, побитовые и другие общие операции над типизированными массивами, и в результате они чрезвычайно быстры в использовании.

Типизированные массивы могут быть созданы для чтения из существующего буфера, инициализированы собственным буфером, заполнены итеративно или из существующего типизированного массива любого типа. Они также могут быть созданы с использованием `<ElementType>.from()` и `<ElementType>.of()`:

```
// Создание буфера из 12 байтов
const buf = new ArrayBuffer(12);
// Создание Int32Array, который ссылается на этот буфер
const ints = new Int32Array(buf);
// Типизированному массиву, который его распознает, нужно 4 байта на элемент,
// потому он будет иметь длину 3
alert(ints.length);               // 3

// Создание Int32Array длиной 6
const ints2 = new Int32Array(6);
// Каждое число занимает 4 байта, поэтому ArrayBuffer состоит из 24 байтов
alert(ints2.length);              // 6
// Как и DataView, типизированные массивы имеют ссылку на связанный буфер
alert(ints2.buffer.byteLength);   // 24

// Создание Int32Array, содержащего [2, 4, 6, 8]
const ints3 = new Int32Array([2, 4, 6, 8]);
alert(ints3.length);              // 4
alert(ints3.buffer.byteLength);   // 16
alert(ints3[2]);                  // 6

// Создание Int16Array со значениями, скопированными из ints3
const ints4 = new Int16Array(ints3);
// Новый типизированный массив выделяет свой собственный буфер, и каждое значение
// конвертируется в новое представление с тем же индексом
alert(ints4.length);              // 4
alert(ints4.buffer.byteLength);   // 8
alert(ints4[2]);                  // 6

// Создание Int16Array из обычного массива
const ints5 = Int16Array.from([3, 5, 7, 9]);
alert(ints5.length);              // 4
```

```
alert(ints5.buffer.byteLength);    // 8
alert(ints5[2]);                   // 7

// Создание Float32Array из параметров
const floats = Float32Array.of(3.14, 2.718, 1.618);
alert(floats.length);               // 3
alert(floats.buffer.byteLength);    // 12
alert(floats[2]);                   // 1.6180000305175781
```

И конструктор, и экземпляры предоставляют свойство `BYTES_PER_ELEMENT`, которое возвращает размер каждого элемента в массиве этого типа:

```
alert(Int16Array.BYTES_PER_ELEMENT); // 2
alert(Int32Array.BYTES_PER_ELEMENT); // 4

const ints = new Int32Array(1),
    floats = new Float64Array(1);

alert(ints.BYTES_PER_ELEMENT);       // 4
alert(floats.BYTES_PER_ELEMENT);     // 8
```

Если типизированный массив не инициализируется значениями, связанный с ним буфер заполняется нулями:

```
const ints = new Int32Array(4);
alert(ints[0]); // 0
alert(ints[1]); // 0
alert(ints[2]); // 0
alert(ints[3]); // 0
```

Поведение типизированного массива

В большинстве случаев типизированные массивы ведут себя так же, как их обычные аналоги. Типизированные массивы поддерживают следующие операторы, методы и свойства:

- `[]`
- `copyWithin()`
- `entries()`
- `every()`
- `fill()`
- `filter()`
- `find()`
- `findIndex()`
- `forEach()`
- `indexOf()`
- `join()`
- `keys()`
- `lastIndexOf()`
- `length`
- `map()`
- `reduce()`
- `reduceRight()`
- `reverse()`
- `slice()`
- `some()`
- `sort()`
- `toLocaleString()`
- `toString()`
- `values()`

Методы, которые возвращают новый массив, вернут новый типизированный массив с тем же типом элементов:

```
const ints = new Int16Array([1, 2, 3]);
const doubleints = ints.map(x => 2*x);
alert(doubleints instanceof Int16Array);    // true
```

Типизированные массивы имеют определенный `Symbol.iterator`, это означает, что для них также могут использоваться циклы `for..of` и операторы распространения:

```
const ints = new Int16Array([1, 2, 3]);
for (const int of ints) {
    alert(int);
}
// 1
// 2
// 3

alert(Math.max(...ints));    // 3
```

Слияние, копирование и изменение типизированных массивов

Типизированные массивы все еще используют буферы в качестве хранилища, и размер буферов массивов не может быть изменен. Следовательно, типизированные массивы не поддерживают следующие методы:

- `concat()`
- `pop()`
- `push()`
- `shift()`
- `splice()`
- `unshift()`

Однако типизированные массивы предлагают два новых метода, которые позволяют быстро копировать значения в массивы и из них: `set()` и `subarray()`.

`set()` копирует значения из предоставленного массива или типизированного массива в текущий типизированный массив по указанному индексу:

```
// Создание массива int16 длиной 8
const container = new Int16Array(8);

// Копирование первых четырех значений в типизированный массив
// Смещение по умолчанию по индексу 0
container.set(Int8Array.of(1, 2, 3, 4));
alert(container); // [1,2,3,4,0,0,0,0]
// Копирование в обычный массив по последним четырем индексам
// Смещение, равное 4, означает, что копирование начинается с индекса 4
container.set([5,6,7,8], 4);
```

```
alert(container); // [1,2,3,4,5,6,7,8]

// Переполнение вызывает ошибку
container.set([5,6,7,8], 7);
// RangeError
```

`subarray()` выполняет операцию, противоположную `set()`, возвращая новый типизированный массив со значениями, скопированными из оригинала. Указывать начальный и конечный индексы необязательно:

```
const source = Int16Array.of(2, 4, 6, 8);

// Копирует массив в новый массив с тем же типом элементов
const fullCopy = source.subarray();
alert(fullCopy);      // [2, 4, 6, 8]

// Копирование массива начиная с индекса 2
const halfCopy = source.subarray(2);
alert(halfCopy);       // [6, 8]

// Копирование массива с индекса 1 до 3
const partialCopy = source.subarray(1, 3);
alert(partialCopy);    // [4, 6]
```

Типизированные массивы не имеют встроенной способности для объединения, но в API типизированных массивов доступно множество инструментов, которые можно создать вручную:

```
// Первый параметр обозначает возвращаемый тип массива
// Остальные параметры — типизированные массивы, которые должны быть объединены
function typedArrayConcat(typedArrayConstructor, ...typedArrays) {
    // Подсчет общего количества элементов в массивах
    const numElements = typedArrays.reduce((x,y) => (x.length || x) + y.length);

    // Создание массива заданного типа с элементами-пробелами
    const resultArray = new typedArrayConstructor(numElements);

    // Успешная транспортировка массива
    let currentOffset = 0;
    typedArrays.map(x => {
        resultArray.set(x, currentOffset);
        currentOffset += x.length;
    });

    return resultArray;
}

const concatArray = typedArrayConcat(Int32Array,
                    Int8Array.of(1, 2, 3),
                    Int16Array.of(4, 5, 6),
                    Float32Array.of(7, 8, 9));

alert(concatArray);    // [1, 2, 3, 4, 5, 6, 7, 8, 9]
alert(concatArray instanceof Int32Array);    // true
```

Опустошение и переполнение

Переполнение и опустошение значений в типизированных массивах не будут перетекать в другие индексы, но все равно нужно учитывать, какой тип элемента массив принимает для своих значений. Типизированные массивы будут принимать только соответствующие биты, которые может содержать каждый индекс в массиве, независимо от того, какое влияние он окажет на фактическое числовое значение. Ниже показана обработка опустошения и переполнения:

```
// Создание знакового массива ints длиной 2
// Каждый индекс содержит целое число со знаком, которое может
// варьироваться от -128 (-1 * 2^7) до 127 (2^7 – 1)
const ints = new Int8Array(2);

// Создание беззнакового массива ints длиной 2
// Каждый индекс содержит беззнаковое целое число, которое может
// варьироваться от 0 до 255 (2^7 – 1)
const unsignedInts = new Uint8Array(2);

// Переполненные биты не будут перетекать в соседние индексы.
// Индекс занимает только младшие 8 бит
unsignedInts[1] = 256;    // 0x100
alert(unsignedInts);      // [0, 0]
unsignedInts[1] = 511;    // 0x1FF
alert(unsignedInts);      // [0, 255]

// Опустошенные биты будут конвертироваться в их беззнаковые эквиваленты.
// 0xFF – это -1 в знаковом варианте (обрезается до 8 бит),
// но 255 – беззнаковое целое число
unsignedInts[1] = -1     // 0xFF (обрезается до 8 бит)
alert(unsignedInts);     // [0, 255]

// Переполнение знакового целого числа происходит прозрачно.
// 0x80 – это 128 в беззнаковом варианте, но -128 в знаковом
ints[1] = 128;          // 0x80
alert(ints);            // [0, -128]

// Опустошение знакового целого числа происходит прозрачно.
// 0xFF – это 255 в беззнаковом варианте, но -1 в знаковом
ints[1] = 255;         // 0xFF
alert(ints);           // [0, -1]
```

В дополнение к восьми типам элементов существует также дополнительный тип «закрепленных» массивов, `Uint8ClampedArray`, который предотвращает переполнение в любом направлении. Значения выше его максимального значения 255 будут округлены до 255, а значения ниже 0 будут округлены до 0.

```
const clampedInts = new Uint8ClampedArray([-1, 0, 255, 256]);
alert(clampedInts);    // [0, 0, 255, 255]
```

По словам Брендана Айча, «`Uint8ClampedArray` — это полностью исторический артефакт элемента холста HTML5. Избегайте его, если только вы действительно не делаете вещи на основе холста».

ТИП MAP

До спецификации ECMAScript 6 реализация хранилища пар ключ–значение в JavaScript могла бы быть эффективно и легко реализована с использованием типа `Object`, используя свойства объекта в качестве ключей и ссылок свойств на их значения. Однако этот стиль реализации не лишен недостатков, и поэтому комитет TC39 счел целесообразным определить спецификацию для истинного хранилища ключей и значений.

Недавно добавленный в ECMAScript 6, `Map` является новым типом коллекций, который вводит истинное поведение пар ключ–значение в язык. Многое из того, что он предлагает, является частичным совпадением с тем, что обеспечивается типом `Object`, но между типами `Object` и `Map` есть тонкие различия, которые следует учитывать при выборе одного из них.

Базовый API

Новый экземпляр типа `Map` создается с помощью ключевого слова `new`:

```
const m = new Map();
```

Если необходимо заполнить `Map` при инициализации, конструктор может принять итеративный объект, ожидая, что тот будет содержать массивы пар ключ–значение. Каждая пара в итерируемом параметре будет вставлена во вновь созданный экземпляр `Map` в том порядке, в котором она были вызвана:

```
// Инициализация экземпляра Map с вложенными массивами
const m1 = new Map([
    ["key1", "val1"],
    ["key2", "val2"],
    ["key3", "val3"]
]);
alert(m1.size); // 3

// Инициализация экземпляра Map с вручную заданным итератором
const m2 = new Map({
    [Symbol.iterator]: function*() {
        yield ["key1", "val1"];
        yield ["key2", "val2"];
        yield ["key3", "val3"];
    }
});
alert(m2.size); // 3

// Map ожидает, что в качестве параметра будут переданы пары ключ-значение
// вне зависимости от того, определены они или нет
const m3 = new Map([[]]);
alert(m3.has(undefined)); // true
alert(m3.get(undefined)); // undefined
```

Пары ключ–значение можно добавлять после инициализации с помощью `set()`, запрашивать с помощью `get()` и `has()`, считывать с помощью свойства `size` и удалять с помощью `delete()` и `clear()`:

```
const m = new Map();

alert(m.has("firstName"));     // false
alert(m.get("firstName "));    // undefined
alert(m.size);                 // 0

m.set("firstName", "Matt")
 .set("lastName", "Frisbie");

alert(m.has("firstName"));     // true
alert(m.get("firstName"));     // Matt
alert(m.size);                 // 2

m.delete("firstName");         // удаление только этой пары ключ-значение

alert(m.has("firstName"));     // false
alert(m.has("lastName"));      // true
alert(m.size);                 // 1

m.clear(); // удаление всех пар ключ-значение в этом экземпляре Map

alert(m.has("firstName"));     // false
alert(m.has("lastName"));      // false
alert(m.size);                 // 0
```

Метод `set()` возвращает экземпляр `Map`, поэтому можно объединить несколько операций над множествами, в том числе в первоначальном объявлении:

```
const m = new Map().set("key1", "val1");

m.set("key2", "val2")
 .set("key3", "val3");

alert(m.size); // 3
```

В отличие от `Object`, который может использовать только цифры или строки в качестве ключей, `Map` может использовать любой тип данных JavaScript в качестве ключа. Он использует операцию сравнения SameValueZero (определенную в спецификации ECMAScript и недоступную в реальном языке) и в основном сопоставим с использованием строгой эквивалентности объектов для проверки соответствия ключа. Как и в случае с `Objects`, нет никаких ограничений для содержимого значений.

```
const m = new Map();

const functionKey = function() {};
const symbolKey = Symbol();
const objectKey = new Object();

m.set(functionKey, "functionValue");
m.set(symbolKey, "symbolValue");
m.set(objectKey, "objectValue");

alert(m.get(functionKey));    // functionValue
alert(m.get(symbolKey));      // symbolValue
alert(m.get(objectKey));      // objectValue

// Проверки SameValueZero означают, что отдельные экземпляры не будут пересекаться
alert(m.get(function() {})); // undefined
```

Как и в случае со строгой эквивалентностью, объекты и другие типы коллекций, используемые для ключей и значений, остаются неизменными внутри `Map` при изменении их содержимого или свойств:

```
const m = new Map();

const objKey = {},
      objVal = {},
      arrKey = [],
      arrVal = [];

m.set(objKey, objVal);
m.set(arrKey, arrVal);

objKey.foo = "foo";
objVal.bar = "bar";
arrKey.push("foo");
arrVal.push("bar");

alert(m.get(objKey)); // {bar: "bar"}
alert(m.get(arrKey)); // ["bar"]
```

Использование операции `SameValueZero` может привести к неожиданным конфликтам:

```
const m = new Map();

const a = 0/"",      // NaN
      b = 0/"",      // NaN
      pz = +0,
      nz = -0;

alert(a === b);      // false
alert(pz === nz);    // true

m.set(a, "foo");
m.set(pz, "bar");

alert(m.get(b));     // foo
alert(m.get(nz));    // bar
```

ПРИМЕЧАНИЕ Операция `SameValueZero` является новой для спецификации ECMAScript. На сайте документации Mozilla есть отличная рецензия на нее и другие соглашения об эквивалентности ECMAScript: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Equality_comparisons_and_sameness.

Порядок и перебор значений

Одним из основных отклонений от соглашений типа `Object` является то, что экземпляры `Map` поддерживают порядок вставки пары ключ–значение и позволяют

выполнять операции перебора в соответствии с порядком вставки. Экземпляр `Map` может предоставить `Iterator`, содержащий массивы пар в форме `[key, value]` в порядке вставки. Этот итератор может быть получен с помощью метода `entries()` или свойства `Symbol.iterator`, которое ссылается на `entries()`:

```
const m = new Map([
    ["key1", "val1"],
    ["key2", "val2"],
    ["key3", "val3"]
]);

alert(m.entries === m[Symbol.iterator]); // true

for (let pair of m.entries()) {
    alert(pair);
}

// [key1,val1]
// [key2,val2]
// [key3,val3]

for (let pair of m[Symbol.iterator]()) {
    alert(pair);
}
// [key1,val1]
// [key2,val2]
// [key3,val3]
```

Поскольку `entries()` является итератором по умолчанию, оператор распространения может использоваться для краткого преобразования `Map` в массив:

```
const m = new Map([
    ["key1", "val1"],
    ["key2", "val2"],
    ["key3", "val3"]
]);

alert([...m]); // [[key1,val1],[key2,val2],[key3,val3]]
```

Для использования соглашения об обратном вызове вместо итератора `forEach(callback, opt_thisArg)` выполняет обратный вызов для каждой пары ключ–значение. При желании он принимает второй параметр, который переопределяет значение `this` внутри каждого обращения к обратному вызову.

```
const m = new Map([
    ["key1", "val1"],
    ["key2", "val2"],
    ["key3", "val3"]
]);

m.forEach((val, key) => alert(`${key} -> ${val}`));
// key1 -> val1
// key2 -> val2
// key3 -> val3
```

`keys()` и `values()` возвращают итератор, который содержит все ключи или все значения в `Map` в порядке вставки:

```
const m = new Map([
    ["key1", "val1"],
    ["key2", "val2"],
    ["key3", "val3"]
]);

for (let key of m.keys()) {
    alert(key);
}
// key1
// key2
// key3

for (let key of m.values()) {
    alert(key);
}
// value1
// value2
// value3
```

Ключи и значения, представленные внутри итератора, являются изменяемыми, но ссылки внутри `Map` не могут быть изменены. Однако это не ограничивает изменение свойств внутри объекта ключа или значения. Это не изменит их идентичность по отношению к экземпляру `Map`:

```
const m1 = new Map([
    ["key1", "val1"]
]);

// Строковый примитив в качестве ключа не был изменен
for (let key of m.keys()) {
    key = "newKey";
    alert(key); // newKey
    alert(m.get("key1")); // val1
}

const keyObj = {id: 1};

const m = new Map([
    [keyObj, "val1"]
]);

// Ключевое свойство объекта изменено, но объект все еще ссылается
// на то же значение внутри Map
for (let key of m.keys()) {
    key.id = "newKey";
    alert(key); // {id: "newKey"}
    alert(m.get(keyObj)); // val1
}
alert(keyObj); // {id: "newKey"}
```

Выбор между Object и Map

Для большинства целей веб-разработки выбор между `Map` и обычным `Object` будет только вопросом предпочтения и не имеет большого значения в других местах. Однако для разработчиков, которые заботятся о памяти и производительности, есть заметные различия между `Object` и `Map`, которые могут иметь важное значение.

Профиль памяти

Реализация `Object` и `Map` на уровне ядра, очевидно, будет отличаться в разных браузерах, но объем памяти, необходимый для хранения одной пары ключ–значение, линейно масштабируется с количеством ключей. Массовое добавление или удаление пар ключ–значение также зависит от того, как механизм реализует распределение памяти для этого типа. Результаты могут отличаться в зависимости от браузера, но при фиксированном объеме памяти `Map` сможет хранить примерно на 50 процентов больше пар ключ–значение, чем `Object`.

Производительность вставки

Вставка новой пары ключ–значение в `Object` по сравнению с `Map` — примерно сравнимая операция, но вставка в `Map` обычно будет происходить немного быстрее во всех браузерных движках. Для обоих типов скорость вставки не соотносится линейно с количеством пар ключ–значение в экземпляре `Object` или `Map`. Если ваш код перегружен операциями вставки, экземпляры `Map` отличаются превосходной производительностью.

Производительность поиска

В отличие от вставки, поиск пары ключ–значение в `Object` по сравнению с `Map` является примерно сопоставимой операцией в масштабе, но в некоторых ситуациях при меньшем количестве пар ключ–значение предпочтительнее выбирать экземпляры `Object`. В ситуациях, когда экземпляр `Object` используется как массив (например, для хранения последовательных целочисленных свойств), механизм браузера может выполнять оптимизации, такие как более эффективный поиск в памяти, что нельзя было бы выполнить с `Map`. Для обоих типов скорость поиска не соотносится линейно с количеством пар ключ–значение в `Object` или `Map`. Если ваш код перегружен операциями поиска, в некоторых сценариях может быть более выгодно использовать `Object`.

Производительность удаления

Производительность операции `delete` в свойствах `Object` ужасна, и эта проблема по-прежнему сохраняется во многих браузерных движках. Обходные пути для псевдоудаления свойств объекта включают назначение `undefined` или `null` значениям свойства, но во многих случаях это является неприятным или неподходящим

компромиссом. В большинстве движков браузера операция `delete()` в `Map` выполняется быстрее, чем вставка и поиск. Если ваш код перегружен операциями удаления, то тип `Map` является самым подходящим.

ТИП WEAKMAP

Недавно добавленный в ECMAScript 6, `WeakMap` («слабый» `Map`) — это новый тип коллекции, который вводит расширенное поведение пар ключ–значение в язык. Тип `WeakMap` является двоюродным братом типа `Map`, а его API является строгим подмножеством `Map`. Обозначение «слабый» описывает то, как сборщик мусора в JavaScript обрабатывает ключи в `WeakMap`.

Базовый API

Новый `WeakMap` можно создать при помощи ключевого слова `new`:

```
const wm = new WeakMap();
```

Ключи в `WeakMap` могут быть только типа `Object` или наследоваться от него — все другие попытки установить ключ с необъектом приведут к ошибке `TypeError`. Нет ограничений по типу значения.

Если необходимо заполнить `WeakMap` при инициализации, конструктор может принять итеративный объект, ожидая, что он будет содержать пары ключ–значение. Каждая пара в итерируемом параметре будет вставлена во вновь созданный `WeakMap` в порядке, в котором они вызываются:

```
const key1 = {id: 1},
      key2 = {id: 2},
      key3 = {id: 3};
// Инициализация WeakMap с вложенными массивами
const wm1 = new WeakMap([
     [key1, "val1"],
     [key2, "val2"],
     [key3, "val3"]
]);
alert(wm.get(key1));    // val2
alert(wm.get(key2));    // val2
alert(wm.get(key3));    // val3

// Инициализация работает по принципу "все или ничего" — единственный неверный ключ
// приведет к ошибке и остановке инициализации
const wm2 = new WeakMap([
    [key1, "val1"],
    ["BADKEY", "val2"],
    [key3, "val3"]
]);
// TypeError: Invalid value used as WeakMap key
typeof wm2;
// ReferenceError: wm2 is not defined
```

```
// Можно использовать примитивы, если добавить оболочку объекта
const stringKey = new String("key1");
const wm3 = new WeakMap([
    stringKey, "val1"
]);
alert(wm3.get(stringKey)); // "val1"
```

Пары ключ–значение можно добавить после инициализации с помощью `set()`, запросить с помощью `get()` и `has()` и удалить с помощью `delete()`:

```
const wm = new WeakMap();

const key1 = {id: 1},
      key2 = {id: 2};

alert(wm.has(key1));    // false
alert(wm.get(key1));    // undefined

wm.set(key1, "Matt")
  .set(key2, "Frisbie");

alert(wm.has(key1)); // true
alert(wm.get(key1)); // Matt

wm.delete(key1); // удаление только этой пары ключ–значение

alert(wm.has(key1)); // false
alert(wm.has(key2)); // true
```

Метод `set()` возвращает экземпляр `WeakMap`, так что можно объединить несколько операций над множествами, в том числе при первоначальном объявлении:

```
const key1 = {id: 1},
      key2 = {id: 2},
      key3 = {id: 3};

const wm = new WeakMap().set(key1, "val1");

wm.set(key2, "val2")
  .set(key3, "val3");

alert(wm.get(key1)); // val1
alert(wm.get(key2)); // val2
alert(wm.get(key3)); // val3
```

Слабые ключи

«Слабое» обозначение появилось из-за того, что ключи в `WeakMap` «слабо удерживаются», то есть они не считаются формальными ссылками, которые в противном случае могли бы предотвратить сборку мусора. Важной особенностью `WeakMap` является то, что ссылка на значение *не* является слабой. Пока ключ существует, пара ключ–значение останется в `Map` и будет считаться ссылкой на значение, тем самым предотвращая сборку его как мусор.

Рассмотрим следующий пример:

```
const wm = new WeakMap();

wm.set({}, "val");
```

Внутри `set()` инициализируется свежий объект для использования в качестве ключа для фиктивной строки. Поскольку нет других ссылок на этот объект, как только эта строка кода будет выполнена, ключ объекта будет свободен для сборки мусора. Когда это произойдет, пара ключ–значение исчезнет из `WeakMap` и станет пустой. В этом примере, поскольку нет других ссылок на значение, это уничтожение пары ключ–значение также будет говорить о том, что значение подходит для сборки в качестве мусора.

Рассмотрим немного другой пример:

```
const wm = new WeakMap();

const container = {
    key: {}
};

wm.set(container.key, "val");

function removeReference()
    container.key = null;
}
```

Здесь объект `container` поддерживает ссылку на ключ в `WeakMap`, поэтому объект не подходит для сборки мусора. Однако как только вызывается `removeReference()`, последняя сильная ссылка на ключевой объект будет разрушена, и сборщик мусора в конечном итоге уничтожит пару ключ–значение.

Неитерируемые ключи

Поскольку пары ключ–значение в `WeakMap` могут быть уничтожены в любой момент, нет смысла предлагать возможность их перебора. Это также исключает возможность одновременного уничтожения всех пар ключ–значение с помощью метода `clear()`, который не является частью API `WeakMap`. Поскольку итерация невозможна, также невозможно получить значение из экземпляра `WeakMap`, если у вас нет ссылки на ключевой объект. Даже если код имеет доступ к экземпляру `WeakMap`, нет способа проверить его содержимое.

Причина, по которой экземпляры `WeakMap` ограничивают ключи только объектами, заключается в том, что они сохраняют соглашение о том, что значения могут быть получены только из `WeakMap` со ссылкой на объект ключа. Если бы примитивы были разрешены, экземпляр `WeakMap` не смог бы отличить строковый примитив, который изначально использовался для установки пары ключ–значение, и идентичный строковый примитив, который был инициализирован позже, — нежелательное поведение.

Полезные стратегии

Экземпляры `WeakMap` поразительно отличаются от существующих инструментов JavaScript, и может быть не сразу понятно, как их следует использовать. На этот вопрос нет однозначного ответа, но уже был придуман ряд некоторых стратегий.

Закрытые переменные

Экземпляры `WeakMap` порождают совершенно новый способ реализации действительно закрытых переменных в JavaScript. Предположение относительно простое: закрытые переменные будут храниться в `WeakMap` с экземпляром объекта в качестве ключа и словарем частных членов в качестве значения.

Одна из реализаций заключается в следующем:

```
const wm = new WeakMap();

class User {
    constructor(id) {
        this.idProperty = Symbol('id');
        this.setId(id);
    }

    setPrivate(property, value) {
        const privateMembers = wm.get(this) || {};
        privateMembers[property] = value;
        wm.set(this, privateMembers);
    }

    getPrivate(property) {
        return wm.get(this)[property];
    }

    setId(id) {
        this.setPrivate(this.idProperty, id);
    }

    getId() {
        return this.getPrivate(this.idProperty);
    }
}

const user = new User(123);
alert(user.getId());    // 123
user.setId(456);
alert(user.getId());    // 456

// Демонстрация не совсем закрыта
alert(wm.get(user)[user.idProperty]);    // 456
```

Наблюдатель может заметить, что в этой реализации внешнему коду требуется только ссылка на экземпляр объекта и `WeakMap` для извлечения «закрытых» переменных. Чтобы предотвратить это, можно заключить `WeakMap` в замыкание, чтобы полностью скрыть экземпляр `WeakMap` от внешнего кода.

```
const User = (() => {
    const wm = new WeakMap();

    class User {
        constructor(id) {
            this.idProperty = Symbol('id');
            this.setId(id);
        }

        setPrivate(property, value) {
            const privateMembers = wm.get(this) || {};
            privateMembers[property] = value;
            wm.set(this, privateMembers);
        }

        getPrivate(property) {
            return wm.get(this)[property];
        }

        setId(id) {
            this.setPrivate(this.idProperty, id);
        }

        getId(id) {
            return this.getPrivate(this.idProperty);
        }
    }
    return User;
})();

const user = new User(123);
alert(user.getId());    // 123
user.setId(456);
alert(user.getId());    // 456
```

Поэтому значение в `WeakMap` не может быть получено без ключа, который использовался для его вставки. Хотя это предотвращает вышеупомянутый шаблон доступа, в некотором смысле он привел код к шаблону закрытой переменной еще до ES6.

Метаданные узла DOM

Поскольку экземпляры `WeakMap` не мешают сборке мусора, они являются потрясающим инструментом для связывания метаданных без очистки. Рассмотрим следующий пример, в котором используется обычный `Map`:

```
const m = new Map();

const loginButton = document.querySelector('#login');

// Ассоциирует некоторые метаданные с узлом
m.set(loginButton, {disabled: true});
```

Предположим, что после выполнения этого кода страница изменяется с помощью JavaScript, а кнопка входа удаляется из дерева DOM. Поскольку ссылка существует

внутри Map, узел DOM будет постоянно оставаться в памяти до тех пор, пока он не будет явно удален из Map или пока Map не будет уничтожен.

Если вместо этого использовался WeakMap, как показано в следующем коде, удаление узла из DOM позволило бы сборщику мусора немедленно освободить выделенную память (при условии, что никаких других устаревших ссылок на объект нет).

```
const wm = new WeakMap();

const loginButton = document.querySelector('#login');

// Ассоциирует некоторые метаданные с узлом
wm.set(loginButton, {disabled: true});
```

ТИП SET

Недавно добавленный в ECMAScript 6, Set — это новый тип коллекции, который вводит поведение набора в язык. Set во многих отношениях ведет себя как расширенный Map, так как большая часть API и поведения у них является общей.

Базовый API

Новый Set можно создать при помощи ключевого слова new:

```
const m = new Set();
```

Если необходимо заполнить набор при инициализации, конструктор может принять итеративный объект, содержащий элементы, которые будут добавлены во вновь созданный экземпляр Set.

```
// Инициализация набора с массивом
const s1 = new Set(["val1", "val2", "val3]);

alert(s1.size); // 3

// Инициализация с вручную заданным итератором
const s2 = new Set({
    [Symbol.iterator]: function*() {
        yield "val1";
        yield "val2";
        yield "val3";
    }
});
alert(s2.size); // 3
```

Значения могут быть добавлены после инициализации с помощью метода add(), запрошены с помощью has(), подсчитаны с помощью свойства size и удалены с помощью delete() и clear():

```
const s = new Set();

alert(s.has("Matt"));      // false
alert(s.size);             // 0
```

```
s.add("Matt")
 .add("Frisbie");

alert(s.has("Matt"));      // true
alert(s.size);             // 2

s.delete("Matt");

alert(s.has("Matt"));     // false
alert(s.has("Frisbie"));  // true
alert(s.size);            // 1

s.clear();                // удаляет все значения в этом экземпляре Set

alert(s.has("Matt"));     // false
alert(s.has("Frisbie"));  // false
alert(s.size);            // 0
```

Метод `add()` возвращает экземпляр `Set`, поэтому можно связать несколько операций вместе, в том числе при первоначальном объявлении:

```
const s = new Set().add("val1");

s.set("val2")
 .set("val3");

alert(s.size);    // 3
```

Как и `Map`, `Set` может содержать любой тип данных JavaScript в качестве значения. Он использует операцию сравнения SameValueZero (определенную в спецификации ECMAScript и недоступную в реальном языке) и в основном сопоставим с использованием строгой эквивалентности объектов для проверки соответствия ключа. Нет никаких ограничений для содержимого значения.

```
const s = new Set();

const functionVal = function() {};
const symbolVal = Symbol();
const objectVal = new Object();

s.add(functionVal);
s.add(symbolVal);
s.add(objectVal);

alert(s.has(functionVal));    // true
alert(s.has(symbolVal));      // true
alert(s.has(objectVal));      // true

// Проверка SameValueZero обозначает, что отдельные экземпляры не пересекаются
alert(s.has(function() {})); // false
```

Как и в случае со строгой эквивалентностью, объекты и другие типы коллекций, используемые для значений, остаются неизменными внутри набора при изменении их содержимого или свойств:

```
const s = new Set();

const objVal = {},
      arrVal = [];

s.add(objVal);
 .add(arrVal);

objVal.bar = "bar";
arrVal.push("bar");

alert(s.has(objVal));    // true
alert(s.has(arrVal));    // true
```

Операции `add()` и `delete()` являются идемпотентными. `delete()` возвращает логическое значение, указывающее на то, присутствовало ли это значение в наборе или нет.

```
const s = new Set();

s.add('foo');
alert(s.size);    // 1
s.add('foo');
alert(s.size);    // 1

// Значение было представлено в наборе
alert(s.delete('foo'));    // true

// Значение не было представлено в наборе
alert(s.delete('foo'));    // false
```

Порядок и перебор значений

`Set` поддерживает порядок вставки значений и позволяет выполнять операции с перебором, следуя порядку вставки.

Экземпляр `Set` может предоставлять `Iterator`, включающий в себя содержимое набора в порядке вставки. Этот итератор может быть получен с помощью метода `values()`, его псевдонима `keys()` или свойства `Symbol.iterator`, которое ссылается на `values()`:

```
const s = new Set(["val1", "val2", "val3"]);

alert(s.values === s[Symbol.iterator]);    // true
alert(s.keys === s[Symbol.iterator]);      // true

for (let value of s.values()) {
    alert(value);
}
// val1
// val2
// val3

for (let value of s[Symbol.iterator]()) {
```

```
    alert(value);
}
// val1
// val2
// val3
```

Поскольку `values()` является итератором по умолчанию, оператор распространения может быть использован для краткого преобразования набора в массив:

```
const s = new Set(["val1", "val2", "val3"]);

alert([...s]); // [val1,val2,val3]
```

`entries()` возвращает итератор, включающий в себя двухэлементный массив, содержащий дубликат всех значений в `Set` в порядке вставки:

```
const s = new Set(["val1", "val2", "val3"]);

for (let pair of s.entries()) {
    alert(pair);
}
// [val1,val1]
// [val2,val2]
// [val3,val3]
```

Для использования соглашения об обратном вызове вместо итератора `forEach(callback, opt_thisArg)` выполняет обратный вызов для каждой пары ключ–значение. При желании он принимает второй параметр, который переопределяет значение `this` внутри каждого обращения к обратному вызову.

```
const s = new Set(["val1", "val2", "val3"]);

s.forEach((val, dupVal) => alert(`${val} -> ${dupVal}`));
// val1 -> val1
// val2 -> val2
// val3 -> val3
```

Изменение свойств значений в `Set` не изменяет идентичность значения по отношению к экземпляру `Set`:

```
const s1 = new Set(["val1"]);

// Строковый примитив в качестве значения не был изменен
for (let value of m.values()) {
    value = "newVal";
    alert(value);          // newVal
    alert(s.has("val1"));  // true
}

const valObj = {id: 1};

const s2 = new Set([valObj]);

// Значение свойства объекта было изменено, но объект все еще существует
// внутри набора
```

```
for (let value of s.values()) {
    value.id = "newVal";
    alert(value);          // {id: "newVal"}
    alert(s.has(valObj));  // true
}

alert(valObj);             // {id: "newKey"}
```

Определение формальных операций над Set

Во многих отношениях `Set` выглядит как `Map` с немного перестроенным API. Это подчеркивается тем фактом, что его API поддерживает только операции со ссылками на самого себя. Операции над `Set` могут понадобиться многим разработчикам, но они требуют ручной реализации и могут принимать форму подкласса `Set` или определять служебную библиотеку. Чтобы воспроизвести оба варианта одновременно, можно реализовать статические методы в подклассе, а затем использовать эти статические методы в методах экземпляра. При реализации этих операций необходимо помнить несколько моментов.

- Некоторые операции над `Set` являются ассоциативными, поэтому полезно иметь возможность реализовать метод таким образом, чтобы он мог обрабатывать произвольное количество экземпляров `Set`.
- `Set` сохраняет порядок вставки, и наборы, возвращаемые этими методами, должны отражать этот факт.
- Эффективно используйте память везде, где это возможно. Операторы распространения предлагают хороший синтаксис, но по возможности избегайте переключения туда-сюда между наборами и массивами, чтобы сэкономить на затратах на инициализацию объектов.
- Не изменяйте существующие экземпляры `Set`. Метод `union(a, b)` или `a.union(b)` возвращает новый экземпляр Set.

```
class XSet extends Set {
    union(...sets) {
        return XSet.union(this, ...sets)
    }

    intersection(...sets) {
        return XSet.intersection(this, ...sets);
    }

    difference(set) {
        return XSet.difference(this, set);
    }

    symmetricDifference(set) {
        return XSet.symmetricDifference(this, set);
    }

    cartesianProduct(set) {
        return XSet.cartesianProduct(this, set);
```

```
}

powerSet() {
    return XSet.powerSet(this);
}

// Возвращает объединение двух или более наборов.
static union(a, ...bSets) {
    const unionSet = new XSet(a);
    for (const b of bSets) {
        for (const bValue of b) {
            unionSet.add(bValue);
        }
    }
    return unionSet;
}

// Возвращает пересечение двух или более наборов.
static intersection(a, ...bSets) {
    const intersectionSet = new XSet(a);
    for (const aValue of intersectionSet) {
        for (const b of bSets) {
            if (!b.has(aValue)) {
                intersectionSet.delete(aValue);
            }
        }
    }
    return intersectionSet;
}

// Возвращает разницу между двумя наборами.
static difference(a, b) {
    const differenceSet = new XSet(a);
    for (const bValue of b) {
        if (a.has(bValue)) {
            differenceSet.delete(bValue);
        }
    }
    return differenceSet;
}

// Возвращает симметричную разницу между двумя наборами.
static symmetricDifference(a, b) {
    // По определению, симметричная разница может быть выражена как
    // (a union b) – (a intersection b)
    return a.union(b).difference(a.intersection(b));
}

// Возвращает декартово произведение (в виде пар массивов) двух наборов.
// Должен возвращать набор массивов, так как декартово произведение может
// содержать пары идентичных значений.
static cartesianProduct(a, b) {
    const cartesianProductSet = new XSet();
    for (const aValue of a) {
        for (const bValue of b) {
            cartesianProductSet.add([aValue, bValue]);
```

```
            }
        }
        return cartesianProductSet;
    }

    // Возвращает мощность набора.
    static powerSet(a) {
        const powerSet = new XSet().add(new XSet());
        for (const aValue of a) {
            for (const set of new XSet(powerSet)) {
                powerSet.add(new XSet(set).add(aValue));
            }
        }
        return powerSet;
    }
}
```

ТИП WEAKSET

Недавно добавленный в ECMAScript 6, `WeakSet` («слабый» `Set`) — это новый тип коллекции, который вводит поведение набора в язык. Тип `WeakSet` является родственником типа `Set`, а его API является строгим подмножеством `Set`. Обозначение «слабый» описывает, как сборщик мусора в JavaScript обрабатывает значения в слабом наборе.

Базовый API

Новый экземпляр `WeakSet` можно создать при помощи ключевого слова `new`:

```
const ws = new WeakSet();
```

Значения в `WeakSet` могут быть только типа `Object` или наследоваться от него — все другие попытки установить значение с необъектом приведут к ошибке `TypeError`.

Если необходимо заполнить `WeakSet` при инициализации, конструктор может принять итеративный объект как параметр, ожидая, что он будет содержать допустимые значения. Каждое значение в итерируемом параметре будет вставлено во вновь созданный `WeakSet` в порядке, в котором они вызываются:

```
const val1 = {id: 1},
      val2 = {id: 2},
      val3 = {id: 3};

// Инициализация WeakSet с вложенными массивами
const ws1 = new WeakSet([val1, val2, val3]);

alert(ws1.has(val1));    // true
alert(ws1.has(val2));    // true
alert(ws1.has(val3));    // true

// Инициализация работает по принципу "все или ничего" – единственное неверное
// значение выдаст ошибку и прервет инициализацию
```

```
const ws2 = new WeakSet([val1, "BADVAL", val3]);
// TypeError: Invalid value used in WeakSet
typeof ws2;
// ReferenceError: ws2 is not defined

// Можно использовать примитивы, если добавить оболочку объекта
const stringVal = new String("val1");
const ws3 = new WeakSet([stringVal]);
alert(ws3.has(stringVal));    // true
```

После инициализации значения можно добавлять с помощью метода `add()`, запрашивать с помощью `has()` и удалять с помощью `delete()`:

```
const ws = new WeakSet();

const val1 = {id: 1},
      val2 = {id: 2};

alert(ws.has(val1)); // false

ws.add(val1)
  .add(val2);

alert(ws.has(val1)); // true
alert(ws.has(val2)); // true

ws.delete(val1); // удаляет только это значение

alert(ws.has(val1)); // false
alert(ws.has(val2)); // true
```

Метод `add()` возвращает экземпляр `WeakSet`, поэтому можно связать несколько операций добавления вместе, в том числе при первоначальном объявлении:

```
const val1 = {id: 1},
      val2 = {id: 2},
      val3 = {id: 3};

const ws = new WeakSet().add(val1);

ws.add(val2)
  .add(val3);

alert(ws.has(val1)); // true
alert(ws.has(val2)); // true
alert(ws.has(val3)); // true
```

Слабые ключи

Обозначение «слабые» появилось из-за того, что значения в `WeakSet` «слабо удерживаются», то есть они не считаются формальными ссылками, которые в противном случае предотвратили бы сборку мусора:

```
const ws = new WeakSet();

ws.add({});
```

Внутри `add()` свежий объект инициализируется и используется как значение. Поскольку нет других ссылок на этот объект, как только эта строка кода будет выполнена, значение объекта станет свободным для сборки мусора. После этого значение исчезнет из `WeakSet` и тот станет пустым.

Рассмотрим немного другой пример:

```
const ws = new WeakSet();

const container = {
    val: {}
};

ws.add(container.val);

function removeReference()
    container.val = null;
}
```

Здесь объект `container` поддерживает ссылку на значение в экземпляре `WeakSet`, поэтому объект не подходит для сборки мусора. Однако как только вызывается `removeReference()`, последняя сильная ссылка на ключевой объект будет разрушена, и сборщик мусора в конечном итоге уничтожит значение.

Неитерируемые значения

Поскольку значения в `WeakSet` могут быть уничтожены в любой момент, нет смысла предлагать возможность их перебора. Это также исключает возможность одновременного уничтожения всех значений с помощью метода `clear()`, который не является частью API `WeakSet`. Поскольку итерация невозможна, также невозможно получить значение из экземпляра `WeakSet`, если у вас нет ссылки на объект значения. Даже если код имеет доступ к экземпляру `WeakSet`, нет способа проверить его содержимое.

Экземпляры `WeakSet` ограничивают ключи только объектами по причине того, что они сохраняют соглашение о том, что значения могут быть получены только из `WeakSet` со ссылкой на объект значения. Если бы примитивы были разрешены, экземпляр `WeakSet` не смог бы отличить строковый примитив, который изначально использовался для установки значения, и идентичный строковый примитив, который был инициализирован позже, — нежелательное поведение.

Полезные стратегии

Экземпляры `WeakSet` более ограничены в своей полезности по сравнению с экземплярами `WeakMap`, но они по-прежнему полезны для маркировки объектов.

Рассмотрим следующий пример, в котором используется обычный `Set`:

```
const disabledElements = new Set();

const loginButton = document.querySelector('#login');

// Помечает узел как "недоступный", добавляя его в соответствующий набор
disabledElements.add(loginButton);
```

Здесь можно проверить, доступен ли элемент, посмотрев, существует ли он внутри `disabledElements`, что можно сделать за постоянное время. Однако если элемент удален из DOM, его присутствие внутри этого `Set` предотвратит перераспределение памяти сборщиком мусора.

Чтобы разрешить сборке мусора перераспределить память элемента, вместо этого можно использовать `WeakSet`:

```
const disabledElements = new WeakSet();

const loginButton = document.querySelector('#login');

// Помечает узел как "недоступный", добавляя его в соответствующий набор
disabledElements.add(loginButton);
```

Теперь, когда любой элемент в `WeakSet` удаляется из DOM, сборщик мусора будет игнорировать его присутствие внутри `WeakSet` при рассмотрении его как претендента на сборку.

ИТЕРАТОРЫ И ОПЕРАТОРЫ РАСПРОСТРАНЕНИЯ

В ECMAScript 6 представлены итераторы и оператор распространения, которые особенно полезны в контексте типов ссылок на коллекции. Эти новые инструменты позволяют легко взаимодействовать, клонировать и изменять типы коллекций.

> **ПРИМЕЧАНИЕ** Глава 7 «Итераторы и генераторы» содержит больше информации о том, как именно работают итераторы.

Как показано ранее в этой главе, четыре стандартных типа ссылок на коллекцию определяют итератор по умолчанию:

```
Array
All typed arrays
Map
Set
```

Проще говоря, это означает, что все они поддерживают упорядоченную итерацию и могут быть переданы в цикл `for..of`:

```
let iterableThings = [
    Array.of(1, 2),
    typedArr = Int16Array.of(3, 4),
    new Map([[5, 6], [7, 8]]),
    new Set([9, 10])
];

for (const iterableThing of iterableThings) {
    for (const x of iterableThing) {
        console.log(x);
```

```
    }
}

// 1
// 2
// 3
// 4
// [5, 6]
// [7, 8]
// 9
// 10
```

Это также означает, что все эти типы совместимы с оператором распространения. Оператор распространения особенно полезен, поскольку он выполняет поверхностное копирование итерируемого объекта. Это позволяет с легкостью клонировать целые объекты, используя краткий синтаксис:

```
let arr1 = [1, 2, 3];
let arr2 = [...arr1];

console.log(arr1);              // [1, 2, 3]
console.log(arr2);              // [1, 2, 3]
console.log(arr1 === arr2);     // false
```

Конструкторам, которые ожидают итерируемый объект в качестве параметра, можно просто передать клонируемый экземпляр:

```
let map1 = new Map([[1, 2], [3, 4]]);
let map2 = new Map(map1);

console.log(map1); // Map {1 => 2, 3 => 4}
console.log(map2); // Map {1 => 2, 3 => 4}
```

Это также допустимо для частичного построения массива:

```
let arr1 = [1, 2, 3];
let arr2 = [0, ...arr1, 4, 5];

console.log(arr2); // [0, 1, 2, 3, 4, 5]
```

Механизм поверхностного копирования означает, что копируются только ссылки на объекты:

```
let arr1 = [{}];
let arr2 = [...arr1];

arr1[0].foo = 'bar';
console.log(arr2[0]); // { foo: 'bar' }
```

Каждый из этих типов коллекций поддерживает несколько методов конструирования, таких как статические методы `Array.of()` и `Array.from()`. В сочетании с оператором распространения это обеспечивает чрезвычайно простую совместимость:

```
let arr1 = [1, 2, 3];

// Копирование массива в типизированный массив

let typedArr1 = Int16Array.of(...arr1);
let typedArr2 = Int16Array.from(arr1);
console.log(typedArr1); // Int16Array [1, 2, 3]
console.log(typedArr2); // Int16Array [1, 2, 3]

// Копирование массива в Map
let map = new Map(arr1.map((x) => [x, 'val' + x]));
console.log(map); // Map {1 => 'val 1', 2 => 'val 2', 3 => 'val 3'}

// Копирование массива в набор
let set = new Set(typedArr2);
console.log(set); // Set {1, 2, 3}

// Копирование набора обратно в массив
let arr2 = [...set];
console.log(arr2); // [1, 2, 3]
```

ИТОГИ

Объекты в JavaScript называются ссылочными значениями, и несколько встроенных ссылочных типов могут использоваться для создания определенных типов объектов, как указано ниже.

- Ссылочные типы похожи на классы в традиционном объектно-ориентированном программировании, но реализованы по-разному.
- Тип `Object` — это основа, от которой все другие ссылочные типы наследуют основное поведение.
- Тип `Array` представляет упорядоченный список значений и предоставляет функциональные возможности для манипулирования и преобразования значений.
- Типизированные массивы охватывают ряд различных ссылочных типов, которые включают управление типами чисел в памяти.
- Тип `Date` предоставляет информацию о датах и времени, включая текущую дату, время и соответствующие расчеты.
- Тип `RegExp` — это интерфейс для поддержки регулярных выражений в ECMAScript, предоставляющий самые основные и некоторые расширенные функциональные возможности регулярных выражений.

Одним из уникальных аспектов JavaScript является то, что функции на самом деле являются экземплярами типа `Function`, то есть функции являются объектами. Поскольку функции являются объектами, у них есть методы, которые можно использовать для улучшения их поведения.

Из-за существования примитивных типов-оболочек примитивные значения в JavaScript могут быть доступны так же, как если бы они были объектами.

Существует три типа примитивных оболочек: `Boolean`, `Number` и `String`. Все они имеют следующие характеристики:

- каждый из типов оболочек соответствует типу примитива с тем же именем;
- при доступе к примитивному значению в режиме чтения создается экземпляр объекта-примитива, чтобы его можно было использовать для манипулирования данными;
- как только выполняется выражение с примитивным значением, объект-оболочка уничтожается.

Есть еще два встроенных объекта, которые существуют в начале выполнения кода: `Global` и `Math`. Объект `Global` недоступен в большинстве реализаций ECMAScript, однако веб-браузеры реализуют его как объект `window`. Объект `Global` содержит все глобальные переменные и функции как свойства. Объект `Math` содержит свойства и методы, полезные для сложных математических вычислений.

В ECMAScript 6 были представлены несколько типов коллекций: `Map`, `WeakMap`, `Set` и `WeakSet`. Они предлагают новые возможности для организации данных приложения, а также упрощают управление памятью.

7

Итераторы и генераторы

- Введение в итерацию
- Паттерн Итератор
- Генераторы

Загрузки с сайта wrox.com для этой главы

Обратите внимание, что все примеры кода для этой главы доступны на веб-сайте книги по ссылке www.wrox.com/go/projavascript4e на вкладке Downloads.

Термин «итерация» происходит от латинского *itero*, что означает «повторить» или «сделать снова». В контексте программного обеспечения «итерация» означает повторное выполнение процедуры несколько раз, последовательно и, как правило, с ожиданием завершения. Спецификация ECMAScript 6 представляет две высокоуровневые особенности языка — итераторы и генераторы — для обеспечения более чистой, быстрой и простой итерации.

ВВЕДЕНИЕ В ИТЕРАЦИЮ

В JavaScript один из самых простых примеров итерации — цикл подсчета:

```
for (let i = 1; i <= 10; ++i) {
    console.log(i);
}
```

Циклы являются фундаментальным итеративным инструментом, потому что они позволяют указать, сколько итераций должно происходить и что делается во время

каждой итерации. Каждая итерация цикла завершает выполнение до начала другой, и порядок, в котором происходит каждая итерация, четко определен.

Итерация может происходить в упорядоченных коллекциях элементов. (Рассмотрим «упорядоченный» в этом контексте, чтобы подразумевать, что существует принятая последовательность, в которой должны быть пройдены все элементы с определенным начальным и конечным элементами.) В JavaScript наиболее распространенным примером такой упорядоченной коллекции является массив.

```
let collection = ['foo', 'bar', 'baz'];

for (let index = 0; index < collection.length; ++index) {
    console.log(collection[index]);
}
```

Поскольку массив имеет известную длину и поскольку каждый элемент в этом массиве может быть извлечен через его индекс, весь массив может быть пройден по порядку путем увеличения диапазона возможных индексов.

Фундаментальная процедура, происходящая в таком цикле, не идеальна по нескольким причинам.

- **Итерация по структуре данных требует определенных знаний о том, как ее использовать.** Каждый элемент в массиве может быть извлечен только путем ссылки на объект массива, а затем извлечения элемента по определенному индексу с помощью оператора `[]`. Это не обобщает другие структуры данных.
- **Порядок обхода не присущ структуре данных.** Увеличение целого числа для доступа к последовательным индексам зависит от типа массива и не распространяется на другие структуры данных, которые имеют неявное упорядочение.

ES5 представил метод `Array.prototype.forEach`, который стал ближе к требуемому (но все еще не является идеальным решением):

```
let collection = ['foo', 'bar', 'baz'];

collection.forEach((item) => console.log(item));
// foo
// bar
// baz
```

Это решает проблему отдельного отслеживания индекса и извлечения элементов через объект массива. Однако нет способа завершить данную итерацию; метод ограничен массивами, а структура обратного вызова громоздка.

В более ранних версиях ECMAScript выполнение итерации требовало использования циклов или других вспомогательных конструкций, что становилось все труднее по мере увеличения сложности кода. Многие языки решают эту проблему с помощью конструкции на родном языке, которая позволяет выполнять итерацию без специальных знаний о том, как на самом деле происходила итерация, а решением является *паттерн Итератор*. Python, Java, C ++ и многие другие языки

предлагают первоклассную поддержку этого шаблона, в том числе и JavaScript со спецификацией ES6.

ПАТТЕРН ИТЕРАТОР

Паттерн Итератор (в частности, в контексте ECMAScript) описывает решение, в котором что-то может быть описано как «итерируемое» и может реализовывать формальный интерфейс `Iterable` и использоваться объектом `Iterator`.

Понятие «итеративный» намеренно абстрактно. Зачастую итерация принимает форму объекта коллекции, такого как массив или набор, оба из которых имеют конечное число счетных элементов и однозначный порядок обхода:

```
// Массивы имеют конечное количество счетных элементов
// Обход "по порядку" посещает каждый индекс в порядке возрастания индекса
let arr = [3, 1, 4];

// Наборы имеют конечное количество счетных элементов
// Обход "по порядку" посещает каждое значение в порядке вставки
let set = new Set().add(3).add(1).add(4);
```

Однако итерируемый объект необязательно должен быть связан с объектом коллекции. Он также может быть связан с чем-то, что ведет себя только как массив — например, с циклом подсчета, описанным ранее в этой главе. Значения, сгенерированные в этом цикле, являются временными, и все же такой цикл выполняет итерацию. И этот счетный цикл, и массив могут вести себя как итеративные.

> **ПРИМЕЧАНИЕ** Переходные итерации могут быть реализованы как генераторы, которые будут рассмотрены позже в этой главе.

Все, что реализует интерфейс `Iterable`, может «потребляться» объектом, который реализует интерфейс `Iterator`. *Итератор* — это отдельный объект, созданный по требованию и предназначенный для однократного использования. Каждый итератор связан с *итерируемым объектом*, и итератор предоставляет API для обхода итерируемого объекта за один раз. Итератору не нужно понимать структуру итерируемого объекта, с которым он связан; он только должен знать, как получить последовательные значения. Именно это разделение интересов делает соглашение `Iterable/Iterator` таким полезным.

Протокол Iterable

Реализация интерфейса `Iterable` требует от объекта как способности идентифицировать себя как объект, поддерживающей итерации, так и способности создавать объект, реализующий интерфейс `Iterator`. В ECMAScript это означает, что он должен предоставлять свойство «итератор по умолчанию», снабженное специальным ключом `Symbol.iterator`. Это свойство итератора по умолчанию должно ссылаться

на функцию фабрики итераторов, которая будет производить новый итератор при вызове.

Многие встроенные типы реализуют интерфейс `Iterable`:

- строки;
- массивы;
- `Map`;
- наборы;
- объект `arguments`;
- некоторые типы коллекций DOM, такие как `NodeList`.

Проверка существования данного свойства итератора по умолчанию предоставит фабричную функцию:

```
let num = 1;
let obj = {};

// Данные типы не имеют фабрики итераторов
console.log(num[Symbol.iterator]);    // undefined
console.log(obj[Symbol.iterator]);    // undefined

let str = 'abc';
let arr = ['a', 'b', 'c'];
let map = new Map().set('a', 1).set('b', 2).set('c', 3);
let set = new Set().add('a').add('b').add('c');
let els = document.querySelectorAll('div');

// Все данные типы имеют фабрики итераторов
console.log(str[Symbol.iterator]);    // f values() { [вложенный код] }
console.log(arr[Symbol.iterator]);    // f values() { [вложенный код] }
console.log(map[Symbol.iterator]);    // f values() { [вложенный код] }
console.log(set[Symbol.iterator]);    // f values() { [вложенный код] }
console.log(els[Symbol.iterator]);    // f values() { [вложенный код] }

// Вызов фабричной функции производит Iterator
console.log(str[Symbol.iterator]());    // StringIterator {}
console.log(arr[Symbol.iterator]());    // ArrayIterator {}
console.log(map[Symbol.iterator]());    // MapIterator {}
console.log(set[Symbol.iterator]());    // SetIterator {}
console.log(els[Symbol.iterator]());    // ArrayIterator {}
```

Для создания итератора необязательно явно вызывать эту фабричную функцию. Все, что реализует этот протокол, автоматически совместимо с любыми языковыми функциями, которые допускают итерируемость. Эти встроенные языковые конструкции включают в себя:

- цикл `for...of`;
- деструктурирование массива;
- оператор распространения;
- `Array.from()`;

- конструирование `Set`;
- конструирование `Map`;
- `Promise.all()`, ожидающий промисы для перебора;
- `Promise.race()`, ожидающий промисы для перебора;
- оператор `yield*`, используемый в генераторах.

За кулисами эти встроенные в язык конструкции вызывают фабричную функцию на заданном итерируемом объекте для создания итератора:

```
let arr = ['foo', 'bar', 'baz'];

// Циклы for...of
for (let el of arr) {
    console.log(el);
}
// foo
// bar
// baz

// Деструктурирование массива
let [a, b, c] = arr;
console.log(a, b, c); // foo, bar, baz

// Оператор распространения
let arr2 = [...arr];
console.log(arr2); // ['foo', 'bar', 'baz']

// Array.from()
let arr3 = Array.from(arr);
console.log(arr3); // ['foo', 'bar', 'baz']

// Конструктор Set
let set = new Set(arr);
console.log(set); // Set(3) {'foo', 'bar', 'baz'}

// Конструктор Map
let pairs = arr.map((x, i) => [x, i]);
console.log(pairs); // [['foo', 0], ['bar', 1], ['baz', 2]]
let map = new Map(pairs);
console.log(map); // Map(3) { 'foo'=>0, 'bar'=>1, 'baz'=>2 }
```

Объект все еще реализует интерфейс `Iterable`, если родительский класс вверху по цепочке прототипов реализует этот интерфейс:

```
class FooArray extends Array {}
let fooArr = new FooArray('foo', 'bar', 'baz');

for (let el of fooArr) {
    console.log(el);
}
// foo
// bar
// baz
```

Протокол Iterator

Итератор — это единожды используемый объект, который будет перебирать все итерируемые коллекции, с которыми он связан. Iterator API использует метод `next()` для продвижения через итерируемый объект. Каждый раз при вызове `next()` он возвращает объект `IteratorResult`, содержащий следующее значение в итераторе. Текущая позиция, в которой находится итератор, не может быть известна без вызова метода `next()`.

Метод `next()` возвращает объект с двумя свойствами: `done`, которое является логическим значением, указывающим, можно ли снова вызывать `next()` для получения большего количества значений, и значением, которое будет содержать следующее значение в итерируемом объекте или `undefined`, если `done` равен `true`. Выражение `done:true` называется «истощение». Это можно продемонстрировать с помощью простого массива:

```
// Итерируемый объект
let arr = ['foo', 'bar'];

// Фабрика итераторов
console.log(arr[Symbol.iterator]); // f values() { [вложенный код] }

// Итератор
let iter = arr[Symbol.iterator]();
console.log(iter); // ArrayIterator {}

// Процесс итерации
console.log(iter.next()); // { done: false, value: 'foo' }
console.log(iter.next()); // { done: false, value: 'bar' }
console.log(iter.next()); // { done: true, value: undefined }
```

Массивы перебираются по порядку, создавая итератор и вызывая метод `next()`, пока он не перестанет генерировать новые значения. Обратите внимание, что итератор не знает, как получить следующие значения внутри итерируемого объекта, и при этом он не знает размер итерируемого объекта. Как только итератор достигает состояния `done:true`, вызов `next()` становится идемпотентным:

```
let arr = ['foo'];
let iter = arr[Symbol.iterator]();
console.log(iter.next()); // { done: false, value: 'foo' }
console.log(iter.next()); // { done: true, value: undefined }
console.log(iter.next()); // { done: true, value: undefined }
console.log(iter.next()); // { done: true, value: undefined }
```

Каждый итератор представляет одноразовый упорядоченный обход итерируемого объекта. Различные экземпляры не знают друг друга и будут проходить итерацию независимо:

```
let arr = ['foo', 'bar'];
let iter1 = arr[Symbol.iterator]();
let iter2 = arr[Symbol.iterator]();

console.log(iter1.next()); // { done: false, value: 'foo' }
```

```
console.log(iter2.next()); // { done: false, value: 'foo' }
console.log(iter2.next()); // { done: false, value: 'bar' }
console.log(iter1.next()); // { done: false, value: 'bar' }
```

Итератор не привязан к экземпляру итерируемого объекта; он просто использует курсор, чтобы отслеживать его продвижение через итерацию. Если итерируемый объект видоизменяется во время итерации, итератор будет учитывать изменения:

```
let arr = ['foo', 'baz'];
let iter = arr[Symbol.iterator]();

console.log(iter.next()); // { done: false, value: 'foo' }

// Вставка значения в середину массива
arr.splice(1, 0, 'bar');

console.log(iter.next()); // { done: false, value: 'bar' }
console.log(iter.next()); // { done: false, value: 'baz' }
console.log(iter.next()); // { done: true, value: undefined }
```

ПРИМЕЧАНИЕ Итератор поддерживает ссылку на итерируемый объект, поэтому имейте в виду, что существование итератора предотвратит захват итерируемого объекта при сборке мусора.

Термин «итератор» может быть несколько туманным, поскольку он относится к обобщенной концепции итерации, интерфейсу и формальным классам типа итератора. В следующем примере сравниваются явная реализация итератора и встроенная реализация итератора:

```
// Данный класс реализует интерфейс Iterable.
// Вызов фабрики итераторов по умолчанию вернет
// объект итератора, который реализует интерфейс Iterator.
class Foo {
    [Symbol.iterator]() {
        return {
            next() {
                return { done: false, value: 'foo' };
            }
        }
    }
}
let f = new Foo();

// Записывает объект, который реализует интерфейс Iterator
console.log(f[Symbol.iterator]()); // { next: f() {} }

// Тип Array реализует интерфейс Iterable.
// Вызов итератора по умолчанию для типа Array
// создаст экземпляр ArrayIterator.
let a = new Array();

// Записывает экземпляр ArrayIterator
console.log(a[Symbol.iterator]()); // Array Iterator {}
```

Определение пользовательского итератора

Как и интерфейс `Iterable`, любой объект, который реализует интерфейс `Iterator`, может использоваться в качестве итератора. Рассмотрим следующий пример, где класс `Counter` определен для итерации определенного количества раз:

```
class Counter {
    // Экземпляр Counter должен итерироваться <limit> раз
    constructor(limit) {
        this.count = 1;
        this.limit = limit;
    }

    next() {
        if (this.count <= this.limit) {
            return { done: false, value: this.count++ };
        } else {
            return { done: true, value: undefined };
        }
    }

    [Symbol.iterator]() {
        return this;
    }
}

let counter = new Counter(3);

for (let i of counter) {
    console.log(i);
}
// 1
// 2
// 3
```

Это удовлетворяет интерфейсу `Iterator`, но эта реализация не является оптимальной, поскольку каждый экземпляр класса может попасть в итерацию только один раз:

```
for (let i of counter) { console.log(i); }
// 1
// 2
// 3

for (let i of counter) { console.log(i); }
// (ничего не записано)
```

Чтобы можно было создавать несколько итераторов из одной итерации, счетчик должен создаваться для каждого итератора. Чтобы решить эту проблему, можно возвращать объект итератора с переменными счетчика, доступными через замыкание:

```
class Counter {
    constructor(limit) {
        this.limit = limit;
    }
```

```
    [Symbol.iterator]() {
        let count = 1,
            limit = this.limit;
        return {
            next() {
                if (count <= limit) {
                    return { done: false, value: count++ };
                } else {
                    return { done: true, value: undefined };
                }
            }
        };
    }
}

let counter = new Counter(3);

for (let i of counter) { console.log(i); }
// 1
// 2
// 3

for (let i of counter) { console.log(i); }
// 1
// 2
// 3
```

Каждый созданный таким образом итератор также реализует интерфейс `Iterable`. Свойство `Symbol.iterator` относится к фабрике, которая будет возвращать тот же итератор:

```
let arr = ['foo', 'bar', 'baz'];
let iter1 = arr[Symbol.iterator]();

console.log(iter1[Symbol.iterator]); // f values() { [вложенный код] }

let iter2 = iter1[Symbol.iterator]();

console.log(iter1 === iter2); // true
```

Поскольку каждый итератор также реализует интерфейс `Iterable`, их можно использовать везде, где ожидается итерация, например в цикле `for...of`:

```
let arr = [3, 1, 4];
let iter = arr[Symbol.iterator]();

for (let item of arr) { console.log(item); }
// 3
// 1
// 4

for (let item of iter) { console.log(item); }
// 3
// 1
// 4
```

Преждевременное завершение итератора

Необязательный метод `return()` позволяет указать поведение, которое будет выполняться, только если итератор закрыт преждевременно. «Закрытие» итератора происходит, когда конструкция, выполняющая итерацию, хочет указать итератору, что она не намерена завершать обход до конца. Сценарии, когда это может произойти:

- цикл `for...of` завершается раньше времени с помощью `break`, `continue`, `return` или `throw`;
- операция деструктурирования использует не все значения.

Метод `return()` должен возвращать правильный объект `IteratorResult`. Простая реализация итератора должна всего лишь возвращать `{done: true}`, поскольку возвращаемое значение используется только в контексте генераторов, о которых я расскажу позже в этой главе.

Как показано в следующем коде, встроенная языковая конструкция автоматически вызовет метод `return()`, как только она определит, что существуют дополнительные значения для перебора, которые не будут использованы.

```
class Counter {
    constructor(limit) {
        this.limit = limit;
    }

    [Symbol.iterator]() {
        let count = 1,
            limit = this.limit;
        return {
            next() {
                if (count <= limit) {
                    return { done: false, value: count++ };
                } else {
                    return { done: true };
                }
            },
            return() {
                console.log('Exiting early');
                return { done: true };
            }
        };
    }
}

let counter1 = new Counter(5);

for (let i of counter1) {
    if (i > 2) {
        break;
    }
    console.log(i);
}
// 1
// 2
```

```
// Exiting early

let counter2 = new Counter(5);

try {
    for (let i of counter2 {
        if (i > 2) {
            throw 'err';
        }
        console.log(i);
    }
} catch(e) {}
// 1
// 2
// Exiting early

let counter3 = new Counter(5);

let [a, b] = counter3;
// Exiting early
```

Если итератор не закрыт, то можно перезапустить итерацию с места остановки. Например, итераторы массива не могут быть закрыты:

```
let a = [1, 2, 3, 4, 5];
let iter = a[Symbol.iterator]();

for (let i of iter) {
    console.log(i);
    if (i > 2) {
        break
    }
}
// 1
// 2
// 3

for (let i of iter) {
    console.log(i);
}
// 4
// 5
```

Поскольку метод `return()` является необязательным, не все итераторы могут быть закрыты. Можно установить, является ли итератор закрываемым, путем проверки, является ли возвращаемое свойство в экземпляре итератора функциональным объектом. Однако простое добавление метода к итератору, не подлежащему закрытию, *не* сделает его закрытым, поскольку вызов `return()` не приводит итератор в закрытое состояние. Однако метод `return()` будет по-прежнему вызываться:

```
let a = [1, 2, 3, 4, 5];
let iter = a[Symbol.iterator]();

iter.return = function() {
    console.log('Exiting early');
    return { done: true };
```

```
};

for (let i of iter) {
    console.log(i);
    if (i > 2) {
        break
    }
}
// 1
// 2
// 3
// Exiting early

for (let i of iter) {
    console.log(i);
}
// 4
// 5
```

ГЕНЕРАТОРЫ

Генераторы представляют собой восхитительно гибкую конструкцию, представленную в спецификации ECMAScript 6, которая дает возможность приостанавливать и возобновлять выполнение кода внутри одного функционального блока. Последствия этой новой способности глубоки; среди прочего, она позволяет определять пользовательские итераторы и реализовывать сопрограммы.

Основы генераторов

Генераторы принимают форму функции, а сам генератор обозначается звездочкой. Везде, где определение функции является допустимым, определение функции генератора также допустимо:

```
// Объявление функции генератора
function* generatorFn() {}

// Функциональное выражение генератора
let generatorFn = function* () {}

// Функция генератора литерала объекта
let foo = {
    * generatorFn() {}
}

// Функция генератора метода экземпляра класса
class Foo {
    * generatorFn() {}
}

// Функция генератора статического метода класса
class Bar {
    static * generatorFn() {}
}
```

ПРИМЕЧАНИЕ Стрелочные функции не могут быть использованы в качестве функций генератора.

Функция будет считаться генератором независимо от пробелов, окружающих звездочку:

```
// Эквивалентные функции генератора:
function* generatorFnA() {}
function *generatorFnB() {}
function * generatorFnC() {}

// Эквивалентные методы генератора:
class Foo {
    *generatorFnD() {}
    * generatorFnE() {}
}
```

При вызове функции генератора создают *объект генератора*. Объекты генератора начинаются в состоянии приостановленного выполнения. Подобно итераторам, объекты генератора реализуют интерфейс `Iterator` и, следовательно, имеют метод `next()`, который при вызове говорит генератору начать или возобновить выполнение.

```
function* generatorFn() {}

const g = generatorFn();

console.log(g);         // generatorFn {<приостановленный>}
console.log(g.next);    // f next() { [вложенный код] }
```

Возвращаемое значение метода `next()` совпадает с возвращаемым значением итератора со свойством `done` и `value`. Функция генератора с пустым телом функции будет действовать как проход; однократный вызов `next()` приведет к тому, что генератор достигнет состояния `done: true`.

```
function* generatorFn() {}

let generatorObject = generatorFn();

console.log(generatorObject);           // generatorFn {<приостановленный>}
console.log(generatorObject.next());    // { done: true, value: undefined }
```

Свойство `value` — это возвращаемое значение функции генератора, которое по умолчанию задано как `undefined` и может быть переопределено через возвращаемое значение функции генератора.

```
function* generatorFn() {
    return 'foo';
}

let generatorObject = generatorFn();

console.log(generatorObject);           // generatorFn {<приостановленный>}
console.log(generatorObject.next());    // { done: true, value: 'foo' }
```

Выполнение функции генератора начнется только после первоначального вызова `next()`:

```
function* generatorFn() {
    console.log('foobar');
}

// Ничего не записывается при начальном вызове функции генератора
let generatorObject = generatorFn();

generatorObject.next(); // foobar
```

Объекты генератора реализуют интерфейс `Iterable`, и их итератор по умолчанию ссылается сам на себя:

```
function* generatorFn() {}

console.log(generatorFn);
// f* generatorFn() {}

console.log(generatorFn()[Symbol.iterator]);
// f [Symbol.iterator]() {вложенный код}

console.log(generatorFn());
// generatorFn {<приостановленный>}

console.log(generatorFn()[Symbol.iterator]());
// generatorFn {<приостановленный}

const g = generatorFn();

console.log(g === g[Symbol.iterator]());
// true
```

Прерывание выполнения с помощью yield

Ключевое слово `yield` позволяет генераторам останавливать и запускать выполнение, и именно это делает генераторы действительно полезными. Функции генератора продолжат нормальное выполнение, пока не встретят ключевое слово `yield`. При обнаружении ключевого слова выполнение будет остановлено, а состояние области действия функции сохранено. Выполнение будет возобновлено только после вызова метода `next()` для объекта генератора:

```
function* generatorFn() {
    yield;
}

let generatorObject = generatorFn();

console.log(generatorObject.next()); // { done: false, value: undefined }
console.log(generatorObject.next()); // { done: true, value: undefined }
```

Ключевое слово `yield` ведет себя как промежуточная функция возврата, а полученное значение доступно внутри объекта, возвращаемого методом `next()`. Функция

генератора, завершающая работу через ключевое слово `yield`, будет иметь значение `false`; функция генератора, завершающаяся через ключевое слово `return`, будет иметь значение `true`:

```
function* generatorFn() {
    yield 'foo';
    yield 'bar';
    return 'baz';
}

let generatorObject = generatorFn();

console.log(generatorObject.next()); // { done: false, value: 'foo' }
console.log(generatorObject.next()); // { done: false, value: 'bar' }
console.log(generatorObject.next()); // { done: true, value: 'baz' }
```

Ход выполнения в функции генератора определяется для каждого экземпляра объекта генератора. Вызов `next()` для одного объекта генератора не влияет на другие:

```
function* generatorFn() {
    yield 'foo';
    yield 'bar';
    return 'baz';
}

let generatorObject1 = generatorFn();
let generatorObject2 = generatorFn();

console.log(generatorObject1.next());    // { done: false, value: 'foo' }
console.log(generatorObject2.next());    // { done: false, value: 'foo' }
console.log(generatorObject2.next());    // { done: false, value: 'bar' }
console.log(generatorObject1.next());    // { done: false, value: 'bar' }
```

Ключевое слово `yield` можно использовать только внутри функции генератора; в любом другом случае его использование приведет к ошибке. Как и ключевое слово `return` функции, ключевое слово `yield` должно появляться сразу внутри определения функции генератора. Дальнейшее вложение его в негенераторную функцию вызовет синтаксическую ошибку:

```
// верно
function* validGeneratorFn() {
    yield;
}

// неверно
function* invalidGeneratorFnA() {
    function a() {
        yield;
    }
}

// неверно
function* invalidGeneratorFnB() {
```

```
    const b = () => {
        yield;
    }
}

// неверно
function* invalidGeneratorFnC() {
    (() => {
        yield;
    })();
}
```

Использование объекта генератора в качестве итерируемого

Вряд ли вам часто понадобится явный вызов `next()` для объекта генератора. Вместо этого генераторы гораздо полезнее, когда используются как итерируемые объекты, как показано ниже:

```
function* generatorFn() {
    yield 1;
    yield 2;
    yield 3;
}

for (const x of generatorFn()) {
    console.log(x);
}
// 1
// 2
// 3
```

Это может быть особенно полезно, когда возникает необходимость определения пользовательских итераций. Например, часто необходимо определить итерацию, которая создаст итератор, выполняющийся определенное количество раз. С генератором этого можно достичь просто с помощью цикла:

```
function* nTimes(n) {
    while(n--) {
        yield;
    }
}

for (let _ of nTimes(3)) {
    console.log('foo');
}
// foo
// foo
// foo
```

Параметр функции одного генератора управляет количеством итераций цикла. Когда `n` достигает 0, условие `while` становится ложным, цикл завершается и функция генератора возвращается.

Использование yield для ввода и вывода

Ключевое слово `yield` также ведет себя как параметр промежуточной функции. Ключевое слово `yield` при выполнении последнего приостановленного генератора будет принимать первое значение, переданное `next()`. Несколько странным кажется то, что значение, предоставленное для первого вызова `next()`, не используется, так как этот `next()` используется для первого запуска функции генератора:

```
function* generatorFn(initial) {
    console.log(initial);
    console.log(yield);
    console.log(yield);
}

let generatorObject = generatorFn('foo');

generatorObject.next('bar');    // foo
generatorObject.next('baz');    // baz
generatorObject.next('qux');    // qux
```

Ключевое слово `yield` может одновременно использоваться как для ввода, так и для вывода:

```
function* generatorFn() {
    return yield 'foo';
}

let generatorObject = generatorFn();

console.log(generatorObject.next());         // { done: false, value: 'foo' }
console.log(generatorObject.next('bar'));    // { done: true, value: 'bar' }
```

Поскольку функция должна оценивать все выражение, чтобы определить возвращаемое значение, она будет приостанавливать выполнение при обнаружении ключевого слова `yield` и оценивать текущее значение `yield` — `foo`. Последующий вызов `next()` предоставляет значение `bar` в качестве значения для `yield`, а это в свою очередь оценивается как возвращаемое значение функции генератора.

Ключевое слово `yield` не ограничено однократным использованием. Функция генератора бесконечного счета может быть определена следующим образом:

```
function* generatorFn() {
    for (let i = 0;;++i) {
        yield i;
    }
}

let generatorObject = generatorFn();

console.log(generatorObject.next().value); // 0
console.log(generatorObject.next().value); // 1
console.log(generatorObject.next().value); // 2
console.log(generatorObject.next().value); // 3
console.log(generatorObject.next().value); // 4
console.log(generatorObject.next().value); // 5
...
```

Предположим, нужно определить функцию генератора, которая будет запускать итерацию настраиваемое число раз и генерировать индекс итерации. Это может быть достигнуто путем создания экземпляра нового массива, но такое же поведение может быть достигнуто и без массива:

```
function* nTimes(n) {
    for (let i = 0; i < n; ++i) {
        yield i;
    }
}

for (let x of nTimes(3)) {
    console.log(x);
}
// 0
// 1
// 2
```

Альтернативная реализация, приведенная ниже, немного менее многословна:

```
function* nTimes(n) {
    let i = 0;
    while(n--) {
        yield i++;
    }
}

for (let x of nTimes(3)) {
    console.log(x);
}
// 0
// 1
// 2
```

Использование генераторов таким способом обеспечивает полезный способ реализации диапазонов или заполнения массивов:

```
function* range(start, end) {
    let i = start;
    while(end > start) {
        yield start++;
    }
}

for (const x of range(4, 7)) {
    console.log(x);
}
// 4
// 5
// 6

function* zeroes(n) {
    while(n--) {
        yield 0;
    }
}

console.log(Array.from(zeroes(8))); // [0, 0, 0, 0, 0, 0, 0, 0]
```

Использование yield с Iterable

Можно дополнить поведение `yield`, чтобы он перебирал итерацию и выводил ее содержимое по одному. Это можно сделать с помощью звездочки:

```
// функция generatorFn эквивалентна этому:
// function* generatorFn() {
//     for (const x of [1, 2, 3]) {
//         yield x;
//     }
// }
function* generatorFn() {
    yield* [1, 2, 3];
}

let generatorObject = generatorFn();

for (const x of generatorFn()) {
    console.log(x);
}
// 1
// 2
// 3
```

Как и звездочка функции генератора, пробел вокруг звездочки `yield` не изменит его поведения:

```
function* generatorFn() {
    yield* [1, 2];
    yield *[3, 4];
    yield * [5, 6];
}

for (const x of generatorFn()) {
    console.log(x);
}
// 1
// 2
// 3
// 4
// 5
// 6
```

Поскольку `yield*` фактически просто сериализует итерируемый объект в последовательные получаемые значения, его использование ничем не отличается от размещения `yield` внутри цикла. Эти две функции генератора эквивалентны по поведению:

```
function* generatorFnA() {
    for (const x of [1, 2, 3]) {
        yield x;
    }
}

for (const x of generatorFnA()) {
```

```
    console.log(x);
}
// 1
// 2
// 3

function* generatorFnB() {
    yield* [1, 2, 3];
}

for (const x of generatorFnB()) {
    console.log(x);
}
// 1
// 2
// 3
```

Значение `yield*` — это свойство `value`, сопровождающее `done:true` соответствующего итератора. Для обычных итераторов это значение будет `undefined`:

```
function* generatorFn() {
    console.log('iter value:', yield* [1, 2, 3]);
}

for (const x of generatorFn()) {
    console.log('value:', x);
}
// value: 1
// value: 2
// value: 3
// iter value: undefined
```

Для итераторов, созданных из функции генератора, это значение примет форму любого значения, возвращаемого функцией генератора:

```
function* innerGeneratorFn() {
    yield 'foo';
    return 'bar';
}

function* outerGeneratorFn(genObj) {
    console.log('iter value:', yield* innerGeneratorFn());
}

for (const x of outerGeneratorFn()) {
    console.log('value:', x);
}
// value: foo
// iter value: bar
```

Рекурсивные алгоритмы с использованием yield*

`yield*` наиболее полезен при использовании в рекурсивной операции, где генератор может сам вызывать `yield`. Рассмотрим следующий пример:

```
function* nTimes(n) {
    if (n > 0) {
        yield* nTimes(n - 1);
        yield n - 1;
    }
}

for (const x of nTimes(3)) {
    console.log(x);
}
// 0
// 1
// 2
```

В данном примере каждый генератор сначала возвращает каждое значение из вновь созданного объекта генератора, а затем возвращает единственное целое число. Результатом этого является то, что функция генератора будет рекурсивно уменьшать значение счетчика и создавать экземпляр другого объекта генератора, который на верхнем уровне создаст одну итерацию, возвращающую инкрементные целые числа.

Использование рекурсивной структуры генератора и `yield*` позволяет элегантно выражать рекурсивные алгоритмы. Рассмотрим следующую реализацию графа, которая генерирует случайный двунаправленный граф:

```
class Node {
    constructor(id) {
        this.id = id;
        this.neighbors = new Set();
    }

    connect(node) {
        if (node !== this) {
            this.neighbors.add(node);
            node.neighbors.add(this);
        }
    }
}

class RandomGraph {
    constructor(size) {
        this.nodes = new Set();

        // Создание узлов
        for (let i = 0; i < size; ++i) {
            this.nodes.add(new Node(i));
        }

        // Случайное соединение узлов
        const threshold = 1 / size;
        for (const x of this.nodes) {
            for (const y of this.nodes) {
                if (Math.random() < threshold) {
                    x.connect(y);
                }
```

```
            }
        }
    }

    // Это только для отладки
    print() {
        for (const node of this.nodes) {
            const ids = [...node.neighbors]
                        .map((n) => n.id)
                        .join(',');

            console.log('${node.id}: ${ids}');
        }
    }
}

const g = new RandomGraph(6);

g.print();
// Пример вывода:
// 0: 2,3,5
// 1: 2,3,4,5
// 2: 1,3
// 3: 0,1,2,4
// 4: 2,3
// 5: 0,4
```

Структура данных графа хорошо подходит для рекурсивного обхода, и использование рекурсивного генератора позволяет добиться именно этого. Здесь функция генератора должна принять итерируемый объект, вывести каждое значение в нем и рекурсивно использовать каждое значение. Простым примером использования этого механизма была бы проверка связности графа, это означает, что в нем нет никаких узлов, которые не могут быть достигнуты. Этот тест можно выполнить, начиная с одного узла и пытаясь полностью посетить каждый узел. Результатом является очень краткая реализация первого обхода графа:

```
class Node {
    constructor(id) {
        ...
    }

    connect(node) {
        ...
    }
}

class RandomGraph {
    constructor(size) {
        ...
    }

    print() {
        ...
    }
```

```
isConnected() {
    const visitedNodes = new Set();

    function* traverse(nodes) {
        for (const node of nodes) {
            if (!visitedNodes.has(node)) {
                yield node;
                yield* traverse(node.neighbors);
            }
        }
    }

    // Получение первого узла в Set
    const firstNode = this.nodes[Symbol.iterator]().next().value;

    // Использование рекурсивного генератора для обхода каждого узла
    for (const node of traverse([firstNode])) {
        visitedNodes.add(node);
    }

    return visitedNodes.size === this.nodes.size;
  }
}
```

Использование генератора в качестве итератора по умолчанию

Поскольку объекты-генераторы реализуют интерфейс `Iterable` и поскольку и функции-генераторы, и итератор по умолчанию вызываются для создания итератора, генераторы отлично подходят для использования в качестве итераторов по умолчанию. Ниже приведен простой пример, где итератор по умолчанию может выдавать содержимое класса в одной строке:

```
class Foo {
    constructor() {
        this.values = [1, 2, 3];
    }
    * [Symbol.iterator]() {
        yield* this.values;
    }
}

const f = new Foo();

for (const x of f) {
    console.log(x);
}
// 1
// 2
// 3
```

Здесь цикл `for...of` вызывает стандартный итератор, который является функцией генератора, и создает объект генератора. Объект генератора является итеративным и поэтому подходящим для использования в итерации.

Преждевременное завершение генераторов

Подобно итераторам, генераторы также поддерживают концепцию «закрываемости». Чтобы объект мог реализовать интерфейс `Iterator`, он должен иметь метод `next()` и, что необязательно, метод `return()` на случай досрочного завершения итератора. Объект генератора имеет оба этих метода и дополнительный третий метод `throw()`.

```
function* generatorFn() {}

const g = generatorFn();

console.log(g);           // generatorFn {<приостановленный>}
console.log(g.next);      // f next() { [вложенный код] }
console.log(g.return);    // f return() { [вложенный код] }
console.log(g.throw);     // f throw() { [вложенный код] }
```

Методы `return()` и `throw()` — два метода, которые можно использовать для приведения генератора в закрытое состояние.

Метод return()

Метод `return()` переведет генератор в закрытое состояние, а значение, переданное в `return()`, будет значением, переданным в конечный объект итератора:

```
function* generatorFn() {
    for (const x of [1, 2, 3]) {
        yield x;
    }
}

const g = generatorFn();

console.log(g);              // generatorFn {<suspended>}
console.log(g.return(4));    // { done: true, value: 4 }
console.log(g);              // generatorFn {<closed>}
```

В отличие от итераторов все объекты генератора имеют метод `return()`, который переводит его в закрытое состояние, куда он не может выйти после его достижения. Последующий вызов `next()` раскроет состояние `done:true`, но любое возвращаемое значение не сохраняется и не распространяется:

```
function* generatorFn() {
    for (const x of [1, 2, 3]) {
        yield x;
    }
}

const g = generatorFn();

console.log(g.next());       // { done: false, value: 1 }
console.log(g.return(4));    // { done: true, value: 4 }
console.log(g.next());       // { done: true, value: undefined }
console.log(g.next());       // { done: true, value: undefined }
console.log(g.next());       // { done: true, value: undefined }
```

Встроенные языковые конструкции, такие как цикл `for...of`, будут разумно игнорировать любые значения, возвращаемые внутри `done:true` объектом `IteratorObject`.

```
function* generatorFn() {
    for (const x of [1, 2, 3]) {
        yield x;
    }
}

const g = generatorFn();

for (const x of g) {
    if (x > 1) {
        g.return(4);
    }
    console.log(x);
}
// 1
// 2
```

Метод throw()

Метод `throw()` вставит переданную в него ошибку в объект генератора в точке, в которой он приостановлен. Если ошибка не обработана, генератор закроется:

```
function* generatorFn() {
    for (const x of [1, 2, 3]) {
        yield x;
    }
}

const g = generatorFn();

console.log(g); // generatorFn {<приостановленный>}
try {
    g.throw('foo');
} catch (e) {
    console.log(e); // foo
}
console.log(g); // generatorFn {<закрытый>}
```

Однако если ошибка обрабатывается внутри функции генератора, то тот не закроется и может возобновить выполнение. Обработка ошибок пропустит этот выход, поэтому в этом примере вы заметите, что она пропускает значение. Рассмотрим еще один пример:

```
function* generatorFn() {
    for (const x of [1, 2, 3]) {
        try {
            yield x;
        } catch(e) {}
    }
}
```

```
const g = generatorFn();

console.log(g.next()); // { done: false, value: 1}
g.throw('foo');
console.log(g.next()); // { done: false, value: 3}
```

В этом примере генератор приостанавливает выполнение по ключевому слову `yield` в блоке `try/catch`. Пока он приостановлен, `throw()` добавляет ошибку `foo`, которая генерируется ключевым словом `yield`. Поскольку эта ошибка возникает в блоке `try/catch` генератора, она впоследствии перехватывается, пока еще находится внутри генератора. Однако поскольку `yield` выдает эту ошибку, значение `2` не будет выдаваться генератором. Вместо этого функция генератора продолжает выполнение, переходя к следующей итерации цикла, где она снова встречает ключевое слово `yield` — на этот раз получая значение `3`.

> **ПРИМЕЧАНИЕ** Если объект генератора еще не начал выполнение, вызов метода `throw()` не может быть перехвачен внутри функции, поскольку ошибка выдается из-за пределов функционального блока.

ИТОГИ

Итератор — это паттерн, который встречается практически во всех языках программирования. Спецификация ECMAScript 6 официально охватывает концепцию итерации, вводя в язык две формальные концепции: итераторы и генераторы.

Итератор — это интерфейс, который может быть реализован любым объектом и позволяет последовательно просматривать значения, которые он создает. Все, что реализует интерфейс `Iterable`, имеет свойство `Symbol.iterator`, которое ссылается на итератор по умолчанию. Итератор по умолчанию ведет себя как фабрика итераторов: функция, которая при вызове создает объект, реализующий интерфейс `Iterator`.

Последовательные значения приводятся из итератора с помощью метода `next()`, который возвращает `IteratorObject`. Этот объект содержит свойство `done`, логическое значение, указывающее, есть ли еще доступные значения, и свойство `value`, которое содержит текущее значение, предоставленное итератором. Этот интерфейс можно использовать вручную, повторно вызывая `next()`, или автоматически использовать встроенные потребители итераторов, такие как цикл `for...of`.

Генераторы — это специальный тип функции, которая при вызове создает объект генератора. Этот объект реализует интерфейс `Iterable` и поэтому может использоваться везде, где ожидается итерация. Генераторы уникальны тем, что поддерживают ключевое слово `yield`, которое используется для приостановки выполнения функции генератора. Ключевое слово `yield` также можно использовать для принятия ввода и вывода с помощью метода `next()`. При дополнении звездочкой ключевое слово `yield` будет служить для сериализации связанного итерируемого объекта.

8

Объекты, классы и объектно-ориентированное программирование

- Объекты
- Создание объектов
- Наследование
- Классы

В ECMA-262 объект определяется как «неупорядоченная коллекция свойств». Строго говоря, это означает, что объект является массивом значений без конкретного порядка. Каждое свойство или метод объекта определяется именем, которое сопоставлено со значением. По этой и ряду других причин, которые мы обсудим позже, полезно рассматривать ECMAScript-объекты как хеш-таблицы — группы пар имен и значений, в которых значениями могут быть данные или функции.

ОБЩИЕ СВЕДЕНИЯ ОБ ОБЪЕКТАХ

Канонический способ определения объекта — создание экземпляра типа `Object` и добавление к нему свойств и методов, например:

```
let person = new Object();
person.name = "Nicholas";
person.age = 29;
person.job = "Software Engineer";

person.sayName = function() {
    console.log(this.name);
};
```

В этом примере создается объект `person` со свойствами `name`, `age` и `job` и методом `sayName()`. Метод выводит значение `this.name`, которое разрешается в `person.name`. Когда-то это был самый популярный способ создания объектов, но теперь для этого обычно используют литералы объектов. С нотацией литералов объектов предыдущий пример можно переписать следующим образом:

```
let person = {
    name: "Nicholas",
    age: 29,
    job: "Software Engineer",
    sayName: function() {
        console.log(this.name);
    }
};
```

Этот код эквивалентен предыдущему примеру. Все свойства объектов создаются с определенными характеристиками, от которых зависит их поведение в JavaScript.

Типы свойств

В ECMA-262 характеристики свойств описываются с помощью внутренних атрибутов, которые подлежат реализации в интерпретаторах JavaScript и недоступны непосредственно в JavaScript. На то, что атрибут является внутренним, указывают двойные квадратные скобки, например `[[Enumerable]]`.

Свойства делятся на два типа: свойства с данными и свойства с функциями доступа.

Свойства с данными

Свойства с данными — это места для хранения значений, которые можно читать и записывать. Поведение свойств с данными описывают четыре атрибута:

- `[[Configurable]]` — указывает, можно ли удалить свойство с помощью оператора `delete`, изменить атрибуты свойства или преобразовать его в свойство с функциями доступа. По умолчанию этот атрибут имеет значение `true` у всех свойств, определенных непосредственно для объекта, как в предыдущем примере.
- `[[Enumerable]]` — указывает, будет ли свойство возвращаться в циклах `for-in`. По умолчанию этот атрибут имеет значение `true` у всех свойств, определенных непосредственно для объекта, как в предыдущем примере.

- `[[Writable]]` — указывает, можно ли изменить значение свойства. По умолчанию этот атрибут имеет значение `true` у всех свойств, определенных непосредственно для объекта, как в предыдущем примере.
- `[[Value]]` — содержит фактические данные. Это место, откуда читается значение свойства и куда сохраняются новые значения. По умолчанию этот атрибут имеет значение `undefined`.

При явном добавлении свойства к объекту (как в предыдущем примере) атрибутам `[[Configurable]]`, `[[Enumerable]]` и `[[Writable]]` присваивается значение `true`, а атрибуту `[[Value]]` — указанное значение, например:

```
let person = {
    name: "Nicholas"
};
```

Здесь создается свойство `name` со значением `"Nicholas"`, то есть атрибут `[[Value]]` свойства становится равен `"Nicholas"` и любые изменения значения сохраняются в этом месте.

Чтобы изменить для свойства предлагаемое по умолчанию значение какого-либо атрибута, нужно использовать метод `Object.defineProperty()`. Он принимает три аргумента: объект, для которого требуется добавить или изменить свойство, имя свойства и объект-дескриптор. Свойства дескриптора соответствуют именам атрибутов: `configurable`, `enumerable`, `writable` и `value`. Задав какие-либо или все свойства, можно изменить значения соответствующих атрибутов, например:

```
let person = {};
Object.defineProperty(person, "name", {
    writable: false,
    value: "Nicholas"
});

console.log(person.name);        // "Nicholas"
person.name = "Greg";
console.log(person.name);        // "Nicholas"
```

В этом примере создается доступное только для чтения свойство `name` со значением `"Nicholas"`. Значение такого свойства изменить нельзя. Любые попытки сделать это в нестрогом режиме игнорируются, а в строгом — приводят к ошибке.

Аналогичные правила действуют и при создании неконфигурируемых свойств:

```
let person = {};
Object.defineProperty(person, "name", {
    configurable: false,
    value: "Nicholas"
});

console.log(person.name);        // "Nicholas"
delete person.name;
console.log(person.name);        // "Nicholas"
```

Присвоение значения `false` атрибуту `configurable` означает, что удалить свойство из объекта невозможно. Выполнение оператора `delete` для этого свойства будет проигнорировано в нестрогом режиме и вызовет ошибку в строгом. Кроме того, свойство, определенное как неконфигурируемое, нельзя снова сделать конфигурируемым. Попытка вызвать метод `Object.defineProperty()` и изменить любой атрибут, кроме `writable`, приведет к ошибке:

```
let person = {};
Object.defineProperty(person, "name", {
    configurable: false,
    value: "Nicholas"
});

// возникает ошибка
Object.defineProperty(person, "name", {
    configurable: true,
    value: "Nicholas"
});
```

Таким образом, метод `Object.defineProperty()` можно вызывать для свойства много раз, но присвоение значения `false` атрибуту `configurable` налагает на это ограничения.

При вызове метода `Object.defineProperty()` атрибуты `configurable`, `enumerable` и `writable` получают по умолчанию значение `false`, если не указано иное. В большинстве случаев мощные возможности метода `Object.defineProperty()` не требуются, но чтобы хорошо разбираться в JavaScript-объектах, важно знать, как он работает.

Свойства с функциями доступа

Свойства с функциями доступа не содержат данные. Вместо этого они содержат функции чтения и записи, хотя обе они необязательны. При чтении такого свойства вызывается функция чтения, которая отвечает за возвращение правильного значения. При записи свойства вызывается функция записи с новым значением, которая решает, что делать с данными. Свойства с функциями доступа имеют четыре атрибута:

- `[[Configurable]]` — указывает, можно ли удалить свойство с помощью оператора `delete`, изменить атрибуты свойства или преобразовать его в свойство с данными. По умолчанию этот атрибут имеет значение `true` у всех свойств, определенных непосредственно для объекта.
- `[[Enumerable]]` — указывает, будет ли свойство возвращаться в циклах `for-in`. По умолчанию этот атрибут имеет значение `true` у всех свойств, определенных непосредственно для объекта.
- `[[Get]]` — функция, вызываемая при чтении свойства. По умолчанию имеет значение `undefined`.
- `[[Set]]` — функция, вызываемая при записи свойства. По умолчанию имеет значение `undefined`.

Явно определить свойство с функциями доступа невозможно. Для этого нужно использовать метод `Object.defineProperty()`, например:

```
// Объявление объекта с псевдозакрытым членом "year_"
// и публичным членом "edition"
let book = {
    _year: 2017,
    edition: 1
};

Object.defineProperty(book, "year", {
    get() {
        return this._year;
    },
    set(newValue) {

        if (newValue > 2017) {
            this._year = newValue;
            this.edition += newValue - 2017;
        }
    }
});

book.year = 2018;
console.log(book.edition);    // 2
```

В этом фрагменте создается объект `book` с двумя свойствами, предлагаемыми по умолчанию: `_year` и `edition`. Знак подчеркивания в имени `_year` — это популярная нотация, указывающая, что значение не предполагается использовать вне методов объекта. Далее определяется свойство `year` с функциями доступа, у которого функция чтения просто возвращает значение `_year`, а функция записи определяет редакцию книги, выполняя некоторые вычисления. Присвоение значения 2018 свойству `year` изменяет `_year` на 2018, а `edition` на 2. Это типичный сценарий применения свойств с функциями доступа, когда задание свойства влечет за собой другие изменения.

Определять обе функции доступа не требуется. Если свойству назначена только функция чтения, это означает, что оно не поддерживает запись. В нестрогом режиме любые попытки записи такого свойства игнорируются, а в строгом приводят к ошибке. Аналогичным образом при чтении свойства, у которого есть только функция записи, в нестрогом режиме возвращается значение `undefined`, а в строгом возникает ошибка.

В браузерах, которые не поддерживают метод `Object.defineProperty()`, изменить атрибуты `[[Configurable]]` и `[[Enumerable]]` невозможно.

> **ПРИМЕЧАНИЕ** До ECMAScript 5 для создания свойств с функциями доступа использовались два нестандартных метода: `__defineGetter__()` и `__defineSetter__()`. Они были впервые разработаны Firefox, а затем скопированы Safari, Chrome и Opera.

Определение нескольких свойств

Чтобы для объекта можно было определить сразу несколько свойств, ECMAScript предоставляет метод `Object.defineProperties()`, принимающий два аргумента: объект, для которого нужно добавить или изменить свойства, и объект с новыми свойствами. Рассмотрим пример:

```
let book = {};

Object.defineProperties(book, {
    _year: {
        value: 2004
    },

    edition: {
        value: 1
    },

    year: {
        get() {
            return this._year;
        },

        set: function(newValue) {
            if (newValue > 2017) {
                this._year = newValue;
                this.edition += newValue - 2004;
            }
        }
    }
});
```

В этом коде для объекта `book` определяются свойства с данными `_year` и `edition` и свойство с функциями доступа `year`. Итоговый объект идентичен объекту из предыдущего раздела. Единственное отличие этого примера в том, что в нем создаются сразу все свойства.

Чтение атрибутов свойств

Получить дескриптор конкретного свойства можно с помощью метода `Object.getOwnPropertyDescriptor()` из ECMAScript 5. Он принимает два аргумента: объект со свойством, дескриптор которого нужно получить, и имя этого свойства. Метод возвращает объект со свойствами `configurable`, `enumerable`, `get` и `set` в случае свойств с функциями доступа или `configurable`, `enumerable`, `writable` и `value` в случае свойств с данными, например:

```
let book = {};

Object.defineProperties(book, {
    _year: {
        value: 2017
    },

    edition: {
```

```
        value: 1
    },

    year: {
        get: function() {
            return this._year;
        },

        set: function(newValue) {
            if (newValue > 2004) {
                this._year = newValue;
                this.edition += newValue - 2004;
            }
        }
    }
});
let descriptor = Object.getOwnPropertyDescriptor(book, "_year");
console.log(descriptor.value);            // 2017
console.log(descriptor.configurable);     // false
console.log(typeof descriptor.get);       // "undefined"

let descriptor = Object.getOwnPropertyDescriptor(book, "year");
console.log(descriptor.value);            // undefined
console.log(descriptor.enumerable);       // false
console.log(typeof descriptor.get);       // "function"
```

Для свойства с данными `_year` возвращается дескриптор, у которого свойство `value` имеет первоначальное значение, `configurable` — значение `false`, а `get` — значение `undefined`. У дескриптора свойства с функциями доступа `year` свойство `value` имеет значение `undefined`, `enumerable` — значение `false`, а свойство `get` является указателем на функцию чтения.

Новым в ECMAScript 2017 является статический метод `Object.getOwnPropertyDescriptors()`. Данный метод эффективно выполняет метод `Object.getOwnPropertyDescriptor()` для всех собственных свойств и возвращает их в новом объекте. Для предыдущего примера использование этого статического метода вернуло бы следующий объект:

```
let book = {};
Object.defineProperties(book, {
    year_: {
        value: 2017
    },

    edition: {
        value: 1
    },

    year: {
        get: function() {
            return this.year_;
    },

    set: function(newValue){
        if (newValue > 2017) {
```

```
            this.year_ = newValue;
            this.edition += newValue - 2017;
        }
    }
  }
});

console.log(Object.getOwnPropertyDescriptors(book));
// {
//         edition: {
//             configurable: false,
//             enumerable: false,
//             value: 1,
//             writable: false
//         },
//         year: {
//             configurable: false,
//             enumerable: false,
//             get: f(),
//             set: f(newValue),
//         },
//         year_: {
//             configurable: false,
//             enumerable: false,
//             value: 2019,
//             writable: false
//         }
// }
```

Слияние объектов

Для JavaScript-разработчиков часто может оказаться полезным выполнение слияния двух объектов. Более конкретно, это объединение будет принимать форму перенесения всех локальных свойств одного исходного объекта в целевой объект. Часто такое поведение также называется использованием смешивания, в котором целевой объект увеличивается путем смешивания в свойствах исходного объекта.

ECMAScript 6 представляет подобное поведение с помощью метода `Object.assign()`. Этот метод принимает один целевой объект и один или несколько исходных объектов, и для каждого исходного объекта копирует перечисляемый (`Object.propertyIsEnumerable` возвращает `true`) и собственный (`Object.hasOwnProperty` возвращает `true`) свойства в целевой объект. Свойства, помеченные строками и символами, будут скопированы. Для каждого подходящего свойства метод будет использовать `[[Get]]` для извлечения значения из исходного объекта и `[[Set]]` в целевом объекте для назначения значения.

```
let dest, src, result;

/**
 * Простое копирование
 */
dest = {};
src = { id: 'src' };
```

```
result = Object.assign(dest, src);

// Object.assign изменяет целевой объект
// и возвращает этот объект после завершения.
console.log(dest === result);    // true
console.log(dest !== src);       // true
console.log(result);             // { id: src }
console.log(dest);               // { id: src }

/**
 * Несколько исходных объектов
 */
dest = {};

result = Object.assign(dest, { a: 'foo' }, { b: 'bar' });

console.log(result); // { a: foo, b: bar }

/**
 * Методы чтения и записи свойств
 */
dest = {
    set a(val) {
        console.log('Invoked dest setter with param ${val}');
    }
};
src = {
    get a() {
        console.log('Invoked src getter');
        return 'foo';
    }
};

Object.assign(dest, src);
// Вызов метода чтения для src
// Вызов метода записи для dest с параметром foo
// Поскольку метод записи свойства не выполняет назначение,
// фактически значение не передается
console.log(dest); // { set a(val) {...} }
```

`Object.assign()` эффективно выполняет поверхностное копирование каждого исходного объекта. Если для нескольких исходных объектов определено одно и то же свойство, последним будет скопировано последнее значение. Кроме того, любое значение, полученное из свойств с функцией доступа, например метода чтения свойств в исходном объекте, будет назначено в качестве статического значения в объекте назначения — нет возможности передавать методы чтения и записи свойств между объектами.

```
let dest, src, result;

/**
 * Перезаписанные свойства
 */
dest = { id: 'dest' };

result = Object.assign(dest, { id: 'src1', a: 'foo' }, { id: 'src2', b: 'bar' });

// Object.assign перезапишет повторяющиеся свойства.
```

```
console.log(result); // { id: src2, a: foo, b: bar }

// Это можно проследить при использовании метода записи свойств
// на объекте назначения:
dest = {
    set id(x) {
        console.log(x);
    }
};

Object.assign(dest, { id: 'first' }, { id: 'second' }, { id: 'third' });
// first
// second
// third

/**
 * Ссылки на объект
 */
dest = {};
src = { a: {} };

Object.assign(dest, src);

// Поверхностное копирование означает, что копируются только ссылки на объекты.
console.log(dest); // { a :{} }
console.log(dest.a === src.a); // true
```

Если во время перезаписи возникнет ошибка, процесс будет прерван и завершится с выводом ошибки. `Object.assign()` не имеет понятия «отката» более ранних назначений свойств, поэтому этот метод — все, что можно сделать в такой ситуации, исправив ее последствия лишь частично.

```
let dest, src, result;

/**
* Обработка ошибок
 */
dest = {};
src = {
    a: 'foo',
    get b() {
        // Ошибка будет выведена через Object.assign()
        // Вызов данного метода чтения свойств.
        throw new Error();
    },
    c: 'bar'
};

try {
    Object.assign(dest, src);
} catch(e) {}

// Object.assign() не имеет способа отката уже проведенных изменений,
// поэтому результаты операции записи, проведенные на объекте назначения
// до появления ошибки, остаются неизменными:
console.log(dest); // { a: foo }
```

Идентичность и равенство объектов

В версиях, предшествующих ECMAScript 6, было несколько сложных случаев, когда оператора === было недостаточно:

```
// Здесь === ведет себя ожидаемо:
console.log(true === 1);    // false
console.log({} === {});     // false
console.log("2" === 2);     // false

// Эти случаи имеют разные представления в движке JS
// и все же рассматриваются как равные
console.log(+0 === -0);     // true
console.log(+0 === 0);      // true
console.log(-0 === 0);      // true

// Чтобы определить равенство NaN, требуется крайне раздражающий метод isNaN()
console.log(NaN === NaN);   // false
console.log(isNaN(NaN));    // true
```

Чтобы исправить это, спецификация ECMAScript 6 представила `Object.is()`, который ведет себя в основном так же, как `===`, но учитывает и крайние случаи, перечисленные ранее. Метод принимает ровно два аргумента:

```
console.log(Object.is(true, 1));    // false
console.log(Object.is({}, {}));     // false
console.log(Object.is("2", 2));     // false

// Корррректные равенство/неравенство 0, -0, +0:
console.log(Object.is(+0, -0));    // false
console.log(Object.is(+0, 0));     // true
console.log(Object.is(-0, 0));     // false

// Корректное равенство NaN:
console.log(Object.is(NaN, NaN)); // true
```

Чтобы проверить более двух объектов, обычно используют транзитивное равенство:

```
function recursivelyCheckEqual(x, ...rest) {
    return Object.is(x, rest[0]) &&
        (rest.length < 2 || recursivelyCheckEqual(...rest));
}
```

Расширенный синтаксис объектов

ECMAScript 6 представила несколько чрезвычайно полезных синтаксических инструментов для определения и взаимодействия с объектами. Ни один из них не меняет существенное поведение существующего движка, но значительно повышает удобство работы с объектами.

Все соглашения о синтаксисе объектов, представленные в этом разделе, также применимы к классам ECMAScript 6, определенным далее в этой главе.

ПРИМЕЧАНИЕ Расширенный синтаксис объектов, описанный в этом разделе, почти всегда превосходит заменяемый синтаксис; следовательно, вы обнаружите, что он используется по умолчанию в этой главе, а также в других разделах книги.

Сокращение значения свойства

Разработчики часто обнаруживают, что при добавлении переменной к объекту имя свойства, используемое для обозначения этой переменной, часто соответствует самому имени переменной:

```
let name = 'Matt';

let person = {
    name: name
};

console.log(person); // { name: 'Matt' }
```

Таким образом было введено условное обозначение значения свойства. Оно позволяет использовать саму переменную без двоеточия, а интерпретатор автоматически использует имя переменной в качестве ключа свойства. Если имя переменной не найдено, будет сгенерирована `ReferenceError`.

Следующий код эквивалентен написанному выше:

```
let name = 'Matt';

let person = {
    name
};

console.log(person); // { name: 'Matt' }
```

Минификаторы сохранят имена свойств между областями видимости, чтобы предотвратить разрыв ссылок. Пример — следующий фрагмент кода:

```
function makePerson(name) {
    return {
        name
    };
}

let person = makePerson('Matt');

console.log(person.name);    // Matt
```

Компиляторы стараются сохранить начальный идентификатор строки `name` внутри определения функции, даже если идентификатор параметра ограничен областью видимости функции. Когда это скомпилировано с использованием компилятора Google Closure, параметр функции будет сокращен, но свойство останется:

```
function makePerson(a) {
    return {
        name: a
    };
}

var person = makePerson("Matt");

console.log(person.name);    // Matt
```

Вычисляемые ключи свойств

До введения вычисляемых ключей свойств не было никакого способа динамически назначать ключи свойства в литерале объекта без объявления объекта, а затем индивидуально использовать квадратные скобки для назначения свойства. Например:

```
const nameKey = 'name';
const ageKey = 'age';
const jobKey = 'job';

let person = {};
person[nameKey] = 'Matt';
person[ageKey] = 27;
person[jobKey] = 'Software engineer';

console.log(person);    // { name: 'Matt', age: 27, job: 'Software engineer' }
```

С появлением вычисляемых свойств назначение свойства может происходить в начальном определении литерала объекта. Квадратные скобки вокруг ключа свойства объекта указывают среде выполнения оценивать его содержимое как выражение JavaScript вместо строки:

```
const nameKey = 'name';
const ageKey = 'age';
const jobKey = 'job';

let person = {
    [nameKey]: 'Matt',
    [ageKey]: 27,
    [jobKey]: 'Software engineer'
};

console.log(person); // { name: 'Matt', age: 27, job: 'Software engineer' }
```

Поскольку содержимое оценивается как выражение JavaScript, можно сделать содержимое вычисляемых свойств сложными выражениями, которые будут оцениваться при создании экземпляра:

```
const nameKey = 'name';
const ageKey = 'age';
const jobKey = 'job';
let uniqueToken = 0;

function getUniqueKey(key) {
```

```
    return '${key}_${uniqueToken++}';
}

let person = {
    [getUniqueKey(nameKey)]: 'Matt',
    [getUniqueKey(ageKey)]: 27,
    [getUniqueKey(jobKey)]: 'Software engineer'
};

console.log(person); // { name_0: 'Matt', age_1: 27, job_2: 'Software engineer' }
```

ПРИМЕЧАНИЕ Любые ошибки, возникшие в выражении ключа вычисляемого свойства, прервут создание объекта. Будьте осторожны, когда выражения, вычисляющие ключ свойства, могут иметь побочные эффекты, так как ошибка, выдаваемая в выражении, не откатит более ранние вычисления.

Краткий синтаксис методов

При определении функциональных свойств объекта их формат почти всегда принимает форму ключа свойства, ссылающегося на анонимное функциональное выражение, как показано здесь:

```
let person = {
    sayName: function(name) {
        console.log('My name is ${name}');
    }
};

person.sayName('Matt'); // My name is Matt
```

Новый сокращенный синтаксис методов следует этому шаблону и позволяет разработчику отказаться от возможности называть функциональное выражение, что в большинстве случаев бесполезно, и взамен резко сократить способ объявления функционального свойства.

Следующий фрагмент кода по поведению идентичен предыдущему:

```
let person = {
    sayName(name) {
        console.log('My name is ${name}');
    }
};

person.sayName('Matt'); // My name is Matt
```

Это также применимо к методам чтения и записи свойств объекта:

```
let person = {
    name_: '',
    get name() {
        return this.name_;
```

```
    },
    set name(name) {
        this.name_ = name;
    },
    sayName() {
        console.log('My name is ${this.name_}');
    }
};

person.name = 'Matt';
person.sayName(); // My name is Matt
```

Сокращенный синтаксис методов и вычисляемые ключи свойств взаимно совместимы:

```
const methodKey = 'sayName';

let person = {
    [methodKey](name) {
        console.log('My name is ${name}');
    }
}

person.sayName('Matt'); // My name is Matt
```

> **ПРИМЕЧАНИЕ** Вы обнаружите, что синтаксис сокращенного метода более полезен в контексте классов ECMAScript 6, о которых пойдет речь ниже в этой главе.

Деструктурирование объектов

В ECMAScript 6 введено деструктурирование объектов, позволяющее выполнять одну или несколько операций с использованием вложенных данных в одном выражении. В отношении объектов это дает возможность выполнять присваивания из свойств объекта, используя синтаксис, соответствующий структуре объекта.

Ниже приведен пример двух эквивалентных фрагментов кода, первый — без деструктурирования объекта:

```
// Без деструктурирования объекта
let person = {
    name: 'Matt',
    age: 27
};

let personName = person.name,
    personAge = person.age;

console.log(personName);    // Matt
console.log(personAge);     // 27
```

Второй — с деструктурированием:

```
// С деструктурированием
let person = {
    name: 'Matt',
    age: 27
};

let { name: personName, age: personAge } = person;

console.log(personName);    // Matt
console.log(personAge);     // 27
```

Деструктурирование позволяет объявлять несколько переменных и одновременно выполнять несколько присваиваний внутри одного литерально-подобного синтаксиса объекта. Если нужно повторно использовать имя свойства в качестве имени локальной переменной, вы можете использовать сокращенный синтаксис, как показано ниже:

```
let person = {
    name: 'Matt',
    age: 27
};

let { name, age } = person;

console.log(name);    // Matt
console.log(age);     // 27
```

Деструктурированные назначения не должны соответствовать тому, что находится внутри объекта. Можно игнорировать свойства при выполнении присваивания; и наоборот, если вы ссылаетесь на несуществующее свойство, ему будет назначено значение `undefined`:

```
let person = {
    name: 'Matt',
    age: 27
};

let { name, job } = person;

console.log(name);    // Matt
console.log(job);     // undefined
```

Также можно определить значения по умолчанию, которые будут применяться в случае, если свойство не существует в исходном объекте:

```
let person = {
    name: 'Matt',
    age: 27
};

let { name, job='Software engineer' } = person;

console.log(name);    // Matt
console.log(job);     // Software engineer
```

Деструктурирование использует внутреннюю функцию `ToObject()` (которая не доступна напрямую во время выполнения) для приведения исходного параметра в объект. Это означает, что примитивные значения будут обрабатываться как объекты при использовании в операции деструктурирования; это также означает, что по определению в спецификации `ToObject()`, `null` и `undefined` не могут быть деструктурированы и это приведет к ошибке.

```
let { length } = 'foobar';
console.log(length); // 6

let { constructor: c } = 4;
console.log(c === Number); // true

let { _ } = null; // TypeError

let { _ } = undefined; // TypeError
```

При деструктурировании не требуется, чтобы объявления переменных происходили внутри выражения деструктурирования. Однако для этого необходимо, чтобы выражение присваивания содержалось в скобках:

```
let personName, personAge;

let person = {
    name: 'Matt',
    age: 27
};

({name: personName, age: personAge} = person);

console.log(personName, personAge);    // Matt, 27
```

Вложенное деструктурирование

Не существует ограничений по ссылкам на вложенные свойства или цели назначения. Это позволяет выполнять такие операции, как копирование свойств объекта:

```
let person = {
    name: 'Matt',
    age: 27,
    job: {
        title: 'Software engineer'
    }
};
let personCopy = {};

({
    name: personCopy.name,
    age: personCopy.age,
    job: personCopy.job
} = person);

// Поскольку ссылка на объект была назначена в personCopy, изменение свойства
// внутри объекта person.job будет распространяться на personCopy:
```

```
person.job.title = 'Hacker'

console.log(person);
// { name: 'Matt', age: 27, job: { title: 'Hacker' } }

console.log(personCopy);
// { name: 'Matt', age: 27, job: { title: 'Hacker' } }
```

Деструктурирующие назначения могут быть вложенными для соответствия вложенным ссылкам на свойства:

```
let person = {
    name: 'Matt',
    age: 27,
    job: {
        title: 'Software engineer'
    }
};

// Объявление переменной title и назначение person.job.title в качестве значения
let { job: { title }} = person;

console.log(title); // Software engineer
```

Нельзя использовать вложенные ссылки на свойства, если внешнее свойство не определено. Это верно как для исходных, так и для целевых объектов:

```
let person = {
    job: {
        title: 'Software engineer'
    }
};
let personCopy = {};

// 'foo' имеет значение undefined в исходном объекте
({
    foo: {
        bar: personCopy.bar
    }
} = person);
// TypeError: Cannot destructure property 'bar' of 'undefined' or 'null'.

// 'job' имеет значение undefined в целевом объекте
({
    job: {
        title: personCopy.job.title
    }
} = person);
// TypeError: Cannot set property 'title' of undefined
```

Завершение частичного деструктурирования

Важно отметить, что деструктурированное присваивание, включающее несколько свойств, является последовательной операцией с независимыми результатами. Если в одном деструктурированном выражении с несколькими присваиваниями начальные присваивания выполняются успешно, но более позднее выдает ошибку,

деструктурированное присваивание завершается только после частичного завершения:

```
let person = {
    name: 'Matt',
    age: 27
};

let personName, personBar, personAge;

try {
    // person.foo имеет значение undefined, поэтому код ниже вернет ошибку
    ({name: personName, foo: { bar: personBar }, age: personAge} = person);
} catch(e) {}

console.log(personName, personBar, personAge);
// Matt, undefined, undefined
```

Соответствие контекста параметра

Также можно выполнить деструктурированное присваивание внутри списка параметров функции. Оно не влияет на объект аргументов, но позволяет объявлять переменные внутри сигнатуры функции, которые сразу же доступны внутри тела функции:

```
let person = {
    name: 'Matt',
    age: 27
};

function printPerson(foo, {name, age}, bar) {
    console.log(arguments);
    console.log(name, age);
}

function printPerson2(foo, {name: personName, age: personAge}, bar) {
    console.log(arguments);
    console.log(personName, personAge);
}

printPerson('1st', person, '2nd');
// ['1st', { name: 'Matt', age: 27 }, '2nd']
// 'Matt', 27

printPerson2('1st', person, '2nd');
// ['1st', { name: 'Matt', age: 27 }, '2nd']
// 'Matt', 27
```

СОЗДАНИЕ ОБЪЕКТОВ

Создать один объект можно с помощью конструктора `Object` или литерала объекта, но для создания многих объектов с одинаковым интерфейсом эти способы не подходят из-за повторения кода.

Обзор

Благодаря последовательному выпуску спецификаций доступные функции ECMAScript следовали весьма необычному шаблону. В спецификации ECMAScript 5.1 не было формальной поддержки объектно-ориентированных конструкций, таких как классы или наследование. Однако, как вы увидите в следующих разделах, умное применение прототипного наследования позволило разработчикам JavaScript эмулировать это поведение — и довольно успешно.

Со спецификацией ECMAScript 6 была введена формальная поддержка классов и наследования. Данные классы ES6 предназначены для полного включения решений для классов на основе прототипов, разработанных в предыдущих спецификациях. Однако их реализация во многих отношениях является просто синтаксической абстракцией для функций конструктора в стиле ES5.1 и наследования прототипов.

> **ПРИМЕЧАНИЕ** Не заблуждайтесь: кодовая база JavaScript, построенная на объектно-ориентированных шаблонах, почти всегда должна использовать классы ECMAScript 6. Тем не менее полезно узнать о соглашениях, которые существовали до классов ES6, тем более что определение класса ES6 можно представить в виде тонкой оболочки вокруг существующих конструкций. Поэтому перед разделом классов ES6 в следующих разделах будут постепенно вводиться основные понятия, которые заменяются классами.

Паттерн Фабрика

Паттерн Фабрика (factory pattern) — это широко известный паттерн проектирования, который используется при разработке ПО для абстрагирования процесса создания специфических объектов (другие паттерны проектирования и их JavaScript-реализации мы обсудим чуть позже). Единственный способ создания объектов со специфическими интерфейсами представлен ниже:

```
function createPerson(name, age, job) {
    let o = new Object();
    o.name = name;
    o.age = age;
    o.job = job;
    o.sayName = function() {
        console.log(this.name);
    };
    return o;
}

let person1 = createPerson("Nicholas", 29, "Software Engineer");
let person2 = createPerson("Greg", 27, "Doctor");
```

Здесь функция `createPerson()` принимает в качестве аргументов все сведения, необходимые для создания объекта `Person`. Функцию можно вызывать любое количество

раз с разными аргументами, и каждый раз она будет возвращать объект с тремя свойствами и одним методом. Это решает проблему создания многих похожих объектов, однако паттерн Фабрика не позволяет узнать тип объекта.

Паттерн Конструктор функции

Как уже отмечалось, конструкторы в ECMAScript используются для создания объектов специфических типов. Конструкторы встроенных типов, таких как `Object` и `Array`, автоматически становятся доступны в среде во время выполнения. Также можно определять конструкторы в форме функций со свойствами и методами для собственных типов. Так, предыдущий пример с использованием *паттерна Конструктор функции* (function constructor pattern) можно переписать следующим образом:

```
function Person(name, age, job) {
    this.name = name;
    this.age = age;
    this.job = job;
    this.sayName = function() {
        console.log(this.name);
    };
}

let person1 = new Person("Nicholas", 29, "Software Engineer");
let person2 = new Person("Greg", 27, "Doctor");

person1.sayName();    // Nicholas
person2.sayName();    // Greg
```

В этом примере функция `Person()` заменяет фабричную функцию `createPerson()`. Новая функция отличается от прежней следующими аспектами:

- объект явно не создается;
- свойства и метод назначаются непосредственно объекту `this`;
- инструкция `return` отсутствует.

Обратите также внимание на то, что имя функции `Person` начинается с прописной буквы. Имена конструкторов всегда начинаются с прописной буквы, а имена обычных функций — со строчной. Это соглашение позаимствовано в других объектно-ориентированных языках, чтобы было проще различать ECMAScript-функции по способам их применения, так как конструкторы — это просто функции, которые создают объекты.

Для создания экземпляров `Person` используется оператор `new`. В результате выполняются следующие действия:

1. Создание нового объекта в памяти.
2. Назначение внутреннего указателя `[[Prototype]]` нового объекта свойству `prototype` конструктора.

3. Назначение нового объекта переменной `this` конструктора (после чего `this` указывает на новый объект).
4. Выполнение кода внутри конструктора (добавление свойств к новому объекту).
5. Возвращение данного объекта, если функция конструктора возвращает ненулевое значение. В противном случае возвращается только что созданный объект.

В конце предыдущего примера переменные `person1` и `person2` создаются как разные экземпляры `Person`. У каждого из этих объектов есть свойство `constructor`, указывающее на `Person`:

```
console.log(person1.constructor == Person);    // true
console.log(person2.constructor == Person);    // true
```

Изначально свойство `constructor` предназначалось для идентификации типа объектов, однако считается, что для этого безопаснее использовать оператор `instanceof`. Как показывает следующий код, оба объекта в примере являются экземплярами типов `Object` и `Person`:

```
console.log(person1 instanceof Object);    // true
console.log(person1 instanceof Person);    // true
console.log(person2 instanceof Object);    // true
console.log(person2 instanceof Person);    // true
```

Определение собственных конструкторов позволяет позднее узнавать типы объектов, созданных с их помощью, что является серьезным преимуществом по сравнению с паттерном Фабрика. В последнем примере объекты `person1` и `person2` считаются экземплярами `Object` потому, что все пользовательские объекты наследуют от типа `Object` (детали см. далее).

Функции конструктора не обязательно должны быть выражены как объявление функции. Выражение функции, назначенное переменной, ведет себя одинаково:

```
let Person = function(name, age, job) {
    this.name = name;
    this.age = age;
    this.job = job;
    this.sayName = function() {
        console.log(this.name);
    };
}

let person1 = new Person("Nicholas", 29, "Software Engineer");
let person2 = new Person("Greg", 27, "Doctor");

person1.sayName(); // Nicholas
person2.sayName(); // Greg

console.log(person1 instanceof Object); // true
console.log(person1 instanceof Person); // true
console.log(person2 instanceof Object); // true
console.log(person2 instanceof Person); // true
```

При создании экземпляра круглые скобки после функции конструктора являются необязательными, если вам не нужно передавать какие-либо аргументы — оператор `new` вызовет функцию конструктора несмотря ни на что:

```
function Person() {
    this.name = "Jake";
    this.sayName = function() {
        console.log(this.name);
    };
}

let person1 = new Person();
let person2 = new Person;

person1.sayName(); // Jake
person2.sayName(); // Jake

console.log(person1 instanceof Object); // true
console.log(person1 instanceof Person); // true
console.log(person2 instanceof Object); // true
console.log(person2 instanceof Person); // true
```

Конструкторы как функции

Единственным отличием конструкторов от других функций является способ их вызова, потому что определяются все функции одинаково. Любая функция, вызванная с помощью оператора `new`, работает как конструктор, а функция, вызванная без `new`, ведет себя обычным образом. Например, функцию `Person()` из предыдущего примера можно вызвать следующими способами:

```
// вызов в качестве конструктора
let person = new Person("Nicholas", 29, "Software Engineer");
person.sayName();    // "Nicholas"

// вызов в качестве обычной функции
Person("Greg", 27, "Doctor");        // функция добавляется к window
window.sayName();    // "Greg"

// вызов в области видимости другого объекта
let o = new Object();
Person.call(o, "Kristen", 25, "Nurse");
o.sayName();        // "Kristen"
```

В первой части примера показано типичное применение конструктора, а именно — создание объекта с помощью оператора `new`. Во второй части функция `Person()` вызывается без оператора `new`, в результате свойства и методы добавляются к объекту `window`. Запомните, что если функция вызывается без явно заданного значения `this` (то есть не как метод объекта и без метода `call()` или `apply()`), значение `this` указывает на объект `Global` (`window` в веб-браузерах). Таким образом, вызов метода `sayName()` для объекта `window` возвращает значение `"Greg"`. Функцию `Person()` также можно вызвать в области видимости конкретного объекта, используя метод `call()`

или `apply()`. В третьей части в качестве `this` указан объект `o`, которому и назначаются все свойства и метод `sayName()`.

Проблемы конструкторов

Хотя концепция конструктора полезна, она не лишена недостатков. Главный из них заключается в том, что методы создаются для каждого экземпляра. Так, в предыдущем примере и у `person1`, и у `person2` есть метод с именем `sayName()`, но эти методы — не один и тот же экземпляр `Function`. Функции в ECMAScript являются объектами, так что при каждом определении функции создается объект. Логически конструктор на самом деле выглядит так:

```
function Person(name, age, job) {
    this.name = name;
    this.age = age;
    this.job = job;
    this.sayName = new Function("console.log(this.name)"); // логический эквивалент
}
```

При таком взгляде на конструктор очевидно, что каждый экземпляр `Person` получает свой экземпляр функции, выводящей на экран свойство `name`. Строго говоря, такое создание функции уникально в плане цепочек областей видимости и разрешения идентификаторов, но технически новый экземпляр `Function` создается обычным образом. В общем, одноименные функции в разных экземплярах не эквивалентны, что подтверждает следующий код:

```
console.log(person1.sayName == person2.sayName);        // false
```

Не имеет смысла создавать два экземпляра `Function`, делающих одно и то же, особенно если учесть, что благодаря объекту `this` можно отложить привязку функций к конкретным объектам до выполнения кода. Чтобы обойти это ограничение, можно вынести определение функции за пределы конструктора:

```
function Person(name, age, job) {
    this.name = name;
    this.age = age;
    this.job = job;
    this.sayName = sayName;
}

function sayName() {
    console.log(this.name);
}

let person1 = new Person("Nicholas", 29, "Software Engineer");
let person2 = new Person("Greg", 27, "Doctor");

person1.sayName(); // Nicholas
person2.sayName(); // Greg
```

В этом примере функция `sayName()` определена вне конструктора в глобальной области видимости и назначается внутри конструктора свойству `sayName`. Поскольку

свойство `sayName` теперь содержит лишь указатель на функцию, она становится общей для объектов `person1` и `person2`. Это решает проблему повторяющихся функций, но при этом засоряет глобальную область видимости функцией, которая на самом деле используется только в связи с объектами. Если у объекта будет много методов, нам придется создать много глобальных функций и определение нашего ссылочного типа перестанет быть аккуратной группой связанных инструкций. Эту проблему устраняет *паттерн Прототип* (prototype pattern).

Паттерн Прототип

Каждая функция создается со свойством `prototype` — объектом, содержащим свойства и методы, которые должны быть доступны в экземплярах конкретного ссылочного типа. Этот объект в буквальном смысле является прототипом для объекта, создаваемого при вызове конструктора. Преимущество использования прототипа в том, что все его свойства и методы общие для объектов. Вместо того чтобы назначать атрибуты объекту в конструкторе, их можно назначить непосредственно прототипу, например:

```
function Person() {
}

Person.prototype.name = "Nicholas";
Person.prototype.age = 29;
Person.prototype.job = "Software Engineer";
Person.prototype.sayName = function() {
    console.log(this.name);
};

let person1 = new Person();
person1.sayName();        // "Nicholas"

let person2 = new Person();
person2.sayName();        // "Nicholas"

console.log(person1.sayName == person2.sayName);        // true
```

Также подойдет использование функционального выражения:

```
let Person = function() {};

Person.prototype.name = "Nicholas";
Person.prototype.age = 29;
Person.prototype.job = "Software Engineer";
Person.prototype.sayName = function() {
    console.log(this.name);
};

let person1 = new Person();
person1.sayName(); // "Nicholas"

let person2 = new Person();
person2.sayName(); // "Nicholas"

console.log(person1.sayName == person2.sayName); // true
```

Здесь свойства и метод `sayName()` добавляются непосредственно к свойству `prototype` объекта `Person`, а конструктор остается пустым. Тем не менее конструктор можно вызывать для создания объектов, и у них будут доступны все свойства и методы. В отличие от паттерна Конструктор, свойства и методы Прототипа являются общими для экземпляров, так что объекты `person1` и `person2` имеют один набор свойств и одну функцию `sayName()` на двоих. Чтобы понять, как все это работает, нужно подробнее обсудить ECMAScript-прототипы.

Подробности работы прототипов

Когда создается функция, по определенным правилам создается также ее свойство `prototype`. По умолчанию во все прототипы автоматически добавляется свойство `constructor`, указывающее на функцию, к которой оно относится (так, в предыдущем примере свойство `Person.prototype.constructor` указывает на функцию `Person`). Затем в зависимости от конструктора в прототип могут быть добавлены другие свойства и методы.

При определении пользовательского конструктора в прототип добавляется по умолчанию только свойство `constructor`, а все остальные методы наследуются от типа `Object`. Когда с помощью конструктора создается новый экземпляр типа, в экземпляре определяется внутренний указатель на прототип конструктора. В ECMA-262 этот указатель называется `[[Prototype]]`. Стандартного способа доступа к нему из сценариев нет, но в Firefox, Safari и Chrome у каждого объекта есть свойство `__proto__`, которое в других браузерах полностью скрыто от JS-сценариев. Важно отметить наличие непосредственной связи между экземпляром и прототипом конструктора, но не между экземпляром и конструктором.

Это отношение может быть трудно визуализировать, поэтому обратите внимание на следующий фрагмент — своего рода таблицу для отражения общего поведения прототипа:

```
/**
 * Функции конструктора могут существовать как функциональные выражения
 * или объявления функции, так что оба примера подходят:
 *     function Person {}
 *     let Person = function() {}
 */
function Person() {}

/**
 * Учитывая объявление, функция конструктора
 * уже имеет связанный объект прототипа:
 */
console.log(typeof Person.prototype);
console.log(Person.prototype);
// {
//     constructor: f Person(),
//     __proto__: Object
// }
```

```
/**
 * Как было сказано ранее, у функции конструктора есть
 * ссылка 'prototype' на объект прототипа, и
 * у объекта прототипа есть ссылка 'constructor' на
 * функцию конструктора. Эти ссылки цикличны:
 */
console.log(Person.prototype.constructor === Person); // true

/**
 * Любая стандартная цепочка прототипов завершится на прототипе Object.
 * Прототип прототипа Object — null.
 */
console.log(Person.prototype.__proto__ === Object.prototype);      // true
console.log(Person.prototype.__proto__.constructor === Object);    // true
console.log(Person.prototype.__proto__.__proto__ === null);        // true

console.log(Person.prototype.__proto__);
// {
//     constructor: f Object(),
//     toString: ...
//     hasOwnProperty: ...
//     isPrototypeOf: ...
//     ...
// }

let person1 = new Person(),
    person2 = new Person();

/**
 * Конструктор, объект прототипа и экземпляр —
 * три совершенно разных объекта:
 */
console.log(person1 !== Person);              // true
console.log(person1 !== Person.prototype);    // true
console.log(Person.prototype !== person);     // true

/**
 * Экземпляр связан с прототипом через __proto__, который
 * является буквальным проявлением скрытого свойства [[Prototype]].
 *
 * Конструктор связан с прототипом через свойство constructor.
 *
 * Экземпляр не имеет прямой ссылки на конструктор, только через прототип.
 */
console.log(person1.__proto__ === Person.prototype);      // true
conosle.log(person1.__proto__.constructor === Person);    // true

/**
 * Два экземпляра одной функции конструктора имеют один и тот же
 * объект прототипа:
 */
console.log(person1.__proto__ === person2.__proto__);    // true

/**
 * instanceof проверит цепочку прототипов экземпляра на наличие
 * свойства prototype функции конструктора:
 */
```

```
console.log(person1 instanceof Person);             // true
console.log(person1 instanceof Object);             // true
console.log(Person.prototype instanceof Object);    // true
```

Связи между объектами в примере с конструктором `Person` и свойством `Person.prototype` показаны на рис. 8.1.

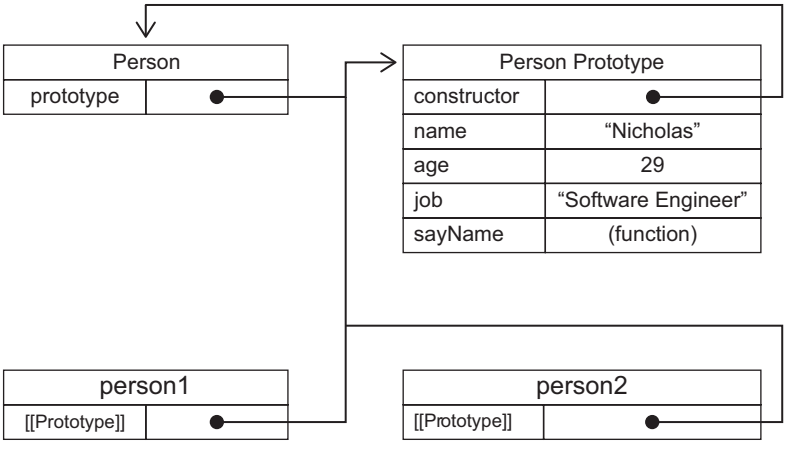

Рис. 8.1

На рисунке изображены конструктор `Person`, прототип `Person` и два экземпляра типа `Person`. Обратите внимание на то, что свойство `Person.prototype` указывает на объект прототипа, а `Person.prototype.constructor` — на функцию `Person`. Прототип содержит свойство `constructor` и другие свойства, которые были добавлены. У каждого экземпляра `Person` (`person1` и `person2`) есть внутренние свойства, указывающие только на свойство `Person.prototype`, но непосредственной связи с конструктором у экземпляров нет. Отметим также, что вызов `person1.sayName()` выполняется нормально, хотя у самих экземпляров нет ни свойств, ни методов. Это возможно благодаря процедуре поиска свойств объекта.

Несмотря на то что указатель `[[Prototype]]` доступен не во всех браузерах, можно проверить наличие связи между объектом и прототипом с помощью метода `isPrototypeOf()`. Он возвращает `true`, если указатель `[[Prototype]]` экземпляра указывает на прототип, для которого вызван метод:

```
console.log(Person.prototype.isPrototypeOf(person1));    // true
console.log(Person.prototype.isPrototypeOf(person2));    // true
```

В этом примере метод `isPrototypeOf()` прототипа вызывается для экземпляров `person1` и `person2`. Поскольку у обоих есть ссылка на `Person.prototype`, метод возвращает `true`.

У типа `Object` в ECMAScript добавлен новый метод `Object.getPrototypeOf()`, который возвращает значение `[[Prototype]]`, например:

```
console.log(Object.getPrototypeOf(person1) == Person.prototype);  // true
console.log(Object.getPrototypeOf(person1).name);                 // "Nicholas"
```

Первая строка этого фрагмента просто подтверждает, что объект, возвращенный методом `Object.getPrototypeOf()`, на самом деле является прототипом объекта. Вторая инструкция читает свойство `name` прототипа, возвращая строку `"Nicholas"`. С помощью метода `Object.getPrototypeOf()` можно легко получить прототип объекта, что важно для реализации наследования на основе прототипов (см. следующую главу).

Тип `Object` также имеет метод `setPrototypeOf()`, который записывает новое значение в свойство `[[Prototype]]` экземпляра. Это позволяет перезаписать иерархию прототипов уже созданного объекта:

```
let biped = {
    numLegs: 2
};
let person = {
    name: 'Matt'
};

Object.setPrototypeOf(person, biped);

console.log(person.name);                                  // Matt
console.log(person.numLegs);                               // 2
console.log(Object.getPrototypeOf(person) === biped);      // true
```

> **ПРИМЕЧАНИЕ** Операция `Object.setPrototypeOf()`, вероятно, приведет к серьезному снижению производительности при ее использовании. Это лучше всего изложено в документации Mozilla: «В каждом браузере и движке JavaScript влияние на производительность изменения наследования является тонким и обширным и не ограничивается только временем, потраченным на оператор `Object.setPrototypeOf()`, но может распространяться на любой код, имеющий доступ к любому объекту, чье свойство `[[Prototype]]` было изменено».

Во избежание этих замедлений проще создать новый объект и указать его прототип с помощью `Object.create()`:

```
let biped = {
    numLegs: 2
};
let person = Object.create(biped);
person.name = 'Matt';

console.log(person.name);                                  // Matt
console.log(person.numLegs);                               // 2
console.log(Object.getPrototypeOf(person) === biped);      // true
```

Понимание иерархии прототипов

При чтении свойства объекта начинается его поиск. Сначала свойство с указанным именем ищется в самом экземпляре объекта. Если оно обнаруживается у экземпляра,

возвращается значение свойства; если его у экземпляра нет, поиск продолжается в прототипе. В случае обнаружения свойства у прототипа возвращается его значение. Так, при вызове `person1.sayName()` интерпретатор JavaScript сначала выясняет, есть ли свойство с именем `sayName` у экземпляра `person1`. Его нет, поэтому интерпретатор ищет свойство `sayName` в прототипе `person1`. В этот раз свойство доступно, поэтому интерпретатор вызывает функцию `sayName()`, связанную с прототипом. При вызове `person2.sayName()` выполняется аналогичная процедура, которая завершается таким же результатом. Так прототипы обеспечивают совместное использование свойств и методов несколькими экземплярами объектов.

> **ПРИМЕЧАНИЕ** Упомянутое свойство `constructor` есть только у прототипа и доступно из экземпляров объектов.

Хотя в экземплярах объектов можно читать значения из прототипа, перезаписать их нельзя. Если добавить к экземпляру свойство с именем, как у свойства прототипа, последнее будет скрыто, например:

```
function Person() {
}

Person.prototype.name = "Nicholas";
Person.prototype.age = 29;
Person.prototype.job = "Software Engineer";
Person.prototype.sayName = function() {
    console.log(this.name);
};

let person1 = new Person();
let person2 = new Person();

person1.name = "Greg";
console.log(person1.name);    // "Greg" – из экземпляра
console.log(person2.name);    // "Nicholas" – из прототипа
```

В этом примере свойство `name` экземпляра `person1` затеняется новым значением. Оба свойства — и `person1.name`, и `person2.name` — работают нормально, возвращая `"Greg"` (из экземпляра объекта) и `"Nicholas"` (из прототипа) соответственно. При чтении значения `person1.name` в методе `console.log()` начинается поиск свойства с именем `name` у экземпляра. Поскольку такое свойство есть у экземпляра, оно используется, а поиск в прототипе не выполняется. У экземпляра `person2` свойства `name` нет, поэтому поиск продолжается в прототипе и используется свойство `name` прототипа.

Как только свойство добавлено к экземпляру объекта, оно *затеняет* (shadows) все одноименные свойства прототипа, то есть блокирует их, не изменяя прототип. Даже если присвоить свойству экземпляра значение `null`, ссылка на прототип не восстанавливается. Оператор `delete` полностью удаляет свойство экземпляра, делая свойство прототипа доступным:

```
function Person() {
}

Person.prototype.name = "Nicholas";
Person.prototype.age = 29;
Person.prototype.job = "Software Engineer";
Person.prototype.sayName = function() {
    console.log(this.name);
};

let person1 = new Person();
let person2 = new Person();

person1.name = "Greg";
console.log(person1.name);    // "Greg" — из экземпляра
console.log(person2.name);    // "Nicholas" — из прототипа

delete person1.name;
console.log(person1.name);    // "Nicholas" — из прототипа
```

Здесь оператор `delete` вызывается для свойства `person1.name`, которое ранее было затенено значением `"Greg"`. Это восстанавливает ссылку на свойство `name` прототипа, так что при следующем доступе к `person1.name` возвращается значение свойства прототипа.

Метод `hasOwnProperty()`, унаследованный от типа `Object`, позволяет выяснить, принадлежит свойство экземпляру или прототипу. Он возвращает `true`, только если свойство с указанным именем есть у экземпляра, например:

```
function Person() {
}

Person.prototype.name = "Nicholas";
Person.prototype.age = 29;
Person.prototype.job = "Software Engineer";
Person.prototype.sayName = function() {
    console.log(this.name);
};

let person1 = new Person();
let person2 = new Person();

console.log(person1.hasOwnProperty("name"));        // false

person1.name = "Greg";
console.log(person1.name);    // "Greg" — из экземпляра
console.log(person1.hasOwnProperty("name"));        // true

console.log(person2.name);    // "Nicholas" — из прототипа
console.log(person2.hasOwnProperty("name"));        // false

delete person1.name;
console.log(person1.name);    // "Nicholas" — из прототипа
console.log(person1.hasOwnProperty("name"));        // false
```

Метод `hasOwnProperty()` в этом примере ясно показывает, когда используется свойство экземпляра, а когда свойство прототипа. Вызов `person1.hasOwnProperty("name")`

возвращает `true` только после перезаписи свойства `name` для экземпляра `person1` (это указывает, что используется свойство экземпляра, а не прототипа). Рисунок 8.2 поясняет выполненные в примере действия (ради простоты связь с конструктором `Person` опущена).

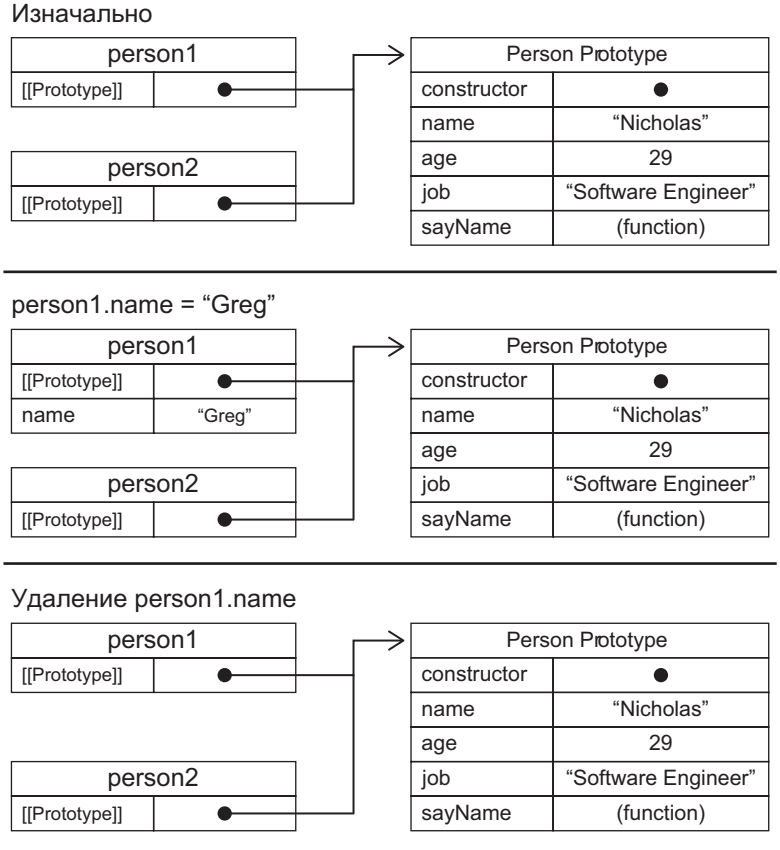

Рис. 8.2

ПРИМЕЧАНИЕ Метод `Object.getOwnPropertyDescriptor()` из ECMAScript работает только со свойствами экземпляра; чтобы получить дескриптор свойства прототипа, необходимо вызвать метод `Object.getOwnPropertyDescriptor()` непосредственно для объекта прототипа.

Прототипы и оператор in

Оператор `in` можно использовать отдельно или в цикле `for-in`. В первом случае он возвращает `true`, если свойство с указанным именем имеется у экземпляра или у прототипа объекта. Рассмотрим пример:

```
function Person() {
}

Person.prototype.name = "Nicholas";
Person.prototype.age = 29;
Person.prototype.job = "Software Engineer";
Person.prototype.sayName = function() {
    console.log(this.name);
};

let person1 = new Person();
let person2 = new Person();

console.log(person1.hasOwnProperty("name"));        // false
console.log("name" in person1);  // true

person1.name = "Greg";
console.log(person1.name);       // "Greg" — из экземпляра
console.log(person1.hasOwnProperty("name"));        // true
console.log("name" in person1);  // true

console.log(person2.name);       // "Nicholas" — из прототипа
console.log(person2.hasOwnProperty("name"));        // false
console.log("name" in person2);   // true

delete person1.name;
console.log(person1.name);        // "Nicholas" — из прототипа
console.log(person1.hasOwnProperty("name"));        // false
console.log("name" in person1);   // true
```

Во всем этом фрагменте свойство `name` доступно у каждого объекта непосредственно или в прототипе. Таким образом, выражение `"name" in person1` всегда возвращает `true` независимо от того, принадлежит или нет свойство именно экземпляру. Объединив вызов метода `hasOwnProperty()` с оператором `in`, можно выяснить, принадлежит ли свойство прототипу:

```
function hasPrototypeProperty(object, name) {
    return !object.hasOwnProperty(name) && (name in object);
}
```

Поскольку оператор `in` возвращает `true`, если свойство так или иначе доступно через объект, а метод `hasOwnProperty()` возвращает `true`, только если свойство имеется у экземпляра, свойство прототипа можно узнать по тому, что оператор `in` возвращает `true`, а метод `hasOwnProperty()` — `false`, например:

```
function Person() {
}

Person.prototype.name = "Nicholas";
Person.prototype.age = 29;
Person.prototype.job = "Software Engineer";
Person.prototype.sayName = function() {
    console.log(this.name);
};
```

```
let person = new Person();
console.log(hasPrototypeProperty(person, "name"));    // true

person.name = "Greg";
console.log(hasPrototypeProperty(person, "name"));    // false
```

В этом коде свойство `name` сначала определено у прототипа, поэтому метод `hasPrototypeProperty()` возвращает `true`. После перезаписи свойства оно относится к экземпляру, и метод `hasPrototypeProperty()` возвращает `false`. Да, свойство `name` есть и у прототипа, но оно больше не используется, потому что затенено свойством экземпляра.

В цикле `for-in` возвращаются все свойства экземпляра и прототипа, которые доступны через объект и могут быть перечислены. Свойства экземпляра, затеняющие неперечислимые свойства прототипа (неперечислимые свойства — это свойства, у которых атрибут `[[Enumerable]]` имеет значение `false`), также возвращаются в цикле `for-in`, потому что все свойства, определенные разработчиком, перечислимы во всех браузерах.

Получить список всех перечислимых свойств экземпляра можно с помощью метода `Object.keys()`, который принимает объект и возвращает массив с именами соответствующих свойств, например:

```
function Person() {
}

Person.prototype.name = "Nicholas";
Person.prototype.age = 29;
Person.prototype.job = "Software Engineer";
Person.prototype.sayName = function() {
    console.log(this.name);
};

let keys = Object.keys(Person.prototype);
console.log(keys);        // "name,age,job,sayName"

let p1 = new Person();
p1.name = "Rob";
p1.age = 31;
let p1keys = Object.keys(p1);
console.log(p1keys);    // "name,age"
```

Здесь переменной `keys` назначается массив со строками `"name"`, `"age"`, `"job"` и `"sayName"`. В таком порядке они отображались бы и в цикле `for-in`. При вызове для экземпляра `Person` метод `Object.keys()` возвращает массив со строками `name` и `age`, двумя свойствами экземпляра.

Если вам нужен список всех свойств экземпляра, перечислимых и неперечислимых, можно аналогичным образом использовать метод `Object.getOwnPropertyNames()`:

```
let keys = Object.getOwnPropertyNames(Person.prototype);
console.log(keys);        // "constructor,name,age,job,sayName"
```

Обратите внимание на то, что список результатов содержит неперечислимое свойство `constructor`. Методы `Object.keys()` и `Object.getOwnPropertyNames()` могут быть полезны как альтернатива циклу `for-in`.

С введением символов в ECMAScript 6 стала очевидной необходимость введения родственного метода для `Object.getOwnPropertyNames()`, потому что свойства с символьными ключами не имеют понятия имени. Поэтому был представлен `Object.getOwnPropertySymbols()`, который предлагает то же поведение, что и `Object.getOwnPropertyNames()`, но в отношении символов:

```
let k1 = Symbol('k1'),
    k2 = Symbol('k2');

let o = {
    [k1]: 'k1',
    [k2]: 'k2'
};

console.log(Object.getOwnPropertySymbols(o));
// [Symbol(k1), Symbol(k2)]
```

Порядок перечисления свойств

Циклы `for-in`, `Object.keys()`, `Object.getOwnPropertyNames/Symbols()` и `Object.assign()` имеют важное различие в том, что касается порядка перечисления свойств. Циклы `for-in` и `Object.keys()` не имеют детерминированного порядка перечисления — они определяются механизмом JavaScript и могут различаться в зависимости от браузера.

Однако `Object.getOwnPropertyNames()`, `Object.getOwnPropertySymbols()` и `Object.assign()` *имеют* определенный порядок перечисления. Сначала будут перечислены числовые ключи в порядке возрастания, затем строковые и символьные ключи — в порядке вставки. Ключи, определенные внутри литерала объекта, будут вставлены в порядке разделения их запятой.

```
let k1 = Symbol('k1'),
    k2 = Symbol('k2');

let o = {
    1: 1,
    first: 'first',
    [k1]: 'sym2',
    second: 'second',
    0: 0
};

o[k2] = 'sym2';
o[3] = 3;
o.third = 'third';
0[2] = 2;

console.log(Object.getOwnPropertyNames(o));
```

```
// ["0", "1", "3", "first", "second", "third"]

console.log(Object.getOwnPropertySymbols(o));
// [Symbol(k1), Symbol(k2)]
```

Итерация по объекту

Для большей части истории JavaScript итерация по свойствам объекта была грязным делом. В ECMAScript 2017 были представлены два статических метода для преобразования содержимого объекта в сериализованный и, что более важно, итеративный формат. Эти статические методы, `Object.values()` и `Object.entries()`, принимают объект и возвращают его содержимое в массиве. `Object.values()` возвращает массив значений объекта, а `Object.entries()` возвращает массив пар массивов, каждая из которых представляет пару `[key, value]` в объекте.

Пример с данными методами приведен ниже:

```
const o = {
    foo: 'bar',
    baz: 1,
    qux: {}
};

console.log(Object.values(o));
// ["bar", 1, {}]

console.log(Object.entries((o)));
// [["foo", "bar"], ["baz", 1], ["qux", {}]]
```

Обратите внимание, что нестроковые свойства преобразуются в строки в выходном массиве. Кроме того, метод выполняет поверхностное копирование объекта:

```
const o = {
    qux: {}
};

console.log(Object.values(o)[0] === o.qux);
// true

console.log(Object.entries(o)[0][1] === o.qux);
// true
```

Свойства с символьным ключом игнорируются:

```
const sym = Symbol();
const o = {
    [sym]: 'foo'
};

console.log(Object.values(o));
// []

console.log(Object.entries((o)));
// []
```

Альтернативный синтаксис прототипов

В предыдущем примере имя `Person.prototype` нужно было вводить при добавлении каждого свойства и метода. Чтобы писать меньше кода и нагляднее группировать элементы прототипов, чаще просто перезаписывают прототип литералом объекта, содержащим все свойства и методы, например:

```
function Person() {
}

Person.prototype = {
    name : "Nicholas",
    age : 29,
    job : "Software Engineer",
    sayName() {
        console.log(this.name);
    }
};
```

В этом примере свойству `Person.prototype` присваивается новый объект, созданный с помощью литерала. Результат получается таким же за одним исключением: свойство `constructor` больше не указывает на объект `Person`. При создании функции создается ее объект `prototype` и автоматически задается свойство `constructor`. Новый синтаксис полностью перезаписывает объект `prototype`, предлагаемый по умолчанию, после чего свойство `constructor` указывает на совершенно новый объект (конструктор `Object`), а не на саму функцию. Хотя оператор `instanceof` по-прежнему работает, полагаться на то, что свойство `constructor` показывает тип объекта, больше нельзя:

```
let friend = new Person();

console.log(friend instanceof Object);        // true
console.log(friend instanceof Person);        // true
console.log(friend.constructor == Person);    // false
console.log(friend.constructor == Object);    // true
```

Здесь оператор `instanceof` все еще возвращает `true` и для `Object`, и для `Person`, однако свойство `constructor` теперь имеет значение `Object`, а не `Person`. Если значение `constructor` важно, можно явно восстановить его:

```
function Person() {
}

Person.prototype = {
    constructor: Person,
    name : "Nicholas",
    age : 29,
    job : "Software Engineer",
    sayName() {
        console.log(this.name);
    }
};
```

Добавление свойства `constructor` со значением `Person` в литерал объекта решает проблему.

Помните, что у конструктора, восстановленного таким способом, атрибут `[[Enumerable]]` имеет значение `true`. Свойство `constructor` встроенных объектов по умолчанию неперечислимо, поэтому если ваш интерпретатор JavaScript совместим с ECMAScript 5, возможно, лучше использовать с той же целью метод `Object.defineProperty()`:

```
function Person() {}

Person.prototype = {
    name : "Nicholas",
    age : 29,
    job : "Software Engineer",
    sayName() {
        console.log(this.name);
    }
};

// восстановление конструктора
Object.defineProperty(Person.prototype, "constructor", {
    enumerable: false,
    value: Person
});
```

Динамическая природа прототипов

Поскольку при доступе к значению, которого нет в экземпляре, выполняется его поиск в прототипе, изменения прототипа немедленно отражаются на экземплярах, в том числе в тех, которые существовали до изменения, например:

```
let friend = new Person();

Person.prototype.sayHi = function() {
    console.log("hi");
};

friend.sayHi(); // "hi" – все работает!
```

В этом коде создается экземпляр `Person`, который сохраняется в переменной `friend`. Затем к свойству `Person.prototype` добавляется метод `sayHi()`. Несмотря на то что экземпляр `friend` был создан до этого изменения, новый метод доступен ему благодаря связи с прототипом. При вызове `friend.sayHi()` сначала выполняется поиск свойства `sayHi` в экземпляре; когда выясняется, что его в экземпляре нет, поиск продолжается в прототипе. Так как экземпляр лишь связан с прототипом с помощью указателя, а не является его копией, интерпретатор обнаруживает в прототипе новое свойство `sayHi` и возвращает назначенную ему функцию.

Хотя свойства и методы, добавленные в прототип, немедленно становятся доступны во всех экземплярах объектов, при перезаписи всего прототипа наблюдается другое поведение. Указатель `[[Prototype]]` задается при вызове конструктора, поэтому изменение прототипа на другой объект нарушает связь между конструктором

и оригинальным прототипом. Помните, у экземпляра есть указатель только на прототип, но не на конструктор. Рассмотрим следующий пример:

```
function Person() {}

let friend = new Person();

Person.prototype = {
    constructor: Person,
    name : "Nicholas",
    age : 29,
    job : "Software Engineer",
    sayName() {
        console.log(this.name);
    }
};

friend.sayName();        // ошибка
```

В этом примере создается экземпляр типа `Person`, а затем объект его прототипа перезаписывается. При вызове метода `friend.sayName()` возникает ошибка, потому что у прототипа, на который указывает экземпляр `friend`, нет свойства с таким именем. Рисунок 8.3 поясняет, почему это происходит.

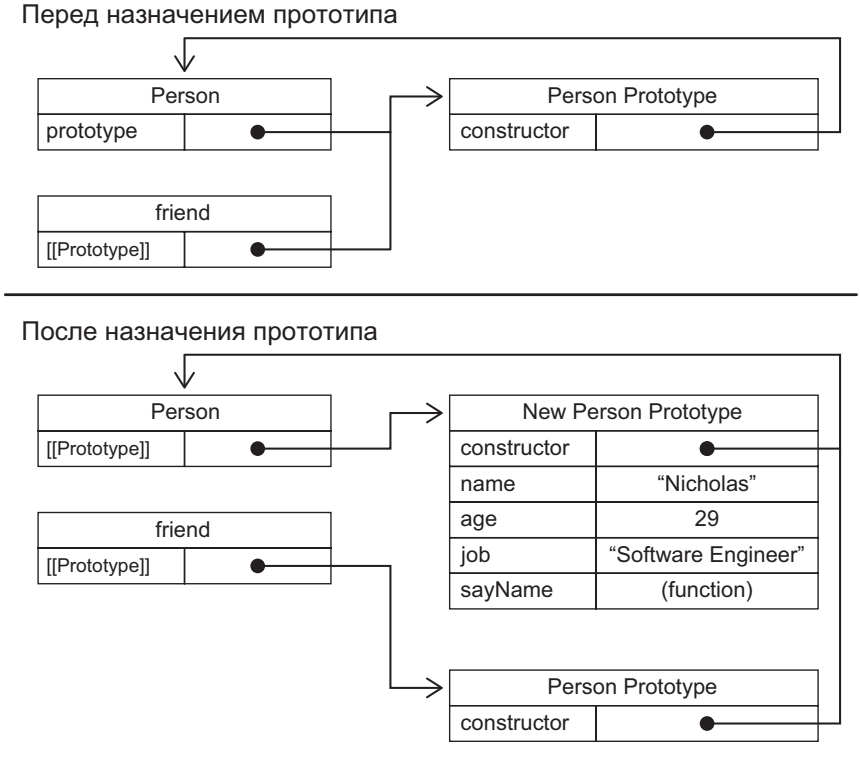

Рис. 8.3

Перезапись прототипа у конструктора приводит к тому, что новые экземпляры ссылаются на новый прототип, а уже существующие — на старый.

Прототипы встроенных объектов

Паттерн Прототип важен не только для определения пользовательских типов, этот паттерн может также служить для реализации всех встроенных ссылочных типов. В каждом из этих типов (включая `Object`, `Array`, `String` и т. д.) методы определены в прототипе конструктора. Например, метод `sort()` принадлежит свойству `Array.prototype`, а `substring()` — свойству `String.prototype`:

```
console.log(typeof Array.prototype.sort);            // "function"
console.log(typeof String.prototype.substring);      // "function"
```

С помощью прототипов встроенных объектов можно получать ссылки на методы, предлагаемые по умолчанию, и определять новые методы. Кроме того, эти прототипы можно изменять, что позволяет добавлять методы к встроенным объектам. Например, следующий код добавляет метод `startsWith()` к оболочке примитивного типа `String`:

```
String.prototype.startsWith = function (text) {
    return this.indexOf(text) == 0;
};

let msg = "Hello world!";
console.log(msg.startsWith("Hello"));    // true
```

В этом примере метод `startsWith()` возвращает `true`, если переданный в него текст совпадает с началом строки. Поскольку метод назначен свойству `String.prototype`, его можно использовать со всеми строками в среде. Чтобы метод `startsWith()` стал доступен для строки `msg`, для нее неявно создается оболочка примитивного типа `String`.

> **ПРИМЕЧАНИЕ** Изменять прототипы встроенных объектов в окончательном коде не рекомендуется. Это затрудняет понимание кода и может вызывать конфликты имен, когда метод, не являющийся встроенным в одном браузере, является таковым в другом. Кроме того, так можно случайно перезаписать встроенный метод. Предпочтительным способом является создание пользовательского класса, который наследуется от встроенного типа.

Проблемы прототипов

К сожалению, у паттерна Прототип есть недостатки. Прежде всего, он не позволяет передать в конструктор аргументы инициализации, из-за чего свойства всех экземпляров имеют по умолчанию одинаковые значения. Это неудобно, но главная проблема прототипов связана с тем, что многие экземпляры используют их совместно.

Все свойства прототипа являются общими для всех экземпляров, что идеально для функций. Свойства с примитивными значениями также работают нормально,

потому что свойство прототипа можно затенить, создав в экземпляре одноименное свойство. Реальная проблема возникает, если свойство содержит ссылочное значение. Рассмотрим пример:

```
function Person() {}

Person.prototype = {
    constructor: Person,
    name : "Nicholas",
    age : 29,
    job : "Software Engineer",
    friends : ["Shelby", "Court"],
    sayName : function () {
        console.log(this.name);
    }
};

let person1 = new Person();
let person2 = new Person();

person1.friends.push("Van");

console.log(person1.friends);        // "Shelby,Court,Van"
console.log(person2.friends);        // "Shelby,Court,Van"
console.log(person1.friends === person2.friends);        // true
```

Здесь для объекта `Person.prototype` определяется свойство `friends`, содержащее массив строк, после чего создаются два экземпляра `Person` и в массив `person1.friends` добавляется новая строка. Поскольку массив `friends` относится к свойству `Person.prototype`, а не к объекту `person1`, внесенные изменения отражаются в свойстве `person2.friends`, которое указывает на тот же массив. Если нужно, чтобы массив был общим для всех экземпляров, это нормально, однако обычно экземпляры должны иметь собственные копии всех свойств. По этой причине паттерн Прототип редко используется сам по себе.

НАСЛЕДОВАНИЕ

В контексте объектно-ориентированного программирования чаще всего вспоминают и обсуждают концепцию наследования. Многие объектно-ориентированные языки поддерживают наследование либо интерфейса, то есть только сигнатур методов, либо реализации, то есть фактических методов. Из-за того что в языке ECMAScript у функций нет сигнатур, наследование интерфейса в нем невозможно, а поддерживается только наследование реализации, которое выполняется преимущественно с помощью цепочек прототипов.

Цепочки прототипов

В ECMA-262 указано, что главный метод наследования в ECMAScript основан на *цепочках прототипов* (prototype chains). Его идея в том, что с помощью

прототипов один ссылочный тип получает свойства и методы другого. Если помните, у каждого конструктора есть объект прототипа, который указывает на этот конструктор, а у экземпляров есть внутренний указатель на прототип. А что, если прототипом окажется экземпляр другого типа? В этом случае у самого прототипа будет указатель на очередной прототип, у которого, в свою очередь, будет указатель на следующий конструктор. Если вторым прототипом также будет экземпляр другого типа, последовательность продолжится, формируя цепочку между экземплярами и прототипами. На этой идее и основано наследование с помощью цепочек прототипов.

Реализация цепочки прототипов включает следующий шаблон кода:

```
function SuperType() {
    this.property = true;
}

SuperType.prototype.getSuperValue = function() {
    return this.property;
};

function SubType() {
    this.subproperty = false;
}

// наследование от SuperType
SubType.prototype = new SuperType();

SubType.prototype.getSubValue = function () {
    return this.subproperty;
};

let instance = new SubType();
console.log(instance.getSuperValue());    // true
```

В этом коде определяются типы `SuperType` и `SubType`, оба с одним свойством и с одним методом. Главное различие между двумя типами в том, что `SubType` наследуется от `SuperType`, для чего новый экземпляр `SuperType` назначается свойству `SubType.prototype`. В результате оригинальный прототип перезаписывается новым объектом, то есть все свойства и методы экземпляра `SuperType` появляются у `SubType.prototype`. После наследования к свойству `SubType.prototype` добавляется новый метод в дополнение к тем, что были унаследованы от `SuperType`. Отношения между экземпляром, конструкторами и прототипами показаны на рис. 8.4.

Таким образом, вместо прототипа, предлагаемого по умолчанию, типу `SubType` назначается новый прототип, который является экземпляром `SuperType`. Он не только получает свойства и методы экземпляра `SuperType`, но и указывает на прототип `SuperType`. В итоге объект `instance` указывает на свойство `SubType.prototype`, указывающее на `SuperType.prototype`. Заметьте, что метод `getSuperValue()` остается у объекта `SuperType.prototype`, а свойство `property` переходит к `SubType.prototype`. Это объясняется тем, что `getSuperValue()` — метод прототипа, а `property` — свойство

экземпляра. `SubType.prototype` теперь является экземпляром `SuperType`, поэтому и свойство `property` хранится в нем. Отметим также, что `instance.constructor` указывает на `SuperType`, потому что свойство `constructor` у объекта `SubType.prototype` было перезаписано.

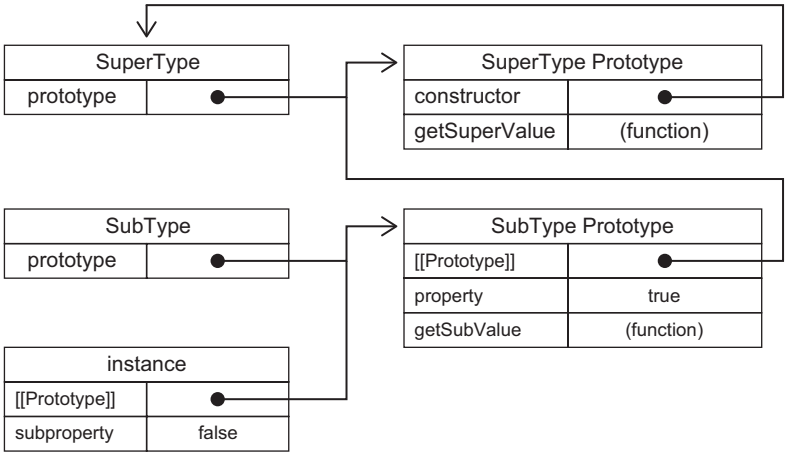

Рис. 8.4

Цепочки прототипов учитываются механизмом поиска в прототипах. Как вы, наверное, помните, при чтении свойства сначала выполняется его поиск в экземпляре, а если свойство там не обнаруживается, инициируется поиск в прототипе. Если наследование реализовано с помощью цепочки прототипов, поиск может быть продолжен в ней. Так, в предыдущем примере при вызове `instance.getSuperValue()` выполняется поиск метода в экземпляре, затем — в свойстве `SubType.prototype` и, наконец, — в свойстве `SuperType.prototype`, где он и определен. Поиск свойств и методов всегда продолжается до конца цепочки прототипов.

Прототипы, предлагаемые по умолчанию

На самом деле в цепочке прототипов есть еще одно звено. Все ссылочные типы наследуются по умолчанию через цепочку прототипов от типа `Object`, а это означает, что в любой функции внутренний указатель указывает на прототип `Object.prototype`. Именно так пользовательские типы наследуют все предлагаемые по умолчанию методы типа `Object`, например `toString()` и `valueOf()`. Полностью цепочка прототипов с дополнительным уровнем наследования для предыдущего примера показана на рис. 8.5.

Тип `SubType` наследуется от `SuperType`, а `SuperType` — от `Object`. Если добавить в код вызов `instance.toString()`, будет вызван метод, принадлежащий свойству `Object.prototype`.

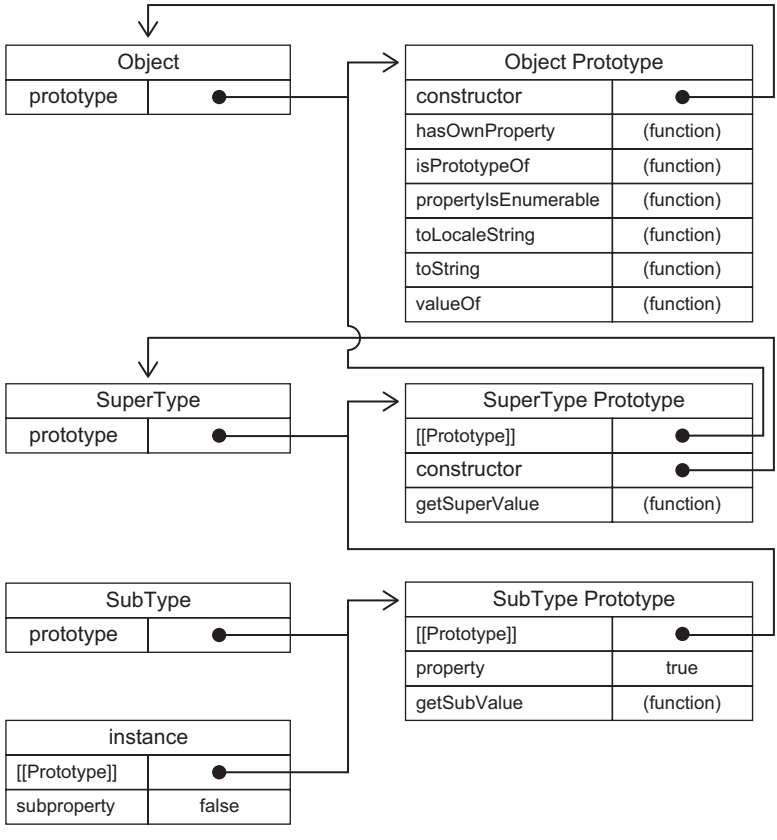

Рис. 8.5

Связи между прототипами и экземплярами

Есть два способа выяснить, связаны ли экземпляр и прототип. Первый — использовать оператор `instanceof`, который возвращает `true`, если указанный конструктор имеется в цепочке прототипов экземпляра, например:

```
console.log(instance instanceof Object);       // true
console.log(instance instanceof SuperType);    // true
console.log(instance instanceof SubType);      // true
```

Объект `instance` благодаря цепочке прототипов является экземпляром типов `Object`, `SuperType` и `SubType`, поэтому оператор `instanceof` возвращает `true` для всех трех конструкторов.

Второй способ основан на имеющемся у каждого прототипа в цепочке методе `isPrototypeOf()`, который возвращает `true`, если переданный ему экземпляр входит в цепочку, например:

```
console.log(Object.prototype.isPrototypeOf(instance));        // true
console.log(SuperType.prototype.isPrototypeOf(instance));     // true
console.log(SubType.prototype.isPrototypeOf(instance));       // true
```

Работа с методами

В подтипе часто требуется переопределить метод супертипа или реализовать новые методы, отсутствующие в супертипе. Для этого необходимо добавить методы в прототип после его назначения подтипу. Рассмотрим пример:

```
function SuperType() {
    this.property = true;
}

SuperType.prototype.getSuperValue = function() {
    return this.property;
};

function SubType() {
    this.subproperty = false;
}

// наследование от SuperType
SubType.prototype = new SuperType();

// новый метод
SubType.prototype.getSubValue = function () {
    return this.subproperty;
};

// переопределение существующего метода
SubType.prototype.getSuperValue = function () {
    return false;
};

let instance = new SubType();
console.log(instance.getSuperValue());        // false
```

Выделенный фрагмент содержит определения двух методов. Первый, `getSubValue()`, представляет собой новый метод типа `SubType`, а второй, `getSuperValue()`, затеняет одноименный метод, который уже имеется в цепочке прототипов. При вызове `getSuperValue()` для экземпляра `SubType` вызывается новая версия, но для экземпляров `SuperType` по-прежнему вызывается оригинал. Следует отметить, что оба метода определяются после назначения экземпляра `SuperType` прототипом `SubType`.

При использовании цепочки прототипов нельзя создавать методы прототипа с помощью литерала объекта, потому что это перезаписывает цепочку, например:

```
function SuperType() {
    this.property = true;
}

SuperType.prototype.getSuperValue = function() {
    return this.property;
};
```

```
function SubType() {
    this.subproperty = false;
}

// наследование от SuperType
SubType.prototype = new SuperType();

// попытка добавить новые методы аннулирует предыдущую строку
SubType.prototype = {
    getSubValue() {
        return this.subproperty;
    },

    someOtherMethod() {
        return false;
    }
};

let instance = new SubType();
console.log(instance.getSuperValue());    //ошибка!
```

В этом коде экземпляр `SuperType`, сделанный первоначально прототипом `SubType`, заменяется литералом объекта. В результате прототипом становится новый экземпляр `Object`, что разрывает цепочку прототипов и, соответственно, связь между типами `SubType` и `SuperType`.

Проблемы с цепочками прототипов

Хотя цепочка прототипов — эффективное средство наследования, оно не лишено недостатков. Главная проблема связана с прототипами, содержащими ссылочные значения. Напомню, что свойства прототипов с такими значениями являются общими для всех экземпляров, поэтому свойства обычно определяют в конструкторе, а не в прототипе. Когда наследование реализуется на основе прототипов, в качестве прототипа на самом деле используется экземпляр другого типа, при этом свойства экземпляра становятся свойствами прототипа. Эту проблему поясняет следующий пример:

```
function SuperType() {
    this.colors = ["red", "blue", "green"];
}

function SubType() {
}

// наследование от SuperType
SubType.prototype = new SuperType();

let instance1 = new SubType();
instance1.colors.push("black");
console.log(instance1.colors);        // "red,blue,green,black"

let instance2 = new SubType();
console.log(instance2.colors);        // "red,blue,green,black"
```

Здесь в конструкторе `SuperType` определяется свойство `colors`, содержащее массив (ссылочное значение). Каждый экземпляр `SuperType` имеет собственное свойство `colors` с отдельным массивом. При наследовании типа `SubType` от `SuperType` с помощью цепочки прототипов объект `SubType.prototype` становится экземпляром `SuperType` и получает собственное свойство `colors`, что аналогично явному созданию свойства `SubType.prototype.colors`. В результате все экземпляры `SubType` совместно используют одно свойство `colors`. Это подтверждается тем, что изменения свойства `instance1.colors` отражаются на `instance2.colors`.

Вторая проблема с цепочкой прототипов заключается в том, что при создании экземпляра подтипа нельзя передать аргументы в конструктор супертипа. Фактически нет способа передать аргументы в конструктор супертипа, не затронув все его экземпляры. Из-за этих двух проблем цепочки прототипов редко используются сами по себе.

Кража конструктора

Пытаясь решить проблему наследования ссылочных значений прототипов, разработчики начали использовать паттерн *Кража конструктора* (constructor stealing), который часто называют маскировкой объекта или классическим наследованием. Суть этого приема проста и сводится к вызову конструктора супертипа в конструкторе подтипа. Так как функции — это просто объекты, выполняющие код в конкретном контексте, мы можем с помощью методов `apply()` и `call()` вызывать конструкторы для только что созданных объектов, например:

```
function SuperType() {
    this.colors = ["red", "blue", "green"];
}

function SubType() {
    // наследование от SuperType
    SuperType.call(this);
}
let instance1 = new SubType();
instance1.colors.push("black");

console.log(instance1.colors);        // "red,blue,green,black"

let instance2 = new SubType();
console.log(instance2.colors);        // "red,blue,green"
```

Кража конструктора выполняется в выделенной строке. Для этого метод `call()` (который можно заменить методом `apply()`) вызывает конструктор `SuperType` в контексте созданного экземпляра `SubType`, выполняя для него весь содержащийся в конструкторе код инициализации. Благодаря этому каждый экземпляр получает собственную копию свойства `colors`.

Передача аргументов

Одним из преимуществ кражи конструктора над цепочками прототипов является возможность передать в конструкторе подтипа аргументы в конструктор супертипа. Рассмотрим пример:

```
function SuperType(name) {
    this.name = name;
}

function SubType() {
    // наследование от SuperType с передачей аргумента в супертип
    SuperType.call(this, "Nicholas");

    // свойство экземпляра
    this.age = 29;
}

let instance = new SubType();
console.log(instance.name);    // "Nicholas";
console.log(instance.age);     // 29
```

В этом коде конструктор `SuperType` принимает единственный аргумент `name`, который просто назначается свойству. В конструкторе типа `SubType` можно передать значение конструктору `SuperType`, задав значение свойства `name` для экземпляра `SubType`. Чтобы гарантировать, что другие свойства подтипа не будут перезаписаны в конструкторе `SuperType`, их можно определить после его вызова.

Проблемы с кражами конструктора

Кража конструктора имеет тот же недостаток, что и паттерн Конструктор с пользовательскими типами: необходимость определять методы в конструкторе не позволяет задействовать их многократно. Кроме того, методы, определенные в прототипе супертипа, недоступны в подтипе, вследствие чего все типы могут использовать только паттерн Конструктор. Из-за этих проблем кража конструктора также редко используется сама по себе.

Комбинированное наследование

Комбинированное наследование (combination inheritance), которое иногда называют псевдоклассическим, объединяет преимущества цепочки прототипов и кражи конструктора. Его идея состоит в том, что для наследования свойств и методов прототипа используется цепочка прототипов, а для наследования свойств экземпляра — кража конструктора. Определение методов в прототипе обеспечивает многократное использование функций, но при этом каждый экземпляр может иметь собственные свойства. Рассмотрим следующий пример:

```
function SuperType(name) {
    this.name = name;
    this.colors = ["red", "blue", "green"];
}

SuperType.prototype.sayName = function() {
    console.log(this.name);
};

function SubType(name, age) {
```

```
    // наследование свойств
    SuperType.call(this, name);

    this.age = age;
}

// наследование методов
SubType.prototype = new SuperType();

SubType.prototype.sayAge = function() {
    console.log(this.age);
};

let instance1 = new SubType("Nicholas", 29);
instance1.colors.push("black");
console.log(instance1.colors);    // "red,blue,green,black"
instance1.sayName();        // "Nicholas";
instance1.sayAge();         // 29

let instance2 = new SubType("Greg", 27);
console.log(instance2.colors);    // "red,blue,green"
instance2.sayName();        // "Greg";
instance2.sayAge();         // 27
```

Здесь в конструкторе `SuperType` определяются свойства `name` и `colors`, а к прототипу `SuperType` добавляется единственный метод `sayName()`. В конструкторе `SubType` мы вызываем конструктор `SuperType`, передавая в него аргумент `name`, и определяем свойство `age`, после чего назначаем экземпляр `SuperType` прототипом `SubType` и определяем в подтипе новый метод `sayAge()`. Завершается этот код созданием двух отдельных экземпляров `SubType` с собственными копиями свойств (включая `colors`), но с общими методами.

Комбинированное наследование преодолело недостатки цепочки прототипов и кражи конструктора и стало самым популярным паттерном наследования в JavaScript. Оно позволяет использовать оператор `instanceof` и метод `isPrototypeOf()` для идентификации связей между объектами.

Прототипное наследование

В 2006 г. Дуглас Крокфорд (Douglas Crockford) в статье «Prototypal Inheritance in JavaScript» («Прототипное наследование в JavaScript») представил способ наследования без строго определенных конструкторов. Идея заключалась в том, что с помощью прототипов можно создавать новые объекты на основе существующих без определения пользовательских типов. Функция, которую он привел в качестве примера, выглядела так:

```
function object(o) {
    function F() {}
    F.prototype = o;
    return new F();
}
```

Функция `object()` создает временный конструктор, назначает полученный объект прототипом конструктора и возвращает новый экземпляр временного типа. По сути, она выполняет поверхностное копирование любого переданного в нее объекта. Рассмотрим такой пример:

```
let person = {
    name: "Nicholas",
    friends: ["Shelby", "Court", "Van"]
};

let anotherPerson = object(person);
anotherPerson.name = "Greg";
anotherPerson.friends.push("Rob");

let yetAnotherPerson = object(person);
yetAnotherPerson.name = "Linda";
yetAnotherPerson.friends.push("Barbie");

console.log(person.friends);    // "Shelby,Court,Van,Rob,Barbie"
```

Согласно Крокфорду, если некоторый объект нужно использовать как основу для создания другого объекта, следует передать его в функцию `object()` и внести нужные изменения в возвращенный объект. В приведенном примере объект `person` содержит информацию, которая должна быть доступна в других объектах, поэтому мы передаем его в функцию `object()`, которая возвращает новый объект. Прототипом этого объекта является экземпляр `person` с примитивным и ссылочным свойствами. После двух вызовов функции `object()` ссылочное свойство `person.friends` является общим для трех объектов: `person`, `anotherPerson` и `yetAnotherPerson`. По сути, этот код создает два клона `person`.

В ECMAScript 5 концепция прототипного наследования была формализована в методе `Object.create()`. Он принимает два аргумента: прототип нового объекта и необязательный объект, определяющий для нового объекта дополнительные свойства. Если второй аргумент опущен, метод `Object.create()` эквивалентен методу `object()`:

```
let person = {
    name: "Nicholas",
    friends: ["Shelby", "Court", "Van"]
};

let anotherPerson = Object.create(person);
anotherPerson.name = "Greg";
anotherPerson.friends.push("Rob");

let yetAnotherPerson = Object.create(person);
yetAnotherPerson.name = "Linda";
yetAnotherPerson.friends.push("Barbie");

console.log(person.friends);    // "Shelby,Court,Van,Rob,Barbie"
```

Второй аргумент метода `Object.create()` имеет такой же формат, что и в методе `Object.defineProperties()`: каждое дополнительное свойство указывается вместе

с его дескриптором. Любые свойства, добавленные таким образом, затеняют одноименные свойства прототипа:

```
let person = {
    name: "Nicholas",
    friends: ["Shelby", "Court", "Van"]
};

let anotherPerson = Object.create(person, {
    name: {
        value: "Greg"
    }
});

console.log(anotherPerson.name);    // "Greg"
```

Прототипное наследование полезно, если накладные расходы на создание отдельных конструкторов нежелательны, но при этом требуется, чтобы новый объект был похож на уже существующий. Помните, что свойства со ссылочными значениями в этом случае будут общими, как и при использовании паттерна Прототип.

Паразитное наследование

С прототипным наследованием тесно связана концепция *паразитного наследования* (parasitic inheritance), также популяризированная Крокфордом. Паттерн Паразитное наследование похож как на паттерн Паразитный конструктор, так и на паттерн Фабрика. Он включает создание объекта в функции, расширение его возможностей и возвращение расширенного объекта из функции. Базовый паттерн Паразитное наследование выглядит следующим образом:

```
function createAnother(original) {
    let clone = object(original);    // создание объекта путем вызова функции
    clone.sayHi = function() {       // расширение возможностей объекта
        console.log("hi");
    };
    return clone;                    // возвращение объекта
}
```

Функция `createAnother()` принимает в качестве единственного аргумента объект, на основе которого нужно создать производный объект. Полученный аргумент передается в функцию `object()`, а возвращенный из нее результат назначается переменной `clone`. К объекту `clone` добавляется метод `sayHi()`, а затем расширенный объект возвращается. Функцию `createAnother()` можно использовать следующим образом:

```
let person = {
    name: "Nicholas",
    friends: ["Shelby", "Court", "Van"]
};

let anotherPerson = createAnother(person);
anotherPerson.sayHi();    // "hi"
```

В этом фрагменте на основе объекта `person` создается объект `anotherPerson`, который содержит все свойства и методы `person`, а также дополнительный метод `sayHi()`.

Паразитное наследование — это еще один паттерн, который полезен, если вы имеете дело в основном с объектами, а не собственными типами и конструкторами. Метод `object()` для паразитного наследования не требуется — подойдет любая функция, возвращающая новый объект.

> **ПРИМЕЧАНИЕ** Как и в паттерне Конструктор, функции, добавленные к объектам при паразитном наследовании, не используются повторно, что делает код менее эффективным.

Паразитное комбинированное наследование

Комбинированное наследование имеет свои слабые стороны. Самое неэффективное в этом паттерне то, что конструктор супертипа всегда вызывается дважды: в первый раз для создания прототипа подтипа, а во второй — внутри конструктора подтипа. По сути, прототип подтипа получает все свойства экземпляра супертипа только для того, чтобы перезаписать их в конструкторе подтипа. Давайте еще раз взглянем на пример с комбинированным наследованием:

```
function SuperType(name) {
    this.name = name;
    this.colors = ["red", "blue", "green"];
}

SuperType.prototype.sayName = function() {
    console.log(this.name);
};

function SubType(name, age) {
    SuperType.call(this, name);             // второй вызов SuperType()

    this.age = age;
}

SubType.prototype = new SuperType();        // первый вызов SuperType()
SubType.prototype.constructor = SubType;
SubType.prototype.sayAge = function() {
    console.log(this.age);
};
```

Выделенные строки кода указывают, когда вызывается конструктор `SuperType`. При выполнении этого кода к объекту `SubType.prototype` добавляются свойства `name` и `colors`, которые первоначально являются свойствами экземпляра типа `SuperType`. При последующем вызове конструктора `SubType` в нем вызывается конструктор `SuperType`, который создает свойства экземпляра `name` и `colors` у нового объекта, маскируя свойства прототипа. Этот процесс показан на рис. 8.6.

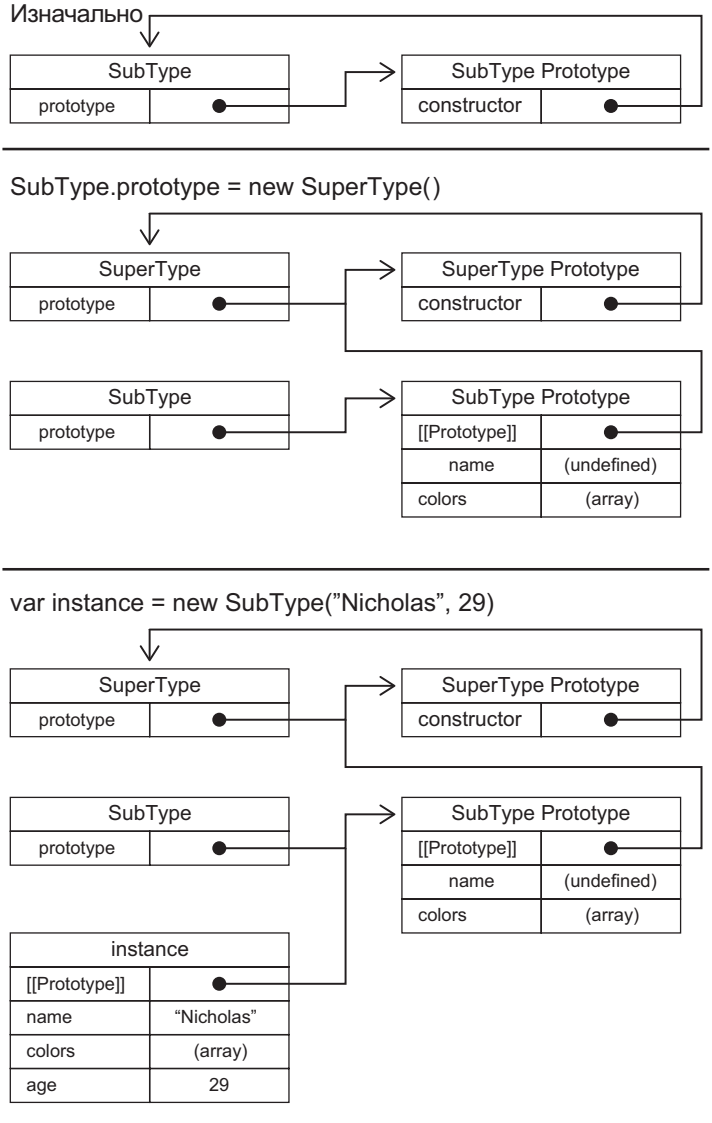

Рис. 8.6

Как видите, из-за двукратного вызова конструктора `SuperType` свойства `name` и `colors` есть как у экземпляра, так и у прототипа `SubType`. К счастью, это можно исправить.

При *паразитном комбинированном наследовании* (parasitic combination inheritance) мы используем кражу конструктора для наследования свойств и гибридную форму цепочки прототипов для наследования методов. Идея в том, что вместо назначения прототипа подтипу путем вызова конструктора супертипа мы просто используем копию прототипа супертипа. Иначе говоря, выполняется паразитное наследование

прототипа супертипа, после чего результат назначается прототипу подтипа. Базовый паттерн таков:

```
function inheritPrototype(subType, superType) {
    let prototype = object(superType.prototype); // создание объекта
    prototype.constructor = subType;             // расширение объекта
    subType.prototype = prototype;               // назначение объекта
}
```

Функция `inheritPrototype()` реализует очень простое паразитное комбинированное наследование. Она принимает два аргумента: конструктор подтипа и конструктор супертипа. Внутри функции первым делом создается клон прототипа супертипа. Затем к прототипу добавляется свойство `constructor`, чтобы компенсировать потерю первоначального свойства конструктора при перезаписи прототипа. Наконец, созданный объект назначается прототипом подтипа. Вызовом функции `inheritPrototype()` можно заменить назначение прототипа подтипа в предыдущем примере:

```
function SuperType(name) {
    this.name = name;
    this.colors = ["red", "blue", "green"];
}

SuperType.prototype.sayName = function() {
    console.log(this.name);
};

function SubType(name, age) {
    SuperType.call(this, name);

    this.age = age;
}

inheritPrototype(SubType, SuperType);

SubType.prototype.sayAge = function() {
    console.log(this.age);
};
```

Этот код более эффективен, потому что конструктор `SuperType` вызывается только один раз и лишние свойства в прототипе `SubType` не создаются. Кроме того, цепочка прототипов остается нетронутой, что сохраняет работоспособность оператора `instanceof` и метода `isPrototypeOf()`. Паразитное комбинированное наследование считается оптимальным способом наследования ссылочных типов.

КЛАССЫ

Предыдущие разделы представляют собой углубленный обзор того, как можно было эмулировать поведение классов, используя только функции, доступные в ECMAScript 5. Нетрудно догадаться, что показанные стратегии представляли различные проблемы и компромиссы. Помимо этого, синтаксис был чрезмерно многословным и, бесспорно, грязным.

Для решения всех этих проблем недавно в ECMAScript 6 появилась возможность формально определять классы с помощью ключевого слова `class`. Классы — это принципиально новая синтаксическая конструкция в ECMAScript, и поэтому они могут поначалу казаться незнакомыми. Хотя классы ECMAScript 6, по-видимому, содержат каноническое объектно-ориентированное программирование, они все еще используют концепции прототипов и конструкторов.

Основы определения классов

Подобно типу функции, есть два основных способа определения класса: объявления класса и классовые выражения. Оба используют ключевое слово `class` и фигурные скобки:

```
// объявление класса
class Person {}

// классовое выражение
const Animal = class {};
```

Подобно функциональным выражениям, на классовые выражения нельзя ссылаться до тех пор, пока они не будут определены при выполнении. Тем не менее важным отклонением от параллельного поведения определения функции является то, что объявления функций поднимаются, а объявления классов нет:

```
console.log(FunctionExpression);    // undefined
var FunctionExpression = function() {};
console.log(FunctionExpression);    // function() {}

console.log(FunctionDeclaration);    // FunctionDeclaration() {}
function FunctionDeclaration() {}
console.log(FunctionDeclaration);    // FunctionDeclaration() {}

console.log(ClassExpression);    // undefined
var ClassExpression = class {};
console.log(ClassExpression);    // class {}

console.log(ClassDeclaration);    // ReferenceError: ClassDeclaration is not
defined
class ClassDeclaration {}
console.log(ClassDeclaration);    // class ClassDeclaration {}
```

Кроме того, в отличие от объявлений функций, которые принадлежат области видимости функции, объявления классов принадлежат области видимости блока:

```
{
    function FunctionDeclaration() {}
    class ClassDeclaration {}
}

console.log(FunctionDeclaration);    // FunctionDeclaration() {}
console.log(ClassDeclaration);       // ReferenceError: ClassDeclaration
                                     // is not defined
```

Композиция классов

Класс может состоять из метода конструктора класса, методов экземпляра, методов чтения и записи свойств и статических методов классов. Ни один из этих элементов не является обязательным; пустое определение класса является допустимым синтаксисом. По умолчанию все внутри определения класса выполняется в строгом режиме.

Как и в случае с конструкторами функций, большинство руководств по стилю направят вас на использование имени класса с заглавной буквы, чтобы отличать его от созданных из него экземпляров (например, класс `Foo {}` может создавать экземпляр `foo`):

```
// Корректное объявление пустого класса
class Foo {}

// Корректное объявление класса с конструктором
class Bar {
    constructor() {}
}

// Корректное объявление класса с методом получения свойств
class Baz {
    get myBaz() {}
}

// Корректное объявление класса со статическим методом
class Qux {
    static myQux() {}
}
```

Классовые выражения могут быть названы произвольно. Когда выражение присваивается переменной, свойство name может использоваться для извлечения строки имени классового выражения, но сам идентификатор недоступен за пределами области видимости классового выражения.

```
let Person = class PersonName {
    identify() {
        console.log(Person.name, PersonName.name);
    }
}

let p = new Person();

p.identify();    // PersonName, PersonName

console.log(Person.name);    // PersonName
console.log(PersonName);     // ReferenceError: PersonName is not defined
```

Конструктор класса

Ключевое слово `constructor` используется внутри блока определения класса для обозначения определения функции конструктора класса. Использование слова

constructor в качестве имени метода сообщит интерпретатору, что эта конкретная функция должна быть вызвана для создания нового экземпляра с использованием оператора new. Определение конструктора необязательно; определение конструктора как пустой функции аналогично определению класса вообще без конструктора.

Создание экземпляров

Создание экземпляра Person с помощью оператора new работает аналогично использованию new с конструктором функции. Единственное заметное отличие — интерпретатор JavaScript понимает, что если использовать new с именем класса, то для создания экземпляра должна использоваться функция конструктора.

Вызов конструктора класса с использованием new выполнит следующее:

1. Новый объект создается в памяти.
2. Внутренний указатель [[Prototype]] нового объекта назначается в качестве свойства прототипа конструктора.
3. Значение this конструктора присваивается новому объекту (поэтому оно указывает на новый объект при ссылке в конструкторе).
4. Выполняется код внутри конструктора (добавляются свойства к новому объекту).
5. Если функция конструктора возвращает объект, то возвращается этот объект. В противном случае возвращается только что созданный объект.

```
class Animal {}

class Person {
    constructor() {
        console.log('person ctor');
    }
}

class Vegetable {
    constructor() {
        this.color = 'orange';
    }
}

let a = new Animal();

let p = new Person();    // person ctor

let v = new Vegetable();
console.log(v.color);    // orange
```

Параметры, предоставляемые при создании экземпляра класса, используются в качестве параметров функции конструктора. Если использование параметров не требуется, пустые скобки после имени класса необязательны:

```
class Person {
    constructor(name) {
        console.log(arguments.length);
        this.name = name || null;
    }
}

let p1 = new Person;    // 0
console.log(p1.name);   // null

let p2 = new Person();  // 0
console.log(p2.name);   // null

let p3 = new Person('Jake');    // 1
console.log(p3.name);    // Jake
```

По умолчанию конструктор вернет объект `this` после выполнения. Если объект возвращается из функции конструктора, это значение будет использоваться как экземпляр объекта, а вновь созданный объект будет отброшен, если ссылка на него не сохранена. Однако если возвращается другой объект, возвращенный объект не будет связан с классом через `instanceof`, поскольку указатель его прототипа не был изменен.

```
class Person {
    constructor(override) {
        this.foo = 'foo';
        if (override) {
            return {
                bar: 'bar'
            };
        }
    }
}

let p1 = new Person(),
    p2 = new Person(true);

console.log(p1);                        // Person{ foo: 'foo' }
console.log(p1 instanceof Person);    // true

console.log(p2);                        // { bar: 'bar' }
console.log(p2 instanceof Person);    // false
```

Одним из основных отличий от конструкторов функций является то, что использование оператора `new` с конструкторами классов обязательно. При использовании конструкторов функций без оператора `new` конструктор будет использовать глобальное значение `this` — обычно объект `window` — внутри конструктора. В случае с конструкторами классов пренебрежение использованием оператора `new` приведет к ошибке:

```
function Person() {}

class Animal {}

// Создает экземпляр, используя window как "this"
```

```
let p = Person();

let a = Animal();
// TypeError: class constructor Animal cannot be invoked without 'new'
```

Метод конструктора класса не является специальным и после создания экземпляра ведет себя как обычный метод экземпляра (с теми же ограничениями конструктора). Благодаря этому можно ссылаться на него и использовать его после создания:

```
class Person {}

// Создание нового экземпляра через класс
let p1 = new Person();

p1.constructor();
// TypeError: Class constructor Person cannot be invoked without 'new'

// Создание нового экземпляра с использованием ссылки на конструктор класса
let p2 = new p1.constructor();
```

Классы как специальные функции

В спецификации ECMAScript не существует формального типа класса, и во многих отношениях классы ECMAScript ведут себя как специальные функции. После объявления идентификатор класса идентифицируется как функция при проверке с помощью оператора `typeof`:

```
class Person {}

console.log(Person);           // class Person {}
console.log(typeof Person);    // function
```

Идентификатор класса имеет свойство `prototype`, а прототип имеет свойство `constructor`, которое ссылается на сам класс:

```
class Person{}

console.log(Person.prototype);                          // { constructor: f() }
console.log(Person === Person.prototype.constructor);   // true
```

Как и в случае с конструкторами функций, можно использовать оператор `instanceof`, чтобы проверить, присутствует ли прототип конструктора в цепочке прототипов экземпляра:

```
class Person {}

let p = new Person();

console.log(p instanceof Person); // true
```

Вероятно, вы уже догадались, что оператор `instanceof` проверяет цепочку прототипов экземпляра с помощью функции конструктора, которая в этом примере

будет проверять экземпляр `p` с помощью функции конструктора `Person`, которая выглядит как класс.

Как показано ранее, класс ведет себя так же, как и функция конструктора, и в контексте классов сам класс считается конструктором, когда к нему применяется `new`. Важно отметить, что метод конструктора внутри определения класса не считается конструктором и будет возвращать `false` при вызове `instanceof`. Если метод конструктора вызывается напрямую, то это будет то же самое, что и использование неклассового конструктора функции, и соглашение `instanceof` будет обратным:

```
class Person {}

let p1 = new Person();

console.log(p1.constructor === Person);           // true
console.log(p1 instanceof Person);                // true
console.log(p1 instanceof Person.constructor);    // false

let p2 = new Person.constructor();

console.log(p2.constructor === Person);           // false
console.log(p2 instanceof Person);                // false
console.log(p2 instanceof Person.constructor);    // true
```

Классы являются почетными гражданами в JavaScript, что означает, что они могут передаваться как любая другая ссылка на объект или функцию:

```
// Классы могут быть объявлены в любом месте, где может быть объявлена функция —
// например, внутри массива:
let classList = [
    class {
        constructor(id) {
            this.id_ = id;
            console.log('instance ${this.id_}');
        }
    }
];

function createInstance(classDefinition, id) {
    return new classDefinition(id);
}

let foo = createInstance(classList[0], 3141);    // экземпляр 3141
```

Подобно немедленно вызываемому функциональному выражению, класс также может быть создан немедленно:

```
// Поскольку это классовое выражение, имя класса необязательно
let p = new class Foo {
    constructor(x) {
        console.log(x);
    }
}('bar');          // bar

console.log(p);    // Foo {}
```

Члены экземпляра, прототипа и класса

Синтаксис определения класса позволяет аккуратно определять члены, которые должны существовать в экземпляре объекта, прототипе объекта и самом классе.

Члены экземпляра класса

Каждый раз, когда вызывается `new <classname>`, будет выполняться функция конструктора. Внутри этой функции можно заполнить только что созданный экземпляр (объект `this`) «собственными» свойствами. Нет никаких ограничений на добавление свойств в новый экземпляр, и нет ограничений на элементы, добавляемые после выхода из конструктора. Каждому экземпляру присваиваются уникальные объекты-члены, это означает, что ничего не было передано в прототип:

```
class Person {
    constructor() {
        // Для этого примера определяется строка с оберткой объекта
        // для проверки равенства объектов между экземплярами ниже
        this.name = new String('Jack');
        this.sayName = () => console.log(this.name);

        this.nicknames = ['Jake', 'J-Dog']
    }
}

let p1 = new Person(),
    p2 = new Person();

p1.sayName(); // Jack
p2.sayName(); // Jack

console.log(p1.name === p2.name);             // false
console.log(p1.sayName === p2.sayName);       // false
console.log(p1.nicknames === p2.nicknames);   // false

p1.name = p1.nicknames[0];
p2.name = p2.nicknames[1];

p1.sayName();    // Jake
p2.sayName();    // J-Dog
```

Методы и средства доступа к прототипам

Для обеспечения совместного использования методов между экземплярами синтаксис определения класса позволяет определять методы объекта-прототипа внутри тела класса.

```
class Person {
    constructor() {
        // Все, что было добавлено в 'this', будет существовать в каждом экземпляре
        this.locate = () => console.log('instance');
    }

    // Все, что было добавлено в тело класса, определяется
    // в объекте прототипа класса
```

```
    locate() {
        console.log('prototype');
    }
}

let p = new Person();

p.locate(); // instance
Person.prototype.locate(); // prototype
```

Методы могут быть определены в любом месте, но данные члена, такие как примитивы и объекты, не могут быть добавлены к прототипу внутри тела класса:

```
class Person {
    name: 'Jake'
}

// Uncaught SyntaxError: Unexpected token :
```

Методы класса ведут себя идентично свойствам объекта, это означает, что они могут быть связаны со строками, символами или вычисляемыми значениями:

```
const symbolKey = Symbol('symbolKey');

class Person {
    stringKey() {
        console.log('invoked stringKey');
    }
    [symbolKey]() {
        console.log('invoked symbolKey');
    }
    ['computed' + 'Key']() {
        console.log('invoked computedKey');
    }
}

let p = new Person();

p.stringKey();      // invoked stringKey
p[symbolKey]();     // invoked symbolKey
p.computedKey();    // invoked computedKey
```

Определения классов также поддерживают средства доступа к методам чтения и записи свойств. Синтаксис и поведение идентичны синтаксису обычных объектов:

```
class Person {
    set name(newName) {
        this.name_ = newName;
    }

    get name() {
        return this.name_;
    }
}

let p = new Person();
p.name = 'Jake';
console.log(p.name); // Jake
```

Статические методы класса и средства доступа

Также можно определить методы для самого класса. Они предназначены для использования в ситуациях, когда функция выполняет действие, не сосредоточенное на конкретном экземпляре и не требующее обязательного существования данного экземпляра. Как и члены-прототипы, они создаются только один раз для каждого класса.

Статические члены класса обозначаются ключевым словом `static` в качестве префикса внутри определения класса. Внутри статических членов `this` относится к самому классу. Все остальные соглашения идентичны членам-прототипам:

```
class Person {
    constructor() {
        // Все, что было добавлено к 'this', будет существовать
        // на каждом отдельном экземпляре
        this.locate = () => console.log('instance', this);
    }

    // Объявлено в объекте прототипа класса
    locate() {
        console.log('prototype', this);
    }

    // Объявлено в классе
    static locate() {
        console.log('class', this);
    }
}

let p = new Person();

p.locate();                   // instance, Person {}
Person.prototype.locate();    // prototype, {constructor: ... }
Person.locate();              // class, class Person {}
```

Часто вы можете обнаружить, что эти статические методы класса полезны в качестве фабрик экземпляров:

```
class Person {
    constructor(age) {
        this.age_ = age;
    }

    sayAge() {
        console.log(this.age_);
    }

    static create() {
    // Создает и возвращает экземпляр person со случайным age
    return new Person(Math.floor(Math.random()*100));
    }
}

console.log(Person.create()); // Person { age_:... }
```

Нефункциональные члены прототипов и классов

Хотя определение класса не поддерживает явно добавление данных-членов в прототипы или классы, вне определения класса не существует ничего, что мешало бы добавлять их вручную:

```
class Person {
    sayName() {
        console.log('${Person.greeting} ${this.name}');
    }
}

// Определение данных-членов класса
Person.greeting = 'My name is';

// Определение данных-членов прототипа
Person.prototype.name = 'Jake';

let p = new Person();
p.sayName();    // My name is Jake
```

ПРИМЕЧАНИЕ Одна из основных причин, по которой эти операции явно не разрешены, заключается в том, что изменяемые данные-члены в общем объекте могут быть антипаттерном. Как правило, экземпляры объектов должны владеть данными, на которые они ссылаются из `this`.

Методы итератора и генератора

Синтаксис определения класса позволяет определять методы генератора как для прототипа, так и для самого класса:

```
class Person {
    // определение генератора в прототипе
    *createNicknameIterator() {
        yield 'Jack';
        yield 'Jake';
        yield 'J-Dog';
 }

    // определение генератора в классе
    static *createJobIterator() {
        yield 'Butcher';
        yield 'Baker';
        yield 'Candlestick maker';
    }
}

let jobIter = Person.createJobIterator();
console.log(jobIter.next().value);    // Butcher
console.log(jobIter.next().value);    // Baker
console.log(jobIter.next().value);    // Candlestick maker
```

```
let p = new Person();
let nicknameIter = p.createNicknameIterator();
console.log(nicknameIter.next().value);    // Jack
console.log(nicknameIter.next().value);    // Jake
console.log(nicknameIter.next().value);    // J-Dog
```

Поскольку методы генератора поддерживаются, можно сделать экземпляр класса итеративным, добавив итератор по умолчанию:

```
class Person {
    constructor() {
        this.nicknames = ['Jack', 'Jake', 'J-Dog'];
    }

    *[Symbol.iterator]() {
        yield *this.nicknames.entries();
    }
}

let p = new Person();
for (let [idx, nickname] of p) {
    console.log(nickname);
}
// Jack
// Jake
// J-Dog
```

Альтернативный вариант, чтобы просто вернуть экземпляр итератора:

```
class Person {
    constructor() {
        this.nicknames = ['Jack', 'Jake', 'J-Dog'];
    }

    [Symbol.iterator]() {
        return this.nicknames.entries();
    }
}

let p = new Person();
for (let [idx, nickname] of p) {
    console.log(nickname);
}
// Jack
// Jake
// J-Dog
```

Наследование

Ранее в этой главе мы рассмотрели трудоемкие детали реализации наследования с использованием механизмов ES5. Одним из лучших дополнений в спецификации ECMAScript 6 является встроенная поддержка механизма наследования классов. Хотя используется новый синтаксис, наследование классов все еще использует цепочку прототипов под капотом.

Основы наследования

Классы ES6 поддерживают единый формат наследования. Используя ключевое слово `extends`, можно наследовать от всего, что имеет свойство `[[Construct]]` и прототип. По большей части это означает наследование от другого класса, но также обеспечивает обратную совместимость с конструкторами функций:

```
class Vehicle {}

// Наследование от класса
class Bus extends Vehicle {}

let b = new Bus();
console.log(b instanceof Bus);        // true
console.log(b instanceof Vehicle);    // true

function Person() {}

// Наследование от конструктора функции
class Engineer extends Person {}

let e = new Engineer();
console.log(e instanceof Engineer);    // true
console.log(e instanceof Person);      // true
```

Методы класса и прототипа переносятся в производный класс. Значение `this` отражает класс или экземпляр, который вызывает метод:

```
class Vehicle {
    identifyPrototype(id) {
        console.log(id, this);
    }

    static identifyClass(id) {
        console.log(id, this);
    }
}

class Bus extends Vehicle {}

let v = new Vehicle();
let b = new Bus();

b.identifyPrototype('bus');          // bus, Bus {}
v.identifyPrototype('vehicle');      // vehicle, Vehicle {}

Bus.identifyClass('bus');            // bus, class Bus {}
Vehicle.identifyClass('vehicle');    // vehicle, class Vehicle {}
```

ПРИМЕЧАНИЕ Ключевое слово `extends` допустимо в классовых выражениях, поэтому `let Bar = class extends Foo {}` – совершенно правильный синтаксис.

Конструкторы, HomeObjects и super()

Методы производного класса имеют ссылку на свой прототип через ключевое слово `super`. Это доступно только для производных классов и только внутри конструктора или внутри статических методов. `super` используется внутри конструктора для контроля необходимости вызова конструктора родительского класса.

```
class Vehicle {
    constructor() {
        this.hasEngine = true;
    }
}

class Bus extends Vehicle {
    constructor() {
        // Нельзя обращаться к 'this' до super(), иначе будет сгенерирована
        // ReferenceError

        super(); // то же, что и super.constructor()

        console.log(this instanceof Vehicle);    // true
        console.log(this);                       // Bus { hasEngine: true }
    }
}

new Bus();
```

`super` также может использоваться внутри статических методов для вызова статических методов, определенных в унаследованном классе:

```
class Vehicle {
    static identify() {
        console.log('vehicle');
    }
}

class Bus extends Vehicle {
    static identify() {
        super.identify();
    }
}

Bus.identify();    // vehicle
```

> **ПРИМЕЧАНИЕ** ES6 дает конструктору и статическим методам ссылку на внутренний `[[HomeObject]]`, который указывает на объект, для которого определен метод. Этот указатель назначается автоматически и доступен только внутри движка JavaScript. `super` всегда будет определяться как прототип `[[HomeObject]]`.

Пара замечаний по использованию `super`.

- `super` может использоваться только в конструкторе производного класса или статическом методе.

```
class Vehicle {
    constructor() {
        super();
        // SyntaxError: 'super' keyword unexpected
    }
}
```

- На ключевое слово `super` нельзя ссылаться; оно должно быть вызвано как конструктор или использовано для ссылки на статический метод.

```
class Vehicle {}

class Bus extends Vehicle {
    constructor() {
        console.log(super);
        // SyntaxError: 'super' keyword unexpected here
    }
}
```

- Вызов `super()` вызовет конструктор родительского класса и запишет полученный экземпляр в `this`.

```
class Vehicle {}

class Bus extends Vehicle {
    constructor() {
        super();

        console.log(this instanceof Vehicle);
    }
}

new Bus();    // true
```

- `super()` ведет себя как функция конструктора; необходимо вручную передать ему аргументы, чтобы дальше передать их родительскому конструктору.

```
class Vehicle {
    constructor(licensePlate) {
        this.licensePlate = licensePlate;
    }
}

class Bus extends Vehicle {
    constructor(licensePlate) {
        super(licensePlate);
    }
}

console.log(new Bus('1337H4X'));    // Bus { licensePlate: '1337H4X' }
```

- При отказе от определения функции конструктора будет вызываться `super()`, и все аргументы будут переданы конструктору производного класса.

```
class Vehicle {
    constructor(licensePlate) {
```

```
        this.licensePlate = licensePlate;
    }
}

class Bus extends Vehicle {}

console.log(new Bus('1337H4X'));    // Bus { licensePlate: '1337H4X' }
```

- Нельзя ссылаться на `this` внутри конструктора до вызова `super()`.

```
class Vehicle {}

class Bus extends Vehicle {
    constructor() {
        console.log(this);
    }
}

new Bus();
// ReferenceError: Must call super constructor in derived class
// before accessing 'this' or returning from derived constructor
```

- Если класс является производным от родительского класса и вы явно определяете конструктор, нужно либо вызвать `super()`, либо вернуть объект из конструктора.

```
class Vehicle {}

class Car extends Vehicle {}

class Bus extends Vehicle {
    constructor() {
        super();
    }
}

class Van extends Vehicle {
    constructor() {
        return {};
    }
}

console.log(new Car());    // Car {}
console.log(new Bus());    // Bus {}
console.log(new Van());    // {}
```

Абстрактные базовые классы

Может возникнуть необходимость определить класс, который должен быть унаследован, но непосредственно не создан. Хотя и без явной поддержки в ECMAScript, это легко реализовать с помощью `new.target`, который проинформирует вас о том, что использовалось вместе с ключевым словом `new`. Можно предотвратить непосредственное создание экземпляра, проверив, что `new.target` не является абстрактным базовым классом:

```
// Абстрактный базовый класс
class Vehicle {
    constructor() {
        console.log(new.target);
        if (new.target === Vehicle) {
            throw new Error('Vehicle cannot be directly instantiated');
        }
    }
}

// Производный класс
class Bus extends Vehicle {}

new Bus();        // class Bus {}
new Vehicle();    // class Vehicle {}
// Error: Vehicle cannot be directly instantiated
```

Также можно потребовать, чтобы метод был определен в производном классе, проверив его в конструкторе абстрактного базового класса. Поскольку методы-прототипы существуют до вызова конструктора, можно проверить их по ключевому слову `this`:

```
// Абстрактный базовый класс
class Vehicle {
    constructor() {
        if (new.target === Vehicle) {
            throw new Error('Vehicle cannot be directly instantiated');
        }

        if (!this.foo) {
            throw new Error('Inheriting class must define foo()');
        }

        console.log('success!');
    }
}

// Производный класс
class Bus extends Vehicle {
    foo() {}
}

// Производный класс
class Van extends Vehicle {}

new Bus(); // success!
new Van(); // Error: Inheriting class must define foo()
```

Наследование встроенных типов

Классы ES6 обеспечивают плавную совместимость с существующими встроенными ссылочными типами, что позволяет легко расширять их:

```
class SuperArray extends Array {
    shuffle() {
        // Тасование Фишера-Йетса
```

```
        for (let i = this.length - 1; i > 0; i--) {
            const j = Math.floor(Math.random() * (i + 1));
            [this[i], this[j]] = [this[j], this[i]];
        }
    }
}

let a = new SuperArray(1, 2, 3, 4, 5);

console.log(a instanceof Array);         // true
console.log(a instanceof SuperArray);    // true

console.log(a); // [1, 2, 3, 4, 5]
a.shuffle();
console.log(a); // [3, 1, 4, 5, 2]
```

У некоторых встроенных типов есть определенные методы, в которых возвращается новый экземпляр объекта. По умолчанию тип этого экземпляра объекта будет соответствовать типу исходного экземпляра:

```
class SuperArray extends Array {}

let a1 = new SuperArray(1, 2, 3, 4, 5);
let a2 = a1.filter(x => !!(x%2))

console.log(a1);    // [1, 2, 3, 4, 5]
console.log(a2);    // [1, 3, 5]
console.log(a1 instanceof SuperArray);    // true
console.log(a2 instanceof SuperArray);    // true
```

Для переопределения этого поведения можно переопределить метод доступа `Symbol.species`, который вызывается для определения класса, используемого для создания возвращаемого экземпляра:

```
class SuperArray extends Array {
    static get [Symbol.species]() {
        return Array;
    }
}

let a1 = new SuperArray(1, 2, 3, 4, 5);
let a2 = a1.filter(x => !!(x%2))

console.log(a1);    // [1, 2, 3, 4, 5]
console.log(a2);    // [1, 3, 5]
console.log(a1 instanceof SuperArray);    // true
console.log(a2 instanceof SuperArray);    // false
```

Классы-примеси

Часто используемый шаблон в JavaScript — объединение поведения нескольких разных классов в один пакет. Хотя классы ES6 явно не поддерживают наследование от нескольких классов, эта функция предлагает расширяемость, которую можно использовать для эмуляции такого поведения.

ПРИМЕЧАНИЕ Метод `Object.assign()` разработан для предоставления поведения классов-примесей из объектных примесей. Реализация собственных выражений примесей необходима только тогда, когда примеси принимают форму классов. Если вам нужно только объединить свойства между несколькими объектами, предпочтительнее использовать `Object.assign()`.

Ссылка после ключевого слова `extends` является выражением JavaScript. Любой ее синтаксис действителен, пока он разрешает конструктор класса или функции. Выражение вычисляется после вычисления определения класса:

```
class Vehicle {}

function getParentClass() {
    console.log('evaluated expression');
    return Vehicle;
}

class Bus extends getParentClass() {}
// вычисленное выражение
```

Шаблон классов-примесей может быть достигнут путем объединения нескольких элементов примесей внутри выражения, в дальнейшем преобразованных в один класс, который может быть унаследован. Если класс `Person` должен включать в себя примеси A, B и C, в некотором виде вы создадите шаблон, который настраивает B для наследования от A, C для наследования от B и `Person` для наследования от C, тем самым объединяя все три примеси в суперкласс. Есть несколько стратегий для выполнения подобного шаблона.

Одна из стратегий состоит в том, чтобы определить «вложенные» функции, которые принимают суперкласс в качестве параметра, определить класс примеси в качестве подкласса параметра и вернуть этот класс. Эти примеси могут быть связаны друг с другом и предоставлены как выражение суперкласса:

```
class Vehicle {}

let FooMixin = (Superclass) => class extends Superclass {
    foo() {
        console.log('foo');
    }
};
let BarMixin = (Superclass) => class extends Superclass {
    bar() {
        console.log('bar');
    }
};
let BazMixin = (Superclass) => class extends Superclass {
    baz() {
        console.log('baz');
    }
};
```

```
class Bus extends FooMixin(BarMixin(BazMixin(Vehicle))) {}

let b = new Bus();
b.foo();    // foo
b.bar();    // bar
b.baz();    // baz
```

Можно сгладить это вложение, используя вспомогательную функцию:

```
class Vehicle {}

let FooMixin = (Superclass) => class extends Superclass {
    foo() {
        console.log('foo');
    }
};
let BarMixin = (Superclass) => class extends Superclass {
    bar() {
        console.log('bar');
    }
};
let BazMixin = (Superclass) => class extends Superclass {
    baz() {
        console.log('baz');
    }
};

function mix(BaseClass, ...Mixins) {
    return Mixins.reduce((accumulator, current) => current(accumulator),
BaseClass);
}

class Bus extends mix(Vehicle, FooMixin, BarMixin, BazMixin) {}

let b = new Bus();
b.foo();    // foo
b.bar();    // bar
b.baz();    // baz
```

ПРИМЕЧАНИЕ Многие JavaScript-фреймворки, в первую очередь `React`, отходят от паттернов классов-примесей и переходят к компоновке (в форме выделения методов в отдельные классы и вспомогательные классы и включения их по частям без использования наследования). Это отражает хорошо известный программный принцип «композиция поверх наследования», который, по мнению многих, обеспечивает превосходную гибкость и дизайн кода.

ИТОГИ

Объекты можно создавать и расширять в любой момент выполнения кода, что делает их динамичными, а не строго определенными сущностями. Для создания объектов используются паттерны.

- Паттерн Фабрика представляет собой простую функцию, которая создает объект, назначает ему свойства и методы, а затем возвращает его. Этот паттерн вышел из употребления, когда появился паттерн Конструктор.
- Паттерн Конструктор позволяет разработчикам определять собственные ссылочные типы, экземпляры которых можно создавать с помощью оператора `new`, как и экземпляры встроенных типов. К сожалению, никакие члены конструктора, включая функции, не используются повторно, хотя ничто не мешает слабо типизированным ECMAScript-функциям быть общими для нескольких экземпляров объектов.
- В паттерне Прототип эта проблема решается путем назначения общих свойств и методов свойству `prototype` конструктора. В комбинированном паттерне Конструктор+Прототип свойства экземпляра определяются в конструкторе, а общие свойства и методы — в прототипе.

Наследование в JavaScript реализуется преимущественно с помощью цепочек прототипов, для формирования которых свойству `prototype` конструктора подтипа назначается экземпляр другого типа. Как и при наследовании с классами, подтип при этом получает все свойства и методы супертипа. Проблема цепочек прототипов состоит в том, что все унаследованные свойства и методы становятся общими для экземпляров объектов, из-за чего этот паттерн редко используется сам по себе. Паттерн Кража конструктора устраняет эту проблему, вызывая конструктор супертипа в конструкторе подтипа. Это позволяет каждому экземпляру иметь собственные свойства, но вынуждает определять типы только с помощью паттерна Конструктор. Чаще всего применяется комбинированное наследование, в котором общие свойства и методы наследуются с помощью цепочки прототипов, а свойства экземпляра — путем кражи конструктора.

Существуют также альтернативные паттерны наследования.

- Прототипное наследование — это наследование без предопределенных конструкторов, при котором выполняется поверхностное копирование базового объекта. Результат копирования можно расширять.
- С прототипным тесно связано паразитное наследование, которое включает создание объекта на основе другого объекта или некоторой информации, расширение созданного объекта и его возвращение. Этот паттерн можно также использовать с комбинированным наследованием, чтобы не вызывать лишний раз конструктор супертипа.
- Паразитное комбинированное наследование считается наиболее эффективным способом реализации наследования на основе типов.

В ECMAScript 6 были добавлены классы, которые в значительной степени являются синтаксической оболочкой для существующих концепций на основе прототипов. Этот синтаксис предоставляет языку возможность элегантно определять классы, которые являются обратно совместимыми и могут наследоваться от встроенных или пользовательских классов. Классы элегантно устраняют разрыв между экземплярами объектов, прототипами объектов и классами объектов.

9

Прокси и Reflect

- Основы прокси
- Прокси-ловушки и методы Reflect
- Паттерны для прокси

Недавно представленные в ECMAScript 6, прокси и отражение — это совершенно новые конструкции, с помощью которых можно перехватывать и добавлять дополнительное поведение в основные операции в языке. Более конкретно, вы можете определить связанный с целевым объектом прокси-объект, который можно будет использовать в качестве абстрактного целевого объекта: в рамках его вы будете контролировать то, что происходит, когда различные операции выполняются до достижения целевого объекта.

Для разработчиков, впервые подходящих к этой теме, это может показаться довольно туманной концепцией в сочетании со здоровым сводом правил новой терминологии. Работа с несколькими примерами поможет закрепить понимание.

ПРИМЕЧАНИЕ Не существует аналогов для прокси в версиях ECMAScript до ES6. Поскольку это принципиально новая языковая способность, многие трансляторы не могут преобразовать поведение прокси в более ранние версии ECMAScript, поскольку репликация поведения прокси фактически невозможна. Поэтому прокси и отражение полезны только в тех случаях, когда встроенная поддержка предоставляется на 100% платформ. Можно определить поддержку прокси и при необходимости переключиться на резервный код, но это приведет к дублированию кода и поэтому не рекомендуется.

ОСНОВЫ ПРОКСИ

Как упомянуто во введении к главе, прокси ведет себя как абстракция для целевого объекта. Во многих отношениях он аналогичен указателю C ++ в том, что его можно использовать в качестве замены целевого объекта — он указывает на него, но фактически полностью отделен от целевого объекта. Целевым объектом можно манипулировать либо напрямую, либо через прокси, но манипулирование напрямую обойдет поведение, которое разрешает прокси.

ПРИМЕЧАНИЕ Есть ключевые различия между прокси в ECMAScript и указателями C++, которые мы рассмотрим позже, но для ознакомительных целей указатели являются подходящим концептуальным строительным блоком.

Создание сквозного прокси

В своей простейшей форме прокси может существовать лишь в виде абстрагированного целевого объекта. По умолчанию все операции, выполняемые с прокси-объектом, будут прозрачно распространяться на целевой объект. Следовательно, можно использовать прокси-объект теми же способами и в тех местах, в которых будет использоваться целевой объект, с которым связан прокси-объект.

Прокси создается с помощью конструктора `Proxy`. Требуется передать как целевой объект, так и объект-обработчик, при отсутствии которого будет генерироваться ошибка `TypeError`. Для обычного сквозного прокси использование простого литерала объекта для объекта-обработчика позволит всем операциям беспрепятственно достигать целевого объекта.

Как показано здесь, все операции, выполняемые на прокси, будут эффективно применяться к целевому объекту. Единственное ощутимое отличие — это идентичность прокси-объекта.

```
const target = {
    id: 'target'
};

const handler = {};

const proxy = new Proxy(target, handler);

// Свойство 'id' получит доступ к тому же значению
console.log(target.id);    // target
console.log(proxy.id);     // target

// Смена значения свойства для target повлияет на оба объекта, поскольку
// оба они обращаются к одному и тому же значению.
target.id = 'foo';
console.log(target.id);    // foo
console.log(proxy.id);     // foo

// Смена значения свойства для proxy повлияет на оба объекта, поскольку
// это повлияет на объект target.
```

```
proxy.id = 'bar';
console.log(target.id);    // bar
console.log(proxy.id);     // bar

// Метод hasOwnProperty() эффективно применяется
// к объекту target в обоих случаях.
console.log(target.hasOwnProperty('id'));    // false
console.log(proxy.hasOwnProperty('id'));     // false

// Оператор instanceof эффективно применяется
// к объекту target в обоих случаях.
console.log(target instanceof Proxy);    // false
console.log(proxy instanceof Proxy);     // false

// Проверка на строгое равенство объектов все еще может использоваться
// для разграничения proxy и target.
console.log(target === proxy);    // false
```

Определение ловушек

Основная цель прокси — предоставить возможность определять *ловушки*, которые ведут себя как «основные перехватчики операций» внутри объекта-обработчика. Каждый объект-обработчик состоит из нуля, одной или нескольких ловушек, и каждая ловушка соответствует фундаментальной операции, которая может быть прямо или косвенно вызвана в прокси. Когда эти фундаментальные операции вызываются для объекта-посредника, перед тем как вызываться для целевого объекта, прокси вместо этого вызывает функцию прерывания, позволяя разработчикам перехватывать и изменять его поведение.

> **ПРИМЕЧАНИЕ** Термин «ловушка» заимствован из мира операционных систем, где ловушка – это синхронное прерывание в потоке программы, которое отклоняет запуск процессора для выполнения подпрограммы перед возвратом к исходному потоку программы.

Например, можно определить ловушку `get()`, которая запускается каждый раз, когда любая операция ECMAScript выполняет `get()` в той или иной форме. Такая ловушка может быть определена следующим образом:

```
const target = {
    foo: 'bar'
};

const handler = {
    // Ловушки вводятся по имени метода внутри объекта-обработчика
    get() {
        return 'handler override';
    }
};

const proxy = new Proxy(target, handler);
```

Когда для этого прокси-объекта вызывается операция `get()`, вместо этого вызывается функция-ловушка, определенная для `get()`. Конечно, `get()` не используется

для объектов ECMAScript. Перехватываемая операция `get()` распределяется между несколькими операциями, которые можно найти в реальном коде JavaScript. Операции вида `proxy [свойство]`, `proxy.свойство` или `Object.create(proxy) [свойство]` будут использовать фундаментальную операцию `get()` для извлечения свойства, и, следовательно, все они будут вызывать вместо нее функцию `trap` при использовании в прокси. Только прокси будет использовать функции обработчика ловушек; эти операции будут вести себя нормально при использовании с целевым объектом.

```
const target = {
    foo: 'bar'
};

const handler = {
    // Ловушки вводятся по имени метода внутри объекта-обработчика
    get() {
        return 'handler override';
    }
};

const proxy = new Proxy(target, handler);

console.log(target.foo);                        // bar
console.log(proxy.foo);                         // handler override

console.log(target['foo']);                     // bar
console.log(proxy['foo']);                      // handler override

console.log(Object.create(target)['foo']);      // bar
console.log(Object.create(proxy)['foo']);       // handler override
```

Параметры ловушек и Reflect API

Все ловушки имеют доступ к параметрам, которые позволят полностью воссоздать исходное поведение захваченного метода. Например, метод `get()` получает ссылку на целевой объект, просматриваемое свойство и ссылку на прокси-объект.

```
const target = {
    foo: 'bar'
};

const handler = {
    get(trapTarget, property, receiver) {
        console.log(trapTarget === target);
        console.log(property);
        console.log(receiver === proxy);
    }
};

const proxy = new Proxy(target, handler);

proxy.foo;
// true
// foo
// true
```

Таким образом можно определить обработчик прерываний, который полностью воссоздает поведение перехваченного метода:

```
const target = {
    foo: 'bar'
};

const handler = {
    get(trapTarget, property, receiver) {
        return trapTarget[property];
    }
};

const proxy = new Proxy(target, handler);

console.log(proxy.foo);     // bar
console.log(target.foo);    // bar
```

Такая тактика может быть реализована для всех ловушек, но не все поведение ловушек так же просто воссоздать, как `get()`; следовательно, это непрактичная стратегия. Вместо того чтобы вручную реализовывать содержимое захваченного метода, исходное поведение захваченного метода помещается в метод с таким же именем в глобальном объекте `Reflect`.

Каждый метод, который может быть захвачен внутри объекта-обработчика, имеет соответствующий метод из Reflect API. Этот метод имеет идентичное имя и сигнатуру функции и выполняет точное поведение, которое перехватывает захваченный метод. Следовательно, можно определить сквозной прокси, используя только Reflect API:

```
const target = {
    foo: 'bar'
};

const handler = {
    get() {
        return Reflect.get(...arguments);
    }
};

const proxy = new Proxy(target, handler);

console.log(proxy.foo);     // bar
console.log(target.foo);    // bar
```

Альтернативный вариант в более кратком формате:

```
const target = {
    foo: 'bar'
};
const handler = {
    get: Reflect.get
};

const proxy = new Proxy(target, handler);
```

```
console.log(proxy.foo);     // bar
console.log(target.foo);    // bar
```

Если нужно создать настоящий сквозной прокси, который перехватывает все доступные методы и перенаправляет каждый из них в соответствующую функцию API Reflect, определение явного объекта-обработчика не требуется:

```
const target = {
    foo: 'bar'
};

const proxy = new Proxy(target, Reflect);

console.log(proxy.foo);     // bar
console.log(target.foo);    // bar
```

Reflect API позволяет изменять захваченный метод с минимальным стандартным кодом. Например, следующий код декорирует возвращаемое значение при каждом обращении к определенному свойству:

```
const target = {
    foo: 'bar',
    baz: 'qux'
};

const handler = {
    get(trapTarget, property, receiver) {
        let decoration = '';
        if (property === 'foo') {
            decoration = '!!!';
        }

        return Reflect.get(...arguments) + decoration;
    }
};

const proxy = new Proxy(target, handler);

console.log(proxy.foo);     // bar!!!
console.log(target.foo);    // bar

console.log(proxy.baz);     // qux
console.log(target.baz);    // qux
```

Инварианты ловушек

Ловушки предоставляют огромные возможности для изменения поведения почти любого фундаментального метода, но и они не могут существовать без ограничений. Каждый захваченный метод знает контекст целевого объекта и сигнатуру функции прерывания, и поведение функции обработчика прерываний должно подчиняться инвариантам прерывания, как указано в спецификации ECMAScript. Инварианты ловушек варьируются в зависимости от метода, но в целом они не позволяют определению ловушки проявлять какое-либо чрезвычайно неожиданное поведение.

Например, если целевой объект имеет ненастраиваемое и не доступное для записи свойство, будет сгенерирована `TypeError` при попытке вернуть значение из ловушки, отличное от свойства целевого объекта:

```
const target = {};
Object.defineProperty(target, 'foo', {
    configurable: false,
    writable: false,
    value: 'bar'
});

const handler = {
    get() {
        return 'qux';
    }
};

const proxy = new Proxy(target, handler);

console.log(proxy.foo);
// TypeError
```

Отзывные прокси

Может возникнуть необходимость в отключении связи между прокси-объектом и целевым объектом. Для обычного прокси, созданного с помощью `new Proxy()`, эта связь длится все время жизни прокси-объекта.

Прокси также предоставляет метод `revocable()`, представляющий собой дополнительную функцию отзыва, которая может быть вызвана для отсоединения прокси-объекта от целевого объекта. Отмена прокси необратима. Кроме того, функция отзыва является идемпотентной и не будет иметь дальнейшего эффекта, если вызывается несколько раз. Любой метод, вызванный прокси после его отзыва, сгенерирует ошибку `TypeError`.

Функция отзыва может быть перехвачена при создании экземпляра прокси:

```
const target = {
    foo: 'bar'
};

const handler = {
    get() {
        return 'intercepted';
    }
};

const { proxy, revoke } = Proxy.revocable(target, handler);

console.log(proxy.foo);     // intercepted
console.log(target.foo);    // bar

revoke();

console.log(proxy.foo);     // TypeError
```

Использование Reflect API

Есть несколько причин, чтобы отдать предпочтение Reflect API в определенных ситуациях.

Reflect API и Object API

При погружении в Reflect API помните:

1. Reflect API не ограничивается обработчиком ловушек.
2. Большинство методов Reflect API имеют аналог по типу Object.

Как правило, методы Object предназначены для общего использования приложения, а методы Reflect предназначены для точного управления объектами и манипулирования ими.

Флаги состояния

Многие методы Reflect возвращают логическое значение, указывающее, будет ли операция, которую они намерены выполнить, успешной или нет. В определенных ситуациях это более полезно, чем поведение других методов Reflect API, которые либо возвращают измененный объект, либо выдают ошибку (в зависимости от метода). Например, можно использовать Reflect API для выполнения следующего рефакторинга:

```
// Первоначальный вариант кода

const o = {};

try {
    Object.defineProperty(o, 'foo', 'bar');
    console.log('success');
} catch(e) {
    console.log('failure');
}
```

В случае возникновения проблемы с определением нового свойства, `Reflect.defineProperty` вернет `false` вместо выдачи ошибки, что позволит сделать следующее:

```
// Исправленный вариант кода

const o = {};

if(Object.defineProperty(o, 'foo', 'bar')) {
    console.log('success');
} else {
    console.log('failure');
}
```

Следующие методы Reflect предоставляют вам флаги состояния:

- `Reflect.defineProperty`
- `Reflect.preventExtensions`

- `Reflect.setPrototypeOf`
- `Reflect.set`
- `Reflect.deleteProperty`

Замена операторов первоклассными функциями

Некоторые методы Reflect предлагают поведение, которое доступно только через операторы:

- `Reflect.get()` обращается к поведению, которое иначе доступно только путем доступа к свойству объекта.
- `Reflect.set()` обращается к поведению, которое иначе доступно только через оператор присваивания `=`.
- `Reflect.has()` обращается к поведению, которое иначе доступно только через оператор `in` или `with()`.
- `Reflect.deleteProperty()` обращается к поведению, которое иначе доступно только в через оператор `delete`.
- `Reflect.construct()` обращается к поведению, которое иначе доступно только через оператор `new`.

Безопасное использование функций

При вызове функции с использованием метода `apply` существует небольшая вероятность того, что вызываемая функция определяет свое собственное свойство `apply`. Чтобы обойти это, можно отключить метод `apply` от прототипа `Function` следующим образом:

```
Function.prototype.apply.call(myFunc, thisVal, argumentsList);
```

Этой кошмарной строки кода можно избежать и полностью воспроизвести ее с помощью `Reflect.apply`:

```
Reflect.apply(myFunc, thisVal, argumentsList);
```

Замещение прокси

Прокси способны перехватывать операции Reflect API, а это означает, что вполне возможно создать прокси для прокси. Это позволяет создавать несколько слоев замещения поверх единственного целевого объекта:

```
const target = {
    foo: 'bar'
};

const firstProxy = new Proxy(target, {
    get() {
        console.log('first proxy');
        return Reflect.get(...arguments);
```

```
    }
});

const secondProxy = new Proxy(firstProxy, {
    get() {
        console.log('second proxy');
        return Reflect.get(...arguments);
    }
});

console.log(secondProxy.foo);
// second proxy
// first proxy
// bar
```

Нюансы и недостатки прокси

Прокси — это новый API, построенный на существующей инфраструктуре ECMAScript, и поэтому их реализация была лучшим вариантом, что можно было сделать. По большей части прокси очень хорошо работают как слой виртуализации для объектов. Тем не менее прокси не всегда могут легко интегрироваться с существующими конструкциями ECMAScript в определенных сценариях.

'this' внутри прокси

Одним из возможных источников проблем с прокси является ценность `this`. Как и следовало ожидать, значение `this` внутри метода будет соответствовать объекту, для которого он был вызван:

```
const target = {
    thisValEqualsProxy() {
        return this === proxy;
    }
}

const proxy = new Proxy(target, {});

console.log(target.thisValEqualsProxy());    // false
console.log(proxy.thisValEqualsProxy());     // true
```

Интуитивно понятно, что это должно иметь смысл: любой метод, вызываемый для прокси, `proxy.outerMethod()`, который, в свою очередь, вызывает другой метод внутри своего тела функции, `this.innerMethod()`, должен вызывать `proxy.innerMethod()`. В большинстве случаев это, безусловно, ожидаемое поведение; однако если ваша цель полагается на идентичность объекта, вы можете столкнуться с неожиданными проблемами.

Вспомните реализацию закрытой переменной `WeakMap` из главы 6 «Ссылочные типы коллекций», сокращенная версия которой показана здесь:

```
const wm = new WeakMap();

class User {
```

```
    constructor(userId) {
        wm.set(this, userId);
    }

    set id(userId) {
        wm.set(this, userId);
    }

    get id() {
        return wm.get(this);
    }
}
```

Поскольку эта реализация опирается на идентичность объекта экземпляра `User`, она столкнется с проблемами, когда экземпляр `user` будет замещен прокси:

```
const user = new User(123);
console.log(user.id);    // 123

const userInstanceProxy = new Proxy(user, {});
console.log(userInstanceProxy.id);    // undefined
```

Экземпляр `user` первоначально вводится в `WeakMap` с целевым объектом, но прокси пытается получить этот экземпляр с помощью *прокси*-объекта. Решение подобной проблемы — перенастроить прокси таким образом, чтобы начальная вставка ключа выполнялась с экземпляром прокси, что может быть достигнуто путем замещения самого класса `User` и создания экземпляра прокси-класса:

```
const UserClassProxy = new Proxy(User, {});
const proxyUser = new UserClassProxy(456);
console.log(proxyUser.id);
```

Прокси и внутренние слоты

Часто вы можете обнаружить, что экземпляры встроенных ссылочных типов могут беспрепятственно работать вместе с прокси, как в случае с `Array`. Однако некоторые встроенные типы ECMAScript могут полагаться на механизмы, которые прокси не могут контролировать. В результате некоторые методы в обернутом экземпляре будут работать некорректно.

Каноническим примером этого является тип `Date`. Согласно спецификации ECMAScript, типы `Date` полагаются на существование «внутреннего слота» с именем `[[NumberData]]` для значения `this` при выполнении методов. Поскольку внутренний слот не существует в прокси и поскольку к этим внутренним значениям слота нет доступа с помощью обычных операций `get` и `set`, которые прокси мог бы иначе перехватить и перенаправить к цели, вызов метода сгенерирует `TypeError`:

```
const target = new Date();
const proxy = new Proxy(target, {});

console.log(proxy instanceof Date);    // true

proxy.getDate();    // TypeError: 'this' is not a Date object
```

ПРОКСИ-ЛОВУШКИ И МЕТОДЫ REFLECT

Прокси способны улавливать тринадцать различных фундаментальных операций. У каждой из них существует своя запись в Reflect API, параметры, связанные операции ECMAScript и инварианты.

Как было показано ранее, несколько различных операций JavaScript могут вызывать один и тот же обработчик ловушек. Однако для любой отдельной операции, выполняемой над прокси, будет вызываться только один обработчик ловушек; не существует перекрытия ловушек.

Все ловушки также будут перехватывать соответствующие им операции Reflect API, если они вызываются в прокси.

get()

Ловушка `get()` вызывается внутри операций, которые получают значение свойства. Соответствующий метод Reflect API — `Reflect.get()`.

```
const myTarget = {};

const proxy = new Proxy(myTarget, {
    get(target, property, receiver) {
        console.log('get()');
        return Reflect.get(...arguments)
    }
});

proxy.foo;
// get()
```

Возвращаемое значение

Возвращаемое значение не ограничено.

Перехватываемые операции

```
proxy.property

proxy[property]

Object.create(proxy)[property]

Reflect.get(proxy, property, receiver)
```

Параметры обработчика ловушек

target: целевой объект.

property: строковое свойство ключа, на которое ссылается целевой объект.

receiver: прокси-объект или объект, который наследуется от прокси-объекта.

Инварианты ловушек

Если `target.property` недоступен для записи и не настраивается, возвращаемое значение обработчика должно соответствовать `target.property`.

Если `target.property` не настраивается и имеет значение атрибута `[[Get]]` — `undefined`, возвращаемое значение обработчика также должно быть `undefined`.

set()

Ловушка `set()` вызывается внутри операций, которые записывают значение свойства. Соответствующий метод Reflect API — `Reflect.set()`.

```
const myTarget = {};

const proxy = new Proxy(myTarget, {
    set(target, property, value, receiver) {
        console.log('set()');
        return Reflect.set(...arguments)
    }
});

proxy.foo = 'bar';
// set()
```

Возвращаемое значение

Возвращаемое значение `true` указывает на успех операции; возвращаемое значение `false` указывает на ошибку, и в строгом режиме генерируется ошибка `TypeError`.

Перехватываемые операции

```
proxy.property = value

proxy[property] = value

Object.create(proxy)[property] = value

Reflect.set(proxy, property, value, receiver)
```

Параметры обработчика ловушек

target: целевой объект.

property: строковое свойство ключа, на которое ссылается целевой объект.

value: значение, присваиваемое свойству.

receiver: прокси-объект или объект, который наследуется от прокси-объекта.

Инварианты ловушек

Если `target.property` недоступен для записи и не конфигурируется, значение свойства `target` нельзя изменить.

Если `target.property` не настраивается и имеет значение атрибута `[[Set]]` — `undefined`, значение свойства `target` нельзя изменить.

Возвращение `false` из обработчика сгенерирует ошибку `TypeError` в строгом режиме.

has()

Ловушка `has()` вызывается внутри оператора `in`. Соответствующий метод Reflect API — `Reflect.has()`.

```
const myTarget = {};

const proxy = new Proxy(myTarget, {
    has(target, property) {
        console.log('has()');
        return Reflect.has(...arguments)
    }
});

'foo' in proxy;
// has()
```

Возвращаемое значение

`has()` должен возвращать логическое значение, указывающее, представлено свойство или нет. Возвращаемые значения, отличные от логического, будут приведены к логическому типу.

Перехватываемые операции

```
property in proxy

property in Object.create(proxy)

with(proxy) {(property);}

Reflect.has(proxy, property)
```

Параметры обработчика ловушек

target: целевой объект.

property: строковое свойство ключа, на которое ссылается целевой объект.

Инварианты ловушек

Если собственное свойство `target.property` существует и не настраивается, обработчик должен вернуть `true`.

Если существует собственное свойство `target.property` и целевой объект не является расширяемым, обработчик должен вернуть `true`.

defineProperty()

Ловушка `defineProperty()` вызывается внутри `Object.defineProperty()`. Соответствующий метод Reflect API — `Reflect.defineProperty ()`.

```
const myTarget = {};

const proxy = new Proxy(myTarget, {
    defineProperty(target, property, descriptor) {
        console.log('defineProperty()');
        return Reflect.defineProperty(...arguments)
    }
});

Object.defineProperty(proxy, 'foo', { value: 'bar' });
// defineProperty()
```

Возвращаемое значение

`defineProperty()` должен возвращать логическое значение, указывающее, было ли успешно определено свойство. Возвращаемые значения, отличные от логического, будут приведены к логическому типу.

Перехватываемые операции

```
Object.defineProperty(proxy, property, descriptor)

Reflect.defineProperty(proxy, property, descriptor)
```

Параметры обработчика ловушек

target: целевой объект.

property: строковое свойство ключа, на которое ссылается целевой объект.

descriptor: объект, содержащий необязательные определения для `enumerable`, `configurable`, `writable`, `value`, `get` или `set`.

Инварианты ловушек

Если целевой объект не является расширяемым, свойства не могут быть добавлены.

Если целевой объект имеет настраиваемое свойство, ненастраиваемое свойство по тому же ключу не может быть добавлено.

Если целевой объект имеет ненастраиваемое свойство, настраиваемое свойство по тому же ключу не может быть добавлено.

getOwnPropertyDescriptor()

Ловушка `getOwnPropertyDescriptor()` вызывается внутри `Object.getOwnPropertyDescriptor()`. Соответствующий метод Reflect API — `Reflect.getOwnPropertyDescriptor()`.

```
const myTarget = {};

const proxy = new Proxy(myTarget, {
    getOwnPropertyDescriptor(target, property) {
        console.log('getOwnPropertyDescriptor()');
        return Reflect.getOwnPropertyDescriptor(...arguments)
    }
});

Object.getOwnPropertyDescriptor(proxy, 'foo');
// getOwnPropertyDescriptor()
```

Возвращаемое значение

`getOwnPropertyDescriptor()` должен возвращать объект или `undefined`, если свойство не существует.

Перехватываемые операции

```
property in proxy

property in Object.create(proxy)

with(proxy) {(property);}

Reflect.has(proxy, property)
```

Параметры обработчика ловушек

target: целевой объект.

property: строковое свойство ключа, на которое ссылается целевой объект.

Инварианты ловушек

Если существует собственное свойство `target.property` и оно ненастраиваемое, обработчик должен вернуть объект, чтобы указать, что свойство существует.

Если существует собственное свойство `target.property` и оно является настраиваемым, обработчик не может вернуть объект, указывающий, что свойство является настраиваемым.

Если существует собственное свойство `target.property`, а `target` не является расширяемым, обработчик должен вернуть объект, чтобы указать, что свойство существует.

Если `target.property` не существует, а `target` не является расширяемым, обработчик должен вернуть `undefined`, чтобы указать, что свойство не существует.

Если `target.property` не существует, обработчик не может вернуть объект, указывающий, что свойство является настраиваемым.

deleteProperty()

Ловушка `deleteProperty()` вызывается внутри оператора `delete`. Соответствующий метод Reflect API — `Reflect.deleteProperty()`.

```
const myTarget = {};

const proxy = new Proxy(myTarget, {
    deleteProperty(target, property) {
        console.log('deleteProperty()');
        return Reflect.deleteProperty(...arguments)
    }
});

delete proxy.foo
// deleteProperty()
```

Возвращаемое значение

`deleteProperty()` должен возвращать логическое значение, указывающее, было ли успешно удалено свойство. Возвращаемые значения, отличные от логического, будут приведены к логическому типу.

Перехватываемые операции

```
delete proxy.property
```

```
delete proxy[property]
```

```
Reflect.deleteProperty(proxy, property)
```

Параметры обработчика ловушек

target: целевой объект.

property: строковое свойство ключа, на которое ссылается целевой объект.

Инварианты ловушек

Если существует собственное свойство `target.property` и оно ненастраиваемое, обработчик не может удалить свойство.

ownKeys()

Ловушка `ownKeys()` вызывается внутри `Object.keys()` и похожих методов. Соответствующий метод Reflect API — `Reflect.ownKeys()`.

```
const myTarget = {};

const proxy = new Proxy(myTarget, {
    ownKeys(target) {
        console.log('ownKeys()');
```

```
        return Reflect.ownKeys(...arguments)    }
});

Object.keys(proxy);
// ownKeys()
```

Возвращаемое значение

`ownKeys()` должен возвращать объект перечисления, который содержит либо строки, либо символы.

Перехватываемые операции

`Object.getOwnPropertyNames(proxy)`

`Object.getOwnPropertySymbols(proxy)`

`Object.keys(proxy)`

`Reflect.ownKeys(proxy)`

Параметры обработчика ловушек

target: целевой объект.

Инварианты ловушек

Возвращаемый перечисляемый объект должен содержать все ненастраиваемые собственные свойства объекта `target`.

Если `target` не является расширяемым, возвращаемый перечисляемый объект должен в точности содержать собственные ключи свойств `target`.

getPrototypeOf()

Ловушка `getPrototypeOf()` вызывается внутри `Object. getPrototypeOf()`. Соответствующий метод Reflect API — `Reflect.getPrototypeOf()`.

```
const myTarget = {};

const proxy = new Proxy(myTarget, {
    getPrototypeOf(target) {
        console.log('getPrototypeOf()');
        return Reflect.getPrototypeOf(...arguments)
    }
});

Object.getPrototypeOf(proxy);
// getPrototypeOf()
```

Возвращаемое значение

`getPrototypeOf()` должен возвращать объект или `null`.

Перехватываемые операции

`Object.getPrototypeOf(proxy)`

`Reflect.getPrototypeOf(proxy)`

`proxy.__proto__`

`Object.prototype.isPrototypeOf(proxy)`

`proxy instanceof Object`

Параметры обработчика ловушек

target: целевой объект.

Инварианты ловушек

Если `target` не является расширяемым, единственным допустимым возвращаемым значением `Object.getPrototypeOf(proxy)` является значение, возвращаемое из `Object.getPrototypeOf (target)`.

setPrototypeOf()

Ловушка `getPrototypeOf()` вызывается внутри `Object.setPrototypeOf()`. Соответствующий метод Reflect API — `Reflect.setPrototypeOf()`.

```
const myTarget = {};

const proxy = new Proxy(myTarget, {
    setPrototypeOf(target, prototype) {
        console.log('getPrototypeOf()');
        return Reflect.setPrototypeOf(...arguments)
    }
});

Object.setPrototypeOf(proxy, Object);
// setPrototypeOf()
```

Возвращаемое значение

`setPrototypeOf()` должен возвращать логическое значение, указывающее, был ли успешно записан прототип. Возвращаемые значения, отличные от логического, будут приведены к логическому типу.

Перехватываемые операции

`Object.setPrototypeOf(proxy)`

`Reflect.setPrototypeOf(proxy)`

Параметры обработчика ловушек

target: целевой объект.

prototype: предполагаемый прототип замены для `target` или `null`, если это должен быть прототип верхнего уровня.

Инварианты ловушек

Если `target` не является расширяемым, единственным допустимым параметром прототипа является значение, возвращаемое из `Object.getPrototypeOf(target)`.

isExtensible()

Ловушка `isExtensible()` вызывается внутри `Object.isExtensible()`. Соответствующий метод Reflect API — `Reflect.isExtensible()`.

```
const myTarget = {};

const proxy = new Proxy(myTarget, {
    isExtensible(target) {
        console.log('isExtensible()');
        return Reflect.isExtensible(...arguments)
    }
});

Object.isExtensible(proxy);
// isExtensible()
```

Возвращаемое значение

`setPrototypeOf()` должен возвращать логическое значение, указывающее, был ли успешно записан прототип. Возвращаемые значения, отличные от логического, будут приведены к логическому типу.

Перехватываемые операции

`Object.isExtensible(proxy)`

`Reflect.isExtensible(proxy)`

Параметры обработчика ловушек

target: целевой объект.

Инварианты ловушек

Если `target` является расширяемым, обработчик должен вернуть `true`.

Если `target` не является расширяемым, обработчик должен вернуть `false`.

preventExtensions()

Ловушка `preventExtensions()` вызывается внутри `Object.preventExtensions()`. Соответствующий метод Reflect API — `Reflect.preventExtensions()`.

```
const myTarget = {};

const proxy = new Proxy(myTarget, {
    preventExtensions(target) {
        console.log('preventExtensions()');
        return Reflect.preventExtensions(...arguments)
    }
});

Object.preventExtensions(proxy);
// preventExtensions()
```

Возвращаемое значение

`preventExtensions()` должен возвращать логическое значение, указывающее, является ли `target` нерасширяемым. Возвращаемые значения, отличные от логического, будут приведены к логическому типу.

Перехватываемые операции

`Object.preventExtensions(proxy)`

`Reflect.preventExtensions(proxy)`

Параметры обработчика ловушек

target: целевой объект.

Инварианты ловушек

Если `Object.isExtensible(proxy)` имеет значение `false`, обработчик должен вернуть `true`.

apply()

Ловушка `apply()` вызывается при вызовах функции. Соответствующий метод Reflect API — `Reflect.apply()`.

```
const myTarget = () => {};

const proxy = new Proxy(myTarget, {
    apply(target, thisArg, ...argumentsList) {
        console.log('apply()');
        return Reflect.apply(...arguments)
    }
});

proxy();
// apply()
```

Возвращаемое значение

Возвращаемое значение не ограничено.

Перехватываемые операции

`proxy(...argumentsList)`

`Function.prototype.apply(thisArg, argumentsList)`

`Function.prototype.call(thisArg, ...argumentsList)`

`Reflect.apply(target, thisArgument, argumentsList)`

Параметры обработчика ловушек

target: целевой объект.

thisArg: параметр `this` для вызова функции.

argumentsList: список параметров для вызова функции.

Инварианты ловушек

`target` должен быть функциональным объектом.

construct()

Ловушка `construct()` вызывается внутри оператора `new`. Соответствующий метод Reflect API — `Reflect.construct()`.

```
const myTarget = {};

const proxy = new Proxy(myTarget, {
    construct(target, argumentsList, newTarget) {
        console.log('construct()');
        return Reflect.construct(...arguments)
    }
});

new proxy;
// construct()
```

Возвращаемое значение

`construct()` должен возвращать объект.

Перехватываемые операции

`new proxy(...argumentsList)`

`Reflect.construct(target, argumentsList, newTarget)`

Параметры обработчика ловушек

target: целевой конструктор.

argumentsList: список параметров для передачи в целевой конструктор.

newTarget: первоначально вызванный конструктор.

Инварианты ловушек

`target` должен иметь возможность использоваться в качестве конструктора.

ПАТТЕРНЫ ДЛЯ ПРОКСИ

Proxy API позволяет вводить в код невероятно полезные паттерны.

Отслеживание доступа к свойствам

Природа `get`, `set` и `has` дает полное представление об обращениях к свойствам объекта и их проверках. Если вы предоставите захваченный прокси для объекта в приложении, то сможете точно определить, когда и где этот объект доступен:

```
const user = {
    name: 'Jake'
};

const proxy = new Proxy(user, {
    get(target, property, receiver) {
        console.log('Getting ${property}');
        return Reflect.get(...arguments);
    },
    set(target, property, value, receiver) {
        console.log('Setting ${property}=${value}');
        return Reflect.set(...arguments);
    }
});

proxy.name;        // Getting name
proxy.age = 27;    // Setting age=27
```

Скрытие свойств

Внутренности прокси полностью скрыты от удаленного кода, поэтому очень легко скрыть существование свойств целевого объекта. Например:

```
const hiddenProperties = ['foo', 'bar'];
const targetObject = {
    foo: 1,
    bar: 2,
    baz: 3
};
const proxy = new Proxy(targetObject, {
```

```
    get(target, property) {
        if (hiddenProperties.includes(property)) {
            return undefined;
        } else {
            return Reflect.get(...arguments);
        }
    },
    has(target, property) {
        if (hiddenProperties.includes(property)) {
            return false;
        } else {
            return Reflect.has(...arguments);
        }
    }
});

// get()
console.log(proxy.foo);    // undefined
console.log(proxy.bar);    // undefined
console.log(proxy.baz);    // 3

// has()
console.log('foo' in proxy);    // false
console.log('bar' in proxy);    // false
console.log('baz' in proxy);    // true
```

Проверка свойств

Поскольку все назначения должны проходить через ловушку `set()`, можно разрешать или отклонять назначения на основе содержимого предполагаемого значения:

```
const target = {
    onlyNumbersGoHere: 0
};

const proxy = new Proxy(target, {
    set(target, property, value) {
        if (typeof value !== 'Number') {
            return false;
        } else {
            return Reflect.set(...arguments);
        }
    }
});

proxy.onlyNumbersGoHere = 1;
console.log(proxy.onlyNumbersGoHere);    // 1
proxy.onlyNumbersGoHere = '2';
console.log(proxy.onlyNumbersGoHere);    // 1
```

Проверка параметров функции и конструктора

Таким же образом, как могут быть проверены и защищены свойства объекта, могут быть проверены параметры функции и конструктора. Например, функция

может гарантировать, что ей предоставляются только значения определенного типа:

```
function median(...nums) {
    return nums.sort()[Math.floor(nums.length / 2)];
}

const proxy = new Proxy(median, {
    apply(target, thisArg, ...argumentsList) {
        for (const arg of argumentsList) {
            if (typeof arg !== 'number') {
                throw 'Non-number argument provided';
            }
        }
        return Reflect.apply(...arguments);
    }
});

console.log(proxy(4, 7, 1)); // 4
console.log(proxy(4, '7', 1));
// Error: Non-number argument provided
```

Точно так же конструктор может гарантировать наличие параметров:

```
class User {
    constructor(id) {
        this.id_ = id;
    }
}

const proxy = new Proxy(User, {
    construct(target, argumentsList, newTarget) {
        if (argumentsList[0] === undefined) {
            throw 'User cannot be instantiated without id';
        } else {
            return Reflect.construct(...arguments);
        }
    }
});

new proxy(1);

new proxy();
// Error: User cannot be instantiated without id
```

Привязка данных и наблюдатели

Замещение с помощью прокси позволяет переплетать существование различных частей среды выполнения, которые в противном случае были бы несопоставимыми. Это допускает широкий спектр паттернов, которые позволяют различным частям кода взаимодействовать друг с другом.

Например, прокси-класс может быть связан с глобальной коллекцией экземпляров, так что каждый созданный экземпляр добавляется в эту коллекцию:

```
const userList = [];

class User {
    constructor(name) {
        this.name_ = name;
    }
}

const proxy = new Proxy(User, {
    construct() {
        const newUser = Reflect.construct(...arguments);
        userList.push(newUser);
        return newUser;
    }
});

new proxy('John');
new proxy('Jacob');
new proxy('Jingleheimerschmidt');

console.log(userList);    // [User {}, User {}, User{}]
```

Кроме того, коллекция может быть связана с эмиттером, который будет запускаться каждый раз, когда вставляется новый экземпляр:

```
const userList = [];

function emit(newValue) {
    console.log(newValue);
}

const proxy = new Proxy(userList, {
    set(target, property, value, receiver) {
        const result = Reflect.set(...arguments);
        if (result) {
            emit(Reflect.get(target, property, receiver));
        }
        return result;
    }
});

proxy.push('John');
// John
proxy.push('Jacob');
// Jacob
```

ИТОГИ

Прокси являются одним из наиболее интересных и динамичных дополнений в спецификации ECMAScript 6. Хотя у них нет поддержки обратной компиляции, они предоставляют совершенно новую область метапрограммирования и абстракции, которая ранее не была доступна.

На высоком уровне прокси — это прозрачная виртуализация реального объекта JavaScript. Когда прокси создан, можно определить объект-обработчик, содержащий ловушки, являющиеся точками перехвата, с которыми сталкиваются почти все основные операторы и методы JavaScript. Эти обработчики ловушек позволяют изменять работу фундаментальных методов, хотя они связаны *инвариантами ловушек*.

Наряду с прокси существует Reflect API с набором методов, которые одинаково инкапсулируют поведение, перехватываемое каждой ловушкой. Reflect API можно рассматривать как набор фундаментальных операций, которые являются строительными блоками практически всех объектных API JavaScript.

Утилита прокси практически неограниченна и позволяет разработчику использовать новые элегантные паттерны, такие как отслеживание доступа к свойствам, скрытие свойств, предотвращение изменения или удаления свойств, проверка параметров функции и параметров конструктора, привязка данных и наблюдатели (но, конечно, не ограничиваясь ими).

10

Функции

- Функции-выражения, объявления функций и стрелочные функции
- Параметры по умолчанию и оператор распространения
- Рекурсия
- Закрытые переменные и замыкания

Загрузки с сайта wrox.com для этой главы

Обратите внимание, что все примеры кода для этой главы доступны на веб-сайте книги по ссылке www.wrox.com/go/projavascript4e на вкладке Downloads.

Одна из наиболее интересных частей ECMAScript — это функции, прежде всего потому, что функции на самом деле являются объектами. Каждая функция является экземпляром типа `Function`, который имеет свойства и методы, как и любой другой ссылочный тип. Поскольку функции являются объектами, имена функций являются просто указателями на функциональные объекты и необязательно связаны с самой функцией. Функции обычно определяются с использованием синтаксиса объявления функции, как в этом примере:

```
function sum (num1, num2) {
    return num1 + num2;
}
```

В этом коде переменная `sum` определяется и инициализируется как функция. Обратите внимание, что после ключевого слова `function` нет имени, потому что оно не нужно — на функцию может ссылаться переменная `sum`. Также обратите внимание, что после конца определения функции нет точки с запятой.

Синтаксис объявления функции практически в точности эквивалентен использованию выражения функции, такого как это:

```
let sum = function(num1, num2) {
    return num1 + num2;
};
```

Обратите внимание, что после функции стоит точка с запятой, как и после любой инициализации переменной.

Другой способ определить функцию, очень похожий на выражение функции, — использование «стрелочного» синтаксиса:

```
let sum = (num1, num2) => {
    return num1 + num2;
};
```

Последний способ определения функции — использовать конструктор `Function`, который принимает любое количество аргументов. Последний аргумент всегда считается телом функции, а предыдущие аргументы перечисляют аргументы новой функции. Рассмотрим этот пример:

```
let sum = new Function("num1", "num2", "return num1 + num2");  // не рекомендуется
```

Этот синтаксис не рекомендуется использовать, поскольку он вызывает двойную интерпретацию кода (один раз для обычного кода ECMAScript и один раз для строк, передаваемых в конструктор) и, таким образом, может влиять на производительность. Однако важно воспринимать функции как объекты, а имена функций как указатели — данный синтаксис отлично подходит для представления этой концепции.

> **ПРИМЕЧАНИЕ** Между этими различными способами создания экземпляра объекта функции есть тонкие, но важные различия, которые будут обсуждаться позже в этой главе. Тем не менее все они вызываются одинаково.

СТРЕЛОЧНЫЕ ФУНКЦИИ

В ECMAScript 6 появилась возможность определять выражение функции с использованием стрелочного синтаксиса. В большинстве случаев стрелочные функции создают экземпляры объектов функций, которые ведут себя так же, как и их аналоги из формальных функций. Везде, где можно использовать выражение функции, можно также использовать стрелочную функцию:

```
let arrowSum = (a, b) => {
    return a + b;
};

let functionExpressionSum = function(a, b) {
```

```
    return a + b;
};

console.log(arrowSum(5, 8));                 // 13
console.log(functionExpressionSum(5, 8));    // 13
```

Стрелочные функции исключительно полезны в однострочных примерах, где они предлагают более сокращенный синтаксис:

```
let ints = [1, 2, 3];

console.log(ints.map(function(i) { return i + 1; }));    // [2, 3, 4]
console.log(ints.map((i) => { return i + 1 }));          // [2, 3, 4]
```

Функции стрелок не требуют скобок, если используется только один параметр. Если параметры вообще не нужны или их больше одного, требуются круглые скобки:

```
// Оба примера допустимы
let double = (x) => { return 2 * x; };
let triple = x => { return 3 * x; };

// При отсутствии параметров указываются пустые скобки
let getRandom = () => { return Math.random(); };

// Для нескольких параметров требуются скобки
let sum = (a, b) => { return a + b; };

// Неправильный синтаксис:
let multiply = a, b => { return a * b; };
```

Стрелочные функции также не требуют фигурных скобок, но неиспользование их меняет поведение функции. Использование фигурных скобок называется синтаксисом «блочного тела» и ведет себя так же, как нормальное выражение функции, так как внутри стрелочной функции может быть записано несколько строк кода, как и для нормального выражения функции. Если вы опускаете фигурные скобки, используется так называемый синтаксис «сжатого тела», ограниченный одной строкой кода, такой как присваивание или выражение. Значение этой строки будет возвращаться неявно, как показано здесь:

```
// Оба примера правильны и вернут значение
let double = (x) => { return 2 * x; };
let triple = (x) => 3 * x;

// Присваивание допускается
let value = {};
let setName = (x) => x.name = "Matt";
setName(value);
console.log(value.name);    // "Matt"

// Неправильный синтаксис:
let multiply = (a, b) => return a * b;
```

Стрелочные функции, хотя и синтаксически лаконичны, не подходят в нескольких ситуациях. Они не позволяют использовать аргументы `super` или `new.target` и не могут использоваться в качестве конструктора. Кроме того, объекты функции, созданные с использованием стрелочного синтаксиса, не имеют определенного прототипа.

ИМЕНА ФУНКЦИЙ

Поскольку имена функций являются просто указателями на функции, они действуют как любая другая переменная, содержащая указатель на объект. Это означает, что одна функция может вызываться под разными именами, как в этом примере:

```
function sum(num1, num2) {
 return num1 + num2;
}
console.log(sum(10, 10));    // 20

let anotherSum = sum;
console.log(anotherSum(10, 10));    // 20

sum = null;
console.log(anotherSum(10, 10));    // 20
```

Этот код определяет функцию с именем `sum()`, которая складывает два числа вместе. Переменная `anotherSum` объявляется и устанавливается равной `sum`. Обратите внимание, что использование имени функции без скобок позволяет получить доступ к указателю на функцию вместо ее выполнения. На этом этапе и `anotherSum`, и `sum` указывают на одну и ту же функцию, а это означает, что может быть вызван `anotherSum()` и возвращен результат. Когда для `sum` задано значение `null`, это разрывает связь с функцией, хотя `anotherSum()` все равно может быть вызвана без проблем.

Все объекты функций в ECMAScript 6 предоставляют свойство имени, доступное только для чтения, которое описывает функцию. Во многих случаях это будет просто идентификатор функции или строковое имя переменной, которая ссылается на функцию. Если функция не названа, об этом будет сообщено. Если функция создана с использованием конструктора, она будет идентифицирована как «анонимная»:

```
function foo() {}
let bar = function() {};
let baz = () => {};
console.log(foo.name);    // foo
console.log(bar.name);    // bar
console.log(baz.name);    // baz
console.log((() => {}).name);    // (пустая строка)
console.log((new Function()).name);    // anonymous
```

Если функция является методом чтения или записи свойств или создается с использованием `bind()`, то перед ее именем будет добавлен префикс для идентификации:

```
function foo() {}

console.log(foo.bind(null).name);    // bound foo

let dog = {
    years: 1,
    get age() {
        return this.years;
    },
    set age(newAge) {
        this.years = newAge;
    }
}

let propertyDescriptor = Object.getOwnPropertyDescriptor(dog, 'age');
console.log(propertyDescriptor.get.name);    // get age
console.log(propertyDescriptor.set.name);    // set age
```

АРГУМЕНТЫ ФУНКЦИЙ

Аргументы функций в ECMAScript не ведут себя так же, как аргументы функций в большинстве других языков. Функция ECMAScript не заботится о том, сколько аргументов передается, и не заботится о типах данных этих аргументов. То, что вы определяете функцию для приема двух аргументов, не означает, что можно передать в нее только два аргумента. Передайте один, три или вообще ни одного — интерпретатор не будет жаловаться.

Это безразличие происходит потому, что технически аргументы в ECMAScript представляются в виде массива. Массив всегда передается функции, но функция не заботится о том, что именно (если вообще что-то) находится в массиве. Если массив прибывает без элементов — это нормально; если он прибывает с большим количеством элементов — это тоже нормально. Фактически, когда функция определяется с помощью ключевого слова `function` (означающего не стрелочную функцию), на самом деле существует объект `arguments`, к которому можно обращаться, находясь внутри функции, чтобы получить значения каждого переданного аргумента.

Объект `arguments` действует как массив (хотя он не является экземпляром `Array`) в том смысле, что можно получить доступ к каждому аргументу, используя запись со скобками (первый аргумент — `arguments[0]`, второй — `arguments[1]` и т. д.), и определить, сколько аргументов было передано с помощью свойства `length`.

В следующем примере первый аргумент функции `sayHi()` называется `name`.

```
function sayHi(name, message) {
    console.log("Hello " + name + ", " + message);
}
```

К тому же значению можно обратиться, ссылаясь на `arguments[0]`. Следовательно, функцию можно переписать без явного указания аргументов, например:

```
function sayHi() {
    console.log("Hello " + arguments[0] + ", " + arguments[1]);
}
```

В этой переписанной версии нет именованных аргументов. Аргументы `name` и `message` были удалены, но функция будет вести себя соответствующим образом. Это иллюстрирует важный момент о функциях в ECMAScript: именованные аргументы — это удобство, а не необходимость. В отличие от других языков, присвоение имен аргументам в ECMAScript не создает сигнатуру функции, которая должна соответствовать ей позже; нет проверки в отношении именованных аргументов.

Объект `arguments` также можно использовать для проверки количества аргументов, переданных в функцию с помощью свойства `length`. В следующем примере выводится число аргументов, передаваемых в функцию при каждом ее вызове:

```
function howManyArgs() {
    console.log(arguments.length);
}

howManyArgs("string", 45);    // 2
howManyArgs();                // 0
howManyArgs(12);              // 1
```

В этом примере показаны предупреждения, отображающие 2, 0 и 1 (в указанном порядке). Таким образом, разработчики могут позволить функциям принимать любое количество аргументов и вести себя соответствующим образом. Нужно учитывать следующее:

```
function doAdd() {
    if (arguments.length === 1) {
        console.log(arguments[0] + 10);
    } else if (arguments.length === 2) {
        console.log(arguments[0] + arguments[1]);
    }
}

doAdd(10);        // 20
doAdd(30, 20);    // 50
```

Функция `doAdd()` добавляет 10 к числу, только если есть один аргумент; если есть два аргумента, они просто складываются и возвращаются. Таким образом, `doAdd(10)` возвращает 20, тогда как `doAdd(30,20)` возвращает 50. Такой подход не лучше перегрузки, но он может служить обходным путем для этого ограничения ECMAScript.

Еще одна важная вещь для понимания аргументов — это то, что объект `arguments` может использоваться в сочетании с именованными аргументами, такими как:

```
function doAdd(num1, num2) {
    if (arguments.length === 1) {
        console.log(num1 + 10);
    } else if (arguments.length === 2) {
        console.log(arguments[0] + num2);
    }
}
```

В этой перезаписи функции doAdd() два именованных аргумента используются вместе с объектом arguments. Именованный аргумент num1 содержит то же значение, что и arguments[0], поэтому их можно использовать взаимозаменяемо (то же самое верно для num2 и arguments [1]).

Другое интересное поведение arguments состоит в том, что его значения всегда синхронизируются со значениями соответствующих именованных параметров. Например:

```
function doAdd(num1, num2) {
    arguments[1] = 10;
    console.log(arguments[0] + num2);
}
```

Эта версия doAdd() всегда перезаписывает второй аргумент значением 10. Поскольку значения в объекте arguments автоматически отражаются соответствующими именованными аргументами, изменение arguments[1] также меняет значение num2, поэтому оба имеют значение 10. Однако это не означает, что оба обращаются к одному и тому же пространству памяти; их области памяти отделены друг от друга, но иногда синхронизируются. Этот эффект работает только в одну сторону: изменение именованного аргумента *не* приводит к изменению соответствующего значения в arguments. Еще одна вещь, которую нужно иметь в виду: если передается только один аргумент, то установка arguments[1] в значение не будет отражена именованным аргументом, поскольку длина объекта arguments устанавливается на основе количества переданных аргументов, а не именованных аргументов, перечисленных в функции.

Любой именованный аргумент, который не передается в функцию, автоматически получает значение undefined. Это похоже на определение переменной без ее инициализации. Например, если в функцию doAdd() передается только один аргумент, num2 имеет значение undefined.

Строгий режим вносит несколько изменений в способ использования объекта arguments. Во-первых, назначение, как в предыдущем примере, не будет работать. Значение num2 остается неопределенным, даже если в arguments[1] записано 10. Во-вторых, попытка перезаписать значение arguments является синтаксической ошибкой (код не выполнится).

Аргументы в стрелочных функциях

Когда функция определена с помощью стрелочного обозначения, аргументы, переданные функции, не могут быть доступны с помощью ключевого слова arguments; доступ к ним возможен только с использованием их именованного токена в определении функции.

```
function foo() {
    console.log(arguments[0]);
}
foo(5);    // 5

let bar = () => {
```

```
    console.log(arguments[0]);
};
bar(5);    // ReferenceError: arguments is not defined
```

Хотя аргументы стрелочной функции могут быть недоступны, имейте в виду — возможно, что ключевое слово `arguments` предоставляется области действия стрелочной функции из области действия вызываемой функции-оболочки:

```
function foo() {
    let bar = () => {
        console.log(arguments[0]);    // 5
    };
    bar();
}

foo(5);
```

ПРИМЕЧАНИЕ Все аргументы в ECMAScript передаются по значению. Невозможно передать аргументы по ссылке. Если объект передается в качестве аргумента, значение является просто ссылкой на объект.

ОТСУТСТВИЕ ПЕРЕГРУЗКИ

Функции ECMAScript не могут быть перегружены в традиционном смысле. В других языках, таких как Java, можно написать два определения функции, если их сигнатуры (тип и количество принятых аргументов) различны. Как только что обсуждалось, функции в ECMAScript не имеют сигнатур, поскольку аргументы представлены в виде массива, содержащего ноль или более значений. Без сигнатур функций настоящая перегрузка невозможна.

Если две функции определены с одинаковым именем в ECMAScript, последняя функция становится владельцем этого имени. Рассмотрим следующий пример:

```
function addSomeNumber(num) {
    return num + 100;
}

function addSomeNumber(num) {
    return num + 200;
}

let result = addSomeNumber(100);    // 300
```

Здесь функция `addSomeNumber()` определяется дважды. Первая версия функции добавляет 100 к аргументу, а вторая добавляет 200. Когда вызывается последняя строка, она возвращает 300, потому что вторая функция перезаписала первую.

Как упоминалось ранее, можно смоделировать перегрузку методов, проверяя тип и количество аргументов, которые были переданы в функцию, и затем реагируя соответствующим образом.

Представление имен функций как указателей также объясняет, почему в ECMAScript не может быть перегрузки функций. Из предыдущего примера ясно, что объявление

двух функций с одинаковым именем всегда приводит к тому, что последняя функция перезаписывает предыдущую. Код выше почти эквивалентен следующему:

```
let addSomeNumber = function(num) {
    return num + 100;
};

addSomeNumber = function(num) {
    return num + 200;
};

let result = addSomeNumber(100);    // 300
```

В этом переписанном коде гораздо проще увидеть, что именно происходит. Переменная `addSomeNumber` просто перезаписывается при создании второй функции.

ЗНАЧЕНИЯ ПАРАМЕТРОВ ПО УМОЛЧАНИЮ

В ECMAScript 5.1 и более ранних версиях общей стратегией для реализации значений параметров по умолчанию было следующее: путем проверки неопределенности параметра сначала определялось, не был ли он предоставлен для вызова функции. Если параметр действительно не был определен прежде, ему присваивалось значение:

```
function makeKing(name) {
    name = (typeof name !== 'undefined') ? name : 'Henry';
    return `King ${name} VIII`;
}

console.log(makeKing());           // 'King Henry VIII'
console.log(makeKing('Louis'));    // 'King Louis VIII
```

В ECMAScript 6 необходимость в этой стратегии отпала, так как он поддерживает явное определение значений для параметров, если они не предоставлены при вызове функции. Эквивалент предыдущей функции с параметрами по умолчанию ES6 выполняется с помощью оператора `=` непосредственно внутри сигнатуры функции:

```
function makeKing(name = 'Henry') {
    return `King ${name} VIII`;
}

console.log(makeKing('Louis'));    // 'King Louis VIII'
console.log(makeKing());           // 'King Henry VIII'
```

Передача `undefined` в качестве аргумента обрабатывается так же, как и передача любого другого аргумента, что позволяет использовать несколько независимых переменных по умолчанию:

```
function makeKing(name = 'Henry', numerals = 'VIII') {
    return `King ${name} ${numerals}`;
}

console.log(makeKing());                    // 'King Henry VIII'
console.log(makeKing('Louis'));             // 'King Louis VIII'
console.log(makeKing(undefined, 'VI'));     // 'King Henry VI'
```

При использовании параметров по умолчанию значение объекта `arguments` отражает не значение параметра по умолчанию, а скорее аргумент, передаваемый функции. Это отражает поведение в строгом режиме ES5 и является ценным, поскольку сохраняет значения такими, какими они были переданы при вызове функции:

```
function makeKing(name = 'Henry') {
    name = 'Louis';
    return `King ${arguments[0]}`;
}

console.log(makeKing());           // 'King undefined'
console.log(makeKing('Louis'));    // 'King Louis'
```

Значения параметров по умолчанию не ограничиваются примитивами или типами объектов, так как дополнительно можно вычислить значение из вызванной функции:

```
let romanNumerals = ['I', 'II', 'III', 'IV', 'V', 'VI'];
let ordinality = 0;

function getNumerals() {
    // Увеличение порядкового номера после использования для его индексации
    // в массив чисел
    return romanNumerals[ordinality++];
}
function makeKing(name = 'Henry', numerals = getNumerals()) {
    return `King ${name} ${numerals}`;
}

console.log(makeKing());                  // 'King Henry I'
console.log(makeKing('Louis', 'XVI'));    // 'King Louis XVI'
console.log(makeKing());                  // 'King Henry II'
console.log(makeKing());                  // 'King Henry III'
```

Параметр функции по умолчанию вызывается только тогда, когда вызывается сама функция, а именно сразу после определения функции. Обратите внимание, что метод, который вычисляет значение по умолчанию, вызывается только в случае, когда аргумент не указан.

Стрелочные функции также могут использовать параметры по умолчанию этим же образом, хотя это означает, что круглые скобки вокруг одного аргумента больше не являются обязательными, если указано значение по умолчанию:

```
let makeKing = (name = 'Henry') => `King ${name}`;

console.log(makeKing());    // King Henry
```

Область параметров по умолчанию и временная мертвая зона

Похоже, что из-за возможности определения объектов и вызова функций «на лету» при оценке значений параметров по умолчанию для параметров функции существует некоторый объем выполнения — и это действительно так.

Определение нескольких параметров со значениями по умолчанию работает практически так же, как последовательное объявление переменных с помощью ключевого слова `let`. Рассмотрим следующую функцию:

```
function makeKing(name = 'Henry', numerals = 'VIII') {
    return `King ${name} ${numerals}`;
}

console.log(makeKing());    // King Henry VIII
```

Значения параметров по умолчанию инициализируются в том порядке, в котором они перечислены в списке параметров. Можно представить это как поведение, похожее на следующее:

```
function makeKing() {
    let name = 'Henry';
    let numerals = 'VIII';

    return `King ${name} ${numerals}`;
}
```

Поскольку параметры инициализируются по порядку, параметры, значения которых по умолчанию определены позже, могут ссылаться на более ранний параметр. Следующий глупый пример делает именно это:

```
function makeKing(name = 'Henry', numerals = name) {
    return `King ${name} ${numerals}`;
}

console.log(makeKing());    // King Henry Henry
```

Порядок инициализации параметров следует тем же правилам временной мертвой зоны, определяя, что значения параметров не могут ссылаться на другие значения параметров, которые будут определены позже. Это сгенерирует ошибку:

```
// Ошибка
function makeKing(name = numerals, numerals = 'VIII') {
    return `King ${name} ${numerals}`;
}
```

Параметры также существуют внутри их собственной области видимости и поэтому не могут ссылаться на область видимости тела функции. При попытке сделать это сгенерируется ошибка:

```
// Ошибка
function makeKing(name = 'Henry', numerals = defaultNumeral) {
    let defaultNumeral = 'VIII';
    return `King ${name} ${numerals}`;
}
```

АРГУМЕНТЫ РАСПРОСТРАНЕНИЯ И ОСТАТОЧНЫЕ ПАРАМЕТРЫ

В ECMAScript 6 представлен оператор распространения, который позволяет очень элегантно управлять коллекциями и группировать их. Одно из наиболее полезных приложений — в области сигнатур функций, где оно особенно ярко проявляется в области слабой типизации и аргументов переменной длины. Оператор распространения полезен как при вызове функции, так и при определении параметров функции.

Аргументы распространения

Вместо передачи массива в качестве одного аргумента функции часто полезно иметь возможность разбивать массив значений и отдельно передавать каждое значение как отдельный аргумент.

Предположим, определена следующая функция, которая суммирует все значения, переданные в качестве аргументов:

```
let values = [1, 2, 3, 4];

function getSum() {
    let sum = 0;
    for (let i = 0; i < arguments.length; ++i) {
        sum += arguments[i];
    }
    return sum;
}
```

Эта функция ожидает, что каждый из ее аргументов будет индивидуальным числом, по которым будет проводиться итерация для нахождения суммы. Массив вне функции, содержащий все значения, которые нужно суммировать, является логическим форматом, но самый разумный способ скомпилировать этот массив в отдельные параметры — это неэффективно использовать `.apply()`:

```
console.log(getSum.apply(null, values));    // 10
```

В ECMAScript 6 можно выполнять это действие более кратко, используя оператор распространения. Применение оператора распространения к итерируемому объекту и передача его в качестве единственного аргумента функции будет разбивать этот итерируемый объект размера N и передавать его функции как N отдельных аргументов.

С помощью оператора распространения можно распаковать внешний массив в отдельные аргументы непосредственно внутри вызова функции:

```
console.log(getSum(...values));    // 10
```

Поскольку размер массива известен, нет никаких ограничений на другие параметры, появляющиеся до или после оператора распространения, включая другие операторы распространения:

```
console.log(getSum(-1, ...values));            // 9
console.log(getSum(...values, 5));             // 15
console.log(getSum(-1, ...values, 5));         // 14
console.log(getSum(...values, ...[5,6,7]));    // 28
```

Наличие оператора распространения совершенно неизвестно объекту `arguments`; он будет обрабатывать значение, разбитое на части, как отдельные части, только потому, что так они передаются в функцию:

```
let values = [1,2,3,4]

function countArguments() {
    console.log(arguments.length);
}

countArguments(-1, ...values);            // 5
countArguments(...values, 5);             // 5
countArguments(-1, ...values, 5);         // 6
countArguments(...values, ...[5,6,7]);    // 7
```

Объект `arguments` является лишь одним из способов использования аргументов распространения. Аргументы распространения могут использоваться как именованные параметры как в стандартных функциях, так и в стрелочных, а также в качестве аргументов по умолчанию:

```
function getProduct(a, b, c = 1) {
    return a * b * c;
}

let getSum = (a, b, c = 0) => {
    return a + b + c;
}

console.log(getProduct(...[1,2]));        // 2
console.log(getProduct(...[1,2,3]));      // 6
console.log(getProduct(...[1,2,3,4]));    // 6

console.log(getSum(...[0,1]));            // 1
console.log(getSum(...[0,1,2]));          // 3
console.log(getSum(...[0,1,2,3]));        // 3
```

Остаточные параметры

При составлении определения функции вместо индивидуальной обработки параметров можно использовать оператор распространения, чтобы объединить диапазоны параметров переменной длины в один массив. Во многих отношениях это очень похоже на работу объекта `arguments`, но в этом случае остаточный параметр становится формальным объектом `Array`.

```
function getSum(...values) {
    // Последовательно складывает все элементы из 'values'
    // Начальная общая сумма = 0
    return values.reduce((x, y) => x + y, 0);
}

console.log(getSum(1,2,3));    // 6
```

Если перед остаточным параметром есть именованные параметры, он примет размер оставшихся безымянных параметров или пустой массив, если таковых нет. Поскольку остаточный параметр имеет переменный размер, можно использовать его только как последний формальный параметр:

```
// Ошибка
function getProduct(...values, lastValue) {}

// Все в порядке
function ignoreF    irst(firstValue, ...values) {
    console.log(values);
}

ignoreFirst();          // []
ignoreFirst(1);         // []
ignoreFirst(1,2);       // [2]
ignoreFirst(1,2,3);     // [2, 3]
```

Хотя стрелочные функции не поддерживают объект `arguments`, они поддерживают остаточные параметры, что предоставляет примерно такое же поведение:

```
let getSum = (...values) => {
    return values.reduce((x, y) => x + y, 0);
}

console.log(getSum(1,2,3));    // 6
```

Как и следовало ожидать, использование остаточного параметра не влияет на объект `arguments` — он все равно будет точно отражать то, что было передано функции:

```
function getSum(...values) {
    console.log(arguments.length);    // 3
    console.log(arguments);           // [1, 2, 3]
    console.log(values);              // [1, 2, 3]
}

console.log(getSum(1,2,3));
```

ОБЪЯВЛЕНИЯ ФУНКЦИИ И ФУНКЦИИ-ВЫРАЖЕНИЯ

В этом разделе объявление функции и функция-выражение упоминаются как почти эквивалентные. Это уклонение связано с одним существенным различием в том, как механизм JavaScript загружает данные в контекст выполнения. Объявления функций читаются и доступны в контексте выполнения перед выполнением любого кода, тогда как функции-выражения не завершаются до тех пор, пока выполнение не достигнет этой строки кода. Нужно учитывать следующее:

```
// Все в порядке
console.log(sum(10, 10));
function sum(num1, num2) {
    return num1 + num2;
}
```

Этот код отлично работает, потому что объявления функций читаются и добавляются в контекст выполнения до того, как код начинает выполняться через процесс, называемый *подъемом объявления функции* (function declaration hoisting). Когда код вычисляется, механизм JavaScript делает первый проход для объявлений функций и вытягивает их в верхнюю часть дерева исходного кода. Таким образом, даже несмотря на то что объявление функции появляется после его использования в коде, движок изменяет его, чтобы поднять объявления функции наверх. Замена объявления функции на эквивалентную функцию-выражение, как в следующем примере, приведет к ошибке во время выполнения:

```
// Ошибка
console.log(sum(10, 10));
let sum = function(num1, num2) {
    return num1 + num2;
};
```

Этот обновленный код сгенерирует ошибку, потому что функция является частью выражения инициализации, а не частью объявления функции. Это означает, что функция недоступна в переменной `sum` до тех пор, пока выделенная строка не будет выполнена, чего не произойдет, поскольку первая строка вызывает ошибку «неожиданный идентификатор». Это не является следствием использования `let`, так как использование ключевого слова `var` приведет к той же проблеме:

```
console.log(sum(10, 10));
var sum = function(num1, num2) {
    return num1 + num2;
};
```

Эти два синтаксиса эквивалентны помимо разницы в том, когда именно функция доступна по заданному имени.

> **ПРИМЕЧАНИЕ** Можно использовать именованные выражения функций, которые выглядят как объявления, например `let sum = function sum() {}`. Это обсуждается в разделе «Функции-выражения» этой главы.

ФУНКЦИИ КАК ЗНАЧЕНИЯ

Поскольку имена функций в ECMAScript — не более чем переменные, функции могут использоваться везде, где могут использоваться переменные. Это означает, что можно не только передать функцию в другую функцию в качестве аргумента, но и вернуть ее как результат другой функции. Рассмотрим следующую функцию:

```
function callSomeFunction(someFunction, someArgument) {
    return someFunction(someArgument);
}
```

Эта функция принимает два аргумента. Первый аргумент должен быть функцией, а второй аргумент — значением, передаваемым этой функции. Любая функция может быть передана следующим образом:

```
function add10(num) {
    return num + 10;
}

let result1 = callSomeFunction(add10, 10);
console.log(result1);    // 20

function getGreeting(name) {
    return "Hello, " + name;
}

let result2 = callSomeFunction(getGreeting, "Nicholas");
console.log(result2);    // "Hello, Nicholas"
```

Функция `callSomeFunction()` является общей, поэтому не имеет значения, в какую функцию передается первый аргумент — результат всегда будет возвращаться из первого исполняемого аргумента. Помните, что для доступа к указателю на функцию вместо ее выполнения нужно поставить скобки, поэтому в `callSomeFunction()` передаются переменные `add10` и `getGreeting`, а не их результаты.

Возврат функции из функции также возможен и может быть весьма полезным. Например, предположим, что у вас есть массив объектов и нужно отсортировать массив по произвольному свойству объекта. Функция сравнения для метода `sort()` массива принимает только два аргумента, которые являются значениями для сравнения, но на самом деле вам нужен способ указать, по какому именно свойству проводить сортировку. Эту проблему можно решить, определив функцию для создания функции сравнения на основе имени свойства, как в следующем примере:

```
function createComparisonFunction(propertyName) {
    return function(object1, object2) {
        let value1 = object1[propertyName];
        let value2 = object2[propertyName];

        if (value1 < value2) {
            return -1;
        } else if (value1 > value2) {
            return 1;
        } else {
            return 0;
        }
    };
}
```

Синтаксис этой функции может показаться сложным, но это всего лишь функция внутри функции, которой предшествует оператор `return`. Аргумент `propertyName` доступен из внутренней функции и используется с нотацией в скобках для получения значения заданного свойства. После получения значений свойств можно выполнить простое сравнение. Эта функция может использоваться как в следующем примере:

```
let data = [
    {name: "Zachary", age: 28},
    {name: "Nicholas", age: 29}
];

data.sort(createComparisonFunction("name"));
console.log(data[0].name);    // Nicholas

data.sort(createComparisonFunction("age"));
console.log(data[0].name);    // Zachary
```

В этом коде массив с именем `data` создается с двумя объектами. Каждый объект имеет свойства `name` и `age`. По умолчанию метод `sort()` будет вызывать `toString()` для каждого объекта, чтобы определить порядок сортировки, который в этом случае не даст логических результатов. Вызов `createComparisonFunction("name")` создает функцию сравнения, которая сортирует элементы на основе свойства `name`, это означает, что первый элемент будет иметь имя `"Nicholas"` и `age` со значением `29`. Когда вызывается `createComparisonFunction("age")`, создается функция сравнения, которая проводит сортировку на основе свойства `age`, то есть первым элементом будет элемент с именем, равным `"Zachary"`, и `age`, равным `28`.

ВНУТРЕННЕЕ УСТРОЙСТВО ФУНКЦИЙ

В ECMAScript 5 внутри функции существовали два специальных объекта: `arguments` и `this`. В ECMAScript 6 было представлено свойство `new.target`.

arguments

Объект `arguments`, как обсуждалось ранее, является массивоподобным объектом, который содержит все аргументы, переданные в функцию. Он доступен только тогда, когда функция объявлена с использованием ключевого слова `function` (в отличие от объявления стрелочной функции). Хотя его основное использование — представление аргументов функции, объект `arguments` также имеет свойство `callee` — указатель на функцию, которой принадлежит объект `arguments`. Рассмотрим следующую классическую функцию вычисления факториала:

```
function factorial(num) {
    if (num <= 1) {
        return 1;
    } else {
        return num * factorial(num - 1);
    }
}
```

Функции вычисления факториала обычно определяются как рекурсивные, как в этом примере, что отлично работает, когда имя функции задано заранее и не будет изменено. Однако правильное выполнение этой функции тесно связано с именем функции `"factorial"`. Это можно отделить с помощью `arguments.callee`:

```
function factorial(num) {
    if (num <= 1) {
        return 1;
    } else {
        return num * arguments.callee(num - 1);
    }
}
```

В этой переписанной версии функции `factorial()` больше нет ссылки на имя `"factorial"` в теле функции, что гарантирует, это рекурсивный вызов будет происходить с правильной функцией независимо от того, как на нее ссылаются. Рассмотрим следующее:

```
let trueFactorial = factorial;

factorial = function() {
    return 0;
};

console.log(trueFactorial(5));    // 120
console.log(factorial(5));        // 0
```

Здесь переменной `trueFactorial` присваивается значение переменной `factorial`, которое эффективно хранит указатель функции во втором месте. Затем переменная `factorial` переназначается на функцию, которая просто возвращает 0. Без использования `arguments.callee` в теле исходной функции `factorial()` вызов `trueFactorial()` вернет `0`. Но если функция отделена от имени функции, `trueFactorial()` правильно вычисляет факториал, а `factorial()` — единственная функция, возвращающая 0.

this

Следующий специальный объект называется `this` и ведет себя по-разному, когда используется внутри стандартной функции и стрелочной.

Внутри стандартной функции он является ссылкой на объект контекста, над которым работает функция, часто называемая *значением this* (когда функция вызывается в глобальной области видимости веб-страницы, объект `this` указывает на `window`). Рассмотрим пример:

```
window.color = 'red';
let o = {
    color: 'blue'
};

function sayColor() {
    console.log(this.color);
}

sayColor();      // 'red'

o.sayColor = sayColor;
o.sayColor();    // 'blue'
```

Функция `sayColor()` определена глобально, но ссылается на объект `this`. Значение `this` не определено до тех пор, пока функция не будет вызвана, поэтому ее значение может быть непоследовательным на протяжении всего выполнения кода. Когда `sayColor()` вызывается в глобальной области видимости, он выдает `"red"`, потому что `this` указывает на `window`, это означает, что `this.color` приравнивается к значению `window.color`. Присваивая функцию объекту `o` и затем вызывая `o.sayColor()`, объект `this` указывает на `o`, поэтому `this.color` приравнивается к `o.color` и отображается «`blue`».

Внутри стрелочной функции `this` ссылается на контекст, в котором определено выражение стрелочной функции. Это демонстрируется в следующем примере, где два различных вызова `sayColor` ссылаются на свойство объекта `window`, являющегося контекстом, внутри которого стрелочная функция была первоначально определена:

```
window.color = 'red';
let o = {
    color: 'blue'
};

let sayColor = () => console.log(this.color);

sayColor();      // 'red'

o.sayColor = sayColor;
o.sayColor();    // 'red'
```

Это особенно полезно в ситуациях, когда события или тайм-ауты будут вызывать функцию внутри функции обратного вызова, когда вызывающий объект не является предполагаемым объектом. Если в этих ситуациях используется стрелочная функция, контекст, на который ссылается `this`, сохраняется:

```
function King() {
    this.royaltyName = 'Henry';
    // 'this' будет экземпляром King
    setTimeout(() => console.log(this.royaltyName), 1000);
}

function Queen() {
    this.royaltyName = 'Elizabeth';
    // 'this' будет объектом window
    setTimeout(function() { console.log(this.royaltyName); }, 1000);
}

new King();     // Henry
new Queen();    // undefined
```

ПРИМЕЧАНИЕ Помните, что имена функций – это просто переменные, содержащие указатели, поэтому глобальная функция `sayColor()` и `o.sayColor()` указывают на одну и ту же функцию, даже если они выполняются в разных контекстах.

caller

ECMAScript 5 также формализует дополнительное свойство объекта функции: `caller`. Хотя оно не было определено в ECMAScript 3, все браузеры, за исключением более ранних версий Opera, поддерживали это свойство, содержащее ссылку на функцию, которая вызвала эту функцию, или `null`, если функция была вызвана из глобальной области видимости. Например:

```
function outer() {
    inner();
}
function inner() {
    console.log(inner.caller);
}
outer();
```

Этот код отображает предупреждение с исходным текстом функции `external()`. Так как `external()` вызывает `inner()`, `inner.caller` указывает обратно на `external()`. Для более слабой связи можно получить доступ к той же информации через `arguments.callee.caller`:

```
function outer() {
    inner();
}

function inner() {
    console.log(arguments.callee.caller);
}

outer();
```

Когда код функции выполняется в строгом режиме, попытка доступа к `arguments.callee` приводит к ошибке. ECMAScript 5 также определяет `arguments.caller`, который также приводит к ошибке в строгом режиме и всегда не определен вне строгого режима. Это должно устранить путаницу между `arguments.caller` и свойством `caller` функций. Данные изменения были внесены как дополнения безопасности к языку, поэтому сторонний код не может проверять другой код, работающий в том же контексте.

Строгий режим накладывает одно дополнительное ограничение: нельзя присвоить значение свойству `caller` функции. Это приводит к ошибке.

new.target

Функции всегда могли вести себя как конструктор для создания экземпляра нового объекта и как обычная вызываемая функция. В ECMAScript 6 появляется возможность определять, была ли функция вызвана с ключевым словом `new`, используя `new.target`. Если функция вызывается обычным способом, `new.target` будет неопределенным. Если функция вызывается с использованием ключевого слова `new`, `new.target` будет ссылаться на конструктор или функцию.

```
function King() {
    if (!new.target) {
        throw 'King must be instantiated using "new"'
    }
    console.log('King instantiated using "new"';
}

new King();    // King instantiated using "new"
King();        // Error: King must be instantiated using "new"
```

СВОЙСТВА И МЕТОДЫ ФУНКЦИЙ

Функции являются объектами в ECMAScript и, как упоминалось ранее, поэтому имеют свои собственные свойства и методы. Каждая функция имеет два свойства: `length` и `prototype`. Свойство `length` указывает на количество именованных аргументов, которые ожидает функция, как в этом примере:

```
function sayName(name) {
    console.log(name);
}

function sum(num1, num2) {
    return num1 + num2;
}

function sayHi() {
    console.log("hi");
}

console.log(sayName.length);    // 1
console.log(sum.length);        // 2
console.log(sayHi.length);      // 0
```

Этот код определяет три функции, каждая из которых имеет разное количество именованных аргументов. Функция `sayName()` задает один аргумент, поэтому ее свойство `length` равно 1. Аналогично функция `sum()` задает два аргумента, поэтому ее свойство `length` равно 2, а `sayHi()` не имеет именованных аргументов, поэтому ее `length` равно 0.

Свойство `prototype` является, пожалуй, самой интересной частью ядра ECMAScript. Прототип — это фактическое расположение всех методов экземпляра для ссылочных типов, то есть такие методы, как `toString()` и `valueOf()`, фактически существуют в `prototype`, а затем доступны из экземпляров объекта. Это свойство очень важно с точки зрения определения ваших собственных ссылочных типов и наследования. (Эти темы рассматриваются в главе 8 «Объекты, классы и объектно-ориентированное программирование».) В ECMAScript 5 свойство `prototype` не является перечисляемым и поэтому не может быть обнаружено с использованием `for-in`.

Существуют два дополнительных метода для функций: `apply()` и `call()`. Оба эти метода вызывают функцию с определенным значением `this`, эффективно устанавливая значение объекта `this` внутри тела функции. Метод `apply()` принимает два

аргумента: значение this внутри функции и массив аргументов. Второй аргумент может быть экземпляром Array, но он также может быть объектом arguments. Рассмотрим пример:

```
function sum(num1, num2) {
    return num1 + num2;
}

function callSum1(num1, num2) {
    return sum.apply(this, arguments);       // передача значений в объект
                                             // arguments
}

function callSum2(num1, num2) {
    return sum.apply(this, [num1, num2]);    // передача значений в массив
}

console.log(callSum1(10, 10));    // 20
console.log(callSum2(10, 10));    // 20
```

В этом примере callSum1() запускает метод sum(), передавая его как значение this (которое равно window, потому что оно вызывается в глобальной области видимости), а также передавая объект arguments. Метод callSum2() также вызывает sum(), но вместо этого он передает массив аргументов. Обе функции будут выполнены и вернут правильный результат.

ПРИМЕЧАНИЕ В строгом режиме значение this функции, вызываемой без объекта контекста, не приводится к window. Вместо этого this становится undefined, если он не установлен явно путем присоединения функции к объекту или использования apply() или call().

Метод call() демонстрирует то же поведение, что и apply(), но аргументы передаются ему по-разному. Первый аргумент — это значение this, но остальные аргументы передаются непосредственно в функцию. Использование аргументов call() должно быть конкретно перечислено, как в этом примере:

```
function sum(num1, num2) {
    return num1 + num2;
}

function callSum(num1, num2) {
    return sum.call(this, num1, num2);
}

console.log(callSum(10, 10));    // 20
```

Метод callSum() должен явно передавать каждый из своих аргументов в метод call(). Результат такой же, как при использовании apply(). Решение использовать apply() или call() зависит исключительно от самого простого способа передачи аргументов в функцию. Если вы намереваетесь передать объект аргументов напрямую или

если у вас уже есть массив данных для передачи, тогда лучше применять `apply()`; в противном случае `call()` может быть более подходящим выбором. (Если нет аргументов для передачи, то эти методы идентичны.)

Истинная сила `apply()` и `call()` заключается не в их способности передавать аргументы, а в их способности расширять значение `this` внутри функции. Рассмотрим следующий пример:

```
window.color = 'red';
let o = {
    color: 'blue'
};

function sayColor() {
    console.log(this.color);
}

sayColor();    // red

sayColor.call(this);      // red
sayColor.call(window);    // red
sayColor.call(o);         // blue
```

Этот пример является модифицированной версией кода, использовавшегося для иллюстрации объекта `this`. Еще раз: `sayColor()` определен как глобальная функция, и когда он вызывается в глобальной области видимости, он отображает `"red"`, потому что `this.color` оценивается как `window.color`. Затем можно явно вызвать функцию в глобальной области видимости, используя `sayColor.call(this)` и `sayColor.call(window)`, которые оба отображают `"red"`. Запуск `sayColor.call(o)` переключает контекст функции так, что он указывает на o, в результате чего отображается `"blue"`.

Преимущество использования `call()` (или `apply()`) для расширения области видимости состоит в том, что объекту не нужно ничего знать о методе. В первой версии этого примера функция `sayColor()` помещалась непосредственно в объект o до его вызова; в обновленном примере этот шаг больше не требуется.

ECMAScript 5 определяет дополнительный метод, называемый `bind()`. Метод `bind()` создает новый экземпляр функции, значение `this` которого *связано* со значением, переданным в `bind()`. Например:

```
window.color = 'red';
var o = {
    color: 'blue'
};

function sayColor() {
    console.log(this.color);
}
let objectSayColor = sayColor.bind(o);
objectSayColor();    // blue
```

Здесь новая функция с именем `objectSayColor()` создается из `sayColor()` путем вызова `bind()` и передачи объекта o. Функция `objectSayColor()` имеет значение `this`,

эквивалентное o, поэтому при вызове функции, даже в качестве глобального вызова, отображается строка "blue".

Для функций унаследованные методы toLocaleString() и toString() всегда возвращают код функции. Точный формат этого кода варьируется от браузера к браузеру — некоторые возвращают код в точности так, как он отображался в исходном варианте, включая комментарии, тогда как другие возвращают внутреннее представление кода, в котором удалены комментарии и, возможно, проведены некоторые изменения в коде интерпретатора. Из-за этих различий нельзя полагаться на то, что возвращается для какой-либо важной функциональности, хотя эта информация может быть полезна для целей отладки. Унаследованный метод valueOf() просто возвращает саму функцию.

ФУНКЦИИ-ВЫРАЖЕНИЯ

Функции-выражения — одна из наиболее мощных возможностей JavaScript, которую многие не понимают. Определить функцию можно с помощью объявления или выражения. Объявление функции имеет следующий формат:

```
function functionName(arg0, arg1, arg2) {
    // тело функции
}
```

Объявления функций примечательны тем, что они *поднимаются* (hoisting), то есть считываются до выполнения кода. Это означает, что вызовы функции могут предшествовать ее объявлению:

```
sayHi();
function sayHi() {
    console.log("Hi!");
}
```

Ошибка в этом примере не возникает как раз благодаря тому, что объявление функции считывается до выполнения кода.

Функцию также можно создать с помощью функции-выражения. Обычно она имеет следующий формат:

```
let functionName = function(arg0, arg1, arg2) {
    // тело функции
};
```

Этот паттерн выглядит как обычная инициализация переменной functionName, только здесь ей присваивается функция. Созданная функция считается *анонимной* (anonymous function), потому что после ключевого слова function нет идентификатора; анонимные функции также иногда называют *лямбда-функциями* (lambda functions). Свойство name у таких функций содержит пустую строку.

Функции-выражения можно использовать только после их создания. Так, при выполнении следующего кода возникает ошибка:

```
sayHi();    // ошибка – функция еще не существует
let sayHi = function() {
    console.log("Hi!");
};
```

Подъем функций — ключевое различие между объявлениями и функциями-выражениями. Например, результат выполнения следующего кода может вас удивить:

```
// Никогда не делайте этого!
if(condition) {
    function sayHi() {
        console.log("Hi!");
    }
} else {
    function sayHi() {
        console.lo("Yo!");
    }
}
```

На первый взгляд этот код выбирает то или иное объявление функции `sayHi()` в зависимости от условия, но в действительности ECMAScript не поддерживает такой синтаксис и интерпретаторы JavaScript пытаются исправить ошибку. Проблема в том, что они делают это по-разному. Большинство браузеров возвращает второе объявление функции независимо от условия, а Firefox возвращает первое, если условие истинно. Такой код опасен, и использовать его не следует, однако функции-выражения в нем вполне допустимы:

```
// все в порядке
let sayHi;

if(condition) {
    sayHi = function() {
         console.log("Hi!");
    };
} else {
    sayHi = function() {
         console.log("Yo!");
    };
}
```

Этот код работает без сюрпризов, назначая одно из функций-выражений переменной `sayHi` в зависимости от условия.

Возможность назначать функции переменным позволяет также возвращать функции из других функций.

```
function createComparisonFunction(propertyName) {

    return function(object1, object2) {
        let value1 = object1[propertyName];
        let value2 = object2[propertyName];

        if (value1 < value2) {
```

```
            return -1;
        } else if (value1 > value2) {
            return 1;
        } else {
            return 0;
        }
    };
}
```

Функция `createComparisonFunction()` возвращает анонимную функцию. Возможно, она будет назначена переменной или вызвана иным образом, но внутри `createComparisonFunction()` она анонимна. Каждый раз, когда функции используются как значения, мы имеем дело с функциями-выражениями. Позже в этой главе мы рассмотрим другие способы их применения.

РЕКУРСИЯ

Рекурсивной функцией (recursive function) обычно называют функцию, которая вызывает сама себя, например:

```
function factorial(num) {
    if (num <= 1) {
        return 1;
    } else {
        return num * factorial(num-1);
    }
}
```

Это классический пример вычисления факториала. Хотя с функцией все в порядке, ее работу легко можно нарушить:

```
let anotherFactorial = factorial;
factorial = null;
console.log(anotherFactorial(4));    // ошибка!
```

Здесь функция `factorial()` назначается переменной `anotherFactorial`, а затем переменной `factorial` присваивается значение `null`, в результате чего остается только одна ссылка на оригинальную функцию. При вызове `anotherFactorial()` возникает ошибка, потому что функция `factorial()` больше недоступна. Эту проблему можно устранить с помощью свойства `arguments.callee`.

Если помните, `arguments.callee` — это указатель на выполняемую функцию, с помощью которого можно вызвать ее рекурсивно:

```
function factorial(num) {
    if (num <= 1) {
        return 1;
    } else {
        return num * arguments.callee(num-1);
    }
}
```

Использование свойства `arguments.callee` в выделенной строке вместо имени функции гарантирует, что функция будет работать независимо от способа доступа к ней. В рекурсивных функциях рекомендуется всегда задействовать свойство `arguments.callee`, а не имя функции.

В строгом режиме значение `arguments.callee` недоступно, а попытка прочитать его приводит к ошибке. К счастью, тот же результат можно получить с помощью *именованных функций-выражений* (named function expressions), например:

```
const factorial = (function f(num) {
    if (num <= 1) {
        return 1;
    } else {
        return num * f(num-1);
    }
});
```

В этом коде именованная функция-выражение `f()` назначается переменной `factorial`. Имя `f` остается тем же, даже если функция назначается другой переменной, благодаря чему рекурсивный вызов всегда выполняется правильно. Этот подход работает и в нестрогом, и в строгом режимах.

ОПТИМИЗАЦИЯ С ПОМОЩЬЮ ХВОСТОВЫХ ВЫЗОВОВ

Спецификация ECMAScript 6 также представила оптимизацию управления памятью, которая позволяет механизму JavaScript повторно использовать кадры стека, когда выполняются определенные условия. В частности, эта оптимизация относится к «хвостовым вызовам», где возвращаемое значение внешней функции — это возвращаемое значение внутренней функции, как показано ниже:

```
function outerFunction() {
    return innerFunction();    // хвостовый вызов
}
```

До оптимизации ES6 выполнение этого примера имело бы следующий эффект в памяти:

1. Выполнение достигает тела `externalFunction`, первый стековый кадр помещается в стек.
2. Тело `externalFunction` выполняется, достигнут оператор `return`. Чтобы вычислить возвращаемое выражение, нужно вызвать `innerFunction`.
3. Выполнение достигает тела `innerFunction`, второй кадр стека помещается в стек.
4. Тело `innerFunction` выполняется, и вычисляется его возвращаемое значение.
5. Возвращаемое значение передается обратно в `externalFunction`, которая, в свою очередь, может вернуть это значение.
6. Стековые кадры выталкиваются из стека.

После оптимизации ES6 выполнение этого примера будет иметь следующий эффект в памяти:

1. Выполнение достигает тела `externalFunction`, первый кадр стека помещается в стек.
2. Тело `externalFunction` выполняется, достигнут оператор `return`. Чтобы вычислить возвращаемое выражение, нужно вызвать `innerFunction`.
3. Механизм распознает, что первый стековый кадр можно безопасно извлечь из стека, поскольку возвращаемое значение `innerFunction` также является возвращаемым значением `externalFunction`.
4. Стековый кадр `externalFunction` выталкивается из стека.
5. Выполнение достигает тела `innerFunction`, кадр стека помещается в стек.
6. Тело `innerFunction` выполняется, и вычисляется его возвращаемое значение.
7. Кадр стека `innerFunction` выталкивается из стека.

Очевидное отличие здесь состоит в том, что первая реализация будет использовать дополнительный стековый кадр для каждого последующего вызова вложенной функции, тогда как вторая реализация будет использовать один стековый кадр для всей очередности выполнения. Это является ядром оптимизации хвостового вызова ES6: если функция структурирована таким образом, чтобы ее можно было безопасно отбрасывать при хвостовом вызове, движок сделает это.

> **ПРИМЕЧАНИЕ** Невозможно напрямую измерить, происходит ли оптимизация хвостового вызова. Однако, поскольку это является частью спецификации ES6, совместимые браузеры гарантированно применят оптимизацию, если код удовлетворяет требованиям.

Требования к оптимизации с помощью хвостовых вызовов

Движок способен выполнять такую оптимизацию только тогда, когда он уверен, что внешний стековый кадр действительно больше не нужен. Это требует следующих условий:

- код выполняется в строгом режиме;
- возвращаемым значением внешней функции является вызванная функция хвостового вызова;
- дальнейшего выполнения не требуется после возврата функции хвостового вызова;
- функция хвостового вызова не является замыканием, относящимся к переменным в области видимости внешней функции.

Ниже приведены несколько примеров, которые нарушают эти условия и поэтому не подлежат оптимизации с помощью хвостового вызова:

```
"use strict";

// Оптимизации нет: хвостовый вызов не возвращается
function outerFunction() {
    innerFunction();
}

// Оптимизации нет: хвостовый вызов не возвращается напрямую
function outerFunction() {
    let innerFunctionResult = innerFunction();
    return innerFunctionResult;
}

// Оптимизации нет: хвостовый вызов должен быть приведен к строке после возврата
function outerFunction() {
    return innerFunction().toString();
}

// Оптимизации нет: хвостовый вызов является замыканием
function outerFunction() {
    let foo = 'bar';
    function innerFunction() { return foo; }

    return innerFunction();
}
```

Ниже приведено несколько примеров, которые не нарушают эти условия и, следовательно, будут оптимизированы:

```
"use strict";

// Можно оптимизировать: вычисление аргумента происходит до
// отбрасывания стекового кадра
function outerFunction(a, b) {
    return innerFunction(a + b);
}

// Можно оптимизировать: начальные возвращаемые значения
// не имеют значения стекового кадра
function outerFunction(a, b) {
    if (a < b) {
        return a;
    }
    return innerFunction(a + b);
}

// Можно оптимизировать: обе внутренние функции располагаются в хвостовой позиции
function outerFunction(condition) {
    return condition ? innerFunctionA() : innerFunctionB();
}
```

Распространенным источником путаницы является природа разграничения хвостовых и рекурсивных вызовов. Оптимизация будет применяться к рекурсивным

или нерекурсивным хвостовым вызовам; механизм не делает различий, если функция вызывает себя или другую функцию в хвостовом вызове. Тем не менее код будет получать наибольшую выгоду от этой оптимизации в рекурсивных ситуациях, так как этот код является форм-фактором, который стремится заполнить стек кадрами.

ПРИМЕЧАНИЕ Причина, по которой требуется использование строгого режима, заключается в том, что вызов функции в нестрогом режиме позволяет использовать `f.arguments` и `f.caller`, которые являются ссылками на стековый кадр внешней функции. Очевидно, это означает, что оптимизация была бы невозможна, и, следовательно, существует данное требование для предотвращения появления этих свойств.

Код для оптимизации с помощью хвостовых вызовов

Для наглядности преобразуем простую рекурсивную функцию в функцию, которая будет оптимизирована. Для примера возьмем следующий простой код Фибоначчи:

```
function fib(n) {
    if (n < 2) {
        return n;
    }

    return fib(n – 1) + fib(n – 2);
}

console.log(fib(0));    // 0
console.log(fib(1));    // 1
console.log(fib(2));    // 1
console.log(fib(3));    // 2
console.log(fib(4));    // 3
console.log(fib(5));    // 5
console.log(fib(6));    // 8
```

Нетрудно заметить, что эта функция не подлежит оптимизации с помощью хвостового вызова, поскольку в возвращаемом выражении происходит дисквалифицирующая операция сложения. В результате количество кадров стека для fib(*n*) будет иметь сложность памяти O(2^n). Поэтому очень легко загрузить браузер даже чем-то простым:

```
fib(1000);
```

Конечно, есть методы, которые решают эту проблему по-разному, такие как запоминание или сглаживание в итеративном цикле, но можно сохранить рекурсивную реализацию и преобразовать ее в форму, подходящую для оптимизации. Отличная стратегия — использовать вложенные функции: внешнюю, действующую в качестве базового случая, и внутреннюю, выступающую в качестве рекурсивного случая:

```
"use strict";

// базовый случай
function fib(n) {
    return fibImpl(0, 1, n);
}

// рекурсивный случай
function fibImpl(a, b, n) {
    if (n === 0) {
        return a;
    }
    return fibImpl(b, a + b, n - 1);
}
```

В этой реализации все требования по оптимизации с помощью хвостовых вызовов удовлетворены, и у браузера не возникнет проблем с `fib(1000);`

ЗАМЫКАНИЯ

Термины «анонимная функция» и «замыкание» часто употребляются как синонимы, но это ошибка. *Замыкание* (closure) — это функция, которой доступны переменные из области видимости другой функции. Обычно для создания замыкания одну функцию определяют внутри другой, как функцию `createComparisonFunction()` в предыдущем примере:

```
function createComparisonFunction(propertyName) {

    return function(object1, object2) {
        let value1 = object1[propertyName];
        let value2 = object2[propertyName];

        if (value1 < value2) {
            return -1;
        } else if (value1 > value2) {
            return 1;
        } else {
            return 0;
        }
    };
}
```

В выделенных строках внутренней (анонимной) функции осуществляется доступ к переменной `propertyName`, полученной внешней функцией. Интересно то, что эта переменная остается доступной для анонимной функции, даже когда она вызывается в другом коде. Это происходит потому, что цепочка областей видимости внутренней функции включает область видимости функции `createComparisonFunction()`. Чтобы понять, как это возможно, давайте обсудим, что происходит при первом вызове функции.

Чтобы хорошо разбираться в замыканиях, важно в деталях рассмотреть создание и использование цепочек областей видимости, с которыми мы познакомились

в главе 4. При вызове функции для нее создаются контекст выполнения и цепочка областей видимости. Объект активации функции инициализируется значениями из объекта `arguments` и имеющимися именованными аргументами. Объект активации внешней функции является вторым в цепочке областей видимости, затем ее продолжают другие оставшиеся функции-контейнеры и завершает глобальный контекст выполнения.

При чтении и записи значений в функции выполняется поиск соответствующих переменных в цепочке областей видимости. Рассмотрим пример:

```
function compare(value1, value2) {
    if (value1 < value2) {
        return -1;
    } else if (value1 > value2) {
        return 1;
    } else {
        return 0;
    }
}

let result = compare(5, 10);
```

В этом коде определена функция `compare()`, которая затем вызывается в глобальном контексте выполнения. При первом вызове функции создается объект активации, содержащий переменные `arguments`, `value1` и `value2`. Следующим в цепочке областей видимости контекста выполнения `compare()` является объект переменных глобального контекста выполнения, который содержит переменные `this`, `result` и `compare`. Эти связи иллюстрирует рис. 10.1.

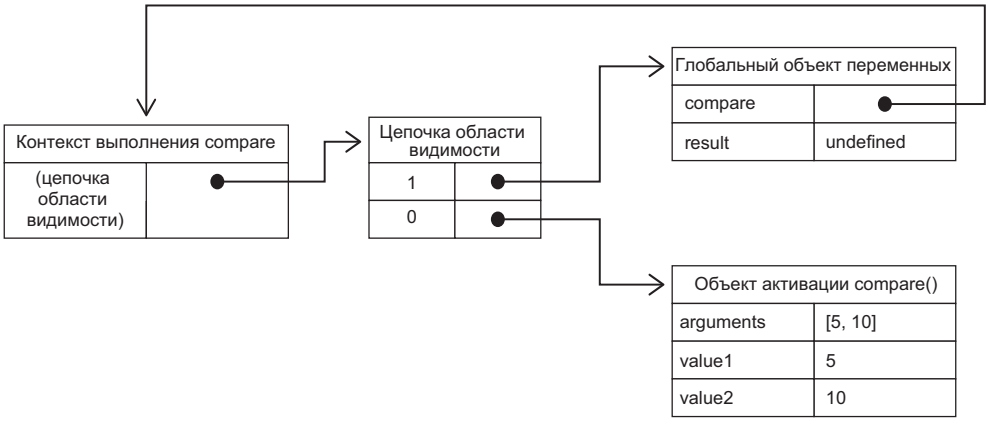

Рис. 10.1

За кулисами переменные в каждом контексте выполнения представляются объектом. Объект переменных глобального контекста существует всегда, тогда как объекты переменных локальных контекстов, например функции `compare()`, создаются только

на время выполнения функции. При определении функции `compare()` создается ее цепочка областей видимости, которая инициализируется глобальным объектом переменных и сохраняется во внутреннем свойстве `[[Scope]]`. При вызове функции создается контекст выполнения и составляется его цепочка областей видимости, для чего в нее копируются объекты из свойства `[[Scope]]` функции. После этого система создает объект активации (используемый также как объект переменных) и помещает его в начало цепочки областей видимости контекста. В нашем примере это означает, что цепочка областей видимости контекста выполнения `compare()` содержит два объекта переменных: локальный объект активации и глобальный объект переменных. Имейте в виду, что цепочка областей видимости является, по сути, списком указателей на объекты переменных и не содержит сами объекты.

Когда переменная используется внутри функции, выполняется поиск имени переменной в цепочке областей видимости. По завершении функции локальный объект активации уничтожается, после чего в памяти остается только глобальная область видимости. Однако замыкания работают иначе.

Если одна функция определена внутри другой, в цепочку областей видимости внутренней функции добавляется объект активации внешней. Так, в примере с функцией `createComparisonFunction()` ссылка на нее содержится в цепочке областей видимости анонимной функции. На рис. 10.2 показаны эти связи при выполнении следующего кода:

```
let compare = createComparisonFunction("name");
let result = compare({ name: "Nicholas" }, { name: "Matt" });
```

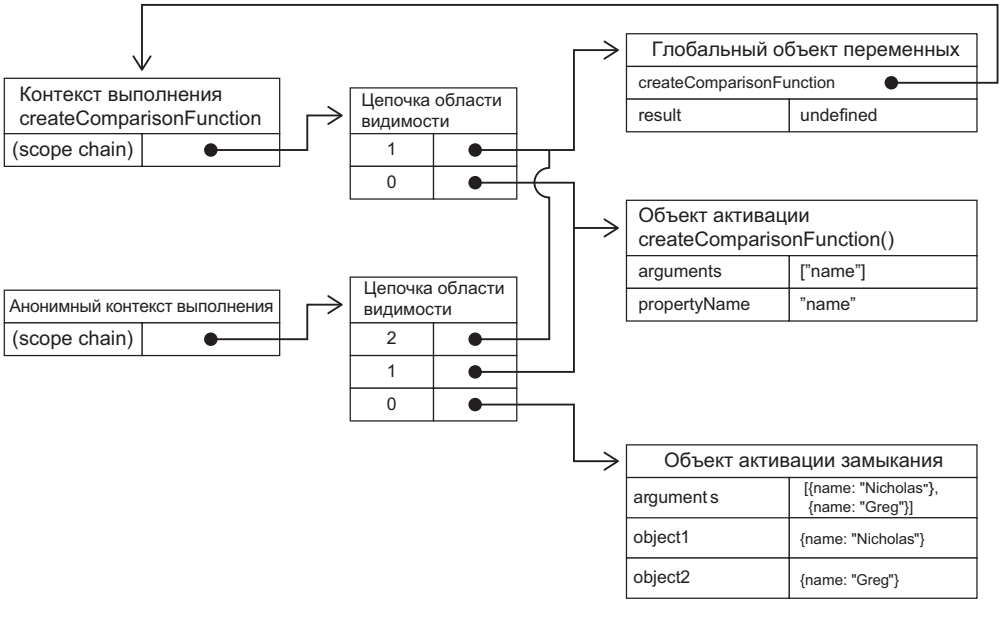

Рис. 10.2

Когда анонимная функция возвращается из `createComparisonFunction()`, ее цепочка областей видимости содержит объект активации `createComparisonFunction()` и глобальный объект переменных, что предоставляет анонимной функции доступ ко всем переменным функции `createComparisonFunction()`. Другой интересный побочный эффект заключается в том, что объект активации функции `createComparisonFunction()` не может быть уничтожен при ее завершении, потому что ссылка на него остается в цепочке областей видимости анонимной функции. Таким образом, при завершении функции `createComparisonFunction()` цепочка областей видимости ее контекста выполнения уничтожается, но ее объект активации остается в памяти вплоть до уничтожения анонимной функции:

```
// создание функции
let compareNames = createComparisonFunction("name");

// вызов функции
let result = compareNames({ name: "Nicholas" }, { name: "Matt"});

// завершение работы с функцией — теперь память может быть возвращена среде
compareNames = null;
```

Здесь функция сравнения назначается переменной `compareNames`. По завершении работы с функцией переменной `compareNames` присваивается значение `null`, что разрывает ее связь с функцией и позволяет сборщику мусора возвратить память среде. После этого все области видимости (кроме глобальной) можно безопасно уничтожить. Состояния цепочек областей видимости при вызове функции `compareNames()` в этом примере показаны на рис. 10.2.

> **ПРИМЕЧАНИЕ** Поскольку замыкание захватывает область видимости внешней функции, оно занимает больше памяти, чем обычная функция. Чрезмерное использование замыканий заметно увеличивает потребление памяти, поэтому применять их без необходимости не следует. Хотя оптимизирующие интерпретаторы JavaScript, такие как V8, пытаются освобождать память, заблокированную в замыканиях, при работе с замыканиями все равно нужно знать меру.

Объект this

При использовании объекта `this` внутри замыканий могут возникать непростые ситуации. Если функция не определена с использованием стрелочного синтаксиса, привязка объекта `this` во время выполнения происходит на основе контекста, в котором выполняется функция: в глобальных функциях `this` имеет значение `window` в нестрогом режиме и `undefined` в строгом, а в методах объектов `this` указывает на объект. Анонимные функции к объектам не привязываются, поэтому в них `this` указывает на `windows` в нестрогом режиме и имеет значение `undefined` в строгом, однако из-за синтаксиса замыканий это не всегда очевидно. Рассмотрим следующий пример:

```
window.identity = "The Window";

let object = {
    identity: "My Object",

    getIdentityFunc() {
        return function() {
            return this.identity;
        };
    }
};

console.log(object. getIdentityFunc()());    // "The Window"
```

В этом коде мы создаем глобальную переменную `identity` и объект со свойством `identity`. У объекта также есть метод `getIdentityFunc()`, возвращающий анонимную функцию, которая, в свою очередь, возвращает `this. identity`. Поскольку `getIdentityFunc()` возвращает функцию, вызов `object. getIdentityFunc()()` немедленно вызывает ее, в результате чего возвращается строка. Однако в нашем случае это строка `"The Window"`, являющаяся значением глобальной переменной `identity`. Почему в анонимной функции не был использован объект `this` внешней функции?

Если помните, каждая вызванная функция автоматически получает специальные переменные `this` и `arguments`. У внутренней функции никогда нет прямого доступа к этим переменным, полученным внешней функцией. Чтобы в замыкании был доступен другой объект `this`, можно сохранить его в переменной, например:

```
window.identity = "The Window";

let object = {
    identity : "My Object",

    getIdentityFunc() {
        let that = this;
        return function() {
            return that.identity;
        };
    }
};

console.log(object.getIdentityFunc()());    // "My Object"
```

Этот пример отличается от предыдущего двумя выделенными строками. Перед определением анонимной функции объект `this` назначается переменной `that`, которая доступна в замыкании как переменная с уникальным именем, определенная в функции-контейнере. Даже после возвращения функции переменная `that` все еще связана с `object`, поэтому вызов `object.getIdentityFunc()()` возвращает строку `"My Object"`.

ПРИМЕЧАНИЕ Так работают оба объекта: и `this`, и `arguments`. Если в замыкании нужен доступ к объекту `arguments`, находящемуся в области видимости функции-контейнера, сохраните ссылку на него в другой переменной, доступной замыканию.

В нескольких специальных случаях значение this может оказаться не таким, как вы думаете, например:

```
window.identity = "The Window";

let object = {
    identity : "My Object",

    getIdentity() {
        return this.identity;
    }
};
```

Здесь метод getIdentity() просто возвращает значение this.identity. Вот разные способы его вызова и соответствующие результаты:

```
object.getIdentity();          // "My Object"
(object.getIdentity)();        // "My Object"
(object.getIdentity = object.getIdentity)();    // "The Window"
```

В первой строке метод object.getIdentity(), вызванный обычным образом, возвращает строку "My Object", потому что значение this.identity совпадает с object.identity. Во второй строке вызов object.getIdentity в скобках может показаться ссылкой на функцию, но на самом деле используется то же значение this, потому что инструкции object.getIdentity и (object.getIdentity) эквивалентны. В третьей строке значением выражения присваивания является сама функция, поэтому значение this не сохраняется и возвращается строка "The Window".

Едва ли вы будете намеренно использовать приемы, представленные во второй и третьей строках, однако полезно знать, как способно меняться значение this даже при таком незначительном изменении синтаксиса.

Утечки памяти

Замыкания вызывают специфические проблемы в браузерах Internet Explorer до версии 9, потому что в них реализованы разные способы сборки мусора для JScript- и COM-объектов (см. главу 4). Из-за сохранения области видимости, в которой находится HTML-элемент, иногда его невозможно уничтожить. Рассмотрим следующий пример:

```
function assignHandler() {
    let element = document.getElementById("someElement");
    element.onclick = () => console.log(element.id);
}
```

В этом коде в качестве обработчика событий переменной element создается замыкание, в результате формируется циклическая ссылка (события обсуждаются в главе 13). Анонимная функция хранит ссылку на объект активации функции assignHandler(), что не позволяет уменьшить счетчик ссылок на element. Пока анонимная функция существует, счетчик ссылок на element не может быть обнулен,

а это означает, что память никогда не будет освобождена. Проблему можно решить, немного изменив код:

```
function assignHandler() {
    let element = document.getElementById("someElement");
    let id = element.id;

    element.onclick = () => console.log(id);

    element = null;
}
```

В этой версии в замыкании используется копия идентификатора элемента, что устраняет циклическую ссылку, но этого недостаточно для решения проблемы памяти. Помните: у замыкания есть ссылка на весь объект активации внешней функции, содержащий переменную `element`. Даже если замыкание не ссылается на `element` непосредственно, ссылка все равно хранится в этом объекте активации, поэтому еще нужно присвоить переменной `element` значение `null`. Это аннулирует ссылку на COM-объект и уменьшает его счетчик ссылок, позволяя возвратить память среде.

НЕМЕДЛЕННО ВЫЗЫВАЕМЫЕ ФУНКЦИИ-ВЫРАЖЕНИЯ

Анонимная функция, которая вызывается немедленно, чаще всего называется *немедленно вызываемой функцией-выражением* (IIFE, immediately invoked function expression). Она напоминает объявление функции, но поскольку заключена в скобки, то интерпретируется как функция-выражение. Затем эта функция вызывается через второй набор скобок в конце. Основной синтаксис выглядит следующим образом:

```
(function() {
    // код блока
})();
```

Использование IIFE для имитации области действия блока использует значения, определенные внутри функции-выражения, которое выполняется немедленно, тем самым предлагая поведение, подобное области действия блока, используя переменные области действия функции. (Полезность IIFE была намного выше в предыдущих версиях ECMAScript 6, где блочные переменные не поддерживались.) Рассмотрим следующий пример:

```
// IIFE
(function () {
    for (var i = 0; i < count; i++) {
        console.log(i);
    }
})();

console.log(i);    // Ошибка
```

Предыдущий код выдаст ошибку при попытке вызова `console.log()` вне IIFE, потому что переменная, определенная внутри IIFE, не имеет области видимости для доступа вне функции-выражения. Для попыток предотвращения утечки переменных в ECMAScript 5.1 или ранее этот паттерн очень полезен. Он также ограничивает проблему с памятью замыкания, поскольку теперь не существует ссылки на анонимную функцию. Следовательно, цепочка областей действия может быть уничтожена сразу после завершения функции.

В ECMAScript 6 IIFE больше не требуется для эмуляции области видимости блока, поскольку переменные в области видимости блоков будут работать точно так же без использования IIFE (показано ниже двумя различными способами):

```
// Область видимости блока
{
    let i;
    for (i = 0; i < count; i++) {
        console.log(i);
    }
}
console.log(i);    // Throws an error

// Область видимости функции
for (let i = 0; i < count; i++) {
    console.log(i);
}

console.log(i);    // Ошибка
```

Канонический пример, демонстрирующий полезность IIFE, включал его способность фиксировать значения параметров:

```
let divs = document.querySelectorAll('div');

// Это не сработает!
for (var i = 0; i < divs.length; ++i) {
    divs[i].addEventListener('click', function() {
        console.log(i);
    });
}
```

При этом используется ключевое слово `var` для объявления итератора цикла, который не ограничен областью действия цикла. В этой презентации клик мышкой по каждому элементу `<div>` выведет общее количество элементов, поскольку переменная счетчика все еще существует при выполнении обработчика.

Ранее решением этой проблемы было использование IIFE для немедленного выполнения функции-выражения, которой передается текущее значение счетчика, тем самым «замораживая» значение счетчика, при определении обработчика:

```
let divs = document.querySelectorAll('div');

for (var i = 0; i < divs.length; ++i) {
    divs[i].addEventListener('click', (function(frozenCounter) {
```

```
        return function() {
            console.log(frozenCounter);
        };
    })(i));
}
```

Это, однако, стало ненужным после введения переменных области видимости блока ECMAScript:

```
let divs = document.querySelectorAll('div');

for (let i = 0; i < divs.length; ++i) {
    divs[i].addEventListener('click', function() {
        console.log(i);
    });
}
```

Этот код будет правильно выводить соответствующий индекс с каждым кликом. Выполняющийся обработчик события будет ссылаться на значение счетчика внутри цикла `for`. В ECMAScript 6 цикл `for`, использующий ключевое слово переменной области видимости — здесь это `let` — для итератора, создаст отдельный экземпляр счетчика для каждой итерации, что позволит каждому обработчику кликов ссылаться на этот конкретный счетчик при выполнении. Важно отметить, что этого не произойдет, если эта переменная будет выведена за пределы цикла `for`. Следующий фрагмент кода столкнется с той же проблемой, что и использование `var i = 0` внутри цикла:

```
let divs = document.querySelectorAll('div');

// Это не сработает
let i;
for (i = 0; i < divs.length; ++i) {
    divs[i].addEventListener('click', function() {
        console.log(i);
    });
}
```

ЗАКРЫТЫЕ ПЕРЕМЕННЫЕ

Строго говоря, в JavaScript нет закрытых членов — все свойства объектов являются открытыми. Тем не менее концепция *закрытых переменных* (private variables) поддерживается. Любая переменная, определенная внутри функции или блока, считается закрытой, поскольку она недоступна вне функции. Это относится к аргументам функций, локальным переменным, а также к функциям, определенным внутри других функций. Рассмотрим пример:

```
function add(num1, num2) {
    let sum = num1 + num2;
    return sum;
}
```

Закрытые переменные `num1`, `num2` и `sum` доступны внутри, но не вне этой функции. Если создать внутри нее замыкание, переменные будут доступны в нем по цепочке областей видимости, что позволяет создавать открытые методы с доступом к закрытым переменным.

Привилегированный метод (privileged method) — это открытый метод, предоставляющий доступ к закрытым переменным и (или) функциям. Есть два способа создания привилегированных методов для объектов. Первый — сделать это в конструкторе:

```
function MyObject() {

    // закрытые переменные и функции
    let privateVariable = 10;

    function privateFunction() {
        return false;
    }

    // привилегированные методы
    this.publicMethod = function () {
        privateVariable++;
        return privateFunction();
    };
}
```

При использовании этого паттерна сначала в конструкторе определяются все закрытые переменные и функции, после чего для доступа к ним создаются привилегированные методы. Этот паттерн работает, потому что при определении в конструкторе привилегированные методы становятся замыканиями, которым доступны все переменные и функции, определенные в области видимости конструктора. В данном примере переменная `privateVariable` и функция `privateFunction()` доступны только методу `publicMethod()`. Как только создан экземпляр `MyObject`, прямой доступ к `privateVariable` и `privateFunction()` невозможен — использовать их можно только через метод `publicMethod()`.

С помощью закрытых и привилегированных членов можно скрывать данные, которые не должны быть доступны напрямую:

```
function Person(name) {

    this.getName = function() {
        return name;
    };

    this.setName = function (value) {
        name = value;
    };
}

let person = new Person("Nicholas");
console.log(person.getName());        // "Nicholas"
person.setName("Greg");
console.log(person.getName());        // "Greg"
```

В этом конструкторе определены привилегированные методы `getName()` и `setName()`. Каждый из них доступен вне конструктора и использует закрытую переменную `name`. Вне конструктора `Person` получить доступ к `name` невозможно. Поскольку оба метода определены внутри конструктора, они являются замыканиями и имеют доступ к `name` по цепочке областей видимости.

Закрытая переменная `name` уникальна для каждого экземпляра `Person`, потому что методы создаются заново при каждом вызове конструктора. Как отмечено в главе 8 «Объекты, классы и объектно-ориентированное программирование», это один из недостатков, присущих паттерну Конструктор. Использование статических закрытых переменных с привилегированными методами позволяет пользователю решить эту проблему.

Статические закрытые переменные

Привилегированные методы можно также создавать для закрытых переменных или функций, определенных в закрытой области видимости:

```
(function() {

    // закрытые переменные и функции
    let privateVariable = 10;

    function privateFunction() {
        return false;
    }

    // конструктор
    MyObject = function() {
    };

    // открытые и привилегированные методы
    MyObject.prototype.publicMethod = function() {
        privateVariable++;
        return privateFunction();
    };

})();
```

В этом паттерне закрытая область видимости содержит конструктор и его методы. Первыми определяются закрытые переменные и функции, за ними следуют конструктор и открытые методы. Открытые методы определяются в прототипе, как и в обычном паттерне Прототип. Обратите внимание, что конструктор определяется в функции-выражении, а не в объявлении. При объявлении всегда создается локальная функция, что в данном случае нежелательно. По этой же причине перед `MyObject` не указано ключевое слово объявления переменной. Помните, что при инициализации необъявленной переменной всегда создается глобальная переменная, так что объект `MyObject` становится глобальным и доступным вне закрытой области видимости. Имейте также в виду, что присвоение значения необъявленной переменной в строгом режиме приводит к ошибке.

Основное различие между этим паттерном и предыдущим состоит в том, что в этот раз закрытые переменные и функции являются общими для всех экземпляров. Поскольку привилегированный метод определен в прототипе, все экземпляры используют одну и ту же функцию, а сам привилегированный метод как замыкание всегда содержит ссылку на область видимости функции-контейнера. Рассмотрим пример:

```
(function() {

    let name = "";

    Person = function(value) {
        name = value;
    };

    Person.prototype.getName = function() {
        return name;
    };

    Person.prototype.setName = function (value) {
        name = value;
    };
})();

let person1 = new Person("Nicholas");
console.log(person1.getName());    // "Nicholas"
person1.setName("Matt");
console.log(person1.getName());    // " Matt"

let person2 = new Person("Michael");
console.log(person1.getName());    // "Michael"
console.log(person2.getName());    // "Michael"
```

Здесь закрытая переменная `name` доступна конструктору `Person`, а также методам `getName()` и `setName()`, потому что в этом паттерне она становится статической и используется всеми экземплярами. Это означает, что вызов `setName()` для одного экземпляра влияет на все остальные экземпляры. При вызове `setName()` или создании экземпляра `Person` переменная `name` получает новое значение, которое после этого возвращается любым экземпляром.

Создание статических закрытых переменных обеспечивает возможность многократного использования кода с помощью прототипов, но при этом у отдельных экземпляров нет собственных закрытых переменных. В конечном счете выбор между закрытыми переменными экземпляра и статическими закрытыми переменными зависит от конкретных требований.

ПРИМЕЧАНИЕ Чем дальше в цепочке областей видимости находится переменная, тем сильнее замедляется поиск из-за использования замыканий и закрытых переменных.

Паттерн Модуль

Предыдущие паттерны создают закрытые переменные и привилегированные методы для пользовательских типов. *Паттерн Модуль* (module pattern), описанный Дугласом Крокфордом, делает то же самое для объектов-одиночек. *Одиночка* (singleton) — это объект, который может быть только единственным. Для создания одиночек в JavaScript традиционно используют нотацию литералов объектов, например:

```
let singleton = {
    name : value,
    method() {
        // код метода
    }
};
```

Паттерн Модуль расширяет объект-одиночку, позволяя использовать в нем закрытые переменные и привилегированные методы:

```
let singleton = function() {

    // закрытые переменные и функции
    let privateVariable = 10;

    function privateFunction() {
        return false;
    }

    // привилегированные/открытые методы и свойства
    return {

        publicProperty: true,

        publicMethod() {
            privateVariable++;
            return privateFunction();
        }

   };
}();
```

В паттерне Модуль используется анонимная функция, которая возвращает объект. Внутри этой функции сначала определяются закрытые переменные и функции, после чего возвращается литерал объекта, содержащий только открытые свойства и методы. Поскольку объект определен внутри анонимной функции, закрытые переменные и функции доступны всем открытым методам. По сути, литерал объекта определяет открытый интерфейс для объекта-одиночки. Это может быть полезно, если требуется инициализировать одиночку и обеспечить доступ к его закрытым переменным, например:

```
let application = function() {

    // закрытые переменные и функции
    let components = new Array();

    // инициализация
```

```
    components.push(new BaseComponent());

    // открытый интерфейс
    return {
        getComponentCount() {
            return components.length;
        },

        registerComponent(component) {
            if (typeof component == "object") {
                components.push(component);
            }
        }
    };
}();
```

В веб-приложениях объект-одиночку часто используют для управления данными уровня приложения. Так, объект-одиночка `application` из приведенного примера служит для управления компонентами. При создании объекта строится закрытый массив `components`, в который добавляется новый экземпляр `BaseComponent` (он используется исключительно для демонстрации инициализации, поэтому его код не важен). Доступ к массиву `components` осуществляется с помощью привилегированных методов `getComponentCount()` и `registerComponent()`. Первый из них просто возвращает количество зарегистрированных компонентов, а второй регистрирует новый компонент.

Паттерн Модуль полезен в ситуациях вроде этой, когда нужно создать один объект, инициализировать его некоторыми данными и предоставить открытые методы для доступа к его закрытым данным. Каждый объект-одиночка, создаваемый таким образом, является экземпляром типа `Object`, поскольку представляется литералом объекта. Это непоследовательно, потому что одиночки обычно используются глобально, а не передаются в функции как аргументы; соответственно, определять их тип с помощью оператора `instanceof` не требуется.

Расширенный паттерн Модуль

Альтернативный способ применения паттерна Модуль подразумевает расширение возможностей объекта перед его возвращением. Этот паттерн полезен, если объект-одиночка должен быть экземпляром конкретного типа, но в него требуется добавить свойства и (или) методы, например:

```
let singleton = function() {

    // закрытые переменные и функции
    let privateVariable = 10;

    function privateFunction() {
        return false;
    }

    // создание объекта
    let object = new CustomType();
```

```
    // добавление привилегированных/открытых свойств и методов
    object.publicProperty = true;

    object.publicMethod = function() {
        privateVariable++;
        return privateFunction();

    };

    // возвращение объекта
    return object;
}();
```

Если бы объект `application` в примере с паттерном Модуль должен был быть экземпляром `BaseComponent`, можно было бы использовать следующий код:

```
let application = function() {

    // закрытые переменные и функции
    let components = new Array();

    // инициализация
    components.push(new BaseComponent());

    // создание локальной копии application
    let app = new BaseComponent();

    // открытый интерфейс
    app.getComponentCount = function() {
        return components.length;
    };

    app.registerComponent = function(component) {
        if (typeof component == "object") {
            components.push(component);
        }
    };

    // возвращение объекта application
    return app;
}();
```

Как и в предыдущем примере, в этой версии объекта-одиночки `application` сначала определяются закрытые переменные. Ее главное отличие — создание локальной переменной `app` типа `BaseComponent`, которая в итоге станет объектом `application`. Затем в объект `app` добавляются открытые методы для доступа к закрытым переменным, после чего он возвращается и назначается переменной `application`.

ИТОГИ

Функции — это полезные и многогранные конструкции JavaScript. В ECMAScript 6 представлен мощный синтаксис, позволяющий использовать их еще более эффективно.

- В отличие от объявлений функций, функции-выражения не требуют указания имени функции. Поэтому такие выражения называют анонимными функциями.
- Появившиеся в ES6, стрелочные функции похожи на функции-выражения, но обладают некоторыми важными отличиями.
- Аргументы и параметры в функциях JavaScript очень гибки. Объект `arguments` вместе с новым оператором распространения в ES6 обеспечивают полностью динамическое определение и вызов.
- Функции предоставляют несколько объектов и ссылок, которые дают информацию о том, как была вызвана функция, где она была вызвана и что первоначально было передано ей.
- Движки оптимизируют функции с помощью хвостовых вызовов, чтобы сохранить пространство стека.
- Цепочка областей видимости замыкания содержит объекты переменных локальной функции, функции-контейнера и глобального контекста.
- Как правило, при завершении функции ее область видимости и все ее переменные уничтожаются.
- Если функция возвращает замыкание, ее область видимости остается в памяти, пока существует замыкание.
- Функция может быть создана и сразу же после создания вызвана с выполнением ее кода и без всякой ссылки на нее.
- В результате такого приема все переменные в функции уничтожаются, если явно не присвоить их переменным во внешней области видимости.
- Хотя JavaScript формально не поддерживает закрытые свойства объектов, с помощью замыканий можно реализовать открытые методы с доступом к переменным, определенным в закрытой области видимости.
- Открытые методы с доступом к закрытым переменным называются привилегированными.
- Привилегированные методы можно создавать для пользовательских типов с помощью паттерна Конструктор или Прототип, а также для объектов-одиночек, используя паттерн Модуль с обычными или расширенными возможностями.

11

Промисы и асинхронные функции

- Введение в асинхронное программирование
- Промисы
- Асинхронные функции

Загрузки с сайта wrox.com для этой главы

Обратите внимание, что все примеры кода для этой главы доступны на веб-сайте книги по ссылке www.wrox.com/go/projavascript4e на вкладке Downloads.

> **ПРИМЕЧАНИЕ** В этой главе в примерах широко используется асинхронное ведение журнала `setTimeout(console.log, 0, ... params)` для демонстрации порядка работы и других характеристик асинхронного поведения. Вывод журнала будет отображаться так, как если бы он печатался синхронно, тогда как фактически он печатался асинхронно. Это делается для того, чтобы такие значения, как промисы, приняли свое окончательное состояние.
>
> Кроме того, вывод консоли браузера часто выводит информацию об объектах, которые иначе недоступны для среды выполнения JavaScript (например, состояние промиса). Эта функция широко используется в примерах по всей главе, чтобы помочь улучшить понимание концепции читателями.

В редакциях ECMAScript, начиная с ES6, поддержка и инструментарий для асинхронного поведения претерпели ренессанс. ECMAScript 6 представляет формальный

ссылочный тип `Promise`, позволяющий элегантно определять и организовывать асинхронное поведение. Более поздние выпуски также расширили язык для поддержки асинхронных функций с ключевыми словами `async` и `await`.

ВВЕДЕНИЕ В АСИНХРОННОЕ ПРОГРАММИРОВАНИЕ

Двойственность между синхронным и асинхронным поведением является фундаментальной концепцией в computer science, особенно в однопоточной модели цикла событий, такой как JavaScript. Асинхронное поведение обусловлено необходимостью оптимизации для более высокой вычислительной пропускной способности в условиях операций с высокой задержкой. Это прагматично, если возможно выполнение других инструкций во время завершения вычислений и при этом поддерживается стабильное состояние системы.

Важно отметить, что асинхронная операция необязательно является вычислительной операцией или операцией с высокой задержкой. Ее можно использовать везде, где нет смысла блокировать поток выполнения, чтобы дождаться возникновения асинхронного поведения.

Синхронный и асинхронный JavaScript

Синхронное поведение аналогично последовательным инструкциям процессора в памяти. Каждая инструкция выполняется строго в том порядке, в котором она появляется, и каждая из них также способна немедленно извлекать информацию, которая хранится локально в системе (например, в регистре процессора или в системной памяти). В результате можно легко определить состояние программы (например, значение переменной) в любой заданной точке кода.

Тривиальным примером этого будет выполнение простой арифметической операции:

```
let x = 3;
x = x + 4;
```

На каждом шаге этой программы можно рассуждать о ее состоянии, потому что выполнение не будет продолжено, пока не будет выполнена предыдущая инструкция. Когда последняя инструкция завершается, вычисленное значение `x` сразу становится доступным для использования.

Этот фрагмент JS-кода легко разложить по полочкам, потому что нетрудно предвидеть, к каким низкоуровневым инструкциям он будет компилироваться (например, от JavaScript до x86). Предположительно, операционная система выделит некоторое количество памяти для числа с плавающей точкой в стеке, выполнит арифметическую операцию с этим значением и запишет результат в выделенную память. Все эти инструкции располагаются последовательно внутри одного потока выполнения. В каждой точке скомпилированной низкоуровневой программы можно с большой долей уверенности утверждать, что можно и что нельзя знать о состоянии системы.

И наоборот, *асинхронное поведение* аналогично прерываниям, когда объект, внешний по отношению к текущему процессу, может инициировать выполнение кода. Часто требуется асинхронная операция, потому что невозможно заставить процесс долго ждать завершения операции (как в случае синхронной операции). Это длительное ожидание может возникнуть из-за того, что код обращается к ресурсу с высокой задержкой, например, отправляет запрос на удаленный сервер и ожидает ответа.

Тривиальным примером JavaScript в этом случае будет выполнение арифметической операции за время ожидания:

```
let x = 3;
setTimeout(() => x = x + 4, 1000);
```

Эта программа в конечном итоге выполняет ту же работу, что и синхронная — складывая два числа вместе, — но этот поток выполнения не может точно знать, когда изменится значение `x`, потому что это зависит от того, когда обратный вызов будет исключен из очереди сообщений и выполнен.

Этот код не так легко разложить по полочкам. Хотя низкоуровневые инструкции, используемые в этом примере, в конечном итоге выполняют ту же работу, что и предыдущий пример, второй блок инструкций (операция сложения и назначение) запускается системным таймером, который генерирует прерывание для постановки в очередь на выполнение. В тот момент, когда прерывание будет запущено, это станет черным ящиком для среды выполнения JavaScript, поэтому невозможно точно знать, когда именно произойдет прерывание (хотя оно гарантированно произойдет *после* завершения текущего потока синхронного выполнения, поскольку обратный вызов еще не был снят с выполнения и утилизирован). Тем не менее обычно нельзя утверждать, когда именно состояние системы изменится после запланирования обратного вызова.

Чтобы значение `x` стало полезным, эта асинхронно выполняемая функция должна сообщить остальной части программы, что она обновила значение `x`. Однако если программе не нужно это значение, тогда она может продолжить и выполнять другую работу вместо ожидания результата.

Разработать систему, которая будет знать, когда можно прочитать значение `x`, на удивление сложно. Реализации такой системы в JavaScript прошли несколько итераций.

Устаревшие паттерны асинхронного программирования

Асинхронное поведение долгое время было важным, но ужасным краеугольным камнем JavaScript. В ранних версиях языка асинхронная операция поддерживала только определение функции обратного вызова для указания, что асинхронная операция завершена. Сериализация асинхронного поведения была распространенной проблемой, обычно решаемой с помощью кодовой базы, полной вложенных функций обратного вызова, в миру называемой «адом обратных вызовов».

Предположим, вы работали со следующей асинхронной функцией, которая использует `setTimeout` для выполнения некоторого поведения через одну секунду:

```
function double(value) {
    setTimeout(() => setTimeout(console.log, 0, value * 2), 1000);
}

double(3);
// 6 (выводится примерно спустя 1000 мс)
```

Здесь не происходит ничего таинственного, но важно точно понять, почему эта функция асинхронна. `setTimeout` позволяет определить обратный вызов, который планируется выполнить по истечении заданного промежутка времени. Спустя 1000 мс во время выполнения JavaScript запланирует обратный вызов, поместив его в очередь сообщений JavaScript. Этот обратный вызов снимается и выполняется способом, который полностью невидим для кода JavaScript. Более того, функция `double()` завершается сразу после успешного выполнения операции планирования `setTimeout`.

Возврат асинхронных значений

Предположим, операция `setTimeout` вернула полезное значение. Как лучше всего вернуть значение туда, где оно необходимо? Широко используемая стратегия заключается в предоставлении обратного вызова для асинхронной операции, где обратный вызов содержит код, требующий доступ к вычисленному значению (предоставляется в качестве параметра). Это выглядит следующим образом:

```
function double(value, callback) {
    setTimeout(() => callback(value * 2), 1000);
}

double(3, (x) => console.log(`I was given: ${x}`));
// I was given: 6 (выводится примерно спустя 1000 мс)
```

Здесь при вызове `setTimeout` команда помещает функцию в очередь сообщений по истечении 1000 мс. Эта функция будет удалена и асинхронно вычислена средой выполнения. Функция обратного вызова и ее параметры все еще доступны в асинхронном исполнении через замыкание функции.

Обработка ошибок

Вероятность сбоя также должна быть включена в эту модель обратного вызова, так что обычно она принимает форму обратного вызова в случае успеха и неудачи:

```
function double(value, success, failure) {
    setTimeout(() => {
        try {
            if (typeof value !== 'number') {
                throw 'Must provide number as first argument';
            }
```

```
            success(2 * value);
        } catch (e) {
            failure(e);
        }
    }, 1000);
}

const successCallback = (x) => console.log(`Success: ${x}`);
const failureCallback = (e) => console.log(`Failure: ${e}`);

double(3, successCallback, failureCallback);
double('b', successCallback, failureCallback);

// Success: 6 (выводится примерно спустя 1000 мс)
// Failure: Must provide number as first argument
// (выводится примерно спустя 1000 мс)
```

Этот формат уже нежелателен, так как обратные вызовы должны быть определены при инициализации асинхронной операции. Значение, возвращаемое из асинхронной функции, является временным, и поэтому только обратные вызовы, которые готовы принять это временное значение в качестве параметра, могут получить к нему доступ.

Вложенные асинхронные обратные вызовы

Ситуация с обратными вызовами еще более усложняется, когда доступ к асинхронным значениям зависит от других асинхронных значений. В мире обратных вызовов это требует вложения обратных вызовов:

```
function double(value, success, failure) {
    setTimeout(() => {
        try {
            if (typeof value !== 'number') {
                throw 'Must provide number as first argument';
            }
            success(2 * value);
        } catch (e) {
            failure(e);
        }
    }, 1000);
}

const successCallback = (x) => {
    double(x, (y) => console.log(`Success: ${y}`));
};
const failureCallback = (e) => console.log(`Failure: ${e}`);

double(3, successCallback, failureCallback);

// Success: 12 (выводится примерно спустя 1000 мс)
```

Неудивительно, что эта стратегия обратного вызова плохо масштабируется по мере роста сложности кода. Выражение «ад обратных вызовов» вполне заслужено, так

как кодовые базы JavaScript, которые были поражены такой структурой, стали почти не поддерживаемыми.

ПРОМИСЫ

Промис — это суррогатная сущность, которая выступает в качестве замены для результата, который еще не существует. Термин «промис» был впервые предложен Дэниелом Фридманом и Дэвидом Уайзом в их статье 1976 г. «*Влияние прикладного программирования на многопроцессорность* (The Impact of Applicative Programming on Multiprocessing)», но концептуальное поведение промиса было формализовано лишь десятилетие спустя Барбарой Лисков и Любой Шрира в их статье 1988 г. «*Промисы: лингвистическая поддержка эффективных асинхронных процедурных вызовов в распределенных системах* (Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems)». Современные компьютерные ученые описали похожие понятия, такие как «возможное», «будущее», «задержка» или «отсроченное»; все они описаны в той или иной форме программным инструментом для синхронизации выполнения программы.

Спецификация Promises/A+

Ранние формы промисов появились в jQuery и Dojo Deferred API, а в 2010 г. растущая популярность привела к появлению спецификации Promises/A внутри проекта CommonJS. Сторонние библиотеки промисов JavaScript, такие как Q и Bluebird, продолжали завоевывать популярность, но каждая реализация немного отличалась от предыдущей. Чтобы устранить разногласия в пространстве промисов, в 2012 г. организация Promises/A + разветвила предложение CommonJS Promises/A и создала одноименную спецификацию промисов Promises/A+ (https://promisesaplus.com/). Эта спецификация в конечном итоге определит, как промисы будут реализованы в спецификации ECMAScript 6.

ECMAScript 6 представил первоклассную реализацию совместимого с Promise/A+ типа Promise. За время, прошедшее с момента его введения, промисы пользовались невероятно высоким уровнем поддержки. Все современные браузеры полностью поддерживают тип промисов ES6, и несколько API-интерфейсов браузера, таких как `fetch()` и Battery API, используют исключительно его.

Основы промисов

Начиная с ECMAScript 6, `Promise` является поддерживаемым ссылочным типом и может быть создан с помощью оператора `new`. Для этого требуется передать параметр функции исполнителя (описанный в следующем разделе), который здесь является пустым объектом функции, чтобы угодить интерпретатору:

```
let p = new Promise(() => {});
setTimeout(console.log, 0, p); // Promise <pending>
```

Если функция исполнителя не предусмотрена, будет сгенерирована ошибка `SyntaxError`.

Машина состояний промисов

При передаче экземпляра промиса в `console.log` выводы консоли (которые могут различаться в разных браузерах) указывают, что этот экземпляр промиса находится в состоянии *ожидания*. Как упоминалось ранее, промис — это объект с состоянием, который может существовать в одном из трех состояний:

- *в ожидании* (Pending);
- *выполнен* (иногда также называется *решенным*) (Fulfilled);
- *отклонен* (Rejected).

Состояние ожидания — это начальное состояние, с которого начинается промис. Из состояния ожидания промис может быть *установлен* путем перехода в *выполненное* состояние, указывающее на успех, или *отклоненное*, указывающее на отказ. Этот переход к *установленному* состоянию необратим; как только происходит переход к *выполненному* или *отклоненному* состоянию, состояние промиса уже не сможет измениться. Кроме того, не гарантируется, что промис когда-либо покинет состояние *ожидания*. Следовательно, хорошо структурированный код должен вести себя правильно, если промис успешно разрешается, если он отклоняется или никогда не выходит из состояния ожидания.

Важно отметить, что состояние промиса является частным и не может быть напрямую проверено в JavaScript. Причина этого заключается прежде всего в том, чтобы предотвратить синхронную программную обработку объекта промиса на основе его состояния при чтении. Кроме того, состояние промиса не может быть изменено внешним JS-кодом по той же причине, по которой состояние не может быть прочитано: промис намеренно инкапсулирует блок асинхронного поведения, а внешний код, выполняющий синхронное определение его состояния, противоречит его цели.

Разрешенные значения, причины отказа и полезность промисов

Существуют две основные причины, по которым конструкция промисов полезна. Первая — это абстрактное представление блока асинхронного выполнения. Состояние промиса указывает на то, должен ли промис подождать с завершением выполнения. Состояние *ожидания* указывает, что выполнение еще не началось или все еще выполняется. *Выполненное* состояние является неспецифическим индикатором того, что выполнение успешно завершено. *Отклоненное* состояние является неспецифическим индикатором того, что выполнение не завершилось успешно.

В некоторых случаях внутренней машиной состояний является вся полезность, которую промис должен предоставить: одного лишь знания о том, что кусок асинхронного кода завершен, достаточно для информирования о ходе выполнения программы. Например, предположим, что промис отправляет HTTP-запрос на

сервер. Запрос, возвращающийся со статусом не 200–299, может быть достаточным для перехода состояния обещания в *выполненное*. Точно так же запрос, возвращающийся со статусом, который *не* является 200–299, перевел бы состояние промиса в *отклоненное*.

В других случаях асинхронное выполнение, которое оборачивает промис, фактически генерирует значение, и поток программы будет ожидать, что это значение будет доступно, когда промис изменит состояние. С другой стороны, если промис отклоняется, поток программы ожидает причину отклонения после изменения состояния промиса. Например, предположим, что промис отправляет HTTP-запрос на сервер и ожидает его возврата в формате JSON. Запрос, возвращающийся со статусом 200–299, может быть достаточным для перевода промиса в *выполненное* состояние, и JSON-строка будет доступна внутри промиса. Точно так же запрос, возвращаемый со статусом, который *не* является 200–299, перевел бы состояние промиса в *отклоненное*, и причиной отклонения может быть объект `Error`, содержащий текст, сопровождающий HTTP-код статуса.

Для поддержки этих двух вариантов использования каждый промис, который переходит в *выполненное* состояние, имеет закрытое внутреннее *значение*. Точно так же каждый промис, который переходит в *отклоненное* состояние, имеет закрытую внутреннюю *причину*. И *значение*, и *причина* являются неизменной ссылкой на примитив или объект. Оба являются необязательными и по умолчанию будут иметь значение `undefined`. Асинхронный код, который планируется выполнить после того, как промис достигает определенного *установленного* состояния, всегда снабжается *значением* или *причиной*.

Контроль состояния промиса с помощью исполнителя

Поскольку состояние промиса является закрытым, им можно манипулировать только изнутри. Эта внутренняя манипуляция выполняется внутри функции-исполнителя промиса. Функция-исполнитель выполняет две основные обязанности: инициализирует асинхронное поведение промиса и контролирует любой возможный переход состояния. Управление переходом между состояниями осуществляется путем вызова одного из двух параметров функции, которые обычно называются `resolve` и `reject`. Вызов `resolve` изменит состояние на *выполненное*; вызов `reject` изменит состояние на *отклоненное*. Вызов `rejected()` также сгенерирует ошибку (это поведение ошибки будет рассмотрено позже).

```
let p1 = new Promise((resolve, reject) => resolve());
setTimeout(console.log, 0, p1); // Promise <resolved>

let p2 = new Promise((resolve, reject) => reject());
setTimeout(console.log, 0, p2); // Promise <rejected>
// Uncaught error (in promise)
```

В предыдущем примере асинхронное поведение на самом деле не происходит, потому что состояние каждого промиса уже изменяется к моменту выхода из функции-исполнителя. Важно отметить, что функция-исполнитель будет выполняться

синхронно, так как она действует как инициализатор для промиса. Этот порядок выполнения демонстрируется здесь:

```
new Promise(() => setTimeout(console.log, 0, 'executor'));
setTimeout(console.log, 0, 'promise initialized');

// executor
// promise initialized
```

Можно отложить переход состояния, добавив `setTimeout`:

```
let p = new Promise((resolve, reject) => setTimeout(resolve, 1000));

// При выполнении console.log обратный вызов тайм-аута еще не будет выполнен:
setTimeout(console.log, 0, p); // Promise <pending>
```

После вызова `resolve` или `reject` переход состояния не может быть отменен. Попытки дальнейшего изменения состояния будут молча игнорироваться:

```
let p = new Promise((resolve, reject) => {
    resolve();
    reject(); // Безрезультатно
});

setTimeout(console.log, 0, p); // Promise <resolved>
```

Вы можете избежать зависания промиса в состоянии ожидания, добавив запланированное поведение выхода. Например, можно установить тайм-аут, чтобы отклонить промис через 10 секунд:

```
let p = new Promise((resolve, reject) => {
    setTimeout(reject, 10000);        // Вызов reject() спустя 10 секунд
    // Код исполнителя
});

setTimeout(console.log, 0, p);        // Promise <pending>
setTimeout(console.log, 11000, p);    // Проверка состояния спустя 11 секунд

// (Спустя 10 секунд) Uncaught error
// (Спустя 11 секунд) Promise <rejected>
```

Поскольку промис может изменить состояние только один раз, такое поведение тайм-аута позволяет безопасно устанавливать максимальное количество времени, в течение которого промис может оставаться в состоянии ожидания. Если код внутри исполнителя должен был разрешить или отклонить промис до истечения времени ожидания, попытка обработчика времени ожидания отклонить промис будет игнорироваться.

Преобразование промисов с помощью Promise.resolve()

Промис не обязательно должен начинаться с состояния ожидания и использовать функцию-исполнитель для достижения установленного состояния. Можно создать

экземпляр промиса в состоянии «разрешено», вызвав статический метод `Promise.resolve()`. Следующие два экземпляра промисов фактически эквивалентны:

```
let p1 = new Promise((resolve, reject) => resolve());
let p2 = Promise.resolve();
```

Значение этого разрешенного обещания станет первым аргументом, переданным `Promise.resolve()`. Это позволяет эффективно «преобразовать» любое значение в промис:

```
setTimeout(console.log, 0, Promise.resolve());
// Promise <resolved>: undefined

setTimeout(console.log, 0, Promise.resolve(3));
// Promise <resolved>: 3

// Дополнительные аргументы игнорируются
setTimeout(console.log, 0, Promise.resolve(4, 5, 6));
// Promise <resolved>: 4
```

Возможно, наиболее важным аспектом этого статического метода является его способность действовать как переход, когда аргумент уже является промисом. В результате `Promise.resolve()` является идемпотентным методом, как показано здесь:

```
let p = Promise.resolve(7);

setTimeout(console.log, 0, p === Promise.resolve(p));
// true

setTimeout(console.log, 0, p === Promise.resolve(Promise.resolve(p)));
// true
```

Эта идемпотентность будет учитывать состояние промиса, переданного ему:

```
let p = new Promise(() => {});

setTimeout(console.log, 0, p);                      // Promise <pending>
setTimeout(console.log, 0, Promise.resolve(p));     // Promise <pending>

setTimeout(console.log, 0, p === Promise.resolve(p)); // true
```

Помните, что этот статический метод с радостью обернет любой не-промис, включая объект ошибки, как разрешенный промис, что может привести к непреднамеренному поведению:

```
let p = Promise.resolve(new Error('foo'));

setTimeout(console.log, 0, p);
// Promise <resolved>: Error: foo
```

Отклонение промисов с помощью Promise.reject()

В принципе, аналогично `Promise.resolve()`, `Promise.reject()` создает отклоненный промис и генерирует асинхронную ошибку (которая не будет перехвачена `try/`

`catch` и может быть перехвачена только обработчиком отклонения). Следующие два экземпляра промисов фактически эквивалентны:

```
let p1 = new Promise((resolve, reject) => reject());
let p2 = Promise.reject();
```

Поле «причины» этого разрешенного промиса будет первым аргументом, переданным `Promise.reject()`. Оно также будет ошибкой, переданной обработчику отклонения:

```
let p = Promise.reject(3);
setTimeout(console.log, 0, p); // Promise <rejected>: 3

p.then(null, (e) => setTimeout(console.log, 0, e)); // 3
```

Важно отметить, что `Promise.reject()` не отражает поведение `Promise.resolve()` в отношении идемпотентности. Если объект промиса передан, он с радостью использует этот промис в качестве поля «причины» отклоненного промиса:

```
setTimeout(console.log, 0, Promise.reject(Promise.resolve()));
// Promise <rejected>: Promise <resolved>
```

Двойственность синхронного/асинхронного выполнения

Большая часть конструкции `Promise` заключается в создании совершенно отдельного режима вычислений в JavaScript. Это аккуратно инкапсулировано в следующем примере, который выдает ошибки двумя различными способами:

```
try {
    throw new Error('foo');
} catch(e) {
    console.log(e); // Error: foo
}

try {
    Promise.reject(new Error('bar'));
} catch(e) {
    console.log(e);
}

// Uncaught (in promise) Error: bar
```

Первый блок `try/catch` генерирует ошибку и затем перехватывает ее, но второй блок `try/catch` генерирует ошибку, которая *не* перехватывается. Это может показаться нелогичным, поскольку кажется, что код синхронно создает отклоненный экземпляр промиса, который затем выдает ошибку при отклонении. Однако причина, по которой второй промис не был перехвачен, заключается в том, что код не пытается перехватить ошибку в соответствующем «асинхронном режиме». Такое поведение подчеркивает, как на самом деле ведут себя промисы: это синхронные объекты, используемые в синхронном режиме выполнения, выступающие в качестве моста к *асинхронному* режиму выполнения.

В предыдущем примере ошибка от отклоненного промиса выдается не в потоке синхронного выполнения, а в результате асинхронного выполнения очереди сообщений в браузере. Следовательно, инкапсулирующего блока `try/catch` будет недостаточно, чтобы перехватить эту ошибку. Как только код начинает выполняться в этом асинхронном режиме, единственный способ взаимодействовать с ним — использование конструкций асинхронного режима, а именно методы промисов.

Методы экземпляра промиса

Методы, представленные в экземпляре промиса, служат для устранения разрыва между синхронным путем внешнего кода и асинхронным внутренним путем кода. Эти методы могут использоваться для доступа к данным, возвращаемым из асинхронной операции, обработки результатов успешного и неудачного выполнения промиса, последовательного вычисления промисов или добавления функций, которые выполняются только после того, как промис входит в состояние завершения.

Реализация интерфейса Thenable

Для целей асинхронных конструкций ECMAScript считается, что любой объект, который предоставляет метод `then()`, реализует интерфейс `Thenable`. Ниже приведен пример простейшего класса, реализующего этот интерфейс:

```
class MyThenable {
    then() {}
}
```

Тип `Promise` в ECMAScript реализует интерфейс `Thenable`. Этот упрощенный интерфейс не следует путать с другими интерфейсами или определениями типов в таких пакетах, как TypeScript, которые представляют собой гораздо более конкретную форму интерфейса `Thenable`.

> **ПРИМЕЧАНИЕ** Полезность и назначение интерфейса `Thenable` вновь рассматриваются далее в разделе «Асинхронные функции».

Promise.prototype.then()

Метод `Promise.prototype.then()` является основным методом, который используется для подключения обработчиков к экземпляру промиса. Метод `then()` принимает до двух аргументов: необязательную функцию-обработчик `onResolved` и необязательную функцию-обработчик `onRejected`. Каждая из них будет выполнена только тогда, когда промис, по которому они определены, достигнет соответствующего состояния «выполнен» или «отклонен».

```
function onResolved(id) {
    setTimeout(console.log, 0, id, 'resolved');
}
```

```
function onRejected(id) {
    setTimeout(console.log, 0, id, 'rejected');
}

let p1 = new Promise((resolve, reject) => setTimeout(resolve, 3000));
let p2 = new Promise((resolve, reject) => setTimeout(reject, 3000));

p1.then(() => onResolved('p1'),
        () => onRejected('p1'));
p2.then(() => onResolved('p2'),
        () => onRejected('p2'));

// (спустя 3 с)
// p1 resolved
// p2 rejected
```

Поскольку промис может переходить в конечное состояние только один раз, гарантируется, что выполнение этих обработчиков является взаимоисключающим.

Как описано ранее, оба аргумента обработчика являются полностью необязательными. Любой нефункциональный тип, предоставленный в качестве аргумента `then()`, будет игнорироваться. Если нужно явно передать только обработчик `onRejected`, каноническим выбором будет передача `undefined` в качестве аргумента `onResolved`. Это позволит избежать создания временного объекта в памяти, который будет игнорироваться интерпретатором, и также порадует системы, которые ожидают наличие необязательного функционального объекта в качестве аргумента.

```
function onResolved(id) {
    setTimeout(console.log, 0, id, 'resolved');
}
function onRejected(id) {
    setTimeout(console.log, 0, id, 'rejected');
}

let p1 = new Promise((resolve, reject) => setTimeout(resolve, 3000));
let p2 = new Promise((resolve, reject) => setTimeout(reject, 3000));

// Нефункциональные обработчики игнорируются, и их использование не рекомендуется
p1.then('gobbeltygook');

// Каноническая форма явного пропуска обработчика onResolved
p2.then(null, () => onRejected('p2'));

// p2 rejected (спустя 3 с)
```

Метод `Promise.prototype.then()` возвращает новый экземпляр промиса:

```
let p1 = new Promise(() => {});
let p2 = p1.then();
setTimeout(console.log, 0, p1);           // Promise <pending>
setTimeout(console.log, 0, p2);           // Promise <pending>
setTimeout(console.log, 0, p1 === p2);    // false
```

Этот новый экземпляр промиса получен из возвращаемого значения обработчика `onResolved`. Возвращаемое значение обработчика помещается в `Promise.resolve()`

для генерации нового промиса. Если никакой функции-обработчика не предоставлено, метод действует как проход для разрешенного значения исходного промиса. Если явного возвращаемого выражения нет, возвращаемое значение записано как `undefined` и обернуто в `Promise.resolve()`.

```
let p1 = Promise.resolve('foo');

// Вызов then() без функций-обработчиков сработает как проход
let p2 = p1.then();

setTimeout(console.log, 0, p2);    // Promise <resolved>: foo

// Эти варианты эквивалентны
let p3 = p1.then(() => undefined);
let p4 = p1.then(() => {});
let p5 = p1.then(() => Promise.resolve());

setTimeout(console.log, 0, p3);    // Promise <resolved>: undefined
setTimeout(console.log, 0, p4);    // Promise <resolved>: undefined
setTimeout(console.log, 0, p5);    // Promise <resolved>: undefined
```

Явные возвращаемые значения обернуты в `Promise.resolve()`:

```
...

// Эти варианты эквивалентны:
let p6 = p1.then(() => 'bar');
let p7 = p1.then(() => Promise.resolve('bar'));

setTimeout(console.log, 0, p6);    // Promise <resolved>: bar
setTimeout(console.log, 0, p7);    // Promise <resolved>: bar

// Promise.resolve() сохраняет возвращенный промис
let p8 = p1.then(() => new Promise(() => {}));
let p9 = p1.then(() => Promise.reject());
// Uncaught (in promise): undefined

setTimeout(console.log, 0, p8);    // Promise <pending>
setTimeout(console.log, 0, p9);    // Promise <rejected>: undefined
```

Генерация ошибки вернет отклоненный промис:

```
...

let p10 = p1.then(() => { throw 'baz'; });
// Uncaught (in promise) baz

setTimeout(console.log, 0, p10);    // Promise <rejected> baz
```

Важно отметить, что возвращение ошибки не приведет к тому же поведению отклонения, а вместо этого обернет объект ошибки в разрешенный промис:

```
...

let p11 = p1.then(() => Error('qux'));

setTimeout(console.log, 0, p11);    // Promise <resolved>: Error: qux
```

Обработчик `onRejected` ведет себя таким же образом: значения, возвращаемые из обработчика `onRejected`, переносятся в `Promise.resolve()`. Поначалу это может показаться нелогичным, но обработчик `onRejected` выполняет свою работу по обнаружению асинхронной ошибки. Следовательно, этот обработчик отклонения, завершающий выполнение без дополнительной ошибки, следует рассматривать как ожидаемое поведение промиса и возвращать разрешенный промис.

Следующий фрагмент кода является аналогом `Promise.reject()` для предыдущих примеров с использованием `Promise.resolve()`:

```
let p1 = Promise.reject('foo');

// Вызов then() без функций-обработчиков сработает как проход
let p2 = p1.then();
// Uncaught (in promise) foo

setTimeout(console.log, 0, p2);    // Promise <rejected>: foo

// Эти варианты эквивалентны
let p3 = p1.then(null, () => undefined);
let p4 = p1.then(null, () => {});
let p5 = p1.then(null, () => Promise.resolve());

setTimeout(console.log, 0, p3);    // Promise <resolved>: undefined
setTimeout(console.log, 0, p4);    // Promise <resolved>: undefined
setTimeout(console.log, 0, p5);    // Promise <resolved>: undefined

// Эти варианты эквивалентны
let p6 = p1.then(null, () => 'bar');
let p7 = p1.then(null, () => Promise.resolve('bar'));

setTimeout(console.log, 0, p6);    // Promise <resolved>: bar
setTimeout(console.log, 0, p7);    // Promise <resolved>: bar

// Promise.resolve() сохраняет возвращенный промис
let p8 = p1.then(null, () => new Promise(() => {}));
let p9 = p1.then(null, () => Promise.reject());
// Uncaught (in promise): undefined

setTimeout(console.log, 0, p8);    // Promise <pending>
setTimeout(console.log, 0, p9);    // Promise <rejected>: undefined

let p10 = p1.then(null, () => { throw 'baz'; });
// Uncaught (in promise) baz

setTimeout(console.log, 0, p10);    // Promise <rejected>: baz

let p11 = p1.then(null, () => Error('qux'));

setTimeout(console.log, 0, p11);    // Promise <resolved>: Error: qux
```

Promise.prototype.catch()

Метод `Promise.prototype.catch()` можно использовать для присоединения только обработчика отклонения к промису. Он принимает только один

аргумент — функцию-обработчик `onRejected`. Метод является не более чем синтаксическим сахаром и ничем не отличается от использования `Promise.prototype.then (null, onRejected)`.

Следующий код демонстрирует эту эквивалентность:

```
let p = Promise.reject();
let onRejected = function(e) {
    setTimeout(console.log, 0, 'rejected');
};

// Эти обработчики отклонений эквивалентны:
p.then(null, onRejected);    // rejected
p.catch(onRejected);         // rejected
```

Метод `Promise.prototype.catch()` возвращает новый экземпляр промиса:

```
let p1 = new Promise(() => {});
let p2 = p1.catch();
setTimeout(console.log, 0, p1);           // Promise <pending>
setTimeout(console.log, 0, p2);           // Promise <pending>
setTimeout(console.log, 0, p1 === p2);    // false
```

Что касается создания нового экземпляра промиса, `Promise.prototype.catch()` ведет себя идентично обработчику `onRejected` для `Promise.prototype.then()`.

Promise.prototype.finally()

Метод `Promise.protoype.finally()` можно использовать для подключения обработчика `onFinally`, который выполняется, когда промис достигает разрешенного или отклоненного состояния. Это полезно для избежания дублирования кода между обработчиками `onResolved` и `onRejected`. Важно отметить, что обработчик не имеет никакого способа определить, был ли промис разрешен или отклонен, поэтому этот метод предназначен только для вещей вроде очистки памяти.

```
let p1 = Promise.resolve();
let p2 = Promise.reject();
let onFinally = function() {
    setTimeout(console.log, 0, 'Finally!')
}

p1.finally(onFinally);    // Finally
p2.finally(onFinally);    // Finally
```

Метод `Promise.prototype.finally()` возвращает новый экземпляр промиса:

```
let p1 = new Promise(() => {});
let p2 = p1.finally();
setTimeout(console.log, 0, p1);           // Promise <pending>
setTimeout(console.log, 0, p2);           // Promise <pending>
setTimeout(console.log, 0, p1 === p2);    // false
```

Этот новый экземпляр промиса получен не так, как из `then()` или `catch()`. Поскольку `onFinally` предназначен для использования, независимого от состояния метода,

в большинстве случаев он будет действовать как проход для родительского промиса. Это верно как для разрешенных, так и отклоненных состояний.

```
let p1 = Promise.resolve('foo');

// Все это сработает как проход
let p2 = p1.finally();
let p3 = p1.finally(() => undefined);
let p4 = p1.finally(() => {});
let p5 = p1.finally(() => Promise.resolve());
let p6 = p1.finally(() => 'bar');
let p7 = p1.finally(() => Promise.resolve('bar'));
let p8 = p1.finally(() => Error('qux'));

setTimeout(console.log, 0, p2);    // Promise <resolved>: foo
setTimeout(console.log, 0, p3);    // Promise <resolved>: foo
setTimeout(console.log, 0, p4);    // Promise <resolved>: foo
setTimeout(console.log, 0, p5);    // Promise <resolved>: foo
setTimeout(console.log, 0, p6);    // Promise <resolved>: foo
setTimeout(console.log, 0, p7);    // Promise <resolved>: foo
setTimeout(console.log, 0, p8);    // Promise <resolved>: foo
```

Единственные исключения из этого правила — когда он возвращает ожидающий промис или выдает ошибку (через явную генерацию или возврат отклоненного промиса). В этих случаях соответствующий промис возвращается (ожидающий или отклоненный), как показано здесь:

```
...

// Promise.resolve() сохраняет возвращенный промис
let p9 = p1.finally(() => new Promise(() => {}));
let p10 = p1.finally(() => Promise.reject());
// Uncaught (in promise): undefined

setTimeout(console.log, 0, p9);     // Promise <pending>
setTimeout(console.log, 0, p10);    // Promise <rejected>: undefined

let p11 = p1.finally(() => { throw 'baz';});
// Uncaught (in promise) baz

setTimeout(console.log, 0, p11);    // Promise <rejected>: baz
```

Возврат ожидающего промиса — необычный случай, так как после разрешения промиса новый промис все равно будет действовать как проход для первоначального:

```
let p1 = Promise.resolve('foo');

// Возвращаемое значение игнорируется
let p2 = p1.finally(
    () => new Promise((resolve, reject) => setTimeout(() => resolve('bar'), 100)));
setTimeout(console.log, 0, p2);    // Promise <pending>

setTimeout(() => setTimeout(console.log, 0, p2), 200);

// Спустя 200 мс:
// Promise <resolved>: foo
```

Невозвратные методы промисов

Когда промис достигает установленного состояния, выполнение обработчиков, связанных с этим состоянием, просто будет *запланировано*, а не выполнено немедленно. Синхронный код, следующий за вложением обработчика, гарантированно будет выполнен до его вызова. Это остается верным, даже если промис уже существует в состоянии, с которым связан новый присоединенный обработчик. Это свойство, называемое невозвратностью, гарантируется средой выполнения JavaScript. Следующий простой пример демонстрирует это свойство:

```
// Создание разрешенного промиса
let p = Promise.resolve();

// Добавление обработчика разрешенного состояния.
// Интуитивно понятно, что это будет выполнено как можно скорее,
// поскольку p уже разрешен.
p.then(() => console.log('onResolved handler'));

// Синхронная запись, указывающая, что then() вернулся
console.log('then() returns');

// Действительный вывод:
// then() returns
// onResolved handler
```

В этом примере вызов `then()` для разрешенного промиса помещает обработчик `onResolved` в очередь сообщений. Этот обработчик не будет выполняться до тех пор, пока он не будет удален из системы после завершения текущего потока выполнения. Следовательно, синхронный код, следующий непосредственно за `then()`, гарантированно будет выполнен перед обработчиком.

Обратное поведение в этом сценарии дает тот же результат. Если обработчики уже присоединены к промису, который позже синхронно поменяет состояние, выполнение обработчика не повторяется при этом изменении состояния. Следующий пример демонстрирует, как, даже с уже подключенным обработчиком `onResolved`, синхронный вызов `resolve()` будет по-прежнему демонстрировать невозвратное поведение:

```
let synchronousResolve;

// Создание промиса и сохранение функции разрешения в локальной переменной.
let p = new Promise((resolve) => {
    synchronousResolve = function() {
        console.log('1: invoking resolve()');
        resolve();
        console.log('2: resolve() returns');
    };
});

p.then(() => console.log('4: then() handler executes'));

synchronousResolve();
console.log('3: synchronousResolve() returns');
```

```
// Действительный вывод:
// 1: invoking resolve()
// 2: resolve() returns
// 3: synchronousResolve() returns
// 4: then() handler executes
```

В этом примере, даже если состояние промиса изменяется синхронно с обработчиками, подключенными к этому состоянию, выполнение обработчика все равно не будет начато до тех пор, пока он не будет удален из очереди сообщений среды выполнения.

Невозвратность гарантируется как для обработчиков `onResolved`, так и для `onRejected`, обработчиков `catch()` и `finally()`. Все они демонстрируются в этом примере:

```
let p1 = Promise.resolve();
p1.then(() => console.log('p1.then() onResolved'));
console.log('p1.then() returns');

let p2 = Promise.reject();
p2.then(null, () => console.log('p2.then() onRejected'));
console.log('p2.then() returns');

let p3 = Promise.reject();
p3.catch(() => console.log('p3.catch() onRejected'));
console.log('p3.catch() returns');

let p4 = Promise.resolve();
p4.finally(() => console.log('p4.finally() onFinally'));
console.log('p4.finally() returns');

// p1.then() returns
// p2.then() returns
// p3.catch() returns
// p4.finally() returns
// p1.then() onResolved
// p2.then() onRejected
// p3.catch() onRejected
// p4.finally() onFinally
```

Порядок выполнения родственных обработчиков

Если к промису прикреплено несколько обработчиков, когда промис переходит в установленное состояние, связанные обработчики будут выполняться в том порядке, в котором они были присоединены. Это верно для `then()`, `catch()` и `finally()`:

```
let p1 = Promise.resolve();
let p2 = Promise.reject();

p1.then(() => setTimeout(console.log, 0, 1));
p1.then(() => setTimeout(console.log, 0, 2));
// 1
// 2

p2.then(null, () => setTimeout(console.log, 0, 3));
p2.then(null, () => setTimeout(console.log, 0, 4));
// 3
```

```
// 4

p2.catch(() => setTimeout(console.log, 0, 5));
p2.catch(() => setTimeout(console.log, 0, 6));
// 5
// 6

p1.finally(() => setTimeout(console.log, 0, 7));
p1.finally(() => setTimeout(console.log, 0, 8));
// 7
// 8
```

Передача разрешенных значений и причин отклонения

При достижении установленного состояния промис предоставит свое разрешенное значение (если он был выполнен) или причину отклонения (если он был отклонен) любым обработчикам, которые присоединены к этому состоянию. Это особенно полезно в тех случаях, когда требуются последовательные блоки вычислений. Например, если для выполнения второго сетевого запроса требуется ответ JSON на первый сетевой запрос, ответ от первого запроса может быть передан в качестве разрешенного значения обработчику `onResolved`. С другой стороны, неудачный сетевой запрос может передать код состояния HTTP обработчику `onRejected`.

Разрешенные значения и причины отклонения присваиваются исполнителю в качестве первого аргумента функций `resolve()` или `reject()`. Эти значения предоставляются их соответствующим обработчикам `onResolved` или `onRejected` в качестве единственного параметра. Эта передача продемонстрирована здесь:

```
let p1 = new Promise((resolve, reject) => resolve('foo'));
p1.then((value) => console.log(value));    // foo

let p2 = new Promise((resolve, reject) => reject('bar'));
p2.catch((reason) => console.log(reason));    // bar
```

`Promise.resolve()` и `Promise.reject()` принимают аргумент значения/причины при вызове статического метода. Обработчикам `onResolved` и `onRejected` предоставляются значение или причина так же, как если бы они были переданы от исполнителя:

```
let p1 = Promise.resolve('foo');
p1.then((value) => console.log(value));    // foo

let p2 = Promise.reject('bar');
p2.catch((reason) => console.log(reason));    // bar
```

Отклонение промисов и обработка ошибок отклонения

Отклонение промиса аналогично выражению `throw` в том смысле, что оба они представляют состояние программы, которое должно вызвать прекращение любых последующих операций. Сгенерировав ошибку в исполнителе или обработчике промиса, вы отклоните его; соответствующий объект ошибки будет причиной отклонения. Поэтому все эти промисы будут отклонены с объектом ошибки:

```
let p1 = new Promise((resolve, reject) => reject(Error('foo')));
let p2 = new Promise((resolve, reject) => { throw Error('foo'); });
let p3 = Promise.resolve().then(() => { throw Error('foo'); });
let p4 = Promise.reject(Error('foo'));

setTimeout(console.log, 0, p1);    // Promise <rejected>: Error: foo
setTimeout(console.log, 0, p2);    // Promise <rejected>: Error: foo
setTimeout(console.log, 0, p3);    // Promise <rejected>: Error: foo
setTimeout(console.log, 0, p4);    // Promise <rejected>: Error: foo
// Также были сгенерированы четыре необрабатываемые ошибки
```

Промисы могут быть отклонены с любым значением, включая `undefined`, но настоятельно рекомендуется постоянно использовать объект ошибки. Основной причиной является то, что создание объекта ошибки позволяет браузеру захватывать трассировку стека внутри объекта ошибки, что очень полезно при отладке. Например, трассировка стека для трех ошибок в предыдущем коде должна выглядеть примерно так:

```
Uncaught (in promise) Error: foo
    at Promise (test.html:5)
    at new Promise (<anonymous>)
    at test.html:5
Uncaught (in promise) Error: foo
    at Promise (test.html:6)
    at new Promise (<anonymous>)
    at test.html:6
Uncaught (in promise) Error: foo
    at test.html:8
Uncaught (in promise) Error: foo
    at Promise.resolve.then (test.html:7)
```

Все ошибки генерируются асинхронно и не обрабатываются, а трассировка стека, захваченная объектами ошибок, показывает путь, по которому прошел объект ошибки. Также обратите внимание на порядок ошибок: `Promise.resolve().then()` ошибка генерируется последней — она требует дополнительной записи в очереди сообщений среды выполнения, потому что создает один *дополнительный* промис до того, как в конечном итоге выдает необрабатываемую ошибку.

Этот пример также раскрывает интересный побочный эффект асинхронных ошибок. Обычно при выдаче ошибки с использованием ключевого слова `throw` поведение JavaScript во время выполнения будет отклоняться, чтобы выполнить любые инструкции, следующие за выданной ошибкой.

```
throw Error('foo');
console.log('bar'); // Этот код никогда не выполнится

// Uncaught Error: foo
```

Тем не менее, когда ошибка появляется в промисе, поскольку ошибка фактически генерируется асинхронно из очереди сообщений, это не помешает среде выполнения продолжить выполнение синхронных инструкций:

```
Promise.reject(Error('foo'));
console.log('bar');
// bar

// Uncaught (in promise) Error: foo
```

Как было показано ранее в этой главе с помощью `Promise.reject()`, асинхронная ошибка может быть обнаружена только с помощью асинхронного обработчика `onRejection`.

```
// Правильно
Promise.reject(Error('foo')).catch((e) => {});

// Неправильно
try {
    Promise.reject(Error('foo'));
} catch(e) {}
```

Это не относится к перехвату ошибки, пока она находится внутри исполнителя, где `try/catch` все равно будет достаточно, чтобы перехватить ошибку, прежде чем она отклонит промис:

```
let p = new Promise((resolve, reject) => {
    try {
        throw Error('foo');
    } catch(e) {}

    resolve('bar');
});

setTimeout(console.log, 0, p);    // Promise <resolved>: bar
```

Обработчик `onRejected` для `then()` и `catch()` аналогичен семантике `try/catch` в том, что перехват ошибки должен эффективно ее нейтрализовать и позволить продолжить нормальные вычисления.

Следовательно, нужно иметь в виду, что обработчик `onRejected`, которому поручено отлавливать асинхронную ошибку, на самом деле возвращает разрешенный промис. Синхронное/асинхронное сравнение демонстрируется в следующем примере:

```
console.log('begin synchronous execution');
try {
    throw Error('foo');
} catch(e) {
    console.log('caught error', e);
}
console.log('continue synchronous execution');

// begin synchronous execution
// caught error Error: foo
// continue synchronous execution

new Promise((resolve, reject) => {
    console.log('begin asynchronous execution');
    reject(Error('bar'));
```

```
}).catch((e) => {
    console.log('caught error', e);
}).then(() => {
    console.log('continue asynchronous execution');
});

// begin asynchronous execution
// caught error Error: bar
// continue asynchronous execution
```

Композиция и цепочки промисов

Объединение нескольких промисов вместе приводит к нескольким мощным паттернам кода. Такое поведение возможно с помощью двух основных способов: цепочка промисов, которая включает в себя строгую последовательность нескольких промисов, и композиция промисов, которая включает в себя объединение нескольких промисов в один.

Цепочки промисов

Одним из наиболее полезных аспектов промисов в ECMAScript является их способность быть строго упорядоченными. Это добавляется через структуру Promise API: каждый из методов экземпляра промиса — `then()`, `catch()` и `finally()` — возвращает отдельный экземпляр промиса, у которого, в свою очередь, может быть вызван другой метод экземпляра. Последовательный вызов методов таким способом называется «цепочкой промисов». Ниже приведен простой пример этого:

```
let p = new Promise((resolve, reject) => {
    console.log('first');
    resolve();
});
p.then(() => console.log('second'))
 .then(() => console.log('third'))
 .then(() => console.log('fourth'));

// first
// second
// third
// fourth
```

Эта реализация в конечном итоге выполняет цепные *синхронные* задачи. Из-за этого выполненная работа не очень полезна или интересна, потому что она примерно такая же, как и последовательный вызов четырех функций по очереди:

```
(() => console.log('first'))();
(() => console.log('second'))();
(() => console.log('third'))();
(() => console.log('fourth'))();
```

Чтобы связать *асинхронные* задачи, этот пример можно перенастроить, чтобы каждый исполнитель возвращал экземпляр промиса. Поскольку каждый последующий

промис будет ожидать разрешения своего предшественника, такую стратегию можно использовать для сериализации асинхронных задач. Например, это можно использовать для последовательного выполнения нескольких промисов, которые разрешаются после истечения времени ожидания:

```
let p1 = new Promise((resolve, reject) => {
    console.log('p1 executor');
    setTimeout(resolve, 1000);
});

p1.then(() => new Promise((resolve, reject) => {
        console.log('p2 executor');
        setTimeout(resolve, 1000);
    }))
    .then(() => new Promise((resolve, reject) => {
        console.log('p3 executor');
        setTimeout(resolve, 1000);
    }))
    .then(() => new Promise((resolve, reject) => {
        console.log('p4 executor');
        setTimeout(resolve, 1000);
    }));

// p1 executor (спустя 1 с)
// p2 executor (спустя 2 с)
// p3 executor (спустя 3 с)
// p4 executor (спустя 4 с)
```

Объединение генерации промисов в одну фабричную функцию делает следующее:

```
function delayedResolve(str) {
    return new Promise((resolve, reject) => {
        console.log(str);
        setTimeout(resolve, 1000);
    });
}

delayedResolve('p1 executor')
    .then(() => delayedResolve('p2 executor'))
    .then(() => delayedResolve('p3 executor'))
    .then(() => delayedResolve('p4 executor'))

// p1 executor (спустя 1 с)
// p2 executor (спустя 2 с)
// p3 executor (спустя 3 с)
// p4 executor (спустя 4 с)
```

Каждый последующий обработчик ожидает разрешения своего предшественника, создает новый экземпляр промиса и возвращает его. Такая структура способна аккуратно сериализовать асинхронный код без принудительного использования обратных вызовов. Без использования промисов предыдущий код будет выглядеть примерно так:

```
function delayedExecute(str, callback = null) {
    setTimeout(() => {
        console.log(str);
        callback && callback();
    }, 1000)
}

delayedExecute('p1 callback', () => {
    delayedExecute('p2 callback', () => {
        delayedExecute('p3 callback', () => {
            delayedExecute('p4 callback');
        });
    });
});

// p1 callback (спустя 1 с)
// p2 callback (спустя 2 с)
// p3 callback (спустя 3 с)
// p4 callback (спустя 4 с)
```

Наблюдательный разработчик заметит, что это приводит к аду обратных вызовов, для обхода которых и были предназначены промисы.

Поскольку `then()`, `catch()` и `finally()` возвращают промис, объединить их в цепочку просто. Следующий пример включает в себя все три метода:

```
let p = new Promise((resolve, reject) => {
    console.log('initial promise rejects');
    reject();
});

p.catch(() => console.log('reject handler'))
 .then(() => console.log('resolve handler'))
 .finally(() => console.log('finally handler'));

// initial promise rejects
// reject handler
// resolve handler
// finally handler
```

Графы промисов

Поскольку к одному промису может быть присоединено произвольное число обработчиков, также возможно формирование ориентированных ациклических графов связанных промисов. Каждый промис является узлом в графе, и присоединение обработчика с использованием метода экземпляра добавляет направленную вершину. Поскольку каждый узел в графе будет ожидать установления своего предшественника, гарантируется, что направление вершин графа будет указывать порядок выполнения промиса.

Ниже приведен пример одного возможного типа ориентированного графа промисов — бинарного дерева:

```
//      A
//     / \
//    B   C
//   / \ / \
//  D  E F  G

let A = new Promise((resolve, reject) => {
    console.log('A');
    resolve();
});

let B = A.then(() => console.log('B'));
let C = A.then(() => console.log('C'));

B.then(() => console.log('D'));
B.then(() => console.log('E'));
C.then(() => console.log('F'));
C.then(() => console.log('G'));

// A
// B
// C
// D
// E
// F
// G
```

Обратите внимание, что порядок записи выражений является обходом уровня этого бинарного дерева. Как обсуждалось в предыдущем разделе, ожидается, что обработчики промисов будут выполняться в том порядке, в котором они были присоединены. Поскольку обработчики промиса *охотно* добавляются в очередь сообщений, но выполняются *лениво*, результатом такого поведения является обход уровня порядка.

Деревья являются лишь одним из проявлений графа промисов. Поскольку не обязательно существует один корневой узел промиса и поскольку несколько промисов можно объединить в один (с помощью `Promise.all()` или `Promise.race()`, как будет обсуждаться в следующем разделе), направленный ациклический граф является наиболее точной характеристикой вселенной возможностей цепочки промисов.

Параллельная композиция промисов с Promise.all() и Promise.race()

Класс `Promise` предоставляет два статических метода, которые позволяют создавать новый экземпляр промиса из нескольких экземпляров. Поведение этого составного промиса основано на том, как ведут себя промисы внутри него.

Promise.all()

Статический метод `Promise.all()` создает промис с принципом «все или ничего», который разрешается только один раз, когда разрешаются все промисы в коллекции. Статический метод принимает итерируемое значение и возвращает новый промис:

```
let p1 = Promise.all([
    Promise.resolve(),
    Promise.resolve()
]);

// Элементы в итерируемом параметре приводятся к промису
// с использованием Promise.resolve()
let p2 = Promise.all([3, 4]);

// Пустой итерируемый параметр эквивалентен to Promise.resolve()
let p3 = Promise.all([]);

// Неверный синтаксис
let p4 = Promise.all();
// TypeError: cannot read Symbol.iterator of undefined
```

Составной промис разрешается только после разрешения каждого содержащегося в нем промиса:

```
let p = Promise.all([
    Promise.resolve(),
    new Promise((resolve, reject) => setTimeout(resolve, 1000))
]);
setTimeout(console.log, 0, p);    // Promise <pending>

p.then(() => setTimeout(console.log, 0, 'all() resolved!'));

// all() resolved! (Спустя ~1000 мс)
```

Если хотя бы один промис из коллекции остается в ожидании, составной промис также останется в ожидании. Если один промис из коллекции отклоняется, отклоняется составной промис:

```
// Навсегда останется в ожидании
let p1 = Promise.all([new Promise(() => {})]);
setTimeout(console.log, 0, p1); // Promise <pending>

// Единственное отклонение вызывает отклонение составного промиса
let p2 = Promise.all([
    Promise.resolve(),
    Promise.reject(),
    Promise.resolve()
]);
setTimeout(console.log, 0, p2); // Promise <rejected>

// Uncaught (in promise) undefined
```

Если все промисы успешно разрешены, разрешенное значение составного промиса будет массивом всех разрешенных значений содержащихся промисов в порядке обхода:

```
let p = Promise.all([
    Promise.resolve(3),
    Promise.resolve(),
    Promise.resolve(4)
]);

p.then((values) => setTimeout(console.log, 0, values));    // [3, undefined, 4]
```

Если один из промисов отклоняется, то тот, что будет отклонен первым, устанавливает причину отклонения составного промиса. Последующие отклонения не влияют на причину отклонения; однако на нормальное поведение отклонения этих содержащихся в нем экземпляров промисов это не влияет. Важно, что составной промис *будет* просто обрабатывать отклонение всех содержащихся промисов, как показано здесь:

```
// Хотя только первая причина отклонения будет указана в
// обработчике отклонений, вторая причина отклонения будет просто
// обработана без ошибок
let p = Promise.all([
    Promise.reject(3),
    new Promise((resolve, reject) => setTimeout(reject, 1000))
]);

p.catch((reason) => setTimeout(console.log, 0, reason)); // 3

// Нет необработанных ошибок
```

Promise.race()

Статический метод `Promise.race()` создает промис, который будет отражать то, что промис в коллекции сначала достигает разрешенного или отклоненного состояния. Статический метод принимает итерируемый параметр и возвращает новый промис:

```
let p1 = Promise.race([
    Promise.resolve(),
    Promise.resolve()
]);

// Элементы в итерируемом параметре приводятся к промисам
// с использованием Promise.resolve()
let p2 = Promise.race([3, 4]);

// Пустой итерируемый объект эквивалентен new Promise(() => {})
let p3 = Promise.race([]);

// Неверный синтаксис
let p4 = Promise.race();
// TypeError: cannot read Symbol.iterator of undefined
```

Метод `Promise.race()` не предоставляет предпочтения разрешенному или отклоненному промису. Составной промис будет проходить через статус и значение/причину первого установленного промиса, как показано здесь:

```
// Сначала происходит разрешение, отклонение по тайм-ауту игнорируется
let p1 = Promise.race([
    Promise.resolve(3),
    new Promise((resolve, reject) => setTimeout(reject, 1000))
]);
setTimeout(console.log, 0, p1);    // Promise <resolved>: 3

// Сначала происходит отклонение, разрешение по тайм-ауту игнорируется
```

```
let p2 = Promise.race([
    Promise.reject(4),
    new Promise((resolve, reject) => setTimeout(resolve, 1000))
]);
setTimeout(console.log, 0, p2);    // Promise <rejected>: 4

// Порядок обхода — значение для расчета
let p3 = Promise.race([
    Promise.resolve(5),
    Promise.resolve(6),
    Promise.resolve(7)
]);
setTimeout(console.log, 0, p3);    // Promise <resolved>: 5
```

Если один из промисов отклоняется, то тот, что будет отклонен первым, устанавливает причину отклонения составного промиса. Последующие отклонения не влияют на причину отклонения; однако на нормальное поведение отклонения этих содержащихся в нем экземпляров промисов это не влияет. Как и в случае с `Promise.all()`, составной промис *будет* просто обрабатывать отклонение всех содержащихся промисов, как показано здесь:

```
// Хотя только первая причина отклонения будет указана в
// обработчике отклонений, вторая причина отклонения будет просто
// обработана без ошибок
let p = Promise.race([
    Promise.reject(3),
    new Promise((resolve, reject) => setTimeout(reject, 1000))
]);

p.catch((reason) => setTimeout(console.log, 0, reason));    // 3

// Нет необработанных ошибок
```

Серийная композиция промисов

До сих пор обсуждение цепочки промисов было сосредоточено на сериализации выполнения и в значительной степени игнорировало основную особенность промисов: их способность асинхронно создавать значение и предоставлять его обработчикам. Цепочка промисов вместе с намерением каждого последующего промиса, использующего значение своего предшественника, является фундаментальной особенностью промисов. Это во многом аналогично *композиции функций*, где несколько функций объединены в новую функцию, продемонстрированную здесь:

```
function addTwo(x) {return x + 2;}
function addThree(x) {return x + 3;}
function addFive(x) {return x + 5;}

function addTen(x) {
    return addFive(addTwo(addThree(x)));
}

console.log(addTen(7));    // 17
```

В этом примере композиция функций используется для объединения этих трех функций в одну для работы с одним значением. Точно так же промисы могут быть составлены вместе, чтобы постепенно потреблять значение и производить один промис, содержащий результат. Явное представление этого выглядит следующим образом:

```
function addTwo(x) {return x + 2;}
function addThree(x) {return x + 3;}
function addFive(x) {return x + 5;}

function addTen(x) {
    return Promise.resolve(x)
        .then(addTwo)
        .then(addThree)
        .then(addFive);
}

addTen(8).then(console.log);    // 18
```

Это можно преобразовать в более лаконичную форму с помощью `Array.prototype.reduce()`:

```
function addTwo(x) {return x + 2;}
function addThree(x) {return x + 3;}
function addFive(x) {return x + 5;}

function addTen(x) {
    return [addTwo, addThree, addFive]
        .reduce((promise, fn) => promise.then(fn), Promise.resolve(x));
}

addTen(8).then(console.log);    // 18
```

Такая стратегия композиции промисов может быть обобщена в функцию, которая объединяет любое количество функций в цепочку промисов с передачей значений. Эта обобщенная функция композиции может быть реализована следующим образом:

```
function addTwo(x) {return x + 2;}
function addThree(x) {return x + 3;}
function addFive(x) {return x + 5;}

function compose(...fns) {
    return (x) => fns.reduce((promise, fn) => promise.then(fn), Promise.resolve(x))
}

let addTen = compose(addTwo, addThree, addFive);

addTen(8).then(console.log);    // 18
```

ПРИМЕЧАНИЕ Эта концепция вновь рассматривается в разделе «Асинхронные функции» далее в этой главе.

Расширения промисов

Реализация промисов в ES6 является надежной, но, как и в случае с любым программным обеспечением, будут появляться недостатки. Существуют два предложения, доступные в некоторых сторонних реализациях промисов, но отсутствующие в формальной спецификации ECMAScript, — это отмена промисов и отслеживание прогресса.

Отмена промисов

Часто промис выполняется, но программа больше не заботится о результате. В такой ситуации была бы полезна возможность «отменить» промис. Некоторые сторонние библиотеки промисов, например Bluebird, предлагают такую функцию, и даже сама ECMAScript должна была предложить такую функцию до того, как она была в итоге отозвана (https://github.com/tc39/proposal-cancelable-promises). В результате промисы ES6 считаются «нетерпеливыми»: как только инкапсулированная функция промиса выполняется, невозможно помешать завершению этого процесса.

Все еще можно реализовать специальный подход, который является факсимильной версией оригинального проекта. Такая реализация использует «токен отмены» — концепцию, воплощенную в эскизном проекте Кевина Смита (https://github.com/zenparsing/es-cancel-token). Сгенерированный токен отмены предоставляет интерфейс, через который можно отменить промис, а также ловушку промиса, с помощью которой можно вызвать поведение отмены и оценить состояние отмены.

Базовая реализация класса `CancelToken` может выглядеть следующим образом:

```
class CancelToken {
    constructor(cancelFn) {
        this.promise = new Promise((resolve, reject) => {
            cancelFn(resolve);
        });
    }
}
```

Этот класс оборачивает промис, который передает метод разрешения в параметр `cancelFn`. Затем внешняя сущность сможет предоставить конструктору функцию, позволяющую этой сущности точно контролировать, когда токен должен быть отменен. Этот промис является открытым членом класса токенов, и, следовательно, можно добавить слушателей к промису отмены.

Пример грубого использования этого класса показан ниже:

```
<button id="start">Start</button>
<button id="cancel">Cancel</button>

<script>
class CancelToken {
    constructor(cancelFn) {
        this.promise = new Promise((resolve, reject) => {
            cancelFn(() => {
```

```
                setTimeout(console.log, 0, "delay cancelled");
                resolve();
            });
        });
    }
}

const startButton = document.querySelector('#start');
const cancelButton = document.querySelector('#cancel');

function cancellableDelayedResolve(delay) {
    setTimeout(console.log, 0, "set delay");

    return new Promise((resolve, reject) => {
        const id = setTimeout((() => {
            setTimeout(console.log, 0, "delayed resolve");
            resolve();
        }), delay);

        const cancelToken = new CancelToken((cancelCallback) =>
            cancelButton.addEventListener("click", cancelCallback));

        cancelToken.promise.then(() => clearTimeout(id));
    });
}

startButton.addEventListener("click", () => cancellableDelayedResolve(1000));
</script>
```

Каждое нажатие кнопки Start запускает тайм-аут и создает новый экземпляр `CancelToken`. Кнопка Cancel настроена таким образом, что нажатие приведет к разрешению токена. После разрешения тайм-аут, изначально установленный нажатием кнопки Start, будет отменен.

Уведомления о прогрессе промисов

Выполняемый промис может иметь несколько отдельных «стадий», через которые он будет проходить до фактического разрешения. В некоторых ситуациях может быть полезно разрешить программе отслеживать промис при достижении этих контрольных точек. Промисы в ECMAScript 6 не поддерживают эту концепцию, но все еще возможно подражать этому поведению, расширяя промис.

Одна потенциальная реализация — расширение класса `Promise` с помощью метода `notify()`, как показано здесь:

```
class TrackablePromise extends Promise {
    constructor(executor) {
        const notifyHandlers = [];

        super((resolve, reject) => {
            return executor(resolve, reject, (status) => {
                notifyHandlers.map((handler) => handler(status));
            }) ;
        });
```

```
        this.notifyHandlers = notifyHandlers;
    }

    notify(notifyHandler) {
        this.notifyHandlers.push(notifyHandler);
        return this;
    }
}
```

После этого `TrackablePromise` сможет использовать функцию `notify()` внутри исполнителя. Создание экземпляра промиса может использовать эту функцию следующим образом:

```
let p = new TrackablePromise((resolve, reject, notify) => {
    function countdown(x) {
        if (x > 0) {
            notify(`${20 * x}% remaining`);
            setTimeout(() => countdown(x - 1), 1000);
        } else {
            resolve();
        }
    }

    countdown(5);
});
```

Этот промис будет рекурсивно устанавливать время ожидания 1000 мс пять раз подряд перед разрешением. Каждый обработчик тайм-аута будет вызывать `notify()` и передавать статус. Предоставление обработчика уведомлений может быть сделано следующим образом:

```
...

let p = new TrackablePromise((resolve, reject, notify) => {
    function countdown(x) {
        if (x > 0) {
            notify(`${20 * x}% remaining`);
            setTimeout(() => countdown(x - 1), 1000);
        } else {
            resolve();
        }
    }

    countdown(5);
});

p.notify((x) => setTimeout(console.log, 0, 'progress:', x));

p.then(() => setTimeout(console.log, 0, 'completed'));

// (спустя 1 с) 80% remaining
// (спустя 2 с) 60% remaining
// (спустя 3 с) 40% remaining
// (спустя 4 с) 20% remaining
// (спустя 5 с) completed
```

Этот метод `notify()` спроектирован так, чтобы создавать цепочку, возвращая самого себя, и выполнение обработчика будет сохраняться для каждого уведомления, как показано здесь:

```
...

p.notify((x) => setTimeout(console.log, 0, 'a:', x))
    .notify((x) => setTimeout(console.log, 0, 'b:', x));

p.then(() => setTimeout(console.log, 0, 'completed'));

// (спустя 1 с) a: 80% remaining
// (спустя 1 с) b: 80% remaining
// (спустя 2 с) a: 60% remaining
// (спустя 2 с) b: 60% remaining
// (спустя 3 с) a: 40% remaining
// (спустя 3 с) b: 40% remaining
// (спустя 4 с) a: 20% remaining
// (спустя 4 с) b: 20% remaining
// (спустя 5 с) completed
```

В целом это довольно грубая реализация, но она должна продемонстрировать, как такая функция уведомления может быть полезной.

> **ПРИМЕЧАНИЕ** Одна из основных причин, по которой в промисах ES6 не предусмотрены функции отмены или уведомления, заключается в том, что это значительно усложняет создание цепочек и композиций промисов. Не совсем ясно, что ожидается в сценариях, где отмены или уведомления происходят в промисах с другими зависимостями, например в цепочке промисов. Говоря риторически, что является разумным поведением: когда обещание внутри `Promise.all()` отменяется или когда уведомление отправляется из предыдущего промиса в цепочке?

АСИНХРОННЫЕ ФУНКЦИИ

Асинхронные функции, также называемые парой оперативных ключевых слов async/await, являются применением парадигмы ES6 Promise к функциям ECMAScript. Поддержка async/await была представлена в спецификации ES7. Это и поведенческое, и синтаксическое усовершенствование спецификации, которое допускает код JavaScript. Он написан синхронно, но на самом деле способен вести себя асинхронно. Простейший пример этого начинается с простого промиса, который разрешается значением после тайм-аута:

```
let p = new Promise((resolve, reject) => setTimeout(resolve, 1000, 3));
```

Этот промис будет разрешен со значением 3 спустя 1000 мс. Чтобы другие части программы могли получить доступ к этому значению, как только оно будет готово, оно должно существовать внутри обработчика разрешения:

```
let p = new Promise((resolve, reject) => setTimeout(resolve, 1000, 3));

p.then((x) => console.log(x));    // 3
```

Это довольно неудобно, так как остальная часть программы теперь должна быть добавлена в обработчик промиса. Можно переместить обработчик в определение функции:

```
function handler(x) { console.log(x); }

let p = new Promise((resolve, reject) => setTimeout(resolve, 1000, 3));

p.then(handler);    // 3
```

Это ненамного исправило ситуацию. Факт остается фактом: любой последующий код, который хочет получить доступ к значению, созданному промисом, должен быть передан этому значению через обработчик. Это означает, что его помещают в функцию-обработчик. ES7 предлагает `async/await` в качестве элегантного решения этой проблемы.

Основы асинхронных функций

async/await в ES7 предназначен для непосредственного решения проблемы организации кода, который использует асинхронные конструкции. Он вводит логическое расширение асинхронного поведения в область функций JavaScript, вводя два новых ключевых слова, `async` и `await`.

Ключевое слово async

Асинхронную функцию можно объявить, добавив ключевое слово `async`. Это ключевое слово может использоваться в объявлениях функций, функциях-выражениях, стрелочных функциях и методах:

```
async function foo() {}

let bar = async function() {};

let baz = async () => {};

class Qux {
    async qux() {}
}
```

Использование ключевого слова `async` создаст функцию, которая демонстрирует некоторые асинхронные характеристики, но в целом вычисляется синхронно. Во всех других отношениях, таких как аргументы и замыкания, она по-прежнему демонстрирует все нормальное поведение функции JavaScript. Рассмотрим этот простой пример, показывающий, что функция `foo()` все еще вычисляется, прежде чем перейти к последующим инструкциям:

```
async function foo() {
    console.log(1);
}

foo();
console.log(2);

// 1
// 2
```

В асинхронной функции любое значение, возвращаемое с ключевым словом `return` (или `undefined`, если нет `return`), будет эффективно преобразовано в объект промиса с помощью `Promise.resolve()`. Асинхронная функция всегда будет возвращать объект промиса. Вне функции вычисляемая функция будет объектом промиса:

```
async function foo() {
    console.log(1);
    return 3;
}

// Добавление обработчика разрешения к возвращаемому промису
foo().then(console.log);

console.log(2);

// 1
// 2
// 3
```

Конечно, это означает, что возвращение объекта промиса покажет идентичное поведение:

```
async function foo() {
    console.log(1);
    return Promise.resolve(3);
}

// Добавление обработчика разрешения к возвращаемому промису
foo().then(console.log);

console.log(2);

// 1
// 2
// 3
```

Возвращаемое значение асинхронной функции ожидает, но на самом деле не требует доступного объекта: оно также будет работать с обычными значениями. Объект с возможностью последующего использования будет развернут через первый аргумент, предоставленный обратному вызову `then()`. Объект, недоступный для последующего просмотра, будет пропущен, как если бы это был уже разрешенный промис. Эти различные сценарии демонстрируются ниже:

```
// Возврат примитива
async function foo() {
    return 'foo';
}
foo().then(console.log);
// foo

// Возврат объекта, недоступного для последующего просмотра
async function bar() {
    return ['bar'];
}
bar().then(console.log);
// ['bar']

// Возврат доступного для последующего просмотра объекта (не промиса)
async function baz() {
    const thenable = {
        then(callback) { callback('baz'); }
    };
    return thenable;
}
baz().then(console.log);
// baz

// Возврат промиса
async function qux() {
    return Promise.resolve('qux');
}
qux().then(console.log);
// qux
```

Как и в случае с функциями обработчика промисов, выдача значения ошибки будет возвращать отклоненный промис:

```
async function foo() {
    console.log(1);
    throw 3;
}

// Прикрепление обработчика отклонения к возвращаемому промису
foo().catch(console.log);
console.log(2);

// 1
// 2
// 3
```

Однако ошибки отклонения промисов *не* будут регистрироваться асинхронной функцией:

```
async function foo() {
    console.log(1);
    Promise.reject(3);
}
```

```
// Прикрепление обработчика отклонения к возвращаемому промису
foo().catch(console.log);
console.log(2);

// 1
// 2
// Uncaught (in promise): 3
```

Ключевое слово await

Поскольку асинхронная функция указывает вызывающему ее коду, что не ожидается своевременного завершения, логическим расширением этого поведения является возможность приостановить и возобновить выполнение. Эта функция возможна с помощью ключевого слова `await`, которое используется для приостановки выполнения в ожидании разрешения промиса. Рассмотрим пример из начала главы:

```
let p = new Promise((resolve, reject) => setTimeout(resolve, 1000, 3));

p.then((x) => console.log(x)); // 3
```

Его можно переписать с помощью `async/await` следующим образом:

```
async function foo() {
    let p = new Promise((resolve, reject) => setTimeout(resolve, 1000, 3));
    console.log(await p);
}

foo();
// 3
```

Ключевое слово `await` приостанавливает выполнение асинхронной функции, освобождая поток выполнения во время выполнения JavaScript. Это поведение мало чем отличается от ключевого слова `yield` в функции генератора. Ключевое слово `await` попытается «развернуть» значение объекта, передать значение в выражение и асинхронно возобновить выполнение асинхронной функции.

Ключевое слово `await` используется так же, как унарный оператор JavaScript. Его можно использовать отдельно или внутри выражения, как показано в следующих примерах:

```
// Асинхронно выводит "foo"
async function foo() {
    console.log(await Promise.resolve('foo'));
}
foo();
// foo

// Асинхронно выводит "bar"
async function bar() {
    return await Promise.resolve('bar');
}
bar().then(console.log);
```

```
// bar
// Асинхронно выводит "baz" спустя 1000 мс
async function baz() {
    await new Promise((resolve, reject) => setTimeout(resolve, 1000));
    console.log('baz');
}
baz();
// baz <спустся 1000 мс>
```

Ключевое слово `await` предвосхищает, но на самом деле не требует наличия доступного объекта: оно также будет работать с обычными значениями. Объект с возможностью последующего использования будет «развернут» через первый аргумент, предоставленный обратному вызову `then()`. Объект, недоступный для последующего просмотра, будет пропущен, как если бы это был уже разрешенный промис. Эти различные сценарии демонстрируются ниже:

```
// Ожидание примитива
async function foo() {
    console.log(await 'foo');
}
foo();
// foo

// Ожидание объекта, недоступного для просмотра
async function bar() {
    console.log(await ['bar']);
}
bar();
// ['bar']

// Ожидание доступного для просмотра объекта (не промиса)
async function baz() {
    const thenable = {
        then(callback) { callback('baz'); }
    };
 console.log(await thenable);
}
baz();
// baz

// Ожидание промиса
async function qux() {
    console.log(await Promise.resolve('qux'));
}
qux();
// qux
```

Как и в случае с функциями обработчика промисов, выдача ошибки вместо этого возвратит отклоненный промис:

```
async function foo() {
    console.log(1);
```

```
    await (() => { throw 3; })();
}

// Прикрепление обработчика отклонения к возвращаемому промису
foo().catch(console.log);
console.log(2);

// 1
// 2
// 3
```

Как было показано ранее, автономный `Promise.reject()` не будет захвачен асинхронной функцией и будет выдаваться как необработанная ошибка. Однако использование `await` для отклоненного промиса развернет значение ошибки:

```
async function foo() {
    console.log(1);
    await Promise.reject(3);
    console.log(4);    // это никогда не сработает
}

// Прикрепление обработчика отклонения к возвращаемому промису
foo().catch(console.log);
console.log(2);

// 1
// 2
// 3
```

Ограничения await

Ключевое слово `await` должно использоваться внутри асинхронной функции; его нельзя использовать в контексте верхнего уровня, таком как тег сценария или модуль. Однако ничто не мешает вам немедленно вызывать асинхронную функцию. Следующие два фрагмента кода фактически идентичны:

```
async function foo() {
    console.log(await Promise.resolve(3));
}
foo();
// 3

// Немедленный вызов асинхронной функции-выражения
(async function() {
    console.log(await Promise.resolve(3));
})();
// 3
```

Кроме того, асинхронная природа функции не распространяется на вложенные функции. Поэтому ключевое слово `await` также может появляться только внутри определения асинхронной функции; попытка использовать `await` внутри синхронной функции вызовет ошибку `SyntaxError`.

Ниже показаны несколько недопустимых примеров:

```
// Недопустимо: 'await' внутри стрелочной функции
function foo() {
    const syncFn = () => {
        return await Promise.resolve('foo');
    };
    console.log(syncFn());
}

// Недопустимо: 'await' внутри объявления функции
function bar() {
    function syncFn() {
        return await Promise.resolve('bar');
    }
    console.log(syncFn());

// Недопустимо: 'await' внутри функции-выражения
function baz() {
    const syncFn = function() {
        return await Promise.resolve('baz');
    };
    console.log(syncFn());
}

// Недопустимо: использование функции-выражения или стрелочной функции с IIFE
function qux() {
    (function () { console.log(await Promise.resolve('qux')); })();
    (() => console.log(await Promise.resolve('qux')))();
}
```

Остановка и возобновление выполнения

Истинная природа использования ключевого слова `await` более тонка, чем может показаться на первый взгляд. Рассмотрим следующий пример, где три функции вызываются по порядку, но их выходные данные печатаются в обратном порядке:

```
async function foo() {
    console.log(await Promise.resolve('foo'));
}

async function bar() {
    console.log(await 'bar');
}

async function baz() {
    console.log('baz');
}

foo();
bar();
baz();

// baz
// bar
// foo
```

В парадигме `async/await` ключевое слово `await` выполняет всю тяжелую работу. Ключевое слово `async` во многих отношениях является просто специальным индикатором для интерпретатора JavaScript. В конце концов, асинхронная функция, которая не содержит ключевое слово `await`, выполняется так же, как обычная функция:

```
async function foo() {
    console.log(2);
}

console.log(1);
foo();
console.log(3);

// 1
// 2
// 3
```

Ключ к полному пониманию ключевого слова `await` заключается в том, что оно не просто ждет, пока значение станет доступным. При обнаружении ключевого слова `await` среда выполнения JavaScript может точно отслеживать, где было приостановлено выполнение. Когда значение справа от `await` будет готово, среда выполнения JavaScript вставит задачу в очередь сообщений, которая асинхронно возобновит выполнение этой функции.

Следовательно, даже когда `await` сопряжено с немедленно доступным значением, остальная часть функции все равно будет выполняться *асинхронно*. Это продемонстрировано в следующем примере:

```
async function foo() {
    console.log(2);
    await null;
    console.log(4);
}

console.log(1);
foo();
console.log(3);

// 1
// 2
// 3
// 4
```

Порядок вывода на консоль лучше всего объяснить с точки зрения того, как среда выполнения обрабатывает этот пример:

1. **Вывод 1**.
2. Вызов асинхронной функции `foo`.
3. (внутри `foo`) **Вывод 2**.
4. (внутри `foo`) Ключевое слово `await` приостанавливает выполнение и добавляет задачу в очередь сообщений для немедленно доступного значения `null`.

5. `foo` завершается.
6. **Вывод 3**.
7. Синхронный поток выполнения заканчивается.
8. Среда выполнения JavaScript удаляет задачу из очереди сообщений, чтобы возобновить выполнение.
9. (внутри `foo`) Выполнение возобновляется; `await` предоставляется со значением `null` (которое здесь не используется).
10. (внутри `foo`) **Вывод 4**.
11. Возвращение `foo`.

Использование `await` с промисом усложняет этот сценарий. В этом случае фактически создаются две отдельные задачи очереди сообщений, которые оцениваются асинхронно для завершения выполнения асинхронной функции. Следующий пример, который может показаться совершенно нелогичным, демонстрирует порядок выполнения:

```
async function foo() {
    console.log(2);
    console.log(await Promise.resolve(8));
    console.log(9);
}

async function bar() {
    console.log(4);
    console.log(await 6);
    console.log(7);
}

console.log(1);
foo();
console.log(3);
bar();
console.log(5);

// 1
// 2
// 3
// 4
// 5
// 6
// 7
// 8
// 9
```

Среда выполнения выполнит этот пример следующим образом:

1. **Вывод 1**.
2. Вызов асинхронной функции `foo`.
3. (внутри `foo`) **Вывод 2**.

4. (внутри `foo`) Ключевое слово `await` приостанавливает выполнение и планирует задачу, которая будет добавлена в очередь сообщений после выполнения промиса.
5. Немедленное разрешение промиса; задача предоставления разрешенного значения для промиса добавляется в очередь сообщений.
6. Завершение `foo`.
7. **Вывод 3**.
8. Вызов асинхронной функции `bar`.
9. (внутри `bar`) **Вывод 4**.
10. (внутри `bar`) Ключевое слово `await` приостанавливает выполнение и добавляет задачу в очередь сообщений для немедленного доступного значения 6.
11. Завершение `bar`.
12. **Вывод 5**.
13. Поток выполнения верхнего уровня заканчивается.
14. Разрешенные во время выполнения JavaScript-обработчики ожидают обработчик промисов и предоставляют ему разрешенное значение 8.
15. Во время выполнения JavaScript ставится в очередь задача для возобновления выполнения `foo` в очереди сообщений.
16. Во время выполнения JavaScript исключается задача возобновить выполнение `bar` со значением 6 вне очереди сообщений.
17. (внутри `bar`) Выполнение возобновляется, промис предоставляется со значением 6.
18. (внутри `bar`) **Вывод 6**.
19. (внутри `bar`) **Вывод 7**.
20. Возвращение `bar`.
21. Завершение асинхронной задачи, JavaScript отменяет задачу, чтобы возобновить выполнение `foo` со значением 8.
22. (внутри `foo`) **Вывод 8**.
23. (внутри `foo`) **Вывод 9**.
24. Возвращение `foo`.

Стратегии для асинхронных функций

Из-за их удобства и полезности асинхронные функции все чаще повсеместно используются в кодовых базах JavaScript. Тем не менее при использовании асинхронных функций помните о некоторых особенностях.

Реализация Sleep()

При первом изучении JavaScript многие разработчики используют конструкцию, аналогичную `Thread.sleep()` в Java, пытаясь ввести неблокирующую задержку в программу. Раньше это был краеугольный педагогический способ представления, как `setTimeout` вписывается в поведение JavaScript во время выполнения.

С асинхронными функциями это уже не так! Построить утилиту, которая позволяет функции `sleep()` — «заснуть» — на миллисекунды, очень просто:

```
async function sleep(delay) {
    return new Promise((resolve) => setTimeout(resolve, delay));
}

async function foo() {
    const t0 = Date.now();
    await sleep(1500);    // sleep for ~1500ms
    console.log(Date.now() - t0);
}
foo();
// 1502
```

Максимизация распараллеливания

Если ключевое слово `await` не используется с осторожностью, программа может упустить возможные ускорения распараллеливания. Рассмотрим следующий пример, который ожидает пять случайных тайм-аутов последовательно:

```
async function randomDelay(id) {
    // Задержка между 0 и 1000 мс
    const delay = Math.random() * 1000;
    return new Promise((resolve) => setTimeout(() => {
        console.log(`${id} finished`);
        resolve();
    }, delay));
}

async function foo() {
 const t0 = Date.now();
 await randomDelay(0);
 await randomDelay(1);
 await randomDelay(2);
 await randomDelay(3);
 await randomDelay(4);
 console.log(`${Date.now() - t0}ms elapsed`);
}
foo();

// 0 finished
// 1 finished
// 2 finished
// 3 finished
// 4 finished
// 2219ms elapsed
```

После свертывания в цикл `for` получаем следующее:

```
async function randomDelay(id) {
    // Задержка между 0 и 1000 мс
    const delay = Math.random() * 1000;
    return new Promise((resolve) => setTimeout(() => {
        console.log(`${id} finished`);
        resolve();
    }, delay));
}

async function foo() {
    const t0 = Date.now();
    for (let i = 0; i < 5; ++i) {
        await randomDelay(i);
    }
    console.log(`${Date.now() - t0}ms elapsed`);
}
foo();

// 0 finished
// 1 finished
// 2 finished
// 3 finished
// 4 finished
// 2219ms elapsed
```

Даже если между промисами нет взаимозависимости, эта асинхронная функция будет приостанавливать работу и ждать завершения каждого из них, прежде чем запускать следующий. Это гарантирует сохранение порядка, но за счет общего времени выполнения.

Если сохранение порядка не требуется, лучше инициализировать промисы сразу и ожидать результатов по мере их появления. Это можно сделать следующим образом:

```
async function randomDelay(id) {
    // Задержка между 0 и 1000 мс
    const delay = Math.random() * 1000;
    return new Promise((resolve) => setTimeout(() => {
        setTimeout(console.log, 0, `${id} finished`);
        resolve();
    }, delay));
}

async function foo() {
    const t0 = Date.now();

    const p0 = randomDelay(0);
    const p1 = randomDelay(1);
    const p2 = randomDelay(2);
    const p3 = randomDelay(3);
    const p4 = randomDelay(4);

    await p0;
```

```
    await p1;
    await p2;
    await p3;
    await p4;

    setTimeout(console.log, 0, `${Date.now() - t0}ms elapsed`);
}
foo();

// 1 finished
// 4 finished
// 3 finished
// 0 finished
// 2 finished
// 2219ms elapsed
```

После свертывания в массив и цикла `for` получаем следующее:

```
async function randomDelay(id) {
    // Задержка между 0 и 1000 мс
    const delay = Math.random() * 1000;
    return new Promise((resolve) => setTimeout(() => {
        console.log(`${id} finished`);
        resolve();
    }, delay));
}

async function foo() {
    const t0 = Date.now();

    const promises = Array(5).fill(null).map((_, i) => randomDelay(i));

    for (const p of promises) {
        await p;
    }

    console.log(`${Date.now() - t0}ms elapsed`);
}

foo();

// 4 finished
// 2 finished
// 1 finished
// 0 finished
// 3 finished
// 877ms elapsed
```

Обратите внимание, что, хотя выполнение промисов нарушило порядок, операторы промисов предоставляются разрешенным значениям *в* порядке:

```
async function randomDelay(id) {
    // Задержка между 0 и 1000 мс
    const delay = Math.random() * 1000;
    return new Promise((resolve) => setTimeout(() => {
        console.log(`${id} finished`);
```

```
        resolve(id);
    }, delay));
}

async function foo() {
    const t0 = Date.now();

    const promises = Array(5).fill(null).map((_, i) => randomDelay(i));

    for (const p of promises) {
        console.log(`awaited ${await p}`);
    }
    console.log(`${Date.now() - t0}ms elapsed`);
}
foo();

// 1 finished
// 2 finished
// 4 finished
// 3 finished
// 0 finished
// awaited 0
// awaited 1
// awaited 2
// awaited 3
// awaited 4
// 645ms elapsed
```

Серийное выполнение промисов

В разделе «Промисы» этой главы обсуждается, как составлять промисы, которые выполняются последовательно и передают значения последующему промису. С async/await цепочка промисов становится очень простой:

```
function addTwo(x) {return x + 2;}
function addThree(x) {return x + 3;}
function addFive(x) {return x + 5;}

async function addTen(x) {
    for (const fn of [addTwo, addThree, addFive]) {
        x = await fn(x);
    }
    return x;
}

addTen(9).then(console.log);    // 19
```

Здесь `await` напрямую передает возвращаемое значение каждой функции и результат выводится итеративно. Предыдущий пример не имеет дела с промисами, но его можно перенастроить на использование асинхронных функций — и, следовательно, промисов — вместо этого:

```
async function addTwo(x) {return x + 2;}
async function addThree(x) {return x + 3;}
```

```
async function addFive(x) {return x + 5;}

async function addTen(x) {
    for (const fn of [addTwo, addThree, addFive]) {
        x = await fn(x);
    }
    return x;
}

addTen(9).then(console.log);    // 19
```

Трассировка стека и управление памятью

Промисы и асинхронные функции имеют значительную степень совпадения с точки зрения функциональности, которую они предоставляют, но они значительно расходятся, когда дело доходит до того, как они представлены в памяти. Рассмотрим следующий пример, который показывает чтение трассировки стека для отклоненного промиса:

```
function fooPromiseExecutor(resolve, reject) {
    setTimeout(reject, 1000, 'bar');
}

function foo() {
    new Promise(fooPromiseExecutor);
}

foo();
// Uncaught (in promise) bar
// setTimeout
// setTimeout (async)
// fooPromiseExecutor
// foo
```

Если вы основываетесь на полученном ранее понимании промисов, это чтение трассировки стека должно вас озадачить. Трассировка стека должна в буквальном смысле, представлять вложенную природу вызовов функций, которая существует в настоящее время в стеке памяти движка JavaScript. Когда обработчик тайм-аута выполняет и отклоняет промис, показанное сообщение об ошибке идентифицирует вложенные функции, которые были вызваны для первоначального создания экземпляра промиса. Однако известно, что эти функции *уже возвращены* и, следовательно, не будут найдены в трассировке стека.

Ответ прост: движок JavaScript выполняет дополнительную работу по сохранению стека вызовов, в то время как это возможно при создании структуры промиса. Когда выдается ошибка, этот стек вызовов извлекается логикой обработки ошибок среды выполнения и поэтому доступен в трассировке стека. Это, конечно, означает, что он должен сохранять трассировку стека в памяти и потребуются затраты на вычисления и хранение.

Рассмотрим предыдущий пример, как если он был переработан с помощью асинхронных функций:

```
function fooPromiseExecutor(resolve, reject) {
    setTimeout(reject, 1000, 'bar');
}

async function foo() {
    await new Promise(fooPromiseExecutor);
}
foo();

// Uncaught (in promise) bar
// foo
// async function (async)
// foo
```

С этой структурой трассировка стека точно представляет текущий стек вызовов, потому что `fooPromiseExecutor` вернулась и больше не находится в стеке, но `foo` приостановлена и еще не завершена. Среда выполнения JavaScript может просто хранить указатель от вложенной функции на ее контейнерную функцию, как это было бы с синхронным стеком вызовов функций. Этот указатель эффективно сохраняется в памяти и может использоваться для генерации трассировки стека в случае ошибки. Такая стратегия не влечет за собой дополнительных затрат, как в случае с предыдущим примером, и поэтому должна быть предпочтительна, если производительность имеет решающее значение для вашего приложения.

ИТОГИ

Освоение асинхронного поведения в однопоточной среде выполнения JavaScript долгое время было сложной задачей. С введением промисов в ES6 и async/await в ES7 асинхронные конструкции в ECMAScript были значительно улучшены. Промисы и async/await не только сделали доступными паттерны, которые ранее было трудно или невозможно реализовать, но и породили совершенно новый способ написания JavaScript, который стал чище, короче и проще для понимания и отладки.

Промисы созданы, чтобы предложить чистую абстракцию вокруг асинхронного кода. Они могут представлять асинхронно исполняемый блок кода, но также могут представлять асинхронно вычисленное значение. Они особенно полезны в ситуации, когда возникает необходимость сериализации блоков асинхронного кода. Промисы — это восхитительно податливая конструкция: они могут быть сериализованы, объединены в цепочку, составлены, расширены и объединены.

Асинхронные функции являются результатом применения парадигмы промисов к функциям JavaScript. Они вводят возможность приостановить выполнение функции, не блокируя основной поток выполнения. Они чрезвычайно полезны как при написании читаемого кода, ориентированного на промисы, так и при управлении сериализацией и распараллеливанием асинхронного кода. Они являются одной из наиболее важных вещей в современном JavaScript-инструментарии.

12

Объектная модель браузера

- Объект `window` – основа BOM
- Управление окнами, фреймами и всплывающими окнами
- Получение сведений о странице с помощью объекта `location`
- Получение сведений о браузере с помощью объекта `navigator`
- Управление стеком истории браузера с помощью объекта истории

Загрузки с сайта wrox.com для этой главы

Обратите внимание, что все примеры кода для этой главы доступны на веб-сайте книги по ссылке www.wrox.com/go/projavascript4e на вкладке Downloads.

Объектная модель браузера (Browser Object Model, BOM) описана в ECMAScript как ядро JavaScript, но правильнее было бы охарактеризовать ее как основу для использования JavaScript в интернете. BOM-объекты обеспечивают доступ к функционалу браузера независимо от контента веб-страницы. Тема BOM интересна и одновременно сложна, потому что из-за длительного отсутствия спецификации производители браузеров свободно расширяли BOM по своему усмотрению. Многие элементы, схожие в разных браузерах, стали стандартами де-факто и соблюдаются по сей день из соображений взаимной совместимости. Чтобы стандартизировать эти фундаментальные аспекты JavaScript, консорциум W3C определил основные BOM-элементы в спецификации HTML5.

ОБЪЕКТ WINDOW

В основе BOM лежит объект `window`, который представляет экземпляр браузера и имеет двойное назначение. С одной стороны, это JavaScript-интерфейс для доступа к окну браузера, а с другой — ECMAScript-объект `Global`. Это означает, что все объекты, переменные и функции, определенные в коде веб-страницы, используют объект `window` как глобальный и могут вызывать его методы, такие как `parseInt()`.

ПРИМЕЧАНИЕ Поскольку свойства объекта `window` доступны в глобальной области видимости, многие API-интерфейсы браузера и связанные конструкторы используют свойство объекта окна в качестве точки доступа. Эти API описаны в других местах книги, преимущественно в главе 20 «API в JavaScript».

Кроме того, некоторые свойства объекта окна будут существенно различаться между браузерами из-за различных реализаций поставщиков. В этой главе не рассматриваются устаревшие, нестандартизированные или специфичные для поставщика свойства `window`.

Глобальная область видимости

Поскольку объект `window` дублируется как ECMAScript-объект `Global`, все переменные и функции, объявленные глобально с помощью `var`, становятся его свойствами и методами, например:

```
var age = 29;
var sayAge = () => alert(this.age);

alert(window.age);        // 29
sayAge();                 // 29
window.sayAge();          // 29
```

При определении в глобальной области видимости переменная `age` и функция `sayAge()` автоматически добавляются к объекту `window`. Таким образом, переменная `age` доступна так же, как свойство `window.age`, а функция `sayAge()` — как `window.sayAge()`. Поскольку `sayAge()` существует в глобальной области видимости, вызов `this.age` проецируется на `window.age`.

Если вместо `var` используется `let` или `const`, прикрепления по умолчанию к глобальному объекту не происходит:

```
let age = 29;
const sayAge = () => alert(this.age);

alert(window.age);    // undefined
sayAge();             // undefined
window.sayAge();      // TypeError: window.sayAge is not a function
```

Попытка доступа к необъявленной переменной также приводит к ошибке, но ее можно предотвратить, проверив наличие потенциально необъявленной переменной у объекта `window`:

```
// ошибка, потому что переменная oldValue не объявлена
var newValue = oldValue;

// ошибки нет, потому что выполняется обращение к свойству
// newValue получает значение undefined
var newValue = window.oldValue;
```

Многие JavaScript-объекты, которые считаются глобальными, например `location` и `navigator` (оба они обсуждаются в этой главе), на самом деле являются свойствами объекта `window`.

Отношения окон

Объект `top` всегда указывает на самое верхнее (самое внешнее) окно, которым является само окно браузера. Другой объект `window` называется parent. Объект `parent` всегда указывает на непосредственное родительское окно текущего окна. Для самого верхнего окна браузера parent элемент равен `top` (и оба равны `window`). В верхнем `window` никогда не будет задано значение для `name`, если только окно не было открыто с помощью `window.open()`, как будет обсуждаться далее в этой главе.

Существует еще одно последнее свойство окна, называемое `self`, которое всегда указывает на `window`. Фактически эти два элемента могут использоваться взаимозаменяемо. Несмотря на то что оно не имеет отдельного значения, `self` включено для согласованности с объектами `top` и `parent`.

Каждый из этих объектов является свойством объекта `window`, доступного через `window.parent`, `window.top` и т. д. Это означает, что можно связать объекты `window` вместе, например `window.parent.parent`.

Положение окна и соотношение пикселей

Положение объекта `window` может быть определено и изменено с использованием различных свойств и методов. Все современные браузеры предоставляют свойства `screenLeft` и `screenTop`, которые указывают расположение окна по отношению к левой и верхней части экрана соответственно в пикселях CSS.

Также возможно переместить окно в новую позицию, используя методы `moveTo()` и `moveBy()`. Оба метода принимают два аргумента. `moveTo()` ожидает, что координаты `x` и `y` переместятся в абсолютную координату, а `moveBy()` ожидает, что число пикселей сместится относительно текущей координаты. Эти методы демонстрируются ниже:

```
// сдвиг окна в верхнюю левую точку
window.moveTo(0,0);

// сдвиг окна вниз на 100 пикселей
window.moveBy(0, 100);

// сдвиг окна на позицию (200, 300)
window.moveTo(200, 300);

// сдвиг окна влево на 50 пикселей
window.moveBy(-50, 0);
```

В зависимости от браузера эти методы могут быть условно или полностью отключены.

Соотношение пикселей

«CSS-пиксель» — это обозначение пикселя, универсально используемого в веб-разработке. Он определяется как угловое измерение: 0,0213°, примерно 1/96 дюйма на устройстве, удерживаемом на расстоянии вытянутой руки. Цель этого определения состоит в том, чтобы задать формальное значение размера пикселя для всех устройств: например, шрифт 12 пикселей (измеряется в пикселях CSS) на планшете с низким разрешением должен иметь размер, равный 12 пикселям (измеряется в пикселях CSS), шрифт на мониторе высокого разрешения 4k. В этом и заключается проблема: такая система требует встроенного масштабного коэффициента, необходимого для преобразования из физических пикселей (фактическое разрешение дисплея) в CSS-пиксели (виртуальное разрешение, сообщаемое веб-браузеру).

Например, экран телефона может иметь *физическое* разрешение 1920 × 1080, но поскольку эти пиксели невероятно малы, веб-браузер уменьшит это разрешение до меньшего *логического* разрешения, например 640 × 360. Этот коэффициент масштабирования предоставляется браузеру как `window.devicePixelRatio`. Для устройства, преобразующего 1920 × 1080 в 640 × 360, `devicePixelRatio` будет указано как 3. Таким образом, шрифт 12px в *физических* пикселях действительно будет шрифтом 36px в *логических* пикселях (или CSS-пикселях).

`devicePixelRatio`, по сути, является аналогом DPI (точек на дюйм). DPI эффективно записывает ту же информацию, но `devicePixelRatio` предлагает ее в безразмерном соотношении.

Размеры окна

Определить размеры окна без привязки к определенным браузерам непросто. Все современные браузеры предоставляют для этого свойства `innerWidth`, `innerHeight`, `outerWidth` и `outerHeight`. Свойства `outerWidth` и `outerHeight` возвращают размеры самого окна браузера (независимо от того, запрашиваются ли они у самого верхнего окна или у фрейма). В Opera эти значения определяют размеры области просмотра страницы. Свойства `innerWidth` и `innerHeight` возвращают размеры области просмотра страницы внутри окна браузера (не учитывая границы и панели инструментов).

Свойства `document.documentElement.clientWidth` и `document.documentElement.clientHeight` определяют ширину и высоту области просмотра страницы.

Короче говоря, надежно определить размеры самого окна браузера нельзя, но зато можно получить размеры области просмотра страницы:

```
let pageWidth = window.inюnerWidth,
    pageHeight = window.innerHeigh6666t;

if (typeof pageWidth != "number") {
    if (document.compatMode == "CSS1Compat") {
```

```
        pageWidth = document.documentElement.clientWidth;
        pageHeight = document.documentElement.clientHeight;
    } else {
        pageWidth = document.body.clientWidth;
        pageHeight = document.body.clientHeight;
    }
}
```

В этом коде переменным `pageWidth` и `pageHeight` присваиваются первоначальные значения `window.innerWidth` и `window.innerHeight` соответственно. Затем мы проверяем, является ли значение `pageWidth` числом; если нет, нужно проверить, работает ли браузер в стандартном режиме, для чего используется свойство `document.compatMode`. Если включен стандартный режим, используются значения `document.documentElement.clientWidth` и `document.documentElement.clientHeight`, в противном случае — значения `document.body.clientWidth` и `document.body.clientHeight`.

Для мобильных устройств свойства `window.innerWidth` и `window.innerHeight` определяют размеры визуальной области просмотра, то есть области страницы, видимой на экране. Internet Explorer для мобильных устройств не поддерживает эти свойства, но предоставляет ту же информацию в виде свойств `document.documentElement.clientWidth` и `document.documentElement.clientHeight`, значения которых изменяются при изменении масштаба страницы.

В других браузерах для мобильных устройств свойства объекта `document.documentElement` определяют размеры области просмотра макета, то есть фактические размеры визуализированной страницы (в отличие от визуальной области просмотра, которая охватывает только небольшую часть всей страницы). Internet Explorer для мобильных устройств хранит эти значения в свойствах `document.body.clientWidth` и `document.body.clientHeight`, которые остаются постоянными при изменении масштаба.

Из-за этих различий браузеров для мобильных устройств и настольных компьютеров перед выбором нужных свойств следует сначала определить систему, с которой работает пользователь.

> **ПРИМЕЧАНИЕ** Тема областей просмотра на мобильных устройствах непроста и имеет много нюансов и исключений. Петер-Пол Кох (Peter-Paul Koch), консультант по разработке мобильных приложений, раскрыл ее в своей статье, доступной на сайте http://quirksmode.org/mobile/viewports2.html. Ознакомьтесь с ней, если вы разрабатываете приложения для мобильных устройств.

Размеры окна браузера можно изменить с помощью методов `resizeTo()` и `resizeBy()`, которые принимают по два аргумента. Метод `resizeTo()` принимает новые значения ширины и высоты, а `resizeBy()` — изменения каждого размера, например:

```
// задание размеров 100 x 100
window.resizeTo(100, 100);

// задание размеров 200 x 150
```

```
window.resizeBy(100, 150);

// задание размеров 300 x 300
window.resizeTo(300, 300);
```

Как и методы перемещения окна, эти методы по умолчанию отключены в некоторых браузерах и работают только с самым верхним объектом `window`.

Положение области просмотра окна

Поскольку окно браузера обычно недостаточно велико для одновременного отображения всего визуализированного документа, пользователю предоставляется возможность прокручивать документ с ограниченным окном просмотра. Смещение в пикселях CSS текущего просматриваемого экрана доступно в виде пары X и Y, представляющей количество пикселей, в которых в данный момент просматриваемый экран прокручивается в этом направлении. Смещения X и Y доступны через два свойства, которые возвращают одинаковые значения: `window.pageXoffset/window.scrollX` и `window.pageYoffset/window.scrollY`.

Также можно явно прокрутить страницу на определенную величину, используя несколько разных оконных методов. Этим методам передаются две координаты, указывающие, как далеко в направлении X и Y должна прокручиваться область просмотра. `window.scroll(x, y)` прокрутит окно просмотра на относительную величину (`window.scrollBy(x, y)` ведет себя так же). `window.scrollTo(x, y)` прокручивает область просмотра до абсолютного смещения.

```
// Прокрутка вниз на 100 пикселей относительно текущей области просмотра окна
window.scroll(0, 100);

// Прокрутка вправо 40 пикселей относительно текущей области просмотра окна
window.scroll(40, 0);

// Прокрутка к левому верхнему углу страницы
// window.scrollTo(0, 0);

// Прокрутка на 100 пикселей от верхней и левой границ страницы
// window.scrollTo(100, 100);
```

Эти методы также принимают словарь `ScrollToOptions`, который в дополнение к значениям смещения может дать указание браузеру сгладить прокрутку с помощью свойства `behavior`.

```
// обычная прокрутка
window.scrollTo({
    left: 100,
    top: 100,
    behavior: 'auto'
});

// сглаженная прокрутка
window.scrollTo({
```

```
    left: 100,
    top: 100,
    behavior: 'smooth'
});
```

Открытие окон и навигация

Метод `window.open()` позволяет перейти по указанному URL-адресу и открыть новое окно браузера. Он принимает четыре аргумента: URL-адрес страницы, которую нужно загрузить, целевое окно, строку параметров и логическое значение, указывающее, должна ли новая страница заменить текущую в журнале браузера. Обычно используют только три первых аргумента; последний указывают, если не нужно открывать новое окно.

Если вторым аргументом метода `window.open()` является имя уже существующего окна или фрейма, страница по указанному URL-адресу загружается в это окно или фрейм, например:

```
//то же, что и <a href="http://www.wrox.com" target="topFrame"></a>
window.open("http://www.wrox.com/", "topFrame");
```

Выполнение этого кода аналогично щелчку на ссылке, у которой атрибут `href` имеет значение `"http://www.wrox.com"`, а атрибут `target` — `"topFrame"`. При наличии окна с именем `"topFrame"` страница загружается в него, в противном случае создается новое окно с именем `"topFrame"`. Вторым аргументом также может быть одно из специальных имен окон: `_self`, `_parent`, `_top` или `_blank`.

Всплывающие окна

Если второй аргумент метода `window.open` не соответствует именам существующих окон, метод создает окно или вкладку на основе строки, переданной ему в качестве третьего аргумента. Если этот аргумент отсутствует, метод открывает в браузере новое окно или вкладку (в зависимости от того, как настроен браузер) с параметрами, предлагаемыми по умолчанию. Элементы окна или вкладки, такие как панели инструментов, адресная строка и строка состояния, также отображаются согласно параметрам, предлагаемым по умолчанию. Если указано, что новое окно открывать не следует, третий аргумент игнорируется.

Третьим аргументом является строка параметров отображения нового окна, разделенных запятыми. Допустимые параметры указаны в таблице.

ПАРАМЕТР	ЗНАЧЕНИЕ	ОПИСАНИЕ
`fullscreen`	`"yes"` или `"no"`	Указывает, нужно ли создать окно браузера развернутым во весь экран (работает только в Internet Explorer)
`height`	Число	Начальная высота нового окна. Не может быть меньше 100

ПАРАМЕТР	ЗНАЧЕНИЕ	ОПИСАНИЕ
`left`	Число	Начальная левая координата нового окна. Не может быть отрицательным числом
`location`	`"yes"` или `"no"`	Указывает, нужно ли отобразить адресную строку. Значение по умолчанию зависит от браузера. Если задано значение «no», адресная строка может быть либо скрыта, либо отключена в зависимости от браузера
`menubar`	`"yes"` или `"no"`	Указывает, нужно ли отобразить панель меню. По умолчанию «no»
`resizable`	`"yes"` или `"no"`	Указывает, можно ли изменять размеры нового окна, перетаскивая его границы. По умолчанию «no»
`scrollbars`	`"yes"` или `"no"`	Указывает, можно ли прокручивать новое окно, если контент не помещается в области просмотра. По умолчанию «no»
`status`	`"yes"` или `"no"`	Указывает, нужно ли отобразить строку состояния. Значение по умолчанию зависит от браузера
`toolbar`	`"yes"` или `"no"`	Указывает, нужно ли отобразить панель инструментов. По умолчанию «no»
`top`	Число	Начальная верхняя координата нового окна. Не может быть отрицательным числом
`width`	Число	Начальная ширина нового окна. Не может быть меньше 100

Любые из этих параметров можно указать как набор разделенных запятыми пар имен и значений. Имя и значение в каждой паре разделяются знаком равенства (пробелы в строке параметров не допускаются). Вот пример:

```
window.open("http://www.wrox.com/",
            "wroxWindow",
            "height=400,width=400,top=10,left=10,resizable=yes");
```

Этот код в 10 пикселях от верхнего и левого краев экрана открывает новое окно с размером 400 × 400, который можно изменять.

Метод `window.open()` возвращает ссылку на созданное окно. Это такой же объект `window`, как и любые другие, только обычно лучше контролируемый. Например, браузеры, которые по умолчанию не позволяют изменять размеры главного окна или перемещать его, могут разрешать это для окон, созданных методом `window.open()`. Используя возвращенный объект, можно управлять новым открытым окном так же, как и любым другим, например:

```
let wroxWin = window.open("http://www.wrox.com/",
                          "wroxWindow",
                          "height=400,width=400,top=10,left=10,resizable=yes");
```

```
// изменение размеров окна
wroxWin.resizeTo(500, 500);

// перемещение окна
wroxWin.moveTo(100, 100);
```

Закрыть новое окно можно, вызвав метод `close()`:

```
wroxWin.close();
```

Этот метод работает только со всплывающими окнами, созданными методом `window.open()`. Закрыть главное окно браузера без подтверждения пользователя невозможно. Однако всплывающие окна могут закрывать себя сами без подтверждения пользователя, вызывая метод `top.close()`. После закрытия окна ссылка на него остается доступной, но годится только для проверки свойства `closed`:

```
wroxWin.close();
alert(wroxWin.closed);    // true
```

Созданное окно ссылается на окно, которое его открыло, с помощью свойства `opener`. Оно определено только для самого верхнего объекта `window` (то есть `top`) всплывающего окна и представляет собой указатель на окно или фрейм, для которого был вызван метод `window.open()`, например:

```
let wroxWin = window.open("http://www.wrox.com/",
                          "wroxWindow",
                          "height=400,width=400,top=10,left=10,resizable=yes");

alert(wroxWin.opener === window);    // true
```

У всплывающего окна есть указатель на исходное окно, но обратное неверно. Окна не следят за тем, какие окна они породили, так что при необходимости вы сами должны их отслеживать.

Некоторые браузеры пытаются запускать отдельный процесс для каждой новой вкладки. Когда одна вкладка открывает другую, объектам `window` нужно взаимодействовать друг с другом, поэтому в такой ситуации вкладки не могут выполняться в разных процессах. В этих браузерах можно указать, что для вкладки следует создать отдельный процесс, присвоив свойству `opener` значение `null`:

```
let wroxWin = window.open("http://www.wrox.com/",
                          "wroxWindow",
                          "height=400,width=400,top=10,left=10,resizable=yes");

wroxWin.opener = null;
```

Присвоение значения `null` свойству `opener` указывает браузеру, что новой вкладке не нужно взаимодействовать с исходной, так что она может работать как отдельный процесс. После разрыва связи между вкладками восстановить ее невозможно.

Ограничения безопасности

В свое время по интернету прокатилась эпидемия рекламных всплывающих окон, которые часто выдавали за системные диалоговые окна, чтобы пользователи не

могли распознать недобросовестную рекламу. В ответ на это производители браузеров начали ограничивать возможности настройки всплывающих окон.

В ранней версии Internet Explorer были реализованы такие меры безопасности, как запрет на создание всплывающих окон и их перемещение за пределы экрана, а также на отключение строки состояния. Начиная с Internet Explorer 7, по умолчанию запрещено отключать адресную строку, перемещать всплывающие окна и изменять их размеры. В Firefox 1 была отключена возможность блокировать строку состояния, из-за чего она отображалась во всех всплывающих окнах независимо от строки параметров, переданной в метод `window.open()`. В Firefox 3 то же самое было сделано для адресной строки. Opera открывает всплывающие окна только в главном окне браузера, но блокирует их при опасности спутать их с системными диалоговыми окнами.

Кроме того, браузеры разрешают создание всплывающего окна только после действия пользователя. Например, вместо вызова метода `window.open()` во время загрузки страницы может быть выведено сообщение об ошибке, потому что всплывающие окна разрешено открывать только в ответ на щелчок мышью или нажатие клавиши.

ПРИМЕЧАНИЕ Internet Explorer снимает некоторые ограничения, связанные со всплывающими окнами, при отображении веб-страниц, сохраненных на жестком диске компьютера. Если тот же код запускается с сервера, ограничения снова вступают в силу.

Блокирование всплывающих окон

Все современные браузеры содержат встроенные средства блокирования всплывающих окон. Так или иначе, большинство непредвиденных всплывающих окон блокируются, при этом происходит одно из двух. Если окно блокируется средством, встроенным в браузер, метод `window.open()` обычно возвращает значение `null`, по которому можно узнать, что случилась блокировка, например:

```
let wroxWin = window.open("http://www.wrox.com", "_blank");
if (wroxWin == null) {
    alert("The popup was blocked!");
}
```

Если всплывающее окно блокируется надстройкой браузера или другой программой, метод `window.open()` обычно генерирует ошибку. Следовательно, чтобы правильно определить, что всплывающее окно было заблокировано, необходимо проверить значение, возвращаемое методом `window.open()`, и заключить его вызов в блок `try-catch`:

```
let blocked = false;

try {
    let wroxWin = window.open("http://www.wrox.com", "_blank");
    if (wroxWin == null) {
        blocked = true;
```

```
    }
} catch (ex) {
    blocked = true;
}

if (blocked) {
    alert("The popup was blocked!");
}
```

Этот код определяет, что вызов `window.open()` был заблокирован любым из указанных способов.

> **ПРИМЕЧАНИЕ** Проверка блокировки окон не препятствует браузеру вывести на экран собственное сообщение о блокировании всплывающего окна.

Интервалы и тайм-ауты

JavaScript работает в браузере в однопоточном режиме, но позволяет планировать выполнение кода в конкретные моменты времени с помощью тайм-аутов и интервалов. Тайм-ауты используются для запуска кода после указанного промежутка времени, а интервалы — для периодического запуска.

Тайм-аут можно задать с помощью метода `window.setTimeout()`, который принимает два аргумента: выполняемый код и интервал (в миллисекундах), по прошествии которого нужно запланировать запуск функции обратного вызова. Первым аргументом может быть либо строка с JS-кодом (как при использовании метода `eval()`), либо функция. Например:

```
// запланировать показ уведомления спустя 1 секунду
setTimeout(() => alert("Hello world!"), 1000);
```

Второй аргумент, время ожидания в миллисекундах, не определяет точный момент выполнения указанного фрагмента. Поскольку JS-код выполняется в однопоточном режиме, в каждый момент обрабатывается только одна инструкция. Для управления выполнением кода используется очередь задач JavaScript, которые запускаются в том же порядке, в каком были добавлены в очередь. Второй аргумент метода `setTimeout()` указывает интерпретатору JavaScript добавить задачу в очередь через указанное время. Если очередь пуста, этот код выполняется незамедлительно, в противном случае он должен дождаться своей очереди.

Метод `setTimeout()` возвращает числовой идентификатор тайм-аута. Он уникально идентифицирует запланированный код и позволяет отменить тайм-аут. Чтобы отменить тайм-аут, который еще не был обработан, вызовите метод `clearTimeout()`, передав ему идентификатор тайм-аута, например:

```
// задание тайм-аута
let timeoutId = setTimeout(() => alert("Hello world!"), 1000);

// тайм-аут больше не нужен
clearTimeout(timeoutId);
```

Если метод `clearTimeout()` вызывается до истечения указанного времени, тайм-аут полностью отменяется. Вызов `clearTimeout()` после выполнения запланированного кода ни на что не влияет.

> **ПРИМЕЧАНИЕ** Весь запланированный код тайм-аута с использованием обычной анонимной функции выполняется в глобальной области видимости, так что объект `this` внутри функции всегда указывает на `window` в нестрогом режиме и имеет значение `undefined` в строгом. Когда вместо `setTimeout` предоставляется стрелочная функция, это сохраняет лексическую область, в которой она была определена.

Интервал работает подобно тайм-ауту, только код запускается периодически через указанные промежутки времени до отмены интервала или до выгрузки страницы. Задать интервал можно с помощью метода `setInterval()`, который принимает те же аргументы, что и `setTimeout()`, то есть выполняемый код в виде строки или функции и период добавления ее обратного вызова в очередь выполнения в миллисекундах. Рассмотрим пример:

```
setInterval(() => alert("Hello world!"), 10000);
```

> **ПРИМЕЧАНИЕ** Важно отметить, что интервал времени, указанный во втором аргументе, представляет собой промежуток времени, в течение которого браузер будет ожидать между добавлением нового обратного вызова в очередь. Например, предположим, что вы вызвали `setInterval()` ровно в 01:00:00 с интервалом 3000 мс. Это означает, что в 01:00:03 браузер запланирует обратный вызов. Браузеру все равно, когда выполняется обратный вызов или сколько времени это займет; он запланирует еще один на 01:00:06. Отсюда следует, что короткие и неблокирующие функции обратного вызова идеально подходят для `setInterval`.

Метод `setInterval()` возвращает идентификатор интервала, который можно использовать для отмены интервала с помощью метода `clearInterval()`. Для интервалов это важнее, чем для тайм-аутов, потому что интервал, оставленный без присмотра, будет запускать код вплоть до выгрузки страницы. Вот пример типичного применения интервала:

```
let num = 0, intervalId = null;
let max = 10;

let incrementNumber = function() {
    num++;

    // при достижении значения max интервал отменяется
    if (num == max) {
        clearInterval(intervalId);
        alert("Done");
    }
}

intervalId = setInterval(incrementNumber, 500);
```

Переменная num увеличивается здесь каждые полсекунды, пока не достигает максимального значения, после чего интервал отменяется. Этот прием также можно реализовать с помощью тайм-аутов:

```
let num = 0;
let max = 10;

let incrementNumber = function() {
    num++;

    //если значение max не достигнуто, задается новый тайм-аут
    if (num < max) {
        setTimeout(incrementNumber, 500);
    } else {
        alert("Done");
    }
}

setTimeout(incrementNumber, 500);
```

Заметьте, что при использовании тайм-аутов не требуется отслеживать идентификатор тайм-аута, потому что выполнение кода прекращается само по себе и продолжается только в случае задания нового тайм-аута. Этот прием считается наилучшим способом задания интервалов, хотя сами интервалы при этом не используются. Настоящие интервалы применяются в окончательном коде редко, потому что браузеры не гарантируют точность времени между окончанием одного интервала и началом другого и могут пропускать интервалы. Использование тайм-аутов, как в последнем примере, гарантирует, что этого не случится. Как правило, интервалов лучше избегать.

Системные диалоговые окна

С помощью методов `alert()`, `confirm()` и `prompt()` можно отображать в браузерах системные диалоговые окна. Эти окна не связаны с веб-страницей, отображаемой в браузере, и не содержат HTML-код, а их вид определяется параметрами операционной системы и (или) браузера, но не стилями CSS. Каждое из этих диалоговых окон является синхронным и модальным, то есть выполнение кода приостанавливается на время показа диалогового окна и возобновляется после его закрытия.

Метод `alert()` вы уже много раз видели в книге. Он просто принимает строку, которую нужно показать пользователю. В отличие от `console.log`, который может принимать переменное количество аргументов и отображать их все одновременно, `alert` ожидает только один аргумент. При вызове `alert()` появляется системное окно сообщения с указанным оповещением и кнопкой OK. Если в `alert()` передан аргумент, который не является строковым примитивом, то он будет преобразован в строку с помощью метода `.toString()`.

Диалоговые окна оповещений обычно применяют, если нужно уведомить пользователя о чем-то, что он не контролирует, например об ошибке. Единственное, что может сделать пользователь, это закрыть диалоговое окно после прочтения оповещения, как показано на рис. 12.1.

Рис. 12.1

С помощью метода `confirm()` можно вывести на экран диалоговое окно запроса подтверждения. Оно также отображает сообщение для пользователя, но помимо кнопки OK содержит кнопку Cancel (Отмена), благодаря чему пользователь может подтвердить или отменить некоторое действие. Например, вызов `confirm("Are you sure?")` выводит на экран диалоговое окно запроса подтверждения, показанное на рис. 12.2.

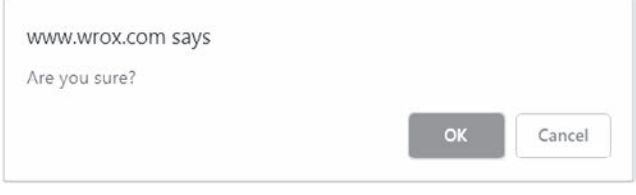

Рис. 12.2

Чтобы можно было определить, какую кнопку выбрал пользователь, метод `confirm()` возвращает `true`, если был щелчок на кнопке OK, и `false`, если он щелкнул на кнопке Cancel (Отмена) или закрыл диалоговое окно щелчком на системной кнопке закрытия окна в его углу (со значком X). Типичный код вызова этого диалогового окна выглядит так:

```
if (confirm("Are you sure?")) {
    alert("I'm so glad you're sure! ");
} else {
    alert("I'm sorry to hear you're not sure. ");
}
```

В этом примере диалоговое окно запроса подтверждения отображается при выполнении условия инструкции `if`. Если пользователь щелкает на кнопке OK, появляется оповещение `"I'm so glad you're sure!"` (Я рад, что вы уверены), если же он выбирает кнопку Cancel (Отмена), выводится оповещение `"I'm sorry to hear you're not sure"` (Жаль, что вы не уверены). Этот шаблонный код может использоваться, когда пользователь пытается что-либо удалить, например сообщение электронной почты. Поскольку диалог полностью нарушит работу пользователя на странице, этот способ должен быть зарезервирован только для действий, которые имеют тяжелые последствия.

Наконец, метод `prompt()` выводит на экран диалоговое окно, которое запрашивает у пользователя информацию. Вместе с кнопками OK и Cancel (Отмена) оно содержит текстовое поле для ввода данных. Метод `prompt()` принимает два аргумента: текст, который нужно показать пользователю, и значение, указанное в текстовом поле по умолчанию (которое может быть пустой строкой). Так, вызов `prompt("What's your name?", "Michael")` выводит на экран диалоговое окно, показанное на рис. 12.3.

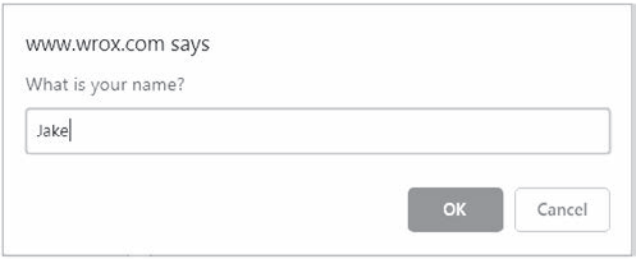

Рис. 12.3

Если пользователь щелкает на кнопке OK, метод `prompt()` возвращает значение, введенное в текстовом поле, а если выбирается кнопка Cancel (Отмена) или окно закрывается иным образом (без щелчка на кнопке OK), метод возвращает `null`. Вот пример:

```
let result = prompt("What's your name? ", "");
if (result !== null) {
    alert("Welcome, " + result);
}
```

Такие системные диалоговые окна полезны, если нужно вывести для пользователя некоторую информацию и запросить подтверждение решения. Не требуя загрузки кода HTML, CSS или JavaScript, они позволяют быстро и просто улучшить веб-приложение.

Многие браузеры с появлением системных диалоговых окон стали предлагать особую функциональность. Если выполняемый сценарий выводит на экран хотя бы два системных диалоговых окна, в каждое окно после первого добавляется флажок, позволяющий отключить вывод последующих диалоговых окон до перезагрузки страницы.

Если установить флажок и закрыть диалоговое окно, последующие системные диалоговые окна трех описанных типов будут блокироваться до перезагрузки страницы. Разработчику не сообщается о том, появилось ли диалоговое окно. При простое браузера счетчик диалоговых окон сбрасывается, так что если два отдельных действия пользователя привели к выводу оповещения, флажок не появляется ни в одном из них; если одно действие пользователя привело к показу двух оповещений подряд, второе окно будет содержать флажок.

В JavaScript доступны также диалоговые окна поиска и печати, которые выводятся асинхронно, тут же возвращая управление сценарию. Это те же окна, которые появляются, когда пользователь выбирает в меню браузера команду поиска или печати. Вывести на экран их можно с помощью методов `find()` и `print()` объекта `window`:

```
// отображение диалогового окна печати
window.print();

// отображение диалогового окна поиска
window.find();
```

Эти методы не показывают, что пользователь сделал с диалоговым окном, так что трудно найти для них полезное применение. Поскольку окна асинхронны, они не влияют на счетчик диалоговых окон в браузерах и на них не распространяется запрет на вывод последующих диалоговых окон.

ОБЪЕКТ LOCATION

Объект `location` считается одним из наиболее полезных в BOM. Он предоставляет сведения о текущем загруженном документе и обеспечивает общий функционал навигации. Он уникален тем, что является свойством и `window`, и `document`, то есть свойства `window.location` и `document.location` указывают на один и тот же объект. С помощью объекта `location` можно не только получить сведения о текущем загруженном документе, но и выполнить синтаксический анализ URL-адреса, разобрав его на отдельные сегменты, доступные в качестве свойств. Эти свойства приведены в следующей таблице (префикс `location` не указан).

Если в браузере в данный момент открыта страница по адресу http://foouser:barpassword@www.wrox.com:80/WileyCDA/?Q=javascript#content, то объект `location` будет вести себя следующим образом:

ИМЯ СВОЙСТВА	ПРИМЕР	ОПИСАНИЕ
`location.hash`	`"#contents"`	Хеш URL-адреса (знак решетки и любое количество других знаков) или пустая строка, если у URL-адреса нет хеша
`location.host`	`"www.wrox.com:80"`	Имя сервера и номер порта при его наличии
`location.hostname`	`"www.wrox.com"`	Имя сервера без номера порта
`location.href`	`"http://www.wrox.com:80/WileyCDA/?q=javascript#contents"`	Полный URL-адрес текущей загруженной страницы. Метод `toString()` объекта `location` возвращает это значение

ИМЯ СВОЙСТВА	ПРИМЕР	ОПИСАНИЕ
location.pathname	"/WileyCDA/"	Каталог и (или) имя файла в URL-адресе
location.port	"80"	Порт запроса, если он указан в URL-адресе. Если URL-адрес не содержит порт, это свойство возвращает пустую строку
location.protocol	"http:"	Протокол доступа к странице (обычно "http:" или "https:")
location.search	"?q=javascript"	Строка запроса в URL-адресе. Это свойство возвращает строку, которая начинается с вопросительного знака
location.username	"foouser"	Имя пользователя, указанное перед доменным именем
location.password	"barpassword"	Пароль, указанный перед доменным именем
location.origin	"http://www.wrox.com"	Происхождение URL. Только для чтения

Аргументы строки запроса

Большинство элементов объекта `location` можно легко получить с помощью этих свойств, но работать с необработанной строкой запроса неудобно. Хотя свойство `location.search` возвращает все, начиная от вопросительного знака и до конца URL-адреса, аргументы строки запроса по отдельности недоступны. Следующая функция разбирает строку запроса и возвращает объект, содержащий отдельные аргументы:

```
let getQueryStringArgs = function() {
    // получение строки запроса без начального вопросительного знака
    let qs = (location.search.length > 0 ?
              location.search.substring(1) : ""),
        // объект для хранения аргументов
        args = {};

    // запись каждого элемента в объект args
    for (let item of qs.split("&").map(kv => kv.split("="))) {
        let name = decodeURIComponent(item[0]),
            value = decodeURIComponent(item[1]);
        if (name.length) {
            args[name] = value;
        }
    }

    return args;
}
```

Первым делом эта функция удаляет начальный вопросительный знак из строки запроса, если свойство `location.search` содержит один или более знаков. Аргументы сохраняются в объекте `args`, который создается с помощью литерала объекта. Затем функция делит строку запроса по знаку «амперсанда» и сохраняет результат в массиве строк формата `имя=значение`. Цикл `for` делит каждый элемент этого массива по знаку равенства, записывая в новый массив имя аргумента и его значение как первый и второй элемент соответственно. Предполагается, что строка запроса закодирована, поэтому далее эти элементы декодируются с помощью метода `decodeURIComponent()` и присваиваются переменным `name` и `value`. Наконец, `name` добавляется к объекту `args` как свойство со значением `value`. Эту функцию можно использовать следующим образом:

```
// предполагается, что указана строка запроса ?q=javascript&num=10

let args = getQueryStringArgs();

alert(args["q"]);          // "javascript"
alert(args["num"]);        // "10"
```

Аргументы строки запроса теперь представлены свойствами возвращенного объекта, что обеспечивает быстрый доступ к каждому аргументу.

URLSearchParams

`URLSearchParams` предлагает набор служебных методов, которые позволяют проверять и изменять параметры запроса с использованием стандартизированного API. Экземпляр `URLSearchParams` создается путем передачи строки запроса в конструктор. Экземпляр предоставляет различные методы, такие как `get()`, `set()` и `delete()`, для выполнения операций строки запроса:

```
let qs = "?q=javascript&num=10";

let searchParams = new URLSearchParams(qs);

alert(searchParams.toString());    // " q=javascript&num=10"
searchParams.has("num");           // true
searchParams.get("num");           // 10

searchParams.set("page", "3");
alert(searchParams.toString());    // " q=javascript&num=10&page=3"

searchParams.delete("q");
alert(searchParams.toString());    // " num=10&page=3"
```

Большинство браузеров, поддерживающих `URLSearchParams`, также поддерживают использование `URLSearchParams` в качестве итерируемого объекта:

```
let qs = "?q=javascript&num=10";

let searchParams = new URLSearchParams(qs);

for (let param of searchParams) {
```

```
    console.log(param);
}
// ["q", "javascript"]
// ["num", "10"]
```

Работа с объектом location

Открыть в браузере другую страницу с помощью объекта `location` можно несколькими способами. Первый, наиболее популярный, — это вызвать метод `assign()` с URL-адресом в качестве аргумента, например:

```
location.assign("http://www.wrox.com");
```

Этот метод немедленно инициирует переход по новому URL-адресу и создает запись в стеке журнала браузера. При назначении URL-адреса свойству `location.href` или `window.location` также вызывается метод `assign()` с указанным значением. Например, обе следующие инструкции эквивалентны явному вызову метода `assign()`:

```
window.location = "http://www.wrox.com";
location.href = "http://www.wrox.com";
```

Из этих трех способов в коде чаще всего встречается последний.

Для изменения текущей загруженной страницы можно использовать свойства `hash`, `search`, `hostname`, `pathname` и `port` объекта `location`, например:

```
// предполагается начальный адрес http://www.wrox.com/WileyCDA/

// изменение URL-адреса на "http://www.wrox.com/WileyCDA/#section1"
location.hash = "#section1";

// изменение URL-адреса на "http://www.wrox.com/WileyCDA/?q=javascript"
location.search = "?q=javascript";

// изменение URL-адреса на "http://www.yahoo.com/WileyCDA/"
location.hostname = "www.yahoo.com";

// изменение URL-адреса на "http://www.yahoo.com/mydir/"
location.pathname = "mydir";

// изменение URL-адреса на "http://www.yahoo.com:8080/WileyCDA/
Location.port = "8080"
```

Каждый раз, когда изменяется свойство объекта `location` (исключая `hash`), загружается страница с новым URL-адресом.

> **ПРИМЕЧАНИЕ** Изменение значения `hash` приводит к сохранению новой записи в журнале браузера. В более ранних версиях Internet Explorer свойство `hash` обновлялось только при щелчке на ссылке с хешированным URL-адресом, но не на кнопке Back (Назад) или Forward (Вперед).

При изменении URL-адреса одним из описанных способов в стеке журнала браузера сохраняется запись, чтобы пользователь мог вернуться к предыдущей странице, щелкнув в браузере на кнопке Back (Назад). Такое поведение можно запретить с помощью метода `replace()`, который выполняет переход по переданному ему URL-адресу, но не сохраняет запись в стеке журнала. После вызова `replace()` пользователь не может вернуться к предыдущей странице. Рассмотрим пример:

```
<!DOCTYPE html>
<html>
<head>
    <title>You won't be able to get back here</title>
</head>
    <body>
    <p>Enjoy this page for a second, because you won't be coming back here.</p>
    <script>
        setTimeout(() => location.replace("http://www.wrox.com/"), 1000);
    </script>
</body>
</html>
```

Если загрузить эту страницу в веб-браузере, через секунду будет выполнен переход на сайт www.wrox.com. При этом кнопка Back (Назад) окажется недоступной и вы не сможете вернуться на страницу примера без повторного ввода полного URL-адреса.

Метод `location.reload()` перезагружает текущую страницу. Если вызвать его без аргументов, страница перезагружается наиболее эффективным образом (из кеша браузера, если она не была изменена с момента последнего запроса). Чтобы перезагрузить страницу с сервера, передайте в метод значение `true`:

```
location.reload();        // перезагрузка – возможно, из кеша
location.reload(true);    // перезагрузка с сервера
```

При значительных задержках в сети или недостатке системных ресурсов код после вызова `reload()` может быть не выполнен, так что этот метод лучше вызывать в последней строке кода.

ОБЪЕКТ NAVIGATOR

Объект `navigator`, представленный в Netscape Navigator 2, обеспечивает стандартный способ идентификации браузера в клиентской системе. Он является общим для всех веб-браузеров с поддержкой JavaScript. Как и с другими объектами спецификации, каждый браузер поддерживает свой собственный набор свойств.

> **ПРИМЕЧАНИЕ** Свойства объекта `navigator`, дающие представление о возможностях системы, подробно описаны в главе 13 «Распознавание клиента».

Объект `navigator` реализует методы и свойства, определенные в интерфейсах `NavigatorID`, `NavigatorLanguage`, `NavigatorOnLine`, `NavigatorContentUtils`,

`NavigatorStorage`, `NavigatorStorageUtils`, `NavigatorConcurrentHardware`, `NavigatorPlugins` и `NavigatorUserMedia`.

В следующей таблице перечислены все доступные свойства и методы.

СВОЙСТВО/МЕТОД	ОПИСАНИЕ
`activeVrDisplays`	Возвращает массив каждого экземпляра `VRDisplay` со свойством `ispresenting`, установленным в значение `true`
`appCodeName`	Имя браузера. Обычно `"Mozilla"` – даже для браузеров не от Mozilla
`appName`	Полное имя браузера
`appVersion`	Версия браузера. Обычно не соответствует фактической версии браузера
`battery`	Возвращает объект `BatteryManager` для взаимодействия с API состояния батареи
`buildID`	Номер сборки браузера
`connection`	Возвращает объект `NetworkInformation` для взаимодействия с Network Information API
`cookieEnabled`	Указывает, включена ли поддержка cookie-файлов
`credentials`	`CredentialsContainer` для взаимодействия с Credentials Management API
`deviceMemory`	Объем памяти устройства в гигабайтах
`doNotTrack`	Пользовательское предпочтение – без отслеживания
`geolocation`	Объект `Geolocation` для взаимодействия с Geolocation API
`getVRDisplays()`	Возвращает массив со всеми доступными экземплярами `VRDisplay`
`getUserMedia()`	Возвращает поток, связанный с доступным оборудованием мультимедийного устройства
`hardwareConcurrency`	Количество ядер процессора
`javaEnabled()`	Этот метод указывает, включена ли в браузере поддержка Java
`language`	Основной язык браузера
`languages`	Массив всех предпочтительных языков браузера
`locks`	Объект `LockManager` для взаимодействия с Web Locks API
`mediaCapabilities`	Объект `MediaCapabilities` для взаимодействия с Media Capabilities API

СВОЙСТВО/МЕТОД	ОПИСАНИЕ
`mediaDevices`	Доступные мультимедийные устройства
`maxTouchPoints`	Максимальное количество поддерживаемых точек касания для сенсорного экрана устройства
`mimeTypes`	Массив MIME-типов, зарегистрированных в браузере
`onLine`	Указывает, подключен ли браузер к интернету
`oscpu`	Операционная система и (или) процессор устройства, на котором работает браузер
`permissions`	Объект `Permissions` для взаимодействия с Permissions API
`platform`	Системная платформа, на которой работает браузер
`plugins`	Массив подключаемых модулей, установленных в браузере. В Internet Explorer это массив всех элементов `<embed>` на странице
`product`	Имя продукта (обычно `Gecko`)
`productSub`	Дополнительные сведения о продукте (обычно сведения о версии Gecko)
`registerProtocolHandler()`	Этот метод регистрирует веб-сайт как обработчик для конкретного протокола
`requestMediaKeySystemAccess()`	Возвращает `Promise`, который разрешается в объект `MediaKeySystemAccess`
`sendBeacon()`	Асинхронно передает небольшую полезную нагрузку
`serviceWorker`	`ServiceWorkerContainer`, используемый для взаимодействия с объектами `ServiceWorker`
`share()`	Если доступно, вызывает собственный механизм совместного использования текущей платформы
`storage`	Возвращает объект `StorageManager` для взаимодействия с `Storage` API
`userAgent`	Строка пользовательского агента для браузера
`vendor`	Производитель браузера
`vendorSub`	Дополнительные сведения о производителе
`vibrate()`	Запускает вибрацию на устройстве, если поддерживается вибрация
`webdriver`	Указывает, управляется ли браузер с помощью автоматизации

Свойства объекта `navigator` обычно используются для определения типа браузера, который обрабатывает веб-страницу.

Обнаружение подключаемых модулей

Нередко требуется выяснить, установлен ли в браузере конкретный подключаемый модуль. В браузерах, отличных от Internet Explorer 10 и более старых версий, это можно сделать с помощью массива `plugins`. Каждый его элемент имеет следующие свойства:

- `name` — имя подключаемого модуля;
- `description` — описание подключаемого модуля;
- `filename` — имя файла подключаемого модуля;
- `length` — количество MIME-типов, обрабатываемых подключаемым модулем.

Обычно свойства `name` достаточно для обнаружения подключаемого модуля, но гарантировать этого нельзя. Чтобы идентифицировать подключаемый модуль, доступные модули перебираются в цикле, а их имена сравниваются с указанным именем, как в этом примере:

```
// обнаружение подключаемого модуля – не работает в Internet Explorer 10 и ниже
let hasPlugin = function(name) {
    name = name.toLowerCase();
    for ((let plugin of window.navigator.plugins) {
        if (plugin[i].name.toLowerCase().indexOf(name) > -1) {
            return true;
        }
    }

    return false;
}

// обнаружение подключаемого модуля Flash
alert(hasPlugin("Flash"));

// обнаружение подключаемого модуля QuickTime
alert(hasPlugin("QuickTime"));
```

Метод `hasPlugin()` принимает в качестве аргумента имя искомого подключаемого модуля, которое тут же преобразуется в нижний регистр, чтобы упростить сравнение. Затем свойство `name` каждого элемента массива `plugins` проверяется с помощью метода `indexOf()` на предмет того, содержит ли оно переданное имя. Во избежание ошибок сравнение выполняется в нижнем регистре. Чтобы не было путаницы, аргумент метода должен быть как можно более специфичным. Строки `"Flash"` и `"QuickTime"` достаточно уникальны, чтобы проблем не возникло. Этот метод обнаруживает подключаемые модули в Firefox, Safari, Opera и Chrome.

В выпуске Internet Explorer 11 `window.navigator` поддерживает `plugins` и `mimeTypes`. Это означает, что функция, определенная ранее, способна правильно обнаруживать

подключаемые модули для всех современных версий браузеров от основных поставщиков. Кроме того, `ActiveXObject` становится скрытым от DOM в IE11, это означает, что его нельзя использовать для целей обнаружения.

ПРИМЕЧАНИЕ Каждый объект `plugin` является также массивом объектов `MimeType`, доступных с помощью скобочной нотации. Каждый объект `MimeType` имеет четыре свойства: `description` – описание MIME-типа; `enabledPlugin` – указатель на объект `plugin`; `suffixes` – строка разделенных запятыми расширений файлов для MIME-типа; `type` – полная строка MIME-типа.

Обнаружение подключаемых модулей в устаревшем Internet Explorer

В браузере Internet Explorer идентифицировать подключаемые модули сложнее, потому что в нем они работают иначе. Единственный способ решить эту задачу в Internet Explorer — попытаться создать экземпляр конкретного подключаемого модуля с помощью фирменного типа `ActiveXObject`. Подключаемые модули реализованы в Internet Explorer как COM-объекты, для идентификации которых используются уникальные строки. Таким образом, чтобы проверить конкретный подключаемый модуль, нужно знать его COM-идентификатор. Например, Flash имеет идентификатор `"ShockwaveFlash.ShockwaveFlash"`. Располагая этой информацией, для обнаружения подключаемого модуля в Internet Explorer можно использовать следующую функцию:

```
// обнаружение подключаемого модуля в устаревшем Internet Explorer
function hasIEPlugin(name) {
    try {
        new ActiveXObject(name);
        return true;
    } catch (ex) {
        return false;
    }
}

// обнаружение подключаемого модуля Flash
alert(hasIEPlugin("ShockwaveFlash.ShockwaveFlash"));

// обнаружение подключаемого модуля QuickTime
alert(hasIEPlugin("QuickTime.QuickTime"));
```

Функция `hasIEPlugin()` принимает в качестве единственного аргумента COM-идентификатор и пытается создать экземпляр `ActiveXObject`. Этот код заключен в блок `try-catch`, потому что попытка создать неизвестный COM-объект приводит к ошибке. Если создать объект удается, функция возвращает `true`, в противном случае выполняется блок `catch` и возвращается значение `false`. Две последние инструкции в коде проверяют, доступны ли в Internet Explorer подключаемые модули Flash и QuickTime.

Поскольку эти два универсальных подхода так сильно различаются, обычно на их основе создают функции, проверяющие наличие конкретных, а не любых подключаемых модулей, например:

```
// обнаружение подключаемого модуля Flash в любых браузерах
function hasFlash() {
    var result = hasPlugin("Flash");
    if (!result) {
        result = hasIEPlugin("ShockwaveFlash.ShockwaveFlash");
    }
    return result;
}

// обнаружение подключаемого модуля QuickTime в любых браузерах
function hasQuickTime() {
    var result = hasPlugin("QuickTime");
    if (!result) {
        result = hasIEPlugin("QuickTime.QuickTime");
    }
    return result;
}

// обнаружение подключаемого модуля Flash
alert(hasFlash());

// обнаружение подключаемого модуля QuickTime
alert(hasQuickTime());
```

Функции `hasFlash()` и `hasQuickTime()` выполняют сначала метод обнаружения подключаемого модуля в браузерах, отличных от Internet Explorer. Если этот метод возвращает `false`, вызывается метод обнаружения подключаемого модуля в Internet Explorer. Если он также возвращает `false`, то и результат всего метода равен `false`. Если какой-либо из методов обнаружения подключаемого модуля возвращает `true`, результат всего метода равен `true`.

ПРИМЕЧАНИЕ Метод `refresh()` коллекции `plugins` обновляет ее согласно сведениям о новых установленных подключаемых модулях. Он принимает один аргумент: логическое значение, указывающее, нужно ли перезагрузить страницу. Если аргумент равен `true`, все страницы с подключаемыми модулями перезагружаются, иначе коллекция `plugins` обновляется без перезарузки страницы.

Регистрация обработчиков

В современных браузерах к объекту `navigator` был добавлен метод `registerProtocolHandler()`, который теперь формально определен в HTML 5. С его помощью можно указать, что веб-сайт способен обрабатывать данные конкретных типов — это позволяет направлять их по умолчанию в соответствующие программы, такие как онлайновые средства чтения RSS-каналов и почтовые веб-приложения.

Запрос можно сделать для протоколов с помощью метода `registerProtocolHandler()`, который принимает три аргумента: протокол (например, `"mailto"` или `"ftp"`), URL-адрес страницы с обработчиком протокола и имя приложения. Например, следующий код регистрирует веб-приложение как почтовый клиент, предлагаемый по умолчанию:

```
navigator.registerProtocolHandler("mailto",
    "http://www.somemailclient.com?cmd=%s",
    "Some Mail Client");
```

После выполнения этого кода протокол `mailto` будет обрабатываться почтовым веб-клиентом. Как и в предыдущем примере, вторым аргументом здесь является URL-адрес страницы обработчика, а `%s` представляет оригинальный запрос.

ОБЪЕКТ SCREEN

Объект `screen` (который также является свойством `window`) — один из немногих JavaScript-объектов, которые практически не используются в коде. Он просто предоставляет сведения о графических параметрах клиентской системы вне окна браузера, таких как ширина и высота в пикселях. Доступность тех или иных свойств объекта `screen` зависит от браузера. Эти свойства указаны в следующей таблице.

СВОЙСТВО	ОПИСАНИЕ
`availHeight`	Высота экрана в пикселях за вычетом системных элементов, таких как панель задач Windows (только для чтения)
`availLeft`	Первый пиксель слева, не занятый системными элементами (только для чтения)
`availTop`	Первый пиксель сверху, не занятый системными элементами (только для чтения)
`availWidth`	Ширина экрана в пикселях за вычетом системных элементов (только для чтения)
`colorDepth`	Количество битов, используемых для представления цветов; 32 в большинстве систем (только для чтения)
`height`	Высота экрана в пикселях
`left`	Смещение левой стороны текущего экрана в пикселях
`pixelDepth`	Глубина цвета в битах (только для чтения)
`top`	Смещение верхней стороны экрана в пикселях
`width`	Ширина экрана в пикселях
`orientation`	Возвращает ориентацию экрана, как указано в Screen Orientation API

ОБЪЕКТ HISTORY

Объект `history` представляет журнал навигации за все время работы с конкретным окном. Так как это свойство объекта `window`, у каждого окна браузера есть собственный объект `history`. Из соображений безопасности браузеры не позволяют определять URL-адреса страниц, которые посещал пользователь, но можно перемещаться по их списку вперед и назад, не зная URL-адреса.

Навигация

Метод `go()` позволяет перемещаться по журналу пользователя в обоих направлениях и принимает один аргумент: количество страниц, на которое нужно перейти назад или вперед. Если значение отрицательное, выполняется переход назад, как при щелчке на кнопке Back (Назад) в браузере, а если положительное — вперед, как при щелчке на кнопке Forward (Вперед), например:

```
// переход к предыдущей странице
history.go(-1);

// переход к следующей странице
history.go(1);

// переход на две страницы вперед
history.go(2);
```

Аргумент метода `go()` может также быть строкой — в этом случае браузер переходит назад или вперед к ближайшей странице, адрес которой содержит эту строку. Если в журнале нет записи, соответствующей строке, метод ничего не делает, например:

```
// переход к ближайшей странице wrox.com
history.go("wrox.com");

// переход к ближайшей странице nczonline.net
history.go("nczonline.net");
```

Вместо `go()` можно использовать методы `back()` и `forward()`, которые имитируют щелчки на кнопках Back (Назад) и Forward (Вперед) в браузере:

```
// возврат на одну страницу
history.back();

// переход к следующей странице
history.forward();
```

У объекта `history` есть также свойство `length`, которое указывает общее количество элементов в стеке журнала. Первой странице, загруженной в окно или на вкладку, соответствует нулевое значение `history.length`. Проверив это свойство, можно определить, имеем ли мы дело с начальной страницей сеанса:

```
if (history.length == 0) {
    // это первая страница в окне пользователя
}
```

Обычно с помощью объекта `history` создают пользовательские кнопки Back (Назад) и Forward (Вперед), а также определяют, является ли страница первой в журнале пользователя. В HTML5 функционал объекта `history` расширен.

ПРИМЕЧАНИЕ Записи в стеке журнала создаются при изменении URL-адреса страницы, а в основных версиях браузеров, выпущенных после 2009 г., еще и при изменении хеша URL-адреса (то есть установка свойства `location.hash` приводит к вставке новой записи в стек журнала). Это поведение обычно используется одностраничными фреймворками, которые хотят имитировать функциональность кнопок Назад и Вперед, не вызывая перезагрузки полной страницы при каждом событии навигации.

Управление состоянием истории

Одним из наиболее сложных аспектов программирования веб-приложений является управление историей. Прошли те времена, когда каждое действие переводило пользователя на совершенно новую страницу, это также означает, что кнопки Назад и Вперед были отняты у пользователей как привычный способ сказать «переведи меня в другое состояние». Первым шагом к решению этой проблемы было событие `hashchange` (обсуждаемое в главе 17 «События»). HTML5 обновляет объект `history`, чтобы обеспечить простое управление состоянием.

Если событие `hashchange` просто сообщает, когда хеш URL-адреса изменился, и ожидает, что вы будете действовать соответствующим образом, API управления состоянием фактически позволяет изменять URL-адрес браузера, не загружая новую страницу. Для этого используется метод `history.pushState()`. Этот метод принимает три аргумента: объект данных, заголовок нового состояния и необязательный относительный URL. Например:

```
let stateObject = {foo:"bar"};

history.pushState(stateObject, "My title", "baz.html");
```

Как только `pushState()` выполняется, информация о состоянии помещается в стек истории, и адресная строка браузера изменяется, отражая новый относительный URL. Несмотря на это изменение, браузер не отправляет запрос на сервер, даже если запрос `location.href` вернет именно то, что находится в адресной строке. Второй аргумент в настоящее время не используется никакими реализациями, поэтому можно либо оставить его в виде пустой строки, либо предоставить короткий заголовок. Первый аргумент должен содержать всю информацию, необходимую для правильной инициализации этого состояния страницы при необходимости. Во избежание злоупотреблений размер объекта состояния ограничен — обычно менее 500 МБ — 1 МБ.

Поскольку `pushState()` создает новую запись в истории, вы заметите, что кнопка Назад включена. При нажатии кнопки Назад событие `popstate` запускается в объекте

window. Объект event для popstate имеет свойство state, которое содержит объект, переданный в pushState() в качестве первого аргумента:

```
window.addEventListener("popstate", (event) => {
    let state = event.state;
    if (state) { // state имеет значение null при первой загрузке страницы
        processState(state);
    }
});
```

Используя это состояние, нужно затем самостоятельно сбросить страницу в состояние, представленное данными в объекте состояния (так как браузер не делает это автоматически). Помните, что при первой загрузке страницы состояние отсутствует, поэтому нажатие кнопки Назад до тех пор, пока вы не перейдете в исходное состояние страницы, приведет к тому, что event.state будет иметь значение null.

Можно получить доступ к текущему состоянию объекта с помощью history.state. Также можно обновить информацию о текущем состоянии, используя replaceState() и передавая те же первые два аргумента, что и в pushState(). Этот метод не создает новую запись в истории, а просто перезаписывает текущее состояние:

```
history.replaceState({newFoo: "newBar"}, "New title");
```

Объект state, передаваемый в pushState() или replaceState(), должен содержать только информацию, которую можно сериализовать. Поэтому такие вещи, как элементы DOM, не подходят для использования в объекте состояния.

> **ПРИМЕЧАНИЕ** При использовании управления состоянием истории HTML5 убедитесь, что любой «поддельный» URL-адрес, созданный с помощью pushState(), поддерживается на реальном физическом URL-адресе на веб-сервере. В противном случае нажатие кнопки Обновить приведет к ошибке 404. Все фреймворки для одностраничных приложений (SPA) должны каким-то образом решить эту проблему посредством настроек на сервере или клиенте.

ИТОГИ

Объектная модель браузера (BOM) основана на объекте window, который представляет окно браузера и видимую область страницы. В ECMAScript он дублируется как объект Global, так что все глобальные переменные и функции становятся его свойствами, а все встроенные конструкторы и функции изначально относятся к нему. В этой главе мы обсудили ряд BOM-элементов, перечисленных далее.

- Ссылаться на другие объекты окна можно с помощью нескольких указателей на объекты window.
- Объект location обеспечивает программный доступ к системе навигации браузера. Задавая его свойства, можно изменять URL-адреса по частям или полностью.

- Метод `replace()` позволяет перейти по новому URL-адресу и заменить текущую страницу в журнале браузера.
- Объект `navigator` предоставляет сведения о браузере. Доступные сведения во многом зависят от используемого браузера, хотя некоторые свойства, такие как `userAgent`, поддерживаются во всех браузерах.

Два других объекта, доступных в BOM, имеют очень ограниченное применение. Объект `screen` возвращает сведения о дисплее клиентской системы, которые иногда собираются на веб-сайтах как метрики. Объект `history` предоставляет некоторые возможности для работы со стеком журнала браузера. С его помощью можно определить количество сайтов в стеке журнала и перейти назад или вперед к любой странице в журнале, а также изменить стек истории.

13

Распознавание клиента

- Распознавание возможностей
- История распознавания пользовательского агента
- Обнаружение программного и аппаратного обеспечения
- Выбор способа распознавания

Производители браузеров прилагают немалые усилия для согласования способов взаимодействия с ними, но все же каждый браузер имеет уникальные особенности. Даже релизы одного браузера для разных платформ часто имеют важные различия, хотя технически это одна версия. Из-за этих различий веб-разработчики вынуждены либо использовать возможности, общие для всех целевых браузеров, либо распознавать браузер и обходить его ограничения. Обычно применяется второй подход.

Распознавание клиента остается одним из самых спорных аспектов веб-разработки, при этом большинство сходится во мнении, что браузеры должны поддерживать некую общую для всех функциональность. Возможно, в идеальном мире так бы и было, но в реальности различий и особенностей браузеров достаточно для того, чтобы рассматривать распознавание клиента как неотъемлемый элемент стратегии разработки. Различия между браузерами сегодня гораздо менее выражены, чем фиаско Internet Explorer прошлых лет, но несоответствия браузеров остаются общей темой в веб-разработке.

Есть несколько подходов, которые можно использовать для определения используемого веб-клиента, каждый со своими достоинствами и недостатками, но распознавание клиента должно быть самым последним вариантом решения проблемы. Если возможен более общий подход, используйте его. Разработайте сначала наиболее универсальное решение, а затем расширьте его возможностями, специфичными для конкретных браузеров.

РАСПОЗНАВАНИЕ ВОЗМОЖНОСТЕЙ

Обнаружение возможностей (capability detection) (также называемое *обнаружением функций*) использует набор простых проверок во время выполнения JavaScript в браузере для проверки поддержки различных функций. Это предполагает, что идентифицировать конкретный браузер не требуется — достаточно выяснить, доступна ли нужная функциональность. Базовая схема распознавания возможностей такова:

```
if (object.propertyInQuestion) {
    // использование object.propertyInQuestion
}
```

Например, DOM-метод `document.getElementById()` в Internet Explorer до версии 5 недоступен, но для решения той же задачи можно использовать нестандартное свойство `document.all`. Распознать эти возможности можно так:

```
function getElement(id) {
    if (document.getElementById) {
        return document.getElementById(id);
    } else if (document.all) {
        return document.all[id];
    } else {
        throw new Error("No way to retrieve element!");
    }
}
```

Функция `getElement()` возвращает элемент с указанным идентификатором. Обычно для этого используется функция `document.getElementById()`, которая и проверяется в первую очередь. Если эта функция существует, она вызывается, в противном случае проверяется наличие свойства `document.all`. Если ни один из этих способов не доступен (что крайне маловероятно), генерируется ошибка.

При распознавании возможностей нужно помнить о двух важных принципах. Как уже отмечалось, типичный способ получения результата следует проверять первым. Так, в предыдущем примере метод `document.getElementById()` проверяется перед свойством `document.all`. Это оптимизирует выполнение кода, предотвращая проверку лишних условий в типичных ситуациях.

Второй важный принцип заключается в том, что проверять нужно в точности ту возможность, которая вам требуется. То, что доступна одна возможность, не означает, что доступна другая, например:

```
function getWindowWidth() {
    if (document.all) {        // предполагается, что используется IE
        return document.documentElement.clientWidth;    // НЕПРАВИЛЬНО!!!
    } else {
        return window.innerWidth;
    }
}
```

В этом примере распознавание возможностей применяется неправильно. Функция `getWindowWidth()` сначала проверяет, доступно ли свойство `document.all`, и если да,

возвращает значение `document.documentElement.clientWidth`, иначе возвращается значение `window.innerWidth`, которое не поддерживается в Internet Explorer 8 и более ранних версий. Проблема в том, что наличие свойства `document.all` не всегда указывает, что браузером является Internet Explorer. Им также может быть ранняя версия Opera, в которой поддерживаются свойства `document.all` и `window.innerWidth`.

Надежное распознавание возможностей

Для надежного распознавания возможности не всегда достаточно проверить ее доступность — надо еще убедиться, что она работает надлежащим образом. В предыдущем разделе для определения доступности элементов выполняется приведение их типов, но это не гарантирует, что это действительно те элементы, которые нам нужны. Рассмотрим следующую функцию, которая определяет, поддерживает ли объект сортировку:

```
// НЕ ДЕЛАЙТЕ ТАК! Неправильное распознавание возможности –
// проверяется только существование элемента
function isSortable(object) {
    return !!object.sort;
}
```

Чтобы определить, можно ли отсортировать объект, эта функция проверяет, есть ли у него метод `sort()`. Проблема в том, что любой объект со свойством `sort` тоже возвращает `true`:

```
let result = isSortable({ sort: true });
```

Поскольку наличие свойства не гарантирует, что объект поддерживает сортировку, лучше проверить, является ли элемент `sort` функцией:

```
// Лучше – код проверяет, является ли sort функцией
function isSortable(object) {
    return typeof object.sort == "function";
}
```

В этом коде оператор `typeof` определяет, является ли элемент `sort` функцией, которой можно воспользоваться для сортировки данных в объекте.

Распознавание возможностей с помощью `typeof` — более надежный подход, но и он не гарантирует правильный результат. В частности, объекты среды не обязаны возвращать осмысленные значения при вызове `typeof`. Наиболее вопиющий пример имеет место в Internet Explorer. В большинстве браузеров, где доступен метод `document.createElement()`, следующая функция возвращает `true`:

```
// работает неправильно в Internet Explorer до версии 8 включительно
function hasCreateElement() {
    return typeof document.createElement == "function";
}
```

Однако в Internet Explorer 8 и более ранних версий эта функция возвращает `false`, потому что выражение `typeof document.createElement` интерпретируется как `"object"`, а не `"function"`. Как уже отмечалось, DOM-объекты являются объектами

среды, которые в Internet Explorer 8 и более ранних версий реализованы с помощью COM, а не JScript. Это относится и к функции `document.createElement()`, поэтому для нее оператор `typeof` возвращает `"object"`. Internet Explorer 9 возвращает для DOM-методов значение `"function"`.

ПРИМЕЧАНИЕ Подробное обсуждение распознавания возможностей в JavaScript см. в статье Питера Мишо «Feature Detection: State of the Art Browser Scripting» («Распознавание возможностей: современные сценарии для браузеров») по адресу http://peter.michaux.ca/articles/feature-detection-state-of-the-art-browser-scripting.

Использование распознавания возможностей для анализа браузера

Хотя некоторые могут считать, что обнаружение возможностей является чем-то вроде взлома, правильное применение обнаружения возможностей может привести к удивительно точному анализу браузера, выполняющего код. Преимущество использования обнаружения возможностей по сравнению с пользовательскими агентами для идентификации браузера заключается в том, что, хотя подделать пользовательский агент исключительно легко, его гораздо сложнее скрыть всеми возможностями браузера таким образом, чтобы надежно обмануть обнаружение возможностей.

Обнаружение поддержки функций

Можно сгруппировать возможности вместе в классы браузеров. Если вы знаете, что вашему приложению нужно использовать определенные функции браузера, может быть полезно выполнить обнаружение всех возможностей за один раз, а не делать это повторно. Рассмотрим этот пример:

```
// определяет, содержит ли браузер подключаемые модули в стиле Netscape
let hasNSPlugins = !!(navigator.plugins && navigator.plugins.length);

// определяет, содержит ли браузер базовые возможности DOM Level 1
let hasDOM1 = !!(document.getElementById && document.createElement &&
    document.getElementsByTagName);
```

В этом примере выполняется два обнаружения: одно — чтобы увидеть, поддерживает ли браузер подключаемые модули в стиле Netscape, и другое — чтобы определить, поддерживает ли браузер базовые возможности DOM Level 1. Эти логические значения могут быть запрошены позже, и для повторного тестирования возможностей потребуется меньше времени.

Обнаружение идентичности браузера

Также можно протестировать браузер на предмет поддержки различных функций и сравнить результаты с известной коллекцией, чтобы определить, какой именно

браузер используется. Это учитывает схему обнаружения, которая защищена от подмены пользовательского агента (обсуждается позже в этой главе), но будущие версии браузеров могут нарушить используемую схему обнаружения возможностей.

Рассмотрим следующий пример, который проверяет известное уникальное поведение различных браузеров, чтобы определить браузер, внутри которого выполняется код. Этот код намеренно не использует `navigator.userAgent`, который обсуждается далее в этой главе:

```
class BrowserDetector {
    constructor() {
        // Тест условной компиляции
        // Поддерживается в IE6-10
        this.isIE_Gte6Lte10 = /*@cc_on!@*/false;

        // Тест присутствия documentMode
        // Поддерживается в IE7-11
        this.isIE_Gte7Lte11 = !!document.documentMode;

        // Тест присутствия конструктора StyleMedia
        // Поддерживается в Edge >= 20
        this.isEdge_Gte20 = !!window.StyleMedia;

        // Тест на наличие патентованного API установки дополнения Firefox
        // Поддерживается во всех версиях Firefox
        this.isFirefox_Gte1 = typeof InstallTrigger !== 'undefined';

        // Тест на наличие объекта Chrome и его свойства webstore. Версии
        // Opera будут иметь window.chrome, но нewindow.chrome.webstore
        // Поддерживается во всех версиях Chrome
        this.isChrome_Gte1 = !!window.chrome && !!window.chrome.webstore;

        // Ранние версии Safari будут добавлять "Constructor" к идентификатору
        // функции конструктора.
        // window.Element.toString(); // [object ElementConstructor]
        // Поддерживается в Safari 3-9.1
        this.isSafari_Gte3Lte9_1 = /constructor/i.test(window.Element);

        // API всплывающих уведомлений появляется на объекте window. Использует
        // параметры по умолчанию, чтобы предупредить приведение значений undefined
        // к строке
        // Поддерживается в Safari 7.1+
        this.isSafari_Gte7_1 =
            (({pushNotification = {}} = {}) =>
                pushNotification.toString() == '[object SafariRemoteNotification]'
            )(window.safari);

        // Проверка на наличие свойства 'addons.
        // Поддерживается в Opera 20+
        this.isOpera_Gte20 = !!window.opr && !!window.opr.addons;
    }

    isIE() { return this.isIE_Gte6Lte10 || this.isIE_Gte7Lte11; }
```

```
    isEdge() { return this.isEdge_Gte20 && !this.isIE(); }
    isFirefox() { return this.isFirefox_Gte1; }
    isChrome() { return this.isChrome_Gte1 };
    isSafari() { return this.isSafari_Gte3Lte9_1 || this.isSafari_Gte7_1; }
    isOpera() { return this.isOpera_Gte20; }
```

Использование такого класса позволяет предоставлять обобщенные методы обнаружения браузера, которые объединяют возможности обнаружения для разных диапазонов браузеров. По мере того как браузеры меняются и развиваются, можно настраивать обнаружение базовых возможностей, оставив основной API без изменений.

Ограничения обнаружения возможностей

Обнаружение конкретной возможности или набора возможностей необязательно указывает на используемый браузер. Следующий код «обнаружения браузера» или что-то подобное можно найти на многочисленных веб-сайтах, это является примером неправильного обнаружения возможностей:

```
// ИЗБЕГАЙТЕ ЭТОГО! Код недостаточно специфичен
let isFirefox = !!(navigator.vendor && navigator.vendorSub);

// ИЗБЕГАЙТЕ ЭТОГО! Код делает слишком много допущений
let isIE = !!(document.all && document.uniqueID);
```

Этот код представляет собой классическое неправильное использование возможностей обнаружения. В прошлом Firefox мог быть определен путем проверки на `navigator.vendor` и `navigator.vendorSub`, но затем появился Safari и реализовал те же свойства; это означает, что этот код сработает ложно. Чтобы обнаружить Internet Explorer, код проверяет наличие `document.all` и `document.uniqueID`. Это предполагает, что оба свойства продолжат существовать в будущих версиях IE и никогда не будут реализованы любым другим браузером. Обе проверки используют двойной оператор NOT для получения логического результата (который более оптимален для хранения и доступа).

> **ПРИМЕЧАНИЕ** Распознавание возможностей лучше всего использовать для определения следующего шага в решении, необязательно в качестве флага, указывающего, что используется конкретный браузер.

РАСПОЗНАВАНИЕ ПОЛЬЗОВАТЕЛЬСКОГО АГЕНТА

Распознавание пользовательского агента (user-agent detection), при котором для определения информации о том, какой браузер используется, применяется его строка пользовательского агента. Она отправляется как заголовок ответа при каждом HTTP-запросе и доступна в JavaScript в виде свойства `navigator.userAgent`. На стороне сервера ее часто используют для идентификации браузера с целью выбора тех или иных действий, но на стороне клиента распознавание пользовательского

агента следует считать ненадежным и использовать только в сценариях, когда другие варианты невозможны.

Одна из сомнительных сторон этого подхода — долгая история *спуфинга* (spoofing). Спуфингом называют искажения строки пользовательского агента с целью ввести в заблуждение сервер. Чтобы понять эту проблему, необходимо обсудить историю применения и изменения строки пользовательского агента.

История композиции пользовательского агента

В спецификации HTTP версий 1.0 и 1.1 указано, что браузер должен отправлять короткую строку пользовательского агента, содержащую имя и версию браузера. В RFC 2616 (спецификация протокола http 1.1) строка пользовательского агента описана следующим образом:

> *Маркеры продуктов требуются для идентификации взаимодействующих приложений по имени и версии. В большинстве полей, где используются маркеры продуктов, можно также указывать через пробел субпродукты, формирующие значительные части приложения. Продукты указываются согласно их значимости для идентификации приложения.*

Далее спецификация предписывает задавать строку пользовательского агента как список продуктов в формате «маркер/версия продукта». Однако в реальности строки пользовательских агентов никогда не были такими простыми.

Ранние браузеры

Первый веб-браузер, Mosaic, был выпущен в 1993 г. Национальным центром суперкомпьютерных приложений (National Center for Supercomputing Applications, NCSA). Его строка пользовательского агента имела совсем простой формат:

```
Mosaic/0.9
```

Строка могла различаться в зависимости от операционной системы и платформы, но ничего сложного в ней не было: перед слешем указывалось название продукта (иногда как NCSA Mosaic или что-то подобное), а после — версия продукта.

Разработчики из Netscape Communications присвоили своему веб-браузеру кодовое название Mozilla (сокращение от Mosaic Killer — «убийца Mosaic»). В Netscape Navigator 2, первой общедоступной версии нового браузера, строка пользовательского агента имела следующий формат:

```
Mozilla/версия[язык] (платформа; шифрование)
```

Компания Netscape оставила название и версию продукта как начало строки пользовательского агента, но добавила позднее следующую информацию:

- **язык** — код языка, указывающий, где предполагается использовать браузер;
- **платформа** — операционная система и (или) платформа, на которой работает браузер;

- **шифрование** — разновидность шифрования; возможные значения — U (128-разрядное шифрование), I (40-разрядное шифрование) и N (нет шифрования).

Типичная строка пользовательского агента Netscape Navigator 2 выглядела следующим образом:

```
Mozilla/2.02 [fr] (WinNT; I)
```

Эта строка определяет браузер Netscape Navigator 2.02 для франкоязычных стран, запущенный на компьютере с системой Windows NT с 40-разрядным шифрованием. В целом можно было легко идентифицировать браузер, просто прочитав название продукта в строке пользовательского агента.

Netscape Navigator 3 и Internet Explorer 3

В 1996 г. был выпущен браузер Netscape Navigator 3, ставший на некоторое время самым популярным. Строка пользовательского агента претерпела в нем небольшие изменения: маркер языка из нее исчез, но были добавлены необязательные сведения об операционной системе или процессоре. Формат строки стал таким:

```
Mozilla/версия (платформа; шифрование [; описание ОС или ЦП])
```

Типичная строка пользовательского агента Netscape Navigator 3 в системе Windows выглядела так:

```
Mozilla/3.0 (Win95; U)
```

Эта строка соответствует Netscape Navigator 3 в системе Windows 95 со 128-разрядным шифрованием. Как видите, при работе под управлением Windows описание ОС или ЦП отсутствовало.

Вскоре после выпуска Netscape Navigator 3 корпорация Microsoft представила свой первый общедоступный веб-браузер Internet Explorer 3. Поскольку браузер Netscape тогда доминировал на рынке, многие серверы явно идентифицировали его, прежде чем отправлять страницы. Проблемы с доступом к страницам в Internet Explorer помешали бы его распространению, поэтому в Microsof выбрали для строки пользовательского агента формат, совместимый со строкой Netscape:

```
Mozilla/2.0 (compatible; версия MSIE; операционная система)
```

Например, у Internet Explorer 3.02 в системе Windows 95 была такая строка пользовательского агента:

```
Mozilla/2.0 (compatible; MSIE 3.02; Windows 95)
```

Для распознавания браузеров в то время обычно проверялось только имя продукта в строке пользовательского агента, поэтому Internet Explorer успешно выдавал себя за Mozilla, подражая Netscape Navigator. Не всем это понравилось, потому что такой подход нарушал конвенцию об идентификации браузеров, к тому же настоящая версия браузера была скрыта в середине строки.

Эта строка интересна еще и номером версии Mozilla. Казалось бы, вместо номера 2.0 логичнее было бы использовать 3.0, ведь именно эта версия была тогда наиболее популярной. Причина принятого решения остается загадкой. Скорее всего, это была банальная оплошность.

Netscape Communicator 4 и Internet Explorer 4–8

В августе 1997 г. был представлен браузер Netscape Communicator 4 (в этом выпуске название браузера было изменено с *Navigator* на *Communicator*). Формат строки пользовательского агента остался в нем таким же, каким был в версии 3:

```
Mozilla/версия (платформа; шифрование [; описание ОС или ЦП])
```

В версии 4 на компьютере с системой Windows 98 строка пользовательского агента выглядела так:

```
Mozilla/4.0 (Win98; I)
```

По мере выпуска исправлений для браузера его версия увеличивалась. Вот, например, строка пользовательского агента для версии 4.79:

```
Mozilla/4.79 (Win98; I)
```

В Internet Explorer 4 строка пользовательского агента также содержала обновленную версию:

```
Mozilla/4.0 (compatible; версия MSIE; операционная система)
```

Например, браузер Internet Explorer 4 в Windows 98 возвращал следующую строку пользовательского агента:

```
Mozilla/4.0 (compatible; MSIE 4.0; Windows 98)
```

С этим изменением возвращаемая версия Mozilla и фактическая версия Internet Explorer совпали, что позволяет легко идентифицировать эти браузеры четвертого поколения. К сожалению, вскоре версии снова разошлись. Когда вышел браузер Internet Explorer 4.5 (доступный только для Mac), версия Mozilla осталась прежней, а версия Internet Explorer изменилась:

```
Mozilla/4.0 (compatible; MSIE 4.5; Mac_PowerPC)
```

В Internet Explorer этот формат использовался вплоть до версии 7:

```
Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1)
```

В Internet Explorer 8 был представлен дополнительный маркер Trident — имя визуализатора. Формат стал следующим:

```
Mozilla/4.0 (compatible; версия MSIE; операционная система;
             Trident/версия Trident)
```

Пример строки:

```
Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 5.1; Trident/4.0)
```

Маркер Trident позволяет определить, когда Internet Explorer 8 работает в режиме совместимости. В этом случае версия MSIE равна 7, но строка пользовательского агента содержит версию Trident:

```
Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; Trident/4.0)
```

Благодаря этому маркеру также можно отличить браузер Internet Explorer 7 (в котором нет маркера Trident) от браузера Internet Explorer 8, работающего в режиме совместимости.

В Internet Explorer 9 версии Mozilla и Trident были увеличены до 5.0. Строка пользовательского агента по умолчанию выглядит в Internet Explorer 9 так:

```
Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; Trident/5.0)
```

Когда Internet Explorer 9 работает в режиме совместимости, восстанавливаются старые версии Mozilla и MSIE, а версия Trident остается равной 5.0. Например, Internet Explorer 9 в режиме совместимости с Internet Explorer 7 имеет следующую строку пользовательского агента:

```
Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 6.1; Trident/5.0)
```

Все эти изменения были внесены, чтобы расширить возможности передачи сведений о клиенте, не нарушив при этом работу прежних сценариев распознавания пользовательского агента.

Gecko

Визуализатор Gecko лежит в основе Firefox. Первоначально он был частью универсального браузера Mozilla, который позднее стал Netscape 6. Для Netscape 6 была разработана спецификация, определяющая формат строки пользовательского агента во всех будущих версиях. Новый формат существенно отличался от простой строки пользовательского агента, которая применялась до версии 4.x включительно, и был таким:

```
Mozilla/версия Mozilla (платформа; шифрование; ОС или ЦП; язык;
    предварительная версия)Gecko/версия Gecko
    приложение/версия приложения
```

Смысл каждого маркера в этой строке описан в следующей таблице.

МАРКЕР	ОБЯЗАТЕЛЬНОСТЬ	ОПИСАНИЕ
Версия Mozilla	Да	Версия Mozilla
Платформа	Да	Платформа, на которой работает браузер. Возможные значения включают Windows, Mac и X11 (для X-windows в Unix)

МАРКЕР	ОБЯЗАТЕЛЬНОСТЬ	ОПИСАНИЕ
Шифрование	Да	Разновидность шифрования: U для 128-разрядного, I для 40-разрядного или N, если шифрование не используется
ОС ил ЦП	Да	Операционная система, в которой работает браузер, или тип процессора компьютера, на котором запущен браузер. Если платформа – Windows, это версия Windows (например, WinNT, Win95 и т. д.). Если платформа – Macintosh, это тип ЦП (68k, PPC для PowerPC или MacIntel). Если платформа – X11, это имя операционной системы Unix, возвращаемое командой `uname –sm`
Язык	Да	Язык, для которого создан браузер
Предварительная версия	Нет	Первоначально – предварительный номер версии Mozilla. Теперь – номер версии визуализатора Gecko
Версия Gecko	Да	Версия визуализатора Gecko, представленная датой в формате *ггггммдд*
Приложение	Нет	Название продукта, в котором используется Gecko. Им может быть Netscape, Firefox и т. д.
Версия приложения	Нет	Версия продукта, в котором используется Gecko; не путайте этот параметр с версией Mozilla и версией Gecko

Чтобы лучше понять формат строки пользовательского агента Gecko, взгляните на ее примеры из различных браузеров, основанных на Gecko.

Netscape 6.21 в Windows XP:

```
Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:0.9.4) Gecko/20011128
    Netscape6/6.2.1
```

SeaMonkey 1.1a в Linux:

```
Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.8.1b2) Gecko/20060823
    SeaMonkey/1.1a
```

Firefox 2.0.0.11 в Windows XP:

```
Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.8.1.11) Gecko/20071127
    Firefox/2.0.0.11
```

Camino 1.5.1 в Mac OS X:

```
Mozilla/5.0 (Macintosh; U; Intel Mac OS X; en; rv:1.8.1.6) Gecko/20070809
    Camino/1.5.1
```

Все эти строки определяют те или иные браузеры на основе Gecko. Часто идентифицировать конкретный браузер не требуется — достаточно узнать, что он основан на Gecko. Версия Mozilla 5.0 не изменялась, начиная с выпуска первого браузера на основе Gecko, и вероятно, не изменится.

В Firefox 4 разработчики из Mozilla упростили строку пользовательского агента. Перечислим основные изменения.

- Маркер языка (`en-US` в приведенных примерах) удален.
- Маркер шифрования отсутствует, если используется сильное шифрование (включено по умолчанию). Это означает, что в строках пользовательского агента Mozilla теперь могут содержаться значения `"I"` и `"N"`, но не `"U"`.
- Маркер платформы удален из строк пользовательского агента Windows, так как он избыточен при наличии маркера «ОС или ЦП», который всегда содержит строку `"Windows"`.
- Маркер версии Gecko теперь имеет фиксированное значение `Gecko/20100101`.

Пример окончательной строки пользовательского агента Firefox 4:

```
Mozilla/5.0 (Windows NT 6.1; rv:2.0.1) Gecko/20100101 Firefox 4.0.1
```

WebKit

В 2003 г. компания Apple анонсировала собственный веб-браузер под названием Safari. Визуализатор Safari, названный WebKit, начался как ответвление от проекта KHTML — визуализатора браузера Konqueror для Linux. Через пару лет WebKit был преобразован в отдельный проект с открытым исходным кодом.

Разработчики нового браузера и визуализатора столкнулись с той же проблемой, что и создатели Internet Explorer 3: как гарантировать, что пользователям нового браузера будут доступны популярные сайты? Было решено добавить в строку пользовательского агента все сведения, необходимые для совместимости с другим популярным браузером. Итоговый формат оказался таким:

```
Mozilla/5.0 (платформа; шифрование; ОС или ЦП; язык)
    AppleWebKit/версия AppleWebKit (KHTML, like Gecko) Safari/версия Safari
```

Пример строки:

```
Mozilla/5.0 (Macintosh; U; PPC Mac OS X; en) AppleWebKit/124
    (KHTML, like Gecko) Safari/125.1
```

Как видите, получилась еще одна длинная строка пользовательского агента, содержащая не только версию Apple WebKit, но и версию Safari. Сомнения по поводу того, выдавать ли браузер за Mozilla, быстро отпали из соображений совместимости. Все браузеры на основе WebKit (и Gecko) идентифицируют себя как Mozilla 5.0. В качестве версии Safari обычно указывается номер сборки браузера, а представление номера выпуска необязательно. Например, хотя для Safari 1.25

в строке пользовательского агента указан номер 125.1, однозначное соответствие наблюдается не всегда.

Наиболее интересным и спорным в этой строке пользовательского агента является элемент `"(KHTML, like Gecko)"`, добавленный в Safari до выпуска версии 1.0. Компания Apple получила много недовольных отзывов от разработчиков, которые восприняли это как наглую попытку выдать Safari за Gecko (как если бы добавления версии `Mozilla/5.0` было недостаточно). Ответ Apple был похож на реакцию Microsoft, когда критике подверглась строка пользовательского агента в Internet Explorer: браузер Safari совместим с Mozilla, и веб-сайты не должны блокировать пользователей Safari из-за того, что их браузер якобы не поддерживается.

В третьей версии Safari строка пользовательского агента была немного расширена. Теперь для идентификации фактической версии Safari используется следующий маркер:

```
Mozilla/5.0 (Macintosh; U; PPC Mac OS X; en) AppleWebKit/522.15.5
    (KHTML, like Gecko) Version/3.0.3 Safari/522.15.5
```

Это изменение внесено только в Safari, но не в WebKit, так что в других браузерах на основе WebKit оно может отсутствовать. Вообще говоря, как и в случае Gecko, обычно определяют, что браузер основан на WebKit, а не пытаются идентифицировать именно Safari.

Konqueror

Браузер Konqueror, поставляемый со средой KDE в Linux, основан на визуализаторе KHTML с открытым исходным кодом. Хотя Konqueror доступен только для Linux, он пользуется определенной популярностью. Ради совместимости для Konqueror была выбрана строка пользовательского агента, похожая на Internet Explorer:

```
Mozilla/5.0 (compatible; Konqueror/версия; ОС или ЦП)
```

В Konqueror 3.2 был добавлен идентификатор `KHTML` в соответствии с изменением строки пользовательского агента в WebKit:

```
Mozilla/5.0 (compatible; Konqueror/версия; ОС или ЦП) KHTML/версия KHTML
    (like Gecko)
```

Вот пример строки:

```
Mozilla/5.0 (compatible; Konqueror/3.5; SunOS) KHTML/3.5.0 (like Gecko)
```

Номера версий Konqueror и KHTML обычно совпадают или различаются после второй точки. Например, в Konqueror 3.5 используется модуль KHTML 3.5.1.

Chrome

Веб-браузер Chrome от Google использует WebKit в качестве визуализатора, но с другим интерпретатором JavaScript. Строка пользовательского агента в Chrome

содержит все сведения из WebKit и дополнительный раздел версии Chrome. Ее формат таков:

```
Mozilla/5.0 (платформа; шифрование; ОС или ЦП; язык)
    AppleWebKit/версия AppleWebKit (KHTML, like Gecko)
    Chrome/версия Chrome Safari/версия Safari
```

Вот полная строка пользовательского агента для Chrome 7:

```
Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US) AppleWebKit/534.7
    (KHTML, like Gecko) Chrome/7.0.517.44 Safari/534.7
```

Вероятно, далее версии WebKit и Safari всегда будут синхронизированы, но это не гарантируется.

Opera

Строка пользовательского агента в Opera заслуживает наибольшего внимания. По умолчанию она логичнее, чем в любых других современных браузерах, поскольку правильно определяет браузер и его версию. До версии 8 строка пользовательского агента в Opera имела следующий формат:

```
Opera/версия (ОС или ЦП; шифрование) [язык]
```

В Opera 7.54 на компьютере с Windows XP строка пользовательского агента такова:

```
Opera/7.54 (Windows NT 5.1; U) [en]
```

В Opera 8 маркер языка был перемещен внутрь скобок для согласования с другими браузерами:

```
Opera/версия (ОС или ЦП; шифрование; язык)
```

Вот, например, строка пользовательского агента в Opera 8 для Windows XP:

```
Opera/8.0 (Windows NT 5.1; U; en)
```

По умолчанию Opera возвращает строку пользовательского агента в этом простом формате. В настоящее время это единственный из основных браузеров, который полностью идентифицирует себя с помощью названия и версии продукта. Однако, как и в других браузерах, с применением строки пользовательского агента в Opera связаны проблемы. Несмотря на то что технически она правильна, многие средства идентификации браузеров в интернете ожидают строку пользовательского агента с названием Mozilla, а некоторые вообще настроены для распознавания Internet Explorer или Gecko. Чтобы не путать такие средства, Opera выдает себя за другой браузер, изменяя собственную строку пользовательского агента.

Начиная с выпуска Opera 9, это можно сделать двумя способами. Первый — изменить строку пользовательского агента строкой из Firefox или Internet Explorer с добавлением маркера `Opera` и номера версии Opera в конце, например:

```
Mozilla/5.0 (Windows NT 5.1; U; en; rv:1.8.1) Gecko/20061208 Firefox/2.0.0
    Opera 9.50

Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; en) Opera 9.50
```

Первая строка идентифицирует Opera 9.5 как Firefox 2 с сохранением сведений о версии Opera. Вторая строка выдает Opera 9.5 за Internet Explorer 6 и также содержит сведения о версии Opera. Эти строки пользовательского агента проходят большинство проверок для Firefox и Internet Explorer, но при желании по ним все же можно распознать Opera.

Второй способ — замаскировать Opera как Firefox или Internet Explorer. При этом строка пользовательского агента в Opera ничем не отличается от строк других браузеров: она не содержит ни маркер `Opera`, ни номер версии. Отличить Opera от других браузеров в этом случае невозможно. Еще больше дело запутывает то, что Opera часто задает строки пользовательского агента, специфичные для конкретных сайтов, не уведомляя об этом пользователя. Например, при переходе на сайт My Yahoo! (http://my.yahoo.com) Opera автоматически выдает себя за Firefox. Это делает идентификацию Opera по строке пользовательского агента очень непростой.

ПРИМЕЧАНИЕ До версии 7 браузеры Opera могли интерпретировать строки операционной системы Windows. Например, Windows NT 5.1 на самом деле означает Windows XP, так что в Opera 6 строка пользовательского агента включала маркер Windows XP, а не Windows NT 5.1. Чтобы улучшить совместимость с другими браузерами, в Opera 7 и более поздних версий используется официальная версия операционной системы.

В Opera 10 формат строки пользовательского агента стал следующим:

```
Opera/9.80 (ОС или ЦП; шифрование; язык) Presto/версия Presto Version/версия
```

Версия Opera/9.80 стала фиксированной. На самом деле браузера Opera 9.8 не было, но разработчики Opera опасались, что средства распознавания браузеров могут ошибочно интерпретировать маркер Opera/10.0 как Opera 1, а не Opera 10. В связи с этим в Opera 10 были представлены дополнительные маркеры `Presto` (Presto — это визуализатор Opera) и `Version` (указывает фактическую версию браузера). Например, Opera 10.63 в системе Windows 7 имеет такую строку пользовательского агента:

```
Opera/9.80 (Windows NT 6.1; U; en) Presto/2.6.30 Version/10.63
```

Современные версии Opera перешли на использование идентификатора OPR внутри пользовательского агента, добавляемого к более стандартизированной строке пользовательского агента. Лексически пользовательский агент теперь напоминает браузер WebKit, за исключением конечного предложения OPR. Это строка пользовательского агента для Opera 52 в Windows 10:

```
Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/65.0.3325.181 Safari/537.36 OPR/52.0.2871.64
```

iOS и Android

Веб-браузеры по умолчанию для мобильных операционных систем iOS и Android основаны на WebKit и имеют строки пользовательского агента почти такого же формата, что и их аналоги для настольных компьютеров. Вот формат iOS:

```
Mozilla/5.0 (платформа; шифрование; ОС или ЦП like Mac OS X; язык)
    AppleWebKit/версия AppleWebKit (KHTML, like Gecko)
    Version/версия браузера Mobile/мобильная версия Safari/версия Safari
```

Обратите внимание на строку `like Mac OS X`, которая помогает распознавать операционные системы для Mac, и новый маркер `Mobile`. Номер версии после этого маркера обычно бесполезен и применяется, если нужно различить версии WebKit для мобильных устройств и настольных компьютеров. Платформой может быть `iPhone`, `iPod` или `iPad` в зависимости от устройства, например:

```
Mozilla/5.0 (iPhone; U; CPU iPhone OS 3_0 like Mac OS X; en-us)
    AppleWebKit/528.18 (KHTML, like Gecko)
    Version/4.0 Mobile/7A341 Safari/528.16
```

До iOS 3 номера версии операционной системы не было в строке пользовательского агента.

В браузере Android по умолчанию строка пользовательского агента в целом соответствует формату iOS, но не содержит номера версии после маркера `Mobile`, например:

```
Mozilla/5.0 (Linux; U; Android 2.2; en-us; Nexus One Build/FRF91)
    AppleWebKit/533.1 (KHTML, like Gecko) Version/4.0 Mobile Safari/533.1
```

Эта строка получена с телефона Google Nexus One, но такой же формат используется и на других устройствах с системой Android.

Использование пользовательского агента для анализа браузера

Большинство разработчиков, желающих определить, в каком браузере работает их код, считают, что анализ строки, возвращаемой из `window.navigator.userAgent`, является канонической стратегией для этого. Все браузеры предлагают такую строку, и если вы доверяете возвращенному значению, проверка пользовательского агента по известному каталогу может дать исключительно точную картину браузера и операционной системы.

Преимущество использования пользовательского агента по сравнению с распознаванием возможностей для идентификации браузера заключается в том, что, хотя обнаружение возможностей обеспечивает более надежную защиту от сценариев, которые могут работать для сокрытия браузера, пользовательские агенты современных браузеров организованы предсказуемым образом для прошлых, текущих и будущих версий браузера поставщика.

Подмена пользовательского агента

Идентификация пользовательского агента является несовершенным решением идентификации браузера, особенно из-за того что можно легко подделать строку агента пользователя. Браузеры, которые правильно реализуют объект `window.navigator` (фактически все), предложат его как свойство только для чтения. Следовательно, использование метода записи свойства ничего не поменяет:

```
console.log(window.navigator.userAgent);
// Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/65.0.3325.181 Safari/537.36

window.navigator.userAgent = 'foobar';

console.log(window.navigator.userAgent);
// Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/65.0.3325.181 Safari/537.36
```

Однако для этого есть ряд простых обходных путей. Например, браузеры, которые предлагают псевдоприватный `__defineGetter__`, делают подделку пользовательского агента очень легкой:

```
console.log(window.navigator.userAgent);
// Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/65.0.3325.181 Safari/537.36

window.navigator.__defineGetter__('userAgent', () => 'foobar');

console.log(window.navigator.userAgent);
// foobar
```

Смягчение таких обходных путей — тяжелая битва. Обнаружение того, был ли пользовательский агент подделан таким образом, возможно, но надежное выполнение этого, по сути, является игрой в «убей крота».

Вместо того чтобы прибегать к обнаружению подделки, предпочтите философию идентификации браузера. Если вы доверяете возвращенному пользовательскому агенту, используйте его для определения личности. Если вы подозреваете, что сценарий или браузер может манипулировать этим значением, лучше использовать обнаружение возможностей.

Использование пользовательских агентов для анализа браузера

Синтаксически анализируя пользовательский агент браузера, можно вывести многие из следующих деталей об окружающей среде с превосходной точностью:

- браузер;
- версия браузера;
- движок виртуализации браузера;
- класс устройства (настольное/мобильное);

- производитель устройства;
- модель устройства;
- операционная система;
- версия операционной системы.

Появляются новые браузеры, операционные системы и устройства, многие с похожими, но уникальными строками пользовательских агентов; в результате реализации синтаксического анализатора пользовательского агента требуют постоянных обновлений, чтобы избежать устаревания. Любая реализация парсера пользовательского агента, созданная вручную, быстро устареет без постоянного обновления и пересмотра. Хотя в предыдущем издании этой книги была создана собственная версия анализатора пользовательского агента, создание анализатора с нуля более не рекомендуется. Вместо этого ниже приведены несколько поддерживаемых анализаторов пользовательских агентов:

- **Bowser** — https://github.com/lancedikson/bowser
- **ua-parser-js** — https://github.com/faisalman/ua-parser-js
- **Platform.js** — https://github.com/bestiejs/platform.js
- **current-device** — https://github.com/matthewhudson/current-device
- **Google Closure** — https://github.com/google/closure-library/tree/master/closure/goog/useragent
- **Mootools** — https://github.com/mootools/mootools-core/blob/master/Source/Browser/Browser.js

ПРИМЕЧАНИЕ Документация Mozilla предлагает страницу анализаторов пользовательских агентов, которые могут идентифицировать браузеры Mozilla (и, по-видимому, все основные браузеры). Они сгруппированы по языку и могут быть найдены по адресу https://wiki.mozilla.org/Compatibility/UADetectionLibraries#JavaScript_2. Страница активно не поддерживается, но в списке по-прежнему представлены все основные библиотеки синтаксического анализа пользовательских агентов. (Обратите внимание, что раздел JavaScript включает как клиентские библиотеки, так и библиотеки NodeJS.) Таблицу для их визуализации можно найти по адресу http://miketaylr.github.io/ arewedetectableyet /.

РАСПОЗНАВАНИЕ ПРОГРАММНОГО И АППАРАТНОГО ОБЕСПЕЧЕНИЯ

Современные браузеры предлагают набор информации о среде выполнения страницы, которая включает в себя информацию о браузере, операционную систему, аппаратное обеспечение и системную периферию. К этой информации можно получить доступ через набор API, которые находятся в объекте `window.navigator`. Однако поддержка этих API в разных браузерах далека от стандартизированной, поэтому в лучшем случае ее следует рассматривать как ненадежную.

ПРИМЕЧАНИЕ Настоятельно рекомендуется изучить браузерную поддержку любой из этих функций, прежде чем использовать их, поскольку большинство из них ненормативные и не поддерживаются во многих браузерах. Кроме того, описанные здесь функции могут быть ненадежными в некоторых случаях.

Идентификация браузера и операционной системы

Хотя обнаружение функций и разбор пользовательского агента являются двумя способами идентификации браузера, используемыми в настоящее время, объекты `navigator` и `screen` также предоставляют информацию о программной среде, в которой страница выполняется в данный момент.

Свойство navigator.oscpu

Свойство `oscpu` может предоставлять строку, которая обычно является компонентом операционной системы/архитектуры пользовательского агента. Согласно стандарту HTML Living:

> *Получатель атрибута* `oscpu` *должен возвращать либо пустую строку, либо строку, представляющую платформу, на которой выполняется браузер, например Windows NT 10.0; Win64; x64, Linux x86_64.*

Например, в Firefox в Windows 10 свойство `oscpu` сообщается следующим образом:

```
console.log(navigator.userAgent);
"Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:58.0) Gecko/20100101 Firefox/58.0"
console.log(navigator.oscpu);
"Windows NT 10.0; Win64; x64"
```

Свойство navigator.vendor

Свойство `vendor` может содержать строку, которая обычно является поставщиком браузера. Возвращаемая строка является функцией режима совместимости навигатора браузера. Согласно стандарту HTML Living:

> `navigator.vendor` *возвращает либо пустую строку, либо строку Apple Computer, Inc., либо строку Google Inc.*

Например, в Chrome свойство `vendor` сообщается следующим образом:

```
console.log(navigator.vendor);    // "Google Inc."
```

Свойство navigator.platform

Свойство `platform` может содержать строку, которая обычно указывает на операционную систему, внутри которой выполняется браузер. Согласно стандарту HTML Living:

> `navigator.platform` *должен возвращать либо пустую строку, либо строку, представляющую платформу, на которой выполняется браузер, например MacIntel, Win32, FreeBSD i386, WebTV OS.*

Например, в Chrome свойство `platform` сообщается следующим образом:

```
console.log(navigator.platform);    // "Win32"
```

Свойства creen.colorDepth и screen.pixelDepth

Свойства `colorDepth` и `pixelDepth` возвращают одно и то же значение: количество цветовых битов, которые могут быть представлены на дисплее. Согласно спецификации CSSOM:

> *Атрибуты* `colorDepth` *и* `pixelDepth` *должны возвращать количество битов, выделенных цветам для пикселя в устройстве вывода, исключая альфа-канал.*

Например, в Chrome эти свойства сообщаются следующим образом:

```
console.log(screen.colorDepth);    // 24
console.log(screen.pixelDepth);    // 24
```

Свойство screen.orientation

Свойство `orientation` возвращает объект `ScreenOrientation`, который содержит информацию об экране браузера в соответствии с ScreenOrientation API. Наиболее интересными свойствами этого объекта являются `angle`, который возвращает угол экрана относительно значения по умолчанию, и `type`, который возвращает строку из перечисления возможных типов ориентации:

```
portrait-primary
portrait-secondary
landscape-primary
landscape-secondary
```

Например, в Chrome на мобильном устройстве `screen.orientation` сообщается следующим образом:

```
// При вертикальном просмотре
console.log(screen.orientation.type);     // portrait-primary
console.log(screen.orientation.angle);    // 0

// При повороте телефона влево
console.log(screen.orientation.type);     // landscape-primary
console.log(screen.orientation.angle);    // 90

// При повороте телефона вправо
console.log(screen.orientation.type);     // landscape-secondary
console.log(screen.orientation.angle);    // 270
```

В соответствии со спецификацией инициализация этих значений зависит от браузера и устройства, поэтому нельзя предполагать, что `portrait-primary` и `0` всегда будут

исходными значениями. Их лучше всего использовать для оценки относительных изменений ориентации браузера при повороте устройства.

Метаданные браузера

Объект `navigator` предоставляет несколько API, которые могут сообщить информацию о состоянии браузера и операционной системы.

Geolocation API

Свойство `navigator.geolocation` обеспечивает доступ к API GeoLocation, который позволяет скриптам браузера узнавать о местоположении текущего устройства. Этот API доступен только в безопасном контексте выполнения (скрипты, которые обслуживаются через HTTPS).

API может отправлять запросы в хост-систему, чтобы вернуть местоположение устройства в меру своих возможностей. В зависимости от оборудования и конфигурации хост-системы точность результатов может отличаться. GPS-координаты мобильного телефона будут иметь чрезвычайно высокую точность, тогда как IP-адрес будет иметь гораздо более низкую точность. Согласно спецификации Geolocation API:

> *Общие источники информации о местоположении включают в себя глобальную систему определения местоположения (GPS) и местоположение, выведенное из сетевых сигналов, таких как IP-адрес, RFID, WiFi и Bluetooth MAC-адреса и идентификаторов cot GSM/CDMA, а также пользовательского ввода.*

ПРИМЕЧАНИЕ Браузеры также могут использовать такие инструменты, как Google Location Services (используемые Chrome и Firefox), чтобы определить местоположение. Вы заметите, что, хотя ваше устройство не имеет GPS-радио, координаты, которые возвращает браузер, часто будут чрезвычайно точными. Браузер делает это, собирая идентификаторы всех видимых беспроводных сетей в зоне действия, как WiFi, так и сотовых вышек, если это возможно. Затем они проверяются по базе данных сетей, местоположение которых уже известно. Таким образом, эти сервисы могут точно определять местоположение устройства с исключительно высокой точностью.

Для одноразового захвата текущего местоположения браузера используется метод `getCurrentPosition()`. Он возвращает объект `Coordinates`, содержимое которого может быть или не быть полным в зависимости от возможностей хост-системы:

```
// обратный вызов getCurrentPosition() происходит с объектом Position в качестве
// единственного аргумента
let p;
navigator.geolocation.getCurrentPosition((position) => p = position);
```

Объект `Position` содержит метку времени, представляющую, когда было получено содержимое объекта, и объект `Coordinates`:

```
console.log(p.timestamp);    // 1525364883361
console.log(p.coords);       // Coordinates {...}
```

Объект `Coordinates` содержит широту/долготу в стандартном формате градусов, а также точность этой пары в метрах. Точность измерения обеспечивается тем же механизмом, который рассчитывал местоположение устройства.

```
console.log(p.coords.latitude, p.coords.longitude);    // 37.4854409, -122.2325506
console.log(p.coords.accuracy);                        // 58
```

Объект `Coordinates` содержит свойство `altitude`, которое определяется измеренным расстоянием в метрах над эллипсоидальной моделью Земли WGS84 (World Geodetic System, 1984). Объект также содержит свойство `altitudeAccuracy`, точность измерения в метрах. Для заполнения внутри объекта `Coordinates` эти значения будут напрямую предоставлены устройством (это означает, что ему потребуется доступ к соответствующему измерительному оборудованию, вероятно, к GPS-радиоприемнику или альтиметру). Многие устройства не способны измерять высоту, поэтому одно или оба из этих значений часто являются нулевыми.

```
console.log(p.coords.altitude);            // -8.800000190734863
console.log(p.coords.altitudeAccuracy);    // 200
```

> **ПРИМЕЧАНИЕ** Более подробную информацию о модели WGS84 можно найти по адресу http://earth-info.nga.mil/GandG/publications/tr8350.2/wgs84fin.pdf.

Объект `Coordinates` содержит свойство `speed`, которое определяется измеренной скоростью устройства в метрах в секунду. Объект также содержит свойство `heading`, которое определяется направлением движения относительно истинного севера в градусах ($0 \leq$ `heading` <360). Для заполнения внутри объекта `Coordinates` эти значения будут напрямую предоставлены устройством (это означает, что ему потребуется доступ к соответствующему измерительному оборудованию, вероятно, к акселерометру и компасу). Многие устройства не способны измерять скорость и курс, поэтому эти значения часто равны нулю.

> **ПРИМЕЧАНИЕ** Устройства не будут пытаться измерять скорость и курс, используя производный вектор двух измерений местоположения. Однако можно вручную рассчитать эти значения, выполнив два последовательных измерения и получив вектор направления движения. Конечно, в зависимости от точности двух измерений местоположения полученные скорость и курс могут быть очень неточными!

Захват местоположения браузера является наиболее эффективной операцией, и существует ряд причин, по которым захват может завершиться неудачно. Метод `getCurrentPosition()` принимает обратный вызов ошибки в качестве второго аргумента, и этому методу передается объект `PositionError`. В случае сбоя этот объект будет содержать свойства `code` и `message` с кратким описанием ошибки. Свойство `code` будет целым числом, указывающим на одну из трех возможных ошибок:

- `PERMISSION_DENIED` — браузер заблокировал доступ к местоположению устройства. При первой попытке страницы использовать Geolocation API браузер предложит пользователю разрешить доступ (это происходит для каждого домена). Если встречается этот код ошибки, то либо пользователь отказал в доступе к расположению устройства, либо доступ к Geolocation API осуществляется из незащищенного контекста. Свойство `message` может предложить дополнительную информацию.
- `POSITION_UNAVAILABLE` — системе не удалось вернуть информацию о местоположении. Это может представлять любое количество возможных точек отказа, но будет относительно необычной ошибкой, поскольку сетевое устройство обычно может преобразовывать IP-адрес в координату низкой точности.
- `TIMEOUT` — системе не удалось вернуть информацию о местоположении в течение периода ожидания. Настройка периода ожидания обсуждается в следующем материале.

```
// Браузер предложит пользователю предоставить доступ к Geolocation API.
// Пример показывает, что происходит после запрета доступа пользователем.
navigator.geolocation.getCurrentPosition(
    () => {},
    (e) => {
        console.log(e.code); // 1
        console.log(e.message); // User denied Geolocation
    }
);

// Пример показывает, что происходит при выполнении из незащищенного контекста.
navigator.geolocation.getCurrentPosition(
    () => {},
    (e) => {
        console.log(e.code); // 1
    console.log(e.message); // Only secure origins are allowed
    }
);
```

Запросы местоположения Geolocation API можно настроить с помощью объекта `PositionOptions`, который предоставляется в качестве третьего аргумента. Этот объект поддерживает три свойства:

- `enableHighAccuracy` — логическое значение, указывающее системе, что при значении `true` возвращаемое значение должно быть максимально точным; значение флага по умолчанию — `false`. По умолчанию устройства обычно предпочитают возвращать координаты самым быстрым и наиболее энергоэффективным способом. Это часто будет означать, что координаты менее точны. Например, на мобильных устройствах поиск местоположения по умолчанию обычно принимает форму получения местоположения устройства только из WiFi и сотовых сетей. Когда `enableHighAccuracy` имеет значение `true`, оно также будет запрашивать, чтобы устройство использовало GPS-радио для определения местоположения устройства, и возвращенные координаты будут гибридным результатом этих значений. Использование GPS-радио является более медленной и более

ресурсоемкой операцией, поэтому нужно оценивать компромиссы при использовании флага `enableHighAccuracy`.

- `Timeout` — число, указывающее максимальное количество миллисекунд, которое должен ожидать вызов API, прежде чем запустить обратный вызов ошибки со статусом `TIMEOUT`; значение по умолчанию 0xFFFFFFFF (2^32-1). Значение 0 полностью пропустит системный вызов и немедленно вызовет обратный вызов ошибки `TIMEOUT`.
- `maximumAge` — число, указывающее максимальный возраст возвращаемых координат в миллисекундах; значение по умолчанию равно 0. Поскольку определение местоположения устройства является дорогостоящей операцией, системы часто будут кешировать координаты и возвращать кешированное значение (в соответствии с политикой истечения срока действия кеша местоположения). Система отслеживает возраст кешированного значения, и если Geolocation API запрашивает координаты, которые должны быть новее, чем кешированное значение, система выполнит новый поиск местоположения и вернет это значение. Значение 0 заставит систему игнорировать любое кешированное значение и сразу запускать новый поиск местоположения. Значение `Infinity` не позволит системе выполнить новый поиск и разрешит использовать только кешированные значения. JavaScript может определить, возвращено ли кешированное значение, проверив наличие дублированных свойств отметки времени внутри объекта `Position`.

Состояние соединения и API NetworkInformation

Браузер отслеживает состояние сетевого подключения и предоставляет эту информацию двумя способами: событиями подключения и свойством `navigator.onLine`. При подключении устройства к сети браузер узнает об этом и запустит событие `online` на объекте `window`. В свою очередь, когда устройство теряет сетевое соединение, браузер запускает событие `offline` на объекте `window`. В любое время текущее состояние браузера можно определить, проверив свойство `navigator.onLine`, которое содержит логическое значение, указывающее, подключен ли браузер.

```
const connectionStateChange = () => console.log(navigator.onLine);

window.addEventListener('online', connectionStateChange);
window.addEventListener('offline', connectionStateChange);

// При подключении устройства:
// true

// При отключении устройства:
// false
```

Конечно, то, что определяет сетевое соединение, зависит от браузера и реализации системы. Некоторые браузеры будут рассматривать подключение к любой локальной сети как «в сети», даже если эта сеть может не иметь надлежащего доступа в интернет.

В объекте `navigator` также отображается NetworkInformation API, который можно найти в свойстве `navigator.connection`. Этот API предлагает несколько свойств, доступных только для чтения, а также источник событий для прикрепления обратных вызовов при изменении свойств соединения.

Доступны следующие свойства:

- `downlink` — целое число, обозначающее текущую пропускную способность устройства в мегабитах в секунду, округленную до ближайших 25 Кбит/с. Это вычисление может быть получено из прошлых измерений пропускной способности сети или возможностей технологии соединения.
- `downlinkMax` — целое число, обозначающее текущую максимальную пропускную способность нисходящей линии в мегабитах в секунду, определенную первым сетевым переходом. Поскольку первый сетевой скачок необязательно указывает на сквозную производительность сети, это значение следует использовать только в качестве приблизительной верхней границы.
- `effectiveType` — строковое перечисление, указывающее общую скорость и качество соединения. Значения обозначены как соединения сотовой сети передачи данных, но они также используются для классификации проводных соединений. Свойство будет иметь одно из четырех значений:
  - `"slow-2g"`
    - время обратной передачи > 2000 мс;
    - пропускная способность нисходящей линии связи < 50 Кбит/с;
  - `"2g"`
    - 2000 мс > время приема-передачи ≥ 1400 мс;
    - 70 Кбит/с > пропускная способность нисходящей линии связи ≥ 50 Кбит/с;
  - `"3g"`
    - 1400 мс > время приема-передачи ≥ 270 мс;
    - 700 Кбит/с > пропускная способность нисходящей линии связи ≥ 70 Кбит/с;
  - `"4g"`
    - 270 мс > время приема-передачи ≥ 0 мс;
    - пропускная способность нисходящей линии связи ≥ 700 Кбит/с.
- `rtt` — целое число, указывающее текущее действующее время приема-передачи сетевого запроса в миллисекундах с округлением до ближайших 25 мс. Это вычисление может быть получено из прошлых измерений пропускной способности сети или возможностей технологии соединения.
- `type` — строковое перечисление, обозначающее технологию сетевого подключения. Свойство будет иметь одно из следующих значений:
  - `bluetooth` — указывает соединение Bluetooth;
  - `cellular` — обозначает сотовую сеть;

- `ethernet` — указывает на проводное соединение Ethernet;
- `none` — указывает на отсутствие сетевого подключения. Эквивалентно `navigator.onLine === false`;
- `mixed` — указывает на несколько типов одновременного подключения;
- `other` — указывает тип соединения, который не указан в качестве допустимого значения в этом перечислении;
- `unknown` — указывает тип соединения, который не может быть определен;
- `wifi` — указывает на соединение WiFi;
- `wimax` — обозначает соединение WiMAX.

- `saveData` — логическое значение, указывающее, включил ли пользователь «режим сокращенных данных» на своем устройстве.
- `onchange` — свойство, которое будет генерировать событие изменения при изменении любого состояния соединения. Оно может быть использовано в виде либо `navigator.connection.addEventListener('change', changeHandler)`, либо `navigator.connection.onchange = changeHandler`.

API статуса батареи

Браузер может получить доступ к информации о батарее устройства и состоянии заряда. Метод `navigator.getBattery()` возвращает промис, который разрешается в объект `BatteryManager`.

```
navigator.getBattery().then((b) => console.log(b));
// BatteryManager { ... }
```

BatteryManager предлагает четыре свойства только для чтения, которые предоставляют информацию о батарее устройства:

- `charging` — логическое значение, указывающее, подключено ли устройство и заряжается ли в данный момент. Если в устройстве нет батареи, возвращается значение `true`;
- `chargingTime` — целое число, обозначающее приблизительное количество секунд до полной зарядки батареи. Возвращает 0, если батарея полностью заряжена или если в устройстве нет батареи;
- `dischargingTime` — целое число, указывающее приблизительное количество секунд, пока батарея полностью не разрядится. Возвращает бесконечность, если устройство не имеет батареи;
- `Level` — число с плавающей точкой, указывающее частичный заряд батареи. Возвращает 0.0 для обозначения полностью разряженной батареи, 1.0 для обозначения полностью заряженной батареи. Возвращает 1.0, если устройство не имеет батареи.

API также предлагает четыре свойства события, которые можно использовать для установки обратных вызовов при изменении какого-либо свойства батареи. Это

может быть достигнуто либо путем добавления прослушивателя событий на объект `BatteryManager` или присвоение обработчику соответствующего свойства:

- `onchargingchange`
- `onchargingtimechange`
- `ondischarcingtimechange`
- `onlevelchange`

```
navigator.getBattery().then((battery) => {
    // Назначает обратный вызов при изменении статуса зарядки:
    const chargingChangeHandler = () => console.log('chargingchange');
    battery.onchargingchange = chargingChangeHandler;
    // или
    battery.addEventListener('chargingchange', chargingChangeHandler);

    // Назначает обратный вызов при изменении времени зарядки:
    const chargingTimeChangeHandler = () => console.log('chargingtimechange');
    battery.onchargingtimechange = chargingTimeChangeHandler;
    // или
    battery.addEventListener('chargingtimechange', chargingTimeChangeHandler);

    // Назначает обратный вызов при изменении времени до разрядки:
    const dischargingChangeHandler = () => console.log('dischargingtimechange');
    battery.ondischargingtimechange = dischargingTimeChangeHandler;
    // или
    battery.addEventListener('dischargingtimechange', dischargingTimeChangeHandler);

    // Назначает обратный вызов при изменении уровня батареи:
    const levelChangeHandler = () => console.log('levelchange');
    battery.onlevelchange = levelChangeHandler;
    // или
    battery.addEventListener('levelchange', levelChangeHandler);
});
```

Аппаратное обеспечение

Способность браузера обнаруживать аппаратное обеспечение системы весьма ограниченна; однако ряд свойств объекта `navigator` может предоставить основную информацию.

Ядра процессора

Можно определить количество ядер логического процессора, которые поддерживает браузер, с помощью свойства `navigator.hardwareConcurrency`, которое содержит целое число, представляющее количество поддерживаемых ядер (или 1, если определение не может быть выполнено). Важно отметить, что это значение указывает максимальное количество одновременных рабочих потоков, которые браузер способен выполнять параллельно, и необязательно количество реальных ядер, которые поддерживает процессор.

Память устройства

Можно определить приблизительный объем системной памяти на устройстве через свойство `navigator.deviceMemory`, которое содержит число с плавающей точкой, представляющее количество гигабайт памяти на устройстве, округленное до ближайшего значения 2: 512 МБ вернет 0,5; 4 ГБ вернули бы 4.

Максимальные точки касания

Можно определить максимальное количество сенсорных контактов, поддерживаемых на сенсорном экране, с помощью свойства `navigator.maxTouchPoints`, которое определяется как целое число.

ИТОГИ

Распознавание клиента — один из самых спорных аспектов применения JavaScript. Из-за различий браузеров часто приходится выбирать одну из ветвей кода на основе возможностей браузера. Есть несколько подходов к распознаванию клиента, но чаще всего используются два.

- **Распознавание возможностей**. Этот подход основан на проверке конкретных возможностей браузера перед их использованием. Например, сценарий может проверять доступность функции перед ее вызовом. Это позволяет не беспокоиться о конкретных типах и версиях браузеров, а довольствоваться сведениями о наличии или отсутствии возможности. Распознавание возможностей не предполагает точную идентификацию браузера или его версии.
- **Распознавание пользовательского агента**. Этот подход идентифицирует браузер по его строке пользовательского агента, которая описывает браузер и часто содержит сведения о его типе, версии, а также платформе и операционной системе. Формат строк пользовательского агента неоднократно изменялся, при этом во многие браузеры включались строки, выдающие их за другие браузеры. Распознавание пользовательского агента — нетривиальная задача, особенно если учесть, что Opera может маскировать строку пользовательского агента. Несмотря на это, по строке пользовательского агента можно определить визуализатор и платформу, на которой он работает, включая мобильные устройства и игровые системы.

При определении клиента предпочтительнее распознавать возможности, а если это невозможно или бессмысленно — особенности. Распознавание пользовательского агента следует рассматривать как крайнюю меру из-за его зависимости от строки пользовательского агента.

Браузеры также предлагают все более широкий обзор программного и аппаратного обеспечения, окружающего его. Посредством объектов `screen` и `navigator` можно получить чрезвычайно точное представление об операционной системе, браузере, оборудовании, расположении устройства, состоянии батареи и множестве других тем.

14

Объектная модель документа

- DOM как иерархия узлов
- Работа с узлами разных типов
- Использование DOM с учетом особенностей браузеров
- Наблюдатели за изменениями

Загрузки с сайта wrox.com для этой главы

Обратите внимание, что все примеры кода для этой главы доступны на веб-сайте книги по ссылке www.wrox.com/go/projavascript4e на вкладке Downloads.

Объектная модель документа (Document Object Model, DOM) — это прикладной программный интерфейс (API) для HTML- и XML-документов. DOM представляет документ как иерархическое дерево узлов, позволяя добавлять, удалять и изменять отдельные части страницы. Основанная первоначально на ранних инновационных технологиях DHTML (Dynamic HTML) от Netscape и Microsoft, DOM обеспечивает по-настоящему кроссплатформенный и не зависящий от языка способ представления страниц и изменения их разметки.

Спецификация DOM 1, определяющая базовые запросы и операции со структурой документа, получила статус рекомендации W3C в октябре 1998 г. В этой главе мы рассмотрим применение DOM для работы с HTML-страницами и DOM JavaScript API.

ПРИМЕЧАНИЕ Все DOM-объекты представлены в Internet Explorer 8 и более ранних версий COM-объектами. Это означает, что они работают не так, как встроенные JavaScript-объекты.

ИЕРАРХИЯ УЗЛОВ

Любой HTML- или XML-документ можно представить с помощью DOM как иерархию узлов. Есть несколько типов узлов, каждый из которых соответствует разным данным и (или) элементам разметки в документе. Каждый тип узлов обладает определенными характеристиками, данными и методами и может быть связан с другими узлами. Эти связи формируют иерархию, которая позволяет изобразить разметку в виде дерева с конкретным узлом в качестве корня. Возьмем для примера следующую HTML-страницу:

```
<html>
    <head>
        <title>Sample Page</title>
    </head>
    <body>
        <p>Hello World!</p>
    </body>
</html>
```

Этому простому HTML-документу соответствует иерархия узлов, показанная на рис. 14.1.

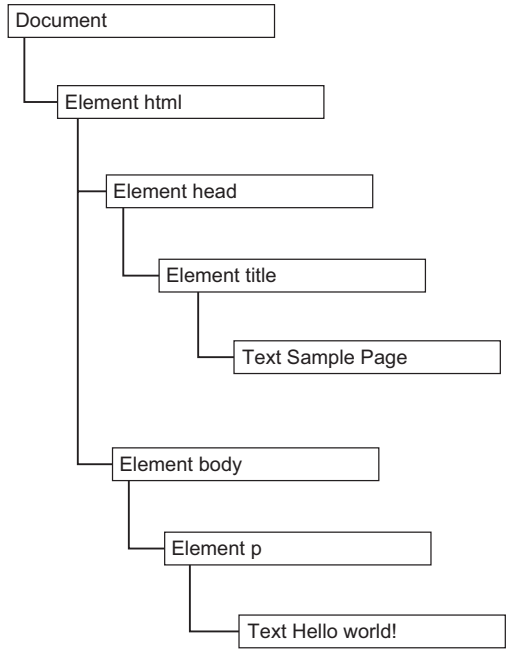

Рис. 14.1

Корнем каждого документа является его узел. В этом примере у документа есть единственный дочерний узел — элемент `<html>`, который называется *элементом документа* (document element). Элемент документа — это самый внешний элемент в документе, содержащий все остальные элементы. В документе может быть только один элемент документа. В HTML-страницах им всегда является элемент `<html>`, а в XML-разметке, где нет предопределенных элементов, им может быть любой элемент.

Всем элементам разметки соответствуют узлы в дереве: HTML-элементам — узлы элементов,

атрибутам — узлы атрибутов, типу документа — узел типа документа, а комментариям — узлы комментариев. Всего существует 12 типов узлов, которые наследуются от одного базового типа.

Тип Node

В DOM Level 1 определен интерфейс `Node`, который реализуют все типы DOM-узлов. В JavaScript он представлен типом `Node`, который доступен во всех браузерах, кроме Internet Explorer. Все типы узлов в JavaScript наследуются от `Node`, благодаря чему имеют ряд общих базовых свойств и методов.

У каждого узла есть свойство `nodeType`, указывающее тип узла. Типы узлов определены в типе `Node` с помощью 12 числовых констант:

- `Node.ELEMENT_NODE` (1)
- `Node.ATTRIBUTE_NODE` (2)
- `Node.TEXT_NODE` (3)
- `Node.CDATA_SECTION_NODE` (4)
- `Node.ENTITY_REFERENCE_NODE` (5)
- `Node.ENTITY_NODE` (6)
- `Node.PROCESSING_INSTRUCTION_NODE` (7)
- `Node.COMMENT_NODE` (8)
- `Node.DOCUMENT_NODE` (9)
- `Node.DOCUMENT_TYPE_NODE` (10)
- `Node.DOCUMENT_FRAGMENT_NODE` (11)
- `Node.NOTATION_NODE` (12)

Можно легко определить тип узла, сравнив его с одной из этих констант:

```
if (someNode.nodeType == Node.ELEMENT_NODE) {
    alert("Node is an element.");
}
```

В этом примере свойство `someNode.nodeType` сравнивается с константой `Node.ELEMENT_NODE`. Если они равны, значит, `someNode` является элементом.

Не все типы узлов поддерживаются в веб-браузерах. Чаще всего в сценариях используются узлы элементов и текста. Поддержку и применение узлов каждого типа мы рассмотрим позже.

Свойства nodeName и nodeValue

Свойства `nodeName` и `nodeValue` предоставляют сведения об узле. Значения этих свойств полностью зависят от типа узла, поэтому рекомендуется всегда проверять его перед их использованием:

```
if (someNode.nodeType == 1) {
    value = someNode.nodeName;    // имя тега элемента
}
```

Этот код проверяет, является ли узел элементом. Если да, значение `nodeName` присваивается переменной. У элементов свойство `nodeName` всегда содержит имя тега элемента, а `nodeValue` всегда имеет значение `null`.

Отношения узлов

Все узлы в документе связаны с другими узлами. Эти связи можно описать в терминах традиционных семейных отношений, как если бы дерево документа было генеалогическим древом. Так, в HTML элемент `<body>` является дочерним по отношению к элементу `<html>`, и наоборот, элемент `<html>` считается родительским для `<body>`. Элементы `<head>` и `<body>` являются одноуровневыми, потому что у них общий непосредственный родитель, элемент `<html>`.

У каждого узла есть свойство `childNodes`, содержащее объект `NodeList`, который похож на массив и используется для хранения упорядоченного списка узлов, доступных по позиции. `NodeList` не является экземпляром `Array`, хотя его значения доступны с помощью скобочной нотации и у него имеется свойство `length`. На самом деле объекты `NodeList` — это запросы к DOM-структуре, поэтому ее изменения отражаются в них автоматически. Иначе говоря, `NodeList` — это динамически обновляемый объект, а не «снимок» узлов на момент первого доступа к нему.

В следующем примере показано, как реализовать доступ к узлам в `NodeList` с помощью скобочной нотации и метода `item()`:

```
let firstChild = someNode.childNodes[0];
let secondChild = someNode.childNodes.item(1);
let count = someNode.childNodes.length;
```

Хотя поддерживается и скобочная нотация, и метод `item()`, большинство разработчиков предпочитают первый способ из-за его сходства с массивами. Свойство `length` указывает количество узлов в `NodeList` в текущий момент времени. Объект `NodeList` можно преобразовать в массив с помощью метода `Array.prototype.slice()`, что уже было показано для объекта `arguments`. Рассмотрим пример:

```
let arrayOfNodes = Array.prototype.slice.call(someNode.childNodes,0);
```

У каждого узла есть свойство `parentNode`, указывающее на его родительский узел в дереве документа. У всех узлов в списке `childNodes` один родитель, так что все их свойства `parentNode` указывают на один и тот же узел. Сами узлы в списке `childNodes` являются одноуровневыми. Переходить от одного узла в этом списке к другому можно с помощью свойств `previousSibling` и `nextSibling`. Свойство `previousSibling` у первого узла в списке и свойство `nextSibling` у последнего узла в списке равны `null`:

```
if (someNode.nextSibling === null) {
    alert("Last node in the parent's childNodes list.");
```

```
} else if (someNode.previousSibling === null) {
    alert("First node in the parent's childNodes list.");
}
```

Если дочерний узел единственный, оба эти свойства равны `null`.

Свойства `firstChild` и `lastChild` родительского узла указывают на первый и последний узлы в его списке `childNodes`. Значение `someNode.firstChild` всегда равно `someNode.childNodes[0]`, а значение `someNode.lastChild` всегда равно `someNode.childNodes[someNode.childNodes.length-1]`. Если дочерний узел только один, свойства `firstChild` и `lastChild` совпадают; если дочерних узлов нет, оба эти свойства равны `null`. С помощью всех описанных отношений можно легко перемещаться между узлами в структуре документа (рис. 14.2).

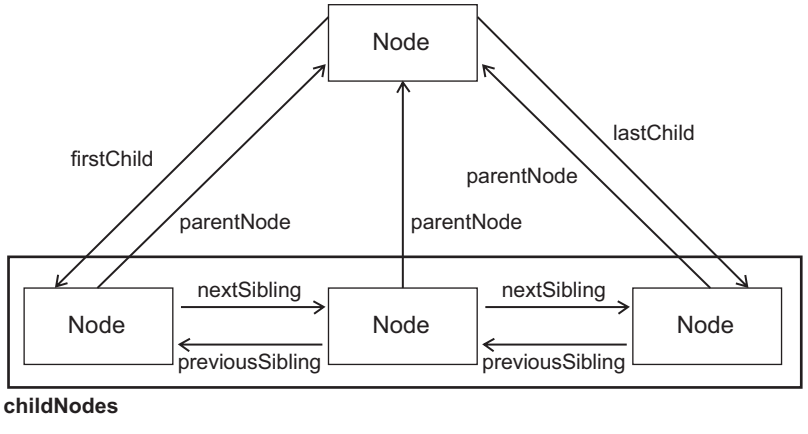

Рис. 14.2

Как видите, можно достигнуть любого узла в дереве документа, просто используя отношения между ними, так что свойство `childNodes` предоставляется скорее ради удобства. В этом смысле на него похож метод `hasChildNodes()`, который возвращает `true`, если у узла есть хотя бы один дочерний узел, и более эффективен, чем запрос свойства `length` списка `childNodes`.

Наконец, у всех узлов есть свойство `ownerDocument`, которое указывает на узел, представляющий весь документ. Узлы принадлежат документу, в котором они были созданы (обычно это документ, где они находятся), и не могут относиться одновременно к двум или более документам. Это свойство обеспечивает быстрый доступ к узлу документа без перебора всех узлов между текущим и корневым.

ПРИМЕЧАНИЕ Не у всех типов узлов могут быть дочерние узлы, хотя все они наследуются от типа `Node`. Различия типов узлов будут описаны позже.

Манипулирование узлами

Все указатели отношений доступны только для чтения, поэтому для манипулирования узлами применяются методы. Чаще всего используется метод `appendChild()`, который добавляет узел в конец списка `childNodes`. При его вызове обновляются все указатели отношений в добавленном узле, родительском узле и прежнем последнем дочернем узле в списке `childNodes`. Метод `appendChild()` возвращает новый добавленный узел, например:

```
let returnedNode = someNode.appendChild(newNode);
alert(returnedNode == newNode);            // true
alert(someNode.lastChild == newNode);      // true
```

Если узел, переданный в метод `appendChild()`, уже содержится в документе, он удаляется из предыдущего места и помещается в новое место. Хотя DOM-дерево связано лишь указателями, никакой DOM-узел не может располагаться в нескольких местах документа. Например, если передать в метод `appendChild()` первый дочерний узел, он станет последним:

```
// предполагается, что у someNode несколько дочерних узлов
let returnedNode = someNode.appendChild(someNode.firstChild);
alert(returnedNode == someNode.firstChild);        // false
alert(returnedNode == someNode.lastChild);         // true
```

Если нужно поместить узел в конкретное место списка `childNodes`, можно использовать метод `insertBefore()`, который принимает два аргумента: вставляемый узел и опорный узел. Вставляемый узел добавляется перед опорным и возвращается методом. Если опорный узел равен `null`, метод `insertBefore()` работает так же, как `appendChild()`:

```
// вставка узла в качестве последнего дочернего узла
returnedNode = someNode.insertBefore(newNode, null);
alert(newNode == someNode.lastChild);    // true

// вставка в качестве первого дочернего узла
returnedNode = someNode.insertBefore(newNode, someNode.firstChild);
alert(returnedNode == newNode);          // true
alert(newNode == someNode.firstChild);   // true

// вставка узла перед последним дочерним узлом
returnedNode = someNode.insertBefore(newNode, someNode.lastChild);
alert(newNode == someNode.childNodes[someNode.childNodes.length-2]);    // true
```

В то время как методы `appendChild()` и `insertBefore()` вставляют узлы, не удаляя другие узлы, метод `replaceChild()` используется для замены узла. В качестве аргументов он принимает вставляемый узел и заменяемый узел. Заменяемый узел полностью удаляется из дерева документа и возвращается из метода, а его место занимает новый узел, например:

```
// замена первого дочернего узла
let returnedNode = someNode.replaceChild(newNode, someNode.firstChild);

// замена последнего дочернего узла
returnedNode = someNode.replaceChild(newNode, someNode.lastChild);
```

Когда узел вставляется с помощью `replaceChild()`, все указатели отношений узла копируются из узла, который он заменяет. Хотя замененный узел технически попрежнему принадлежит тому же документу, у него больше нет конкретного места в документе.

Удалить узел можно с помощью метода `removeChild()`. Он принимает удаляемый узел как аргумент и возвращает удаленный узел:

```
// удаление первого дочернего узла
let formerFirstChild = someNode.removeChild(someNode.firstChild);

// удаление последнего дочернего узла
let formerLastChild = someNode.removeChild(someNode.lastChild);
```

Как и в случае метода `replaceChild()`, узел, удаленный методом `removeChild()`, все еще принадлежит документу, но не имеет конкретного положения в нем.

Все четыре метода, упомянутые в этом разделе, работают с непосредственными дочерними узлами конкретного узла, то есть для их использования необходимо знать родительский узел (он доступен через свойство `parentNode`). Не у всех типов узлов могут быть дочерние узлы, и попытки использовать эти методы с такими узлами приводят к ошибкам.

Другие методы

Другие два метода есть у узлов всех типов. Первый метод, `cloneNode()`, создает точную копию узла, для которого он вызван. Единственным аргументом `cloneNode()` является логическое значение, указывающее, нужно ли выполнять глубокое копирование. Если оно равно `true`, узел клонируется со всем его поддеревом, в противном случае клонируется только начальный узел. Клонированный узел, который возвращается методом, принадлежит документу, но не имеет родительского узла. Такой узел отсутствует в документе, пока его не добавили с помощью метода `appendChild()`, `insertBefore()` или `replaceChild()`. Рассмотрим, например, следующий HTML-код:

```
<ul>
    <li>item 1</li>
    <li>item 2</li>
    <li>item 3</li>
</ul>
```

Вот примеры двух режимов работы метода `cloneNode()` (предполагается, что переменная `myList` содержит ссылку на элемент `<ul>`):

```
let deepList = myList.cloneNode(true);
alert(deepList.childNodes.length);      // 3 (IE до версии 9)
                                        // или 7 (другие браузеры)

let shallowList = myList.cloneNode(false);
alert(shallowList.childNodes.length);   // 0
```

В этом примере переменной `deepList` присваивается глубокая копия `myList`. Это означает, что `deepList` содержит три элемента списка, каждый из которых включает

текст. Переменной `shallowList` назначается поверхностная копия `myList`, так что у нее нет дочерних узлов. Значения `deepList.childNodes.length` различаются в браузерах из-за того, что в Internet Explorer 8 и более ранних версий не создаются узлы для свободного пространства в коде.

ПРИМЕЧАНИЕ Метод `cloneNode()` не копирует JavaScript-свойства, добавленные в DOM-узлы, такие как обработчики событий. Копируются только атрибуты и, возможно, дочерние узлы, а все остальное теряется. В Internet Explorer есть дефект, из-за которого обработчики событий также клонируются, поэтому рекомендуется удалять их перед клонированием.

Второй метод называется `normalize()` и служит для работы с текстовыми узлами в поддереве документа. В результате синтаксического анализа или манипуляций с DOM можно получить одноуровневые текстовые узлы или текстовые узлы без текста. Метод `normalize()` ищет такие узлы среди потомков узла, для которого он вызван. Пустые текстовые узлы при этом удаляются, а соседние текстовые узлы одного уровня объединяются в один текстовый узел. К этому методу мы еще вернемся.

Тип Document

Узел документа представляется в JavaScript с помощью типа `Document`. В браузерах объект документа является экземпляром типа `HTMLDocument` (производного от `Document`) и соответствует всей HTML-странице. Объект `document` доступен глобально как свойство объекта `window`. Узел `Document` имеет следующие свойства:

- `nodeType` имеет значение 9;
- `nodeName` имеет значение `"#document"`;
- `nodeValue` имеет значение `null`;
- `parentNode` имеет значение `null`;
- `ownerDocument` имеет значение `null`;
- дочерними узлами могут быть объекты `DocumentType` (не более одного), `Element` (не более одного), `ProcessingInstruction` и `Comment`.

Тип `Document` может представлять HTML-страницы или другие XML-документы, но чаще всего он фигурирует в коде как объект `document` типа `HTMLDocument`. Этот объект можно использовать для получения сведений о странице, а также для изменения ее вида и структуры.

Дочерние узлы узла Document

Хотя в спецификации DOM сказано, что дочерними узлами у `Document` могут быть узлы `DocumentType`, `Element`, `ProcessingInstruction` и `Comment`, среди них есть также два встроенных ярлыка. Первым является свойство `documentElement`, которое всегда

указывает на элемент <html> в HTML-странице. Элемент document также всегда есть в списке childNodes, но свойство documentElement предоставляет более быстрый и прямой доступ к элементу <html>. Рассмотрим следующую простую страницу:

```
<html>
    <body>

    </body>
</html>
```

Когда эта страница обрабатывается браузером, у документа есть только один дочерний узел, элемент <html>. Он доступен с помощью свойства documentElement и списка childNodes:

```
let html = document.documentElement;       // получение ссылки на элемент <html>
alert(html === document.childNodes[0]);    // true
alert(html === document.firstChild);       // true
```

Этот пример показывает, что значения documentElement, firstChild и childNodes[0] одинаковы: все три указывают на элемент <html>.

Как экземпляр HTMLDocument объект document также имеет свойство body, указывающее на элемент <body>. Неудивительно, что оно используется в JavaScript очень часто:

```
let body = document.body;    // получение ссылки на элемент <body>
```

Свойства document.documentElement и document.body поддерживаются во всех основных браузерах.

Дочерним узлом у Document может быть также узел DocumentType. Тег <!DOCTYPE> считается самостоятельной сущностью, отдельной от других частей документа и доступной как свойство doctype (document.doctype в браузерах):

```
let doctype = document.doctype;    // получение ссылки на элемент <!DOCTYPE>
```

Комментарии вне элемента <html> технически являются дочерними узлами документа, но браузеры распознают и представляют их по-разному. Рассмотрим следующую HTML-страницу:

```
<!-- первый комментарий -->
<html>
    <body>

    </body>
</html>
<!-- второй комментарий -->
```

Казалось бы, у этой страницы три дочерних узла: комментарий, элемент <html> и еще один комментарий. Можно было бы ожидать, что и список document.childNodes будет содержать три элемента, но на практике браузеры обрабатывают комментарии вне элемента <html> совершенно по-разному в отношении игнорирования одного или обоих узлов комментариев.

Как правило, методы `appendChild()`, `removeChild()` и `replaceChild()` не используются с объектом `document`, поскольку тип документа (если он есть) доступен только для чтения, а дочерний элемент у документа может быть только один и уже существует.

Сведения о документе

В качестве экземпляра типа `HTMLDocument` объект `document` имеет несколько свойств, которых нет у стандартных объектов `Document`. Эти свойства предоставляют сведения о загруженной веб-странице. Первое из них, `title`, содержит текст элемента `<title>`, отображаемый в заголовке или на вкладке окна браузера. С помощью этого свойства можно получить заголовок текущей страницы или изменить его, при этом элемент `<title>` не изменяется, например:

```
// получение заголовка документа
let originalTitle = document.title;

// задание заголовка документа
document.title = "New page title"; // новый заголовок страницы
```

Следующие три свойства связаны с запросами веб-страниц. Свойство `URL` содержит полный URL-адрес страницы (который отображается в адресной строке), свойство `domain` — только доменное имя страницы, а `referrer` — URL-адрес страницы, с которой был выполнен переход на текущую страницу. Свойство `referrer` может быть пустой строкой, если текущая страница была открыта без ссылки. Все эти сведения есть в HTTP-заголовке запроса, а указанные свойства просто предоставляют доступ к ним в JavaScript:

```
// получение полного URL-адреса
let url = document.URL;

// получение домена
let domain = document.domain;

// получение источника ссылки
let referrer = document.referrer;
```

Свойства `URL` и `domain` связаны. Например, если свойство `document.URL` содержит адрес `http://www.wrox.com/WileyCDA/`, значением `document.domain` будет `www.wrox.com`.

Из этих трех свойств задать можно только свойство `domain`, при этом действуют некоторые ограничения, связанные с безопасностью. Если URL-адрес содержит поддомен, например `p2p.wrox.com`, значением `domain` может быть только `"wrox.com"` (то же верно, если URL-адрес содержит префикс `"www"`, например `www.wrox.com`). Этому свойству нельзя назначить домен, который не содержится в URL-адресе:

```
// страница с сайта p2p.wrox.com

document.domain = "wrox.com";          // успех

document.domain = "nczonline.net";     // ошибка!
```

Возможность задать свойство `document.domain` полезна, если на странице есть обычный фрейм или встроенный фрейм (iframe) из другого поддомена. Страницы из разных поддоменов не могут взаимодействовать с помощью JavaScript из-за ограничений безопасности. Если присвоить свойству `document.domain` на всех страницах одно и то же значение, им станут доступны JavaScript-объекты других страниц. Например, если на странице, загруженной с адреса www.wrox.com, есть встроенный фрейм со страницей p2p.wrox.com, их значения `document.domain` будут разными, из-за чего JavaScript-объекты внешней страницы будут недоступны внутренней странице, и наоборот. Если у обеих страниц свойство `document.domain` будет иметь значение `"wrox.com"`, страницы смогут взаимодействовать.

Другое ограничение запрещает детализацию свойства домена после того, как ему было назначено более общее значение. Это означает, например, что нельзя сначала присвоить свойству `document.domain` значение `"wrox.com"`, а затем восстановить значение `"p2p.wrox.com"`. Попытка сделать это приведет к ошибке:

```
// страница с сайта p2p.wrox.com

document.domain = "wrox.com";        // обобщение – успех

document.domain = "p2p.wrox.com";    // детализация - ошибка!
```

Получение элементов

Вероятно, самой востребованной DOM-операцией является получение ссылки на конкретный элемент или множество элементов для выполнения каких-либо действий с ними. Для этого тип `Document` предоставляет методы `getElementById()` и `getElementsByTagName()`.

Метод `getElementById()` принимает идентификатор элемента, который нужно получить, и возвращает этот элемент или `null`, если его не существует. Сравнение идентификатора с атрибутом `id` элемента на странице выполняется с учетом регистра. Возьмем для примера следующий элемент:

```
<div id="myDiv">Some text</div>
```

Этот элемент можно получить так:

```
let div = document.getElementById("myDiv");    // получение ссылки на <div>
```

Однако следующий код возвращает `null`:

```
let div = document.getElementById("mydiv");
```

Если страница содержит несколько элементов с одним идентификатором, метод `getElementById()` возвращает первый из них.

Для получения ссылок на элементы также часто используется метод `getElementsByTagName()`. Он принимает имя тега элементов, которые нужно получить, и возвращает объект `NodeList`, содержащий эти элементы. В HTML-документах этот метод возвращает объект `HTMLCollection`, который очень похож на `NodeList` тем,

что тоже обновляется динамически. Например, следующий код возвращает объект HTMLCollection, содержащий все элементы <img> на странице:

```
let images = document.getElementsByTagName("img");
```

Этот код сохраняет объект HTMLCollection в переменной images. Подобно NodeList, элементы объекта HTMLCollection доступны с помощью скобочной нотации и метода item(), а количество элементов можно получить с помощью свойства length:

```
alert(images.length);          // количество изображений
alert(images[0].src);          // атрибут src первого изображения
alert(images.item(0).src);     // атрибут src первого изображения
```

У объекта HTMLCollection есть дополнительный метод namedItem(), позволяющий обращаться к элементам в коллекции по атрибуту name. Предположим, на странице есть такой элемент <img>:

```
<img src="myimage.gif" name="myImage">
```

Получить ссылку на этот элемент <img> из переменной images можно следующим образом:

```
let myImage = images.namedItem("myImage");
```

Таким образом, элементы в объекте HTMLCollection доступны по индексу и по имени, что позволяет легко извлекать их. Именованные элементы также доступны с помощью скобочной нотации:

```
let myImage = images["myImage"];
```

Как видите, скобочную нотацию можно использовать и с числовыми, и со строковыми индексами, при этом в первом случае неявно вызывается метод item(), а во втором — namedItem().

Чтобы получить все элементы в документе, передайте в метод getElementsByTagName() звездочку (*). Как правило, в JavaScript и CSS (Cascading Style Sheets) звездочка означает «все». Вот пример:

```
let allElements = document.getElementsByTagName("*");
```

Этот код возвращает объект HTMLCollection, содержащий все элементы в исходном порядке, то есть первым элементом является <html>, вторым — <head> и т. д.

ПРИМЕЧАНИЕ Хотя в спецификации сказано, что имена тегов чувствительны к регистру, метод getElementsByTagName() не учитывает регистр для совместимости с существующими HTML-страницами. При работе с XML-страницами (включая XHTML) метод getElementsByTagName() переключается в режим, чувствительный к регистру.

Третий метод, getElementsByName(), доступен только для типа HTMLDocument. Как можно догадаться, он возвращает все элементы с конкретным значением атрибута

name. Чаще всего он используется с переключателями, которые должны иметь одно имя на всех, чтобы серверу отправлялось правильное значение:

```
<fieldset>
    <legend>Which color do you prefer?</legend>
    <ul>
        <li>
            <input type="radio" value="red" name="color" id="colorRed">
            <label for="colorRed">Red</label>
        </li>
        <li>
            <input type="radio" value="green" name="color" id="colorGreen">
            <label for="colorGreen">Green</label>
        </li>
        <li>
            <input type="radio" value="blue" name="color" id="colorBlue">
            <label for="colorBlue">Blue</label>
        </li>
    </ul>
</fieldset>
```

Здесь у всех кнопок-переключателей атрибут name имеет значение "color", хотя их идентификаторы различны. Идентификаторы позволяют применить элементы <label> к переключателям, а атрибут name гарантирует, что серверу будет отправлено только одно значение из трех. Эти переключатели можно получить следующим образом:

```
let radios = document.getElementsByName("color");
```

Как и getElementsByTagName(), метод getElementsByName() возвращает объект HTMLCollection, но теперь метод namedItem() этого объекта всегда возвращает первый элемент (потому что все элементы имеют одно имя).

Специальные коллекции

У объекта document есть несколько специальных коллекций. Каждая из них является объектом HTMLCollection и предоставляет быстрый доступ к часто используемым частям документа:

- document.anchors — содержит все элементы <a> с атрибутом name;
- document.applets — содержит все элементы <applet> (эта коллекция устарела, поэтому использовать элемент <applet> больше не рекомендуется);
- document.forms — содержит все элементы <form> (то же, что document.getElementsByTagName("form"));
- document.images — содержит все элементы <img> (то же, что document.getElementsByTagName("img"));
- document.links — содержит все элементы <a> с атрибутом href.

Эти специальные коллекции доступны в любой момент и, как все объекты HTMLCollection, динамически обновляются в соответствии с содержимым текущего документа.

Соответствие спецификации DOM

DOM включает много частей и уровней, поэтому иногда требуется точно определить, какие DOM-компоненты реализованы в браузере. Сведения об этом содержит свойство `document.implementation`. В DOM Level 1 определен только один метод объекта `document.implementation`, а именно `hasFeature()`. Он принимает два аргумента: имя и версию DOM-компонента, который нужно проверить. Если браузер поддерживает запрошенный компонент указанной версии, метод возвращает `true`, например:

```
let hasXmlDom = document.implementation.hasFeature("XML", "1.0");
```

Значения, которые можно проверить, перечислены в таблице.

КОМПОНЕНТ	ПОДДЕРЖИВАЕМЫЕ ВЕРСИИ	ОПИСАНИЕ
Core	1.0, 2.0, 3.0	Основные возможности DOM, регламентирующие представление документа в виде иерархического дерева
XML	1.0, 2.0, 3.0	Расширение компонента Core для работы с XML, обеспечивающее поддержку разделов CDATA, инструкций по обработке и сущностей
HTML	1.0, 2.0	Расширение компонента XML для работы с HTML, обеспечивающее поддержку элементов и сущностей, специфичных для HTML
Views	2.0	Компонент для форматирования документа на основе стилей
StyleSheets	2.0	Компонент, сопоставляющий таблицы стилей с документами
CSS	2.0	Поддержка CSS Level 1
CSS2	2.0	Поддержка CSS Level 2
Events	2.0, 3.0	Универсальные DOM-события
UIEvents	2.0, 3.0	События пользовательского интерфейса
TextEvents	3.0	События, запущенные с устройств ввода текста
MouseEvents	2.0, 3.0	События мыши (щелчок, наведение и т. д.)
MutationEvents	2.0, 3.0	События, генерируемые при изменении DOM-дерева
MutationNameEvents	3.0	События, возникающие при переименовании элементов DOM или атрибутов элементов
HTMLEvents	2.0	События HTML 4.01

КОМПОНЕНТ	ПОДДЕРЖИВАЕМЫЕ ВЕРСИИ	ОПИСАНИЕ
Range	2.0	Объекты и методы для манипулирования диапазоном в DOM-дереве
Traversal	2.0	Методы для обхода DOM-дерева
LS	3.0	Компонент для синхронной загрузки и сохранения DOM-дерева
LS-Async	3.0	Компонент для асинхронной загрузки и сохранения DOM-дерева
Validation	3.0	Методы для изменения DOM-дерева с соблюдением правил
XPath	3.0	Язык для адресации частей XML-документа

Недостаток метода `hasFeature()` в том, что разработчики сами решают, соответствует ли реализация конкретного компонента спецификации DOM. Если он возвращает `true`, это вовсе не означает, что соблюдены все требования спецификации. Например, Safari 2.x и более ранних версий возвращает `true` для нескольких компонентов, которые реализованы не полностью. Прежде чем использовать те или иные части DOM, имеет смысл не только вызвать метод `hasFeature()`, но и распознать требуемые возможности.

Запись документа

С помощью объекта `document` можно добавлять данные в поток вывода веб-страницы, используя методы `write()`, `writeln()`, `open()` и `close()`. Методы `write()` и `writeln()` принимают в качестве аргумента строку, которую нужно добавить в поток вывода. Метод `write()` добавляет ее как есть, а метод `writeln()` дополняет текст знаком перевода строки (`\n`). Эти два метода можно использовать во время загрузки страницы для динамического добавления контента, например:

```
<html>
<head>
    <title>document.write() Example</title>
</head>
<body>
    <p>The current date and time is:
    <script type="text/javascript">
        document.write("<strong>" + (new Date()).toString() + "</strong>");
    </script>
    </p>
</body>
</html>
```

Этот код выводит при загрузке страницы текущие дату и время. Дата заключается в элемент `<strong>`, который обрабатывается так же, как если бы он содержался в HTML-коде страницы. Это означает, что для него создается DOM-элемент,

к которому позднее можно получать доступ. Так обрабатывается любой HTML-код, выводимый с помощью метода `write()` или `writeln()`.

Методы `write()` и `writeln()` часто используются для динамического включения внешних ресурсов, таких как JS-файлы. При включении JS-файлов необходимо проследить за тем, чтобы не добавить строку `"</script>"`, потому что она будет интерпретирована как конец сценария, из-за чего остальной код не выполнится, например:

```
<html>
<head>
    <title>document.write() Example</title>
</head>
<body>
    <script type="text/javascript">
        document.write("<script type=\"text/javascript\" src=\"file.js\">" +
            "</script>");
    </script>
</body>
</html>
```

Хотя этот файл выглядит правильно, тег `"</script>"` в аргументе метода `write()` интерпретируется как конец главного сценария, из-за чего на странице выводятся знаки `");`. Чтобы исправить ситуацию, нужно изменить строку:

```
<html>
<head>
    <title>Document.write() Example</title>
</head>
<body>
    <script type="text/javascript">
        document.write("<script type=\"text/javascript\" src=\"file.js\">" +
             "<\/script>");
    </script>
</body>
</html>
```

Теперь строка `"<\/script>"` не интерпретируется как закрывающий тег внешнего тега `<script>` и лишний контент на странице не выводится.

В предыдущих примерах метод `document.write()` использовался для вывода контента непосредственно на странице во время ее визуализации. Если вызвать метод `document.write()` после завершения загрузки страницы, она будет перезаписана, например:

```
<html>
<head>
    <title>Document.write() Example</title>
</head>
<body>
    <p> This is some content that you won't get to see because it will be
overwritten.</p>
    <script type="text/javascript">
        window.onload = function() {
```

```
            document.write("Hello World!");
        };
    </script>
</body>
</html>
```

Чтобы отложить выполнение функции до полной загрузки страницы, здесь используется обработчик события `window.onload`. По завершении загрузки весь контент страницы перезаписывается строкой `"Hello World!"`.

Методы `open()` и `close()` открывают и закрывают поток вывода веб-страницы соответственно. При использовании методов `write()` и `writeln()` во время загрузки страницы этого не требуется.

> **ПРИМЕЧАНИЕ** Запись в XHTML-документы строгой версии не поддерживается. Со страницами, для которых указан тип контента `application/xml+xhtml`, эти методы работать не будут.

Тип Element

После типа `Document` чаще всего в веб-программировании используется тип `Element`. Он представляет XML- или HTML-элемент, обеспечивая доступ к его сведениям, таким как имя тега, дочерние элементы и атрибуты. Узел `Element` имеет следующие свойства:

- `nodeType` имеет значение 1;
- `nodeName` содержит имя тега элемента;
- `nodeValue` имеет значение `null`;
- `parentNode` может указывать на узел `Document` или `Element`;
- дочерними узлами могут быть объекты `Element`, `Text`, `Comment`, `ProcessingInstruction`, `CDATASection` и `EntityReference`.

Имя тега доступно как свойство `nodeName` или `tagName`; оба свойства возвращают одно и то же значение (ради ясности обычно используется `tagName`). Возьмем для примера следующий элемент:

```
<div id="myDiv"></div>
```

Получить этот элемент и его имя тега можно так:

```
let div = document.getElementById("myDiv");
alert(div.tagName);                    // "DIV"
alert(div.tagName == div.nodeName);    // true
```

Элемент имеет имя тега `div` и идентификатор `"myDiv"`, но `div.tagName` на самом деле возвращает `"DIV"`, а не `"div"`. При работе с HTML имена тегов всегда представляются в верхнем регистре, а с XML (включая XHTML) всегда используется исходный регистр. Если вы не знаете, в каком документе — HTML или XML — будет

использоваться ваш сценарий, имеет смысл унифицировать регистр имен тегов перед сравнением:

```
if (element.tagName == "div") {    // НЕ ДЕЛАЙТЕ ТАК! Возможны ошибки!
    // какие-то действия
}

if (element.tagName.toLowerCase() == "div") {    // Лучше — работает везде
    // какие-то действия
}
```

Этот пример демонстрирует два варианта сравнения имени тега со свойством `tagName`. Первый менее надежен, потому что он не работает в HTML-документах. Второй подход с преобразованием имени тега в нижний регистр работает и в XML-, и в HTML-документах.

Элементы HTML

Все HTML-элементы представляются объектами типа `HTMLElement` или его подтипов. Тип `HTMLElement` унаследован непосредственно от `Element` и содержит несколько дополнительных свойств. Каждое свойство соответствует одному из следующих стандартных атрибутов, доступных у каждого HTML-элемента:

- `id` — уникальный идентификатор элемента в документе;
- `title` — дополнительные сведения об элементе, обычно отображаемые во всплывающей подсказке;
- `lang` — язык содержимого элемента (используется редко);
- `dir` — направление языка (`"ltr"` — слева направо или `"rtl"` — справа налево; также используется редко);
- `className` — эквивалент атрибута `class`, который указывает для элемента CSS-класс. Это свойство нельзя было назвать `class`, потому что `class` — зарезервированное слово в ECMAScript.

Каждое из этих свойств можно использовать для получения и изменения значения соответствующего атрибута. Рассмотрим следующий HTML-элемент:

```
<div id="myDiv" class="bd" title="Body text" lang="en" dir="ltr"></div>
```

Все атрибуты этого элемента можно получить следующим образом:

```
let div = document.getElementById("myDiv");
alert(div.id);            // "myDiv"
alert(div.className);     // "bd"
alert(div.title);         // "Body text"
alert(div.lang);          // "en"
alert(div.dir);           // "ltr"
```

Также можно изменить атрибуты, присвоив свойствам новые значения:

```
div.id = "someOtherId";
div.className = "ft";
```

```
div.title = "Some other text";
div.lang = "fr";
div.dir = "rtl";
```

Не все свойства при перезаписи изменяют вид страницы. Изменения свойств `id` и `lang` незаметны для пользователя (если они не используются в стилях CSS), а изменение свойства `title` проявляется только при наведении указателя мыши на элемент. Свойство `dir` переключает способ выравнивания текста (по левому или правому краю), а изменения `className` видны, если новый стиль CSS отличается от прежнего.

Как уже отмечалось, каждый HTML-элемент представляется экземпляром типа `HTMLElement` или более специфичного подтипа. Доступные элементы и соответствующие им типы указаны в таблице (элементы, выделенные курсивом, устарели).

ЭЛЕМЕНТ	ТИП	ЭЛЕМЕНТ	ТИП
A	`HTMLAnchorElement`	INPUT	`HTMLInputElement`
ABBR	`HTMLElement`	INS	`HTMLModElement`
ACRONYM	`HTMLElement`	*ISINDEX*	*`HTMLIsIndexElement`*
ADDRESS	`HTMLElement`	KBD	`HTMLElement`
APPLET	*`HTMLAppletElement`*	LABEL	`HTMLLabelElement`
AREA	`HTMLAreaElement`	LEGEND	`HTMLLegendElement`
B	`HTMLElement`	LI	`HTMLLIElement`
BASE	`HTMLBaseElement`	LINK	`HTMLLinkElement`
BASEFONT	*`HTMLBaseFontElement`*	MAP	`HTMLMapElement`
BDO	`HTMLElement`	*MENU*	*`HTMLMenuElement`*
BIG	`HTMLElement`	META	`HTMLMetaElement`
BLOCKQUOTE	`HTMLQuoteElement`	NOFRAMES	`HTMLElement`
BODY	`HTMLBodyElement`	NOSCRIPT	`HTMLElement`
BR	`HTMLBRElement`	OBJECT	`HTMLObjectElement`
BUTTON	`HTMLButtonElement`	OL	`HTMLOListElement`
CAPTION	`HTMLTableCaptionElement`	OPTGROUP	`HTMLOptGroupElement`
CENTER	*`HTMLElement`*	OPTION	`HTMLOptionElement`
CITE	`HTMLElement`	P	`HTMLParagraphElement`
CODE	`HTMLElement`	PARAM	`HTMLParamElement`
COL	`HTMLTableColElement`	PRE	`HTMLPreElement`
COLGROUP	`HTMLTableColElement`	Q	`HTMLQuoteElement`
DD	`HTMLElement`	*S*	*`HTMLElement`*
DEL	`HTMLModElement`	SAMP	`HTMLElement`

ЭЛЕМЕНТ	ТИП	ЭЛЕМЕНТ	ТИП
DFN	`HTMLElement`	SCRIPT	`HTMLScriptElement`
DIR	*`HTMLDirectoryElement`*	SELECT	`HTMLSelectElement`
DIV	`HTMLDivElement`	SMALL	`HTMLElement`
DL	`HTMLDListElement`	SPAN	`HTMLElement`
DT	`HTMLElement`	*STRIKE*	*`HTMLElement`*
EM	`HTMLElement`	STRONG	`HTMLElement`
FIELDSET	`HTMLFieldSetElement`	STYLE	`HTMLStyleElement`
FONT	*`HTMLFontElement`*	SUB	`HTMLElement`
FORM	`HTMLFormElement`	SUP	`HTMLElement`
FRAME	`HTMLFrameElement`	TABLE	`HTMLTableElement`
FRAMESET	`HTMLFrameSetElement`	TBODY	`HTMLTableSectionElement`
H1	`HTMLHeadingElement`	TD	`HTMLTableCellElement`
H2	`HTMLHeadingElement`	TEXTAREA	`HTMLTextAreaElement`
H3	`HTMLHeadingElement`	TFOOT	`HTMLTableSectionElement`
H4	`HTMLHeadingElement`	TH	`HTMLTableCellElement`
H5	`HTMLHeadingElement`	THEAD	`HTMLTableSectionElement`
H6	`HTMLHeadingElement`	TITLE	`HTMLTitleElement`
HEAD	`HTMLHeadElement`	TR	`HTMLTableRowElement`
HR	`HTMLHRElement`	TT	`HTMLElement`
HTML	`HTMLHtmlElement`	*U*	*`HTMLElement`*
I	`HTMLElement`	UL	`HTMLUListElement`
IFRAME	`HTMLIFrameElement`	VAR	`HTMLElement`
IMG	`HTMLImageElement`		

У каждого из этих типов есть атрибуты и методы. Многие из типов обсуждаются в этой книге.

Получение атрибутов

У каждого элемента могут быть атрибуты, которые обычно предоставляют дополнительные сведения о нем или его содержимом. Три основных DOM-метода для работы с атрибутами — `getAttribute()`, `setAttribute()` и `removeAttribute()`. Они предназначены для работы с любыми атрибутами, включая те, которые определены как свойства типа `HTMLElement`, например:

```
let div = document.getElementById("myDiv");
alert(div.getAttribute("id"));        // "myDiv"
alert(div.getAttribute("class"));     // "bd"
```

```
alert(div.getAttribute("title"));     // "Body text"
alert(div.getAttribute("lang"));      // "en"
alert(div.getAttribute("dir"));       // "ltr"
```

В метод `getAttribute()` передается фактическое имя атрибута, так что для получения значения атрибута `class` используется имя `"class"` (а не `"className"`, которое необходимо при доступе к атрибуту через свойство объекта). Если атрибут с указанным именем не существует, метод `getAttribute()` всегда возвращает `null`.

С помощью метода `getAttribute()` можно также получать значения пользовательских атрибутов, отсутствующих в языке HTML. Рассмотрим следующий элемент:

```
<div id="myDiv" my_special_attribute="hello!"></div>
```

В этом элементе определен пользовательский атрибут `my_special_attribute` со значением `"hello!"`, которое можно получить с помощью метода `getAttribute()`, как и любое другое значение:

```
let value = div.getAttribute("my_special_attribute");
```

Имена атрибутов нечувствительны к регистру, так что `"ID"` и `"id"` считаются одним и тем же атрибутом. Имейте также в виду, что согласно спецификации HTML5 для успешного прохождения проверки к пользовательским атрибутам нужно добавлять префикс `data-`.

Все атрибуты элемента также доступны как свойства объекта DOM-элемента. Мы уже обсудили пять свойств объекта `HTMLElement`, которые непосредственно соответствуют атрибутам, но к нему добавляются и все остальные общепризнанные (непользовательские) атрибуты. Рассмотрим следующий элемент:

```
<div id="myDiv" align="left" my_special_attribute="hello"></div>
```

Поскольку `id` и `align` являются в HTML общепризнанными атрибутами элемента `<div>`, они будут представлены свойствами объекта элемента, а пользовательский атрибут `my_special_attribute` не будет.

У двух категорий атрибутов их значения в виде свойств не соответствуют значениям, которые возвращает метод `getAttribute()`. Первый — это атрибут `style`, который используется для указания CSS-стиля элемента. При доступе с помощью метода `getAttribute()` атрибут `style` содержит CSS-текст, тогда как при доступе к нему с помощью свойства возвращается объект. Свойство `style` используется для программного доступа к стилю элемента, поэтому оно не соответствует атрибуту `style`.

Вторая категория атрибутов, которые работают иначе, это атрибуты обработчиков событий, такие как `onclick`. При использовании с элементом атрибут `onclick` содержит JS-код, который в виде строки возвращается методом `getAttribute()`. Однако при доступе к свойству `onclick` оно возвращает JavaScript-функцию (или `null`, если атрибут не указан). Свойство `onclick` и другие свойства обработки событий реализованы так, чтобы им можно было назначать функции.

Из-за этих различий программисты часто пренебрегают методом `getAttribute()` при работе с DOM в JavaScript, используя вместо него исключительно свойства объектов. А метод `getAttribute()` применяется преимущественно для получения значений пользовательских атрибутов.

Задание атрибутов

У метода `getAttribute()` есть обратный метод `setAttribute()`, принимающий два аргумента: имя атрибута, который нужно задать, и его значение. Если атрибут уже существует, метод `setAttribute()` обновляет его значение. Если атрибут не существует, метод `setAttribute()` создает его и присваивает ему значение, например:

```
div.setAttribute("id", "someOtherId");
div.setAttribute("class", "ft");
div.setAttribute("title", "Some other text");
div.setAttribute("lang","fr");
div.setAttribute("dir", "rtl");
```

Метод `setAttribute()` работает с HTML-атрибутами и пользовательскими атрибутами одинаково. При его вызове имена атрибутов преобразуются в нижний регистр (например, `"ID"` преобразуется в `"id"`).

Задавать значения атрибутов можно и с помощью соответствующих свойств:

```
div.id = "someOtherId";
div.align = "left";
```

Имейте в виду, что добавление пользовательского свойства к DOM-элементу не делает его автоматически атрибутом элемента:

```
div.mycolor = "red";
alert(div.getAttribute("mycolor"));    // null (кроме Internet Explorer)
```

Этот код добавляет к элементу пользовательское свойство `mycolor` со значением `"red"`. В большинстве браузеров оно не становится автоматически атрибутом элемента, поэтому вызов метода `getAttribute()` с целью получения одноименного атрибута возвращает `null`.

Метод `removeAttribute()` полностью удаляет атрибут элемента, не ограничиваясь очисткой его значения:

```
div.removeAttribute("class");
```

Этот метод используется не очень часто, но может быть полезен для отбора нужных атрибутов при сериализации DOM-элемента.

Свойство attributes

Тип `Element` является единственным типом DOM-узла, у которого есть свойство `attributes`. Оно содержит коллекцию `NamedNodeMap`, динамически обновляемую,

подобно NodeList. Каждый атрибут элемента представляется в объекте NamedNodeMap узлом Attr. У объекта NamedNodeMap есть следующие методы:

- getNamedItem(*имя*) — возвращает узел, у которого свойство nodeName равно указанному имени;
- removeNamedItem(*имя*) — удаляет из списка узел, у которого свойство nodeName равно указанному имени;
- setNamedItem(*узел*) — добавляет узел в список, индексируя его по свойству nodeName;
- item(*позиция*) — возвращает узел в указанную позицию.

Каждым элементом свойства attributes является узел, у которого свойство nodeName содержит имя атрибута, а nodeValue — значение атрибута. Например, получить значение атрибута id элемента можно следующим образом:

```
let id = element.attributes.getNamedItem("id").nodeValue;
```

То же самое можно сделать, используя скобочную нотацию:

```
let id = element.attributes["id"].nodeValue;
```

С помощью скобочной нотации можно также задавать значения атрибутов:

```
element.attributes["id"].nodeValue = "someOtherId";
```

Метод removeNamedItem() работает так же, как и метод removeAttribute() элемента: он просто удаляет атрибут с указанным именем. Единственное его отличие в том, что он возвращает узел Attr, который представляет атрибут:

```
let oldAttr = element.attributes.removeNamedItem("id");
```

Метод setNamedItem() позволяет добавить новый атрибут к элементу, но используется он редко. В качестве параметра он принимает добавляемый узел атрибута:

```
element.attributes.setNamedItem(newAttr);
```

Вообще говоря, вместо этих методов атрибутов лучше использовать методы getAttribute(), removeAttribute() и setAttribute(), потому что они проще.

Свойство attributes полезно, если нужно перебрать атрибуты элемента. Чаще всего это требуется при сериализации DOM-структуры в XML- или HTML-строку. Следующий код перебирает все атрибуты элемента и составляет строку формата *атрибут1*="*значение1*" *атрибут2*="*значение2*":

```
function outputAttributes(element) {
    let pairs = [];
    for (let i = 0, len = element.attributes.length; i < len; ++i) {
        const attribute = element.attributes[i];
        pairs.push(`${attribute.nodeName}="${attribute.nodeValue}"`);
    }

    return pairs.join(" ");
}
```

Эта функция сохраняет пары имен и значений в массиве, а затем объединяет их, добавляя между ними пробел (эта методика часто используется при сериализации данных в длинные строки). Цикл `for`, который выполняется до индекса `attributes.length`, перебирает каждый атрибут, добавляя его имя и значение в строку. Браузеры возвращают элементы объекта `attributes` в разном порядке, поэтому расположение атрибутов в HTML- или XML-коде может отличаться от их очередности в объекте `attributes`.

Создание элементов

Элементы можно создавать с помощью метода `document.createElement()`, который принимает имя тега создаваемого элемента. В HTML-документах регистр имени тега не учитывается, а в XML-документах (включая XHTML) — учитывается. Например, создать элемент `<div>` можно следующим образом:

```
let div = document.createElement("div");
```

Метод `createElement()` создает элемент и задает его свойство `ownerDocument`, после чего можно манипулировать атрибутами элемента, добавлять к нему дочерние элементы и т. д., например:

```
div.id = "myNewDiv";
div.className = "box";
```

Задание этих атрибутов для нового элемента только настраивает его, но не влияет на его отображение в браузере, потому что он не является частью дерева документа. Добавить элемент в дерево документа можно с помощью метода `appendChild()`, `insertBefore()` или `replaceChild()`. Следующий код добавляет новый элемент в элемент `<body>` документа:

```
document.body.appendChild(div);
```

Как только элемент добавлен в дерево документа, браузер сразу же его визуализирует. Любые последующие изменения элемента немедленно отражаются в браузере.

Дочерние узлы элементов

У элементов может быть любое количество дочерних узлов и более дальних потомков. Свойство `childNodes` содержит все непосредственные дочерние узлы элемента, которыми могут быть другие элементы, текстовые узлы, комментарии или инструкции по обработке. Способы идентификации этих узлов в браузерах существенно различаются. Рассмотрим, например, следующий код:

```
<ul id="myList">
    <li>Item 1</li>
    <li>Item 2</li>
    <li>Item 3</li>
</ul>
```

Браузеры идентифицируют семь узлов: три элемента `<li>` и четыре узла `text`, представляющих свободное пространство между элементами. Если удалить свободное пространство, возвращается одинаковое количество дочерних узлов, а именно три:

```
<ul id="myList"><li>Item 1</li><li>Item 2</li><li>Item 3</li></ul>
```

При перемещении по дочерним узлам с помощью свойства `childNodes` важно помнить об этих различиях. Часто перед той или иной операцией требуется проверить тип узла:

```
for (let i=0, len=element.childNodes.length; i < len; i++) {
    if (element.childNodes[i].nodeType == 1) {
        // обработка
    }
}
```

Этот код перебирает все дочерние узлы конкретного элемента, запуская обработку, только если тип узла равен 1 (то есть узел является элементом).

Для получения дочерних узлов и других потомков с конкретным именем тега можно использовать имеющийся у элементов метод `getElementsByTagName()`. Он работает почти так же, как аналогичный метод документа, но начинает поиск с элемента, поэтому возвращаются только его потомки. В предыдущем примере со списком `<ul>` все элементы `<li>` можно получить так:

```
let ul = document.getElementById("myList");
let items = ul.getElementsByTagName("li");
```

В данном случае все потомки элемента `<ul>` относятся к одному уровню, но если бы уровней было больше, этот код возвратил бы элементы `<li>` всех уровней.

Тип Text

Узлы `Text` представляются типом `Text` и содержат обычный текст, который интерпретируется буквально и может включать экранированные HTML-символы, но не HTML-код. Узел `Text` имеет следующие свойства:

- `nodeType` имеет значение 3;
- `nodeName` имеет значение `"#text"`;
- `nodeValue` содержит текст узла;
- `parentNode` указывает на узел `Element`;
- дочерние узлы не поддерживаются.

Для доступа к тексту в узле `Text` можно использовать свойства `nodeValue` и `data`, которые содержат одно и то же значение, а изменение одного из свойств дублируется в другом. Для работы с текстом в узле предназначены следующие методы:

- `appendData(`*текст*`)` — добавляет указанный текст в конец узла;
- `deleteData(`*смещение*, *количество*`)` — удаляет указанное количество символов, начиная с указанного смещения;

- `insertData(`*смещение*, *текст*`)` — вставляет текст в позиции, заданной смещением;
- `replaceData(`*смещение*, *количество*, *текст*`)` — заменяет указанное количество символов указанным текстом, начиная со смещения;
- `splitText(`*смещение*`)` — разделяет текстовый узел на два в позиции, заданной смещением;
- `substringData(`*смещение*, *количество*`)` — извлекает из текста указанное количество символов, начиная со смещения.

Свойство `length` возвращает количество символов в узле, которое совпадает со значениями `nodeValue.length` и `data.length`.

По умолчанию у элементов, способных содержать контент, может быть не более одного текстового узла, например:

```
<!-- элемент пуст — текстового узла нет -->
<div></div>

<!-- элемент содержит пробел — один текстовый узел -->
<div> </div>

<!-- элемент содержит текст — один текстовый узел -->
<div>Hello World!</div>
```

В первом элементе `<div>` нет контента, поэтому и текстового узла у него нет. Если между открывающим и закрывающим тегами имеется какой-либо контент, пусть даже пробел, для него создается текстовый узел. Таким образом, у второго и третьего элементов `<div>` есть по одному дочернему текстовому узлу с пробелом и текстом `"Hello World!"` в качестве значения `nodeValue`. Для доступа к этому узлу можно использовать следующий код:

```
let textNode = div.firstChild;    // или div.childNodes[0]
```

Получив ссылку на текстовый узел, его можно изменить:

```
div.firstChild.nodeValue = "Some other message";    // какой-то другой текст
```

Пока текстовый узел находится в дереве документа, изменения узла вступают в силу немедленно. Отметим также, что значения текстовых узлов кодируются в формате HTML или XML (в зависимости от типа документа), то есть любые знаки «меньше», «больше» и кавычки экранируются:

```
// выводится как "Some &lt;strong&gt;other&lt;/strong&gt; message"
div.firstChild.nodeValue = "Some <strong>other</strong> message";
```

Это эффективный способ кодирования строки в формате HTML перед вставкой в DOM-документ.

Создание текстовых узлов

Текстовые узлы можно создавать с помощью метода `document.createTextNode()`. В качестве аргумента он принимает строку, которую нужно вставить в узел. Как

и при задании значения существующего текстового узла, текст при этом кодируется в формате HTML или XML, например:

```
let textNode = document.createTextNode("<strong>Hello</strong> world!");
```

При создании текстового узла задается его свойство `ownerDocument`, но он не отображается в окне браузера, пока не будет добавлен к узлу в дереве документа. Следующий код создает элемент `<div>` и добавляет в него сообщение:

```
let element = document.createElement("div");
element.className = "message";

let textNode = document.createTextNode("Hello world!");
element.appendChild(textNode);

document.body.appendChild(element);
```

Этот код создает элемент `<div>`, назначает ему класс `"message"`, а затем создает текстовый узел и добавляет его к уже имеющемуся элементу. Наконец, элемент добавляется в тело документа, при этом элемент и текстовый узел отображаются в браузере.

Как правило, у элементов имеется только один дочерний текстовый узел, но их может быть несколько, например:

```
let element = document.createElement("div");
element.className = "message";

let textNode = document.createTextNode("Hello world!");
element.appendChild(textNode);

let anotherTextNode = document.createTextNode("Yippee!");
element.appendChild(anotherTextNode);

document.body.appendChild(element);
```

Если один текстовый узел добавляется вслед за другим, их содержимое выводится на экран без пробела.

Нормализация текстовых узлов

Одноуровневые текстовые узлы в DOM-документах могут вносить путаницу, потому что любую простую строку можно представить одним текстовым узлом. Тем не менее текстовые узлы в DOM-документах можно часто встретить по соседству. Объединить такие узлы можно с помощью метода `normalize()`, который относится к типу `Node` и благодаря этому доступен для узлов всех типов. При вызове этого метода для родителя двух или более текстовых узлов они сливаются в один узел со свойством `nodeValue`, содержащим объединенные значения свойств `nodeValue` исходных узлов, например:

```
let element = document.createElement("div");
element.className = "message";

let textNode = document.createTextNode("Hello World!");
```

```
element.appendChild(textNode);

let anotherTextNode = document.createTextNode("Yippee!");
element.appendChild(anotherTextNode);

document.body.appendChild(element);

alert(element.childNodes.length);        // 2

element.normalize();
alert(element.childNodes.length);        // 1
alert(element.firstChild.nodeValue);     // "Hello World!Yippee!"
```

При синтаксическом анализе документа браузер никогда не создает одноуровневые текстовые узлы. Они могут появляться только при программном DOM-манипулировании.

Разделение текстовых узлов

У типа `Text` есть также метод `splitText()`, который создает из одного текстового узла два, разделяя значение `nodeValue` по указанному смещению. В исходном текстовом узле остается текст до смещения, а остальной текст сохраняется в новом текстовом узле, который возвращается из метода. Свойство `parentNode` нового узла имеет такое же значение, что и у исходного узла. Рассмотрим пример:

```
let element = document.createElement("div");
element.className = "message";

let textNode = document.createTextNode("Hello world!");
element.appendChild(textNode);

document.body.appendChild(element);

let newNode = element.firstChild.splitText(11);
alert(element.firstChild.nodeValue);     // "Hello"
alert(newNode.nodeValue);                // " world!"
alert(element.childNodes.length);        // 2
```

В этом примере текстовый узел со значением `"Hello world!"` делится на два узла в позиции 11, которая соответствует пробелу между словами. После этого исходный текстовый узел содержит строку `"Hello"`, а новый — текст `" world!"` (с начальным пробелом).

Разделение текстовых узлов чаще всего применяется при анализе DOM для извлечения нужных данных.

Тип Comment

Комментарии представляются в DOM экземплярами типа `Comment` со следующими свойствами:

- `nodeType` имеет значение 8;
- `nodeName` имеет значение `"#comment"`;

- `nodeValue` содержит комментарий;
- `parentNode` указывает на узел `Document` или `Element`;
- дочерние узлы не поддерживаются.

Тип `Comment` наследуется от того же базового типа, что и `Text`, поэтому у него есть такие же методы манипулирования строками, исключая `splitText()`. Подобно типу `Text`, получить фактический комментарий можно с помощью свойства `nodeValue` или `data`.

Узел комментария доступен из родительского узла. Рассмотрим следующий HTML-код:

```
<div id="myDiv"><!-- Комментарий --></div>
```

Этот комментарий является дочерним узлом элемента `<div>` и доступен следующим образом:

```
let div = document.getElementById("myDiv");
let comment = div.firstChild;
alert(comment.data);    // "Комментарий"
```

Узел комментария можно создать с помощью метода `document.createComment()`, передав в него текст комментария:

```
let comment = document.createComment("Комментарий");
```

Разработчики редко создают и используют узлы комментариев, потому что функционально они почти бесполезны. Кроме того, браузеры не распознают комментарии после закрывающего тега `</html>`. Если вам нужен доступ к узлам комментариев, убедитесь, что они являются потомками элемента `<html>`.

Тип CDATASection

CDATA-разделы специфичны для XML-документов и представляются типом `CDATASection`. Подобно типу `Comment`, он наследуется от базового типа `Text` и содержит все его методы манипулирования строками, кроме `splitText()`. Перечислим свойства узла `CDATASection`:

- `nodeType` имеет значение 4;
- `nodeName` имеет значение `"#cdata-section"`;
- `nodeValue` представляет собой содержимое CDATA-раздела;
- `parentNode` указывает на узел `Document` или `Element`;
- дочерние узлы не поддерживаются.

CDATA-разделы допустимы только в XML-документах и ошибочно интерпретируются в большинстве браузеров как узлы `Comment` или `Element`. Рассмотрим пример:

```
<div id="myDiv"><![CDATA[Какой-то контент.]]></div>
```

В этом примере узел CDATASection должен быть первым дочерним узлом элемента , но ни один из четырех основных браузеров его так не интерпретирует. Даже правильные XHTML-страницы со встроенными разделами CDATA обрабатываются браузерами некорректно.

Создать раздел CDATA в XML-документе можно с помощью метода document.createCDataSection(), передав ему содержимое узла.

Тип DocumentType

Объект DocumentType содержит все сведения о типе документа. Свойства объекта DocumentType:

- nodeType имеет значение 10;
- nodeName содержит имя типа документа;
- nodeValue имеет значение null;
- parentNode указывает на узел Document;
- дочерние узлы не поддерживаются.

В DOM Level 1 объекты DocumentType нельзя создавать динамически; они создаются только при синтаксическом анализе кода документа. В браузерах, которые поддерживают этот тип, объект DocumentType хранится в свойстве document.doctype. DOM Level 1 определяет три свойства объектов DocumentType: name — имя типа документа, entities — набор NamedNodeMap, который содержит сущности, описываемые типом документа, и notations — набор NamedNodeMap, который содержит обозначения, описываемые типом документа. Поскольку документы, которые отображаются в браузерах, чаще всего имеют тип HTML или XHTML, списки entities и notations обычно пусты (они заполняются только встроенными типами документов). На практике полезно только свойство name, которому присваивается имя типа документа, то есть текст сразу после <!DOCTYPE. Рассмотрим следующий строгий тип документа HTML 4.01:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
  "http://www.w3.org/TR/html4/strict.dtd">
```

У этого типа документа свойство name имеет значение "HTML":

```
alert(document.doctype.name);    // "HTML"
```

Тип DocumentFragment

Тип DocumentFragment — единственный тип узла, у которого нет соответствия в разметке. В DOM фрагмент документа определен как «облегченный» документ, который может содержать узлы и манипулировать ими без накладных расходов, связанных с целым документом. Узлы DocumentFragment обладают следующими свойствами:

- `nodeType` имеет значение 11;
- `nodeName` имеет значение `"#document-fragment"`;
- `nodeValue` имеет значение `null`;
- `parentNode` имеет значение `null`;
- дочерними узлами могут быть узлы `Element`, `ProcessingInstruction`, `Comment`, `Text`, `CDATASection` и `EntityReference`.

Фрагмент документа невозможно добавить в документ. Вместо этого он выступает в роли хранилища других узлов, которые может потребоваться добавить. Фрагменты документа создаются методом `document.createDocumentFragment()`:

```
let fragment = document.createDocumentFragment();
```

Фрагменты документа наследуют все методы от типа `Node` и обычно используются для выполнения разных операций с DOM, которые затем применяются к документу. При добавлении узла во фрагмент документа этот узел удаляется из дерева документа и больше не отображается в браузере. Новые узлы, добавляемые во фрагмент документа, тоже не становятся частью дерева документа. Содержимое фрагмента можно добавить в документ с помощью метода `appendChild()` или `insertBefore()`. Когда фрагмент документа передается как аргумент в один из этих методов, в документ добавляются все дочерние узлы фрагмента, но сам он никогда не добавляется в дерево документа. Рассмотрим, например, следующий HTML-код:

```
<ul id="myList"></ul>
```

Предположим, что нам нужно добавить в этот элемент `<ul>` три элемента списка. При добавлении элементов по отдельности браузер каждый раз визуализировал бы страницу заново. Чтобы избежать этого, можно создать фрагмент документа с элементами списка и добавить их за один раз:

```
let fragment = document.createDocumentFragment();
let ul = document.getElementById("myList");

for (let i=0; i < 3; ++i) {
    let li = document.createElement("li");
    li.appendChild(document.createTextNode(`Item ${i + 1}`));
    fragment.appendChild(li);
}

ul.appendChild(fragment);
```

Этот код начинается с создания фрагмента документа и получения ссылки на элемент `<ul>`. Затем в цикле `for` мы создаем три элемента списка с порядковыми номерами. Для этого в теле цикла мы создаем элемент `<li>` с текстовым узлом и добавляем его к фрагменту документа с помощью метода `appendChild()`. По завершении цикла все элементы списка добавляются в элемент `<ul>` с помощью метода `appendChild()`, которому передается фрагмент документа. При вызове метода все дочерние узлы фрагмента документа удаляются из него и помещаются в элемент `<ul>`.

Тип Attr

Атрибуты элемента представляются в DOM типом `Attr`. Его конструктор и прототип доступны во всех браузерах. Технически атрибуты являются узлами, которые хранятся в свойстве `attributes` элемента. Узлы атрибутов обладают следующими свойствами:

- `nodeType` имеет значение 11;
- `nodeName` содержит имя атрибута;
- `nodeValue` содержит значение атрибута;
- `parentNode` имеет значение `null`;
- в HTML дочерние узлы не поддерживаются;
- в XML дочерними узлами могут быть объекты `Text` и `EntityReference`.

Хотя атрибуты являются узлами, они не считаются частями дерева DOM-документа. Большинство разработчиков редко обращаются к ним напрямую, предпочитая методы `getAttribute()`, `setAttribute()` и `removeAttribute()`.

У объекта `Attr` есть три свойства: `name` — имя атрибута (то же, что `nodeName`); `value` — значение атрибута (то же, что `nodeValue`); `specified` — логическое значение, указывающее, был ли атрибут задан в коде или имеет значение, предлагаемое по умолчанию.

Узел атрибута можно создать с помощью метода `document.createAttribute()`, передав в него имя атрибута. Например, следующий код добавляет к элементу атрибут `align`:

```
let attr = document.createAttribute("align");
attr.value = "left";
element.setAttributeNode(attr);

alert(element.attributes["align"].value);           // "left"
alert(element.getAttributeNode("align").value);     // "left"
alert(element.getAttribute("align"));               // "left"
```

В первой строке создается узел атрибута, при этом его свойство `name` задается методом `createAttribute()`, так что назначать его позже не потребуется. Затем свойству `value` присваивается значение `"left"`. Для добавления нового атрибута к элементу вызывается метод `setAttributeNode()`. Когда атрибут добавлен, к нему можно обращаться с помощью свойства `attributes`, а также методов `getAttributeNode()` и `getAttribute()`. В первых двух случаях возвращается фактический узел `Attr` атрибута, тогда как метод `getAttribute()` возвращает только значение атрибута.

> **ПРИМЕЧАНИЕ** Трудно придумать уважительную причину для непосредственного доступа к узлам атрибутов. Вместо этого лучше использовать методы `getAttribute()`, `setAttribute()` и `removeAttribute()`.

РАБОТА С DOM

Во многих случаях работать с DOM довольно легко, и с помощью JavaScript можно легко создавать разметку, которая обычно пишется на HTML. Однако иногда DOM не так проста, как кажется. Из-за многих скрытых особенностей и несоответствий браузеров одни части DOM более сложны, чем другие.

Динамические сценарии

Элемент `<script>` позволяет вставить в страницу JS-сценарий из внешнего файла с помощью атрибута `src` или добавить код непосредственно. Динамическими называют такие сценарии, которые не существуют при загрузке страницы, а добавляются позднее с помощью DOM. Как и в случае элемента `<script>`, это можно сделать двумя способами: прочитать код из внешнего файла или вставить его непосредственно.

В динамической загрузке внешнего JS-файла нет ничего необычного. Возьмем для примера следующий элемент `<script>`:

```
<script src="foo.js"></script>
```

Этому элементу соответствует следующий DOM-код:

```
let script = document.createElement("script");
script.src = "foo.js";
document.body.appendChild(script);
```

Внешний файл загружается с сервера, только когда элемент `<script>` добавляется в код страницы в последней строке. Его также можно добавить в элементе `<head>` с тем же эффектом. На основе этого кода можно создать универсальную функцию загрузки сценариев:

```
function loadScript(url) {
    let script = document.createElement("script");
    script.src = url;
    document.body.appendChild(script);
}
```

Теперь внешние JS-файлы можно загружать следующим образом:

```
loadScript("client.js");
```

После загрузки сценарий становится доступен в остальной части страницы, но как узнать, когда он полностью загружен? К сожалению, стандартного способа сделать это не существует. Можно, например, использовать определенные события, которые мы обсудим в главе 17 «События».

Другой способ добавить в страницу JS-код — встраивание, например:

```
<script type="text/javascript">
    function sayHi() {
        alert("Hello");
}
</script>
```

Логично было бы предположить, что соответствующий DOM-код будет таким:

```
let script = document.createElement("script");
script.appendChild(document.createTextNode(
    "function sayHi() {alert('Hello');}"));
document.body.appendChild(script);
```

Этот код работает в Firefox, Safari, Chrome и Opera, но в старых версиях Internet Explorer он приводит к ошибке, потому что Internet Explorer интерпретирует элементы `<script>` как специальные и блокирует доступ к их дочерним узлам в стиле DOM. Вместо этого можно назначить JS-код свойству `text`, которое есть у всех элементов `<script>`:

```
let script = document.createElement("script");
script.text = "function sayHi() {alert('Hello');}";
document.body.appendChild(script);
```

Этот обновленный код работает в Internet Explorer, Firefox, Opera, а также в Safari 3 и более поздних версий. Safari до версии 3 реализует свойство `text` неправильно, но и эти старые версии позволяют назначить нужный код текстовому узлу. Следовательно, если нужно обеспечить поддержку ранних версий Safari, можно использовать такой код:

```
var script = document.createElement("script");
var code = "function sayHi() {alert('Hello');}";
try {
    script.appendChild(document.createTextNode("code"));
} catch (ex) {
    script.text = "code";
}
document.body.appendChild(script);
```

Здесь мы сначала пробуем стандартный в DOM способ с текстовым узлом, потому что он работает во всех браузерах, кроме Internet Explorer. Если возникает ошибка, значит, мы имеем дело с Internet Explorer и должны использовать свойство `text`. Теперь можно создать универсальную функцию:

```
function loadScriptString(code) {
    var script = document.createElement("script");
    script.type = "text/javascript";
    try {
        script.appendChild(document.createTextNode(code));
    } catch (ex) {
        script.text = code;
    }
    document.body.appendChild(script);
}
```

Используется эта функция следующим образом:

```
loadScriptString("function sayHi() {alert('Hello');}");
```

Код, загруженный таким образом, выполняется в глобальной области видимости и доступен сразу же после завершения сценария. По сути, этот подход эквивалентен передаче строки кода в метод `eval()` в глобальной области видимости.

Важно отметить, что все элементы `<script>`, созданные с помощью innerHTML, никогда не будут выполнены. Браузер покорно создаст элемент `<script>` и текст сценария внутри него, но анализатор пометит `<script>` как элемент, который никогда не должен выполняться. После создания элемента с использованием `innerHTML` невозможно перенести запуск сценария.

Динамические стили

CSS-стили добавляются в HTML-код с помощью элементов `<link>` и `<style>`. Первый включает CSS из внешнего файла, а второй используется для указания встроенных стилей. Как и динамические сценарии, динамические стили отсутствуют в коде страницы при ее первоначальной загрузке и добавляются, только когда страница загружена.

Вот пример типичного элемента `<link>`:

```
<link rel="stylesheet" type="text/css" href="styles.css">
```

Его также можно легко создать с помощью следующего DOM-кода:

```
let link = document.createElement("link");
link.rel = "stylesheet";
link.type = "text/css";
link.href = "styles.css";
let head = document.getElementsByTagName("head")[0];
head.appendChild(link);
```

Этот код нормально работает во всех основных браузерах. Имейте в виду, что элементы `<link>` нужно добавлять в элемент `<head>`, а не `<body>`, чтобы все браузеры обрабатывали их правильно. Код загрузки стилей можно обобщить с помощью следующей функции:

```
function loadStyles(url) {
    let link = document.createElement("link");
    link.rel = "stylesheet";
    link.type = "text/css";
    link.href = url;
    let head = document.getElementsByTagName("head")[0];
    head.appendChild(link);
}
```

Использовать эту функцию можно так:

```
loadStyles("styles.css");
```

Стили из внешнего файла загружаются асинхронно, то есть независимо от другого JS-кода. Определять, когда завершается загрузка стилей, обычно не требуется.

Другой способ определить стили — использовать элемент `<style>`, содержащий встроенный CSS-код, например:

```
<style type="text/css">
body {
    background-color: red;
}
</style>
```

По идее, с той же целью можно использовать следующий DOM-код:

```
let style = document.createElement("style");
style.type = "text/css";
style.appendChild(document.createTextNode("body{background-color:red}"));
let head = document.getElementsByTagName("head")[0];
head.appendChild(style);
```

Этот код работает в Firefox, Safari, Chrome и Opera, но не в Internet Explorer. Подобно узлу `<script>`, Internet Explorer считает узел `<style>` специальным и запрещает доступ к его дочерним узлам. При этом генерируется такая же ошибка, что и при попытке добавить дочерний узел в элемент `<script>`. Обойти эту проблему можно с помощью свойства `styleSheet`, содержащего, в свою очередь, свойство `cssText`, которому можно назначить CSS-код:

```
let style = document.createElement("style");
style.type = "text/css";
try{
    style.appendChild(document.createTextNode(
        "body{background-color:red}"));
} catch (ex) {
    style.styleSheet.cssText = "body{background-color:red}";
}
let head = document.getElementsByTagName("head")[0];
head.appendChild(style);
```

Как и при динамическом добавлении встроенных сценариев, ошибка в Internet Explorer перехватывается здесь в блоке `catch`, в котором затем используется способ задания стилей, специфичный для Internet Explorer. Универсальное решение будет таким:

```
function loadStyleString(css) {
    let style = document.createElement("style");
    style.type = "text/css";
    try{
        style.appendChild(document.createTextNode(css));
    } catch (ex) {
        style.styleSheet.cssText = css;
    }
    let head = document.getElementsByTagName("head")[0];
    head.appendChild(style);
}
```

Вызвать эту функцию можно следующим образом:

```
loadStyleString("body{background-color:red}");
```

Стили, заданные таким образом, добавляются в страницу незамедлительно и сразу же изменяют ее вид.

ПРИМЕЧАНИЕ Если вы пишете код специально для Internet Explorer, будьте осторожны со свойством `styleSheet.cssText`. Если при повторном использовании элемента `<style>` попытаться задать это свойство более одного раза, может произойти критический сбой в работе браузера. К нему также может привести присваивание свойству `cssText` пустой строки.

Работа с таблицами

Элемент `<table>` является в HTML одним из самых сложных. Чтобы создать таблицу, обычно нужно добавить большое количество тегов строк, ячеек, заголовков и т. д. Неудивительно, что для создания и изменения таблиц с использованием базовых DOM-методов может потребоваться много кода. Допустим, мы хотим создать с помощью DOM следующую HTML-таблицу:

```
<table border="1" width="100%">
    <tbody>
        <tr>
            <td>Cell 1,1</td>
            <td>Cell 2,1</td>
        </tr>
        <tr>
            <td>Cell 1,2</td>
            <td>Cell 2,2</td>
        </tr>
    </tbody>
</table>
```

Если использовать только базовые DOM-методы, получится примерно такой код:

```
// создание таблицы
let table = document.createElement("table");
table.border = 1;
table.width = "100%";

// создание тела таблицы
let tbody = document.createElement("tbody");
table.appendChild(tbody);

// создание первой строки
let row1 = document.createElement("tr");
tbody.appendChild(row1);
let cell1_1 = document.createElement("td");
cell1_1.appendChild(document.createTextNode("Cell 1,1"));
row1.appendChild(cell1_1);
let cell2_1 = document.createElement("td");
cell2_1.appendChild(document.createTextNode("Cell 2,1"));
row1.appendChild(cell2_1);

// создание второй строки
```

```
let row2 = document.createElement("tr");
tbody.appendChild(row2);
let cell1_2 = document.createElement("td");
cell1_2.appendChild(document.createTextNode("Cell 1,2"));
row2.appendChild(cell1_2);
let cell2_2= document.createElement("td");
cell2_2.appendChild(document.createTextNode("Cell 2,2"));
row2.appendChild(cell2_2);

// добавление таблицы в тело документа
document.body.appendChild(table);
```

Этот код довольно объемен и сложен для понимания. Чтобы упростить создание таблиц, в DOM HTML для элементов `<table>`, `<tbody>` и `<tr>` определены дополнительные свойства и методы.

Для элемента `<table>` доступны следующие свойства и методы:

- `caption` — указатель на элемент `<caption>` (если он существует);
- `tBodies` — коллекция `HTMLCollection`, содержащая элементы `<tbody>`;
- `tFoot` — указатель на элемент `<tfoot>` (если он существует);
- `tHead` — указатель на элемент `<thead>` (если он существует);
- `rows` — коллекция `HTMLCollection`, содержащая все строки таблицы;
- `createTHead()` — создает элемент `<thead>`, помещает его в таблицу и возвращает ссылку на него;
- `createTFoot()` — создает элемент `<tfoot>`, помещает его в таблицу и возвращает ссылку на него;
- `createCaption()` — создает элемент `<caption>`, помещает его в таблицу и возвращает ссылку на него;
- `deleteTHead()` — удаляет элемент `<thead>`;
- `deleteTFoot()` — удаляет элемент `<tfoot>`;
- `deleteCaption()` — удаляет элемент `<caption>`;
- `deleteRow(`*позиция*`)` — удаляет строку в указанной позиции;
- `insertRow(`*позиция*`)` — вставляет строку в коллекцию `rows` в указанной позиции.

Для элемента `<tbody>` доступны следующие свойства и методы:

- `rows` — набор `HTMLCollection`, содержащий строки элемента `<tbody>`;
- `deleteRow(`*позиция*`)` — удаляет строку в указанной позиции;
- `insertRow(`*позиция*`)` — вставляет строку в коллекцию `rows` в указанной позиции и возвращает ссылку на нее.

Для элемента `<tr>` доступны следующие свойства и методы:

- `cells` — коллекция `HTMLCollection`, содержащая ячейки элемента `<tr>`;
- `deleteCell(`*позиция*`)` — удаляет ячейку в указанной позиции;

- `insertCell(`*позиция*`)` — вставляет ячейку в коллекцию `cells` в указанной позиции и возвращает ссылку на нее.

Эти свойства и методы значительно сокращают код создания таблицы. Например, предыдущий пример можно переписать с их помощью следующим образом (новый код выделен):

```
// создание таблицы
let table = document.createElement("table");
table.border = 1;
table.width = "100%";

// создание тела таблицы
let tbody = document.createElement("tbody");
table.appendChild(tbody);

// создание первой строки
tbody.insertRow(0);
tbody.rows[0].insertCell(0);
tbody.rows[0].cells[0].appendChild(document.createTextNode("Cell 1,1"));
tbody.rows[0].insertCell(1);
tbody.rows[0].cells[1].appendChild(document.createTextNode("Cell 2,1"));

// создание второй строки
tbody.insertRow(1);
tbody.rows[1].insertCell(0);
tbody.rows[1].cells[0].appendChild(document.createTextNode("Cell 1,2"));
tbody.rows[1].insertCell(1);
tbody.rows[1].cells[1].appendChild(document.createTextNode("Cell 2,2"));

// добавление таблицы в тело документа
document.body.appendChild(table);
```

Код создания элементов `<table>` и `<tbody>` остался прежним. Изменилось только создание двух строк, для чего теперь используются свойства и методы DOM HTML. Чтобы создать первую строку, для элемента `<tbody>` вызывается метод `insertRow()` с аргументом 0, который указывает позицию новой строки в таблице. После этого строка доступна как `tbody.rows[0]`.

Ячейки создаются подобным образом — с помощью метода `insertCell()`, который вызывается для элемента `<tr>` с аргументом, указывающим позицию новой ячейки. Например, первая ячейка в первой строке доступна после создания как `tbody.rows[0].cells[0]`.

Использование этих свойств и методов для создания таблиц делает код более логичным и упрощает его чтение, хотя и первый подход вполне приемлем.

Использование объектов NodeList

Знание особенностей объекта `NodeList` и похожих на него объектов `NamedNodeMap` и `HTMLCollection` критически важно для хорошего понимания DOM в целом. Все эти наборы динамически обновляются при изменении структуры документа,

благодаря чему всегда содержат наиболее актуальные сведения. На самом деле все объекты `NodeList` являются запросами, которые выполняются для DOM-документа при обращении к ним. Например, выполнение следующего кода приводит к бесконечному циклу:

```
let divs = document.getElementsByTagName("div");
for (let i=0; i < divs.length; ++i) {
    let div = document.createElement("div");
    document.body.appendChild(div);
}
```

Первая строка этого кода возвращает объект `HTMLCollection`, содержащий все элементы `<div>` в документе. Так как эта коллекция динамически обновляется, в нее заносится каждый новый элемент `<div>`, добавляемый на страницу. Браузер не ведет список всех созданных коллекций, а обновляет их, только когда они используются в коде, из-за чего возникает интересная проблема с циклом. В начале каждой его итерации оценивается условие `i <divs.length`, при этом выполняется запрос на получение всех элементов `<div>`. Поскольку в теле цикла в документ добавляется новый элемент `<div>`, значение `divs.length` увеличивается при каждой итерации цикла и всегда превышает `i`.

Использование итератора ES6 не исправляет ситуацию, потому что постоянно растущая живая коллекция остается предметом итерации. Это все равно приведет к бесконечному циклу:

```
for (let div of document.getElementsByTagName("div")){
    let newDiv = document.createElement("div");
    document.body.appendChild(newDiv);
}
```

Всякий раз, когда нужно перебрать коллекцию `NodeList`, лучше инициализировать вторую переменную значением ее длины и сравнивать итератор с этой переменной:

```
let divs = document.getElementsByTagName("div");

for (let i=0, len=divs.length; i < len; ++i) {
    let div = document.createElement("div");
    document.body.appendChild(div);
}
```

Здесь переменной `len` присваивается значение `divs.length` на момент начала цикла. Так как при каждой итерации оно остается неизменным, это предотвращает бесконечный цикл. Этот прием использовался в данной главе при демонстрации рекомендуемого способа перебора объектов `NodeList`.

С другой стороны, во избежание использования второй переменной можно также перебрать список в обратном порядке:

```
let divs = document.getElementsByTagName("div");

for (let i = divs.length - 1; i >= 0; --i) {
```

```
    let div = document.createElement("div");
    document.body.appendChild(div);
}
```

Обращаться к объекту `NodeList` без необходимости не следует, потому что при этом каждый раз выполняется запрос документа. Для экономии ресурсов старайтесь кешировать часто используемые значения, полученные из `NodeList`.

НАБЛЮДАТЕЛИ ЗА ИЗМЕНЕНИЯМИ

`MutationObserver` API, сравнительно недавнее дополнение к спецификации DOM, позволяет асинхронно выполнять обратный вызов при изменении DOM. С помощью `MutationObserver` можно наблюдать весь документ, поддерево DOM или только один элемент. Кроме того, также есть возможность наблюдать изменения атрибутов элемента, дочерних узлов, текста или любой их комбинации.

> **ПРИМЕЧАНИЕ** `MutationObservers` были введены для замены `MutationEvents`.

Основные примеры использования

Экземпляр `MutationObserver` создается путем вызова конструктора `MutationObserver` и передачи функции обратного вызова:

```
let observer = new MutationObserver(() => console.log('DOM was mutated!'))
```

Метод observe()

Сначала этот экземпляр не связан с какой-либо частью DOM. Чтобы связать наблюдателя с DOM, используется метод `observe()`. Он принимает два обязательных аргумента: целевой DOM-узел, который наблюдается для изменений, и объект `MutationObserverInit`.

Объект `MutationObserverInit` используется для управления изменениями, которые должен отслеживать наблюдатель. Он принимает форму словаря конфигурационных параметров в виде пар ключ–значение. Например, следующий код создает наблюдателя и настраивает его для отслеживания изменений атрибутов элемента `body`:

```
let observer = new MutationObserver(() => console.log('<body> attributes
changed'));

observer.observe(document.body, { attributes: true });
```

На этом этапе любые изменения атрибута элемента `<body>` будут обнаружены экземпляром `MutationObserver`, и обратный вызов будет выполняться асинхронно. Модификации дочерних или других неатрибутивных изменений DOM не запланируют обратный вызов. Такое поведение демонстрируется здесь:

```
let observer = new MutationObserver(() => console.log('<body> attributes
changed'));

observer.observe(document.body, { attributes: true });

document.body.className = 'foo';
console.log('Changed body class');

// Changed body class
// <body> attributes changed
```

Обратите внимание: обратный вызов `console.log` выполняется вторым, это указывает на то, что обратный вызов не выполняется синхронно с фактическим изменением DOM.

Работа с обратными вызовами и MutationRecords

Каждый обратный вызов предоставляется с массивом экземпляров `MutationRecord`. Каждый экземпляр содержит информацию о том, какое изменение произошло и какая часть DOM была затронута. Поскольку существует вероятность того, что несколько изменений произошли до выполнения обратного вызова, каждому запуску обратного вызова передается резервная копия в очереди экземпляров `MutationRecord`.

Массив `MutationRecord` для изменения одного атрибута:

```
let observer = new MutationObserver(
    (mutationRecords) => console.log(mutationRecords));

observer.observe(document.body, { attributes: true });

document.body.setAttribute('foo', 'bar');

// [
//    {
//        addedNodes: NodeList [],
//        attributeName: "foo",
//        attributeNamespace: null,
//        nextSibling: null,
//        oldValue: null,
//        previousSibling: null
//        removedNodes: NodeList [],
//        target: body
//        type: "attributes"
//    }
// ]
```

Подобное изменение с участием пространства имен:

```
let observer = new MutationObserver(
    (mutationRecords) => console.log(mutationRecords));

observer.observe(document.body, { attributes: true });
```

```
document.body.setAttributeNS('baz', 'foo', 'bar');

// [
//    {
//        addedNodes: NodeList [],
//        attributeName: "foo",
//        attributeNamespace: "baz",
//        nextSibling: null,
//        oldValue: null,
//        previousSibling: null
//        removedNodes: NodeList [],
//        target: body
//        type: "attributes"
//    }
// ]
```

Последовательные изменения будут генерировать несколько экземпляров `MutationRecord`, и при следующем запуске обратного вызова будут переданы все ожидающие экземпляры в порядке их постановки в очередь:

```
let observer = new MutationObserver(
    (mutationRecords) => console.log(mutationRecords));

observer.observe(document.body, { attributes: true });

document.body.className = 'foo';
document.body.className = 'bar';
document.body.className = 'baz';

// [MutationRecord, MutationRecord, MutationRecord]
```

Экземпляр `MutationRecord` имеет следующие свойства.

КЛЮЧ	ЗНАЧЕНИЕ
`target`	Узел, на который повлияло изменение
`type`	Строка, указывающая, какой тип изменения произошел. Может иметь значение `attribute`, `characterData` или `childList`
`oldValue`	При включении в объекте `MutationObserverInit` атрибуты или изменения `characterData` будут устанавливать в этом поле значение, которое было заменено. Это значение предоставляется только в том случае, если `attributeOldValue` или `characterDataOldValue` имеет значение true, в противном случае оно равно `null`. Изменение `childList` всегда будет устанавливать это поле в `null`
`attributeName`	Для изменений `attributes` – строковое имя атрибута, который был изменен. Для всех остальных изменений это поле имеет значение `null`

КЛЮЧ	ЗНАЧЕНИЕ
attributeNamespace	Для изменений attributes с использованием пространства имен – строковое пространство имен атрибута, который был изменен. Для всех остальных изменений это поле имеет значение null
addedNodes	Для изменений childList возвращает NodeList узлов, добавленных при изменении. По умолчанию используется пустой NodeList
removedNodes	Для мутаций childList возвращает NodeList узлов, удаленных при изменении. По умолчанию используется пустой NodeList
previousSibling	Для мутаций childList возвращает предыдущий Node, родственный измененному узлу. По умолчанию равен null
nextSibling	Для мутаций childList возвращает следующий Node, родственный измененному узлу. По умолчанию равен null

Вторым аргументом для обратного вызова является экземпляр MutationObserver, который обнаружил изменение:

```
let observer = new MutationObserver(
    (mutationRecords, mutationObserver) => console.log(mutationRecords,
mutationObserver));

observer.observe(document.body, { attributes: true });

document.body.className = 'foo';
// [MutationRecord], MutationObserver
```

Метод disconnect()

По умолчанию обратный вызов MutationObserver будет выполняться для каждого изменения DOM в назначенной ему области до тех пор, пока элемент не будет собран сборщиком мусора. Чтобы досрочно прекратить выполнение обратного вызова, можно вызвать метод disconnect(). Этот пример демонстрирует, как синхронный вызов disconnect() будет останавливать обратные вызовы и отбрасывать любые ожидающие асинхронные обратные вызовы, даже если они были вызваны изменением DOM во время наблюдения:

```
let observer = new MutationObserver(() => console.log('<body> attributes
               changed'));

observer.observe(document.body, { attributes: true });
```

```
document.body.className = 'foo';

observer.disconnect();

document.body.className = 'bar';

// (ничего не было записано)
```

Чтобы разрешить выполнение обратных вызовов в очереди перед вызовом `disconnect()`, можно использовать `setTimeout`, чтобы разрешить выполнение ожидающих обратных вызовов:

```
let observer = new MutationObserver(() => console.log('<body> attributes
changed'));

observer.observe(document.body, { attributes: true });

document.body.className = 'foo';

setTimeout(() => {
    observer.disconnect();
    document.body.className = 'bar';
}, 0);

// <body> attributes changed
```

Мультиплексирование MutationObserver

`MutationObserver` может быть связан с несколькими различными целевыми элементами при многократном вызове `observe()`. Свойство `target` в `MutationRecord` может идентифицировать, какой элемент был подвержен этому конкретному изменению. Такое поведение демонстрируется ниже:

```
let observer = new MutationObserver(
               (mutationRecords) => console.log(mutationRecords.map((x) =>
x.target)));

// Добавление двух дочерних элементов в body
let childA = document.createElement('div'),
    childB = document.createElement('span');
document.body.appendChild(childA);
document.body.appendChild(childB);

// Привязка наблюдения к обоим элементам
observer.observe(childA, { attributes: true });
observer.observe(childB, { attributes: true });

// Изменение каждого дочернего элемента
childA.setAttribute('foo', 'bar');
childB.setAttribute('foo', 'bar');

// [<div>, <span>]
```

Метод `disconnect()` — грубый инструмент, отключающий все наблюдаемые узлы:

```
let observer = new MutationObserver(
              (mutationRecords) => console.log(mutationRecords.map((x) =>
x.target)));

// Добавление двух дочерних элементов в body
let childA = document.createElement('div'),
    childB = document.createElement('span');

document.body.appendChild(childA);
document.body.appendChild(childB);

// Привязка наблюдения к обоим элементам
observer.observe(childA, { attributes: true });
observer.observe(childB, { attributes: true });

observer.disconnect();

// Изменение каждого дочернего элемента
childA.setAttribute('foo', 'bar');
childB.setAttribute('foo', 'bar');

// (ничего не было записано)
```

Повторное использование MutationObserver

Вызов `disconnect()` не является окончательным событием для `MutationObserver`. Этот же экземпляр может быть снова присоединен к узлу. В следующем примере демонстрируется это поведение путем отключения и повторного подключения в двух последовательных асинхронных блоках:

```
let observer = new MutationObserver(() => console.log('<body> attributes
changed'));

observer.observe(document.body, { attributes: true });

// Это будет зарегистрировано как изменение
document.body.setAttribute('foo', 'bar');

setTimeout(() => {
    observer.disconnect();

// Это не будет зарегистрировано как изменение
 document.body.setAttribute('bar', 'baz');
}, 0);

setTimeout(() => {

    // Повторное прикрепление
    observer.observe(document.body, { attributes: true });

    // Это будет зарегистрировано как изменение
    document.body.setAttribute('baz', 'qux');
```

```
}, 0);

// <body> attributes changed
// <body> attributes changed
```

Управление областью наблюдения с помощью MutationObserverInit

Объект `MutationObserverInit` используется для управления тем, какие элементы должен отслеживать наблюдатель и на какие изменения этих элементов он должен реагировать. Вообще говоря, наблюдатель может следить за изменениями атрибутов, текста или дочерних узлов.

Ниже приведены ожидаемые свойства объекта `MutationObserverInit`.

КЛЮЧ	ЗНАЧЕНИЕ
`subtree`	Логическое значение, указывающее, следует ли отслеживать поддерево узла целевого элемента в дополнение к целевому элементу. При значении `false` только целевой элемент будет отслеживаться для обозначенных изменений. При значении `true` целевой элемент и все его поддерево узла будут отслеживаться для обозначенных изменений. По умолчанию имеет значение `false`
`attributes`	Логическое значение, указывающее, должны ли модификации атрибутов узла регистрироваться как изменение. По умолчанию имеет значение `false`
`attributeFilter`	Массив строковых значений, указывающий, какие конкретные атрибуты следует отслеживать при изменениях. Установка этого значения в `true` также приведет к установке значения `attributes` в `true`. По умолчанию установлено наблюдение всех атрибутов
`attributeOldValue`	Логическое значение, указывающее, должны ли данные символов до изменения быть записаны в `MutationRecord`. Установка этого значения в массив также приведет к установке значения `attributes` в `true`. По умолчанию имеет значение `false`
`characterData`	Логическое значение, указывающее, должны ли регистрироваться изменения в символьных данных. По умолчанию имеет значение `false`

КЛЮЧ	ЗНАЧЕНИЕ
`characterDataOldValue`	Логическое значение, указывающее, должны ли данные символов до изменения быть записаны в `MutationRecord`. Установка этого значения в `true` также приведет к установке значения свойства `characterData` в `true`. По умолчанию имеет значение `false`
`childList`	Логическое значение, указывающее, должны ли модификации дочерних узлов целевого узла регистрироваться как изменение. По умолчанию имеет значение `false`

ПРИМЕЧАНИЕ При вызове `observe()` объект `MutationObserverInit` должен указывать, что по крайней мере один из атрибутов, `characterData` или `childList`, имеет значение `true` (явным или неявным образом через связанное свойство, такое как `attributeOldValue`). В противном случае будет сгенерирована ошибка, поскольку не существует изменения, которое вызывало бы обратный вызов.

Отслеживание изменений атрибутов

`MutationObserver` способен регистрировать добавление, удаление или изменение атрибута узла. Регистрация обратного вызова достигается установкой свойства `attribute` внутри объекта `MutationObserverInit` в значение `true`, как показано здесь:

```
let observer = new MutationObserver(
    (mutationRecords) => console.log(mutationRecords));

observer.observe(document.body, { attributes: true });

// Добавление атрибута
document.body.setAttribute('foo', 'bar');

// Изменение существующего атрибута
document.body.setAttribute('foo', 'baz');

// Удаление атрибута
document.body.removeAttribute('foo');

// Все три действия зарегистрированы как изменения
// [MutationRecord, MutationRecord, MutationRecord]
```

Поведение по умолчанию — наблюдать за всеми изменениями атрибутов и не записывать старое значение внутри `MutationRecord`. Если необходимо отслеживать подмножество атрибутов, свойство `attributeFilter` можно использовать в качестве списка имен атрибутов:

```
let observer = new MutationObserver(
    (mutationRecords) => console.log(mutationRecords));

observer.observe(document.body, { attributeFilter: ['foo'] });

// Добавление атрибута в список
document.body.setAttribute('foo', 'bar');

// Добавление исключенного атрибута
document.body.setAttribute('baz', 'qux');

// Для изменения атрибута 'foo' создается только одна запись
// [MutationRecord]
```

Если нужно сохранить старое значение внутри записи об изменении, `attributeOldValue` может быть установлен в значение `true`:

```
let observer = new MutationObserver(
    (mutationRecords) => console.log(mutationRecords.map((x) => x.oldValue)));

observer.observe(document.body, { attributeOldValue: true });

document.body.setAttribute('foo', 'bar');
document.body.setAttribute('foo', 'baz');
document.body.setAttribute('foo', 'qux');

// При каждом изменении записывается предыдущее значение
// [null, 'bar', 'baz']
```

Отслеживание изменений символьных данных

`MutationObserver` способен регистрировать изменение текстового узла (такого как узлы `Text`, `Comment` или `ProcessingInstruction`) при добавлении, удалении или изменении символьных данных. Это достигается путем установки для свойства `characterData` внутри объекта `MutationObserverInit` значения `true`, как показано здесь:

```
let observer = new MutationObserver(
    (mutationRecords) => console.log(mutationRecords));

// Создание начального текстового узла для отслеживания
document.body.innerText = 'foo';

observer.observe(document.body.firstChild, { characterData: true });

// Назначение строки, идентичной по содержанию
document.body.innerText = 'foo';

// Назначение новой строки
document.body.innerText = 'bar';

// Назначение через средство записи свойства узла
document.body.firstChild.textContent = 'baz';

// Все три изменения зарегистрированы
// [MutationRecord, MutationRecord, MutationRecord]
```

Поведение по умолчанию — не записывать старое значение внутри `MutationRecord`. Если нужно сохранить старое значение внутри записи мутации, `attributeOldValue` может быть установлен в `true`:

```
let observer = new MutationObserver(
    (mutationRecords) => console.log(mutationRecords.map((x) => x.oldValue)));
document.body.innerText = 'foo';

observer.observe(document.body.firstChild, { characterDataOldValue: true });

document.body.innerText = 'foo';
document.body.innerText = 'bar';
document.body.firstChild.textContent = 'baz';

// При каждом изменении записывается предыдущее значение
// ["foo", "foo", "bar"]
```

Наблюдение за изменениями дочерних элементов

`MutationObserver` способен регистрировать изменения при добавлении или удалении дочернего узла для элемента. Это можно сделать при установке для свойства `childList` внутри объекта `MutationObserverInit` значения `true`.

Добавление дочернего узла демонстрируется здесь:

```
// очищение body
document.body.innerHTML = '';

let observer = new MutationObserver(
    (mutationRecords) => console.log(mutationRecords));

observer.observe(document.body, { childList: true });

document.body.appendChild(document.createElement('div'));

// [
//    {
//        addedNodes: NodeList[div],
//        attributeName: null,
//        attributeNamespace: null,
//        oldValue: null,
//        nextSibling: null,
//        previousSibling: null,
//        removedNodes: NodeList[],
//        target: body,
//        type: "childList",
//    }
// ]
```

Удаление дочернего узла:

```
// очищение body
document.body.innerHTML = '';

let observer = new MutationObserver(
```

```
    (mutationRecords) => console.log(mutationRecords));

observer.observe(document.body, { childList: true });

document.body.appendChild(document.createElement('div'));

// [
//    {
//        addedNodes: NodeList[],
//        attributeName: null,
//        attributeNamespace: null,
//        oldValue: null,
//        nextSibling: null,
//        previousSibling: null,
//        removedNodes: NodeList[div],
//        target: body,
//        type: "childList",
//    }
// ]
```

Изменение порядка дочернего элемента, хотя оно может быть выполнено одним способом, будет зарегистрировано как два отдельных изменения, поскольку технически это удаление узла и последующее повторное добавление:

```
// очищение body
document.body.innerHTML = '';

let observer = new MutationObserver(
    (mutationRecords) => console.log(mutationRecords));

// Создание двух начальных дочерних элементов
document.body.appendChild(document.createElement('div'));
document.body.appendChild(document.createElement('span'));

observer.observe(document.body, { childList: true });

// Изменение порядка дочерних элементов
document.body.insertBefore(document.body.lastChild, document.body.firstChild);

// Зарегистрированы два изменения: удаление элемента — по индексу 0,
// добавление элемента — по индексу 1
// [
//    {
//        addedNodes: NodeList[],
//        attributeName: null,
//        attributeNamespace: null,
//        oldValue: null,
//        nextSibling: null,
//        previousSibling: div,
//        removedNodes: NodeList[span],
//        target: body,
//        type: childList,
//    },
//    {
//        addedNodes: NodeList[span],
```

```
//        attributeName: null,
//        attributeNamespace: null,
//        oldValue: null,
//        nextSibling: div,
//        previousSibling: null,
//        removedNodes: NodeList[],
//        target: body,
//        type: "childList",
//    }
// ]
```

Наблюдение за изменениями поддерева

По умолчанию `MutationObserver` находится в области видимости изменений только для одного элемента и его списка дочерних узлов. Эта область действия может быть расширена до целого поддерева DOM, если для свойства `subtree` внутри объекта `MutationObserverInit` задать значение `true`.

Просмотр поддерева для изменений атрибута может быть выполнен следующим образом:

```
// очищение body
document.body.innerHTML = '';

let observer = new MutationObserver(
    (mutationRecords) => console.log(mutationRecords));

// Создание начального элемента
document.body.appendChild(document.createElement('div'));

// Просмотр поддерева <body>
observer.observe(document.body, { attributes: true, subtree: true });

// Изменение поддерева <body>
document.body.firstChild.setAttribute('foo', 'bar');

// Измененеие поддерева было зарегистрировано
// [
//    {
//        addedNodes: NodeList[],
//        attributeName: "foo",
//        attributeNamespace: null,
//        oldValue: null,
//        nextSibling: null,
//        previousSibling: null,
//        removedNodes: NodeList[],
//        target: div,
//        type: "attributes",
//    }
// ]
```

Интересно, что обозначение поддерева узла будет сохраняться, даже когда этот узел перемещается из наблюдаемого дерева. Это означает, что после того как узел поддерева покидает это конкретное поддерево, изменения, которые сейчас технически

находятся за пределами наблюдаемого поддерева, все равно будут регистрироваться как отслеживаемые.

Пример такого поведения:

```
// очищение body
document.body.innerHTML = '';

let observer = new MutationObserver(
    (mutationRecords) => console.log(mutationRecords));

let subtreeRoot = document.createElement('div'),
    subtreeLeaf = document.createElement('span');

// Создание начального поддерева высотой 2
document.body.appendChild(subtreeRoot);
subtreeRoot.appendChild(subtreeLeaf);

// Просмотр поддерева
observer.observe(subtreeRoot, { attributes: true, subtree: true });

// Вынос узла из наблюдаемого поддерева
document.body.insertBefore(subtreeLeaf, subtreeRoot);

subtreeLeaf.setAttribute('foo', 'bar');

// Изменения поддерева все еще будут регистрироваться
// [MutationRecord]
```

Асинхронные обратные вызовы и очередь записи

Спецификация `MutationObserver` предназначена для повышения производительности, и в основе ее конструкции лежит асинхронный обратный вызов и модель очереди записей. Чтобы обеспечить возможность регистрации большого количества изменений без снижения производительности, информация о каждом соответствующем изменении (определяемом экземпляром наблюдателя) собирается в `MutationRecord` и затем помещается в *очередь записи*. Эта очередь уникальна для каждого экземпляра `MutationObserver` и представляет запись по порядку каждого изменения DOM.

Поведение очереди записи

Каждый раз, когда `MutationRecord` добавляется в очередь записей `MutationObserver`, обратный вызов наблюдателя (первоначально предоставляемый конструктору `MutationObserver`) планируется как микрозадача, только если микропроцессорная функция обратного вызова уже не запланирована, например, длина очереди > 0. Это гарантирует, что не произойдет двойной обработки обратного вызова содержимого очереди записи.

Вполне возможно, что к тому времени, когда микрозадача обратного вызова выполнится асинхронно, произойдут еще другие изменения помимо того

единственного, которое первоначально запланировало микрозадачу обратного вызова. Запущенный обратный вызов передается в массив экземпляров `MutationRecord` в том же порядке, как они появляются в очереди записи. Обратный вызов отвечает за полную обработку каждого экземпляра в массиве, поскольку они не сохранятся после выхода из функции. После выполнения обратного вызова ожидается, что все экземпляры `MutationRecord` больше не нужны, поэтому очередь записей очищается, а ее содержимое отбрасывается.

Метод takeRecords()

Можно очистить очередь записей экземпляра `MutationObserver` с помощью метода `takeRecords()`. Он вернет массив экземпляров `MutationRecord`, которые существуют в очереди, и очистит саму очередь:

```
let observer = new MutationObserver(
    (mutationRecords) => console.log(mutationRecords));

observer.observe(document.body, { attributes: true });

document.body.className = 'foo';
document.body.className = 'bar';
document.body.className = 'baz';

console.log(observer.takeRecords());
console.log(observer.takeRecords());

// [MutationRecord, MutationRecord, MutationRecord]
// []
```

Это может быть полезно, когда нужно вызвать `disconnect()`, но с обработкой всех ожидающих экземпляров `MutationRecord` в очереди, которые отбрасываются вызовом `disconnect()`.

Производительность, память и сборка мусора

`MutationEvent`, представленный в спецификации DOM Level 2, определил несколько событий, которые вызываются различными изменениями DOM. При реализации из-за характера событий в браузере эта спецификация приводила к существенным проблемам с производительностью, а в спецификации DOM Level 3 эти события были объявлены устаревшими. `MutationObserver` был представлен, чтобы заменить их более прагматичным и производительным дизайном.

Передача выполнения обратных вызовов изменений в микрозадачу гарантирует, что синхронный характер событий и беспорядок, сопровождающий их, исключен. Реализация очереди записей для спецификации `MutationObserver` гарантирует, что даже преобладание событий изменений не приведет к чрезмерной загрузке браузера.

Тем не менее использование `MutationObserver` по-прежнему повлечет за собой *некоторые* накладные расходы, и важно понимать, где они проявятся.

Ссылки MutationObserver

Ссылочные отношения между `MutationObserver` и узлом (или узлами), который он отслеживает, асимметричны. `MutationObserver` имеет слабую ссылку на наблюдаемый целевой узел. Поскольку эта ссылка слабая, она не помешает целевому узлу быть уничтоженным при сборке мусора.

Однако у узла есть сильная ссылка на его `MutationObserver`. Если целевой узел удаляется из DOM, а затем уничтожается сборщиком мусора, связанный с ним `MutationObserver` также уничтожается.

Ссылки MutationRecord

Каждый экземпляр `MutationRecord` в очереди записей будет содержать хотя бы одну ссылку на существующий узел DOM; в случае изменения `childList` он будет содержать множество ссылок. Поведение по умолчанию очереди записи и обработки обратного вызова состоит в том, чтобы поочередно очищать очередь, обрабатывать каждый `MutationRecord` и позволять им выходить из области видимости и уничтожаться сборкой мусора.

Может возникнуть ситуация, когда полезно сохранить полную запись изменений от данного наблюдателя. Сохранение каждого экземпляра `MutationRecord` также сохранит содержащиеся в нем ссылки на узлы, тем самым предотвращая сборку мусора для узлов. Если требуется сборка мусора на узле, извлекайте минимально необходимую информацию из каждого `MutationRecord` в новый объект и отбрасывайте `MutationRecord`.

ИТОГИ

DOM — это API, который не зависит от языка и предназначен для доступа к HTML- и XML-документам и выполнения различных операций с ними и их содержимым. DOM Level 1 представляет HTML- и XML-документы в виде иерархии узлов, манипулируя которыми с помощью JavaScript можно изменять вид и структуру документов.

DOM определяет несколько типов узлов.

- В основе всех узлов лежит тип `Node`, который является абстрактным представлением отдельной части документа; от него наследуются все остальные типы узлов.
- Тип `Document` представляет весь документ и располагается в корне иерархии. В JavaScript к этому типу относится объект `document`, с помощью которого можно запрашивать и получать узлы разными способами.
- Все элементы HTML или XML в документе имеют тип `Element`, который обеспечивает средства для манипулирования их содержимым и атрибутами.
- Существуют типы узлов для текстового контента, комментариев, типов документов, CDATA-разделов и фрагментов документов.

В большинстве случаев DOM работает без неожиданностей, но при использовании элементов `<script>` и `<style>` часто возникают осложнения. Поскольку эти элементы содержат сценарии и стили, они часто обрабатываются в браузерах не так, как другие элементы.

Крайне важно понимать, как DOM влияет на общее быстродействие кода. Манипуляции с DOM-элементами входят в число самых ресурсоемких JavaScript-операций, что особенно заметно в случае объектов `NodeList`, которые обновляются при каждом обращении к ним. По этой причине количество операций с DOM желательно свести к минимуму.

`MutationObserver` был введен для замены менее производительного `MutationEvent`. Он позволяет осуществлять эффективный и точный мониторинг изменений в DOM с помощью относительно простого API.

15

Расширения DOM

- Selectors
- Использование расширений DOM HTML5

Загрузки с сайта wrox.com для этой главы

Обратите внимание, что все примеры кода для этой главы доступны на веб-сайте книги по ссылке www.wrox.com/go/projavascript4e на вкладке Downloads.

Хотя DOM API определен довольно четко, его функциональность часто расширяют с помощью стандартизированных и фирменных компонентов. До 2008 г. почти все DOM-расширения в браузерах были фирменными. Некоторые из них, ставшие стандартами де-факто, консорциум W3C описал в формальных спецификациях.

Двумя основными стандартами DOM-расширений являются Selectors и HTML5. Они были разработаны для стандартизации актуальных API и методик разработки в соответствии с интересами и требованиями сообщества. Дополнительные DOM-свойства определены также в более компактной спецификации, которая называется Element Traversal. Хотя Selectors и особенно HTML5 охватывают большое количество DOM-расширений, по-прежнему используются фирменные расширения, которые также рассматриваются в этой главе.

Содержимое этой главы поддерживается всеми основными браузерами, то есть всеми релизами поставщиков, которые имеют значимый веб-трафик, если не указано иное.

SELECTORS API

Одной из наиболее востребованных возможностей JavaScript-библиотек является получение DOM-элементов с помощью CSS-селекторов. Например, библиотека

jQuery (www.jquery.com) полностью построена на этом подходе, который заменяет методы `getElementById()` и `getElementsByTagName()`.

Консорциум W3C разработал спецификацию Selectors API (www.w3.org/TR/selectors-api), чтобы стандартизировать встроенную поддержку CSS-запросов в браузерах. Ранее для реализации CSS-запросов в JavaScript-библиотеках приходилось разрабатывать синтаксические анализаторы CSS-кода и использовать существующие DOM-методы для перемещения по документу и идентификации запрошенных узлов. Несмотря на все усилия разработчиков по оптимизации обработки таких запросов на JavaScript, результаты оставляли желать лучшего. Созданный для решения проблемы встроенный API позволил осуществлять синтаксический анализ запросов и навигацию по дереву на уровне браузера на компилируемом языке, что существенно повысило быстродействие кода.

В основе Selectors API Level 1 лежат методы `querySelector()` и `querySelectorAll()`, доступные для типов `Document` и `Element`.

Спецификация Selectors API Level 2 (https://www.w3.org/TR/selectors-api2/) представляет дополнительные методы: `match()`, `find()` и `findAll()` для типа `Element`, хотя в настоящее время ни один браузер не имеет возможности поддерживать `find()` или `findAll()` и не заявил о намерении это изменить.

Метод querySelector()

Метод `querySelector()` принимает CSS-запрос и возвращает первый соответствующий ему элемент или значение `null`, если таких элементов нет, например:

```
// Получение элемента body
let body = document.querySelector("body");

// Получение элемента с идентификатором "myDiv"
let myDiv = document.querySelector("#myDiv");

// Получение первого элемента класса "selected"
let selected = document.querySelector(".selected");

// Получение первого изображения класса "button"
let img = document.body.querySelector("img.button");
```

При вызове метода `querySelector()` для типа `Document` сопоставление с шаблоном начинается с элемента документа; при вызове для типа `Element` поиск совпадения начинается с элемента и выполняется в нисходящем порядке только среди его потомков.

CSS-запрос может быть сколь угодно сложным. Если он имеет неправильный синтаксис или содержит неподдерживаемый селектор, возникает ошибка.

Метод querySelectorAll()

Метод `querySelectorAll()` принимает CSS-запрос и возвращает все соответствующие ему узлы в статическом экземпляре `NodeList`.

Если точнее, возвращается объект NodeList со всеми свойствами и методами, но на самом деле он реализован как «снимок» элементов на текущий момент времени, а не как динамический запрос документа. Благодаря этому он работает гораздо быстрее, чем обычные объекты NodeList.

Любой вызов метода querySelectorAll() с допустимым CSS-запросом возвращает объект NodeList независимо от количества соответствующих запросу элементов; если соответствий нет, объект NodeList будет пустым.

Как и querySelector(), метод querySelectorAll() доступен для типов Document, DocumentFragment и Element. Вот несколько примеров:

```
// Получение всех элементов <em> в <div> (аналог getElementsByTagName("em"))
let ems = document.getElementById("myDiv").querySelectorAll("em");

// Получение всех элементов класса "selected"
let selecteds = document.querySelectorAll(".selected");

// Получение всех элементов <strong> в элементах <p>
let strongs = document.querySelectorAll("p strong");
```

Возвращенный объект NodeList можно перебирать, извлекая из него отдельные элементы с помощью итерационных хуков, метода item() или скобочной нотации:

```
let strongElements = document.querySelectorAll("p strong");

// Все три цикла будут иметь одинаковый эффект:

for (let strong of strongElements) {
    strong.className = "important";
}

for (let i = 0; i < strongElements.length; ++i) {
    strongElements.item(i).className = "important";
}

for (let i = 0; i < strongElements.length; ++i) {
    strongElements [i].className = "important";
}
```

Если CSS-селектор не поддерживается браузером или имеет неправильный синтаксис, вызов метода querySelectorAll() завершается ошибкой.

Метод matches()

Метод matches() ранее упоминался в проекте спецификации как matchesSelector(). Он принимает CSS-селектор и возвращает true, если элемент соответствует селектору, и false в противном случае, например:

```
if (document.body.matches("body.page1")) {
    // true
}
```

Так можно легко проверить, будет ли возвращен элемент методом `querySelector()` или `querySelectorAll()`, если у вас уже есть ссылка на элемент.

Все основные браузеры поддерживают некоторую форму `matches()`. Edge, Chrome, Firefox, Safari и Opera полностью поддерживают его; IE 9–11 и второстепенные мобильные браузеры поддерживают его с префиксами поставщиков.

ELEMENT TRAVERSAL

В отличие от других браузеров, Internet Explorer до версии 9 не создает текстовые узлы для свободного пространства между элементами, из-за чего возникают несоответствия при использовании таких свойств, как `childNodes` и `firstChild`. Для преодоления этих различий с соблюдением требований модели DOM была определена спецификация Element Traversal API (www.w3.org/TR/ElementTraversal/).

Этот API добавляет ко всем DOM-элементам пять новых свойств:

- `childElementCount` — возвращает количество дочерних элементов (исключая текстовые узлы и комментарии);
- `firstElementChild` — указывает на первый дочерний элемент (версия свойства `firstChild` только для элементов);
- `lastElementChild` — указывает на последний дочерний элемент (версия свойства `lastChild` только для элементов);
- `previousElementSibling` — указывает на предыдущий элемент того же уровня (версия свойства `previousSibling` только для элементов);
- `nextElementSibling` — указывает на следующий элемент того же уровня (версия свойства `nextSibling` только для элементов).

Эти свойства упрощают перебор DOM-элементов, позволяя не беспокоиться о текстовых узлах для свободного пространства.

Например, традиционный кроссбраузерный способ перебора всех дочерних элементов конкретного элемента выглядит так:

```
let parentElement = document.getElementById('parent');
let currentChildNode = parentElement.firstChild;

// При отсутствии дочерних элементов firstChild возвращает null,
// и цикл пропускается
while (currentChildNode) {
    if (currentChildNode.nodeType === 1) {
        // Если это ELEMENT_NODE, вся необходимая работа делается здесь
        processChild(currentChildNode);
    }
    if (currentChildNode === parentElement.lastChild) {
        break;
    }
    currentChildNode = currentChildNode.nextSibling;
}
```

Использование свойств из спецификации Element Traversal позволяет упростить код:

```
let parentElement = document.getElementById('parent');
let currentChildElement = parentElement.firstElementChild;

// При отсутствии дочерних элементов firstChild возвращает null,
// и цикл пропускается
while (currentChildElement) {
    // Вы уже знаете, что это ELEMENT_NODE, и вся необходимая работа делается здесь
    processChild(currentChildElement);
    if (currentChildElement === parentElement.lastElementChild) {
        break;
    }
    currentChildElement = currentChildElement.nextElementSibling;
}
```

Element Traversal API реализован в Internet Explorer 9 и всех современных браузерах.

HTML5

HTML5 представляет радикальный отход от традиций HTML. Никакие предыдущие спецификации HTML не описывали JavaScript-интерфейсы, а определяли исключительно разметку, при этом связи между HTML и JavaScript регламентировала спецификация DOM.

Что касается спецификации HTML5, то она содержит множество JavaScript API, разработанных для использования с новыми элементами разметки. Некоторые из этих API перекрываются с DOM и определяют DOM-расширения, подлежащие реализации в браузерах.

> **ПРИМЕЧАНИЕ** Спецификация HTML5 очень широка, поэтому в данном разделе рассматриваются только те ее части, которые относятся ко всем DOM-узлам. Другие части HTML5 мы обсудим позже.

Новые средства работы с классами

За время существования HTML4 веб-разработчики стали чаще использовать атрибут `class` для указания стилистической и семантической информации об элементах. Многие JS-сценарии включают код динамического изменения CSS-классов и поиска элементов, относящихся к конкретному классу. HTML5 поддерживает ряд новых средств, которые упрощают работу с классами.

Метод getElementsByClassName()

Одним из наиболее популярных новшеств в HTML5 стал метод `getElementsByClassName()`, который доступен для объекта `document` и всех HTML-элементов.

Благодаря встроенной реализации он значительно превосходит по быстродействию свои аналоги из JavaScript-библиотек, основанные на использовании DOM.

Метод `getElementsByClassName()` принимает строку с одним или несколькими именами классов и возвращает объект `NodeList` с элементами, к которым применены все эти классы. Порядок следования классов в строке не имеет значения, например:

```
// Получение всех элементов, относящихся к классам "username
// " и "current", без учета порядка следования классов
let allCurrentUsernames =
    document.getElementsByClassName("username current");

// Получение всех элементов класса "selected" в поддереве myDiv
var selected =
    document.getElementById("myDiv").getElementsByClassName("selected");
```

Метод `getElementsByClassName()` возвращает только те элементы, которые находятся в поддереве опорного элемента. Если он вызывается для объекта `document`, возвращаются все элементы указанных классов в документе.

С помощью этого метода можно подключать события к элементам, распознавая их по классу, а не по идентификатору или по имени тега. Помните, что он возвращает объект `NodeList`, а значит, ему присущи те же проблемы с быстродействием, что и методу `getElementsByTagName()` и другим DOM-методам, которые возвращают `NodeList`.

Метод `getElementsByClassName()` реализован в Internet Explorer 9 и во всех современных браузерах.

Свойство classList

Свойство `className` используется для добавления, удаления и замены имен классов. Оно содержит строку, которую нужно задавать целиком при каждом изменении, даже самом малом. Рассмотрим пример:

```
<div class="bd user disabled">...</div>
```

Этому элементу `<div>` назначены три класса. Чтобы удалить один из них, нужно разделить атрибут `class` на отдельные классы, убрать нежелательный класс, а затем составить новую строку из оставшихся классов, например:

```
// Удаление класса "user"

// Сначала получаем список имен классов
let classNames = div.className.split(/\s+/);

// Ищем имя удаememого класса
let idx = classNames.indexOf(targetClass);

// Удаляем найденный класс
if (idx > -1) {
    classNames.splice(i,1);
```

```
}

// Возвращаем имя класса
div.className = classNames.join(" ");
```

Весь этот код необходим для удаления класса `"user"` из атрибута `class` элемента `<div>`. Подобный алгоритм нужно использовать и для замены или поиска классов. При добавлении классов путем конкатенации их имен нужно следить за тем, чтобы они не повторялись. Многие JavaScript-библиотеки содержат методы, помогающие решать эти задачи.

Чтобы упростить работу с именами классов и сделать ее более безопасной, в HTML5 ко всем элементам добавлено свойство `classList`, которое является экземпляром новой коллекции `DOMTokenList`. Как и другие DOM-коллекции, `DOMTokenList` имеет свойство `length`, содержащее количество элементов в коллекции, и поддерживает доступ к отдельным элементам с помощью метода `item()` или скобочной нотации. Кроме того, у нее есть несколько дополнительных методов:

- `add(`*значение*`)` — добавляет указанное строковое значение в список (если значение уже существует, оно не добавляется);
- `contains(`*значение*`)` — указывает, есть ли в списке указанное значение (возвращает `true`, если есть, и `false` в противном случае);
- `remove(`*значение*`)` — удаляет указанное строковое значение из списка;
- `toggle(`*значение*`)` — удаляет указанное значение, если оно уже есть в списке, и добавляет значение, если оно отсутствует.

Таким образом, можно заменить весь код из предыдущего примера одной строкой:

```
div.classList.remove("user");
```

Этот код гарантирует, что изменение не повлияет на остальные имена классов. Другие методы также значительно упрощают базовые операции с классами, например:

```
// Удаление класса "disabled"
div.classList.remove("disabled");

// Добавление класса "current"
div.classList.add("current");

// Переключение класса "user"
div.classList.toggle("user");

// Идентификация классов элемента
if (div.classList.contains("bd") && !div.classList.contains("disabled")) {
    // Какие-то действия
}

// Перебор имен классов
for (let class of div.classList){
    doStuff(class);
}
```

Свойство `classList` делает ненужным свойство `className`, если только вы не собираетесь полностью удалить или перезаписать атрибут `class` элемента. Свойство `classList` частично реализовано Internet Explorer 10+ и полностью во всех основных современных браузерах.

Управление фокусом

В HTML5 добавлены средства, помогающие управлять выделением DOM-элементов. Прежде всего, это свойство `document.activeElement`, которое всегда содержит указатель на выделенный DOM-элемент. Элемент может быть выделен автоматически во время загрузки страницы, в результате действий пользователя (как правило, при нажатии клавиши табуляции) или программно методом `focus()`, например:

```
let button = document.getElementById("myButton");
button.focus();
console.log(document.activeElement === button);    // true
```

По умолчанию при загрузке документа в первый раз свойству `document.activeElement` присваивается значение `document.body`, но до полной загрузки документа оно имеет значение `null`.

Второе новшество — метод `document.hasFocus()`, который возвращает логическое значение, указывающее, содержит ли документ выделенный элемент:

```
let button = document.getElementById("myButton");
button.focus();
console.log(document.hasFocus());        // true
```

С помощью этого метода можно узнать, взаимодействует ли пользователь со страницей.

Возможности идентифицировать выделенный элемент и узнать, есть ли он в документе, крайне важны для эффективного доступа к веб-приложению. Правильное управление фокусом делает работу с приложением более удобной, и прямое определение выделенного элемента намного предпочтительнее в этом плане, чем прежние бессистемные подходы.

Изменения типа HTMLDocument

В HTML5 также расширен тип `HTMLDocument`. Как и другие DOM-расширения, определенные в HTML5, его изменения основаны на фирменных расширениях, реализованных во многих браузерах. Таким образом, хотя стандартизировать расширения стали сравнительно недавно, в ряде браузеров они доступны не первый день.

Свойство readyState

В Internet Explorer 4 впервые было представлено свойство `readyState` объекта `document`. Позднее оно было добавлено в другие браузеры и в итоге вошло в спецификацию HTML5. Оно может иметь следующие значения:

- `loading` — документ загружается;
- `complete` — документ полностью загружен.

Свойство `document.readyState` полезно как индикатор загрузки документа. Пока оно широко не применялось, вместо него задействовали обработчик события `onload`, в котором устанавливали флаг, указывающий, что документ загружен. Используется это свойство следующим образом:

```
if (document.readyState == "complete") {
    // Какие-то действия
}
```

Режим совместимости

С выпуском Internet Explorer 6 и появлением возможности визуализировать документ в стандартном режиме или режиме совместимости потребовалось определять, в каком режиме браузер отображает страницу. В Internet Explorer исключительно для этого к объекту `document` было добавлено свойство `compatMode`. В стандартном режиме оно имеет значение `"CSS1Compat"`, а в режиме совместимости — `"BackCompat"`:

```
if (document.compatMode == "CSS1Compat") {
    console.log("Standards mode");   // "Стандартный режим"
} else {
    console.log("Quirks mode");      // "Режим совместимости"
}
```

Это свойство было стандартизировано в HTML5.

Свойство head

В HTML5 представлено свойство `document.head`, которое указывает на элемент `<head>` документа, логически дополняя свойство `document.body`. Его можно использовать как альтернативу старому способу получения ссылки на элемент `<head>`:

```
let head = document.head;
```

Свойства кодировки

HTML5 описывает несколько свойств для работы с кодировкой документа. Свойство `characterSet` указывает фактическую кодировку документа и позволяет задать новую. По умолчанию оно имеет значение `"UTF-16"`, которое можно изменить с помощью элементов `<meta>`, заголовков ответа или непосредственно, например:

```
console.log(document.characterSet);    // "UTF-16"
document.characterSet = "UTF-8";
```

Пользовательские атрибуты данных

В HTML5 можно использовать нестандартные атрибуты с префиксом `data-` для добавления сведений, не влияющих на визуализацию или семантику элементов.

Имена этих атрибутов могут быть любыми, но должны начинаться с префикса `data-`, например:

```
<div id="myDiv" data-appId="12345" data-myname="Nicholas"></div>
```

Такие атрибуты доступны через свойство `dataset` элемента. Оно содержит экземпляр типа `DOMStringMap`, в котором хранятся пары имен и значений. Каждый атрибут формата `data-name` представляется одноименным свойством без префикса `data-` (например, атрибуту `data-myname` соответствует свойство `myname`):

```
// Методы в этом примере используются исключительно для демонстрации

let div = document.getElementById("myDiv");

// Получение значений
let appId = div.dataset.appId;
let myName = div.dataset.myname;

// Задание значений
div.dataset.appId = 23456;
div.dataset.myname = "Michael";

// Существует ли значение "myname"?
if (div.dataset.myname) {
    console.log("Hello, " + div.dataset.myname);
}
```

Пользовательские атрибуты данных могут связать с элементом какую-то невизуальную информацию. Их часто используют для отслеживания ссылок, а также для идентификации частей страницы в гибридных веб-приложениях. Они также широко используются в многочисленных одностраничных фреймворках приложений.

Вставка разметки

Хотя DOM обеспечивает детализированный контроль над узлами в документе, вставлять новые элементы HTML-разметки с помощью DOM неудобно. Вместо того чтобы создавать последовательность DOM-узлов и связывать их в правильном порядке, гораздо проще (и быстрее) вставить в документ строку HTML-кода. Для этого в HTML5 определены описываемые далее DOM-расширения.

Свойство innerHTML

В режиме чтения свойство `innerHTML` возвращает HTML-код, представляющий все дочерние узлы элемента, в том числе элементы, комментарии и текстовые узлы. При записи свойства `innerHTML` все дочерние узлы элемента заменяются новым DOM-поддеревом. Рассмотрим следующий HTML-код:

```
<div id="content">
    <p>This is a <strong>paragraph</strong> with a list following it.</p>
    <ul>
        <li>Item 1</li>
```

```
        <li>Item 2</li>
        <li>Item 3</li>
    </ul>
</div>
```

Для элемента `<div>` в этом примере свойство `innerHTML` возвращает следующую строку:

```
<p>This is a <strong>paragraph</strong> with a list following it.</p>
<ul>
    <li>Item 1</li>
    <li>Item 2</li>
    <li>Item 3</li>
</ul>
```

Точный текст, возвращаемый свойством `innerHTML`, зависит от браузера. Internet Explorer и Opera обычно преобразуют все теги в верхний регистр, а Safari, Chrome и Firefox возвращают HTML без изменений, с пробелами и отступами. Не полагайтесь на то, что значение свойства `innerHTML` будет одинаковым во всех браузерах.

В режиме записи свойство `innerHTML` составляет из назначенной ему строки DOM-поддерево и заменяет им все дочерние узлы элемента. Поскольку строка интерпретируется как HTML-код, все теги в ней преобразуются в элементы стандартным для браузера способом (который также зависит от браузера). Если строка не содержит HTML-тегов, в свойстве сохраняется обычный текст:

```
div.innerHTML = "Hello world!";
```

Если свойству `innerHTML` назначается строка с HTML-тегами, выполняется синтаксический анализ, например:

```
div.innerHTML = "Hello & welcome, <b>\"reader\"!</b>";
```

Эта операция дает следующий результат:

```
<div id="content">Hello &amp; welcome, <b>&quot;reader&quot;!</b></div>
```

После задания свойства `innerHTML` новые узлы можно использовать так же, как и любые другие узлы в документе.

> **ПРИМЕЧАНИЕ** При задании свойства `innerHTML` браузер преобразует указанную HTML-строку в соответствующее DOM-поддерево. При последующем чтении того же свойства `innerHTML` обычно возвращается другая строка, потому что она является результатом сериализации DOM-поддерева, созданного из первоначальной строки.

Использование innerHTML в устаревшем Internet Explorer

У свойства `innerHTML` есть некоторые ограничения. Так, все современные браузеры не поддерживают выполнение кода в элементах `<script>`, вставленных через свойство

innerHTML. Это возможно лишь в Internet Explorer 8 и более ранних версий, если указан атрибут defer и элементу <script> предшествует так называемый *визуальный элемент* (scoped element). <script> и <style>, а также комментарии к визуальным элементам не относятся, поскольку они не отображаются на странице. Internet Explorer игнорирует такие элементы в начале строк, вставленных с помощью свойства innerHTML, то есть следующий код не сработает:

```
// Не сработает
div.innerHTML = "<script defer>alert('hi');<\/script>";  // не работает
```

Здесь свойству innerHTML присваивается значение, которое начинается с невизуального элемента, поэтому вся строка становится пустой. Чтобы этот сценарий работал должным образом, нужно добавить перед ним визуальный элемент, например текстовый узел или элемент без закрывающего тега, такой как <input>. Все следующие варианты работают правильно:

```
// Все это сработает
div.innerHTML = "_<script defer>console.log('hi');<\/script>";
div.innerHTML = "<div> </div><script defer>console.log('hi');<\/script>";
div.innerHTML =
    "<input type=\"hidden\"><script defer>console.log('hi');<\/script>";
```

Первая строка добавляет перед элементом <script> текстовый узел, который, возможно, позднее потребуется удалить, чтобы не искажать вид страницы. Во втором примере с этой же целью используется элемент с неразрывным пробелом. Пустого элемента недостаточно — он должен иметь некоторое содержимое, чтобы был создан текстовый узел. Этот узел также может потребоваться удалить для восстановления правильной разметки. Наконец, в третьем примере используется скрытое поле <input>. Поскольку оно не влияет на разметку, обычно это оптимальный выбор.

Большая часть браузеров поддерживает вставку элементов <style> через свойство innerHTML:

```
div.innerHTML =
    "<style type=\"text/css\">body {background-color: red; }</style>";
```

В Internet Explorer 8 и более ранних версий элементу <style> должен предшествовать визуальный элемент, например:

```
div.innerHTML =
    "_<style type=\"text/css\">body {background-color: red; }</style>";
div.removeChild(div.firstChild);
```

ПРИМЕЧАНИЕ Firefox предъявляет более строгие требования к свойству innerHTML в XHTML с типом контента application/xhtml+xml. Если попытаться назначить ему XHTML-код с нарушениями формата, операция просто игнорируется, а ошибка не возвращается.

Свойство outerHTML

При чтении свойства `outerHTML` оно возвращает HTML-код элемента, которому принадлежит, и всех его дочерних узлов. При записи оно заменяет узел, которому принадлежит, DOM-поддеревом, соответствующим полученной HTML-строке. Рассмотрим следующий HTML-код:

```
<div id="content">
    <p>This is a <strong>paragraph</strong> with a list following it.</p>
    <ul>
        <li>Item 1</li>
        <li>Item 2</li>
        <li>Item 3</li>
    </ul>
</div>
```

Если вызвать свойство `outerHTML` этого элемента `<div>`, будет возвращен точно такой же код, в том числе сам элемент `<div>`. Имейте в виду, что из-за особенностей синтаксического анализа и интерпретации HTML-кода в разных браузерах результаты могут различаться (наблюдаются те же различия, что и при работе со свойством `innerHTML`).

Задается свойство `outerHTML` следующим образом:

```
div.outerHTML = "<p>This is a paragraph.</p>";
```

Этот код эквивалентен следующему DOM-коду:

```
let p = document.createElement("p");
p.appendChild(document.createTextNode("This is a paragraph."));
div.parentNode.replaceChild(p, div);
```

В этом примере новый элемент `<p>` заменяет исходный элемент `<div>` в DOM-дереве.

Методы insertAdjacentHTML() и insertAdjacentText()

Вставлять разметку можно с помощью методов `insertAdjacentHTML()` и `insertAdjacentText()`, которые также появились в Internet Explorer. Они принимают два аргумента: позицию, в которой нужно вставить разметку, и HTML-код или текст. Первым аргументом может быть одно из следующих значений:

- `"beforebegin"` — вставляет HTML-код непосредственно перед элементом на том же уровне;
- `"afterbegin"` — вставляет HTML-код внутри элемента как новый дочерний узел или как несколько дочерних узлов перед первым уже существующим;
- `"beforeend"` — вставляет HTML-код внутри элемента как новый дочерний узел или как несколько дочерних узлов вслед за последним уже существующим;
- `"afterend"` — вставляет HTML-код сразу после элемента на том же уровне.

Все эти значения нечувствительны к регистру. Второй аргумент метода обрабатывается как HTML-строка (подобно свойствам `innerHTML/outerHTML`) или как обычный

текст (то же самое, что и innerText/outerText). Если интерпретировать его не удается, возникает ошибка. Вот некоторые примеры:

```
// Вставка узла перед элементом на том же уровне
element.insertAdjacentHTML("beforebegin", "<p>Hello world!</p>");
element.insertAdjacentText("beforebegin", "Hello world!");

// Вставка узла в качестве первого дочернего
element.insertAdjacentHTML("afterbegin", "<p>Hello world!</p>");
element.insertAdjacentText("afterbegin", "Hello world!");

// Вставка узла в качестве последнего дочернего
element.insertAdjacentHTML("beforeend", "<p>Hello world!</p>");
element.insertAdjacentText("beforeend", "Hello world!");

// Вставка узла после элемента на том же уровне
element.insertAdjacentHTML("afterend", "<p>Hello world!</p>");
insertAdjacentText("afterend", "Hello world!");
```

Проблемы с памятью и быстродействием

Замена дочерних узлов с помощью методов, описываемых в этом разделе, может вызывать в браузерах проблемы с памятью, особенно в Internet Explorer. Проблемы возникают при удалении элементов поддерева, которым назначены обработчики событий или другие JavaScript-объекты. Если у элемента есть обработчик событий (или свойство, содержащее JavaScript-объект) и этот элемент удаляется из дерева документа с помощью одного из описанных свойств, связь между элементом и обработчиком событий остается в памяти. Если это повторяется много раз, потребление памяти существенно увеличивается. При использовании свойств innerHTML, outerHTML и метода insertAdjacentHTML() рекомендуется вручную удалять все обработчики событий и свойства-объекты у элементов, которые будут удалены.

Тем не менее обычно выгодно использовать эти свойства, особенно innerHTML. Если нужно вставить много нового HTML-кода, эффективнее сделать это с помощью innerHTML, а не многочисленных DOM-операций, создающих узлы и связи между ними. Это объясняется тем, что при назначении строки свойству innerHTML (или outerHTML) всегда создается HTML-анализатор. Он реализуется на уровне браузера (часто на C++) и работает гораздо быстрее, чем JavaScript. И все же создание и уничтожение HTML-анализатора требует ресурсов, так что увлекаться свойствами innerHTML и outerHTML не следует. Например, следующий код создает с помощью свойства innerHTML несколько элементов списка:

```
for (let value of values){
    ul.innerHTML += '<li>${value}</li>';    // не делайте так!
}
```

Этот код неэффективен, потому что свойство innerHTML задается при каждой итерации цикла. Хуже того: оно еще считывается при каждой итерации, что удваивает расходы. Лучше создать строку отдельно и назначить ее свойству innerHTML только один раз в конце:

```
let itemsHtml = "";
for (let value of values) {
    itemsHtml += '<li>${value}</li>';
}
ul.innerHTML = itemsHtml;
```

Этот код более эффективен, потому что доступ к свойству `innerHTML` выполняется один раз. Конечно, при необходимости можно сжать это до одной строки:

```
ul.innerHTML = values.map(value => '<li>${value}</li>').join('');
```

Особенности межсайтового скриптинга

Хотя `innerHTML` не выполняет созданные им теги сценариев, он все же обеспечивает чрезвычайно широкую поверхность атаки для злоумышленников, стремящихся скомпрометировать веб-страницу, поскольку он с готовностью создает элементы и исполняемые атрибуты, такие как `onclick`.

Везде, где выполняется интерполяция предоставленной пользователем информации на страницу, ее почти всегда нежелательно применять, используя `innerHTML`. Головные боли по предотвращению XSS-уязвимостей намного перевешивают любые преимущества и удобства, полученные от использования `innerHTML`. Разделите интерполированные данные и не стесняйтесь использовать библиотеки, которые экранируют интерполированные данные, прежде чем вставлять их на страницу.

Метод scrollIntoView()

Спецификация DOM не определяет средства прокрутки областей страницы. Чтобы восполнить этот пробел, разработчики браузеров реализовали несколько разных методов прокрутки, из которых в HTML5 был добавлен только `scrollIntoView()`.

Метод `scrollIntoView()` доступен для всех HTML-элементов; он прокручивает окно браузера или другой контейнер так, чтобы элемент был виден в области просмотра.

- Если аргумент метода имеет значение `true`, он указывает `alignToTop`: окно прокручивается так, что верхняя часть элемента находится в верхней части области просмотра.
- Если аргумент метода имеет значение `false`, он указывает `alignToTop`: окно прокручивается так, что нижняя часть элемента находится в верхней части области просмотра.
- Если в качестве аргумента используется объект, пользователь может предоставить значения для свойства `behavior`, которое указывает, как должна происходить прокрутка: автоматически, мгновенно или плавно (ограниченная поддержка вне Firefox), а свойство `block` такое же, как `alignToTop`.
- Если аргумент не указан, элемент прокручивается так, что он полностью виден в области просмотра, но может не выравниваться сверху. Например:

```
// Проверка видимости элемента
document.forms[0].scrollIntoView();

// То же поведение
document.forms[0].scrollIntoView(true);
document.forms[0].scrollIntoView({block: true});

// Попытка плавной прокрутки элемента в область просмотра:
document.forms[0].scrollIntoView({behavior: 'smooth', block: true});
```

Этот метод наиболее полезен, если нужно привлечь внимание пользователя к каким-либо изменениям на странице. Отметим, что при установке фокуса для элемента браузер также при необходимости прокручивает страницу, чтобы отобразить выделенный элемент в области просмотра.

ФИРМЕННЫЕ РАСШИРЕНИЯ

Хотя производители браузеров понимают важность соблюдения стандартов, все они добавляли в DOM собственные расширения, если считали, что в ней чего-то не хватает. На первый взгляд это может показаться плохой идеей, однако фирменные расширения предоставили сообществу веб-разработчиков много возможностей, которые позднее были формализованы в HTML5 и других стандартах.

В то же время многие фирменные DOM-расширения по-прежнему не отражены в стандартах. Вполне возможно, что какие-то из них будут стандартизированы, но пока они реализованы только в некоторых браузерах.

Свойство children

Различия в интерпретации свободного пространства в текстовых узлах в Internet Explorer до версии 9 и других браузерах стали поводом для создания свойства `children`. Это коллекция `HTMLCollection`, содержащая только те дочерние узлы элемента, которые сами являются элементами. Если все дочерние узлы элемента представляют собой элементы, содержимое свойств `children` и `childNodes` не различается. Доступ к свойству `children` осуществляется так:

```
let childCount = element.children.length;
let firstChild = element.children[0];
```

Метод contains()

Часто требуется определить, является ли один узел потомком другого. Метод `contains()` позволяет выяснить это без обхода дерева DOM-документа. Он вызывается для узла, с которого нужно начать поиск, и принимает в качестве аргумента предполагаемый узел-потомок. Если переданный в метод узел входит в число потомков опорного узла, метод возвращает `true`, иначе — `false`, например:

```
console.log(document.documentElement.contains(document.body));    // true
```

Этот код проверяет, является ли элемент `<body>` потомком элемента `<html>`, что действительно имеет место во всех правильных HTML-страницах.

Определить отношения между узлами можно также с помощью метода `compareDocumentPosition()` из DOM Level 3. Сведения об отношениях узлов возвращаются в виде битовой маски, отдельные биты которой описаны в таблице.

МАСКА	ОТНОШЕНИЯ МЕЖДУ УЗЛАМИ
0x1	Узлы не связаны (переданный в метод узел отсутствует в документе)
0x2	Отношение «предшествует» (переданный в метод узел располагается в DOM-дереве до опорного узла)
0x4	Отношение «следует» (переданный в метод узел располагается в DOM-дереве после опорного узла)
0x8	Отношение «содержит» (переданный в метод узел является предком опорного узла)
0x10	Отношение «содержится» (переданный в метод узел является потомком опорного узла)

С помощью маски 16 можно имитировать метод `contains()`. Для этого к маске и результату вызова метода `compareDocumentPosition()` нужно применить поразрядный оператор И:

```
let result =
    document.documentElement.compareDocumentPosition(document.body);
console.log(!!(result & 0x10));
```

Переменная `result` в этом примере получает значение 20 или 0x14 (0x4 за отношение «следует» и 0x10 за «содержится»). Применение поразрядного оператора И к битовой маске 0x10 и результату возвращает ненулевое число, которое затем преобразуется в логическое значение `true` с помощью двух операторов НЕ.

IE9 + и все современные браузеры поддерживают и `contains`, и `CompareDocumentPosition`.

Вставка разметки

Кроме свойств `innerHTML` и `outerHTML`, добавленных в HTML5 из Internet Explorer, для вставки разметки можно использовать свойства `innerText` и `outerText`, которые не вошли в HTML5.

Свойство innerText

Свойство `innerText` предназначено для работы со всем текстовым контентом элемента независимо от того, насколько глубоко в поддереве находится этот текст. При чтении свойства `innerText` значения всех текстовых узлов в поддереве объединяются в строку с использованием поиска в глубину. При записи свойства `innerText`

все дочерние узлы элемента заменяются текстовым узлом, содержащим указанное значение. Рассмотрим следующий HTML-код:

```
<div id="content">
    <p>This is a <strong>paragraph</strong> with a list following it.</p>
    <ul>
        <li>Item 1</li>
        <li>Item 2</li>
        <li>Item 3</li>
    </ul>
</div>
```

В этом примере свойство `innerText` возвратило бы для элемента `<div>` следующую строку:

```
This is a paragraph with a list following it.
Item 1
Item 2
Item 3
```

Имейте в виду, что браузеры обрабатывают свободное пространство по-разному, так что форматирование может не включать отступы, присутствующие в исходном HTML-коде.

Задать содержимое элемента `<div>` с помощью свойства `innerText` можно следующим образом:

```
div.innerText = "Hello world!";
```

При выполнении этой строки HTML-код страницы изменится:

```
<div id="content">Hello world!</div>
```

Задание свойства `innerText` удаляет все существующие дочерние узлы, полностью изменяя DOM-поддерево. Отметим также, что при этом кодируются все знаки HTML-синтаксиса в тексте («меньше», «больше», кавычки и амперсанды), например:

```
div.innerText = "Hello & welcome, <b>\"reader\"!</b>";
```

В результате получается следующий HTML-код:

```
<div id="content"> Hello &amp;
    welcome, &lt;b&gt;&quot;reader&quot;!&lt;/b&gt;</div>
```

Задание свойства `innerText` всегда дает в результате единственный дочерний текстовый узел, поэтому HTML-кодирование текста необходимо для его правильной обработки. С помощью свойства `innerText` можно также легко удалить HTML-теги, присвоив его самому себе:

```
div.innerText = div.innerText;
```

Если выполнить этот код, в элементе-контейнере останется только текстовый контент.

ПРИМЕЧАНИЕ Для версий Firefox до 45 (выпущенных в марте 2016 г.) единственным поддерживаемым методом был `textContent`. Он похож на `innerText`, с той лишь разницей, что `innerText` пропускает встроенный стиль и блоки скрипта, а `textContent` возвращает любой встроенный стиль или код скрипта вместе с другим текстом. `innerText` теперь поддерживается во всех браузерах и должен быть основным инструментом для получения и настройки текстового содержимого.

Свойство outerText

Свойство `outerText` аналогично свойству `innerText`, но в отличие от него включает узел, которому принадлежит. При чтении текстовых значений свойства `outerText` и `innerText` работают, в общем-то, одинаково, а в режиме записи сильно различаются. Вместо того чтобы заменять только дочерние узлы опорного элемента, свойство `outerText` заменяет весь элемент с дочерними узлами, например:

```
div.outerText = "Hello world!";
```

Эта строка кода эквивалентна двум следующим:

```
let text = document.createTextNode("Hello world!");
div.parentNode.replaceChild(text, div);
```

По сути, новый текстовый узел полностью заменяет элемент, для которого было задано свойство `outerText`. После этого исходный элемент в документе недоступен.

Свойство `outerText` не соответствует стандарту. Не рекомендуется полагаться на него в каких-то особенно важных случаях. `outerText` поддерживается во всех современных браузерах, кроме Firefox.

Прокрутка

Как уже отмечалось, до HTML5 никаких спецификаций прокрутки не было. Метод `scrollIntoView()` был стандартизирован в HTML5, но есть еще несколько фирменных методов, доступных в разных браузерах. `scrollIntoViewIfNeeded` расширяет тип `HTMLElement`, а потому доступен для всех элементов. `scrollIntoViewIfNeeded(`*выравниваниеПоЦентру*`)` — прокручивает окно браузера или элемент-контейнер, чтобы элемент появился в области просмотра, но только при условии, что он не виден. Если элемент уже отображается в области просмотра, метод ничего не делает. Если необязательный аргумент имеет значение `true`, предпринимается попытка центрировать элемент в области просмотра. Этот метод реализован в Safari, Chrome и Opera.

Пример использования:

```
// вывод элемента в область просмотра, только если он не виден
document.images[0].scrollIntoViewIfNeeded();
```

Поскольку `scrollIntoView()` — единственный метод, который поддерживается во всех браузерах, обычно используется только он.

ИТОГИ

Стандартная модель DOM определяет базовый API для взаимодействия с XML- и HTML-документами, но существует несколько спецификаций, расширяющих ее возможности. Многие такие расширения основаны на фирменных разработках, которые по мере реализации аналогичных функций в других браузерах стали стандартами де-факто. В этой главе мы рассмотрели три спецификации.

- **Selectors** определяет методы `querySelector()`, `querySelectorAll()` и `matches()`, служащие для получения DOM-элементов на основе CSS-селекторов.
- **Element Traversal** описывает дополнительные свойства DOM-элементов, позволяющие легко переходить к ближайшим связанным элементам. Эти возможности потребовались из-за различий в обработке свободного пространства между DOM-элементами.
- **HTML5** определяет целый ряд расширений стандартной DOM, таких как свойство `innerHTML`, средства управления фокусом, кодировками, прокруткой и т. д.

В настоящее время DOM-расширений немного, но по мере развития веб-технологий их количество будет расти. Удачные фирменные расширения могут со временем стать стандартами де-факто и войти в будущие версии спецификаций.

16

DOM Level 2 и 3

- Изменения в спецификациях Level 2 и 3
- DOM API для работы со стилями
- Обход и диапазоны DOM

Базовая структура HTML- и XML-документов, определенная в спецификации DOM Level 1, была расширена в DOM Level 2 и 3 интерактивными возможностями и улучшенными XML-механизмами. В результате на данный момент спецификации DOM Level 2 и 3 составляют следующие связанные модули, описывающие конкретные подмножества DOM:

- **DOM Core** — дополняет DOM Level 1 Core, добавляя к узлам методы и свойства;
- **DOM Views** — определяет для документа разные представления на основе стилей;
- **DOM Events** — обеспечивает интерактивность DOM-документов с помощью событий;
- **DOM Style** — описывает программный доступ к CSS-стилям и их изменение;
- **DOM Traversal and Range** — предоставляет новые интерфейсы для обхода DOM-документа и выделения его частей;
- **DOM HTML** — расширяет HTML Level 1 новыми свойствами, методами и интерфейсами.
- **Наблюдатели за изменениями DOM** — позволяют определить обратные вызовы при изменениях в DOM. Наблюдатели за изменениями были определены в спецификации DOM4 для замены событий изменений.

В данной главе мы обсудим все эти модули, кроме событий DOM, которые подробно описаны в главе 14 «Объектная модель документа». DOM Level 3 содержит также модули XPath и Load and Save, описываемые в главе 22 «XML в JavaScript».

ПРИМЕЧАНИЕ Очень старые версии браузера, такие как Internet Explorer 8, имеют ограниченную поддержку для некоторого содержимого этой главы. Если вы хотите поддерживать эти браузеры, перед использованием этих API рекомендуется тщательно изучить поддержку браузеров. https://caniuse.com – отличный инструмент для таких целей.

ИЗМЕНЕНИЯ DOM

Спецификации DOM Level 2 и 3 Core были разработаны для того, чтобы реализовать в DOM API все требования языка XML и улучшить обработку ошибок и распознавание функциональных возможностей. В основном это сводится к поддержке XML-пространств имен. DOM Level 2 Core не определяет никаких новых типов, а просто добавляет новые методы и свойства к типам DOM Level 1. DOM Level 3 Core расширяет доступные типы и определяет несколько новых.

Модули DOM Views и DOM HTML, которые также содержат ряд новых свойств и методов, довольно малы, и поэтому мы обсудим их вместе с DOM Core.

ПРИМЕЧАНИЕ В этой главе описаны только те части DOM, которые уже реализованы в браузерах.

XML-пространства имен

XML-пространства имен позволяют использовать элементы из разных XML-подобных языков в одном документе правильного формата без риска конфликтов имен. Технически XML-пространства имен не поддерживаются в HTML, поэтому примеры в этом разделе написаны на XHTML.

Пространства имен указываются с помощью атрибута `xmlns`. Языку XHTML соответствует пространство имен `http://www.w3.org/1999/xhtml`, которое нужно включать в элемент `<html>` любой XHTML-страницы правильного формата, например:

```
<html xmlns="http://www.w3.org/1999/xhtml">
    <head>
        <title>Example XHTML page</title>
    </head>
    <body>
        Hello world!
    </body>
</html>
```

В этом примере все элементы считаются по умолчанию частью пространства имен XHTML. Можно явно создать префикс для пространства имен XML, указав атрибут `xmlns`, двоеточие и префикс, например:

```
<xhtml:html xmlns:xhtml="http://www.w3.org/1999/xhtml">
    <xhtml:head>
        <xhtml:title>Example XHTML page</xhtml:title>
    </xhtml:head>
    <xhtml:body>
        Hello world!
    </xhtml:body>
</xhtml:html>
```

Здесь определяется пространство имен XHTML с префиксом `xhtml`, в результате все XHTML-элементы должны начинаться с этого префикса. Пространства имен можно также использовать с атрибутами для предотвращения конфликтов между языками:

```
<xhtml:html xmlns:xhtml="http://www.w3.org/1999/xhtml">
    <xhtml:head>
        <xhtml:title>Example XHTML page</xhtml:title>
    </xhtml:head>
    <xhtml:body xhtml:class="home">
        Hello world!
    </xhtml:body>
</xhtml:html>
```

Атрибуту `class` в этом примере предшествует префикс `xhtml`. Указывать пространства имен не требуется, если в документе используется только один язык, основанный на XML, но это полезно, если языков больше. Например, следующий документ содержит XHTML- и SVG-код:

```
<html xmlns="http://www.w3.org/1999/xhtml">
    <head>
        <title>Example XHTML page</title>
    </head>
    <body>
        <svg xmlns="http://www.w3.org/2000/svg" version="1.1"
             viewBox="0 0 100 100" style="width:100%; height:100%">
            <rect x="0" y="0" width="100" height="100" style="fill:red" />
        </svg>
    </body>
</html>
```

В этом примере для элемента `<svg>` задано отдельное пространство имен `http://www.w3.org/2000/svg`, которое указывает, что он является посторонним для документа. Оно применяется также ко всем дочерним элементам элемента `<svg>` и всем их атрибутам. Хотя технически это XHTML-документ, благодаря пространству имен SVG-код обрабатывается правильно.

При вызове методов, работающих с узлами такого документа, возникают интересные проблемы. Например, к какому пространству имен будут относиться создаваемые

элементы? Какие элементы будут возвращены, если запросить теги с конкретным именем? К счастью, в DOM Level 2 Core для большинства методов DOM Level 1 определены версии, учитывающие пространства имен.

Изменения типа Node

В DOM Level 2 тип `Node` включает следующие новые свойства, учитывающие пространства имен:

- `localName` — имя узла без префикса пространства имен;
- `namespaceURI` — URI пространства имен узла или `null`, если свойство не задано;
- `prefix` — префикс пространства имен или `null`, если свойство не задано.

Если для узла задан префикс пространства имен, свойство `nodeName` эквивалентно значению `prefix + ":" + localName`. Рассмотрим пример:

```
<html xmlns="http://www.w3.org/1999/xhtml">
    <head>
        <title>Example XHTML page</title>
    </head>
    <body>
        <s:svg xmlns:s="http://www.w3.org/2000/svg" version="1.1"
             viewBox="0 0 100 100" style="width:100%; height:100%">
            <s:rect x="0" y="0" width="100" height="100" style="fill:red" />
        </s:svg>
    </body>
</html>
```

У элемента `<html>` свойства `localName` и `tagName` имеют значение `"html"`, `namespaceURI` — значение `"http://www.w3.org/1999/xhtml"`, а `prefix` — значение `null`. У элемента `<s:svg>` свойство `localName` имеет значение `"svg"`, `tagName` — значение `"s:svg"`, `namespaceURI` — значение `"http://www.w3.org/2000/svg"`, а `prefix` — значение `"s"`.

В DOM Level 3 представлены следующие методы для работы с пространствами имен:

- `isDefaultNamespace(`*URIПространстваИмен*`)` — возвращает `true`, если аргумент представляет пространство имен, предлагаемое по умолчанию для узла;
- `lookupNamespaceURI(`*префикс*`)` — возвращает URI пространства имен для указанного префикса;
- `lookupPrefix(`*URIПространстваИмен*`)` — возвращает префикс для указанного URI пространства имен.

Для предыдущего примера эти методы возвратили бы следующие значения:

```
// возвращает true
console.log(document.body.isDefaultNamespace("http://www.w3.org/1999/xhtml"));

// предполагается, что svg содержит ссылку на элемент <s:svg>
console.log(svg.lookupPrefix("http://www.w3.org/2000/svg"));    // "s"
console.log(svg.lookupNamespaceURI("s"));    // "http://www.w3.org/2000/svg"
```

Эти методы полезны, если имеется ссылка на узел и нужно определить, как он связан с остальным документом.

Изменения типа Document

В DOM Level 2 тип `Document` содержит следующие новые методы, специфичные для пространств имен:

- `createElementNS(`*`URIПространстваИмен`*`, `*`имяТега`*`)` — создает элемент с заданным именем тега в пространстве имен с указанным URI;
- `createAttributeNS(`*`URIПространстваИмен`*`, `*`имяАтрибута`*`)` — создает узел атрибута в пространстве имен с указанным URI;
- `getElementsByTagNameNS(`*`URIПространстваИмен`*`, `*`имяТега`*`)` — возвращает коллекцию `NodeList`, содержащую элементы с заданным именем тега из пространства имен с указанным URI.

Обратите внимание, что в эти методы передается URI нужного пространства имен (а не префикс), например:

```
// создание SVG-элемента
let svg = document.createElementNS("http://www.w3.org/2000/svg", "svg");

// создание атрибута random в пространстве имен
let att = document.createAttributeNS("http://www.somewhere.com", "random");

// получение всех XHTML-элементов
let elems =
    document.getElementsByTagNameNS("http://www.w3.org/1999/xhtml", "*");
```

Методы, специфичные для пространств имен, могут потребоваться, только если документ содержит более одного пространства имен.

Изменения типа Element

Изменения типа `Element` в DOM Level 2 Core связаны в основном с атрибутами. Он содержит следующие новые методы:

- `getAttributeNS(`*`URIПространстваИмен`*`, `*`локальноеИмя`*`)` — получает атрибут с заданным именем из пространства имен с указанным URI;
- `getAttributeNodeNS(`*`URIПространстваИмен`*`, `*`локальноеИмя`*`)` — получает узел атрибута с заданным именем из пространства имен с указанным URI;
- `getElementsByTagNameNS(`*`URIПространстваИмен`*`, `*`имяТега`*`)` — возвращает коллекцию `NodeList`, содержащую элементы из поддерева опорного элемента, которые имеют заданное имя тега и относятся к пространству имен с указанным URI;
- `hasAttributeNS(`*`URIПространстваИмен`*`, `*`локальноеИмя`*`)` — определяет, есть ли у элемента атрибут с заданным именем из пространства имен с указанным URI (в DOM Level 2 Core есть также метод `hasAttribute()`, не учитывающий пространство имен);

- `removeAttributeNS(`*URIПространстваИмен*, *локальноеИмя*`)` — удаляет атрибут с заданным именем из пространства имен с указанным URI;
- `setAttributeNS(`*URIПространстваИмен*, *квалифицированноеИмя*, *значение*`)` — присваивает указанное значение атрибуту с заданным именем из пространства имен с указанным URI;
- `setAttributeNodeNS(`*узелАтрибута*`)` — задает узел атрибута из пространства имен с указанным URI.

Эти методы работают так же, как их аналоги из DOM Level 1, и отличаются от них только первым аргументом, которым у всех методов, кроме `setAttributeNodeNS()`, является URI пространства имен.

Изменения типа NamedNodeMap

Тип `NamedNodeMap` также содержит несколько новых методов для работы с пространствами имен. Поскольку он используется для представления атрибутов, эти методы в основном применяются к атрибутам:

- `getNamedItemNS(`*URIПространстваИмен*, *локальноеИмя*`)` — получает элемент с заданным именем из пространства имен с указанным URI;
- `removeNamedItemNS(`*URIПространстваИмен*, *локальноеИмя*`)` — удаляет элемент с заданным именем из пространства имен с указанным URI;
- `setNamedItemNS(`*узел*`)` — добавляет узел, которому уже должно быть назначено пространство имен.

Эти методы используются редко, потому что доступ к атрибутам обычно осуществляется через элемент.

Другие изменения

DOM Level 2 Core содержит также ряд других небольших изменений DOM, которые не имеют отношения к XML-пространствам имен и были внесены в основном для обеспечения гибкости и полноты API.

Изменения типа DocumentType

В тип `DocumentType` добавлены свойства `publicId`, `systemId` и `internalSubset`. Первые два из них представляют данные, которые содержатся в объявлении типа документа, но недоступны в DOM Level 1. Рассмотрим следующее объявление HTML-документа:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
     "http://www.w3.org/TR/html4/strict.dtd">
```

Здесь свойство `publicId` имеет значение `"-//W3C//DTD HTML 4.01//EN"`, а `systemId` — значение `"http://www.w3.org/TR/html4/strict.dtd"`. Если браузер поддерживает DOM Level 2, он способен выполнить следующий код:

```
console.log(document.doctype.publicId);
console.log(document.doctype.systemId);
```

Едва ли эти сведения когда-нибудь могут потребоваться на веб-страницах.

Свойство `internalSubset` обеспечивает доступ к любым дополнительным определениям в объявлении типа документа, например:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"
[<!ELEMENT name (#PCDATA)>] >
```

Для этого кода свойство `document.doctype.internalSubset` возвращает `"<!ELEMENT name (#PCDATA)>"`. Оно редко используется в HTML и чуть чаще в XML.

Изменения типа Document

Единственный новый метод типа `Document`, не связанный с пространствами имен, — это метод `importNode()`, который импортирует узел из одного документа в другой, чтобы его можно было добавить в структуру документа. Если помните, у каждого узла есть свойство `ownerDocument`, которое указывает, к какому документу относится узел. Если в метод вроде `appendChild()` передается узел, у которого свойство `ownerDocument` указывает на другой документ, происходит ошибка. При вызове метода `importNode()` для узла из другого документа возвращается новая версия узла, принадлежащая текущему документу.

Метод `importNode()` похож на метод `cloneNode()` элемента. Он принимает узел, который нужно клонировать, и логическое значение, указывающее, нужно ли также скопировать его дочерние узлы, и возвращает копию узла, например:

```
// импорт узла и всех его дочерних узлов
let newNode = document.importNode(oldNode, true);
document.body.appendChild(newNode);
```

Этот метод используется в основном с XML-документами (см. также главу 22 «XML в JavaScript»).

В DOM Level 2 Views доступно новое свойство `defaultView`, содержащее указатель на окно (или фрейм), которому принадлежит документ. В спецификации Views не указано, когда могут быть доступны другие представления, так что это единственное добавленное свойство. Оно поддерживается во всех браузерах, кроме Internet Explorer 8 и более ранних версий, где доступно эквивалентное свойство `parentWindow` (оно доступно также в Opera). Определить окно, которому принадлежит документ, можно следующим образом:

```
let parentWindow = document.defaultView || document.parentWindow;
```

В DOM Level 2 Core к объекту `document.implementation` добавлены методы `createDocumentType()` и `createDocument()`. Первый из них создает узел `DocumentType`, принимая три аргумента: тип документа и свойства `publicId` и `systemId`. Например, следующий код создает тип документа HTML 4.01 Strict:

```
let doctype = document.implementation.createDocumentType("html",
                  "-//W3C//DTD HTML 4.01//EN",
                  "http://www.w3.org/TR/html4/strict.dtd");
```

Тип существующего документа изменить нельзя, поэтому метод `createDocumentType()` полезен только при создании документов, для чего можно использовать метод `createDocument()`. Он принимает три аргумента: URI пространства имен элемента документа, имя тега элемента документа и тип нового документа. Например, так можно создать пустой XML-документ:

```
let doc = document.implementation.createDocument("", "root", null);
```

Этот код создает документ с элементом документа `<root>` без пространства имен и типа документа. Создать XHTML-документ можно следующим образом:

```
let doctype = document.implementation.createDocumentType("html",
                  "-//W3C//DTD XHTML 1.0 Strict//EN",
                  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd");

let doc = document.implementation.createDocument(
    "http://www.w3.org/1999/xhtml", "html", doctype);
```

В этом примере создается XHTML-документ с соответствующими пространством имен и типом документа. Он содержит единственный элемент `<html>`, все остальное содержимое нужно добавлять.

Модуль DOM Level 2 HTML добавляет к объекту `document.implementation` метод `createHTMLDocument()`, который создает полный HTML-документ с элементами `<html>`, `<head>`, `<title>` и `<body>`. Он принимает заголовок нового документа (строковое содержимое элемента `<title>`) и возвращает HTML-документ:

```
let htmldoc = document.implementation.createHTMLDocument("New Doc");
console.log(htmldoc.title);              // "New Doc"
console.log(typeof htmldoc.body);        // "object"
```

Объект, созданный с помощью метода `createHTMLDocument()`, имеет тип `HTMLDocument` и содержит все его свойства и методы, включая свойства `title` и `body`.

Изменения типа Node

Для сравнения узлов в DOM Level 3 представлены методы `isSameNode()` и `isEqualNode()`. Они принимают узел и возвращают `true`, если полученный и опорный узлы одинаковы или эквивалентны соответственно. Узлы одинаковы, если они ссылаются на один объект, и эквивалентны, если они имеют один тип, содержат равные свойства `nodeName`, `nodeValue` и т. д., причем их свойства `attributes` и `childNodes` эквивалентны (содержат одинаковые значения в аналогичных позициях), например:

```
let div1 = document.createElement("div");
div1.setAttribute("class", "box");

let div2 = document.createElement("div");
```

```
div2.setAttribute("class", "box");

console.log(div1.isSameNode(div1));     // true
console.log(div1.isEqualNode(div2));    // true
console.log(div1.isSameNode(div2));     // false
```

Здесь создаются два элемента `<div>` с одними и теми же атрибутами. Эти элементы эквивалентны, но не одинаковы.

В DOM Level 3 добавлены также методы присоединения дополнительных данных к DOM-узлам. Метод `setUserData()`, который назначает данные узлу, принимает три аргумента: задаваемый ключ, фактические данные (которые могут иметь любой тип) и функцию-обработчик. Назначить данные узлу можно следующим образом:

```
document.body.setUserData("name", "Nicholas", function() {});
```

Получить эти данные можно с помощью метода `getUserData()`, передав в него тот же ключ:

```
let value = document.body.getUserData("name");
```

Функция-обработчик для метода `setUserData()` вызывается при клонировании, удалении, переименовании узла с данными или при его импорте в другой документ, позволяя указать, что нужно сделать с данными пользователя в каждом из этих случаев. Она принимает пять аргументов: число, задающее тип операции (1 — клонирование; 2 — импорт; 3 — удаление; 4 — переименование), ключ данных, значение данных, исходный узел и целевой узел. Исходный узел равен `null`, если узел удаляется, а целевой узел равен `null` при всех операциях, кроме клонирования. Затем можно указать, как следует сохранить данные. Вот пример:

```
let div = document.createElement("div");
div.setUserData("name", "Nicholas",
                function(operation, key, value, src, dest) {
    if (operation == 1) {
        dest.setUserData(key, value, function() {}); }
});

let newDiv = div.cloneNode(true);
console.log(newDiv.getUserData("name"));    // "Nicholas"
```

Созданному элементу `<div>` здесь назначаются некоторые данные, включая пользовательские. Затем этот элемент клонируется методом `cloneNode()`, при этом вызывается функция-обработчик, и данные автоматически назначаются клону. В результате метод `getUserData()` возвращает для клона то же значение, которое было назначено оригиналу.

Изменения фреймов

Встроенные фреймы, представленные типом `HTMLIFrameElement`, имеют в DOM Level 2 HTML новое свойство `contentDocument`. Оно содержит указатель на объект

document или содержимое фрейма. Данное свойство можно использовать следующим образом:

```
let iframe = document.getElementById("myIframe");
let iframeDoc = iframe.contentDocument;
```

Также существует свойство contentWindow, которое возвращает для фрейма объект window со свойством document. Свойства contentWindow и contentDocument доступны во всех современных браузерах.

> **ПРИМЕЧАНИЕ** Доступ к объекту document встроенного фрейма регламентируется междоменными ограничениями. Попытка доступа к объекту document фрейма со страницей, загруженной из другого домена или поддомена либо по другому протоколу, приведет к ошибке.

СТИЛИ

Чтобы определить в HTML-коде стили, можно включить в файл внешнюю таблицу стилей с помощью элемента <link>, добавить в файл встроенные стили, используя элемент <style>, или задать стили для отдельного элемента, указав атрибут style. DOM Level 2 Styles предоставляет API для всех этих механизмов.

Доступ к стилям элементов

У любого HTML-элемента, поддерживающего атрибут style, есть свойство style, доступное в JS-коде. Объект style является экземпляром типа CSSStyleDeclaration и содержит все стили, заданные с помощью HTML-атрибута style, но не каскадно применяемые стили из внешних или встроенных таблиц стилей. Каждое CSS-свойство, заданное в атрибуте style, представлено свойством объекта style. Поскольку в именах CSS-свойств слова разделяются дефисом (например, background-image), для использования в JS-коде их нужно преобразовать в верблюжью нотацию. В следующей таблице указаны некоторые часто используемые CSS-свойства и соответствующие им свойства объекта style.

CSS-СВОЙСТВО	JAVASCRIPT-СВОЙСТВО
background-image	style.backgroundImage
color	style.color
display	style.display
font-family	style.fontFamily

Имена CSS-свойств преобразуются в имена JavaScript-свойств напрямую, исключая свойство float. Поскольку в JavaScript это зарезервированное слово, его нельзя использовать как имя свойства. В спецификации DOM Level 2 Style сказано, что соответствующее свойство объекта style должно называться cssFloat.

Стили можно задавать с помощью JS-кода в любое время, если доступна действительная ссылка на DOM-элемент, например:

```
let myDiv = document.getElementById("myDiv");

// задание фонового цвета
myDiv.style.backgroundColor = "red";

// изменение размеров
myDiv.style.width = "100px";
myDiv.style.height = "200px";

// настройка границы
myDiv.style.border = "1px solid black";
```

При изменении стилей таким способом вид элемента автоматически обновляется.

> **ПРИМЕЧАНИЕ** В стандартном режиме все размеры должны включать единицу измерения. Если присвоить свойству `style.width` значение «`20`» в режиме совместимости, оно будет интерпретировано как `"20px"`, но в стандартном режиме оно игнорируется. На практике лучше всегда указывать единицу измерения.

Стили, заданные в атрибуте `style`, можно также получать с помощью объекта `style`. Рассмотрим следующий HTML-код:

```
<div id="myDiv" style="background-color: blue; width: 10px; height: 25px">
</div>
```

Содержимое атрибута `style` этого элемента можно получить следующим образом:

```
console.log(myDiv.style.backgroundColor);    // "blue"
console.log(myDiv.style.width);              // "10px"
console.log(myDiv.style.height);             // "25px"
```

Если для элемента не указан атрибут `style`, объект `style` будет содержать пустые значения для всех возможных свойств CSS.

Свойства и методы Style

Спецификация DOM Level 2 Style определяет также несколько свойств и методов объекта `style`, которые предоставляют информацию о содержимом атрибута `style` и изменяют его:

- `cssText` — возвращает CSS-код атрибута `style`;
- `length` — количество CSS-свойств, примененных к элементу;
- `parentRule` — объект `CSSRule`, представляющий CSS-информацию (тип `CSSRule` обсуждается позже);
- `getPropertyCSSValue(`*`имяСвойства`*`)` — возвращает объект `CSSValue`, содержащий значение указанного свойства;

- `getPropertyPriority(`*имяСвойства*`)` — возвращает значение `"important"`, если указанное свойство задано с объявлением `!important`, иначе возвращает пустую строку;
- `getPropertyValue(`*имяСвойства*`)` — возвращает строковое значение указанного свойства;
- `item(`*индекс*`)` — возвращает имя CSS-свойства в указанной позиции;
- `removeProperty(`*имяСвойства*`)` — удаляет указанное свойство из стиля;
- `setProperty(`*имяСвойства*, *значение*, *приоритет*`)` — присваивает указанному свойству полученное значение с заданным приоритетом (`"important"` или пустая строка).

Свойство `cssText` обеспечивает доступ к CSS-коду стиля. В режиме чтения оно возвращает внутреннее представление значения атрибута `style` в браузере. При записи свойства `cssText` присваиваемое ему значение перезаписывает все содержимое атрибута `style`, то есть все прежние параметры стиля, заданные с помощью этого атрибута, удаляются. Например, если для элемента в атрибуте `style` задана граница и свойство `cssText` перезаписывается по правилам, которые не включают границу, она удаляется из элемента. Свойство `cssText` используется следующим образом:

```
myDiv.style.cssText = "width: 25px; height: 100px; background-color: green";
console.log(myDiv.style.cssText);
```

Задание свойства `cssText` — это самый быстрый способ внесения многих изменений в стиль элемента, потому что все они применяются одновременно.

Свойство `length` используется вместе с методом `item()` для перебора CSS-свойств элемента. По сути, объект `style` при этом обрабатывается как коллекция, так что для получения имени CSS-свойства в конкретной позиции можно задействовать скобочную нотацию вместо метода `item()`:

```
for (let i=0, len=myDiv.style.length; i < len; i++) {
    console.log(myDiv.style[i]);        // или myDiv.style.item(i)
}
```

Используя скобочную нотацию или метод `item()`, можно получить имя CSS-свойства (`"background-color"`, но не `"backgroundColor"`), а затем передать его в метод `getPropertyValue()` для получения значения свойства:

```
let prop, value, i, len;
for (i=0, len=myDiv.style.length; i < len; i++) {
    prop = myDiv.style[i];        // или myDiv.style.item(i)
    value = myDiv.style.getPropertyValue(prop);
    console.log('prop: ${value}');
}
```

Метод `getPropertyValue()` всегда возвращает значение CSS-свойства как строку. Если вам нужны дополнительные сведения, можно использовать метод `getPropertyCSSValue()`, который возвращает объект `CSSValue` со свойствами `cssText` и `cssValueType`. Свойство `cssText` не отличается от значения, возвращаемого методом

`getPropertyValue()`, а свойство `cssValueType` содержит числовую константу, указывающую тип значения свойства: 0 — унаследованное значение; 1 — примитивное значение; 2 — список; 3 — пользовательское значение. Следующий код выводит значение CSS-свойства и его тип:

```
let prop, value, i, len;
for (i=0, len=myDiv.style.length; i < len; i++) {
    prop = myDiv.style[i];        //или myDiv.style.item(i)
    value = myDiv.style.getPropertyCSSValue(prop);
    console.log('prop: ${value.cssText} (${value.cssValueType})');
}
```

Метод `removeProperty()` удаляет CSS-свойство из стиля элемента, при этом к элементу применяется предлагаемый по умолчанию стиль, полученный по каскаду из других таблиц стилей. Например, следующий код удаляет свойство `border`, которое было задано в атрибуте `style`:

```
myDiv.style.removeProperty("border");
```

Этот метод полезен, если вы не знаете, каково предлагаемое по умолчанию значение конкретного CSS-свойства. Для восстановления значения, предлагаемого по умолчанию, можно просто удалить свойство.

Вычисляемые стили

Объект `style` позволяет получить значение атрибута `style` любого элемента, который его поддерживает, но не содержит сведений о стилях, примененных к элементу по каскаду из таблиц стилей. DOM Level 2 Style расширяет объект `document.defaultView` методом `getComputedStyle()`, принимающим два аргумента: элемент, для которого нужно получить вычисляемый стиль, и строку псевдоэлемента (например, `":after"`). Вторым аргументом может быть значение `null`, если сведения о псевдоэлементе не нужны. Метод `getComputedStyle()` возвращает объект `CSSStyleDeclaration` (как и свойство `style`), содержащий все вычисляемые стили элемента. Рассмотрим следующую HTML-страницу:

```
<!DOCTYPE html>
<html>
<head>
    <title>Computed Styles Example</title>
    <style type="text/css">
        #myDiv {
            background-color: blue;
            width: 100px;
            height: 200px;
        }
    </style>
</head>
<body>
    <div id="myDiv" style="background-color: red; border: 1px solid black">
    </div>
</body>
</html>
```

В этом примере к элементу `<div>` применены стили из встроенной таблицы стилей (элемент `<style>`) и из атрибута `style`. Объект `style` содержит значения свойств `backgroundColor` и `border`, но не `width` и `height`, которые берутся из таблицы стилей. Следующий код получает вычисляемый стиль элемента:

```
let myDiv = document.getElementById("myDiv");
let computedStyle = document.defaultView.getComputedStyle(myDiv, null);

console.log(computedStyle.backgroundColor);    // "red"
console.log(computedStyle.width);              // "100px"
console.log(computedStyle.height);             // "200px"
console.log(computedStyle.border);             // "1px solid black" в ряде
браузеров
```

Этот код возвращает фоновый цвет `"red"`, ширину `"100px"` и высоту `"200px"`. Обратите внимание, что фоновый цвет отличается от `"blue"`, потому что он переопределен в самом элементе. Свойство `border` может не возвратить точное правило `border` из таблицы стилей (Opera его возвращает, но не остальные браузеры). Это несоответствие связано с тем, как браузеры интерпретируют агрегирующие свойства вроде `border`, которые на самом деле задают ряд других свойств. При установке свойства `border` вы на самом деле задаете правила для ширины, цвета и стиля всех четырех границ (`border-left-width`, `border-top-color`, `border-bottom-style` и т. д.). Таким образом, хотя свойство `computedStyle.border` возвращает значение не во всех браузерах, свойство `computedStyle.borderLeftWidth` работает везде.

> **ПРИМЕЧАНИЕ** В браузерах, которые поддерживают этот функционал, значения стилей представляются по-разному. Firefox и Safari преобразуют все цвета в формат RGB (например, `rgb(255,0,0)` для красного), а Opera представляет цвета в шестнадцатеричном формате (`#ff0000` для красного). При использовании метода `getComputedStyle()` следует тестировать код в разных браузерах.

Вычисляемые стили во всех браузерах доступны только для чтения — изменить CSS-свойства в объекте вычисляемого стиля нельзя. Кроме того, вычисляемый стиль содержит стили из внутренней таблицы стилей браузера, а потому в нем представлено любое CSS-свойство, у которого есть значение, предлагаемое по умолчанию. Например, у свойства `visibility` есть значение, предлагаемое по умолчанию во всех браузерах, но оно зависит от реализации. В некоторых браузерах это свойство имеет по умолчанию значение `"visible"`, в других — `"inherit"`. Нельзя полагаться на то, что значение CSS-свойства, предлагаемое по умолчанию, будет одинаковым во всех браузерах. Если требуется, чтобы элементы по умолчанию имели определенное значение, следует задать его вручную в таблице стилей.

Работа с таблицами стилей

Тип `CSSStyleSheet` представляет таблицу CSS-стилей, включенную в файл с помощью элемента `<link>` или определенную в элементе `<style>`. Сами элементы при этом

имеют типы HTMLLinkElement и HTMLStyleElement соответственно. Тип CSSStyleSheet достаточно универсален, чтобы с его помощью можно было представить таблицу стилей независимо от того, как она определена в HTML. Типы, специфичные для элементов, позволяют изменять HTML-атрибуты, тогда как объект CSSStyleSheet за исключением одного свойства доступен только для чтения.

Тип CSSStyleSheet наследуется от типа StyleSheet, который можно использовать как основу для определения таблиц стилей, отличных от CSS. От типа StyleSheet наследуются следующие свойства.

- disabled — логическое значение, указывающее, отключена ли таблица стилей. Это свойство доступно для чтения и записи, так что можно отключить таблицу стилей, присвоив ему значение true.
- href — URL-адрес таблицы стилей, если она включается в файл с помощью элемента <link>, иначе значение null.
- media — носители информации, поддерживаемые таблицей стилей. Как и все DOM-коллекции, эта содержит свойство length и метод item(). Для доступа к элементам коллекции можно задействовать скобочную нотацию. Пустой список указывает, что таблица стилей должна использоваться со всеми носителями.
- ownerNode — указатель на узел, которому принадлежит таблица стилей. В HTML им может быть элемент <link> или <style>, а в XML — инструкция по обработке. Если таблица стилей включается в другую таблицу стилей с помощью правила @import, это свойство имеет значение null.
- parentStyleSheet — если таблица стилей включается в файл с помощью правила @import, это свойство содержит указатель на таблицу стилей, из которой импортируется первая таблица.
- title — значение атрибута title узла ownerNode.
- type — строка, указывающая тип таблицы стилей ("text/css" в случае таблицы CSS-стилей).

За исключением disabled, все эти свойства доступны только для чтения. В дополнение к ним тип CSSStyleSheet поддерживает свои свойства и методы.

- cssRules — набор правил, содержащихся в таблице стилей.
- ownerRule — если таблица стилей была включена в файл с помощью правила @import, это свойство содержит указатель на правило, представляющее импорт, в противном случае оно имеет значение null.
- deleteRule(*индекс*) — удаляет правило в указанной позиции в коллекции cssRules.
- insertRule(*правило*, *индекс*) — вставляет полученное строковое правило в указанной позиции в коллекцию cssRules.

Таблицы стилей документа содержатся в коллекции document.styleSheets. Количество таблиц стилей у документа можно узнать с помощью свойства length, а для доступа к отдельным таблицам стилей можно использовать метод item() или скобочную нотацию, например:

```
let sheet = null;
for (let i=0, len=document.styleSheets.length; i < len; i++) {
    sheet = document.styleSheets[i];
    console.log(sheet.href);
}
```

Этот код выводит на экран свойство `href` каждой таблицы стилей в документе (у элементов `<style>` нет свойства `href`).

Таблицы стилей, возвращаемые свойством `document.styleSheets`, зависят от браузера. Все браузеры возвращают элементы `<style>`, а также элементы `<link>`, у которых атрибут `rel` имеет значение `"stylesheet"`, а Internet Explorer и Opera — еще и элементы `<link>`, у которых атрибут `rel` равен `"alternate stylesheet"`.

Можно также получить объект `CSSStyleSheet` непосредственно из элемента `<link>` или `<style>` с помощью свойства `sheet`, которое содержит объект `CSSStyleSheet`.

Правила CSS

Объект `CSSRule` представляет отдельное правило в таблице стилей. От типа `CSSRule` наследуются несколько других типов, из которых чаще всего используется тип `CSSStyleRule`, представляющий данные стиля (среди других правил — `@import`, `@font-face`, `@page` и `@charset`, но в сценариях они требуются редко). Перечислим свойства объекта `CSSStyleRule`.

- `cssText` — возвращает текст всего правила. Он может отличаться от фактического текста в таблице стилей из-за особенностей внутренней обработки таблиц стилей в браузерах (например, Safari всегда преобразует текст в нижний регистр).
- `parentRule` — если правило импортируется, это свойство содержит правило-контейнер, иначе оно равно `null`.
- `parentStyleSheet` — таблица стилей, в которую входит правило.
- `selectorText` — возвращает текст селектора для правила. Он может отличаться от фактического текста в таблице стилей из-за особенностей внутренней обработки таблиц стилей в браузерах. Это свойство доступно только для чтения в Firefox, Safari, Chrome и Internet Explorer (где оно генерирует ошибку). В Opera его также можно изменять.
- `style` — объект `CSSStyleDeclaration`, позволяющий задавать и читать отдельные стили в правиле.
- `type` — константа, указывающая тип правила. У правил стилей она всегда равна 1.

Чаще всего используются свойства `cssText`, `selectorText` и `style`. Свойство `cssText` похоже на `style.cssText`, но не идентично ему. Первое включает текст селектора и данные стиля в фигурных скобках, а второе содержит только данные стиля (подобно свойству `style.cssText` элемента). Кроме того, свойство `cssText` доступно только для чтения, а `style.cssText` можно перезаписывать.

В большинстве случаев свойства `style` достаточно для манипулирования правилами стилей. Его можно использовать для чтения и изменения стилей в правиле так же, как и аналогичное свойство элемента. Рассмотрим следующее CSS-правило:

```
div.box {
    background-color: blue;
    width: 100px;
    height: 200px;
}
```

Если это правило содержится в первой таблице стилей на странице и является в ней единственным, получить все эти данные можно следующим образом:

```
let sheet = document.styleSheets[0];
let rules = sheet.cssRules || sheet.rules;    // получение списка правил
let rule = rules[0];                          // получение первого правила
console.log(rule.selectorText);                     // "div.box"
console.log(rule.style.cssText);                    // полный CSS-код
console.log(rule.style.backgroundColor);            // "blue"
console.log(rule.style.width);                      // "100px"
console.log(rule.style.height);                     // "200px"
```

Используя эту методику, можно читать стили, связанные с правилами, так же как встроенные стили элементов. Изменение стилей также возможно, например:

```
let sheet = document.styleSheets[0];
let rules = sheet.cssRules || sheet.rules;    // получение списка правил
let rule = rules[0];                          // получение первого правила
rule.style.backgroundColor = "red"
```

Изменение правила таким способом влияет на все элементы, к которым оно применено. Например, если страница содержит два элемента `<div>` класса `box`, будут изменены оба элемента.

Создание правил

Для добавления новых правил в существующие таблицы стилей используется метод `insertRule()`, который принимает два аргумента: текст правила и позицию, где нужно его вставить, например:

```
sheet.insertRule("body { background-color: silver }", 0);    // DOM-метод
```

Этот код вставляет в таблицу стилей правило, изменяющее фоновый цвет документа. Новое правило становится первым (позиция 0), что важно для определения порядка каскадного применения правил в документе.

К сожалению, этот способ слишком громоздок, если нужно добавить много правил. В этом случае лучше использовать динамическую загрузку стилей (см. главу 14 «Объектная модель документа»).

Удаление правил

Для удаления правил из таблицы стилей в DOM используется метод `deleteRule()`, принимающий индекс правила, которое нужно удалить. Например, удалить первое правило в таблице стилей можно следующим образом:

```
sheet.deleteRule(0);    // DOM-метод
```

Добавлять и удалять правила при разработке веб-приложений требуется редко, при этом нужно соблюдать осторожность, чтобы не нарушить каскадное применение стилей.

Размеры элементов

Описываемые здесь свойства и методы не входят в спецификацию DOM Level 2 Style, но все же связаны со стилями HTML-элементов. DOM не описывает способы определения фактических размеров элементов на странице. Некоторые такие свойства впервые появились в Internet Explorer, а затем были реализованы во всех основных браузерах.

Смещения

Свойства из первой группы представляют *размеры смещений* (offset dimensions), которые охватывают все визуальное пространство, занимаемое элементом на экране. Оно определяется высотой и шириной элемента и включает все отступы, полосы прокрутки и границы элемента (но не поля). Получить размеры смещений можно с помощью четырех следующих свойств:

- `offsetHeight` — размер элемента по вертикали в пикселях, включающий высоту самого элемента, горизонтальной полосы прокрутки (если она отображается), а также верхней и нижней границ;
- `offsetLeft` — расстояние в пикселях между левой внешней границей элемента и левой внутренней границей элемента-контейнера;
- `offsetTop` — расстояние в пикселях между верхней внешней границей элемента и верхней внутренней границей элемента-контейнера;
- `offsetWidth` — размер элемента по горизонтали в пикселях, включающий ширину самого элемента, вертикальной полосы прокрутки (если она отображается), а также левой и правой границ.

Значения `offsetLeft` и `offsetTop` рассчитываются относительно элемента-контейнера, который хранится в свойстве `offsetParent` и может отличаться от `parentNode`. Например, у элемента `<td>` свойство `offsetParent` указывает на соответствующий элемент `<table>`, потому что это первый элемент в иерархии, для которого определены размеры. Размеры, соответствующие этим свойствам, показаны на рис. 16.1.

Смещение элемента на странице можно примерно определить, сложив значения свойств `offsetLeft` и `offsetTop` с аналогичными значениями элемента `offsetParent`

и более высокоуровневых элементов в иерархии вплоть до корневого элемента, например:

```
function getElementLeft(element) {
    let actualLeft = element.offsetLeft;
    let current = element.offsetParent;

    while (current !== null) {
        actualLeft += current.offsetLeft;
        current = current.offsetParent;
    }

    return actualLeft;
}

function getElementTop(element) {
    let actualTop = element.offsetTop;
    let current = element.offsetParent;

    while (current !== null) {
        actualTop += current.offsetTop;
        current = current.offsetParent;
    }

    return actualTop;
}
```

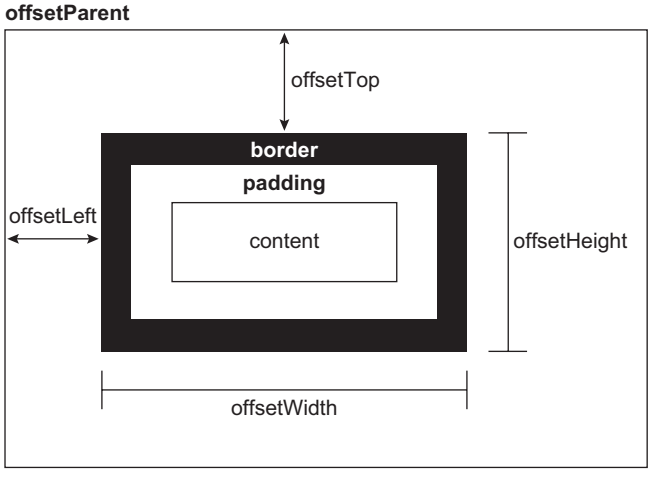

Рис. 16.1

Эти функции поднимаются по DOM-иерархии с помощью свойства `offsetParent`, суммируя смещения на каждом уровне. Если страницы основаны на простых CSS-макетах, эти функции очень точны. Если страницы содержат таблицы и встроенные фреймы, возвращаемые этими функциями значения могут зависеть от браузера из-за различий реализации. Как правило, у всех элементов, содержащихся

в элементах `<div>`, свойство `offsetParent` указывает на элемент `<body>`, так что функции `getElementLeft()` и `getElementTop()` возвращают для них те же значения, что и свойства `offsetLeft` и `offsetTop`.

ПРИМЕЧАНИЕ Все свойства смещений доступны только для чтения и вычисляются при каждом обращении к ним. Чтобы не снижать быстродействие, лучше не вызывать эти свойства повторно, а кешировать их значения в локальных переменных.

Клиентские размеры

Клиентские размеры (client dimensions) элемента, которые представлены свойствами `clientWidth` и `clientHeight`, определяют пространство, занимаемое содержимым элемента и отступами. Значение `clientWidth` равно ширине области контента с левым и правым отступами, а `clientHeight` — высоте области контента с верхним и нижним отступами. Эти размеры показаны на рис. 16.2.

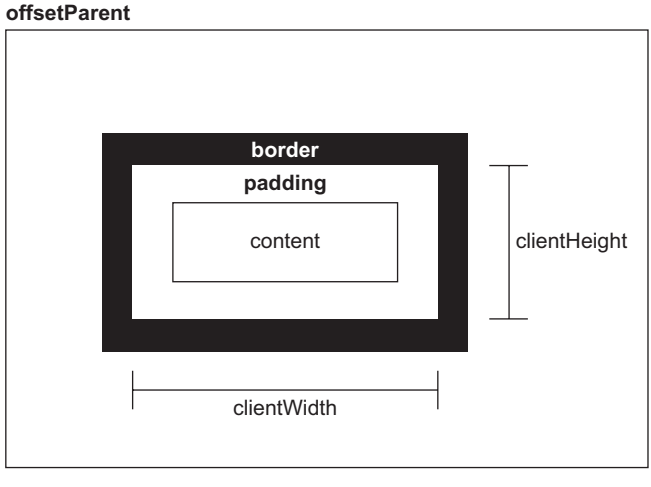

Рис. 16.2

Клиентские размеры не включают полосы прокрутки и чаще всего используются для определения размеров области просмотра браузера (см. главу 8). Это делается с помощью свойств `clientWidth` и `clientHeight` объекта `document.documentElement`, которые представляют размеры области просмотра (элемент `<html>` или `<body>`).

ПРИМЕЧАНИЕ Как и смещения, клиентские размеры доступны только для чтения и вычисляются при каждом обращении к ним.

Размеры области прокрутки

Последняя группа размеров — это *размеры области прокрутки* (scroll dimensions), предоставляющие сведения об элементе, содержимое которого можно прокручивать. Некоторые элементы, такие как `<html>`, прокручиваются автоматически без дополнительного кода, тогда как другие можно прокручивать с помощью CSS-свойства `overflow`. Четыре размера прокрутки таковы:

- `scrollHeight` — общая высота контента без полос прокрутки;
- `scrollLeft` — количество скрытых пикселей слева от области контента (с помощью этого свойства можно прокрутить область элемента);
- `scrollTop` — количество скрытых пикселей сверху от области контента (с помощью этого свойства можно прокрутить область элемента);
- `scrollWidth` — общая ширина контента без полос прокрутки.

Эти размеры показаны на рис. 16.3.

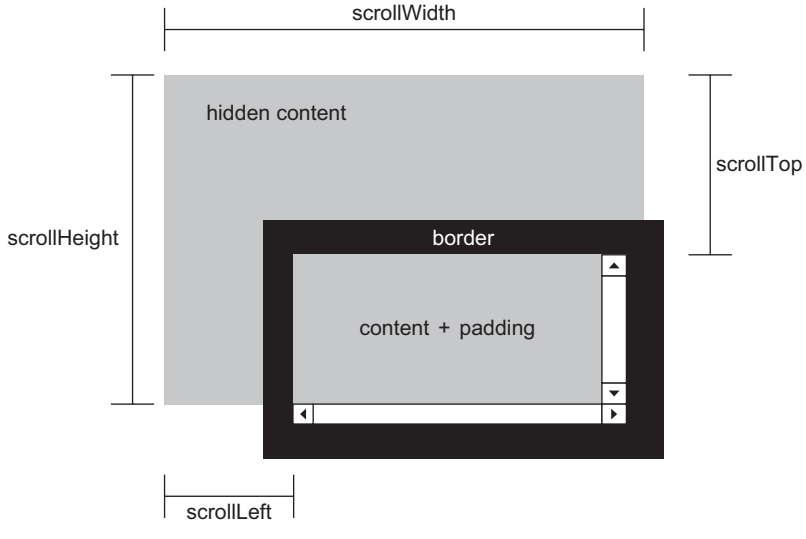

Рис. 16.3

С помощью свойств `scrollWidth` и `scrollHeight` можно узнать фактические размеры содержимого элемента. Например, прокручиваемая область просмотра в веб-браузере представлена элементом `<html>`. Следовательно, высота всей страницы, которую можно прокручивать по вертикали, равна `document.documentElement.scrollHeight`.

Если документ не прокручивается, отношение между значениями `scrollWidth` и `scrollHeight`, с одной стороны, и `clientWidth` и `clientHeight` — с другой, не определено. У объекта `document.documentElement` эти свойства имеют несогласованные значения в разных браузерах:

- в Firefox эти свойства равны, но их значения определяются по фактическому контенту документа, а не по области просмотра;
- в Opera, Safari и Chrome значения `scrollWidth` и `scrollHeight` соответствуют размерам области просмотра, а `clientWidth` и `clientHeight` — размерам контента документа;
- в Internet Explorer (в стандартном режиме) значения `scrollWidth` и `scrollHeight` соответствуют размерам контента документа, а `clientWidth` и `clientHeight` — размерам области просмотра.

При определении общих размеров документа, в том числе минимальных размеров по области просмотра, для получения правильных результатов в разных браузерах нужно брать максимальные значения из пар `scrollWidth`/`clientWidth` и `scrollHeight`/`clientHeight`, например:

```
let docHeight = Math.max(document.documentElement.scrollHeight,
                         document.documentElement.clientHeight);

let docWidth = Math.max(document.documentElement.scrollWidth,
                        document.documentElement.clientWidth);
```

Свойства `scrollLeft` и `scrollTop` можно использовать как для определения текущих параметров прокрутки, так и для их установки. Если элемент не прокручивался, оба свойства равны нулю. Если элемент прокручивался по вертикали, свойство `scrollTop` указывает объем скрытого содержимого сверху от отображаемой области элемента, а свойство `scrollLeft` — слева от нее. Обнулив свойства `scrollLeft` и `scrollTop`, можно показать в элементе начальную область. Следующая функция прокручивает содержимое элемента по вертикали к самому началу, если оно скрыто:

```
function scrollToTop(element) {
    if (element.scrollTop != 0) {
        element.scrollTop = 0;
    }
}
```

Для прокрутки содержимого функция использует свойство `scrollTop`.

Определение размеров элемента

В браузерах у каждого элемента есть метод `getBoundingClientRect()`. Он возвращает объект `DOMRect` со свойствами `left`, `top`, `right`, `bottom`, `height` и `width`, которые определяют расположение элемента на странице относительно области просмотра.

ОБХОД

В DOM Level 2 Traversal and Range определены типы `NodeIterator` и `TreeWalker`, служащие для последовательного обхода DOM-структуры в глубину, начиная с указанной точки.

Обход DOM-структуры возможен как минимум в двух направлениях (в зависимости от используемого типа), при этом подняться по DOM-дереву выше корневого узла нельзя. Рассмотрим следующую HTML-страницу:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Example</title>
    </head>
    <body>
        <p><b>Hello</b> world!</p>
    </body>
</html>
```

Этой странице соответствует DOM-дерево, показанное на рис. 16.4.

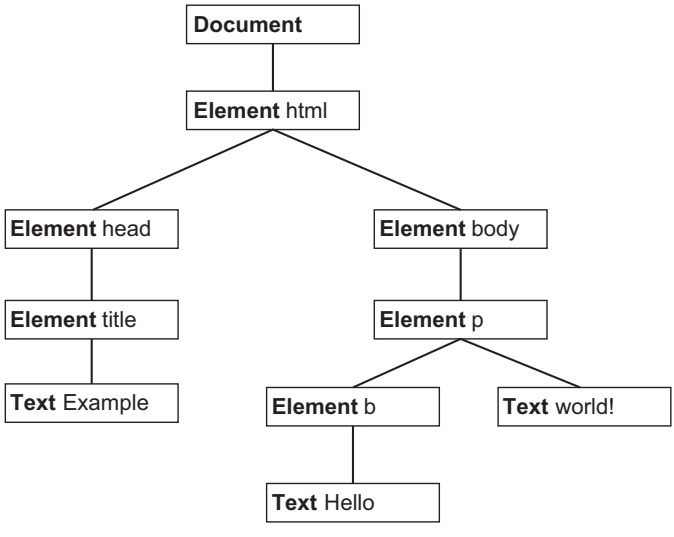

Рис. 16.4

Корневым при обходе может быть любой узел. Предположим, например, что им является элемент `<body>`. Тогда при обходе могут быть посещены элементы `<p>` и `<b>`, а также два текстовых узла, которые являются потомками `<body>`, но не элементы `<html>`, `<head>` или любые другие узлы, не входящие в поддерево элемента `<body>`. В то же время при обходе с корневым узлом `document` доступны все узлы документа. На рис. 16.5 показан обход DOM-дерева с корнем `document` в глубину.

Обход начинается узлом `document` и завершается текстовым узлом `" world!"`, но можно обойти узлы и в обратном направлении. В этом случае текстовый узел `" world!"` будет посещен первым, а узел `document` — последним. Типы `NodeIterator` и `TreeWalker` поддерживают оба способа.

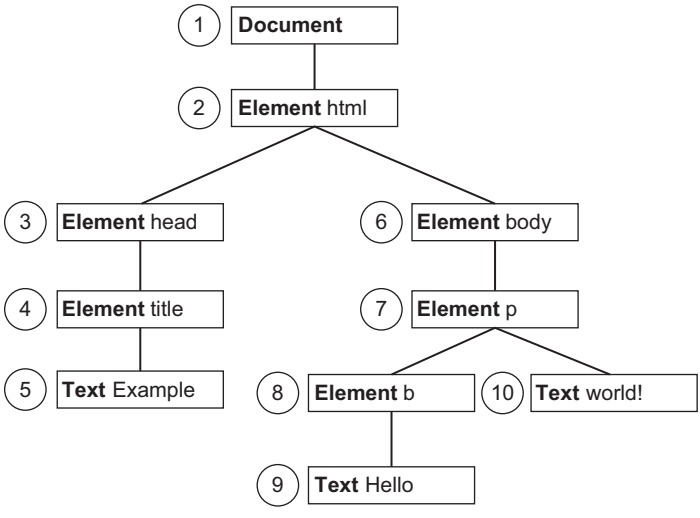

Рис. 16.5

Тип NodeIterator

Тип `NodeIterator` проще, чем `TreeWalker`; создать его экземпляр можно с помощью метода `document.createNodeIterator()`, который принимает четыре аргумента:

- `root` — узел в дереве, с которого нужно начать обход;
- `whatToShow` — числовой код, указывающий, какие узлы нужно посетить;
- `filter` — объект `NodeFilter` или некая функция, указывающая, нужно посетить или нет конкретный узел;
- `entityReferenceExpansion` — логическое значение, указывающее, нужно ли развертывать ссылки на сущности (в HTML-страницах ссылки на сущности никогда не развертываются).

Аргумент `whatToShow` — это битовая маска с фильтрами, которые определяют, какие узлы нужно посетить. Значениями этого аргумента могут быть следующие константы типа `NodeFilter`:

- `NodeFilter.SHOW_ALL` — отображать узлы всех типов;
- `NodeFilter.SHOW_ELEMENT` — отображать узлы элементов;
- `NodeFilter.SHOW_ATTRIBUTE` — отображать узлы атрибутов (из-за DOM-структуры использовать это значение невозможно);
- `NodeFilter.SHOW_TEXT` — отображать текстовые узлы;
- `NodeFilter.SHOW_CDATA_SECTION` — отображать узлы CDATA-разделов (в HTML-страницах не используется);
- `NodeFilter.SHOW_ENTITY_REFERENCE` — отображать узлы ссылок на сущности (в HTML-страницах не используется);

- `NodeFilter.SHOW_ENTITY` — отображать узлы сущностей (в HTML-страницах не используется);
- `NodeFilter.SHOW_PROCESSING_INSTRUCTION` — отображать узлы инструкций по обработке (в HTML-страницах не используется);
- `NodeFilter.SHOW_COMMENT` — отображать узлы комментариев;
- `NodeFilter.SHOW_DOCUMENT` — отображать узлы документов;
- `NodeFilter.SHOW_DOCUMENT_TYPE` — отображать узлы типов документов;
- `NodeFilter.SHOW_DOCUMENT_FRAGMENT` — отображать узлы фрагментов документов (в HTML-страницах не используется);
- `NodeFilter.SHOW_NOTATION` — отображать узлы обозначений (в HTML-страницах не используется).

С помощью поразрядного оператора ИЛИ можно комбинировать эти параметры (исключая значение `NodeFilter.SHOW_ALL`), например:

```
let whatToShow = NodeFilter.SHOW_ELEMENT | NodeFilter.SHOW_TEXT;
```

С помощью аргумента `filter` метода `createNodeIterator()` можно задать в качестве фильтра узлов собственный объект `NodeFilter` или функцию, действующую как фильтр узлов. У объекта `NodeFilter` есть единственный метод `acceptNode()`, который возвращает значение `NodeFilter.FILTER_ACCEPT`, если конкретный узел нужно посетить, или `NodeFilter.FILTER_SKIP` в противном случае. Поскольку тип `NodeFilter` является абстрактным, создать его экземпляр невозможно. Вместо этого просто создайте объект с методом `acceptNode()` и передайте его в метод `createNodeIterator()`. Например, следующий код отфильтровывает элементы `<p>`:

```
let filter = {
    acceptNode(node) {
        return node.tagName.toLowerCase() == "p" ?
               NodeFilter.FILTER_ACCEPT :
               NodeFilter.FILTER_SKIP;
    }
};

let iterator = document.createNodeIterator(root, NodeFilter.SHOW_ELEMENT,
                   filter, false);
```

Третьим аргументом также может быть функция, аналогичная методу `acceptNode()`:

```
let filter = function(node) {
         return node.tagName.toLowerCase() == "p" ?
                NodeFilter.FILTER_ACCEPT :
                NodeFilter.FILTER_SKIP;
};

let iterator = document.createNodeIterator(root, NodeFilter.SHOW_ELEMENT,
                   filter, false);
```

Этот формат используется в JavaScript чаще других, потому что он проще и больше похож на другой JS-код. Если фильтр не требуется, третий аргумент должен иметь значение `null`.

Создать простой объект NodeIterator, посещающий узлы всех типов, можно следующим образом:

```
let iterator = document.createNodeIterator(document, NodeFilter.SHOW_ALL,
                   null, false);
```

Два основных метода объекта NodeIterator называются nextNode() и previousNode(). Первый из них делает один шаг вперед при обходе DOM-поддерева в глубину, а второй возвращается на шаг назад. При создании объекта NodeIterator его внутренний указатель указывает на корневой узел, который и возвращается при первом вызове nextNode(). Когда обход достигает последнего узла в DOM-поддереве, метод nextNode() возвращает null. Метод previousNode() работает подобным образом. При завершении обхода он возвращает корневой узел, а при следующем вызове — значение null.

Рассмотрим такой HTML-код:

```
<div id="div1">
    <p><b>Hello</b> world!</p>
    <ul>
        <li>List item 1</li>
        <li>List item 2</li>
        <li>List item 3</li>
    </ul>
</div>
```

Предположим, нам нужно обойти все элементы внутри элемента . Это можно сделать следующим образом:

```
let div = document.getElementById("div1");
let iterator = document.createNodeIterator(div, NodeFilter.SHOW_ELEMENT,
                   null, false);

let node = iterator.nextNode();
while (node !== null) {
    console.log(node.tagName);    // вывод имени тега
    node = iterator.nextNode();
}
```

Первый вызов метода nextNode() в этом примере возвращает элемент <p>. Поскольку при достижении конца DOM-поддерева метод nextNode() возвращает null, мы используем это условие в цикле while. Если запустить этот код, появятся оповещения со следующими именами тегов:

```
DIV
P
B
UL
LI
LI
LI
```

Если при обходе нужно возвратить только элементы <li>, это можно сделать с помощью фильтра:

```
let div = document.getElementById("div1");
let filter = function(node) {
    return node.tagName.toLowerCase() == "li" ?
        NodeFilter.FILTER_ACCEPT :
        NodeFilter.FILTER_SKIP;
};

let iterator = document.createNodeIterator(div, NodeFilter.SHOW_ELEMENT,
                   filter, false);

let node = iterator.nextNode();
while (node !== null) {
    console.log(node.tagName);    // вывод имени тега
    node = iterator.nextNode();
}
```

В этом примере итератор возвращает только элементы `<li>`.

При работе с DOM-структурой методы `nextNode()` и `previousNode()` используют внутренний указатель объекта `NodeIterator`, который отражает любые изменения структуры.

Тип TreeWalker

`TreeWalker` — это улучшенная версия типа `NodeIterator`, которая поддерживает весь его функционал, включая методы `nextNode()` и `previousNode()`, и расширяет его следующими методами для обхода DOM-структуры в разных направлениях:

- `parentNode()` — переходит к родительскому узлу текущего узла;
- `firstChild()` — переходит к первому дочернему узлу текущего узла;
- `lastChild()` — переходит к последнему дочернему узлу текущего узла;
- `nextSibling()` — переходит к следующему узлу текущего уровня;
- `previousSibling()` — переходит к предыдущему узлу текущего уровня.

Создать объект `TreeWalker` можно методом `document.createTreeWalker()`, который принимает те же аргументы, что и метод `document.createNodeIterator()`: корневой узел обхода, типы отображаемых узлов, фильтр и логическое значение, указывающее, нужно ли развертывать ссылки на сущности. Благодаря такому подобию объект `TreeWalker` всегда можно использовать вместо `NodeIterator`, например:

```
let div = document.getElementById("div1");
let filter = function(node) {
    return node.tagName.toLowerCase() == "li" ?
        NodeFilter.FILTER_ACCEPT :
        NodeFilter.FILTER_SKIP;
};

let walker = document.createTreeWalker(div, NodeFilter.SHOW_ELEMENT,
                   filter, false);
```

```
let node = iterator.nextNode();
while (node !== null) {
    console.log(node.tagName);    // вывод имени тега
    node = iterator.nextNode();
}
```

Одно из различий этих типов заключается в значениях, которые может возвращать фильтр. Кроме `NodeFilter.FILTER_ACCEPT` и `NodeFilter.FILTER_SKIP` доступно также значение `NodeFilter.FILTER_REJECT`. При использовании с объектом `NodeIterator` значения `NodeFilter.FILTER_SKIP` и `NodeFilter.FILTER_REJECT` указывают на необходимость пропустить узел. При использовании с объектом `TreeWalker` значение `NodeFilter.FILTER_SKIP` указывает на необходимость пропустить узел и перейти к следующему узлу в поддереве, тогда как `NodeFilter.FILTER_REJECT` пропускает узел и все его поддерево. Например, если в предыдущем фрагменте кода возвращать из фильтра значение `NodeFilter.FILTER_REJECT`, а не `NodeFilter.FILTER_SKIP`, никакие узлы посещены не будут. Дело в том, что первым возвращаемым элементом является `<div>`, у которого нет тега с именем `"li"`, поэтому для него возвращается значение `NodeFilter.FILTER_REJECT`, указывающее, что все поддерево должно быть пропущено. Поскольку элемент `<div>` является корнем обхода, на этом обход завершается.

Конечно, истинная мощь объекта `TreeWalker` — это возможность перемещения по DOM-структуре. С его помощью можно обойти элементы `<li>` без использования фильтра:

```
let div = document.getElementById("div1");
let walker = document.createTreeWalker(div, NodeFilter.SHOW_ELEMENT, null,
                                       false);

walker.firstChild();     // переход к элементу <p>
walker.nextSibling();    // переход к элементу <ul>

let node = walker.firstChild();    // переход к первому элементу <li>
while (node !== null) {
    console.log(node.tagName);
    node = walker.nextSibling();
}
```

Поскольку нам известно, где в документе находятся элементы `<li>`, можно добраться до них, вызвав метод `firstChild()` для перехода к элементу `<p>`, метод `nextSibling()` для перехода к элементу `<ul>` и метод `firstChild()` для перехода к первому элементу `<li>`. Имейте в виду, что из-за второго аргумента, переданного в метод `createTreeWalker()`, объект `TreeWalker` в этом примере возвращает только элементы. Затем метод `nextSibling()` перебирает в цикле элементы `<li>`, возвращая при их исчерпании значение `null`.

У типа `TreeWalker` также есть свойство `currentNode`, указывающее последний узел, возвращенный любым из методов обхода. Задав это свойство, можно изменить место возобновления обхода, например:

```
let node = walker.nextNode();
console.log(node === walker.currentNode);   // true
walker.currentNode = document.body;         // изменение места возобновления обхода
```

В сравнении с типом `NodeIterator` тип `TreeWalker` обеспечивает больше гибкости при обходе DOM.

ДИАПАЗОНЫ

Еще большего контроля над страницей можно добиться с помощью так называемых диапазонов, определенных в модуле DOM Level 2 Traversal and Range. Диапазон можно использовать для выделения части документа независимо от границ узлов (выделение выполняется неявно и незаметно для пользователя). Диапазоны полезны, если обычные DOM-манипуляции недостаточно конкретны.

Диапазоны в DOM

В DOM Level 2 для типа `Document` определен метод `createRange()`, который принадлежит объекту `document`. DOM-диапазон можно создать методом `createRange()`:

```
let range = document.createRange();
```

Подобно узлам, новый диапазон связывается с документом, для которого был создан. Это означает, что его можно использовать для неявного выделения частей только этого, но не других документов. Как только диапазон создан и настроен, можно выполнять различные операции с его содержимым, что делает возможным более тонкое манипулирование базовым DOM-деревом.

Каждый диапазон представляется экземпляром типа `Range`, у которого есть ряд свойств и методов. Указанные далее свойства определяют расположение диапазона в документе.

- `startContainer` — узел, в котором начинается диапазон (родительский узел первого узла в выделенном фрагменте).
- `startOffset` — смещение начала диапазона в узле `startContainer`. Если узел `startContainer` является текстовым, узлом комментария или узлом `CData`, свойство `startOffset` указывает количество знаков, пропускаемых перед началом диапазона, иначе — индекс первого дочернего узла в диапазоне.
- `endContainer` — узел, в котором завершается диапазон (родительский узел последнего узла в выделенном фрагменте).
- `endOffset` — смещение конца диапазона в узле `endContainer` (действуют те же правила, что и для узла `startOffset`).
- `commonAncestorContainer` — наиболее глубокий узел в документе, являющийся предком `startContainer` и `endContainer`.

Эти свойства заполняются при определении диапазона в конкретной позиции в документе.

Простое выделение с помощью DOM-диапазонов

Самый простой способ выделить часть документа с помощью диапазона — это использовать метод `selectNode()` или `selectNodeContents()`. Каждый из них принимает DOM-узел и заполняет диапазон информацией из этого узла. Метод `selectNode()` выделяет весь узел вместе с дочерними узлами, а метод `selectNodeContents()` — только дочерние узлы. Рассмотрим, например, такой HTML-код:

```
<!DOCTYPE html>
<html>
    <body>
        <p id="p1"><b>Hello</b> world!</p>
    </body>
</html>
```

Для доступа к этому фрагменту можно использовать следующий JS-код:

```
let range1 = document.createRange(),
    range2 = document.createRange(),
    p1 = document.getElementById("p1");
range1.selectNode(p1);
range2.selectNodeContents(p1);
```

Два диапазона в этом примере содержат разные разделы документа: `range1` — элемент `<p>` и все его дочерние элементы, а `range2` — элемент `<b>` и текстовые узлы `"Hello"` и `" world!"` (рис. 16.6).

Рис. 16.6

При вызове метода `selectNode()` свойства `startContainer`, `endContainer` и `commonAncestorContainer` содержат родительский узел узла, переданного в метод (`document.body` в данном случае). Значение `startOffset` равно индексу конкретного узла в коллекции `childNodes` родительского узла (1 в данном примере, потому что браузеры, соответствующие требованиям DOM, интерпретируют свободное пространство как текстовый узел), а значение `endOffset` на единицу превышает `startOffset`, потому что выделен только один узел.

При вызове метода `selectNodeContents()` свойства `startContainer`, `endContainer` и `commonAncestorContainer` содержат узел, переданный в метод, или `<p>` в нашем случае. Свойство `startOffset` всегда равно 0, так как диапазон начинается с первого

дочернего узла опорного узла, а свойство `endOffset` равно количеству дочерних узлов (`node.childNodes.length`), или 2 в примере.

Для более точного управления выделением узлов можно использовать перечисленные методы.

- `setStartBefore(`*опорныйУзел*`)` — задает начальную точку диапазона перед опорным узлом, который становится первым узлом в диапазоне. Свойству `startContainer` назначается родительский узел опорного узла, а свойству `startOffset` — индекс опорного узла в коллекции `childNodes` его родительского узла.
- `setStartAfter(`*опорныйУзел*`)` — задает начальную точку диапазона после опорного узла, который не включается в диапазон, а первым выделяемым узлом становится следующий узел того же уровня. Свойству `startContainer` назначается родительский узел опорного узла, а свойству `startOffset` — индекс опорного узла в коллекции `childNodes` его родительского узла, увеличенный на единицу.
- `setEndBefore(`*опорныйУзел*`)` — задает конечную точку диапазона перед опорным узлом, который не включается в диапазон. Свойству `endContainer` назначается родительский узел опорного узла, а свойству `endOffset` — индекс опорного узла в коллекции `childNodes` его родительского узла.
- `setEndAfter(`*опорныйУзел*`)` — задает конечную точку диапазона перед опорным узлом, который становится последним узлом в диапазоне. Свойству `endContainer` назначается родительский узел опорного узла, а свойству `endOffset` — индекс опорного узла в коллекции `childNodes` его родительского узла, увеличенный на единицу.

При использовании всех этих методов свойства задаются автоматически, но при желании их можно назначать вручную для более точного выделения диапазонов.

Сложное выделение с помощью DOM-диапазонов

Для создания сложных диапазонов нужно использовать методы `setStart()` и `setEnd()`, которые принимают два аргумента: опорный узел и смещение. В методе `setStart()` опорный узел назначается свойству `startContainer`, а смещение — свойству `startOffset`. В методе `setEnd()` опорный узел назначается свойству `endContainer`, а смещение — свойству `endOffset`.

С помощью этих методов можно имитировать методы `selectNode()` и `selectNodeContents()`, например:

```
let range1 = document.createRange(),
    range2 = document.createRange(),
    p1 = document.getElementById("p1"),
    p1Index = -1,
    i, len;
for (i=0, len=p1.parentNode.childNodes.length; i < len; i++) {
    if (p1.parentNode.childNodes[i] === p1) {
        p1Index = i;
```

```
            break;
        }
}

range1.setStart(p1.parentNode, p1Index);
range1.setEnd(p1.parentNode, p1Index + 1);
range2.setStart(p1, 0);
range2.setEnd(p1, p1.childNodes.length);
```

Заметьте, что для выделения узла `p1` с помощью диапазона `range1` нужно сначала определить его индекс в коллекции `childNodes` родительского узла. Для выделения содержимого узла с помощью диапазона `range2` вычислять что-либо не требуется — можно использовать в методах `setStart()` и `setEnd()` значения, предлагаемые по умолчанию. Хотя имитировать методы `selectNode()` и `selectNodeContents()` может быть полезно, реальная мощь методов `setStart()` и `setEnd()` — это возможность частичного выделения узлов.

Предположим, что нам нужно выделить в предыдущем HTML-коде текст с букв `"llo"` в слове `"Hello"` до буквы `"o"` в слове `"world!"`. Сделать это довольно легко. Сначала нужно получить ссылки на соответствующие узлы:

```
let p1 = document.getElementById("p1"),
helloNode = p1.firstChild.firstChild,
worldNode = p1.lastChild
```

Текстовый узел `"Hello"` приходится внуком узлу `<p>` и является дочерним по отношению к узлу `<b>`, так что его можно получить с помощью свойства `p1.firstChild.firstChild` (свойство `p1.firstChild` возвращает узел `<b>`). Текстовый узел `"world!"` — это второй (и последний) дочерний узел `<p>`, которому соответствует свойство `p1.lastChild`. Получив узлы, нужно создать диапазон и задать его границы:

```
let range = document.createRange();
range.setStart(helloNode, 2);
range.setEnd(worldNode, 3);
```

Поскольку выделение должно начаться после буквы `"e"` в слове `"Hello"`, мы передаем в метод `setStart()` узел `helloNode` и смещение 2 (буква `"H"` находится в позиции 0). Чтобы указать конец диапазона, мы передаем в метод `setEnd()` узел `worldNode` и смещение 3, задающее первый знак, который выделять не следует, то есть `"r"` (позиции 0 соответствует пробел). Содержимое диапазона показано на рис. 16.7.

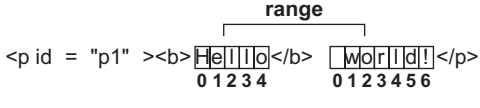

Рис. 16.7

Узлы `helloNode` и `worldNode` являются текстовыми, поэтому они становятся свойствами `startContainer` и `endContainer`, а свойства `startOffset` и `endOffset` определяют смещения в тексте, а не дочерние узлы (это происходит, если в методы передается

элемент). Свойству `commonAncestorContainer` назначается элемент `<p>` — первый общий предок обоих узлов.

Разумеется, выделение фрагментов документа полезно, только если с ними можно что-нибудь сделать. Сейчас мы это обсудим.

Работа с контентом DOM-диапазона

При создании диапазона неявно создается узел фрагмента документа, к которому присоединяются все узлы в выделенной области. Чтобы это было возможно, содержимое диапазона должно быть синтаксически правильным. В предыдущем примере диапазон определяет недопустимую DOM-структуру, потому что выделение начинается в одном текстовом узле и завершается в другом. Однако диапазоны распознают отсутствие открывающих и закрывающих тегов и могут воссоздавать допустимую DOM-структуру.

В предыдущем примере в выделенном фрагменте отсутствует открывающий тег `<b>`, поэтому диапазон динамически добавляет его вместе с закрывающим тегом `</b>` для строки `"He"`, заменяя DOM-структуру следующим фрагментом:

```
<p><b>He</b><b>llo</b> world!</p>
```

Кроме того, текстовый узел `" world!"` разделяется на два, из которых первый содержит символы `" wo"`, а второй — `"rld!"`. Итоговое DOM-дерево вместе с содержимым фрагмента документа показано на рис. 16.8.

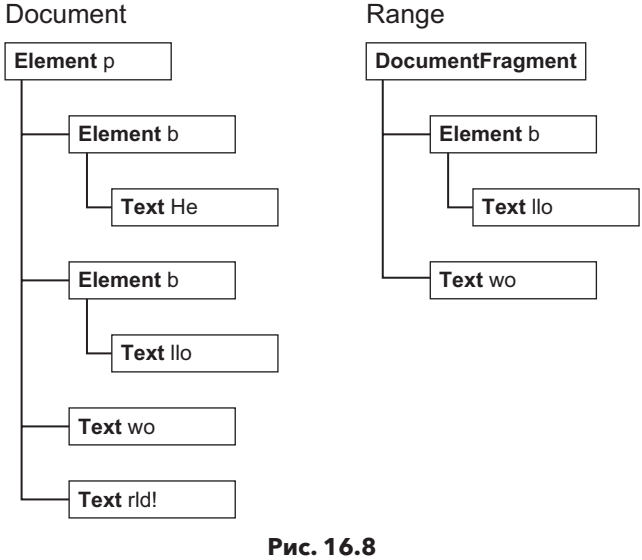

Рис. 16.8

После создания диапазона можно выполнять различные операции над его содержимым (имейте в виду, что все узлы во внутреннем фрагменте документа диапазона просто указывают на узлы в документе).

Первый метод, `deleteContents()`, просто удаляет содержимое диапазона из документа, например:

```
let p1 = document.getElementById("p1"),
    helloNode = p1.firstChild.firstChild,
    worldNode = p1.lastChild,
    range = document.createRange();

range.setStart(helloNode, 2);
range.setEnd(worldNode, 3);

range.deleteContents();
```

Этот код дает следующий результат:

```
<p><b>He</b>rld!</p>
```

Поскольку при выделении диапазона DOM-структура была подкорректирована, она остается синтаксически правильной даже после удаления содержимого.

Метод `extractContents()` похож на `deleteContents()` тем, что он также удаляет выделенный диапазон из документа. Разница между ними в том, что `extractContents()` возвращает соответствующий диапазону фрагмент документа, что позволяет вставить его в другое место, например:

```
let p1 = document.getElementById("p1"),
    helloNode = p1.firstChild.firstChild,
    worldNode = p1.lastChild,
    range = document.createRange();

range.setStart(helloNode, 2);
range.setEnd(worldNode, 3);

let fragment = range.extractContents();
p1.parentNode.appendChild(fragment);
```

Этот код извлекает фрагмент и добавляет его в конец элемента `<body>` (помните, что при передаче фрагмента документа в метод `appendChild()` добавляются только его дочерние узлы, но не сам фрагмент). В итоге получается такой HTML-код:

```
<p><b>He</b>rld!</p>
<b>llo</b> wo
```

С помощью метода `cloneContents()` можно создать копию диапазона, оставив его на месте. Скопированный фрагмент затем можно вставить в другое место:

```
let p1 = document.getElementById("p1"),
    helloNode = p1.firstChild.firstChild,
    worldNode = p1.lastChild,
    range = document.createRange();

range.setStart(helloNode, 2);
range.setEnd(worldNode, 3);

let fragment = range.cloneContents();
p1.parentNode.appendChild(fragment);
```

Этот метод очень похож на `extractContents()`, потому что оба они возвращают фрагмент документа, однако в случае метода `cloneContents()` он содержит копии узлов в диапазоне, а не фактические узлы. Предыдущий фрагмент генерирует следующий HTML-код:

```
<p><b>Hello</b> world!</p>
<b>llo</b> wo
```

Важно отметить, что из-за разделения узлов синтаксис документа не станет правильным, пока не будет вызван один из этих методов. Исходный HTML-документ остается неизменным до изменения DOM-структуры.

Вставка контента DOM-диапазона

С помощью диапазонов можно не только удалять и клонировать контент. Так, метод `insertNode()` позволяет вставить узел в начало выделенного диапазона. Предположим, например, что нам нужно вставить перед HTML-кодом предыдущего примера следующий HTML-код:

```
<span style="color: red">Inserted text</span>
```

Это можно сделать следующим образом:

```
let p1 = document.getElementById("p1"),
    helloNode = p1.firstChild.firstChild,
    worldNode = p1.lastChild,
    range = document.createRange();

range.setStart(helloNode, 2);
range.setEnd(worldNode, 3);

let span = document.createElement("span");
span.style.color = "red";
span.appendChild(document.createTextNode("Inserted text"));
range.insertNode(span);
```

В результате будет сгенерирован такой фрагмент:

```
<p id="p1"><b>He<span style="color: red">Inserted text</span>llo</b> world</p>
```

Обратите внимание, что тег `<span>` вставляется непосредственно перед строкой `"llo"` в слове `"Hello"`, которая является первой частью выделенного диапазона. Заметьте также, что элементов `<b>` в исходном HTML-коде не стало больше или меньше, потому что мы не используем ни один из методов, упомянутых в предыдущем разделе. Эту методику можно применять для вставки полезных сведений, например для добавления изображений рядом со ссылками, которые открывают новые окна.

Кроме содержимого диапазона можно вставить в документ и контент, охватывающий диапазон. Для этого используется метод `surroundContents()`, принимающий узел, в который должен быть заключен диапазон. За кулисами при этом происходит следующее.

1. Содержимое диапазона извлекается.
2. Указанный узел вставляется в документ там, где был диапазон.
3. Содержимое фрагмента документа добавляется в новый узел.

Этот метод полезен для выделения слов на веб-странице, например:

```
let p1 = document.getElementById("p1"),
    helloNode = p1.firstChild.firstChild,
    worldNode = p1.lastChild,
    range = document.createRange();

range.selectNode(helloNode);

let span = document.createElement("span");
span.style.backgroundColor = "yellow";
range.surroundContents(span);
```

Этот код выделяет диапазон желтым фоновым цветом, при этом генерируется следующий HTML-код:

```
<p><b><span style="background-color:yellow">Hello</span></b> world!</p>
```

Чтобы можно было вставить элемент `<span>`, диапазон должен содержать полноценный DOM-фрагмент, но не частично выделенные узлы.

Свертывание DOM-диапазона

Если диапазон не выделяет никакую часть документа, говорят, что он *свернут* (collapsed). Свертывание диапазона напоминает работу текстового поля. Если в нем есть текст, вы может выделить целое слово с помощью мыши. Однако если щелкнуть левой кнопкой мыши еще раз, выделение исчезнет, а курсор установится между двумя буквами. При свертывании диапазона он располагается между частями документа в начале выделенного фрагмента или в его конце. На рис. 16.9 показано, что происходит при свертывании диапазона.

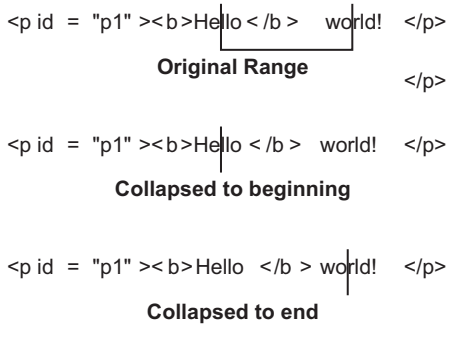

Рис. 16.9

Свернуть диапазон можно методом `collapse()`, который принимает единственный аргумент — логическое значение, указывающее, в какую сторону нужно свернуть диапазон. Если оно равно `true`, диапазон свертывается к началу, иначе — к концу. Узнать, не свернут ли уже диапазон, можно с помощью свойства `collapsed`:

```
range.collapse(true);            // свертывание к началу
console.log(range.collapsed);    // "true"
```

Проверив, свернут ли диапазон, можно узнать, находятся ли два узла рядом. Возьмем для примера следующий HTML-код:

```
<p id="p1">Paragraph 1</p><p id="p2">Paragraph 2</p>
```

Если вы не знаете точную разметку подобного кода (например, если он был сгенерирован автоматически), можно попробовать создать диапазон:

```
let p1 = document.getElementById("p1"),
    p2 = document.getElementById("p2"),
    range = document.createRange();
range.setStartAfter(p1);
range.setStartBefore(p2);
console.log(range.collapsed);        // "true"
```

В данном случае созданный диапазон свернут, потому что между узлами `p1` и `p2` ничего нет.

Сравнение DOM-диапазонов

Если диапазонов более одного, с помощью метода `compareBoundaryPoints()` можно определить, есть ли у них общие границы (начальная или конечная). Этот метод принимает два аргумента: сравниваемый диапазон и способ сравнения. Вторым аргументом может быть одна из следующих констант:

- `Range.START_TO_START (0)` — сравнивается начало первого диапазона с началом второго;
- `Range.START_TO_END (1)` — сравнивается начало первого диапазона с концом второго;
- `Range.END_TO_END (2)` — сравнивается конец первого диапазона с концом второго;
- `Range.END_TO_START (3)` — сравнивается конец первого диапазона с началом второго.

Метод `compareBoundaryPoints()` возвращает –1, если граница первого диапазона предшествует границе второго; 0, если границы совпадают, и 1, если граница первого диапазона следует за границей второго, например:

```
let range1 = document.createRange();
let range2 = document.createRange();
let p1 = document.getElementById("p1");

range1.selectNodeContents(p1);
```

```
range2.selectNodeContents(p1);
range2.setEndBefore(p1.lastChild);

console.log(range1.compareBoundaryPoints(Range.START_TO_START, range2));    // 0
console.log(range1.compareBoundaryPoints(Range.END_TO_END, range2));        // 1
```

В этом коде начала двух диапазонов совпадают, потому что мы создаем их одинаковыми вызовами метода `selectNodeContents()`; соответственно, в первый раз метод `compareBoundaryPoints()` возвращает 0. Во второй раз он возвращает 1, потому что после изменения конца второго диапазона методом `setEndBefore()` он предшествует концу первого диапазона (рис. 16.10).

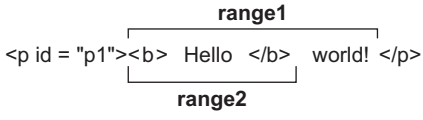

Рис. 16.10

Клонирование DOM-диапазонов

Диапазон можно клонировать методом `cloneRange()`, который создает его точную копию:

```
let newRange = range.cloneRange();
```

Новый диапазон содержит все свойства исходного, а изменения его конечных точек никак не влияют на оригинал.

Очистка

Завершив работу с диапазоном, следует вызвать метод `detach()`, который разрывает его связь с документом, а затем присвоить диапазону значение `null`, чтобы сборщик мусора мог возвратить память системе:

```
range.detach();         // разрыв связи диапазона с документом
range = null;           // уничтожение ссылки
```

Это оптимальный способ завершения работы с диапазоном. Когда его связь с документом разорвана, использовать диапазон нельзя.

ИТОГИ

Спецификации DOM Level 2 определяют несколько модулей, расширяющих функционал DOM Level 1. Модуль DOM Level 2 Core добавляет к некоторым DOM-типам методы поддержки пространств имен, но эти изменения задействуются только при работе с XML- и XHTML-документами и не используются в HTML-коде.

Новые возможности, не связанные с XML-пространствами имен, включают методы для программного создания экземпляров `Document` и `DocumentType`.

Модуль DOM Level 2 Style описывает, как работать со стилями элементов.

- У каждого элемента есть объект `style`, с помощью которого можно определять и изменять встроенные стили.
- Для определения вычисляемого стиля элемента, в том числе всех применимых к нему CSS-правил, можно использовать метод `getComputedStyle()`.
- Для доступа к таблицам стилей можно также использовать коллекцию `document.styleSheets`.

Модуль DOM Level 2 Traversals and Range определяет способы взаимодействия с DOM-структурой.

- С помощью объектов `NodeIterator` и `TreeWalker` можно обходить DOM-дерево в глубину.
- Интерфейс более простого типа `NodeIterator` позволяет перемещаться лишь вперед и назад на один шаг. Интерфейс `TreeWalker` поддерживает перемещение по DOM-структуре в этих и во всех других направлениях, включая переходы к родительским, одноуровневым и дочерним узлам.
- Диапазоны позволяют выделять части DOM-структуры для выполнения каких-либо операций над ними.
- С помощью диапазонов можно удалять выделенные части содержимого с сохранением синтаксически правильной структуры документа и клонировать их.

17

События

- Распространение событий
- Обработчики событий
- Типы событий

Взаимодействие JavaScript с HTML осуществляется посредством *событий* (events), которые сигнализируют, что в документе или окне браузера произошло что-то, что может нас заинтересовать. На события можно подписаться с помощью *слушателей* (listeners), называемых также обработчиками, которые выполняются только при возникновении события. Эта модель, в традиционном программировании реализуемая с помощью *паттерна Наблюдатель* (observer pattern), позволяет ослабить связь между поведением страницы (определенным на JavaScript) и ее видом (описанным с помощью HTML и CSS).

События, впервые реализованные в Internet Explorer 3 и Netscape Navigator 2, позволяли частично обрабатывать в браузере формы еще до отправки данных серверу. Ко времени выпуска Internet Explorer 4 и Netscape 4 каждый браузер предоставлял похожие, но разные API, которые просуществовали еще несколько поколений. В DOM Level 2 была предпринята первая попытка стандартизировать API событий DOM. Основные части спецификации DOM Level 2 Events реализованы во всех современных браузерах. Internet Explorer 8 стал последним популярным браузером, в котором использовалась исключительно фирменная система событий.

Система событий браузера непроста. Хотя спецификация DOM Level 2 Events реализована во всех основных браузерах, она охватывает не все типы событий. BOM поддерживает собственные события, которые связаны с DOM-событиями запутанными отношениями из-за длительного отсутствия документации (хотя в HTML5 эта ситуация отчасти исправлена). Расширение API событий DOM в DOM Level 3 только добавило разработчикам проблем. Иногда работать с событиями

сравнительно просто, а иногда крайне сложно, но без понимания базовых концепций в любом случае не обойтись.

РАСПРОСТРАНЕНИЕ СОБЫТИЙ

Приступив к созданию веб-браузеров четвертого поколения (Internet Explorer 4 и Netscape Communicator 4), разработчики столкнулись с интересным вопросом: как понять, какой части страницы принадлежит то или иное конкретное событие? Чтобы понять проблему, представьте концентрические окружности на листе бумаги. Пометив центр одной из них, вы автоматически пометите центры и всех остальных окружностей. Разработчики из обеих групп смотрели на проблему точно так же. Когда пользователь щелкает на элементе управления, рассудили они, он щелкает также на его контейнере и на странице в целом. Однако в вопросе *распространения событий* (event flow), то есть порядка, в котором события поступают на страницу, их взгляды разошлись. В Internet Explorer было реализовано так называемое всплытие событий, а в Netscape Communicator — перехват событий.

Всплытие событий

При *всплытии событий* (event bubbling), реализованном в Internet Explorer, событие срабатывает у наиболее конкретного элемента (самого глубокого узла в дереве документа), а затем поднимается по иерархии до наименее конкретного узла (самого документа). Рассмотрим следующую HTML-страницу:

```
<!DOCTYPE html>
<html>
<head>
    <title>Event Bubbling Example</title>
</head>
<body>
    <div id="myDiv">Click Me</div>
</body>
</html>
```

Если щелкнуть на элементе `<div>` этой страницы, событие `click` будет возникать для элементов в следующем порядке:

1. `<div>`.
2. `<body>`.
3. `<html>`.
4. `document`.

Первым делом событие `click` возникнет для элемента `<div>`, на котором произошел щелчок, а затем — для каждого последующего узла в DOM-иерархии вплоть до объекта `document`. Этот процесс показан на рис. 17.1.

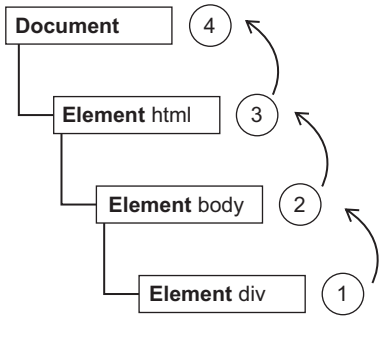

Рис. 17.1

Всплытие событий поддерживают все современные браузеры, хотя его реализации немного различаются. В Internet Explorer 5.5 и более ранних версий пропускается элемент `<html>` (после элемента `<body>` событие возникает для элемента `document`). В современных браузерах всплытие продолжается до объекта `window`.

Перехват событий

В Netscape Communicator вместо всплытия был реализован *перехват событий* (event capturing), при котором событие сначала возникает для наименее конкретного узла, а в конце — для наиболее конкретного. На самом деле этот механизм был разработан для того, чтобы событие можно было обработать, пока оно еще не достигло своего целевого элемента. В этом случае для предыдущей страницы при щелчке на элементе `<div>` событие `click` поочередно возникает для следующих элементов:

1. `document`.
2. `<html>`.
3. `<body>`.
4. `<div>`.

При перехвате событие `click` генерируется для документа, а затем продолжает спуск по DOM-дереву до своей фактической цели — элемента `<div>` (рис. 17.2).

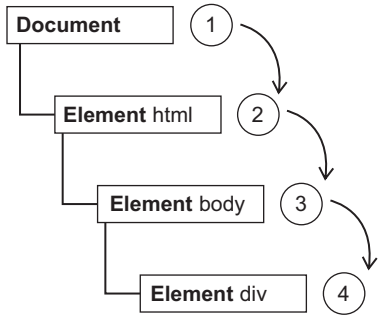

Рис. 17.2

Хотя первоначально эта модель распространения событий использовалась только в Netscape Communicator, теперь ее поддерживают все современные браузеры. Все они начинают перехват на уровне объекта `window`, хотя в спецификации DOM Level 2 Events сказано, что первым должен быть объект `document`.

Перехват событий не поддерживается в старых браузерах, поэтому рекомендуется применять его только в специальных ситуациях и использовать в основном всплытие событий.

Распространение DOM-событий

Процесс распространения событий, описанный в DOM Level 2 Events, включает три этапа: фаза перехвата, фаза цели и фаза всплытия. На первом этапе событие можно перехватить, если это требуется. Затем событие обрабатывается целевым элементом, а после этого всплывает, что позволяет выполнить какие-то заключительные действия в ответ на событие. В нашем прежнем примере щелчок на элементе `<div>` инициирует процесс распространения события, показанный на рис. 17.3.

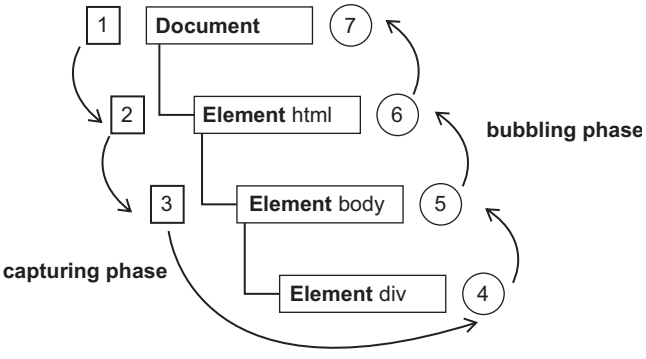

Рис. 17.3

В ходе распространения DOM-событий фактический целевой элемент (`<div>`) не получает его на этапе перехвата. Иначе говоря, событие распространяется от объекта `document` только до элемента `<body>`. После этого начинается следующий этап, когда событие генерируется для целевого элемента. В контексте обработки событий (см. далее) этот этап считается частью фазы всплытия. Затем событие всплывает, возвращаясь к узлу документа.

Большинство браузеров, поддерживающих распространение DOM-событий, имеют особенность. Хотя в спецификации DOM Level 2 Events сказано, что во время перехвата события оно не достигает целевого элемента, современные браузеры генерируют событие для целевого элемента на этапе перехвата. Это предоставляет еще одну возможность обработки события на уровне его целевого элемента.

ПРИМЕЧАНИЕ Распространение DOM-событий поддерживается во всех современных браузерах кроме Internet Explorer 8 и более ранних версий.

ОБРАБОТЧИКИ СОБЫТИЙ

События соответствуют определенным действиям, которые выполняет пользователь или сам браузер, и имеют имена вроде `click`, `load` и `mouseover`. Функция, выполняемая в ответ на событие, называется *обработчиком события* (event handler), или *слушателем события* (event listener). Имена таких функций начинаются с префикса `"on"`: например, обработчик события `click` имеет имя `onclick`, а обработчик события `load` называется `onload`. Назначать обработчики событиям можно несколькими способами.

HTML-обработчики событий

Каждому событию, поддерживаемому конкретным элементом, можно назначить обработчик, указав одноименный HTML-атрибут. Значением атрибута должен

быть JS-код. Например, для обработки щелчка на кнопке можно использовать следующий код:

```
<input type="button" value="Click Me" onclick="console.log('Clicked')" />
```

При щелчке на этой кнопке в консоли появится сообщение, заданное как значение атрибута `onclick`. Из-за того что значением атрибута является JS-код, в нем нужно экранировать знаки, относящиеся к синтаксису HTML, такие как амперсанд, двойные кавычки, меньше и больше. Именно по этой причине в примере используются одинарные кавычки вместо двойных. Для применения двойных кавычек нужно было бы изменить код:

```
<input type="button" value="Click Me"
       onclick="console.log(&quot;Clicked&quot;)" />
```

Обработчик события, определенный в HTML-коде, может также вызывать сценарий, находящийся в другом месте:

```
<script>
    function showMessage() {
        console.log("Hello world!");
    }
</script>
<input type="button" value="Click Me" onclick="showMessage()" />
```

Здесь при щелчке на кнопке вызывается функция `showMessage()`, которая определена в отдельном элементе `<script>` и может даже находиться во внешнем файле. Обработчикам событий доступен весь код в глобальной области видимости.

Обработчики событий, назначенные таким способом, имеют уникальные особенности. Так, для каждого из них создается функция, в которую заключается значение атрибута. У этой функции есть специальная локальная переменная `event`, или объект события (см. далее):

```
<!-- выводит "click" -->
<input type="button" value="Click Me" onclick="console.log(event.type)">
```

Это обеспечивает доступ к объекту события, благодаря чему можно не определять его самостоятельно и не извлекать из списка аргументов функции-контейнера.

Значение `this` внутри этой функции указывает на целевой элемент события, например:

```
<!-- выводит "Click Me" -->
<input type="button" value="Click Me" onclick="console.log(this.value)">
```

Эта динамически создаваемая функция интересна еще и тем, как она наращивает цепочку областей видимости. В ней доступны члены объекта `document` и самого элемента, как если бы они были ее локальными переменными. Это возможно благодаря приращению цепочки областей видимости с помощью инструкции `with`:

```
function() {
    with(document) {
        with(this) {
            // значение атрибута
        }
    }
}
```

Это означает, что в обработчике события можно без проблем использовать его собственные свойства. Так, следующий код эквивалентен предыдущему:

```
<!-- выводит "Click Me" -->
<input type="button" value="Click Me" onclick="console.log(value)">
```

Если элементом является элемент ввода формы, цепочка областей видимости также содержит запись родительского элемента формы:

```
function() {
    with(document) {
        with(this.form) {
            with(this) {
                // значение атрибута
            }
        }
    }
}
```

По сути, это позволяет обработчику события обращаться к другим членам той же формы, не ссылаясь на сам элемент формы, например:

```
<form method="post">
    <input type="text" name="username" value="">
    <input type="button" value="Echo Username"
           onclick="console.log(username.value)">
</form>
```

При щелчке на кнопке в этом примере выводится текст из текстового поля. Обратите внимание, что мы ссылаемся на `username` непосредственно.

Назначение обработчиков событий в HTML-коде имеет несколько недостатков. Первый — проблема времени: возможна ситуация, когда HTML-элемент появится на странице и пользователь начнет взаимодействовать с ним, прежде чем будет готов код обработчика события. Представьте, что в предыдущем примере функция `showMessage()` определена позже кнопки. Если пользователь щелкнет на кнопке до загрузки кода `showMessage()`, произойдет ошибка. По этой причине большинство HTML-обработчиков событий заключают в блоки `try-catch`, например:

```
<input type="button" value="Click Me"
       onclick="try{showMessage();}catch(ex) {}">
```

Если щелкнуть на этой кнопке до определения функции `showMessage()`, JavaScript-ошибка останется незамеченной, потому что будет перехвачена, прежде чем достигнет браузера.

Другим недостатком является то, что приращение цепочки областей видимости в функции обработчика событий может давать разные результаты в разных браузерах. Из-за того, что правила разрешения идентификаторов в интерпретаторах JavaScript немного различаются, доступ к неквалифицированным членам объектов может приводить к ошибкам.

Наконец, назначение обработчиков событий в HTML-коде усиливает связь HTML- и JS-кода. Если такой обработчик события нужно будет изменить, это может потребовать внесения изменений в двух местах вместо одного. Это главная причина того, что многие разработчики предпочитают реализовывать обработчики событий в JS-коде.

Обработчики событий DOM Level 0

Традиционный способ обработки событий в JavaScript включает назначение функции свойству обработчика события. Этот способ был представлен еще в браузерах четвертого поколения и до сих пор используется во всех современных браузерах благодаря простоте и широкой поддержке. Чтобы назначить обработчик события в JavaScript, нужно сначала получить ссылку на целевой объект.

У каждого элемента (а также у объектов `window` и `document`) есть свойства обработчиков событий, которые обычно имеют имена в нижнем регистре, например `onclick`. Для обработки события нужно назначить функцию-обработчик такому свойству:

```
let btn = document.getElementById("myBtn");
btn.onclick = function() {
    console.log("Clicked");
};
```

Этот код назначает кнопке обработчик события `click`. Имейте в виду, что обработчик события назначается только при выполнении этого кода, и если он располагается на странице после кода кнопки, какое-то время щелчки на кнопке могут срабатывать вхолостую.

Обработчик события, назначенный таким способом, считается методом элемента и выполняется в его области видимости, то есть переменная `this` указывает на элемент:

```
let btn = document.getElementById("myBtn");
btn.onclick = function() {
    console.log(this.id);        // "myBtn"
};
```

Этот код при нажатии на кнопку выводит ее идентификатор, для получения которого используется свойство `this.id`. Через переменную `this` в обработчике события доступны любые свойства и методы его элемента. Обработчики событий, добавленные таким способом, используются на этапе всплытия событий.

Чтобы удалить обработчик события, назначенный в стиле DOM Level 0, достаточно присвоить его свойству значение `null`, например:

```
btn.onclick = null;        // удаление обработчика события
```

После удаления обработчика при щелчке на кнопке больше ничего происходить не будет.

ПРИМЕЧАНИЕ Если обработчик события назначается в HTML-коде, свойству обработчика присваивается функция, содержащая код HTML-атрибута. Такой обработчик события также можно удалить, присвоив его свойству значение `null`.

Обработчики событий DOM Level 2

В DOM Level 2 Events для назначения и удаления обработчиков событий используются методы `addEventListener()` и `removeEventListener()`, которые есть у всех DOM-узлов. Каждый из них принимает три аргумента: имя обрабатываемого события, функцию-обработчик и логическое значение, указывающее, нужно ли вызывать обработчик события на этапе перехвата (`true`) или всплытия (`false`).

Например, добавить обработчик щелчка на кнопке можно следующим образом:

```
let btn = document.getElementById("myBtn");
btn.addEventListener("click", () => {
    console.log(this.id);
}, false);
```

Этот код назначает кнопке обработчик события `click`, который будет вызываться на этапе всплытия (поскольку последний аргумент имеет значение `false`). Как и при подходе DOM Level 0, обработчик события выполняется в области видимости элемента, к которому он подключен. Основное преимущество подхода DOM Level 2 в том, что событию можно назначить несколько обработчиков. Рассмотрим пример:

```
let btn = document.getElementById("myBtn");
btn.addEventListener("click", () => {
    console.log(this.id);
}, false);
btn.addEventListener("click", () => {
    console.log("Hello world!");
}, false);
```

Этот код определяет два обработчика щелчка на кнопке, которые будут срабатывать в порядке добавления, так что первое сообщение в консоли будет содержать идентификатор элемента, а второе — сообщение `"Hello world!"`.

Обработчик события, добавленный с помощью метода `addEventListener()`, можно удалить, только вызвав метод `removeEventListener()` с теми же аргументами. Это означает, что анонимные функции, добавляемые методом `addEventListener()`, удалить нельзя:

```
let btn = document.getElementById("myBtn");
btn.addEventListener("click", () => {
    console.log(this.id);
}, false);
```

```
// другой код

btn.removeEventListener("click", function() {    // не сработает!
    console.log(this.id);
}, false);
```

Здесь метод `addEventListener()` добавляет в качестве обработчика события анонимную функцию. На первый взгляд, в вызове `removeEventListener()` используются такие же аргументы, но на самом деле вторым аргументом является совершенно другая функция. Чтобы обработчик события можно было удалить методом `removeEventListener()`, его функция должна быть той же, что была передана в метод `addEventListener()`, например:

```
let btn = document.getElementById("myBtn");
let handler = function() {
    console.log(this.id);
};
btn.addEventListener("click", handler, false);

// другой код

btn.removeEventListener("click", handler, false);    // все в порядке!
```

Этот код работает без сюрпризов, потому что в этот раз аргументы методов `addEventListener()` и `removeEventListener()` действительно одинаковы.

В большинстве случаев события обрабатывают на этапе всплытия, потому что этот способ поддерживается шире. Обрабатывать событие на этапе перехвата имеет смысл, только если нужно перехватить его, прежде чем оно достигнет целевого элемента. Если это не требуется, лучше не перехватывать события.

Обработчики событий в Internet Explorer

В Internet Explorer реализованы аналогичные DOM Level 2 методы `attachEvent()` и `detachEvent()`. Они принимают в качестве аргументов имя обработчика события и его функцию. Поскольку в Internet Explorer 8 и более ранних версий поддерживается только всплытие событий, обработчики событий, добавленные методом `attachEvent()`, выполняются на этапе всплытия.

Добавить обработчик щелчка на кнопке можно следующим образом:

```
var btn = document.getElementById("myBtn");
btn.attachEvent("onclick", function() {
    console.log("Clicked");
});
```

Обратите внимание, что первым аргументом метода `attachEvent()` является строка `"onclick"`, а не `"click"`, как в DOM-методе `addEventListener()`.

Этот способ и подход DOM Level 0 в Internet Explorer различаются областью видимости обработчика события. При использовании DOM Level 0 значение `this` в обработчике события указывает на элемент, к которому он подключен, тогда

как обработчик, назначенный методом `attachEvent()`, выполняется в глобальном контексте и его переменная `this` указывает на `window`. Рассмотрим пример с использованием `attachEvent()`:

```
var btn = document.getElementById("myBtn");
btn.attachEvent("onclick", function() {
    console.log(this === window);        // true
});
```

Это различие важно иметь в виду при написании кроссбраузерного кода.

Подобно методу `addEventListener()`, метод `attachEvent()` позволяет назначить несколько обработчиков одному событию:

```
var btn = document.getElementById("myBtn");
btn.attachEvent("onclick", function() {
    console.log("Clicked");
});
btn.attachEvent("onclick", function() {
    console.log("Hello world!");
});
```

Здесь мы дважды вызываем метод `attachEvent()`, добавляя два обработчика щелчка на кнопке. Однако в отличие от DOM-метода, эти обработчики вызываются в обратном порядке. При щелчке на кнопке первым появится сообщение со строкой `"Hello world!"`, а вторым — со строкой `"Clicked"`.

Обработчики, добавленные методом `attachEvent()`, можно удалить, вызвав метод `detachEvent()` с такими же аргументами. Как и в DOM, это означает, что назначенные событиям анонимные функции удалить невозможно — метод `detachEvent()` должен получить ссылку на ту же функцию. Пример добавления и удаления события:

```
var btn = document.getElementById("myBtn");
var handler = function() {
    console.log("Clicked");
};
btn.attachEvent("onclick", handler);

// другой код

btn.detachEvent("onclick", handler);
```

Этот код назначает событию обработчик, сохраненный в переменной `handler`, а затем удаляет его методом `detachEvent()`.

Кроссбраузерные обработчики событий

Для кроссбраузерной обработки событий многие разработчики либо задействуют JavaScript-библиотеку, которая абстрагирует различия браузеров, либо пишут собственный код, основанный на распознавании возможностей. Чтобы такой код можно было применять максимально широко, он должен выполняться только на этапе всплытия событий.

Первым делом следует создать метод `addHandler()`, добавляющий обработчик события, используя подход DOM Level 0, DOM Level 2 или Internet Explorer в зависимости от того, какой из них доступен. Мы создадим обработчик в объекте `EventUtil`, который во всей главе будет применяться для сглаживания различий между браузерами. Метод `addHandler()` принимает три аргумента: целевой элемент, имя события и функцию-обработчик.

Нам также потребуется метод `removeHandler()`, который принимает те же три аргумента и удаляет ранее добавленный обработчик события, используя один из специфичных подходов или традиционный подход DOM Level 0, если никакой другой способ недоступен.

Вот полный код объекта `EventUtil`:

```
var EventUtil = {

    addHandler: function(element, type, handler) {
        if (element.addEventListener) {
            element.addEventListener(type, handler, false);
        } else if (element.attachEvent) {
            element.attachEvent("on" + type, handler);
        } else {
            element["on" + type] = handler;
        }
    },

    removeHandler: function(element, type, handler) {
        if (element.removeEventListener) {
            element.removeEventListener(type, handler, false);
        } else if (element.detachEvent) {
            element.detachEvent("on" + type, handler);
        } else {
            element["on" + type] = null;
        }
    }

};
```

Оба метода сначала проверяют, доступен ли для полученного элемента метод DOM Level 2. Если да, мы вызываем его, передавая в качестве аргументов тип события, обработчик и значение `false`, которое указывает, что метод должен быть выполнен на этапе всплытия. Во второй ветви кода мы пытаемся воспользоваться методом Internet Explorer. Заметьте, что перед типом события требуется префикс `"on"`, чтобы код работал в Internet Explorer 8 и более ранних версий. В третьей ветви вызывается метод DOM Level 0 (в современных браузерах эта ветвь выполняться не должна). Обратите внимание, что мы назначаем свойству обработчик события или значение `null` с помощью скобочной нотации.

Этот вспомогательный объект применяется следующим образом:

```
let btn = document.getElementById("myBtn");
let handler = function() {
```

```
    console.log("Clicked");
};
EventUtil.addHandler(btn, "click", handler);

// другой код

EventUtil.removeHandler(btn, "click", handler);
```

Конечно, методы `addHandler()` и `removeHandler()` не в состояния сгладить различия в функционале всех браузеров (например, проблема с областью видимости в Internet Explorer остается), но все же они упрощают добавление и удаление обработчиков событий. Помните также, что DOM Level 0 поддерживает только один обработчик для каждого события. К счастью, браузеры DOM Level 0 уже почти вышли из употребления, так что это не должно быть проблемой.

ОБЪЕКТ EVENT

Когда генерируется DOM-событие, все релевантные данные сохраняются в объекте `event`. Они включают базовые сведения, такие как целевой элемент и тип события, а также любые другие данные о конкретном событии. Например, для события мыши сохраняются сведения о позиции мыши, а для события клавиатуры — сведения о нажатых клавишах. Объект `event` поддерживают все браузеры, но по-разному.

Объект event в DOM

В браузерах, соответствующих требованиям DOM, объект `event` является единственным аргументом обработчика события независимо от того, был ли тот назначен в стиле DOM Level 0 или DOM Level 2. Пример двух способов обращения к объекту `event` внутри обработчика:

```
let btn = document.getElementById("myBtn");
btn.onclick = function(event) {
    console.log(event.type);    // "click"
};

btn.addEventListener("click", (event) => {
   console.log(event.type);     // "click"
}, false);
```

Оба обработчика в этом примере выводят в сообщение в консоли значение свойства `event.type`, которое всегда содержит тип произошедшего события, например `"click"` (это то же значение, которое передается в методы `addEventListener()` и `removeEventListener()`).

При назначении обработчика события с помощью HTML-атрибута объект `event` доступен как переменная `event`:

```
<input type="button" value="Click Me" onclick="console.log(event.type)">
```

Благодаря этому HTML-обработчики событий можно использовать аналогично JavaScript-функциям.

Объект `event` содержит свойства и методы, связанные с конкретным событием, приведшим к его созданию. Доступные свойства и методы различаются в зависимости от типа события, но члены из приведенной таблицы есть у всех событий.

СВОЙСТВО/МЕТОД	ТИП	ЧТЕНИЕ/ ЗАПИСЬ	ОПИСАНИЕ
`bubbles`	Boolean	Только чтение	Указывает, всплывает ли событие
`cancelable`	Boolean	Только чтение	Указывает, можно ли отменить поведение события, предлагаемое по умолчанию
`currentTarget`	Element	Только чтение	Элемент, чей обработчик обрабатывает событие в текущий момент
`defaultPrevented`	Boolean	Только чтение	Значение true указывает, что был вызван метод `preventDefault()` (свойство было добавлено в DOM Level 3 Events)
`detail`	Integer	Только чтение	Дополнительные сведения о событии
`eventPhase`	Integer	Только чтение	Этап, на котором вызывается обработчик события (1 – перехват; 2 – обработка в целевом элементе; 3 – всплытие)
`preventDefault()`	Function	Только чтение	Отменяет поведение по умолчанию. Этот метод можно использовать, если свойство cancelable имеет значение true
`stopImmediatePropagation()`	Function	Только чтение	Останавливает перехват или всплытие события и предотвращает вызов других обработчиков события (метод был добавлен в DOM Level 3 Events)
`stopPropagation()`	Function	Только чтение	Отменяет перехват или всплытие события. Этот метод можно использовать, если свойство `bubbles` имеет значение `true`

СВОЙСТВО/МЕТОД	ТИП	ЧТЕНИЕ/ ЗАПИСЬ	ОПИСАНИЕ
`target`	Element	Только чтение	Целевой элемент события
`trusted`	Boolean	Только чтение	Значение true указывает, что событие было сгенерировано браузером. Значение `false` указывает, что событие было создано разработчиком с помощью JavaScript (свойство было добавлено в DOM Level 3 Events)
`type`	String	Только чтение	Тип события
`view`	AbstractView	Только чтение	Абстрактное представление, связанное с событием (объект `window`, в котором произошло событие)

Внутри обработчика события объект `this` всегда имеет значение `currentTarget`, тогда как свойство `target` содержит фактический целевой элемент события. Если обработчик события назначен самому целевому элементу, значения `this`, `currentTarget` и `target` совпадают, например:

```
let btn = document.getElementById("myBtn");
btn.onclick = function(event) {
    console.log(event.currentTarget === this);     // true
    console.log(event.target === this);            // true
};
```

Этот код сравнивает свойства `currentTarget` и `target` со значением `this`. Поскольку событие `click` было сгенерировано кнопкой, все они равны. В обработчике события, назначенном родительскому узлу кнопки, такому как `document.body`, значения были бы разными. Рассмотрим пример обработчика события `click` на `document.body`:

```
document.body.onclick = function(event) {
    console.log(event.currentTarget === document.body);               // true
    console.log(this === document.body);                              // true
    console.log(event.target === document.getElementById("myBtn"));   // true
};
```

Здесь при щелчке на кнопке свойства `this` и `currentTarget` равны `document.body`, потому что именно этому узлу назначен обработчик события. Однако генерирует событие кнопка, поэтому свойство `target` указывает на нее. Поскольку самой кнопке обработчик события не назначен, событие всплывает к узлу `document.body`, где и обрабатывается.

Свойство type полезно, если нужно обработать несколько событий с помощью одной функции, например:

```
let btn = document.getElementById("myBtn");
let handler = function(event) {
    switch(event.type) {
        case "click":
            console.log("Clicked");
            break;

        case "mouseover":
            event.target.style.backgroundColor = "red";
            break;

        case "mouseout":
            event.target.style.backgroundColor = "";
            break;
    }
};

btn.onclick = handler;
btn.onmouseover = handler;
btn.onmouseout = handler;
```

В этом примере функция handler обрабатывает три события: click, mouseover и mouseout. При щелчке на кнопке функция выводит сообщение в консоли, при наведении указателя мыши на кнопку изменяет ее цвет на красный, а при смещении указателя с кнопки возвращает ей цвет, предлагаемый по умолчанию. Используя свойство event.type, функция определяет, какое событие произошло, и реагирует надлежащим образом.

Метод preventDefault() отменяет для конкретного события выполнение действия, предлагаемого по умолчанию. Например, при щелчке на ссылке таким действием является переход по URL-адресу, указанному в ее атрибуте href. При желании это действие можно отменить в обработчике события следующим образом:

```
let link = document.getElementById("myLink");
link.onclick = function(event) {
    event.preventDefault();
};
```

У любого события, которое можно отменить методом preventDefault(), свойство cancelable имеет значение true.

Метод stopPropagation() немедленно останавливает распространение события по DOM-структуре, блокируя последующий перехват или всплытие события. Например, в обработчике, назначенном кнопке, с помощью метода stopPropagation() можно предотвратить выполнение обработчика, назначенного узлу document.body:

```
let btn = document.getElementById("myBtn");
btn.onclick = function(event) {
    console.log("Clicked");
    event.stopPropagation();
};
```

```
document.body.onclick = function(event) {
    console.log("Body clicked");
};
```

Без вызова метода `stopPropagation()` этот код при щелчке на кнопке вывел бы в консоль два сообщения, но в приведенной версии событие `click` не достигает узла `document.body` и его обработчик `onclick` не выполняется.

Свойство `eventPhase` позволяет определить текущий этап распространения события. При вызове обработчика события на этапе перехвата значение `eventPhase` равно 1, при обработке события в целевом элементе — 2, а во время всплытия — 3. Имейте в виду, что хотя обработка события в целевом элементе относится к этапу всплытия, значение `eventPhase` в этом случае всегда равно 2. Рассмотрим пример:

```
let btn = document.getElementById("myBtn");
btn.onclick = function(event) {
    console.log(event.eventPhase);    // 2
};

document.body.addEventListener("click", function(event) {
    console.log(event.eventPhase);    // 1
}, true);

document.body.onclick = function(event) {
    console.log(event.eventPhase);    // 3
};
```

В этом примере при щелчке на кнопке первым вызывается один из обработчиков, назначенных узлу `document.body`. Он срабатывает на этапе перехвата и отображает оповещение со значением 1. Вторым вызывается обработчик самой кнопки, в котором свойство `eventPhase` имеет значение 2. Наконец, на этапе всплытия вызывается другой обработчик, назначенный узлу `document.body`, при этом значение `eventPhase` равно 3. Если это свойство равно 2, значения `this`, `target` и `currentTarget` всегда одинаковы.

> **ПРИМЕЧАНИЕ** Объект `event` существует только во время выполнения обработчиков событий, а после выполнения всех обработчиков он уничтожается.

Объект event в Internet Explorer

В отличие от DOM, в устаревшем Internet Explorer способ доступа к объекту `event` зависит от того, как был назначен обработчик события. Если использовался подход DOM Level 0, объект `event` доступен только как свойство объекта `window`, например:

```
var btn = document.getElementById("myBtn");
btn.onclick = function() {
    let event = window.event;
    console.log(event.type);        // "click"
};
```

Здесь объект event используется для определения типа произошедшего события (свойство type в Internet Explorer идентично одноименному DOM-свойству). Если же обработчик события был назначен с помощью метода attachEvent(), объект event передается в обработчик как его единственный аргумент:

```
var btn = document.getElementById("myBtn");
btn.attachEvent("onclick", function(event) {
    console.log(event.type);        // "click"
});
```

При использовании метода attachEvent() объект event также доступен в объекте window, как и при подходе DOM Level 0. В обработчик события он передается ради удобства.

Если обработчик события назначается с помощью HTML-атрибута, объект event доступен как переменная event (как и в модели DOM), например:

```
<input type="button" value="Click Me" onclick="console.log(event.type)">
```

В Internet Explorer объект event также содержит свойства и методы, связанные с событием, приведшим к его созданию. Многие из них соответствуют свойствам и методам DOM или связаны с ними. Как и в DOM, свойства и методы объекта event зависят от типа произошедшего события, при этом члены, указанные в таблице, доступны для всех событий.

СВОЙСТВО/МЕТОД	ТИП	ЧТЕНИЕ/ЗАПИСЬ	ОПИСАНИЕ
cancelBubble	Boolean	Чтение и запись	По умолчанию это свойство имеет значение false, но ему можно присвоить значение true, чтобы отменить всплытие события (это аналог DOM-метода stopPropagation())
returnValue	Boolean	Чтение и запись	По умолчанию это свойство имеет значение true, но ему можно присвоить значение false, чтобы отменить поведение события, предлагаемое по умолчанию (это аналог DOM-метода preventDefault())
srcElement	Element	Только чтение	Целевой элемент события (аналог DOM-свойства target)
type	String	Только чтение	Тип события

Поскольку область видимости обработчика события зависит от того, как он был назначен, объект this не всегда указывает на целевой элемент события. Поэтому используйте вместо этого свойство event.srcElement. Пример использования this с двумя разными объектами event:

```
var btn = document.getElementById("myBtn");
btn.onclick = function() {
```

```
    console.log(window.event.srcElement === this);     // true
};

btn.attachEvent("onclick", function(event) {
    console.log(event.srcElement === this);            // false
});
```

В первом обработчике события, назначенном с использованием подхода DOM Level 0, свойство `srcElement` равно `this`, но во втором эти значения различаются.

Свойство `returnValue` аналогично DOM-методу `preventDefault()` — оно тоже отменяет поведение, предлагаемое по умолчанию конкретного события. Чтобы предотвратить действие, предлагаемое по умолчанию, нужно лишь присвоить ему значение `false`, например:

```
var link = document.getElementById("myLink");
link.onclick = function() {
    window.event.returnValue = false;
};
```

В этом примере свойство `returnValue` отменяет действие для щелчка по ссылке, предлагаемое по умолчанию. В отличие от DOM, при этом никак нельзя узнать, возможна ли отмена события.

Свойство `cancelBubble` делает то же самое, что и DOM-метод `stopPropagation()` — останавливает всплытие события. Поскольку в Internet Explorer 8 и более ранних версий этап перехвата не поддерживается, оно отменяет только всплытие события, тогда как метод `stopPropagation()` отменяет и перехват, и всплытие. Вот пример отмены всплытия:

```
var btn = document.getElementById("myBtn");
btn.onclick = function() {
    console.log("Clicked");
    window.event.cancelBubble = true;
};

document.body.onclick = function() {
    console.log("Body clicked ");
};
```

Присвоение значения `true` свойству `cancelBubble` в обработчике щелчка на кнопке предотвращает всплытие события до обработчика, назначенного узлу `document.body`. В результате при щелчке на кнопке отображается только одно сообщение в консоли.

Кроссбраузерный объект event

Хотя объекты `event` в DOM и Internet Explorer различаются, они достаточно похожи, чтобы можно было создавать кроссбраузерные решения для обработки событий. Все данные и возможности IE-объекта `event` доступны в несколько иной форме в DOM-объекте, что позволяет легко провести параллели между двумя моделями

событий. Созданный ранее объект `EventUtil` можно расширить методами, компенсирующими их различия:

```
var EventUtil = {

    addHandler: function(element, type, handler) {
        // код опущен с целью сокращения объема листинга
    },

    getEvent: function(event) {
        return event ? event : window.event;
    },

    getTarget: function(event) {
        return event.target || event.srcElement;
    },

    preventDefault: function(event) {
        if (event.preventDefault) {
            event.preventDefault();
        } else {
            event.returnValue = false;
        }
    },

    removeHandler: function(element, type, handler) {
        // код опущен с целью сокращения объема листинга
    },

    stopPropagation: function(event) {
        if (event.stopPropagation) {
            event.stopPropagation();
        } else {
            event.cancelBubble = true;
        }
    }

};
```

Этот код добавляет к объекту `EventUtil` четыре новых метода. Первый, `getEvent()`, возвращает ссылку на объект `event` независимо от того, как был назначен обработчик события (напомним, что в Internet Explorer от этого зависит способ доступа к событию). В качестве аргумента этот метод принимает объект `event`, переданный в обработчик события. Пример нормализации объекта `event` с помощью `EventUtil`:

```
btn.onclick = function(event) {
    event = EventUtil.getEvent(event);
};
```

Если браузер соответствует требованиям DOM, метод `getEvent()` просто возвращает полученную переменную `event`. В Internet Explorer аргумент `event` не определен, поэтому метод возвратит свойство `window.event`. Добавление этой строки в начало обработчиков событий гарантирует, что объект `event` будет доступен всегда, независимо от браузера.

Метод `getTarget()` возвращает целевой элемент события. В нем мы проверяем, есть ли у объекта `event` свойство `target`, и если да — возвращаем его, иначе вместо него используется свойство `srcElement`. Этот метод применяется следующим образом:

```
btn.onclick = function(event) {
    event = EventUtil.getEvent(event);
    let target = EventUtil.getTarget(event);
};
```

Метод `preventDefault()` отменяет для события поведение, предлагаемое по умолчанию. Получив объект `event`, он проверяет, есть ли у события метод `preventDefault()`, и если да, вызывает его, в противном случае присваивает свойству `returnValue` значение `false`. Вот пример его вызова:

```
let link = document.getElementById("myLink");
link.onclick = function(event) {
    event = EventUtil.getEvent(event);
    EventUtil.preventDefault(event);
};
```

Этот код предотвращает переход на другую страницу при щелчке на ссылке. Здесь мы вызываем метод `EventUtil.getEvent()` для получения объекта `event`, который затем передается в метод `EventUtil.preventDefault()` для отмены действия, предлагаемого по умолчанию.

Последний метод, `stopPropagation()`, работает похожим образом. Сначала он пытается остановить распространение события с помощью DOM-метода, а если тот недоступен, использует свойство `cancelBubble`. Применяется он следующим образом:

```
let btn = document.getElementById("myBtn");
btn.onclick = function(event) {
    console.log("Clicked");
    event = EventUtil.getEvent(event);
    EventUtil.stopPropagation(event);
};

document.body.onclick = function(event) {
    console.log("Body clicked ");
};
```

В этом фрагменте мы получаем объект `event` с помощью метода `EventUtil.getEvent()`, а затем передаем его в метод `EventUtil.stopPropagation()`. Помните, что в зависимости от браузера он останавливает и перехват, и всплытие события либо только всплытие.

ТИПЫ СОБЫТИЙ

В спецификации DOM Level 3 Events определены категории событий, указанные в приведенном списке. Как уже отмечалось, от типа события зависят доступные сведения о нем.

- **События пользовательского интерфейса** — это общие события браузера, которые иногда связаны с BOM.
- **События изменения фокуса** генерируются, когда элемент получает или теряет фокус.
- **События мыши** генерируются при выполнении каких-либо действий на странице с помощью мыши.
- **События колесика** генерируются при использовании колесика мыши (или аналогичного устройства).
- **События редактирования текста** генерируются при вводе текста в документ.
- **События клавиатуры** генерируются при выполнении каких-либо действий на странице с помощью клавиатуры.
- **События композиции** генерируются при вводе символов в редакторе метода ввода (Input Method Editor, IME).

В дополнение к этим категориям доступны HTML5-события, а также фирменные DOM- и BOM-события. Фирменные события обычно определяют, исходя из требований разработчиков, а не спецификаций, и их реализации могут зависеть от браузера.

DOM Level 3 Events переопределяет группы событий DOM Level 2 Events и содержит дополнительные события. Спецификации DOM Level 2 и 3 Events поддерживают все основные браузеры.

События пользовательского интерфейса

События пользовательского интерфейса (User Interface, UI) не всегда связаны с действиями пользователя. Они существовали в той или иной форме еще до спецификации DOM и были оставлены для обеспечения обратной совместимости.

- `DOMActivate` — генерируется при активации элемента с помощью мыши или клавиатуры (оно более общее, чем события `click` и `keydown`). Это событие объявлено устаревшим в DOM Level 3 Events. Из-за различий реализации рекомендуется не использовать его.
- `load` — генерируется для объекта `window` при завершении загрузки страницы, для набора фреймов при завершении загрузки всех фреймов, а также для элементов `<img>` и `<object>` при завершении их загрузки.
- `unload` — генерируется для объекта `window` при завершении выгрузки страницы, для набора фреймов при завершении выгрузки всех фреймов и для элемента `<object>` при завершении его выгрузки.
- `abort` — генерируется для элемента `<object>`, если пользователь останавливает загрузку, а элемент загружен не полностью.
- `error` — генерируется для объекта `window`, если возникает JavaScript-ошибка; для элемента `<img>`, если невозможно загрузить указанное изображение; для

элемента <object>, если невозможно загрузить его, или для набора фреймов, если невозможно загрузить один или несколько фреймов.

- select — генерируется, когда пользователь выделяет один или несколько символов в текстовом поле (<input> или <textarea>).
- resize — генерируется для объекта window или фрейма при изменении его размеров.
- scroll — генерируется для любого элемента с полосой прокрутки, когда пользователь его прокручивает. Полоса прокрутки загруженной страницы принадлежит элементу <body>.

Большинство HTML-событий связаны с объектом window или с элементами управления форм.

За исключением DOMActivate, в спецификации DOM Level 2 Events эти события входили в группу HTML Events (событие DOMActivate входило в группу UI Events).

Имейте в виду, что этот вызов должен возвращать true, только если HTML-события реализованы в браузере согласно спецификации DOM Level 2 Events. Браузеры могут поддерживать эти события нестандартным образом и возвращать false.

Событие load

Наверное, событие load используется в JavaScript чаще любых других событий. Для объекта window оно возникает, когда загружена вся страница, включая все внешние ресурсы, такие как изображения, JavaScript- и CSS-файлы. Определить обработчик onload можно двумя способами. Первый — использовать JavaScript:

```
window.addEventListener("load", (event) => {
    console.log("Loaded!");
});
```

Этот код основан на использовании кроссбраузерного объекта EventUtil, который был рассмотрен ранее. Как и при других событиях, в обработчик передается объект event. Он не предоставляет никаких дополнительных сведений об этом событии, хотя интересно отметить, что в браузерах, соответствующих DOM, свойство event.target указывает на объект document, тогда как Internet Explorer до версии 8 не задает свойство srcElement для этого события.

Второй способ назначить обработчик события load — это добавить атрибут onload в элемент <body>, например:

```
<!DOCTYPE html>
<html>
<head>
    <title>Load Event Example</title>
</head>
<body onload="console.log('Loaded!')">

</body>
</html>
```

Вообще говоря, обработчики любых событий объекта `window` назначаются в HTML с помощью атрибутов элемента `<body>`, потому что в HTML-коде элемент `window` недоступен. Этот трюк применяется для обеспечения обратной совместимости и все еще хорошо поддерживается во всех браузерах, но по возможности лучше использовать подход JavaScript.

> **ПРИМЕЧАНИЕ** Согласно спецификации DOM Level 2 Events, событие `load` должно генерироваться для объекта `document`, а не `window`, однако из соображений обратной совместимости во всех браузерах оно генерируется для объекта `window`.

Событие `load` также генерируется для изображений, причем как соответствующих, так и не соответствующих модели DOM. В HTML-коде можно назначить обработчик события `load` для любых изображений в документе следующим образом:

```
<img src="smile.gif" onload="console.log('Image loaded.')">
```

По окончании загрузки указанного изображения этот код выводит сообщение в консоль `"Image loaded."`. То же самое можно сделать с помощью JavaScript:

```
let image = document.getElementById("myImage");
image. addEventListener("load", (event) => {
    console.log(event.target.src);
});
```

Этот обработчик события `load` принимает объект `event`, но интересного в нем мало. Целевым элементом события является элемент `<img>`, так что мы можем отобразить его свойство `src`.

При создании элемента `<img>` можно назначить ему обработчик события, вызываемый при завершении загрузки изображения. В этом случае важно назначить обработчик до установки свойства `src`, например:

```
window.addEventListener("load", () => {
    let image = document.createElement("img");
    image.addEventListener("load", (event) => {
        event = EventUtil.getEvent(event);
        console.log(event.target.src);
    });
    document.body.appendChild(image);
    image.src = "smile.gif";
});
```

В первой части примера мы назначаем обработчик события загрузки окна. Это важно потому, что мы собираемся добавить новый DOM-элемент, а попытка использовать элемент `document.body` до полной загрузки страницы может привести к ошибке. Далее мы создаем элемент `image` и устанавливаем обработчик его события `load`, а затем добавляем изображение на страницу и задаем его атрибут `src`. Чтобы началась загрузка изображения, элемент не нужно добавлять к документу — оно начинает загружаться, как только задано свойство `src`.

Эту методику можно также использовать с объектом `Image` из DOM Level 0, который еще до внедрения DOM применялся на клиентских устройствах для предварительной загрузки изображений. Он работает так же, как элемент `<img>`, но его нельзя добавить в DOM-дерево. Рассмотрим пример создания нового экземпляра объекта `Image`:

```
window.addEventListener("load", () => {
    let image = new Image();
    image.addEventListener("load", (event) => {
        console.log("Image loaded!");
    });
    image.src = "smile.gif";
});
```

Этот код обрабатывает загрузку изображения, созданного с помощью конструктора `Image`. Некоторые браузеры реализуют объект `Image` как элемент `<img>`, но не все, так что лучше считать, что они различаются.

> **ПРИМЕЧАНИЕ** Internet Explorer 8 и более ранних версий не создают объект `event`, если событие `load` генерируется для изображения, которое не является частью DOM-документа. Это относится и к элементам `<img>`, которые не были добавлены к документу, и к объектам `Image`. В Internet Explorer 9 этот дефект исправлен.

Есть и другие элементы, которые поддерживают событие `load` нестандартным образом. Элемент `<script>` генерирует событие `load`, уведомляя о завершении загрузки динамически загружаемого JS-файла. В отличие от изображений загрузка JS-файлов с сервера начинается только после задания свойства `src` и добавления элемента в документ, так что порядок назначения обработчика события и установки свойства `src` не важен. Следующий пример поясняет назначение обработчика события элементу `<script>`:

```
window.addEventListener("load", () => {
    let script = document.createElement("script");
    script.addEventListener("load", (event) => {
        console.log("Loaded");
    });
    script.src = "example.js";
    document.body.appendChild(script);
});
```

Здесь обработчик события `load` назначается элементу `<script>` с помощью кроссбраузерного объекта `EventUtil`. В большинстве браузеров целевым элементом события будет узел `<script>`. В Internet Explorer 8 и более ранних версий элементы `<script>` не поддерживают событие `load`.

В Internet Explorer и Opera событие `load` генерируется также для элементов `<link>`, что позволяет узнать о завершении загрузки таблицы стилей, например:

```
window.addEventListener("load", () => {
    let link = document.createElement("link");
    link.type = "text/css";
    link.rel= "stylesheet";
    link.addEventListener("load", (event) => {
        console.log("css loaded");
    });
    link.href = "example.css";
    document.getElementsByTagName("head")[0].appendChild(link);
});
```

Подобно сценариям, загрузка таблицы стилей начинается только после задания свойства `href` и добавления элемента `<link>` в документ.

Событие unload

Событие `unload` сигнализирует о завершении выгрузки документа. Оно обычно генерируется при переходе с одной страницы на другую и чаще всего используется для очистки ссылок с целью предотвращения утечек памяти. Подобно событию `load` обработчик события `unload` можно назначить двумя способами. Первый — с помощью JavaScript:

```
window.addEventListener("unload", (event) => {
    console.log("Unloaded!");
});
```

В браузерах, соответствующих DOM, объект `event` этого события содержит только свойство `target` (со значением `document`). Internet Explorer 8 и более ранних версий не предоставляют для этого события свойство `srcElement`.

Второй способ задать обработчик `onunload` — это добавить атрибут в элемент `<body>`:

```
<!DOCTYPE html>
<html>
<head>
    <title>Пример события unload</title>
</head>
<body onunload="console.log('Unloaded!')">

</body>
</html>
```

Какой бы подход вы ни выбрали, будьте осторожны с кодом внутри обработчика события `unload`. Поскольку оно генерируется после выгрузки всего контента, в обработчике доступны не все объекты, которые были доступны при загрузке страницы. Попытка изменить расположение или вид DOM-узла может привести к ошибке.

ПРИМЕЧАНИЕ Согласно спецификации DOM Level 2 Events, событие `unload` должно генерироваться для элемента `<body>`, а не для объекта `window`, однако из соображений обратной совместимости во всех браузерах оно генерируется для объекта `window`.

Событие resize

Событие `resize` происходит при изменении высоты или ширины окна браузера. Оно генерируется для объекта `window`, так что его обработчик можно назначить с помощью либо JS-кода, либо атрибута `onresize` элемента `<body>`. Как уже отмечалось, первый подход предпочтительнее:

```
window.addEventListener("resize", (event) => {
    console.log("Resized");
});
```

Подобно другим событиям объекта `window` в браузерах, соответствующих модели DOM, для события `resize` создается объект `event`, у которого свойство `target` имеет значение `document`. В Internet Explorer 8 и более ранних версий у него нет полезных свойств.

Способ генерирования события `resize` зависит от браузера. Internet Explorer, Safari, Chrome и Opera генерируют его, когда размеры окна браузера меняются на один пиксель, и продолжают генерировать, пока пользователь не прекратит изменять размеры окна. В Firefox это событие генерируется, когда пользователь прекращает изменять размеры окна браузера. Из-за повторения события не следует выполнять в его обработчике ресурсоемкие операции, потому что это может заметно замедлить работу браузера.

> **ПРИМЕЧАНИЕ** Событие `resize` также возникает при свертывании и развертывании окна браузера.

Событие scroll

Хотя событие `scroll` генерируется для объекта `window`, на самом деле оно сигнализирует об изменениях элемента уровня страницы. В режиме совместимости изменения можно отслеживать с помощью свойств `scrollLeft` и `scrollTop` элемента `<body>`, а в стандартном режиме изменения применяются на уровне элемента `<html>` во всех браузерах, за исключением Safari (где используется элемент `<body>`). Вот пример правильной обработки события:

```
window.addEventListener("scroll", (event) => {
    if (document.compatMode == "CSS1Compat") {
        console.log(document.documentElement.scrollTop);
    } else {
        console.log(document.body.scrollTop);
    }
});
```

Этот код назначает обработчик события, который выводит позицию прокрутки по вертикали с учетом режима визуализации. Safari до версии 3.1 не поддерживает свойство `document.compatMode`, поэтому в старых версиях этого браузера выполняется вторая ветвь кода.

Подобно событию `resize` событие `scroll` генерируется во время прокрутки многократно, так что обрабатывать его следует как можно проще.

События изменения фокуса

События изменения фокуса генерируются, когда элементы страницы получают или теряют фокус. В сочетании со свойствами `document.hasFocus()` и `document.activeElement` они предоставляют сведения о том, что пользователь делает на странице. К этой категории относится шесть событий.

- `blur` — генерируется, когда элемент теряет фокус. Это событие не всплывает и поддерживается во всех браузерах.
- `DOMFocusIn` — генерируется, когда элемент получает фокус. Это всплывающая версия HTML-события `focus`, которая поддерживается только в Opera. В DOM Level 3 Events событие `DOMFocusIn` объявлено устаревшим, а вместо него предложено использовать событие `focusin`.
- `DOMFocusOut` — генерируется, когда элемент теряет фокус. Это универсальная версия HTML-события `blur`, которая поддерживается только в Opera. В DOM Level 3 Events событие `DOMFocusOut` объявлено устаревшим, а вместо него предложено использовать событие `focusout`.
- `focus` — генерируется, когда элемент получает фокус. Это событие не всплывает и поддерживается во всех браузерах.
- `focusin` — генерируется, когда элемент получает фокус. Это всплывающая версия HTML-события `focus`.
- `focusout` — генерируется, когда элемент теряет фокус. Это универсальная версия HTML-события `blur`.

Основными в этой группе являются события `focus` и `blur`, которые поддерживаются в браузерах с ранних дней JavaScript. Проблема в том, что они не всплывают, поэтому в Internet Explorer были добавлены события `focusin` и `focusout`, а в Opera — события `DOMFocusIn` и `DOMFocusOut`. IE-события были стандартизированы в DOM Level 3 Events.

При перемещении фокуса от одного элемента страницы к другому события генерируются в следующем порядке.

1. Для элемента, теряющего фокус, генерируется событие `focusout`.
2. Для элемента, получающего фокус, генерируется событие `focusin`.
3. Для элемента, теряющего фокус, генерируется событие `blur`.
4. Для элемента, теряющего фокус, генерируется событие `DOMFocusOut`.
5. Для элемента, получающего фокус, генерируется событие `focus`.
6. Для элемента, получающего фокус, генерируется событие `DOMFocusIn`.

Целевым элементом событий `blur`, `DOMFocusOut` и `focusout` является элемент, теряющий фокус, а целевым элементом событий `focus`, `DOMFocusIn` и `focusin` — элемент, получающий фокус.

События мыши и колесика мыши

События мыши (mouse events) используются в веб-программировании чаще любых других, потому что большинство действий в браузерах выполняются с помощью мыши. В DOM Level 3 Events эта группа включает девять событий.

- `click` — генерируется, когда пользователь щелкает основной кнопкой мыши (обычно левой) или нажимает клавишу Enter. То, что обработчики `onclick` могут запускаться и клавиатурой, и мышью, помогает упростить работу со страницей для людей с ограниченными возможностями.
- `dblclick` — генерируется, когда пользователь дважды щелкает основной кнопкой мыши (обычно левой). Это событие отсутствует в DOM Level 2 Events, но благодаря широкой поддержке оно было стандартизировано в DOM Level 3 Events.
- `mousedown` — генерируется, когда пользователь нажимает любую кнопку мыши. Это событие не может генерироваться с клавиатуры.
- `mouseenter` — генерируется при наведении указателя мыши на элемент. Это событие не всплывает и не генерируется при пересечении границ потомков элемента. Оно было добавлено в DOM Level 3 Events.
- `mouseleave` — генерируется при смещении указателя мыши, находящегося на элементе, за его пределы. Это событие не всплывает и не генерируется при пересечении границ потомков элемента. Оно было добавлено в DOM Level 3.
- `mousemove` — многократно генерируется при перемещении указателя мыши на элементе. Это событие не может генерироваться с клавиатуры.
- `mouseout` — генерируется при перемещении указателя мыши, находящегося на элементе, в область другого элемента. Новый элемент может находиться вне исходного или быть его дочерним элементом. Это событие не может генерироваться с клавиатуры.
- `mouseover` — генерируется при наведении указателя мыши на элемент. Это событие не может генерироваться с клавиатуры.
- `mouseup` — генерируется, когда пользователь отпускает кнопку мыши. Это событие не может генерироваться с клавиатуры.

События мыши поддерживаются всеми элементами страницы. Все события мыши, кроме `mouseenter` и `mouseleave`, всплывают и могут быть отменены, что может влиять на другие события из-за связей между ними.

Событие `click` может возникнуть, только если вслед за событием `mousedown` для того же элемента генерируется событие `mouseup`; если отменить одно из них, событие `click` не возникнет. Аналогичным образом событие `dbclick` требует двух событий `click`. Если отменить одно из них (`mousedown` или `mouseup`), событие `dblclick` не возникнет. Эти четыре события мыши всегда генерируются в следующем порядке:

1. `mousedown`.
2. `mouseup`.
3. `click`.

4. `mousedown`.
5. `mouseup`.
6. `click`.
7. `dblclick`.

Таким образом, события `click` и `dblclick` зависят от других событий, а `mousedown` и `mouseup` — нет.

Internet Explorer до версии 8 включительно пропускает вторые события `mousedown` и `click` при двойном щелчке:

1. `mousedown`.
2. `mouseup`.
3. `click`.
4. `mouseup`.
5. `dblclick`.

Заметьте, что компонент DOM Level 3 называется `"MouseEvent"`, а не `"MouseEvents"`.

Группа событий мыши включает также *событие колесика* (wheel event). Оно называется `mousewheel` и используется для работы не только с колесиком мыши, но и с похожими устройствами, такими как сенсорная панель в Mac.

Клиентские координаты

Каждое событие мыши происходит в конкретном месте области просмотра, при этом координаты указателя мыши сохраняются в свойствах `clientX` и `clientY` объекта `event`, которые поддерживаются во всех браузерах. Клиентские координаты (client coordinates) области просмотра показаны на рис. 17.4.

Получить клиентские координаты события мыши можно следующим образом:

```
let div = document.getElementById("myDiv");
div.addEventListener("click", (event) => {
    console.log(`Client coordinates: ${event.clientX}, ${event.clientY}`);
});
```

Этот обработчик события `click` элемента `<div>` выводит на экран клиентские координаты щелчка. Имейте в виду, что они не отражают состояние прокрутки, а потому не показывают расположение курсора на странице.

Страничные координаты

В то время как клиентские координаты указывают место возникновения события в области просмотра, *страничные координаты* (page coordinates) определяют это место на странице. Они представлены свойствами `pageX` и `pageY` объекта `event` и рассчитываются относительно левого и верхнего краев самой страницы, а не области просмотра.

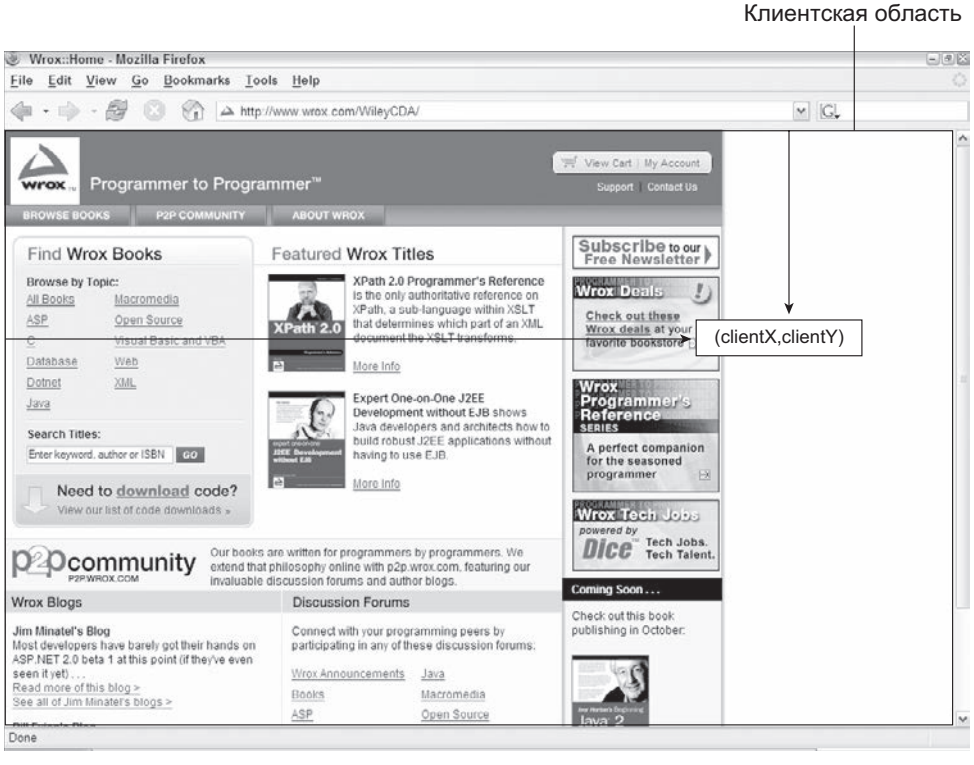

Рис. 17.4

Получить страничные координаты события мыши можно следующим образом:

```
let div = document.getElementById("myDiv");
div.addEventListener("click", (event) => {
    console.log(`Page coordinates: ${event.pageX}, ${event.pageY}`);
})
```

Если страница не прокручивалась, страничные координаты совпадают с клиентскими.

В Internet Explorer 8 и более ранних версий объект `event` не поддерживает страничные координаты, но их можно вычислить по клиентским координатам и свойствам прокрутки. Для этого нужно использовать свойства `scrollLeft` и `scrollTop` объекта `document.body` (в режиме совместимости) или `document.documentElement` (в стандартном режиме):

```
let div = document.getElementById("myDiv");
div.addEventListener("click", (event) => {
    let pageX = event.pageX,
        pageY = event.pageY;

    if (pageX === undefined) {
        pageX = event.clientX + (document.body.scrollLeft ||
```

```
                document.documentElement.scrollLeft);
    }

    if (pageY === undefined) {
        pageY = event.clientY + (document.body.scrollTop ||
                document.documentElement.scrollTop);
    }

    console.log(`Page coordinates: ${pageX}, ${pageY}`);
});
```

Экранные координаты

С помощью свойств `screenX` и `screenY` можно также определить координаты события мыши относительно всего экрана. *Экранные координаты* (screen coordinates) в браузере показаны на рис. 17.5.

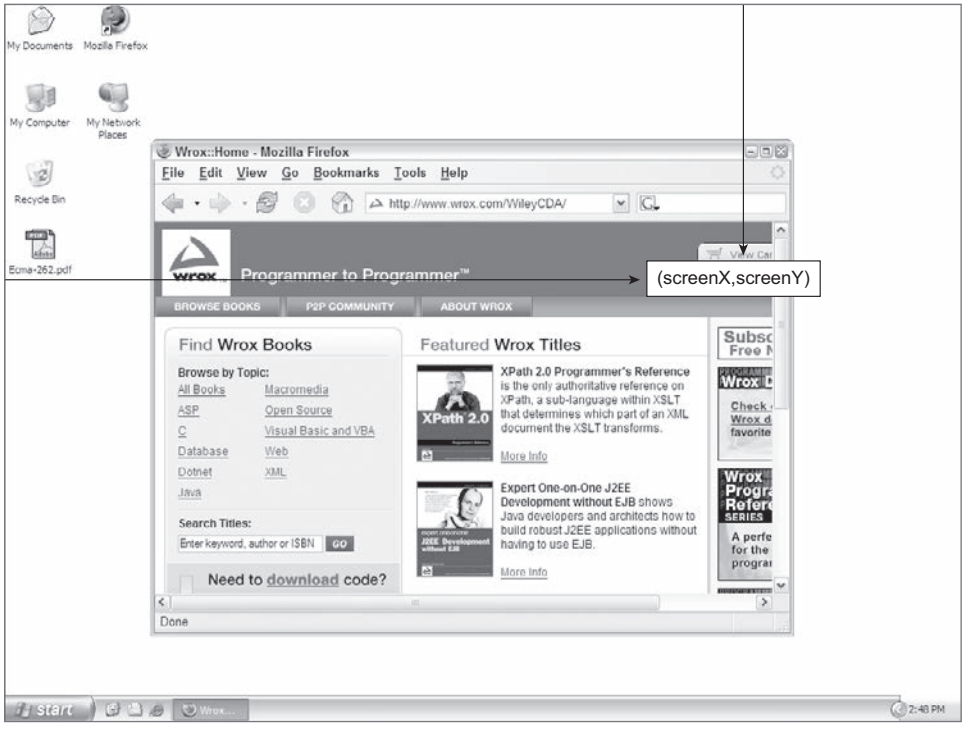

Рис. 17.5

Получить экранные координаты события мыши можно следующим образом:

```
let div = document.getElementById("myDiv");
div.addEventListener("click", (event) => {
    console.log(`Screen coordinates: ${event.screenX}, ${event.screenY}`);
});
```

Как и в предыдущих примерах, здесь обрабатывается событие `click` элемента `<div>`. Его обработчик отображает экранные координаты события.

Клавиши-модификаторы

Хотя события мыши инициируются преимущественно мышью, иногда для выбора нужного действия важно учесть состояние некоторых клавиш. Для изменения поведения событий мыши часто используются *клавиши-модификаторы* (modifier keys) Shift, Ctrl, Alt и Meta, которым в DOM соответствуют свойства `shiftKey`, `ctrlKey`, `altKey` и `metaKey`. Каждое из них содержит логическое значение `true`, если клавиша нажата, или `false` в противном случае. С помощью этих свойств можно определить состояние клавиш-модификаторов при возникновении события мыши. Рассмотрим пример, который проверяет состояние ключа модификатора при срабатывании события `click`:

```
let div = document.getElementById("myDiv");
div.addEventListener("click", (event) => {
    let keys = new Array();

    if (event.shiftKey) {
        keys.push("shift");
    }

    if (event.ctrlKey) {
        keys.push("ctrl");
    }

    if (event.altKey) {
        keys.push("alt");
    }

    if (event.metaKey) {
        keys.push("meta");
    }

    console.log("Keys: " + keys.join(","));

});
```

Этот обработчик `onclick` добавляет названия нажатых клавиш-модификаторов в массив `keys`, а затем выводит его содержимое в сообщение в консоли.

> **ПРИМЕЧАНИЕ** Современные браузеры поддерживают все четыре клавиши, а более ранние версии Internet Explorer не поддерживают свойство `metaKey`.

Связанные элементы

События `mouseover` и `mouseout` генерируются при перемещении указателя мыши из одного элемента на другой, а потому имеют отношение сразу к двум элементам.

Целевым элементом события mouseover является элемент, на который наводится указатель, а связанным элементом — элемент, где указатель находился ранее, тогда как у события mouseout все наоборот. Рассмотрим следующую HTML-страницу:

```
<!DOCTYPE html>
<html>
<head>
    <title>Related Elements Example</title>
</head>
<body>
    <div id="myDiv"
         style="background-color:red;height:100px;width:100px;"></div>
</body>
</html>
```

На этой странице отображается единственный элемент . Если сначала указатель мыши находится на нем, а затем перемещается за его пределы, для элемента генерируется событие mouseout, при этом связанным элементом является <body>. Одновременно для элемента <body> генерируется событие mouseover, при этом связанным считается элемент .

При возникновении событий mouseover и mouseout связанный элемент сохраняется в свойстве relatedTarget объекта event, которое при любых других событиях равно null. Браузеры Internet Explorer 8 и более ранних версий не поддерживают свойство relatedTarget, но предоставляют доступ к связанному элементу с помощью других свойств. При возникновении события mouseover связанный элемент доступен в Internet Explorer как свойство fromElement, а при возникновении события mouseout — как свойство toElement (Internet Explorer 9 поддерживает все свойства). Теперь мы можем добавить в объект EventUtil кроссбраузерный метод получения связанного элемента:

```
var EventUtil = {

    // другой код

    getRelatedTarget: function(event) {
        if (event.relatedTarget) {
            return event.relatedTarget;
        } else if (event.toElement) {
            return event.toElement;
        } else if (event.fromElement) {
            return event.fromElement;
        } else {
            return null;
        }
    },

    // другой код

};
```

Как и в предыдущих кроссбраузерных методах, для выбора возвращаемого значения здесь применяется механизм распознавания возможностей. Метод EventUtil.getRelatedTarget() можно использовать следующим образом:

```
let div = document.getElementById("myDiv");
div.addEventListener("mouseout", (event) => {
    let target = event.target;
    let relatedTarget = EventUtil.getRelatedTarget(event);
    console.log(
        `Moused out of ${target.tagName} to ${relatedTarget.tagName}`);
});
```

Этот код обрабатывает событие `mouseout` элемента `<div>`, выводя сообщение в консоль со сведениями о том, откуда и куда был перемещен указатель мыши.

Кнопки мыши

Событие `click` генерируется только при щелчке по основной кнопке мыши (или при нажатии клавиши Enter на клавиатуре), так что для его обработки сведения о кнопке не нужны. Для событий `mousedown` и `mouseup` в объекте `event` доступно свойство `button`, указывающее кнопку, которая была нажата или отпущена. В DOM оно может иметь одно из трех значений: 0 соответствует основной кнопке мыши, 1 — средней (обычно это кнопка-колесико), а 2 — дополнительной. В традиционной конфигурации основной кнопкой мыши является левая, а дополнительной правая.

В Internet Explorer до версии 8 свойство `button` тоже доступно, но имеет совершенно другие значения:

- `0` — никакая кнопка не нажата;
- `1` — нажата основная кнопка мыши;
- `2` — нажата дополнительная кнопка мыши;
- `3` — нажаты основная и дополнительная кнопки мыши;
- `4` — нажата средняя кнопка мыши;
- `5` — нажаты основная и средняя кнопки мыши;
- `6` — нажаты дополнительная и средняя кнопки мыши;
- `7` — нажаты все три кнопки мыши.

Как видите, в модели DOM свойство `button` гораздо проще, чем в Internet Explorer, но при этом столь же полезно, потому что сочетания кнопок мыши почти не используются. На практике все модели работы с кнопками мыши обычно преобразуют в формат DOM, потому что он встроен во все браузеры, кроме Internet Explorer 8 и более ранних версий. Преобразование кодов основной, средней и дополнительной кнопок мыши из Internet Explorer тривиально, а сочетания кнопок транслируются в нажатие одной кнопки, при этом приоритет всегда отдается основной кнопке. Например, коды 5 и 7 в Internet Explorer соответствуют значению 0 в DOM.

Дополнительные сведения о событиях

В спецификации DOM Level 2 Events для объекта `event` определено свойство `detail`, которое предоставляет дополнительные сведения о событии. У событий мыши оно

содержит число, указывающее, сколько раз был выполнен щелчок в указанном месте. Щелчком считается последовательность событий `mousedown` и `mouseup` в одной точке. Значение `detail` начинается с единицы и увеличивается при каждом щелчке. Если между событиями `mousedown` и `mouseup` указатель мыши перемещается, свойство `detail` обнуляется.

Internet Explorer предоставляет также следующие сведения о каждом событии мыши:

- `altLeft` — логическое значение, указывающее, нажата ли левая клавиша Alt (если это свойство равно `true`, то и свойство `altKey` равно `true`);
- `ctrlLeft` — логическое значение, указывающее, нажата ли левая клавиша Ctrl (если это свойство равно `true`, то и свойство `ctrlKey` равно `true`);
- `offsetX` — координата *x* указателя мыши относительно границ целевого элемента;
- `offsetY` — координата *y* указателя мыши относительно границ целевого элемента;
- `shiftLeft` — логическое значение, указывающее, нажата ли левая клавиша Shift (если это свойство равно `true`, то и свойство `shiftKey` равно `true`).

Польза от этих свойств невелика, потому что они доступны только в Internet Explorer и предоставляют сведения, которые можно получить иными способами.

Событие mousewheel

Событие `mousewheel` появилось в Internet Explorer 6 и позднее было добавлено в Opera, Chrome и Safari. Оно генерируется для любых элементов при вращении колесика мыши и всплывает к объекту `document` (в Internet Explorer 8) или `window` (в современных браузерах). Его объект `event` содержит всю стандартную информацию события мыши, а также дополнительное свойство `wheelDelta`, значением которого является число, кратное 120, — положительное при вращении колесика вперед и отрицательное при вращении назад (рис. 17.6).

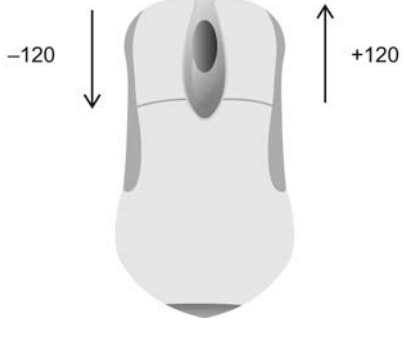

Рис. 17.6

Обработчик события `mousewheel` можно назначить любому элементу на странице или объекту `document` для обработки всех взаимодействий с колесиком мыши, например:

```
document.addEventListener("mousewheel", (event) => {
    console.log(event.wheelDelta);
});
```

Это обработчик просто отображает значение `wheelDelta`. В большинстве случаев нужно узнать лишь направление вращения колесика, что можно легко определить по знаку этого значения.

> **ПРИМЕЧАНИЕ** Благодаря популярности и широкой поддержке события `mousewheel` оно было добавлено в HTML5.

Поддержка устройств с сенсорным вводом

Устройства с сенсорным вводом с системой iOS или Android не поддерживают мышь, поэтому написание кода для них имеет ряд особенностей.

- Событие `dblclick` не поддерживается. При двойном щелчке в окне браузера увеличивается масштаб страницы, и переопределить это поведение нельзя.
- Касание элемента, поддерживающего щелчок, инициирует событие `mousemove`. Если это приводит к изменению контента, другие события не генерируются; если контент не изменяется, по очереди генерируются события `mousedown`, `mouseup` и `click`. При касании элемента, не поддерживающего щелчок, никакие события не возникают. Элемент, поддерживающий щелчок, — это элемент, при щелчке на котором выполняется какое-то действие, предлагаемое по умолчанию (такой как ссылка), или вызывается обработчик события `click`.
- Событие `mousemove` также инициирует события `mouseover` и `mouseout`.
- Если пользователь касается экрана двумя пальцами и в результате их перемещения страница прокручивается, возникают события `mousewheel` и `scroll`.

Специальные возможности

Если нужно обеспечить доступ к веб-приложению или веб-сайту для людей, использующих программы чтения с экрана, требуется повышенное внимание к событиям мыши. Как уже отмечалось, событие `click` может быть сгенерировано с помощью клавиши Enter, но другие события нельзя инициировать с клавиатуры. Не рекомендуется использовать их для отображения функциональных элементов или выполнения кода, так как это серьезно ограничивает возможности людей с плохим зрением. Перечислим советы по обработке событий мыши в контексте специальных возможностей.

- Используйте для выполнения кода событие `click`. Многие люди с нормальным зрением утверждают, что приложение воспринимается как более быстрое, если код запускается по событию `mousedown`, но программам чтения с экрана такой код недоступен.
- Не используйте событие `mouseover` для показа новых пунктов меню и подобных элементов, потому что программы чтения с экрана не могут инициировать его.

Если другие варианты не подходят, попробуйте добавить сочетания клавиш для вывода тех же элементов.

- Не используйте событие `dblclick` для выполнения важных действий. Его невозможно сгенерировать с клавиатуры.

Следование этим простым правилам значительно упростит работу с приложением или сайтом людей с ограниченными возможностями.

ПРИМЕЧАНИЕ Дополнительные сведения о доступности веб-страниц для людей с физическими нарушениями см. на сайтах www.webaim.org и http://accessibility.yahoo.com.

События клавиатуры и редактирования текста

События клавиатуры (keyboard events) первоначально входили в спецификацию DOM Level 2 Events, но соответствующий раздел был удален еще до принятия окончательной версии спецификации. В результате поддержка событий клавиатуры в браузерах основана преимущественно на реализациях DOM Level 0.

События клавиатуры из DOM Level 3 Events впервые были полностью реализованы в Internet Explorer 9. Разработчики других браузеров также начали внедрять этот стандарт, но и унаследованных реализаций все еще много.

Опишем три доступных события клавиатуры.

- `keydown` — генерируется при нажатии клавиши и повторяется, пока клавиша нажата.
- `keypress` — генерируется при нажатии символьной клавиши и повторяется, пока она нажата. Это событие также возникает для клавиши Esc. В DOM Level 3 Events оно объявлено устаревшим, а вместо него предложено использовать событие `textInput`.
- `keyup` — генерируется при отпускании клавиши.

Эти события наиболее наглядны при вводе данных в текстовом поле, но вообще их поддерживают все элементы.

Единственное событие редактирования текста называется `textInput`. Оно дополняет событие `keypress`, упрощая перехват вводимого текста до его отображения. Это событие генерируется непосредственно перед вставкой текста в текстовое поле.

Когда пользователь нажимает символьную клавишу, сначала генерируется событие `keydown`, за которым следует событие `keypress`, а за ним — `keyup`. События `keydown` и `keypress` предшествуют изменению содержимого текстового поля, а событие `keyup` возникает после изменения. Если пользователь удерживает символьную клавишу нажатой, события `keydown` и `keypress` генерируются многократно, пока клавиша не будет отпущена.

При нажатии несимвольной клавиши генерируется событие keydown, а вслед за ним — keyup. Если такая клавиша удерживается нажатой, событие keydown повторяется, а при отпускании клавиши возникает событие keyup.

ПРИМЕЧАНИЕ События клавиатуры поддерживают те же клавиши-модификаторы, что и события мыши, и имеют аналогичные свойства shiftKey, ctrlKey, altKey и metaKey. В Internet Explorer 8 и более ранних версий свойство metaKey не поддерживается.

Коды клавиш

Для событий keydown и keyup свойству keyCode объекта event назначается код соответствующей клавиши. У алфавитно-цифровых клавиш он равен ASCII-коду буквы в нижнем регистре или числа: например, для клавиши 7 свойству keyCode присваивается значение 55, а для клавиши A — 65 (независимо от состояния клавиши Shift). Свойство keyCode есть у объекта event и в DOM, и в Internet Explorer. Вот пример кода с ним:

```
let textbox = document.getElementById("myText");
textbox.addEventListener("keyup", (event) => {
    console.log(event.keyCode);
});
```

Этот обработчик выводит на экран значение keyCode при каждом событии keyup. Коды несимвольных клавиш приведены в таблице.

КЛАВИША	КОД КЛАВИШИ	КЛАВИША	КОД КЛАВИШИ
Backspace	8	8 на цифровой клавиатуре	104
Tab	9	9 на цифровой клавиатуре	105
Enter	13	Плюс на цифровой клавиатуре	107
Shift	16	Минус на цифровой и обычной клавиатурах	109
Ctrl	17	Точка на цифровой клавиатуре	110
Alt	18	Косая черта на цифровой клавиатуре	111
Pause/Break	19	F1	112
Caps Lock	20	F2	113
Esc	27	F3	114
Page Up	33	F4	115

КЛАВИША	КОД КЛАВИШИ	КЛАВИША	КОД КЛАВИШИ
Page Down	34	F5	116
End	35	F6	117
Home	36	F7	118
Стрелка влево	37	F8	119
Стрелка вверх	38	F9	120
Стрелка вправо	39	F10	121
Стрелка вниз	40	F11	122
Ins	45	F12	123
Del	46	Num Lock	144
Левая клавиша Windows	91	Scroll Lock	145
Правая клавиша Windows	92	Точка с запятой (IE/Safari/ Chrome)	186
Клавиша контекстного меню	93	Точка с запятой (Opera/FF)	59
0 на цифровой клавиатуре	96	Меньше	188
1 на цифровой клавиатуре	97	Больше	190
2 на цифровой клавиатуре	98	Косая черта	191
3 на цифровой клавиатуре	99	Гравис (`)	192
4 на цифровой клавиатуре	100	Равно	61
5 на цифровой клавиатуре	101	Левая квадратная скобка	219
6 на цифровой клавиатуре	102	Обратная косая черта (\)	220
7 на цифровой клавиатуре	103	Правая квадратная скобка	221
		Одинарная кавычка	222

Коды символов

Событие `keypress` сигнализирует о том, что нажатие клавиши изменило текст на экране. Для клавиш, которые добавляют или удаляют символ, оно генерируется во всех браузерах, а способ обработки нажатий других клавиш зависит от браузера. Поскольку реализация спецификации DOM Level 3 Events началась сравнительно недавно, в браузерах существуют заметные различия.

В браузерах у объекта `event` есть свойство `charCode`, которое задается только для события `keypress`, при этом ему назначается ASCII-код символа нажатой клавиши. Свойство `keyCode` в этом случае обычно имеет значение 0 или содержит код нажатой клавиши. В Internet Explorer 8 и более ранних версий и Opera ASCII-код символа передается в свойстве `keyCode`. Таким образом, чтобы получить код символа

кроссбраузерным способом, нужно сначала проверить, используется ли свойство charCode, и если нет, возвратить вместо него значение keyCode:

```
var EventUtil = {

    // другой код

    getCharCode: function(event) {
        if (typeof event.charCode == "number") {
            return event.charCode;
        } else {
            return event.keyCode;
        }
    },

    // другой код
};
```

Этот метод проверяет, содержит ли свойство charCode число (если свойство не поддерживается, оно не определено). Если да, метод возвращает его, в противном случае возвращается значение keyCode. Использовать метод можно следующим образом:

```
let textbox = document.getElementById("myText");
textbox.addEventListener("keypress", (event) => {
    console.log(EventUtil.getCharCode(event));
});
```

Когда код знака получен, можно преобразовать его в фактический символ методом String.fromCharCode().

Изменения в DOM Level 3

В спецификации DOM Level 3 Events события клавиатуры претерпели ряд изменений. Например, свойство charCode заменено свойствами key и char.

Свойство key предназначено для замены свойства keyCode и содержит строку. При нажатии символьной клавиши ему присваивается символ (например "k" или "M"), а при нажатии несимвольной клавиши — название этой клавиши (например "Shift" или "Down"). Свойство char при нажатии символьной клавиши работает так же, а если нажимается несимвольная клавиша, получает значение null.

Internet Explorer 9 поддерживает свойство key, но не char. Safari 5 и Chrome поддерживают свойство keyIdentifier, которое возвращает то же значение, что и key, если нажимается несимвольная клавиша (например, Shift). Для символьных клавиш свойство keyIdentifier возвращает строку формата «U+0000», содержащую символ в кодировке Юникод.

```
let textbox = document.getElementById("myText");
textbox.addEventListener("keypress", (event) => {
    let identifier = event.key || event.keyIdentifier;
    if (identifier) {
        console.log(identifier);
    }
});
```

Поскольку свойства `key`, `keyIdentifier` и `char` доступны не во всех браузерах, использовать их не рекомендуется.

В DOM Level 3 Events также добавлено числовое свойство `location`, указывающее, где была нажата клавиша. Его возможные значения таковы: 0 — клавиатура, предлагаемая по умолчанию; 1 — левая часть клавиатуры (например, левая клавиша Alt); 2 — правая часть клавиатуры (например, правая клавиша Shift); 3 — числовая клавиатура; 4 — клавиатура мобильного устройства (виртуальная); 5 — джойстик (например, контроллер Nintendo Wii). Это свойство доступно в Internet Explorer 9. В Safari 5 и Chrome есть идентичное свойство `keyLocation`, но из-за ошибки оно может содержать только число 3, если нажата клавиша на цифровой клавиатуре, и 0 в остальных случаях.

```
let textbox = document.getElementById("myText");
textbox.addEventListener("keypress", (event) => {
    let loc = event.location || event.keyLocation;
    if (loc) {
        console.log(loc);
    }
});
```

Из-за ограниченной поддержки использовать свойство `location` в кроссбраузерном коде не следует.

Наконец, к объекту `event` добавлен также метод `getModifierState()`. Он принимает строковое значение `"Shift"`, `"Control"`, `"Alt"`, `"AltGraph"` или `"Meta"`, указывающее клавишу-модификатор, которую нужно проверить. Если указанный модификатор активен (клавиша нажата), метод возвращает `true`, иначе — `false`:

```
let textbox = document.getElementById("myText");
textbox.addEventListener("keypress", (event) => {
    if (event.getModifierState) {
        console.log(event.getModifierState("Shift"));
    }
});
```

Некоторые из этих сведений можно также получить с помощью свойств `shiftKey`, `altKey`, `ctrlKey` и `metaKey` объекта `event`.

Событие textInput

В DOM Level 3 Events определено событие `textInput`, которое генерируется при вводе символа в редактируемой области. Оно предназначено для замены события `keypress` и работает немного иначе. Во-первых, событие `keypress` генерируется для любого элемента, который может получить фокус ввода, а `textInput` — только для областей с возможностью редактирования. Во-вторых, событие `textInput` генерируется только для символьных клавиш, а событие `keypress` — для клавиш, способных изменять текст как угодно (включая Backspace).

Для события `textInput` в объекте `event` доступно свойство `data`, которое содержит введенный символ (а не код символа). Символ представляется точно, например в результате нажатия клавиши S без клавиши Shift свойство `data` получает значение `"s"`, а при нажатой клавише Shift — значение `"S"`.

Событие `textInput` можно обработать следующим образом:

```
let textbox = document.getElementById("myText");
textbox.addEventListener("textInput", (event) => {
    console.log(event.data);
});
```

В этом примере символ, введенный в текстовом поле, отображается в сообщении в консоли.

У объекта `event` есть также свойство `inputMethod`, которое показывает способ ввода текста в элемент управления:

- `0` — браузер не смог определить способ ввода;
- `1` — текст был введен с клавиатуры;
- `2` — текст был вставлен;
- `3` — текст был добавлен путем перетаскивания;
- `4` — текст был введен с помощью редактора метода ввода (IME);
- `5` — текст был добавлен путем выбора элемента формы;
- `6` — текст был введен от руки (например, с помощью стилуса);
- `7` — текст был введен голосом;
- `8` — текст был введен несколькими способами;
- `9` — текст был добавлен программно.

С помощью этого свойства можно узнать, как текст был введен в элемент управления, чтобы проверить, допустим ли он.

События клавиатуры на других устройствах

Игровая приставка Nintendo Wii генерирует события клавиатуры при нажатии некоторых кнопок на пульте дистанционного управления Wii. Коды клавиш показаны на рис. 17.7.

События клавиатуры генерируются при нажатии крестообразной кнопки (коды клавиш 175–178), кнопок с минусом (170), плюсом (174), а также кнопок `1` (172) и `2` (173). Обработать нажатия кнопок A, B и Home, а также кнопки питания невозможно.

Мобильные сенсорные устройства генерируют события клавиатуры при использовании экранной клавиатуры.

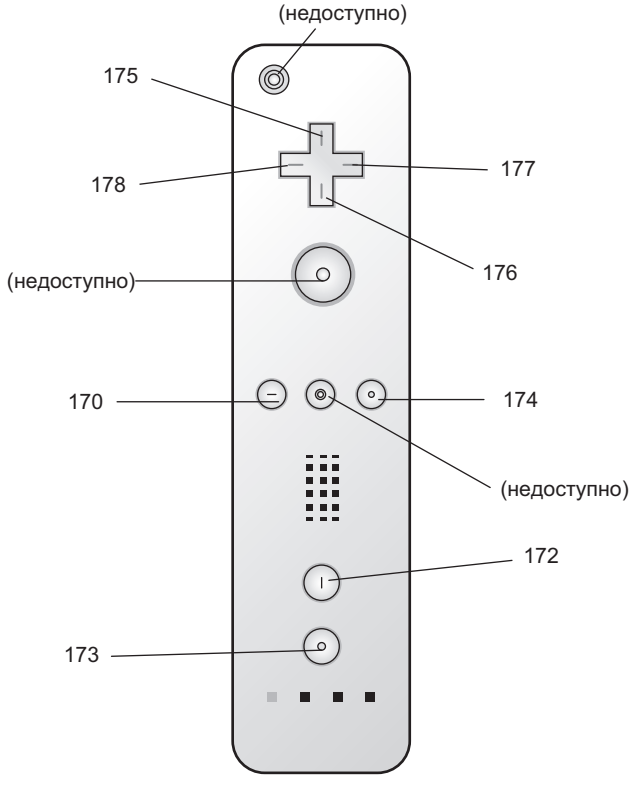

Рис. 17.7

События композиции

События композиции (composition events) представлены в DOM Level 3 Events для обработки сложных сочетаний клавиш, которые обычно используются в редакторах метода ввода (Input Method Editor, IME) для ввода символов, отсутствующих на физической клавиатуре. Например, с помощью IME на клавиатуре с латиницей можно вводить символы японского алфавита. Для ввода единственного символа в IME часто требуется нажать несколько клавиш. Три события композиции помогают распознавать такой ввод и работать с ним:

- `compositionstart` — генерируется при открытии IME-системы композиции текста, отмечая начало ввода символов;
- `compositionupdate` — генерируется при вставке нового символа в поле ввода;
- `compositionend` — генерируется при закрытии системы композиции текста, информируя о возврате к обычному вводу с клавиатуры.

События композиции во многом похожи на события редактирования текста. Целевым элементом события композиции является поле ввода, принимающее текст,

а единственное дополнительное свойство `data` содержит одно из следующих значений:

- при обработке события `compositionstart` — редактируемый текст (например, если текст был выделен и сейчас будет заменен);
- при обработке события `compositionupdate` — вставляемый символ;
- при обработке события `compositionend` — весь текст, введенный в течение сеанса композиции.

Как и события редактирования текста, события композиции можно использовать для фильтрации ввода. Назначить им обработчики можно следующим образом:

```
let textbox = document.getElementById("myText");
textbox.addEventListener("compositionstart", (event) => {
    console.log(event.data);
});
textbox.addEventListener("compositionupdate", (event) => {
    console.log(event.data);
});
textbox.addEventListener("compositionend", (event) => {
    console.log(event.data);
});
```

События изменения DOM-структуры

События изменения DOM-структуры (mutation events) были разработаны для уведомлений об изменениях частей DOM-структуры.

> **ПРИМЕЧАНИЕ** Эти события устарели, и их поддержка постепенно прекращается. Данная функция заменена наблюдателями за изменениями, описанными в главе 12 «Объектная модель документа».

События HTML5

В спецификации DOM определены не все события, поддерживаемые браузерами. Многие браузеры реализуют уникальные события для решения специфических задач. HTML5 определяет целый ряд дополнительных событий, которые в идеале должны поддерживаться браузерами, и в этом разделе мы обсудим некоторые из них. Имейте в виду, что это не исчерпывающий перечень всех событий.

Событие contextmenu

Контекстные меню, которые появляются при щелчке правой кнопкой мыши, появились в Windows 95 и вскоре были взяты на вооружение веб-разработчиками, но сразу возникли проблемы. Для использования контекстного меню требовался какой-то механизм, уведомляющий о том, что меню нужно вывести на экран (в Windows

оно вызывается правой кнопкой мыши, а в Mac — нажатием клавиши Ctrl с одновременным щелчком мыши), и позволяющий отменить вывод контекстного меню, предлагаемого по умолчанию. Для решения этих задач было реализовано событие `contextmenu`, генерируемое перед выводом контекстного меню.

Событие `contextmenu` всплывает, так что его можно обработать для всей страницы, назначив обработчик объекту `document`. Целевым элементом события является элемент, меню которого вызывается. Чтобы отменить событие `contextmenu`, нужно использовать метод `event.preventDefault()` в браузерах, соответствующих DOM, или присвоить свойству `event.returnValue` значение `false` в Internet Explorer 8 и более ранних версий. Событие `contextmenu` считается событием мыши, а потому имеет все свойства, связанные с положением указателя. Как правило, разработчики выводят на экран собственные контекстные меню в обработчике `oncontextmenu` и скрывают их в обработчике `onclick`. Возьмем для примера следующую HTML-страницу:

```
<!DOCTYPE html>
<html>
<head>
    <title>ContextMenu Event Example</title>
</head>
<body>
    <div id=" myDiv">Right click or Ctrl+click me to get a custom
        context menu. Click anywhere else to get the default context
        menu.</div>
    <ul id="myMenu" style="position:absolute;visibility:hidden;
        background-color:silver">
        <li><a href="http://www.nczonline.net">Сайт Николаса</a></li>
        <li><a href="http://www.wrox.com">Сайт Wrox</a></li>
        <li><a href="http://www.yahoo.com">Yahoo!</a></li>
    </ul>
</body>
</html>
```

Этот код создает элемент `<div>` с элементом `<ul>` в качестве пользовательского контекстного меню, которое сначала скрыто. Чтобы этот пример заработал, нужен следующий JS-код:

```
window.addEventListener("load", (event) => {
    let div = document.getElementById("myDiv");

    div.addEventListener("contextmenu", (event) => {
        event.preventDefault();

        let menu = document.getElementById("myMenu");
        menu.style.left = event.clientX + "px";
        menu.style.top = event.clientY + "px";
        menu.style.visibility = "visible";
    });

    document.addEventListener("click", (event) => {
        document.getElementById("myMenu").style.visibility = "hidden";
    });
});
```

Здесь элементу `<div>` назначается обработчик события `contextmenu`. Сначала он отменяет действие, предлагаемое по умолчанию, блокируя вывод контекстного меню браузера, а затем помещает элемент `<ul>` в позиции, заданной свойствами `clientX` и `clientY` объекта `event`. После этого для вывода меню его свойству `visibility` присваивается значение `"visible"`. Кроме того, объекту `document` назначается обработчик события `click`, который скрывает меню при щелчке (так работают системные контекстные меню).

Хотя этот пример очень прост, подобный код лежит в основе всех пользовательских контекстных меню на веб-сайтах. Применив к меню CSS-стили, можно добиться еще лучшего результата.

Событие beforeunload

Событие `beforeunload` генерируется для объекта `window` перед началом выгрузки страницы из браузера, чтобы можно было отменить выгрузку и продолжить работу со страницей. Просто отменить это событие нельзя, потому что в результате пользователь будет заблокирован на странице. Вместо этого в обработчике события следует вывести сообщение, предлагающее пользователю закрыть страницу или остаться на ней (рис. 17.8).

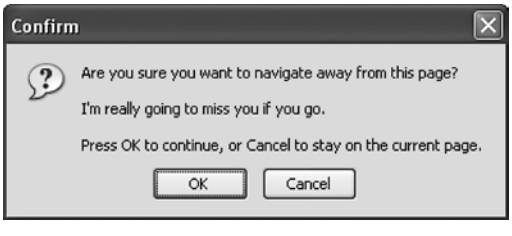

Рис. 17.8

Чтобы показать такое диалоговое окно, назначьте свойству `event.returnValue` строку, которую нужно вывести в окне (для Internet Explorer и Firefox), и возвратите ее как значение функции (для Safari и Chrome):

```
window.addEventListener("beforeunload", (event) => {
    let message = "I'm really going to miss you if you go.";
    event.returnValue = message;
    return message;
});
```

Событие DOMContentLoaded

Событие `load` объекта `window` генерируется при завершении загрузки страницы, на что может потребоваться некоторое время, если страница содержит много внешних ресурсов. Событие `DOMContentLoaded` генерируется после завершения формирования DOM-дерева независимо от изображений, JavaScript- и CSS-файлов, а также других подобных ресурсов. В сравнении с событием `load` событие `DOMContentLoaded`

возникает на более раннем этапе загрузки страницы, позволяя пользователям быстрее приступить к работе с ней.

Обработчик события `DOMContentLoaded` можно подключить к объекту `document` или `window` (целевым элементом события на самом деле является `document`, хотя оно всплывает к `window`). Вот пример обработчика этого события на `document`:

```
document.addEventListener("DOMContentLoaded", (event) => {
    console.log("Content loaded");
});
```

Объект `event` не предоставляет дополнительных сведений о событии `DOMContentLoaded`.

Событие `DOMContentLoaded` обычно используется для подключения обработчиков событий или выполнения других манипуляций с DOM-структурой. Оно всегда генерируется до события `load`.

В браузерах, которые не поддерживают событие `DOMContentLoaded`, можно попытаться имитировать его, задав во время загрузки страницы тайм-аут с нулевой задержкой:

```
setTimeout(() => {
    // подключите здесь обработчики событий
}, 0);
```

По сути, этот код заставляет вызвать функцию сразу же по завершении текущего JavaScript-процесса, который загружает и конструирует страницу. Совпадет ли это по времени с событием `DOMContentLoaded`, зависит от браузера и другого кода на странице. Чтобы тайм-аут сработал правильно, он должен быть первым на странице, но даже это не гарантирует, что он всегда будет предшествовать событию `load`.

Событие readystatechange

Событие `readystatechange` было впервые определено в Internet Explorer для нескольких объектов DOM-документа. Оно предоставляет сведения о состоянии загрузки документа или элемента, но часто работает с ошибками. У каждого объекта, поддерживающего событие `readystatechange`, есть свойство `readyState`, которое может иметь одно из пяти строковых значений:

- `uninitialized` — объект существует, но не инициализирован;
- `loading` — объект загружает данные;
- `loaded` — объект завершил загрузку данных;
- `interactive` — объект загружен не полностью, но с ним можно взаимодействовать;
- `complete` — объект полностью загружен.

Казалось бы, все просто, но проблема в том, что не все объекты проходят все эти этапы. В документации сказано, что объекты могут пропускать неважные этапы, но не указано, какие этапы считаются важными для тех или иных объектов. Это

означает, что событие readystatechange часто генерируется менее четырех раз и что последовательности значений readyState могут быть разными.

Для объекта document, у которого свойство readyState изменяет значение на "interactive", событие readystatechange генерируется примерно в то же время, что и событие DOMContentLoaded. Интерактивный этап наступает, когда все DOM-дерево загружено и с ним можно безопасно взаимодействовать. Доступность изображений и других внешних ресурсов в это время не гарантируется. Событие readystatechange можно обработать так:

```
document.addEventListener("readystatechange", (event) => {
    if (document.readyState == "interactive") {
        console.log("Content loaded");
    }
});
```

Объект event не предоставляет дополнительных сведений об этом событии, а его свойство target не задано.

Порядок генерирования этого события относительно события load не определен. На страницах с многочисленными или крупными внешними ресурсами интерактивный этап наступает задолго до события load, а если внешних ресурсов мало или они не слишком объемны, событие load может возникнуть раньше, чем readystatechange.

Более того, интерактивный этап может наступить до или после полной загрузки страницы (значение "complete"). Чем больше внешних ресурсов содержит страница, тем выше вероятность того, что интерактивный этап будет достигнут до ее полной загрузки. Следовательно, чтобы пользователи могли как можно раньше приступить к работе со страницей, нужно проверить оба условия:

```
document.addEventListener("readystatechange", (event) => {
    if (document.readyState == "interactive" ||
        document.readyState == "complete") {
        document.removeEventListener("readystatechange", arguments.callee);
        console.log("Content loaded");
    }
});
```

При вызове этого обработчика мы первым делом хотим убедиться, что документ уже загружен или достиг интерактивного этапа. Если это так, мы удаляем обработчик события, чтобы исключить его запуск для других этапов. Поскольку обработчик является анонимной функцией, для его удаления используется свойство arguments.callee. В конце в консоли выводится сообщение о том, что контент загружен. Такой код позволяет максимально приблизиться по времени к событию DOMContentLoaded.

ПРИМЕЧАНИЕ Хотя с помощью события readystatechange можно имитировать событие DOMContentLoaded, они не идентичны. Порядок генерирования событий load и readystatechange зависит от страницы.

События pageshow и pagehide

Firefox и Opera поддерживают так называемый *кеш состояния страниц* (back-forward cache, bfcache), который ускоряет смену страниц при щелчках на кнопках Назад и Вперед. Этот кеш хранит не только данные страницы, но и ее DOM- и JavaScript-состояния — по сути, она полностью удерживается в памяти. Если страница находится в этом кеше, событие `load` при переходе к ней не генерируется. Обычно это не проблема, потому что доступно все состояние страницы, но разработчики Firefox все же решили добавить некоторые события, предоставляющие больший контроль над кешем состояния страниц.

Первое из них, `pageshow`, генерируется при выводе страницы на экран из кеша состояния страниц или иным образом. Для страницы, загружаемой из интернета, событие `pageshow` генерируется после события `load`, а для страницы из кеша — сразу после полного восстановления ее состояния. Хотя целевым элементом этого события является объект `document`, обрабатывать его следует у объекта `window`. Пример отслеживания таких событий:

```
(function() {
    let showCount = 0;

    window.addEventListener("load", () => {
        console.log("Load fired");
    });

    window.addEventListener("pageshow", () => {
        showCount++;
        console.log(`Show has been fired ${showCount} times.`);
    });
})();
```

В этом примере используется закрытая область видимости, чтобы переменная `showCount` не была доступна глобально. Если страница загружается впервые, переменная `showCount` имеет значение 0, которое затем увеличивается при каждом событии `pageshow`. Каждый раз, когда вы покинете страницу с этим кодом и возвратитесь к ней с помощью кнопки Назад, значение `showCount` будет увеличиваться, потому что оно кешируется в памяти вместе с остальным состоянием страницы. Если щелкнуть в браузере на кнопке обновления страницы, значение `showCount` обнулится, потому что страница будет полностью перезагружена.

Кроме обычных свойств, объект `event` события `pageshow` содержит свойство `persisted`, которое равно `true`, если страница сохранена в кеше состояния страниц, и `false` в противном случае, например:

```
(function() {
    let showCount = 0;

    window.addEventListener("load", () => {
        console.log("Load fired");
    });
```

```
    window.addEventListener("pageshow", () => {
        showCount++;
        console.log(`Show has been fired ${showCount} times.`,
            `Persisted? ${event.persisted}`);    });
})();
```

Свойство `persisted` можно использовать для выбора того или иного действия в зависимости от того, кеширована ли страница.

Событие `pagehide` возникает непосредственно перед событием `unload` при выгрузке страницы из браузера. Как и событие `pageshow`, оно генерируется для объекта `document`, но его обработчик следует подключать к объекту `window`. Его объект `event` также содержит свойство `persisted`, хотя используется оно немного иначе. Пример проверки свойства `event.persisted`:

```
window.addEventListener("pagehide", (event) => {
    console.log("Hiding. Persisted? " + event.persisted);
});
```

При обработке события `pagehide` можно использовать свойство `persisted` для изменения логики сценария. У события `pageshow` свойство `persisted` равно `true`, если страница была загружена из кеша состояния страниц, а у события `pagehide` — если страница будет сохранена в кеше после выгрузки. Таким образом, при первом событии `pageshow` свойство `persisted` всегда равно `false`, а при первом событии `pagehide` оно всегда равно `true` (конечно, при условии, что страница может быть сохранена в кеше).

> **ПРИМЕЧАНИЕ** Страницы с обработчиком события `onunload` автоматически исключаются из кеша состояния страниц, даже если этот обработчик пуст. Дело в том, что событие `onunload` обычно используется для отмены действий, выполненных в обработчике события `onload`, поэтому пропуск события `onload` при следующем выводе страницы на экран может привести к сбою.

Событие hashchange

В HTML5 представлено событие `hashchange`, уведомляющее об изменении хеша URL-адреса (часть после знака `#`). Разработчики часто используют хеш URL-адреса для хранения данных состояния или навигации в Ajax-приложениях.

Обработчик события `hashchange` подключается к объекту `window` и вызывается при любом изменении хеша URL-адреса. Объект `event` этого события имеет дополнительные свойства `oldURL` и `newURL`, которые содержат полный URL-адрес (включая хеш) до и после изменения. Проивоположные URL-адреса отслеживаются в следующем примере:

```
window.addEventListener("hashchange", (event) => {
    console.log(`Old URL: ${event.oldURL}, New URL: ${event.newURL}`);
});
```

Для определения текущего хеша лучше использовать объект `location`:

```
window.addEventListener("hashchange", (event) => {
    console.log(`Current hash: ${location.hash}`);
});
```

События устройств

С появлением смартфонов и планшетных компьютеров сфера применения браузеров значительно расширилась. Чтобы разработчики могли узнать, как используется то или иное устройство, были определены новые события — так называемые *события устройств* (device events). В 2011 г. консорциум W3C приступил к работе над новой спецификацией этих событий под названием DeviceOrientation Event, призванной охватить постоянно растущее количество устройств.

Событие orientationchange

Событие `orientationchange` было реализовано в мобильной версии Safari, чтобы разработчики могли определять, когда пользователь изменяет альбомную ориентацию экрана на книжную и наоборот. В мобильной версии Safari доступно свойство `window.orientation`, которое может иметь одно из трех значений: 0 для книжного режима, 90 для альбомной ориентации с поворотом влево (кнопка Home находится справа) и –90 для альбомной ориентации с поворотом вправо (кнопка Home находится слева). В документации также упоминается значение 180, которое указывает, что устройство перевернуто, но это значение пока не поддерживается. Разные варианты ориентации устройства показаны на рис. 17.9.

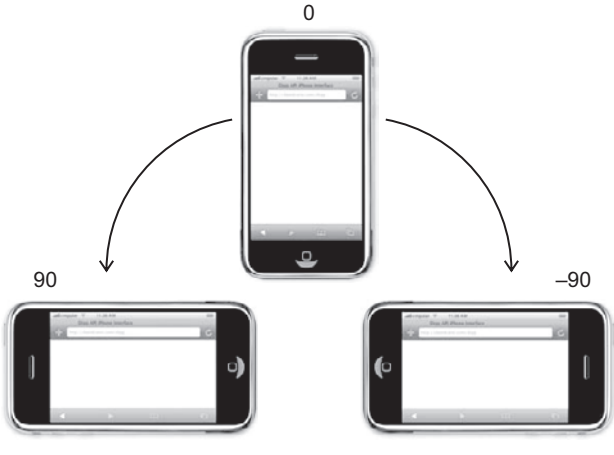

Рис. 17.9

При изменении ориентации генерируется событие `orientationchange`. Его объект `event` не содержит никакой полезной информации, поскольку все, что нужно знать, это значение `window.orientation`. Обычно это событие используется следующим образом:

```
window.addEventListener("load", (event) => {
    let div = document.getElementById("myDiv");
    div.innerHTML = "Current orientation is " + window.orientation;

    window.addEventListener("orientationchange", (event) => {) {
        div.innerHTML = "Current orientation is: " + window.orientation;
    });
});
```

Этот код сначала выводит значение первоначальной ориентации в обработчике события `load`, а затем назначает обработчик события `orientationchange`, при возникновении которого на странице выводится новое значение ориентации.

Событие `orientationchange` и свойство `window.orientation` поддерживают все устройства с системой iOS.

> **ПРИМЕЧАНИЕ** Поскольку `orientationchange` считается событием `window`, можно также назначить обработчик события, добавив атрибут `onorientationchange` к элементу `<body>`.

Событие deviceorientation

Событие `deviceorientation` определено в спецификации DeviceOrientationEvent. Оно генерируется для объекта `window` при изменении показаний акселерометра. Помните, что это событие предназначено для описания ориентации устройства в пространстве, а не его движения.

Устройство располагается в трехмерном пространстве с осями *x*, *y* и *z*. Ось *x* пересекает устройство слева направо, ось *y* — снизу вверх, а ось *z* — сзади вперед (рис. 17.10). Когда устройство покоится на горизонтальной поверхности, все три параметра ориентации равны нулю.

Рис. 17.10

Событие `deviceorientation` возвращает сведения о том, насколько углы наклона каждой оси отличаются от состояния покоя. В объекте `event` содержит пять свойств:

- `alpha` — угол поворота оси *y* вокруг оси *z* в градусах (число с плавающей точкой в интервале от 0 до 360);
- `beta` — угол поворота оси *z* вокруг оси *x* в градусах (число с плавающей точкой в интервале от –180 до 180);
- `gamma` — угол поворота оси *z* вокруг оси *y* в градусах (число с плавающей точкой в интервале от –90 до 90);
- `absolute` — логическое значение, указывающее, возвращает ли устройство абсолютные значения;
- `compassCalibrated` — логическое значение, указывающее, правильно ли откалиброван компас устройства.

На рис. 17.11 показано, как вычисляются значения `alpha`, `beta` и `gamma`.

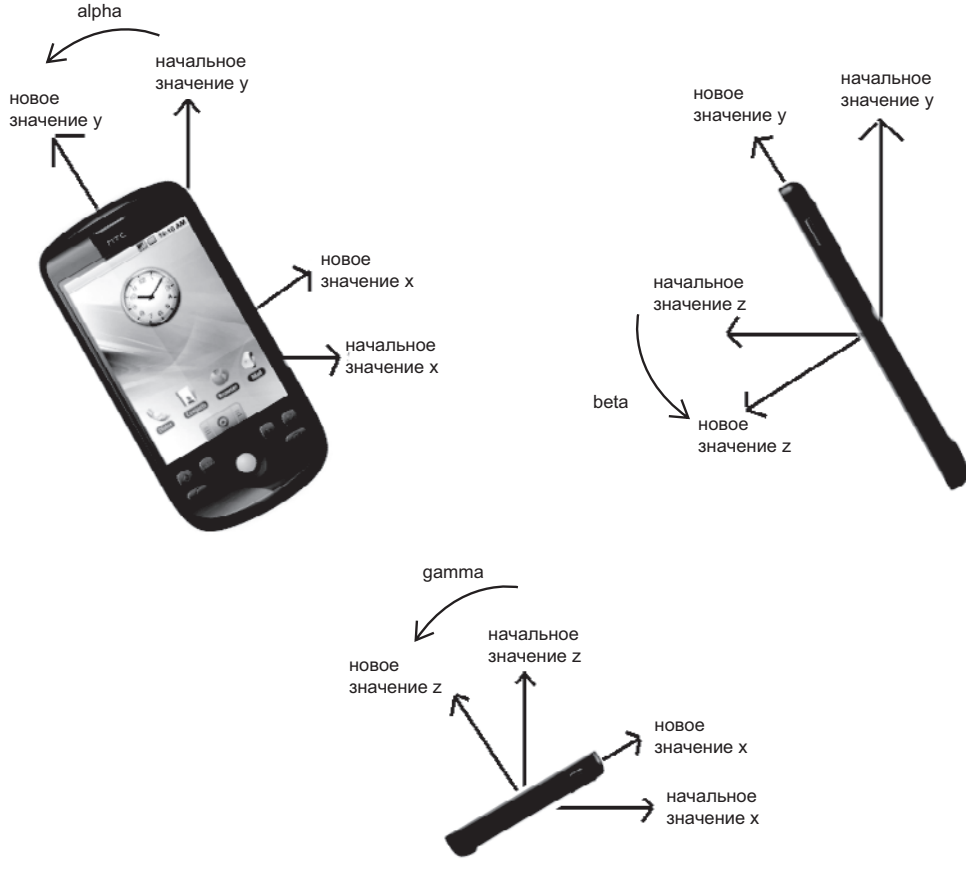

Рис. 17.11

В следующем примере на экран просто выводятся значения свойств `alpha`, `beta` и `gamma`:

```
window.addEventListener("deviceorientation", (event) => {
    let output = document.getElementById("output");
    output.innerHTML =
        `Alpha=${event.alpha}, Beta=${event.beta], Gamma=${event.gamma$}<br>`;
});
```

Эти данные можно использовать для переупорядочения или перемещения элементов на экране при изменении ориентации устройства. Например, следующий код поворачивает элемент:

```
window.addEventListener("deviceorientation", (event) => {
    let arrow = document.getElementById("arrow");
    arrow.style.webkitTransform = `rotate(${Math.round(event.alpha)}deg)`;
});
```

Этот пример работает только в мобильных браузерах WebKit, потому что в нем используется фирменное свойство `webkitTransform` (временная версия стандартного CSS-свойства `transform`). В этом коде элемент `"arrow"` поворачивается в соответствии со значением `event.alpha`, что делает его похожим на компас. Для плавного поворота элемента выполняется CSS3-преобразование с округленным значением свойства.

Событие devicemotion

Спецификация DeviceOrientationEvent включает также событие `devicemotion`, которое описывает перемещение устройства, а не только изменение его ориентации. Например, с его помощью можно определить, что устройство падает или используется во время ходьбы.

Объект `event` этого события содержит следующие дополнительные свойства:

- `acceleration` — объект со свойствами `x`, `y` и `z`, которые показывают ускорение в каждом направлении без учета гравитации;
- `accelerationIncludingGravity` — объект со свойствами `x`, `y` и `z`, которые показывают ускорение в каждом направлении с учетом естественной гравитации по оси *z*;
- `interval` — интервал между событиями `devicemotion` в миллисекундах (это значение должно быть постоянным);
- `rotationRate` — объект со свойствами `alpha`, `beta` и `gamma`, показывающими ориентацию устройства.

Если определить значение `acceleration`, `accelerationIncludingGravity` или `rotationRate` невозможно, оно считается равным `null`, поэтому перед использованием любого из этих свойств нужно проверять, не равно ли оно `null`, например:

```
window.addEventListener("devicemotion", (event) => {
    let output = document.getElementById("output");
    if (event.rotationRate !== null) {
```

```
        output.innerHTML += `Alpha=${event.rotationRate.alpha}` +
                            `Beta=${event.rotationRate.beta}` +
                            `Gamma=${event.rotationRate.gamma}`;
    }
});
```

События касаний и жестов

Поскольку устройства с iOS используются без мыши и клавиатуры, обычных событий мыши и клавиатуры было недостаточно для создания полноценных интерактивных веб-страниц для мобильного браузера Safari. В связи с этим разработчики Safari для iOS добавили в него несколько фирменных специализированных событий. С выходом WebKit для Android многие такие события стали стандартом де-факто, что подтолкнуло консорциум W3C к разработке спецификации Touch Events. События, описанные в этом разделе, работают только на устройствах с поддержкой сенсорного ввода.

События касаний

Смартфон iPhone 3G с системой iOS 2.0 поставлялся с новой версией Safari, поддерживающей несколько новых событий, связанных с сенсорным вводом. Позднее такие же события были реализованы в браузере для Android. *События касаний* (touch events) генерируются, когда пользователь касается экрана устройства пальцем, перемещает палец и отрывает его от экрана:

- `touchstart` — генерируется при касании экрана пальцем, даже если пользователь уже касается экрана другим пальцем;
- `touchmove` — постоянно генерируется при перемещении пальца по экрану (вызов метода `preventDefault()` во время этого события предотвращает прокрутку);
- `touchend` — генерируется, когда пользователь отрывает палец от экрана;
- `touchcancel` — генерируется, когда система прекращает отслеживать касание (из документации не понятно, когда может произойти это событие).

Каждое из этих событий всплывает и может быть отменено. Хотя события касаний не входят в спецификацию DOM, они реализованы в соответствии с DOM. Так, объект `event` каждого события касания содержит свойства, общие для событий мыши: `bubbles`, `cancelable`, `view`, `clientX`, `clientY`, `screenX`, `screenY`, `detail`, `altKey`, `shiftKey`, `ctrlKey` и `metaKey`.

В дополнение к этим общим DOM-свойствам события касания имеют три свойства для отслеживания касаний:

- `touches` — массив объектов `Touch`, которые представляют касания, отслеживаемые в текущий момент;
- `targetTouches` — массив объектов `Touch`, специфичных для целевого элемента события;
- `changedTouches` — массив объектов `Touch`, измененных в результате последнего действия пользователя.

Каждый объект Touch, в свою очередь, имеет следующие свойства:

- clientX — координата *x* целевого элемента касания в области просмотра;
- clientY — координата *y* целевого элемента касания в области просмотра;
- identifier — уникальный идентификатор касания;
- pageX — координата *x* целевого элемента касания на странице;
- pageY — координата *y* целевого элемента касания на странице;
- screenX — координата *x* целевого элемента касания на экране;
- screenY — координата *y* целевого элемента касания на экране;
- target — целевой элемент касания (DOM-узел).

С помощью этих свойств можно отслеживать касание экрана, например:

```
function handleTouchEvent(event) {

    // работает только для одного касания
    if (event.touches.length == 1)

        let output = document.getElementById("output");
        switch(event.type) {
            case "touchstart":
                output.innerHTML = `<br>Touch started:` +
                                   `(${event.touches[0].clientX}` +
                                   ` ${event.touches[0].clientY})`;
                break;
            case "touchend":
                output.innerHTML += `<br>Touch ended:` +
                                    `(${event.changedTouches[0].clientX}` +
                                    ` ${event.changedTouches[0].clientY})`;
                break;
            case "touchmove":
                event.preventDefault();    // предотвращение прокрутки
                output.innerHTML += `<br>Touch moved:` +
                                    `(${event.changedTouches[0].clientX}` +
                                    ` ${event.changedTouches[0].clientY})`;
                break;
        }
    }
}

document.addEventListener("touchstart", handleTouchEvent);
document.addEventListener("touchend", handleTouchEvent);
document.addEventListener("touchmove", handleTouchEvent);
```

Ради простоты примера здесь мы отслеживаем только одно активное касание. Когда происходит событие touchstart, обработчик выводит координаты касания в элементе . При событии touchmove мы отменяем действие, предлагаемое по умолчанию, чтобы страница не прокручивалась (перемещение пальца по экрану обычно сопровождается прокруткой), а затем выводим информацию об измененном касании. Наконец, при событии touchend выводятся финальные координаты касания. При

обработке события touchend коллекция touches пуста, так как активного касания больше нет, поэтому вместо нее нужно использовать коллекцию changedTouches.

Эти события генерируются для всех элементов документа, что позволяет работать с разными частями страницы по отдельности. Если пользователь касается элемента и тут же отпускает его без перемещения пальца, события генерируются в следующем порядке:

1. touchstart.
2. mouseover.
3. mousemove (один раз).
4. mousedown.
5. mouseup.
6. click.
7. touchend.

События жестов

В Safari для iOS 2.0 также были представлены события жестов. *Жест* (gesture) имеет место, если экрана касаются два пальца, и обычно вызывает изменение масштаба или поворот элемента. Событий жестов три:

- gesturestart — генерируется при касании экрана, если пользователь уже касается его другим пальцем;
- gesturechange — генерируется при изменении положения любого из пальцев, касающихся экрана;
- gestureend — генерируется, когда пользователь отрывает от экрана один из пальцев.

Эти события генерируются, только если пользователь касается элемента двумя пальцами. Чтобы генерировались события жестов, оба пальца должны находиться в пределах целевого элемента, которому назначены обработчики. Поскольку эти события всплывают, можно также обрабатывать все жесты на уровне документа. При использовании этого подхода целевым элементом события будет элемент, в пределах которого находятся оба пальца.

События касаний и жестов связаны. При касании экрана первым пальцем возникает событие touchstart. При касании экрана вторым пальцем сначала генерируется событие gesturestart, а за ним следует событие touchstart для этого пальца. При перемещении одного или обоих пальцев генерируется событие gesturechange, а когда пользователь отрывает один из пальцев от экрана, возникает событие gestureend, за которым следует событие touchend для этого пальца.

Как и при касаниях, при каждом жесте объект event содержит все стандартные свойства мыши: bubbles, cancelable, view, clientX, clientY, screenX, screenY, detail,

`altKey`, `shiftKey`, `ctrlKey` и `metaKey`. Кроме того, у него есть дополнительные свойства `rotation` и `scale`. Свойство `rotation` указывает угол поворота пальцев в градусах, при этом положительные значения соответствует повороту по часовой стрелке, а отрицательные — против (начальное значение равно 0). Свойство `scale` указывает изменение расстояния между пальцами. Оно начинается с единицы и увеличивается или уменьшается при раздвигании или сдвигании пальцев соответственно.

Эти события можно использовать следующим образом:

```
function handleGestureEvent(event) {
    let output = document.getElementById("output");
    switch(event.type) {
        case "gesturestart":
            output.innerHTML = `Gesture started: ` +
                               `rotation=${event.rotation},` +
                               `scale=${event.scale}`;
            break;
        case "gestureend":
            output.innerHTML += `Gesture ended: ` +
                                `rotation=${event.rotation},` +
                                `scale=${event.scale}`;
            break;
        case "gesturechange":
            output.innerHTML += `Gesture changed: ` +
                                `rotation=${event.rotation},` +
                                `scale=${event.scale}`;
            break;
    }
}

document.addEventListener("gesturestart", handleGestureEvent, false);
document.addEventListener("gestureend", handleGestureEvent, false);
document.addEventListener("gesturechange", handleGestureEvent, false);
```

Как и пример с событиями касаний, этот код просто подключает ко всем трем событиям один и тот же обработчик, в котором выводит сведения о каждом из них.

> **ПРИМЕЧАНИЕ** События касаний тоже возвращают свойства `rotation` и `scale`, но их значения изменяются, только если пользователь касается экрана двумя пальцами. Применять в подходящих ситуациях события жестов обычно проще, чем управлять всеми взаимодействиями с помощью событий касаний.

Справка по событиям

Ниже приведен список всех доступных событий браузера, определенных в спецификации DOM, HTML5 и любых других опубликованных в настоящее время спецификациях, которые описывают поведение событий. Они классифицируются по API и/или спецификации, в которой появляются.

ПРИМЕЧАНИЕ В этот список не вошли некоторые категории, которые содержат только специфичные для поставщика события.

События Ambient Light
devicelight

События кеша приложения
cached
checking
downloading
noupdate
obsolete
updateready

События Audio Channels API
headphoneschange
mozinterruptbegin
mozinterruptend

События Battery API
chargingchange
chargingtimechange
dischargingtimechange
levelchange

События Broadcast Channel API
message

События Channel Messaging API
message

События Clipboard API
beforecopy
beforecut
beforepaste
copy
cut
paste

События Contacts API
contactchange
error
success

События CSS Font Loading API
loading
loadingdone
loadingerror

События CSSOM
animationend
animationiteration
animationstart
transitionend

События просмотра CSSOM
resize
scroll

События ориентации устройств
compassneedscalibration
devicemotion
deviceorientation

События Device Storage API
Change

События DOM
abort
beforeinput
blur
click
compositionend
compositionstart
compositionupdate
dblclick
error
focus
focusin
focusout
input
keydown
keypress
keyup
load
mousedown
mouseenter
mouseleave
mousemove
mouseout
mouseover
mouseup
resize
scroll
select
unload
wheel

События Download API
statechange

События расширения зашифрованных носителей
encrypted
keystatuschange
message
waitingforkey

События Engineering Mode API
message

События File API
abort
error
load
loadend
loadstart
progress

События File System API
error
writeend

События FMRadio API
antennaavailablechange
disabled
enabled
frequencychange

События Fullscreen API
fullscreenchange
fullscreenerror

События Gamepad API
gamepadconnected
gamepaddisconnected

События HTML DOM
DOMContentLoaded
abort
afterprint
afterscriptexecute
beforeprint
beforescriptexecute
beforeunload
blur
cancel
canplay
canplaythrough
change
click
close
connect
contextmenu
durationchange
emptied
error
focus
hashchange
input
invalid
languagechange
load
loadeddata
loadedmetadata
loadend
loadstart
message
offline
online
open
pagehide
pageshow
play
playing
popstate
progress
readystatechange
rejectionhandled
reset
seeked
seeking
select
show
sort
stalled
storage
submit
suspend
timeupdate
toggle
unhandledrejection
unload
volumechange
waiting

События HTML Drag and Drop API
drag
dragend
dragenter
dragexit
dragleave
dragover
dragstart
drop

События IndexedDB
abort
blocked
close
complete

error
success
upgradeneeded
versionchange

События Inter-App Connection API
message

События захвата медиа и потоков
active
addtrack
devicechange
ended
inactive
mute
overconstrained
ratechange
removetrack
started
unmute

События расширений медиаресурсов
abort
addsourcebuffer
error
removesourcebuffer
sourceclose
sourceended
sourceopen
update
updateend
updatestart

События записей медиапотоков
dataavailable
error
pause
resume
start
stop

События Mobile Connection API
cardstatechange
icccardlockerror

События Mobile Messaging API
close
deliveryerror
deliverysuccess
error
failed
message
open
received
retrieving
sending
sent

События Network Information API
change

События Page Visibility API
visibilitychange

События Payment Request API
shippingaddresschange
shippingoptionchange

События Performance API
resourcetimingbufferfull
Pointer events
gotpointercapture
lostpointercapture
pointercancel
pointerdown
pointerenter
pointerleave
pointermove
pointerout
pointerover
pointerup

События Pointer Lock API
pointerlockchange
pointerlockerror

События Presentation API
change
sessionavailable
sessionconnect
Proximity events
deviceproximity
userproximity

События Push API
push
pushsubscriptionchange

События Screen Orientation API
change

События Selection API
selectionchange
selectstart

События сервера
error
message
open

События Service Workers API
activate
controllerchange
error
fetch
install
message
statechange
updatefound

События Settings API
settingchange

События Simple Push API
error
success

События Speaker Manager API
speakerforcedchange

События SVG
DOMAttrModified
DOMCharacterDataModified
DOMNodeInserted
DOMNodeInsertedIntoDocument
DOMNodeRemoved
DOMNodeRemovedFromDocument
DOMSubtreeModified
SVGAbort
SVGError
SVGLoad
SVGResize
SVGScroll
SVGUnload
SVGZoom
activate
beginEvent
click
endEvent
focusin
focusout
mousedown
mousemove
mouseout
mouseover
mouseup
repeatEvent

События TCP Socket API
connect
data
drain
error

События Time and Clock API
moztimechange

События касаний
touchcancel
touchend
touchmove
touchstart

События TV API
currentchannelchanged
currentsourcechanged
eitbroadcasted
scanningstatechanged

События UDP Socket API
message

События Web Audio API
audioprocess
complete
ended
loaded
message
nodecreate
statechange

События веб-компонентов
slotchange

События WebGL
webglcontextcreationerror
webglcontextlost
webglcontextrestored

События веб-манифестов
install

События Web MIDI API
midimessage
statechange

События веб-уведомлений
click
close
error
show

События WebRTC
addstream
close
datachannel
error

```
icecandidate
iceconnectionstatechange
icegatheringstatechange
identityresult
idpassertionerror
idpvalidationerror
isolationchange
message
negotiationneeded
open
peeridentity
removestream
signalingstatechange
tonechange

События Websockets API
close
error
message
open

События Web Speech API
audioend
audiostart
boundary
end_(SpeechRecognition)
end_(SpeechSynthesis)
error_(SpeechRecognitionError)
error_(SpeechSynthesis)
mark
nomatch
pause_(SpeechSynthesis)
result
resume
soundend
soundstart
speechend
speechstart
start_(SpeechRecognition)
start_(SpeechSynthesis)

События Web Storage API
storage

События Web Telephony API
incoming

События WebVR API
vrdisplayactivate
vrdisplayblur
vrdisplayconnected
vrdisplaydeactivate
vrdisplaydisconnected
vrdisplayfocus
vrdisplaypresentchange

События WebVTT
addtrack
change
cuechange
enter
exit
removetrack

События WiFi Information API
connectioninfoupdate
statuschange

События WiFi P2P API
disabled
enabled
peerinfoupdate
statuschange

События XMLHttpRequest
abort
error
load
loadend
loadstart
progress
readystatechange
timeout
```

ПАМЯТЬ И БЫСТРОДЕЙСТВИЕ

События обеспечивают интерактивность современных веб-приложений, но многие разработчики злоупотребляют ими. В языках, таких как C#, используемых для создания GUI, можно без заметного падения производительности обрабатывать событие `click` каждой кнопки, однако в JavaScript от количества обработчиков событий на странице напрямую зависит ее общее быстродействие. Это объясняется несколькими причинами. Во-первых, каждая функция является объектом и занимает память; чем больше объектов в памяти, тем хуже быстродействие. Во-вторых, первоначальное назначение всех обработчиков событий в DOM задерживает момент,

когда можно приступить к работе со страницей. Используя обработчики событий с умом, можно увеличить быстродействие страниц.

Делегирование событий

Проблему чрезмерного количества обработчиков событий можно решить за счет *делегирования событий* (event delegation). Этот прием, основанный на всплытии событий, предполагает, что все события определенного типа обрабатываются одним обработчиком. Например, событие `click` всплывает до уровня `document`, благодаря чему можно назначить один обработчик `onclick` всей странице, а не по одному для каждого элемента, поддерживающего щелчок. Рассмотрим следующий HTML-код:

```
<ul id="myLinks">
    <li id="goSomewhere">Go somewhere</li>
    <li id="doSomething">Do something</li>
    <li id="sayHi">Say hi</li>
</ul>
```

При щелчке на каждом из трех элементов выполняется то или иное действие. В традиционном подходе мы просто подключили бы к ним три обработчика событий:

```
let item1 = document.getElementById("goSomewhere");
let item2 = document.getElementById("doSomething");
let item3 = document.getElementById("sayHi");

item1.addEventListener("click", (event) => {
    location.href = "http://www.wrox.com";
});

item2.addEventListener("click", (event) => {
    document.title = "I changed the document's title";
});

item3.addEventListener("click", (event) => {
    console.log("hi");
});
```

Если бы в сложном веб-приложении это нужно было сделать для всех элементов, поддерживающих щелчок, только подключение обработчиков заняло бы несколько десятков, а то и сотен строк кода. Делегирование событий решает эту проблему, позволяя подключить единственный обработчик к наивысшей точке в DOM-дереве, например:

```
let list = document.getElementById("myLinks");

list.addEventListener("click", (event) => {
    let target = event.target;

    switch(target.id) {
        case "doSomething":
            document.title = "I changed the document's title";
            break;
```

```
        case "goSomewhere":
            location.href = "http://www.wrox.com";
            break;

        case "sayHi":
            console.log("hi");
            break;
    }
});
```

Этот код подключает единственный обработчик события `click` к элементу `<ul>`, к которому всплывают события всех элементов списка. Целевым элементом события является элемент списка, на котором был выполнен щелчок, так что для выбора нужного действия можно использовать его свойство `id`. В сравнении с предыдущим примером без делегирования событий этот код изначально менее требователен к ресурсам, потому что он получает единственный DOM-элемент и подключает только один обработчик события. Для пользователя ничего не меняется, но такой код требует гораздо меньше памяти. Потенциально он подходит для обработки любых событий кнопок (большинство событий мыши и клавиатуры).

Все события конкретного типа на странице можно также обрабатывать на уровне объекта `document`, если это практично. В сравнении с традиционным такой подход обеспечивает определенные преимущества.

- Обработчик события можно назначить объекту `document` в любой момент жизненного цикла страницы (дожидаться события `DOMContentLoaded` или `load` не требуется). Это означает, что щелчки на элементе будут правильно обрабатываться сразу после его визуализации.
- Для регистрации обработчиков событий требуется меньше времени и меньше обращений к DOM.
- Благодаря меньшему потреблению памяти повышается общее быстродействие страницы.

Для делегирования более всего подходят события `click`, `mousedown`, `mouseup`, `keydown`, `keyup` и `keypress`. События `mouseover` и `mouseout` всплывают, но их сложнее обработать правильно, потому что для этого часто требуется вычислить позицию элемента (событие `mouseout` генерируется при наведении указателя мыши на один из дочерних узлов элемента и при его перемещении за пределы элемента).

Удаление обработчиков событий

При назначении обработчиков событий элементам формируются связи между кодом, который выполняет браузер, и JS-кодом, предназначенным для взаимодействия со страницей. Чем больше таких связей, тем медленнее работает страница. Один из способов решения этой проблемы — делегировать обработку событий для уменьшения количества связей. Другой способ — удалять обработчики событий, ставшие ненужными. Если просто оставлять их в памяти, веб-приложение будет тратить гораздо больше ресурсов, чем могло бы.

Проблема возникает на двух этапах жизненного цикла страницы. Первый имеет место при удалении элемента, к которому подключены обработчики событий. Иногда при этом используются методы `removeChild()` и `replaceChild()`, но чаще всего это происходит при замене фрагмента кода с помощью свойства `innerHTML`. Любые обработчики событий, назначенные элементу, который был удален с помощью свойства `innerHTML`, невозможно правильно удалить в ходе сборки мусора. Рассмотрим пример:

```
<div id="myDiv">
    <input type="button" value="Click Me" id="myBtn">
</div>
<script type="text/javascript">
    let btn = document.getElementById("myBtn");
    btn.onclick = function() {

        // какие-то действия

        document.getElementById("myDiv").innerHTML =
            "Processing...";        // Плохо!!!
    };
</script>
```

Здесь элемент `<div>` содержит кнопку, которая при щелчке заменяется сообщением. Этот прием предотвращает двойной щелчок на кнопке и очень часто используется на веб-сайтах. Проблема в том, что при удалении кнопки с помощью свойства `innerHTML` к ней остается подключенным обработчик события. В некоторых браузерах, особенно в Internet Explorer 8 и более ранних версий, ссылки на элемент и обработчик события обычно остаются в памяти. Таким образом, перед удалением элемента лучше вручную удалить подключенные к нему обработчики событий, например:

```
<div id="myDiv">
    <input type="button" value="Click Me" id="myBtn">
</div>
<script type="text/javascript">
    let btn = document.getElementById("myBtn");
    btn.onclick = function() {

        // какие-то действия

        btn.onclick = null;    // удаление обработчика события

        document.getElementById("myDiv").innerHTML = "Processing...";
    };
</script>
```

Этот код удаляет обработчик события кнопки, прежде чем задать свойство `innerHTML` элемента `<div>`. Это гарантирует, что память будет возвращена среде, и позволяет безопасно удалить кнопку из DOM-структуры.

Имейте в виду, что удаление кнопки в обработчике события отменяет всплытие события. Событие всплывает, только если его целевой элемент все еще находится в документе.

ПРИМЕЧАНИЕ В этой ситуации может помочь делегирование событий. Если вам известно, что какой-то фрагмент страницы будет заменен с помощью свойства `innerHTML`, не подключайте обработчики к элементам в этом фрагменте, а обработайте их события на более высоком уровне.

Проблема обработчиков событий удаленных элементов возникает также при выгрузке страницы. В той или иной степени она присуща всем браузерам, но и здесь выделяются Internet Explorer 8 и более ранних версий. Если не удалить обработчики событий перед выгрузкой страницы, они остаются в памяти, при этом количество объектов в памяти растет с каждым последующим циклом загрузки и выгрузки страницы (в результате перезагрузки или щелчком на кнопках Назад и Вперед).

Перед выгрузкой страницы имеет смысл удалить все обработчики событий в обработчике `onunload`. Делегирование событий полезно и здесь, потому что следить за обработчиками событий проще, если их меньше. Как правило, все, что делается в обработчике `onload`, следует отменять в обработчике `onunload`.

ПРИМЕЧАНИЕ Помните, что если на странице обрабатывается событие `unload`, она не сохраняется в кеше состояния страниц. Если это важно, можно использовать событие `unload` для удаления обработчиков событий только в Internet Explorer.

ИМИТАЦИЯ СОБЫТИЙ

События предназначены для уведомления о важных моментах жизненного цикла веб-страницы и часто генерируются при взаимодействии с пользователем или вызове тех или иных функций браузера. Однако мало кому известно, что в JavaScript можно в любое время инициировать определенные события, которые ничем не отличаются от событий, генерируемых браузером, — они точно так же всплывают и вызывают назначенные им обработчики. Эта возможность очень полезна при тестировании веб-приложений. Способы имитации событий определены в спецификации DOM Level 3. В Internet Explorer 8 и более ранних версий используется фирменный способ имитации событий.

Имитация DOM-событий

Вы можете в любой момент создать объект `event`, вызвав для объекта `document` метод `createEvent()`. Он принимает строку, которая указывает тип создаваемого события. В DOM Level 2 тип события указывался во множественном числе, но в DOM Level 3 используется единственное число. Перечислим возможные типы событий:

- `UIEvents` — универсальное событие пользовательского интерфейса, от которого наследуются события мыши и клавиатуры. В DOM Level 3 используется имя `UIEvent`.

- `MouseEvents` — универсальное событие мыши. В DOM Level 3 используется имя `MouseEvent`.
- `HTMLEvents` — универсальное HTML-событие. Эквивалентного события в спецификации DOM Level 3 нет, потому что в ней HTML-события распределены по другим группам.

События клавиатуры не определены в DOM Level 2 Events, но были представлены в DOM Level 3 Events.

Как только объект `event` создан, его нужно инициализировать сведениями о событии. Для этого используется специальный метод объекта `event`, зависящий от аргумента, который был передан в метод `createEvent()`.

Наконец, для имитации события нужно сгенерировать его. Это делается с помощью метода `dispatchEvent()`, который доступен для всех DOM-узлов, поддерживающих события. В качестве аргумента он принимает объект `event` генерируемого события. Как только этот метод вызван, событие всплывает и запускает обработчики.

Имитация событий мыши

Событие мыши можно имитировать, создав его объект и назначив ему нужные данные. Чтобы создать объект события мыши, требуется вызвать метод `createEvent()` с аргументом `"MouseEvents"`. После этого можно назначить данные события возвращенному объекту с помощью его метода `initMouseEvent()`. Этот метод принимает 15 перечисленных далее аргументов — по одному для каждого свойства, обычно имеющегося у события мыши.

- `type` (строка) — тип генерируемого события, например `"click"`.
- `bubbles` (логическое значение) — указывает, должно ли событие всплывать. Для правильной имитации события мыши этот аргумент должен быть равен `true`.
- `cancelable` (логическое значение) — указывает, можно ли отменить событие. Для правильной имитации события мыши этот аргумент должен быть равен `true`.
- `view` (AbstractView) — представление, связанное с событием. Им почти всегда является объект `document.defaultView`.
- `detail` (целое число) — дополнительные сведения о событии. Этот аргумент используется только обработчиками событий, но обычно имеет значение 0.
- `screenX` (целое число) — координата *x* события относительно экрана.
- `screenY` (целое число) — координата *y* события относительно экрана.
- `clientX` (целое число) — координата *x* события относительно области просмотра.
- `clientY` (целое число) — координата *y* события относительно области просмотра.
- `ctrlKey` (логическое значение) — указывает, нажата ли клавиша Ctrl. По умолчанию `false`.
- `altKey` (логическое значение) — указывает, нажата ли клавиша Alt. По умолчанию `false`.

- `shiftKey` (логическое значение) — указывает, нажата ли клавиша Shift. По умолчанию `false`.
- `metaKey` (логическое значение) — указывает, нажата ли клавиша Meta. По умолчанию `false`.
- `button` (целое число) — кнопка, которая была нажата. По умолчанию 0.
- `relatedTarget` (объект) — объект, связанный с событием. Используется только при имитации событий `mouseover` и `mouseout`.

Как видите, у событий мыши аргументы метода `initMouseEvent()` напрямую соответствуют свойствам объекта `event`. Для правильного генерирования события важны только первые четыре аргумента, потому что их использует браузер; остальные аргументы применяются только в обработчиках событий. Свойство `target` объекта `event` задается автоматически, когда он передается в метод `dispatchEvent()`. Например, следующий код имитирует щелчок на кнопке с аргументами, предлагаемыми по умолчанию:

```
let btn = document.getElementById("myBtn");

// создание объекта event
let event = document.createEvent("MouseEvents");

// инициализация объекта event
event.initMouseEvent("click", true, true, document.defaultView,
                     0, 0, 0, 0, 0, false, false, false, false, 0, null);

// генерирование события
btn.dispatchEvent(event);
```

В браузерах, соответствующих DOM, точно так же можно имитировать и все остальные события мыши, включая `dblclick`.

Имитация событий клавиатуры

События клавиатуры были определены в черновой версии DOM Level 2 Events, но не вошли в окончательную спецификацию, поэтому имитировать их непросто. События, определенные в DOM Level 3, существенно отличаются от них.

В DOM Level 3 для создания события клавиатуры нужно передать строку `"KeyboardEvent"` в метод `createEvent()`. Он возвращает объект `event` с методом `initKeyboardEvent()`, который принимает следующие параметры:

- `type` (строка) — тип генерируемого события, например `"keydown"`.
- `bubbles` (логическое значение) — указывает, должно ли событие всплывать. Для правильной имитации события клавиатуры этот аргумент должен быть равен `true`.
- `cancelable` (логическое значение) — указывает, можно ли отменить событие. Для правильной имитации события клавиатуры этот аргумент должен быть равен `true`.

- `view` (AbstractView) — представление, связанное с событием. Им почти всегда является объект `document.defaultView`.
- `key` (строка) — строковый код нажатой клавиши.
- `location` (целое число) — расположение нажатой клавиши: 0 — клавиатура, предлагаемая по умолчанию; 1 — левая часть клавиатуры; 2 — правая часть клавиатуры; 3 — цифровая клавиатура; 4 — клавиатура мобильного устройства (виртуальная); 5 — джойстик.
- `modifiers` (строка) — список разделенных пробелами клавиш-модификаторов, таких как `"Shift"`.
- `repeat` (целое число) — количество нажатий клавиши подряд.

В DOM Level 3 Events событие `keypress` объявлено устаревшим, поэтому этим способом можно имитировать только события `keydown` и `keyup`, например:

```
let textbox = document.getElementById("myTextbox"),
    event;

// создание объекта event в стиле DOM Level 3
if (document.implementation.hasFeature("KeyboardEvents", "3.0")) {
    event = document.createEvent("KeyboardEvent");

    // инициализация объекта event
    event.initKeyboardEvent("keydown", true, true, document.defaultView,
                            "a", 0, "Shift", 0);
}

// генерирование события
textbox.dispatchEvent(event);
```

Этот пример имитирует нажатие клавиши A при нажатой клавише Shift. Прежде чем вызывать метод `document.createEvent("KeyboardEvent")`, всегда проверяйте, поддерживаются ли события клавиатуры DOM Level 3; если нет, браузер может возвратить нестандартный объект события клавиатуры.

В Firefox можно создать событие клавиатуры, передав строку `"KeyEvents"` в метод `createEvent()`. Он возвращает объект `event` с методом `initKeyEvent()`, который принимает десять аргументов:

- `type` (строка) — тип генерируемого события, например `"keydown"`.
- `bubbles` (логическое значение) — указывает, должно ли событие всплывать. Для правильной имитации события клавиатуры этот аргумент должен быть равен `true`.
- `cancelable` (логическое значение) — указывает, можно ли отменить событие. Для правильной имитации события клавиатуры этот аргумент должен быть равен `true`.
- `view` (AbstractView) — представление, связанное с событием. Им почти всегда является объект `document.defaultView`.

- `ctrlKey` (логическое значение) — указывает, нажата ли клавиша Ctrl. По умолчанию `false`.
- `altKey` (логическое значение) — указывает, нажата ли клавиша Alt. По умолчанию `false`.
- `shiftKey` (логическое значение) — указывает, нажата ли клавиша Shift. По умолчанию `false`.
- `metaKey` (логическое значение) — указывает, нажата ли клавиша Meta. По умолчанию `false`.
- `keyCode` (целое число) — код нажатой или отпущенной клавиши. Используется с событиями `keydown` и `keyup`. По умолчанию 0.
- `charCode` (целое число) — ASCII-код символа, сгенерированного нажатием клавиши. Используется с событием `keypress`. По умолчанию 0.

Чтобы сгенерировать событие клавиатуры, нужно передать настроенный объект `event` в метод `dispatchEvent()`, например:

```
// только для Firefox
let textbox = document.getElementById("myTextbox");

// создание объекта event
let event = document.createEvent("KeyEvents");

// инициализация объекта event
event.initKeyEvent("keydown", true, true, document.defaultView, false,
                   false, true, false, 65, 65);

// генерирование события
textbox.dispatchEvent(event);
```

Этот пример имитирует нажатие клавиши A при нажатой клавише Shift. Так же можно имитировать события `keyup` и `keypress`.

В других браузерах нужно создать универсальное событие и назначить ему данные, специфичные для клавиатуры, например:

```
let textbox = document.getElementById("myTextbox");

// создание объекта event
let event = document.createEvent("Events");

// инициализация объекта event
event.initEvent(type, bubbles, cancelable);
event.view = document.defaultView;
event.altKey = false;
event.ctrlKey = false;
event.shiftKey = false;
event.metaKey = false;
event.keyCode = 65;
event.charCode = 65;

// генерирование события
textbox.dispatchEvent(event);
```

Этот код создает универсальное событие, инициализирует его методом `initEvent()` и назначает ему данные о событии клавиатуры. Универсальное событие требуется вместо события пользовательского интерфейса потому, что последнее не позволяет добавить новые свойства к объекту `event` (это возможно только в Safari). Такой подход имитирует событие клавиатуры не совсем точно: оно генерируется, но никакой текст в текстовое поле не добавляется.

Имитация других событий

Чаще всего в браузерах имитируют события мыши и клавиатуры, но можно также имитировать HTML-события.

Для имитации HTML-события нужно создать объект `event`, вызвав метод `createEvent("HTMLEvents")`, и инициализировать его помощью метода `initEvent()`, например:

```
let event = document.createEvent("HTMLEvents");
event.initEvent("focus", true, false);
target.dispatchEvent(event);
```

Этот код генерирует событие `focus` для объекта `target`. Другие HTML-события можно имитировать так же.

> **ПРИМЕЧАНИЕ** HTML-события редко применяются в браузерах, потому что пользы от них мало.

Пользовательские DOM-события

В DOM Level 3 определены также так называемые *пользовательские события* (custom events). Они изначально не генерируются, а предоставляются для того, чтобы разработчики могли создавать собственные события. Создать пользовательское событие можно, вызвав метод `createEvent("CustomEvent")`. Он возвращает объект `event` с методом `initCustomEvent()`, который принимает четыре аргумента:

- `type` (строка) — тип генерируемого события, например `"keydown"`;
- `bubbles` (логическое значение) — указывает, должно ли событие всплывать;
- `cancelable` (логическое значение) — указывает, можно ли отменить событие;
- `detail` (объект) — любое значение, которое присваивается свойству `detail` объекта `event`.

Затем созданное событие можно сгенерировать в DOM, как и любое другое, например:

```
let div = document.getElementById("myDiv"),
    event;

div.addEventListener("myevent", (event) => {
    console.log("DIV: " + event.detail);
```

```
});
document.addEventListener("myevent", (event) => {
    console.log("DOCUMENT: " + event.detail);
});

if (document.implementation.hasFeature("CustomEvents", "3.0")) {
    event = document.createEvent("CustomEvent");
    event.initCustomEvent("myevent", true, false, "Hello world!");
    div.dispatchEvent(event);
}
```

Этот код создает всплывающее событие `"myevent"` с простой строкой в качестве значения `event.detail`. Событие прослушивается у элемента `<div>` и на уровне документа, к которому оно всплывает благодаря второму аргументу метода `initCustomEvent()`.

Имитация событий в Internet Explorer

Общий алгоритм имитации событий в Internet Explorer 8 и более ранних версий такой же, что и в случае DOM-событий: вы создаете объект `event`, назначаете ему нужные значения и генерируете событие с его помощью. Конечно, в Internet Explorer это делается немного иначе.

Создать объект `event` можно с помощью метода `createEventObject()` объекта `document`. В отличие от DOM-метода, он не принимает аргументов и возвращает универсальный объект `event`, которому затем необходимо вручную назначить все нужные свойства (соответствующего метода нет). После этого нужно вызвать для целевого элемента события метод `fireEvent()`, передав ему имя обработчика события и настроенный объект `event`. При его вызове объекту `event` автоматически назначаются свойства `srcElement` и `type`, а все остальные свойства нужно задать вручную. Этот способ применяется для имитации всех событий, которые поддерживает Internet Explorer. Например, следующий код генерирует событие `click` для кнопки:

```
var btn = document.getElementById("myBtn");

// создание объекта event
var event = document.createEventObject();

// инициализация объекта event
event.screenX = 100;
event.screenY = 0;
event.clientX = 0;
event.clientY = 0;
event.ctrlKey = false;
event.altKey = false;
event.shiftKey = false;
event.button = 0;

// генерирование события
btn.fireEvent("onclick", event);
```

Здесь мы создаем объект `event`, а затем инициализируем его некоторыми значениями. Свойства события задаются произвольным образом, при этом можно задать даже те свойства, которые по умолчанию не поддерживаются в Internet Explorer 8

и более ранних версий. Значения свойств не влияют на событие, потому что они используются только его обработчиками.

По такому же алгоритму можно сгенерировать событие `keypress`:

```
var textbox = document.getElementById("myTextbox");

// создание объекта event
var event = document.createEventObject();

// инициализация объекта event
event.altKey = false;
event.ctrlKey = false;
event.shiftKey = false;
event.keyCode = 65;

// генерирование события
textbox.fireEvent("onkeypress", event);
```

Поскольку объекты `event` событий мыши, клавиатуры и других событий не различаются, для генерирования любых событий можно использовать универсальный объект `event`. Как и при имитации DOM-события клавиатуры, в этом примере никакие знаки в текстовом поле не появляются, хотя обработчик события выполняется нормально.

ИТОГИ

События — это основной механизм подключения JS-кода к веб-страницам. События, используемые чаще всего, определены в спецификациях DOM Level 3 Events и HTML5, но во многих браузерах доступны также дополнительные фирменные события, позволяющие лучше контролировать взаимодействие с пользователями. Некоторые фирменные события предназначены для работы с конкретными устройствами.

При использовании событий следует помнить о том, что они потребляют ресурсы, и соблюдать перечисленные здесь принципы.

- Обработчиков событий на странице не должно быть слишком много, потому что они занимают память и замедляют отклик страницы на действия пользователя.
- Для уменьшения количества обработчиков на странице можно использовать механизм делегирования событий, в основе которого лежит всплытие событий.
- Рекомендуется удалять обработчики событий перед выгрузкой страницы.

С помощью JavaScript можно имитировать события в браузере. Спецификации DOM Level 2 и 3 позволяют с легкостью имитировать все определенные в них события. Имитировать события клавиатуры сложнее, но тоже возможно. В Internet Explorer 8 и более ранних версий используется фирменный способ имитации событий.

Обработка событий — одна из важнейших составляющих JavaScript-программирования. Каждый серьезный веб-программист должен понимать, как они работают и как влияют на быстродействие кода.

18

Анимация и рисование на холсте

- Использование requestAnimationFrame
- Элемент <canvas>
- Рисование простой 2D-графики
- 3D-рисование с помощью WebGL

Графика и анимация в браузере становятся все более важными компонентами современного интернета, но их также чрезвычайно сложно сделать хорошо. Визуально сложные функции требуют настройки улучшения производительности и аппаратного ускорения, чтобы они не замедляли работу браузера. Все более надежный набор API и инструментов позволяет разрабатывать такие функции.

Возможно, наиболее популярным дополнением HTML5 является элемент `<canvas>`. Этот элемент обозначает область страницы, где графика может быть создана на лету с использованием JavaScript. Первоначально предложенный Apple для использования со своими виджетами Dashboard, `<canvas>` быстро был добавлен в HTML5 и снискал очень высокую популярность в браузерах. Все основные браузеры в некоторой степени поддерживают `<canvas>`.

Подобно другим частям среды браузера, `<canvas>` состоит из нескольких наборов API, при этом не все браузеры поддерживают все наборы API. Существует 2D-контекст с базовыми возможностями рисования и предлагаемым 3D-контекстом WebGL. Последние версии поддерживаемых браузеров теперь поддерживают как 2D-контекст, так и WebGL.

ИСПОЛЬЗОВАНИЕ REQUESTANIMATIONFRAME

В течение долгого времени таймеры и интервалы считались современными для анимации на основе JavaScript. В то время как CSS-переходы и анимация облегчают веб-разработчикам некоторые анимации, в мире анимации на основе JavaScript с годами мало что изменилось. Firefox 4 был первым браузером, который добавил новый API для анимации JavaScript под именем `mozRequestAnimationFrame()`. Этот метод указывает браузеру, что происходит анимация, так что браузер, в свою очередь, может определить лучший способ запланировать перерисовку. С момента своего появления API получил широкое распространение и теперь доступен во всех основных браузерах как `requestAnimationFrame()`.

Ранние анимационные циклы

Типичный способ создания анимации в JavaScript — это использование `setInterval()` для управления всеми анимациями. Базовый цикл анимации с использованием `setInterval()` выглядит следующим образом:

```
(function() {
    function updateAnimations() {
        doAnimation1();
        doAnimation2();
        // и т. д.
    }
    setInterval(updateAnimations, 100);
})();
```

Чтобы создать небольшую библиотеку анимаций, метод `updateAnimations()` должен циклически проходить по запущенным анимациям и вносить соответствующие изменения в каждую (например, считыватель новостей и индикатор выполнения, работающие вместе). Если нет анимаций для обновления, метод может завершить работу, ничего не делая, и, возможно, даже остановить цикл анимации, пока больше анимаций не будет готово к обновлению.

Самое сложное в этом цикле анимации — знать, какой должна быть задержка. Интервал должен быть достаточно коротким, чтобы обрабатывать различные типы анимации плавно, но достаточно долгим, чтобы произвести изменения, которые браузер может реально отобразить. Большинство компьютерных мониторов обновляются с частотой 60 Гц, что в основном означает, что перерисовка выполняется 60 раз в секунду. Большинство браузеров переписывают свои перерисовки, поэтому они не пытаются перерисовываться чаще, зная, что для конечного пользователя ничего не изменится.

Следовательно, лучший интервал для плавной анимации составляет 1000 мс/60, или около 17 мс. Вы увидите самую плавную анимацию с такой скоростью, потому что так более точно отражаются возможности браузера. Может потребоваться регулировка нескольких анимаций, чтобы они не выполнялись слишком быстро при использовании цикла анимации с интервалом 17 мс.

Несмотря на то, что анимационные циклы на основе `setInterval()` более эффективны, чем использование нескольких наборов циклов на основе `setTimeout()`, с ними все еще существуют проблемы. Ни `setInterval()`, ни `setTimeout()` не могут быть точными. Задержка, указанная в качестве второго аргумента, является лишь указанием того, когда код добавляется в очередь потоков пользовательского интерфейса браузера для возможного выполнения. Если в очереди есть другие задания, то этот код ожидает выполнения. Короче говоря, миллисекундная задержка не является показателем того, когда код будет выполнен, а лишь указанием того, когда задание будет поставлено в очередь. Если поток пользовательского интерфейса занят, возможно, имея дело с действиями пользователя, то этот код не будет выполнен немедленно.

Проблемы с интервалами

Понимание того, когда будет отрисован следующий кадр, является ключом к плавной анимации, и до недавнего времени не было способа гарантировать, что следующий кадр будет нарисован в браузере. По мере того как <canvas> становился популярным и появлялись новые браузерные игры, разработчики все больше расстраивались из-за неточности `setInterval()` и `setTimeout()`.

Усугубляет эти проблемы разрешение таймера браузера. Однако таймеры не имеют точности до миллисекунды. Вот некоторые общие разрешения таймера:

- Internet Explorer 8 и более ранние версии имеют разрешение таймера 15,625 мс.
- Internet Explorer 9 и более поздние версии имеют разрешение таймера 4 мс.
- Firefox и Safari имеют разрешение таймера ~ 10 мс.
- Chrome имеет разрешение таймера 4 мс.

Internet Explorer до версии 9 имеет разрешение таймера 15,625 мс, поэтому любое значение от 0 до 15 может быть 0 или 15, но не более того. Internet Explorer 9 улучшил разрешение таймера до 4 мс, но это все же не очень конкретно, когда дело доходит до анимации. Разрешение таймера Chrome составляет 4 мс, а Firefox и Safari — 10 мс. Еще более усложняет ситуацию то, что браузеры начали регулировать таймеры для вкладок, которые находятся в фоновом режиме или неактивны. Поэтому, даже если вы установите интервал для оптимального отображения, то все равно лишь приблизитесь к желаемому времени.

requestAnimationFrame

Роберт О'Каллахан из Mozilla задумался над этой проблемой и нашел уникальное решение. Он отметил, что CSS-переходы и анимации выигрывают от того, что браузер знает, что должна происходить некоторая анимация, и поэтому определил правильный интервал обновления интерфейса.

При JavaScript-анимации браузер не знает, что анимация выполняется. Его решением было создание нового метода, называемого `mozRequestAnimationFrame()`, который указывает браузеру, что некоторый код JavaScript выполняет анимацию.

Это позволяет браузеру соответствующим образом оптимизироваться после выполнения некоторого кода. Все браузеры сходятся к версии этого метода без префикса `requestAnimationFrame()`.

Метод `requestAnimationFrame()` принимает один аргумент, который является функцией, вызываемой перед перерисовкой экрана. Эта функция предназначена для внесения соответствующих изменений в стили DOM, которые будут отражены при следующей перерисовке. Для создания цикла анимации можно объединить несколько вызовов `requestAnimationFrame()`, как это было ранее сделано с помощью `setTimeout()`. Например:

```
function updateProgress() {
    var div = document.getElementById("status");
    div.style.width = (parseInt(div.style.width, 10) + 5) + "%";
    if (div.style.left != "100%") {
        requestAnimationFrame(updateProgress);
    }
}
requestAnimationFrame(updateProgress);
```

Поскольку `requestAnimationFrame()` запускает данную функцию только один раз, нужно вызвать ее снова вручную в следующий раз, когда вы захотите изменить пользовательский интерфейс для анимации. Также необходимо решить, когда анимация будет таким же образом остановлена. В результате получается очень плавная анимация.

До сих пор `requestAnimationFrame()` решала проблему, когда браузеры не знали, когда происходит анимация JavaScript, и проблему незнания наилучшего интервала, но как насчет проблемы незнания, когда именно код будет выполняться? Эта проблема покрывается тем же решением.

Функция, передаваемая в `requestAnimationFrame()`, фактически получает аргумент, который является кодом времени DOMHighResTimeStamp (например, значение, возвращаемое из `performance.now()`) для случая, когда на самом деле произойдет следующее перерисовывание. Это очень важный момент: `requestAnimationFrame()` фактически планирует перерисовку для некоторого известного момента в будущем и может сообщить, когда это произойдет. После этого можно будет определить, как лучше настроить анимацию.

cancelAnimationFrame

Подобно `setTimeout()`, `requestAnimationFrame()` возвращает идентификатор запроса, который можно использовать для отмены запроса с помощью `cancelAnimationFrame()`. В следующем примере обратный вызов запроса ставится в очередь, но сразу отменяется:

```
let requestID = window.requestAnimationFrame(() => {
    console.log('Repaint!');
});
window.cancelAnimationFrame(requestID);
```

Управление производительностью с помощью requestAnimationFrame

Имя `requestAnimationFrame` несколько вводит в заблуждение относительно выполняемой задачи. Браузеры, поддерживающие этот метод, эффективно выставляют очередь перехвата вызовов. Перехват — это временная точка перед тем как браузер выполнит следующую перерисовку. Очередь обратных вызовов представляет собой изменяемый список функций, которые следует вызывать перед выполнением перерисовки. Вызов `requestAnimationFrame()` помещает функцию обратного вызова в эту неограниченную по длине очередь.

Поведение обратного вызова в очереди не должно включать анимацию. Однако рекурсивная постановка в очередь функций обратного вызова с помощью `requestAnimationFrame()` гарантирует, что обратный вызов будет вызываться не более одного раза за перерисовку, что является отличным инструментом ограничения скорости. Это особенно полезно при работе с часто вызываемым кодом, который влияет на внешний вид страницы, например обработчиками события прокрутки.

Рассмотрим следующую простую реализацию, которая вызовет псевдодорогостоящую операцию при запуске события прокрутки из объекта окна. При прокрутке веб-страницы это событие может быть запущено сотни или тысячи раз с очень большой скоростью:

```
function expensiveOperation() {
    console.log('Invoked at', Date.now());
}
window.addEventListener('scroll', () => {
    expensiveOperation();
});
```

Если нужно ограничить обратный вызов только перед перерисовкой, вы можете обернуть его внутри `requestAnimationFrame`:

```
function expensiveOperation() {
    console.log('Invoked at', Date.now());
}
window.addEventListener('scroll', () => {
    window.requestAnimationFrame(expensiveOperation);
});
```

Это сведет все выполнения обратного вызова в ловушку перерисовки, но не предотвратит избыточное выполнение перерисовки. Можно предотвратить избыточное выполнение перерисовки, введя флаг, который устанавливается и снимается обратным вызовом:

```
let enqueued = false;

function expensiveOperation() {
    console.log('Invoked at', Date.now());
    enqueued = false;
```

```
}
window.addEventListener('scroll', () => {
    if (!enqueued) {
        enqueued = true;
        window.requestAnimationFrame(expensiveOperation);
    }
});
```

Поскольку перерисовка — очень частая операция, это не так уж и сложно. Гораздо лучше объединить этот процесс с таймером, который будет регулировать частоту, с которой может происходить операция. Таким образом, таймер может ограничить реальный временной интервал, в котором происходит операция, и `requestAnimationFrame` будет контролировать, когда в цикле отрисовки браузера происходит выполнение. В следующем примере будет запрещено выполнять обратный вызов более одного раза каждые 50 мс:

```
let enabled = true;

function expensiveOperation() {
    console.log('Invoked at', Date.now());
}

window.addEventListener('scroll', () => {
    if (enabled) {
        enabled = false;
        window.requestAnimationFrame(expensiveOperation);
        window.setTimeout(() => enabled = true, 50);
    }
});
```

ОСНОВЫ РАБОТЫ С ЭЛЕМЕНТОМ <CANVAS>

Для использования элемента `<canvas>` нужно как минимум задать его атрибуты `width` и `height`, указывающие ширину и высоту рисунка. Контент между его открывающим и закрывающим тегами добавляется для страховки и выводится на экран, только если элемент `<canvas>` не поддерживается, например:

```
<canvas id="drawing" width="200" height="200">A drawing of something.</canvas>
```

Как и у других элементов, атрибуты `width` и `height` доступны в качестве свойств соответствующего объекта, которые в любой момент можно изменить. Весь элемент можно стилизовать средствами CSS, а пока это не сделано и на холсте ничего не нарисовано, он остается невидимым.

Чтобы начать рисовать на холсте, нужно получить контекст рисования методом `getContext()`, который принимает имя контекста. Например, передав ему значение `"2d"`, можно получить объект двухмерного контекста:

```
let drawing = document.getElementById("drawing");

// проверка полной поддержки элемента <canvas>
```

```
if (drawing.getContext){

    let context = drawing.getContext("2d");

    // другой код
}
```

Перед использованием элемента `<canvas>` важно проверить наличие метода `getContext()`. Для элементов, которые не входят официально в HTML, некоторые браузеры создают объекты, предлагаемые по умолчанию, и тогда метод `getContext()` становится недоступен, хотя объект `drawing` содержит действительную ссылку на элемент.

Изображения, созданные в элементе `<canvas>`, можно экспортировать методом `toDataURL()`. Он принимает целевой формат изображения в виде MIME-типа и работает независимо от того, какой контекст был использован для создания изображения. Например, чтобы возвратить изображение с холста в формате PNG, используйте следующий код:

```
let drawing = document.getElementById("drawing");

// проверка полной поддержки элемента <canvas>
if (drawing.getContext){

    // получение URI изображения
    let imgURI = drawing.toDataURL("image/png");

    // вывод изображения на экран
    let image = document.createElement("img");
    image.src = imgURI;
    document.body.appendChild(image);
}
```

По умолчанию браузеры кодируют изображения в формате PNG. В Firefox и Opera можно также использовать формат JPEG (тип `"image/jpeg"`). Он доступен в более поздних версиях браузеров, включая Internet Explorer 9, Firefox 3.5 и Opera 10.

> **ПРИМЕЧАНИЕ** Если на холсте нарисовано изображение из другого домена, метод `toDataURL()` генерирует ошибку. Подробности мы обсудим немного позже.

ДВУХМЕРНЫЙ КОНТЕКСТ

Двухмерный контекст рисования предоставляет методы для рисования простых двухмерных фигур, таких как прямоугольники, дуги и пути. Началом координат в двухмерном контексте является верхний левый угол элемента `<canvas>`, который считается точкой (0,0). Значение x увеличивается слева направо, а y — сверху вниз. По умолчанию свойства `width` и `height` указывают, сколько пикселей доступно в каждом направлении.

Заливка и рисование контура

Для двухмерного контекста доступны две основные операции рисования: заливка и рисование контура. При заливке фигура автоматически заполняется указанным содержимым с указанным стилем (цвет, градиент или изображение), а при рисовании контура окрашиваются только контуры фигуры. У большинства операций в двухмерном контексте есть варианты с заливкой и рисованием контура, которые настраиваются с помощью свойств `fillStyle` и `strokeStyle`.

Каждое из этих свойств может быть строкой, градиентом или узором и имеет по умолчанию значение `"#000000"`. Если используется строка, она должна определять цвет в одном из форматов CSS: по имени, шестнадцатеричному коду, в формате `rgb`, `rgba`, `hsl` или `hsla`. Вот пример:

```
let drawing = document.getElementById("drawing");

    // проверка полной поддержки элемента <canvas>
    if (drawing.getContext){

        let context = drawing.getContext("2d");
        context.strokeStyle = "red";
        context.fillStyle = "#0000ff";
}
```

Этот код присваивает свойству `strokeStyle` значение `"red"` (именованный CSS-цвет), а свойству `fillStyle` — `"#0000ff"` (синий цвет). Все последующие операции заливки и рисования контура будут использовать эти значения, пока они не изменятся. Этим свойствам также можно назначить градиент или узор, о чем мы поговорим позже.

Рисование прямоугольников

Прямоугольник — единственная фигура, которую можно нарисовать непосредственно в двухмерном контексте. Для работы с прямоугольниками можно использовать методы `fillRect()`, `strokeRect()` и `clearRect()`. Каждый из них принимает четыре аргумента: координаты x и y, ширину и высоту прямоугольника. Значения аргументов измеряются в пикселях.

Метод `fillRect()` рисует на холсте прямоугольник, залитый цветом, указанным с помощью свойства `fillStyle`, например:

```
let drawing = document.getElementById("drawing");

// проверка полной поддержки элемента <canvas>
if (drawing.getContext){

    let context = drawing.getContext("2d");

    /*
     * Код основан на документации Mozilla:
     * http://developer.mozilla.org/en/docs/Canvas_tutorial:Basic_usage
     */

    // рисование красного квадрата
```

```
    context.fillStyle = "#ff0000";
    context.fillRect(10, 10, 50, 50);

    // рисование синего полупрозрачного квадрата
    context.fillStyle = "rgba(0,0,255,0.5)";
    context.fillRect(30, 30, 50, 50);
}
```

Этот код сначала назначает свойству `fillStyle` красный цвет и рисует квадрат с левым верхним углом в точке (10,10) и стороной, равной 50 пикселей. Затем он назначает свойству `fillStyle` синий полупрозрачный цвет методом `rgba()` и рисует второй квадрат, который перекрывает первый. В результате красный квадрат виден через синий (рис. 18.1).

Рис. 18.1

Метод `strokeRect()` рисует контур прямоугольника, используя цвет, указанный с помощью свойства `strokeStyle`, например:

```
let drawing = document.getElementById("drawing");

// проверка полной поддержки элемента <canvas>
if (drawing.getContext){

    let context = drawing.getContext("2d");

    /*
     * Код основан на документации Mozilla:
     * http://developer.mozilla.org/en/docs/Canvas_tutorial:Basic_usage
     */

    // рисование красного контура квадрата
    context.strokeStyle = "#ff0000";
    context.strokeRect(10, 10, 50, 50);

    // рисование синего полупрозрачного контура квадрата
    context.strokeStyle = "rgba(0,0,255,0.5)";
    context.strokeRect(30, 30, 50, 50);
}
```

Этот код рисует только контуры перекрывающихся квадратов (рис. 18.2).

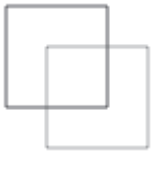

Рис. 18.2

ПРИМЕЧАНИЕ Толщина контура определяется свойством `lineWidth`, которому можно присвоить любое целое число. Свойство `lineCap` описывает концы линий; доступные значения: `"butt"` (срез), `"round"` (круг), `"square"` (квадрат), а `lineJoin` указывает, как должны соединяться линии; доступные значения: `"round"` (закругление), `"bevel"` (скос), `"miter"` (клин).

Метод `clearRect()` очищает область холста и используется, если нужно сделать прямоугольную часть контекста рисования прозрачной. Рисуя фигуры и очищая

их области, можно создавать интересные эффекты, например вырезать фрагменты других фигур:

```
let drawing = document.getElementById("drawing");

// проверка полной поддержки элемента <canvas>
if (drawing.getContext){

    let context = drawing.getContext("2d");

    /*
     * Код основан на документации Mozilla:
     * http://developer.mozilla.org/en/docs/Canvas_tutorial:Basic_usage
     */

    // рисование красного квадрата
    context.fillStyle = "#ff0000";
    context.fillRect(10, 10, 50, 50);

    // рисование синего полупрозрачного квадрата
    context.fillStyle = "rgba(0,0,255,0.5)";
    context.fillRect(30, 30, 50, 50);

    // очистка квадрата в области наложения двух квадратов
    context.clearRect(40, 40, 10, 10);
}
```

Рис. 18.3

Этот код сначала рисует два перекрывающихся квадрата с заливкой, а затем очищает в области их наложения меньший квадрат (рис. 18.3).

Рисование путей

Двухмерный контекст рисования поддерживает несколько методов рисования путей, позволяющих создавать сложные фигуры и линии. Чтобы приступить к созданию пути, нужно вызвать метод `beginPath()`, после чего для продолжения пути можно использовать следующие методы:

1. `arc(`*x*, *y*, *радиус*, *начальныйУгол*, *конечныйУгол*, *противЧасовойСтрелки*`)` — рисует дугу указанного радиуса с центром в точке (*x*, *y*) между начальным и конечным углами (в радианах). Последним аргументом является логическое значение, указывающее, следует отсчитывать углы против часовой стрелки или по ней.
2. `arcTo(`*x1*, *y1*, *x2*, *y2*, *радиус*`)` — рисует дугу указанного радиуса от последней точки до (*x2*, *y2*), проходящую через (*x1*, *y1*).
3. `bezierCurveTo(`*c1x*, *c1y*, *c2x*, *c2y*, *x*, *y*`)` — рисует кривую от последней точки до точки (*x*, *y*), используя контрольные точки (*c1x*, *c1y*) и (*c2x*, *c2y*).
4. `lineTo(`*x*, *y*`)` — рисует линию от последней точки до точки (*x*, *y*).
5. `moveTo(`*x*, *y*`)` — перемещает курсор в точку (*x*, *y*) без рисования линии.

6. `quadraticCurveTo(`*cx*, *cy*, *x*, *y*`)` — рисует квадратичную кривую от последней точки до точки (*x*, *y*), используя контрольную точку (*cx*, *cy*).
7. `rect(`*x*, *y*, *ширина*, *высота*`)` — рисует прямоугольник с левым верхним углом в точке (*x*, *y*) и указанными значениями ширины и высоты. Этот метод отличается от методов `strokeRect()` и `fillRect()` тем, что создает путь, а не отдельную фигуру.

Как только путь создан, возможны несколько вариантов. Можно вызвать метод `closePath()`, чтобы провести линию к началу пути. Если путь уже завершен и вы хотите залить его, используя стиль `fillStyle`, вызовите метод `fill()`. Можно также отобразить путь без заливки, вызвав метод `stroke()`, при этом будет задействован стиль `strokeStyle`. Последний вариант — создать на основе пути область отсечения с помощью метода `clip()`.

Рассмотрим, например, следующий код, который рисует часы без чисел:

```
let drawing = document.getElementById("drawing");

// проверка полной поддержки элемента <canvas>
if (drawing.getContext){

    let context = drawing.getContext("2d");

    // начало пути
    context.beginPath();

    // рисование внешней окружности
    context.arc(100, 100, 99, 0, 2 * Math.PI, false);

    // рисование внутренней окружности
    context.moveTo(194, 100);
    context.arc(100, 100, 94, 0, 2 * Math.PI, false);

    // рисование минутной стрелки
    context.moveTo(100, 100);
    context.lineTo(100, 15);

    // рисование часовой стрелки
    context.moveTo(100, 100);
    context.lineTo(35, 100);

    // вывод на экран пути без заливки
    context.stroke();
}
```

Чтобы создать границу часов, мы рисуем две концентрические окружности методом `arc()`. Внешняя окружность имеет радиус 99 пикселей и центр в точке (100,100), которая совпадает с центром холста. Чтобы нарисовать полную окружность, мы проводим дугу от 0 до 2π радиан, используя в качестве числа π значение `Math.PI`. Перед рисованием внутренней окружности нужно переместить путь в точку, которая будет находиться на ней, чтобы предотвратить рисование лишней линии. Во втором вызове `arc()` для создания эффекта границы

используется немного меньший радиус. После этого методами `moveTo()` и `lineTo()` мы рисуем минутную и часовую стрелки. Наконец, метод `stroke()` придает часам вид, показанный на рис. 18.4.

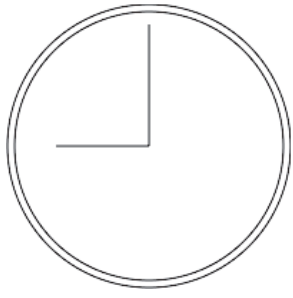

Рис. 18.4

Пути — основной механизм рисования в двухмерном контексте, потому что они обеспечивают более точный контроль над фигурами. Чтобы упростить работу с ними, можно использовать метод `isPointInPath()`, который принимает координаты точки по осям *x* и *y*. С помощью этого метода можно узнать, содержит ли путь конкретную точку:

```
if (context.isPointInPath(100, 100)){
    alert("Point (100, 100) is in the path.");
}
```

API путей достаточно надежен и позволяет создавать весьма сложные изображения с разнообразными заливками, контурами и т. д.

Рисование текста

Поскольку рисунки часто должны содержать не только графику, но и текст, двухмерный контекст рисования предоставляет методы `fillText()` и `strokeText()` для рисования текста. Каждый из них принимает четыре аргумента: строку, которую нужно нарисовать, координаты *x* и *y* и необязательное значение максимальной ширины текста в пикселях. Оба метода рисуют текст, используя значения трех свойств:

1. `font` — начертание, размер и семейство шрифта в формате CSS, например `"10px Arial"`.
2. `textAlign` — способ выравнивания текста. Возможные значения: `"start"`, `"end"`, `"left"`, `"right"` и `"center"`. Рекомендуется использовать значения `"start"` и `"end"` вместо `"left"` и `"right"`, потому что они правильно отражают суть дела в языках с написанием как слева направо, так и справа налево.
3. `textBaseline` — базовая линия текста. Возможные значения: `"top"`, `"hanging"`, `"middle"`, `"alphabetic"`, `"ideographic"` и `"bottom"`.

У этих свойств есть значения, предлагаемые по умолчанию, так что задавать их каждый раз при рисовании текста не требуется. Метод `fillText()` использует при рисовании текста свойство `fillStyle`, а метод `strokeText()` — свойство `strokeStyle`. Вероятно, в большинстве случаев вы будете использовать метод `fillText()`, так как он имитирует обычное отображение текста на веб-страницах. Например, следующий код выводит число 12 в верхней части часов, созданных в предыдущем разделе:

```
context.font = "bold 14px Arial";
context.textAlign = "center";
context.textBaseline = "middle";
context.fillText("12", 100, 20);
```

Итоговое изображение показано на рис. 18.5.

Поскольку свойство `textAlign` имеет значение `"center"`, а `textBaseline` — `"middle"`, координаты (100, 20) указывают центр текста по горизонтали и его среднюю линию по вертикали. Если бы свойство `textAlign` имело значение `"start"` или `"end"`, координата *x* представляла бы в языке с письмом слева направо начало или конец текста соответственно:

```
// выравнивание по центру
context.font = "bold 14px Arial";
context.textAlign = "center";
context.textBaseline = "middle";
context.fillText("12", 100, 20);

// выравнивание по началу текста
context.textAlign = "start";
context.fillText("12", 100, 40);

// выравнивание по концу текста
context.textAlign = "end";
context.fillText("12", 100, 60);
```

Этот код трижды отображает строку `"12"` с одним и тем же значением *x*, но с тремя разными значениями координаты *y* и свойства `textAlign`. Итоговое изображение показано на рис. 18.6.

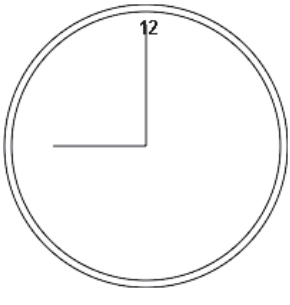

Рис. 18.5

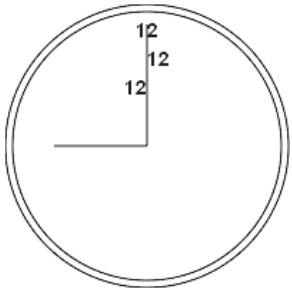

Рис. 18.6

Минутная стрелка часов располагается вертикально, так что выравнивание текста очевидно. Настроить выравнивание текста по вертикали можно с помощью свойства `textBaseline`. Если оно имеет значение `"top"` или `"bottom"`, координата *y* определяет соответственно верх или низ текста, а значения `"hanging"`, `"alphabetic"` и `"ideographic"` указывают специфические координаты базовой линии шрифта.

Чтобы рисовать текст было проще, особенно если требуется отобразить его в определенной области, можно определить его размеры методом `measureText()`. Он принимает текст, который нужно нарисовать, и возвращает объект `TextMetrics`. В настоящее время этот объект имеет единственное свойство `width`, но предполагается, что к нему будут добавлены и другие параметры.

Для вычисления размеров текста метод `measureText()` использует текущие значения `font`, `textAlign` и `textBaseline`. Предположим, например, что нам нужно вместить текст `"Hellow world!"` в прямоугольник шириной 140 пикселей. Для этого следующий код задает первоначальный размер шрифта (100 пикселей) и уменьшает его, пока текст не поместится в прямоугольник:

```
let fontSize = 100;
context.font = fontSize + "px Arial";

while(context.measureText("Hello world!").width > 140){
    fontSize--;
    context.font = fontSize + "px Arial";
}

context.fillText("Hello world!", 10, 10);
context.fillText("Font size is " + fontSize + "px", 10, 50);
```

У методов `fillText()` и `strokeText()` есть также четвертый аргумент — максимальная ширина текста. Он не обязателен и пока поддерживается не во всех браузерах (впервые он появился в Firefox 4). Если этот аргумент указан и строка, переданная в метод `fillText()` или `strokeText()`, превышает максимальную ширину, текст сжимается по горизонтали, при этом его высота не меняется (рис. 18.7).

Рис. 18.7

Из-за сложности рисования текста соответствующий API пока доступен в браузерах не полностью.

Преобразования

Путем преобразования контекста можно манипулировать изображениями на холсте. Двухмерный контекст рисования поддерживает все базовые преобразования. При создании контекста рисования матрица преобразования инициализируется значениями, предлагаемыми по умолчанию, при которых все операции рисования выполняются без изменений. Если применить преобразование к контексту рисования, операции будут выполняться с другой матрицей, а потому и их результаты окажутся другими.

Матрицу преобразования можно настроить с помощью следующих методов:

- `rotate(угол)` — поворачивает изображение вокруг начала координат на указанный угол в радианах.

- `scale(масштабX, масштабY)` — масштабирует изображение по осям x и y. По умолчанию оба аргумента равны 1.0.
- `translate(x, y)` — перемещает начало координат в точку (x, y).
- `transform(m1_1, m1_2, m2_1, m2_2, dx, dy)` — умножает матрицу преобразования на следующую матрицу:

```
m1_1 m1_2 dx
m2_1 m2_2 dy
0    0    1
```

- `setTransform(m1_1, m1_2, m2_1, m2_2, dx, dy)` — сбрасывает матрицу преобразования, а затем вызывает метод.

Преобразования могут быть сколь угодно простыми или сложными. Скажем, в примере с часами для рисования стрелок можно переместить начало координат в центр часов:

```
let drawing = document.getElementById("drawing");

// проверка полной поддержки элемента <canvas>
if (drawing.getContext){

    let context = drawing.getContext("2d");

    // начало пути
    context.beginPath();

    // рисование внешней окружности
    context.arc(100, 100, 99, 0, 2 * Math.PI, false);

    // рисование внутренней окружности
    context.moveTo(194, 100);
    context.arc(100, 100, 94, 0, 2 * Math.PI, false);

    // перенос начала координат в центр часов
    context.translate(100, 100);

    // рисование минутной стрелки
    context.moveTo(0, 0);
    context.lineTo(0, -85);

    // рисование часовой стрелки
    context.moveTo(0, 0);
    context.lineTo(-65, 0);

    // отображение пути без заливки
    context.stroke();
}
```

После переноса начала координат в центр часов с координатами (100, 100) нарисовать стрелки стало еще проще, потому что все координаты теперь рассчитываются

относительно точки (0,0). Можно не ограничиваться этим и повернуть стрелки методом `rotate()`:

```
let drawing = document.getElementById("drawing");

// проверка полной поддержки элемента <canvas>
if (drawing.getContext){

    let context = drawing.getContext("2d");

    // начало пути
    context.beginPath();

    // рисование внешней окружности
    context.arc(100, 100, 99, 0, 2 * Math.PI, false);

    // рисование внутренней окружности
    context.moveTo(194, 100);
    context.arc(100, 100, 94, 0, 2 * Math.PI, false);

    // перенос начала координат в центр часов
    context.translate(100, 100);

    // поворот стрелок
    context.rotate(1);

    // рисование минутной стрелки
    context.moveTo(0, 0);
    context.lineTo(0, -85);

    // рисование часовой стрелки
    context.moveTo(0, 0);
    context.lineTo(-65, 0);

    // отображение пути без заливки
    context.stroke();
}
```

Поскольку начало координат к моменту поворота уже перенесено в центр часов, эта точка остается на месте, а изображение поворачивается вокруг нее по часовой стрелке (рис. 18.8).

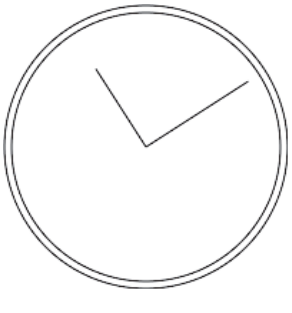

Рис. 18.8

Все эти преобразования и свойства, такие как `fillStyle` и `strokeStyle`, остаются в силе, пока их не изменить явно. Вручную восстановить их значения, предлагаемые по умолчанию, нельзя, но с помощью двух методов можно отслеживать их изменения. Если вы думаете, что позднее вам придется вернуться к текущей конфигурации свойств и преобразований, вызовите метод `save()`, чтобы сохранить все текущие параметры в стеке. После этого можно продолжить изменять контекст, а когда решите вернуться к сохраненным параметрам, просто вызовите метод `restore()`, чтобы извлечь их из стека. Сохранять

и восстанавливать конфигурации с помощью этих методов можно многократно в произвольном порядке, например:

```
context.fillStyle = "#ff0000";
context.save();

context.fillStyle = "#00ff00";
context.translate(100, 100);
context.save();

context.fillStyle = "#0000ff";
// рисование синего прямоугольника
// с левым верхним углом в точке (100, 100)
context.fillRect(0, 0, 100, 200);

context.restore();
// рисование зеленого прямоугольника
// с левым верхним углом в точке (110, 110)
context.fillRect(10, 10, 100, 200);

context.restore();
// рисование красного прямоугольника
// с левым верхним углом в точке (0, 0)
context.fillRect(0, 0, 100, 200);
```

Здесь мы сначала назначаем свойству `fillStyle` красный цвет и вызываем метод `save()`, после чего изменяем значение `fillStyle` на зеленый цвет, переносим начало координат в точку (100,100) и еще раз вызываем метод `save()` для сохранения параметров. Затем мы назначаем свойству `fillStyle` синий цвет и рисуем прямоугольник, левый верхний угол которого из-за переноса начала координат отображается в точке (100,100). Первый вызов метода `restore()` восстанавливает для свойства `fillStyle` зеленый цвет, так что следующий прямоугольник рисуется зеленым цветом, но его левый верхний угол находится в точке (110,110), потому что перенос все еще действует. Второй вызов `restore()` отменяет перенос и восстанавливает первоначальное значение свойства `fillStyle`, поэтому последний прямоугольник рисуется красным цветом с левым верхним углом в точке (0,0).

Имейте в виду, что метод `save()` сохраняет только параметры и преобразования, примененные к контексту рисования, но не его содержимое.

Рисование изображений

Двухмерный контекст рисования содержит встроенные средства для работы с изображениями. Готовое изображение рисуется на холсте методом `drawImage()`, который можно вызывать с тремя разными наборами аргументов в зависимости от нужного результата. Если передать в метод HTML-элемент `<img>` и координаты, он просто выведет изображение в указанном месте, например:

```
let image = document.images[0];
context.drawImage(image, 10, 10);
```

Этот код получает первое изображение в документе и выводит его в контексте рисования в позиции (10, 10) с сохранением масштаба. В метод можно также передать ширину и высоту итогового изображения, чтобы масштабировать его без изменения матрицы преобразования контекста, например:

```
context.drawImage(image, 50, 10, 20, 30);
```

Этот код изменяет ширину и высоту изображения на 20 и 30 пикселей соответственно.

Кроме того, можно вывести в контексте только часть изображения. Для этого нужно передать в метод `drawImage()` девять аргументов: исходное изображение, его координаты x и y, ширину и высоту, а также координаты x и y, ширину и высоту целевого изображения. Эта перегруженная версия метода `drawImage()` обеспечивает наибольший контроль над рисованием, например:

```
context.drawImage(image, 0, 10, 50, 50, 0, 100, 40, 60);
```

Этот метод выводит на холсте только часть исходного изображения, которая занимает 50 пикселей в ширину и высоту и имеет левый верхний угол в точке (0, 10). Итоговое изображение выводится в области с размерами 40 × 60 пикселей и левым верхним углом в точке (0, 100).

Эти операции рисования позволяют создавать интересные эффекты вроде тех, что показаны на рис. 18.9.

Рис. 18.9

Кроме HTML-элемента `<img>` в метод `drawImage()` в качестве первого аргумента можно передать другой элемент `<canvas>`, чтобы нарисовать содержимое одного холста на другом.

Используя метод `drawImage()` с другими методами, можно легко выполнять базовые операции над изображениями, результат которых можно получить с помощью метода `toDataURL()`. Однако если в контексте рисования выводится изображение не с текущего сайта, а из другого источника, при вызове метода `toDataURL()` произойдет ошибка. Например, если на странице на сайте www.example.com попытаться вывести изображение с сайта www.wrox.com, контекст будет воспринят как «грязный», что приведет к ошибке.

Тени

При рисовании фигур и путей в двухмерном контексте к ним автоматически добавляются тени согласно значениям следующих свойств:

1. `shadowColor` — цвет тени в формате CSS. По умолчанию черный.
2. `shadowOffsetX` — ширина тени по оси x. По умолчанию 0.
3. `shadowOffsetY` — ширина тени по оси y. По умолчанию 0.
4. `shadowBlur` — ширина области размытия по краям тени в пикселях. Если 0, края тени не размываются. По умолчанию 0.

Все эти свойства контекста можно читать и записывать. Достаточно задать их значения перед рисованием, и тени будут добавлены автоматически, например:

```
let context = drawing.getContext("2d");

// настройка тени
context.shadowOffsetX = 5;
context.shadowOffsetY = 5;
context.shadowBlur = 4;
context.shadowColor = "rgba(0, 0, 0, 0.5)";

// рисование красного квадрата
context.fillStyle = "#ff0000";
context.fillRect(10, 10, 50, 50);

// рисование синего квадрата
context.fillStyle = "rgba(0,0,255,1)";
context.fillRect(30, 30, 50, 50);
```

Этот код выводит на экран два квадрата с одинаковыми тенями (рис. 18.10).

Рис. 18.10

Градиенты

Работая с двухмерным контекстом, можно с легкостью создавать и изменять градиенты, которые представляются экземплярами типа `CanvasGradient`. Создать линейный градиент можно с помощью метода `createLinearGradient()`, который принимает четыре аргумента: начальные координаты x и y и конечные координаты x и y. Получив эти данные, он создает объект `CanvasGradient` соответствующих размеров и возвращает его.

После создания объекта градиента следует назначить ему границы методом `addColorStop()`. Он принимает два аргумента: смещение границы градиента и CSS-цвет. Смещением границы может быть число в интервале от 0 (первый цвет) до 1 (последний цвет), например:

```
let gradient = context.createLinearGradient(30, 30, 70, 70);

gradient.addColorStop(0, "white");
gradient.addColorStop(1, "black");
```

Этот объект `gradient` определяет градиент от точки (30, 30) до точки (70, 70) с начальным белым цветом и конечным черным. Далее можно назначить его свойству `fillStyle` или `strokeStyle` и нарисовать фигуру с градиентом:

```
// рисование красного квадрата
context.fillStyle = "#ff0000";
context.fillRect(10, 10, 50, 50);

// рисование квадрата с градиентной заливкой
context.fillStyle = gradient;
context.fillRect(30, 30, 50, 50);
```

Чтобы увидеть весь диапазон цветов градиента, нужно правильно подобрать координаты фигур. Приведенному коду соответствует рис. 18.11.

Рис. 18.11

Если сместить квадрат с градиентной заливкой ниже и правее, можно будет увидеть только часть градиента:

```
context.fillStyle = gradient;
context.fillRect(50, 50, 50, 50);
```

Этот код рисует черный квадрат с небольшим серым фрагментом в левом верхнем углу. Это объясняется тем, что левый верхний угол квадрата приходится как раз на середину градиента, который становится черным, даже не достигнув центра квадрата. Чтобы упростить расчет градиентов, можно использовать следующую функцию:

```
function createRectLinearGradient(context, x, y, width, height){
    return context.createLinearGradient(x, y, x+width, y+height);
}
```

Эта функция создает градиент на основе его начальных координат x и y, ширины и высоты, то есть с ней можно использовать те же числа, что и с методом `fillRect()`:

```
let gradient = createRectLinearGradient(context, 30, 30, 50, 50);

gradient.addColorStop(0, "white");
gradient.addColorStop(1, "black");

// рисование квадрата с градиентной заливкой
context.fillStyle = gradient;
context.fillRect(30, 30, 50, 50);
```

Отслеживание координат при работе с холстом — важная и нетривиальная задача, которую можно упростить за счет вспомогательных функций, таких как `createRectLinearGradient()`.

Радиальные градиенты создают методом `createRadialGradient()`, который принимает шесть аргументов. Первые три определяют центр и радиус начальной окружности, а последние три — те же параметры для конечной окружности. При работе с радиальными градиентами полезно представлять усеченный конус, основание и секущая плоскость которого соответствуют окружностям градиента.

В симметричном радиальном градиенте центры окружностей должны совпадать. Например, чтобы создать радиальный градиент в центре черного квадрата из предыдущего примера, нужно центрировать обе окружности в точке (55, 55), потому что противоположные углы квадрата расположены в точках (30, 30) и (80, 80):

```
let gradient = context.createRadialGradient(55, 55, 10, 55, 55, 30);

gradient.addColorStop(0, "white");
gradient.addColorStop(1, "black");

// рисование красного квадрата
```

```
context.fillStyle = "#ff0000";
context.fillRect(10, 10, 50, 50);

// рисование квадрата с градиентной заливкой
context.fillStyle = gradient;
context.fillRect(30, 30, 50, 50);
```

Рис. 18.12

Результат выполнения этого кода показан на рис. 18.12.

С радиальными градиентами работать немного сложнее, чем с линейными. В большинстве случаев для создания эффектов с их помощью используют окружности с совпадающими центрами и разными радиусами.

Узоры

Узоры — это просто повторяющиеся изображения, которые можно использовать для заливки или рисования контуров фигуры. Чтобы создать узор, вызовите метод `createPattern()`, передав в него HTML-элемент `<img>` и строку, определяющую способ повтора изображения. Второй аргумент может принимать такие же значения, что и CSS-свойство `background-repeat`, то есть `"repeat"`, `"repeat-x"`, `"repeat-y"` и `"no-repeat"`, например:

```
let image = document.images[0],
    pattern = context.createPattern(image, "repeat");

// рисование квадрата
context.fillStyle = pattern;
context.fillRect(10, 10, 150, 150);
```

Как и градиент, узор на самом деле начинается на холсте в точке (0, 0). Если узор задан в качестве стиля заливки, в конкретном месте холста просто демонстрируется его соответствующая часть. Например, приведенному коду соответствует узор, показанный на рис. 18.13.

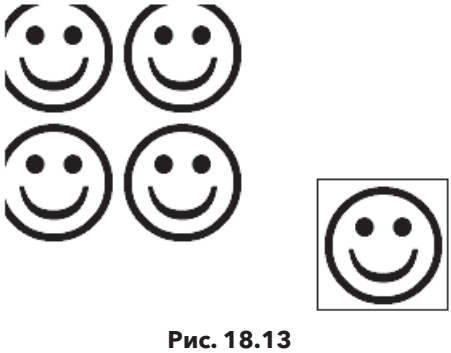

Рис. 18.13

Первым аргументом метода `createPattern()` также может быть элемент `<video>` или другой элемент `<canvas>`.

Работа с данными изображений

Одним из наиболее мощных механизмов двухмерного контекста является механизм получения необработанных данных изображения методом `getImageData()`. Он принимает четыре аргумента: координаты левого верхнего угла, ширину и высоту изображения, данные которого нужно получить. Например, получить данные области с размерами 50 на 50 и началом в точке (10, 5) можно следующим образом:

```
let imageData = context.getImageData(10, 5, 50, 50);
```

Этот метод возвращает экземпляр типа `ImageData`, который содержит всего три свойства: `width`, `height` и `data`. Свойство `data` является массивом с необработанными данными изображения. Каждый пиксель представляется в массиве четырьмя элементами, которые соответствуют красному, зеленому и синему компонентам, а также прозрачности (альфа-каналу) пикселя. Таким образом, данные первого пикселя содержатся в элементах с 0 по 3:

```
let data = imageData.data,
    red = data[0],
    green = data[1],
    blue = data[2],
    alpha = data[3];
```

Каждое значение в массиве может быть числом от 0 до 255 включительно. Доступ к необработанным данным изображения позволяет обрабатывать его разными способами. Например, можно создать простой черно-белый фильтр:

```
let drawing = document.getElementById("drawing");

// проверка полной поддержки элемента <canvas>
if (drawing.getContext){

    let context = drawing.getContext("2d"),
        image = document.images[0],
        imageData, data,
        i, len, average,
        red, green, blue, alpha;

    // вывод изображения без масштабирования
    context.drawImage(image, 0, 0);

    // получение данных изображения
    imageData = context.getImageData(0, 0, image.width, image.height);
    data = imageData.data;

    for (i=0, len=data.length; i < len; i+=4){

        red = data[i];
        green = data[i+1];
        blue = data[i+2];
        alpha = data[i+3];

        // получение среднего значения компонентов rgb
        average = Math.floor((red + green + blue) / 3);
```

```
        // задание новых цветов (без изменения прозрачности)
        data[i] = average;
        data[i+1] = average;
        data[i+2] = average;
    }

    // вывод черно-белого изображения
    imageData.data = data;
    context.putImageData(imageData, 0, 0);
}
```

Этот код выводит на экран исходное изображение, получает его данные и перебирает каждый пиксель в цикле `for`. Обратите внимание, что на каждой итерации цикла к значению `i` добавляется 4. Как только значения красного, зеленого и синего цветов получены, они усредняются, а затем среднее значение записывается в массив вместо каждого из трех исходных элементов. В результате все цвета заменяются серым цветом той же яркости. После цикла измененный массив `data` снова назначается объекту `imageData`. Наконец, метод `putImageData()` выводит черно-белое изображение на холсте.

Конечно, преобразование цветного изображения в черно-белое — это лишь пример того, что можно делать с необработанными значениями пикселей. Дополнительные сведения о создании фильтров для таких данных см. в статье Илмари Хайкинен «Создание фильтров изображений с помощью холста» (Ilmari Heikkinen, «Making Image Filters with Canvas») по адресу www.jsl5rocks.com/en/tutorials/canvas/imagefilters/.

> **ПРИМЕЧАНИЕ** Данные изображения доступны, только если холст не «загрязнен» ресурсом из другого домена, в противном случае при доступе к ним возникает JavaScript-ошибка.

Композиция изображений

Во всех операциях рисования в двухмерном контексте учитываются свойства `globalAlpha` и `globalCompositionOperation`. Свойство `globalAlpha` содержит число от 0 до 1 включительно, которое задает прозрачность для всех операций рисования. По умолчанию оно равно 0. Если с несколькими операциями нужно использовать одно значение прозрачности, следует присвоить его свойству `globalAlpha`, выполнить рисование и снова обнулить свойство, например:

```
// рисование красного квадрата
context.fillStyle = "#ff0000";
context.fillRect(10, 10, 50, 50);

// глобальное изменение прозрачности
context.globalAlpha = 0.5;

// рисование синего квадрата
context.fillStyle = "rgba(0,0,255,1)";
context.fillRect(30, 30, 50, 50);
```

```
// сброс значения
context.globalAlpha = 0;
```

Этот код рисует синий квадрат поверх красного. Поскольку перед его рисованием свойству `globalAlpha` присваивается значение 0.5, в итоговом изображении красный квадрат виден через синий.

Свойство `globalCompositionOperation` показывает, как новые фигуры должны сочетаться с уже имеющимся в контексте изображением. Оно может содержать одно из следующих строковых значений:

- `source-over` (значение по умолчанию) — новое изображение рисуется спереди существующего;
- `source-in` — новое изображение рисуется только там, где оно перекрывает существующее, а все остальное место становится прозрачным;
- `source-out` — новое изображение рисуется только там, где оно не перекрывает существующее, а все остальное место становится прозрачным;
- `source-atop` — новое изображение рисуется только там, где оно перекрывает существующее, а остальные места существующего изображения остаются неизменными;
- `destination-over` — новое изображение рисуется позади существующего и видно только через его прозрачные пиксели;
- `destination-in` — новое изображение рисуется позади существующего, а все места, где изображения не перекрываются, становятся прозрачными;
- `destination-out` — новое изображение стирает части существующего, с которыми перекрывается;
- `destination-atop` — новое изображение рисуется позади существующего, которое становится прозрачным там, где изображения не перекрываются;
- `lighter` — новое изображение объединяется с существующим, в результате получается более светлое изображение;
- `copy` — новое изображение стирает существующее, полностью заменяя его;
- `xor` — результат рисования получается путем применения исключающего «или» к существующему и новому изображениям.

Эти операции сложно описать словами или пояснить черно-белыми изображениями. Вот простой пример композиции изображений:

```
// рисование красного квадрата
context.fillStyle = "#ff0000";
context.fillRect(10, 10, 50, 50);

// настройка композиции
context.globalCompositeOperation = "destination-over";

// рисование синего квадрата
context.fillStyle = "rgba(0,0,255,1)";
context.fillRect(30, 30, 50, 50);
```

В обычной ситуации этот код рисует синий квадрат спереди красного, но из-за того, что свойству `globalCompositeOperation` присвоено значение `"destination-over"`, синий квадрат оказывается сзади.

При использовании свойства `globalCompositionOperation` не забывайте тестировать код в нескольких браузерах, потому что реализации этих операций все еще заметно различаются. В Safari и Chrome имеются проблемы, которые можно увидеть, перейдя по приведенному URL-адресу и сравнив результат с той же страницей в Internet Explorer или Firefox.

WEBGL

WebGL — это трехмерный контекст холста. В отличие от других веб-технологий, WebGL разрабатывается не в W3C, а в Khronos Group. Согласно веб-сайту этой организации, «Khronos Group — это некоммерческий консорциум, задачей которого является разработка свободных открытых стандартов параллельных вычислений, графических и динамических мультимедийных технологий для широкого диапазона платформ и устройств». Khronos Group разработала также ряд других графических API, таких как OpenGL ES 2.0, который лежит в основе WebGL.

Языки для работы с трехмерной графикой, такие как OpenGL, — сложная тема, и мы не будем углубляться в детали. Для использования WebGL рекомендуется познакомиться с OpenGL ES 2.0, потому что многие концепции в них одинаковы.

В этом разделе предполагается, что вы обладаете рабочими знаниями концепций OpenGL ES 2.0 и хотите узнать, как некоторые из них реализованы в WebGL. Дополнительные сведения об OpenGL доступны на сайте www.opengl.org, а на сайте www.learningwebgl.com можно найти серию отличных уроков по WebGL.

> **ПРИМЕЧАНИЕ** Типизированные массивы являются важной частью выполнения операций в WebGL. Они подробно рассматриваются в главе 5 «Ссылочные типы».

Контекст WebGL

Название контекста WebGL 2.0 в полностью поддерживающих его браузерах — `"webgl2"`, а контекста WebGL 1.0 — `"webgl"`. Браузеры, не поддерживающие WebGL, при попытке получить WebGL-контекст возвращают значение `null`. Прежде чем использовать контекст, всегда проверяйте возвращенное значение:

```
let drawing = document.getElementById("drawing");

// проверка полной поддержки элемента <canvas>
if (drawing.getContext){
```

```
    let gl = drawing.getContext("experimental-webgl");
    if (gl){
        // продолжение работы с WebGL
    }
}
```

В большинстве приложений и примеров WebGL-контекст называется `gl`, потому что методы и значения, относящиеся к OpenGL ES 2.0, обычно имеют префикс `"gl"`. Следование этой конвенции сделает JS-код более похожим на OpenGL-программу.

Основы WebGL

Как только WebGL-контекст получен, можно приступать к рисованию трехмерной графики. Как уже отмечалось, WebGL является версией OpenGL ES 2.0 для веб-приложений, поэтому концепции, описанные в этом разделе, на самом деле перенесены в JavaScript из OpenGL.

Чтобы настроить параметры WebGL-контекста, можно передать в метод `getContext()` второй аргумент — объект с одним или несколькими свойствами из перечисленных:

- `alpha` — если это свойство равно `true`, для контекста создается буфер альфа-канала. По умолчанию `true`.
- `depth` — если это свойство равно `true`, доступен 16-разрядный буфер глубины. По умолчанию `true`.
- `stencil` — если это свойство равно `true`, доступен 8-разрядный буфер трафарета. По умолчанию `false`.
- `antialias` — если это свойство равно `true`, выполняется сглаживание с использованием механизма, предлагаемого по умолчанию. По умолчанию `true`.
- `premultipliedAlpha` — если это свойство равно `true`, предполагается, что буфер рисования содержит предварительно умноженные значения альфа. По умолчанию `true`.
- `preserveDrawingBuffer` — если это свойство равно `true`, буфер рисования сохраняется после завершения рисования. По умолчанию `false`. Изменяйте это значение, только если хорошо понимаете, что делаете, иначе могут возникнуть проблемы с быстродействием.

Передать объект с параметрами в метод `getContext()` можно следующим образом:

```
let drawing = document.getElementById("drawing");

// проверка полной поддержки элемента <canvas>
if (drawing.getContext){

    let gl = drawing.getContext("webgl", { alpha: false});
    if (gl){
        // продолжение работы с WebGL
    }
}
```

Большинство параметров контекста используются лишь в нетривиальных сценариях, а во многих случаях подходят значения, предлагаемые по умолчанию.

Если WebGL-контекст невозможно создать с помощью метода `getContext()`, некоторые браузеры генерируют ошибку, поэтому лучше заключить его вызов в блок `try-catch`:

```
let drawing = document.getElementById("drawing"),

// проверка полной поддержки элемента <canvas>
if (drawing.getContext){
    try {
        gl = drawing.getContext("webgl");
    } catch (ex) {
        // пустой блок
    }

    if (gl){
        // продолжение работы с WebGL
    } else {
        alert("WebGL context could not be created.");
    }
}
```

Константы

Если вы знакомы с OpenGL, вам должны быть известны многие константы с префиксом `GL_`. В WebGL каждая константа доступна в объекте WebGL-контекста с префиксом `gl.` вместо префикса `GL_`. Например, константа `GL_COLOR_BUFFER_BIT` доступна как `gl.COLOR_BUFFER_BIT`. В WebGL есть такие аналоги большинства OpenGL-констант (хотя все же некоторые константы отсутствуют).

Именование методов

Имена многих OpenGL- и WebGL-методов включают сведения о типах их аргументов. Если метод может принимать разное количество аргументов разных типов, к его имени добавляется соответствующий суффикс. Число в нем указывает количество аргументов (от 1 до 4), а буква — их тип («f» для чисел с плавающей точкой или «i» для целых чисел). Например, метод `gl.uniform4f()` ожидает четыре числа с плавающей точкой, а метод `gl.uniform3i()` — три целых числа.

Многие методы также могут принимать массив вместо отдельных аргументов, на что указывает буква «v» (сокращение от «vector»). Так, метод `gl.uniform3iv()` принимает массив с тремя целыми числами. Помните об этой конвенции при обсуждении WebGL.

Подготовка к рисованию

Одно из первых действий при работе с WebGL-контекстом — заполнение элемента `<canvas>` сплошным цветом для подготовки к рисованию. Для этого сначала нужно

задать цвет методом `clearColor()`, который принимает четыре аргумента: значения красного, зеленого и синего цветов, а также прозрачность. Каждый аргумент должен быть числом от 0 до 1, определяющим долю значения в составе окончательного цвета, например:

```
gl.clearColor(0,0,0,1);         // черный
gl.clear(gl.COLOR_BUFFER_BIT);
```

Этот код задает для буфера цвета черный цвет, а затем вызывает метод `clear()`, который эквивалентен OpenGL-методу `glClear()`. Аргумент `gl.COLOR_BUFFER_BIT` указывает WebGL использовать ранее определенный цвет для заполнения области. С ее очистки начинаются практически все операции рисования.

Области просмотра и координаты

Перед началом рисования в WebGL имеет смысл определить область просмотра, которая по умолчанию занимает весь холст. Чтобы изменить область просмотра, вызовите метод `viewport()`, передав ему координаты начала области просмотра, а также ее ширину и высоту относительно холста. Например, следующий вызов определяет область просмотра во весь холст:

```
gl.viewport(0, 0, drawing.width, drawing.height);
```

Система координат области просмотра отличается от той, которая обычно используется на веб-странице. Как показано на рис. 18.14, левый нижний угол элемента `<canvas>` имеет координаты (0, 0), а правый верхний — (ширина–1, высота–1).

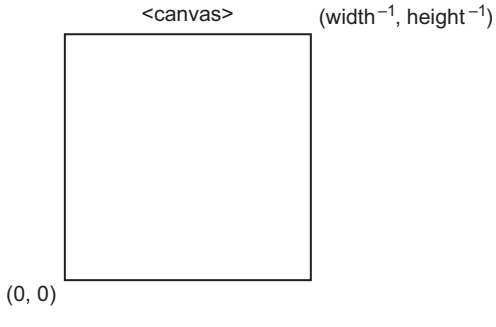

Рис. 18.14

Умение рассчитывать область просмотра позволяет использовать для рисования только часть элемента `<canvas>`, например:

```
// область просмотра - левая нижняя четверть элемента <canvas>
gl.viewport(0, 0, drawing.width/2, drawing.height/2);
// область просмотра - левая верхняя четверть элемента <canvas>
gl.viewport(0, drawing.height/2, drawing.width/2, drawing.height/2);
// область просмотра - правая нижняя четверть элемента <canvas>
gl.viewport(drawing.width/2, 0, drawing.width/2, drawing.height/2);
```

Внутри области просмотра используется другая система координат, показанная на рис. 18.15. Начало координат находится в центре области просмотра, ее левому нижнему углу соответствует точка (–1, –1), а правому верхнему — точка (1, 1).

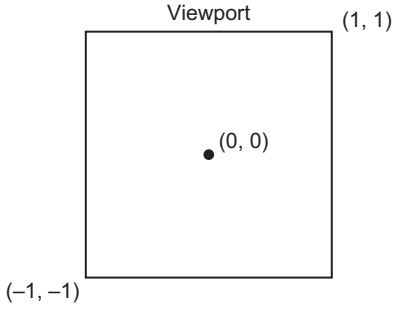

Рис. 18.15

Если при рисовании указать координаты вне области просмотра, например (1, 2), рисунок будет обрезан.

Буферы

В JavaScript информация о вершинах хранится в типизированных массивах, но для работы ее нужно преобразовать в WebGL-буферы. Для этого сначала необходимо создать буфер методом `gl.createBuffer()`, а затем связать его с WebGL-контекстом методом `gl.bindBuffer()`. После этого можно заполнить буфер данными, например:

```
let buffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, buffer);
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array([0, 0.5, 1]),
              gl.STATIC_DRAW);
```

Вызов метода `gl.bindBuffer()` делает объект `buffer` текущим буфером контекста, после чего все операции с буфером выполняются непосредственно с объектом `buffer`. Так, вызов `gl.bufferData()` не содержит явную ссылку на `buffer`, но все равно работает с ним. Последняя строка инициализирует буфер данными из массива `Float32Array`, в котором обычно хранится вся информация о вершинах. Если предполагается задействовать метод `drawElements()` для вывода содержимого буфера, можно указать константу `gl.ELEMENT_ARRAY_BUFFER`.

Последний аргумент метода `gl.bufferData()` показывает, как будет использоваться буфер. Им может быть одна из следующих констант:

- `gl.STATIC_DRAW` — данные будут загружены один раз и использованы многократно;
- `gl.STREAM_DRAW` — данные будут загружены один раз и использованы всего несколько раз;
- `gl.DYNAMIC_DRAW` — данные будут многократно изменяться и использоваться для рисования.

Если у вас нет солидного опыта работы с OpenGL, вероятнее всего, в большинстве случаев вы будете использовать константу `gl.STATIC_DRAW`.

Буферы остаются в памяти до выгрузки страницы-контейнера. Если буфер больше не требуется, лучше освободить занимаемую им память, вызвав метод `gl.deleteBuffer()`:

```
gl.deleteBuffer(buffer);
```

Ошибки

В отличие от JavaScript, WebGL-операции обычно не генерируют ошибки. Вместо этого после вызова метода, в котором может произойти ошибка, вы должны вызвать метод `gl.getError()`. Он возвращает одну из следующих констант, описывающих тип произошедшей ошибки:

- `gl.NO_ERROR` — последняя операция выполнена без ошибок (значение 0);
- `gl.INVALID_ENUM` — в метод, принимающий одну из WebGL-констант, передан неправильный аргумент;
- `gl.INVALID_VALUE` — вместо числа без знака использовано отрицательное число;
- `gl.INVALID_OPERATION` — операция не может быть выполнена в текущем состоянии;
- `gl.OUT_OF_MEMORY` — недостаточно памяти для выполнения операции;
- `gl.CONTEXT_LOST_WEBGL` — WebGL-контекст утрачен из-за внешнего события, такого как сбой электропитания.

Метод `gl.getError()` возвращает одно значение ошибки, поэтому каждый последующий его вызов может возвращать другое значение. Если ошибок несколько, это продолжается, пока не будет возвращено значение `gl.NO_ERROR`. Если вы выполнили несколько операций, для обработки ошибок можно использовать цикл с методом `getError()`:

```
let errorCode = gl.getError();
while(errorCode){
    console.log("Error occurred: " + errorCode);    // Сообщение об ошибке
    errorCode = gl.getError();
}
```

Если WebGL-сценарий не выводит правильный результат, добавление нескольких вызовов `gl.getError()` в код может помочь решить проблему.

Шейдеры

Шейдеры (shaders) — это еще одна концепция из OpenGL. В WebGL доступны шейдеры двух типов: *вершинные* (vertex shaders) и *фрагментные* (fragment shaders). Вершинные шейдеры служат для преобразования трехмерных вершин в двухмерные точки с целью их визуализации, а фрагментные — для вычисления правильного цвета пикселей. WebGL-шейдеры примечательны тем, что их создают не на

JavaScript, а на *GLSL* (OpenGL Shading Language), который никак не пересекается с JavaScript или C.

Создание шейдеров

GLSL — это C-подобный язык, специально предназначенный для определения OpenGL-шейдеров. Поскольку WebGL является реализацией OpenGL ES 2, OpenGL-шейдеры можно без изменений использовать в WebGL, что позволяет легко переносить графику из приложений для настольных компьютеров в веб-приложения.

У каждого шейдера есть метод `main()`, который многократно выполняется во время рисования. Передать данные в шейдер можно двумя способами: с помощью *атрибутов* (attributes) и *однородных значений* (uniforms). Атрибуты используются для передачи вершин в вершинные шейдеры, а однородные значения — для передачи констант в шейдеры обоих типов. Атрибуты и однородные значения определяются вне метода `main()` с помощью ключевых слов `attribute` и `uniform` соответственно, после которых указывается тип данных, а за ним имя. Вот простой пример вершинного шейдера:

```
// OpenGL Shading Language
// Шейдер от Бартека Дроздзя (Bartek Drozdz)
attribute vec2 aVertexPosition;

void main() {
        gl_Position = vec4(aVertexPosition, 0.0, 1.0);
}
```

Для этого шейдера определен единственный атрибут `aVertexPosition`, который представляет собой массив из двух элементов (тип данных `vec2`), определяющих координаты *x* и *y*. Вершинный шейдер всегда должен назначать специальной переменной `gl_Position` вершину из четырех частей, даже если ему переданы только две координаты. Шейдер в показанном примере создает новый массив из четырех элементов (`vec4`) и добавляет в него недостающие значения, преобразуя двухмерные координаты в трехмерные.

Фрагментные шейдеры похожи на вершинные, но передавать им данные можно только как однородные значения. Вот пример фрагментного шейдера:

```
// OpenGL Shading Language
// Шейдер от Бартека Дроздзя (Bartek Drozdz) из статьи по адресу
// http://www.netmagazine.com/tutorials/get-started-webgl-draw-square
uniform vec4 uColor;

void main() {
        gl_FragColor = uColor;
}
```

Фрагментный шейдер должен присваивать значение переменной `gl_FragColor`, которая задает цвет, используемый при рисовании. Для шейдера в показанном примере определяется однородный (uniform) цвет из четырех частей (`vec4`) с именем `uColor`, а сам шейдер только назначает полученное значение переменной `gl_FragColor`. Значение `uColor` изменить в шейдере нельзя.

ПРИМЕЧАНИЕ GLSL — непростой язык с множеством нюансов, которому посвящены целые книги. В этом разделе приведены лишь поверхностные сведения о нем, касающиеся WebGL. Дополнительные сведения о GLSL см. в книге «OpenGL Shading Language» (Randi J. Rost, Addison-Wesley, 2006).

Создание программ с шейдерами

Изначально браузеры не понимают GLSL-код, поэтому для создания программы с шейдерами необходима GLSL-строка, готовая к компиляции и компоновке. Ради простоты и удобства шейдеры обычно включают в код страницы с помощью элементов `<script>` с пользовательским атрибутом `type`. Применение неправильного атрибута `type` не позволяет браузеру интерпретировать содержимое элемента `<script>`, при этом обеспечивается легкий доступ к шейдеру, например:

```
<script type="x-webgl/x-vertex-shader" id="vertexShader">
attribute vec2 aVertexPosition;

void main() {
        gl_Position = vec4(aVertexPosition, 0.0, 1.0);
}
</script>
<script type="x-webgl/x-fragment-shader" id="fragmentShader">
uniform vec4 uColor;

void main() {
        gl_FragColor = uColor;
}
</script>
```

Содержимое такого элемента `<script>` можно извлечь с помощью свойства `text`:

```
let vertexGlsl = document.getElementById("vertexShader").text,
    fragmentGlsl = document.getElementById("fragmentShader").text;
```

Более сложные WebGL-приложения могут загружать шейдеры динамически, здесь же важно понять, что для использования шейдера требуется GLSL-строка.

Когда GLSL-строка получена, нужно создать объект шейдера, вызвав метод `gl.createShader()` и передав ему в качестве аргумента тип создаваемого шейдера (`gl.VERTEX_SHADER` или `gl.FRAGMENT_SHADER`). После этого следует назначить шейдеру исходный GLSL-код с помощью метода `gl.shaderSource()` и скомпилировать его методом `gl.compileShader()`, например:

```
let vertexShader = gl.createShader(gl.VERTEX_SHADER);
gl.shaderSource(vertexShader, vertexGlsl);
gl.compileShader(vertexShader);

let fragmentShader = gl.createShader(gl.FRAGMENT_SHADER);
gl.shaderSource(fragmentShader, fragmentGlsl);
gl.compileShader(fragmentShader);
```

Этот код создает два шейдера и сохраняет их в переменных `vertexShader` и `fragmentShader`, которые затем можно скомпоновать в программе следующим образом:

```
let program = gl.createProgram();
gl.attachShader(program, vertexShader);
gl.attachShader(program, fragmentShader);
gl.linkProgram(program);
```

Этот код создает переменную `program`, добавляет к программе шейдеры методом `attachShader()`, а затем инкапсулирует их в ней методом `gl.linkProgram()`. После компоновки программы можно дать WebGL-контексту команду использовать ее, вызвав метод `gl.useProgram()`:

```
gl.useProgram(program);
```

После вызова метода `gl.useProgram()` указанная программа будет использоваться во всех операциях рисования.

Передача значений шейдерам

Каждый из определенных нами шейдеров нуждается в данных для выполнения своей работы. Для передачи значений в шейдер нужно сначала найти переменную, которую необходимо заполнить. В случае однородной переменной это можно сделать с помощью метода `gl.getUniformLocation()`, который возвращает объект, представляющий расположение переменной в памяти. Затем это расположение можно задействовать для назначения данных, например:

```
let uColor = gl.getUniformLocation(program, "uColor");
gl.uniform4fv(uColor, [0, 0, 0, 1]);
```

Этот код находит однородную переменную `uColor` в объекте `program` и возвращает ее расположение в памяти. Во второй строке метод `gl.univorm4fv()` задает переменной `uColor` значение.

Похожая процедура соблюдается и при работе с переменными-атрибутами в вершинных шейдерах. Расположение переменной-атрибута можно получить методом `gl.getAttribLocation()`, а когда оно получено, его можно использовать следующим образом:

```
let aVertexPosition = gl.getAttribLocation(program, "aVertexPosition");
gl.enableVertexAttribArray(aVertexPosition);
gl.vertexAttribPointer(aVertexPosition, itemSize, gl.FLOAT, false, 0, 0);
```

Этот код получает расположение атрибута `aVertexPosition` и инициирует его использование с помощью метода `gl.enableVertexAttribArray()`. Последняя строка создает указатель на последний буфер, заданный с помощью метода `gl.bindBuffer()`, и сохраняет его в переменной `aVertexPosition`, чтобы его мог использовать вершинный шейдер.

Отладка шейдеров и программ

Как и другие действия в WebGL, операции с шейдерами могут завершаться ошибками без уведомления об этом. Если вы считаете, что могла произойти ошибка, нужно вручную запросить сведения о шейдере или программе у WebGL-контекста.

Чтобы получить статус шейдера после попытки его компиляции, вызовите метод `gl.getShaderParameter()`:

```
if (!gl.getShaderParameter(vertexShader, gl.COMPILE_STATUS)){
    alert(gl.getShaderInfoLog(vertexShader));
}
```

Этот код проверяет статус компиляции шейдера `vertexShader`. Если он был скомпилирован успешно, метод `gl.getShaderParameter()` возвращает `true`. Если он возвращает `false`, это означает, что при компиляции шейдера произошла ошибка. Получить сведения о ней можно методом `gl.getShaderInfoLog()`, который принимает шейдер и возвращает строку с описанием проблемы. Методы `gl.getShaderParameter()` и `gl.getShaderInfoLog()` можно использовать и с вершинными, и с фрагментными шейдерами.

Для проверки статуса программы служит похожий метод `gl.getProgramParameter()`. Чаще всего сбои программ происходят во время компоновки, статус которой можно проверить с помощью следующего кода:

```
if (!gl.getProgramParameter(program, gl.LINK_STATUS)){
    alert(gl.getProgramInfoLog(program));
}
```

Как и `gl.getShaderParameter()`, метод `gl.getProgramParameter()` возвращает `true`, если компоновка выполнена успешно, и `false` в противном случае. Кроме того, доступен метод `gl.getProgramInfoLog()`, с помощью которого можно получить сведения о сбое программы.

Эти методы используются преимущественно во время отладки. Если от них не зависит другой код, их можно удалить из окончательного продукта.

Обновление с GLSL 100 до GLSL 300

Одним из основных изменений в WebGL2 является обновление до шейдеров GLSL 3.00 ES. Это обновление предоставляет широкий спектр новых функций шейдеров, таких как 3D-текстуры, которые доступны на устройствах, поддерживающих OpenGL ES 3.0. Для использования обновленной версии шейдера первую строку шейдеров нужно указать следующим образом:

```
#version 300 es
```

Это обновление требует нескольких синтаксических изменений:

- Переменные атрибута объявляются с использованием ключевого слова `in` вместо `attribute`.

- Переменные, использующие ключевое слово `varying` для таких вещей, как вершинные или фрагментные шейдеры, теперь должны использовать вход или выход в зависимости от их поведения относительно шейдера.
- Предопределенная выходная переменная `gl_FragColor` больше не существует; фрагментные шейдеры должны объявить свою собственную переменную `out` для вывода цвета.
- Функции поиска текстур, такие как `texture2D` и `textureCube`, были объединены в одну функцию `texture`.

Рисование

WebGL позволяет рисовать только точки, линии и треугольники, а все остальные фигуры нужно составлять из этих трех базовых компонентов, рисуемых в трехмерном пространстве. Рисование выполняется методами `drawArrays()` и `drawElements()`, первый из которых работает с буферами массивов, а второй — с буферами массивов элементов.

Первым аргументом методов `gl.drawArrays()` и `drawElements()` является константа, задающая тип фигуры, которую нужно нарисовать. Поддерживаемые значения:

- `gl.POINTS` — указывает, что каждую вершину нужно нарисовать как точку.
- `gl.LINES` — указывает, что массив содержит последовательность вершин, которые нужно соединить линиями. Каждое множество вершин содержит начальную и конечную точки, так что количество вершин в массиве должно быть четным, чтобы были нарисованы все линии.
- `gl.LINE_LOOP` — указывает, что массив содержит последовательность вершин, которые нужно соединить линиями. Линии рисуются от первой вершины ко второй, от второй к третьей и т. д., пока не будет достигнута последняя вершина. После этого последняя вершина соединяется с первой. В результате получается контур фигуры.
- `gl.LINE_STRIP` — то же, что и `gl.LINE_LOOP`, но без рисования линии от последней вершины к первой.
- `gl.TRIANGLES` — указывает, что массив содержит последовательность вершин треугольников. Каждый треугольник рисуется отдельно от предыдущего без общих вершин, если иное не указано явно.
- `gl.TRIANGLES_STRIP` — то же, что и `gl.TRIANGLES`, но каждая последующая вершина после первых трех определяет новый треугольник, включающий кроме нее две предыдущие вершины. Например, если массив содержит вершины *A*, *B*, *C* и *D*, первым треугольником будет *ABC*, а вторым — *BCD*.
- `gl.TRIANGLES_FAN` — то же, что и `gl.TRIANGLES`, но каждая последующая вершина после первых трех определяет новый треугольник, включающий кроме нее предыдущую и первую вершины. Например, если массив содержит вершины *A*, *B*, *C* и *D*, первым треугольником будет *ABC*, а вторым — *ACD*.

Метод `gl.drawArrays()` принимает одно из этих значений в качестве первого аргумента, начальный индекс в буфере массива в качестве второго и количество вершин в буфере массива в качестве третьего. В следующем примере он рисует на холсте один треугольник:

```
// предполагается, что область просмотра очищена
// описанных ранее с помощью шейдеров

// определение координат трех вершин
let vertices = new Float32Array([ 0, 1, 1, -1, -1, -1 ]),
    buffer = gl.createBuffer(),
    vertexSetSize = 2,
    vertexSetCount = vertices.length/vertexSetSize,
    uColor, aVertexPosition;

// запись данных в буфер
gl.bindBuffer(gl.ARRAY_BUFFER, buffer);
gl.bufferData(gl.ARRAY_BUFFER, vertices, gl.STATIC_DRAW);

// передача цвета фрагментному шейдеру
uColor = gl.getUniformLocation(program, "uColor");
gl.uniform4fv(uColor, [ 0, 0, 0, 1 ]);

// передача информации о вершинах вершинному шейдеру
aVertexPosition = gl.getAttribLocation(program, "aVertexPosition");
gl.enableVertexAttribArray(aVertexPosition);
gl.vertexAttribPointer(aVertexPosition, vertexSetSize, gl.FLOAT,
                       false, 0, 0);

// рисование треугольника
gl.drawArrays(gl.TRIANGLES, 0, vertexSetCount);
```

Массив `Float32Array` в этом примере содержит три вершины, каждая из которых имеет две координаты. Размер и количество вершин важно отслеживать для использования в дальнейших вычислениях. Переменной `vertexSetSize`, которая определяет количество координат вершины, присваивается значение 2, тогда как количество вершин `vertexSetCount` вычисляется. Затем информация о вершинах сохраняется в буфере и фрагментному шейдеру передаются сведения о цвете.

Вершинному шейдеру передается количество координат вершины, при этом указывается, что они представлены числами с плавающей точкой (`gl.FLOAT`). Четвертым аргументом метода `gl.vertexAttribPointer` является логическое значение, указывающее, что координаты не нормализованы. Пятый аргумент — это *значение шага* (stride value), которое показывает, сколько элементов массива нужно пропустить для получения следующего значения. Если вы не уверены, каким оно должно быть, используйте значение 0. Последний аргумент метода определяет начальное смещение в массиве. Значение 0 соответствует первому элементу.

Наконец, метод `gl.drawArrays()` рисует треугольник. Аргумент `gl.TRIANGLES` предписывает ему нарисовать треугольник с вершинами (0, 1), (1, –1) и (–1, –1), залив его цветом, переданным фрагментному шейдеру. Второй аргумент метода указывает

начальное смещение в буфере, а последний — общее количество вершин. Результат выполнения этого кода показан на рис. 18.16.

Рис. 18.16

Передав другой первый аргумент методу `gl.drawArrays()`, можно изменить способ рисования треугольника. Два возможных варианта показаны на рис. 18.17.

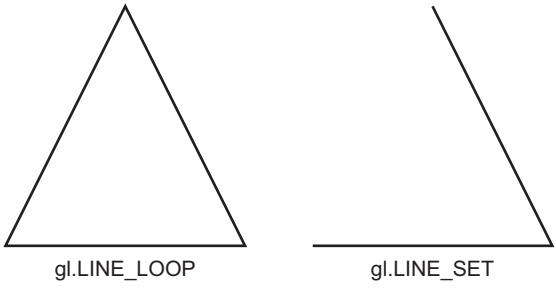

Рис. 18.17

Текстуры

С DOM-изображениями можно использовать WebGL-текстуры. Для этого нужно создать текстуру методом `gl.createTexture()`, а затем связать с ней изображение. Если изображение еще не загружено, для его динамической загрузки можно создать экземпляр типа `Image`. Текстура не инициализируется, пока изображение не загружено полностью, так что настраивать ее нужно после события `load`, например:

```
let image = new Image(),
    texture;
image.src = "smile.gif";
image.onload = function(){
    texture = gl.createTexture();
    gl.bindTexture(gl.TEXTURE_2D, texture);
    gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, true);

    gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE,
                  image);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.NEAREST);
```

```
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST);

    // очистка текущей текстуры
    gl.bindTexture(gl.TEXTURE_2D, null);
}
```

Если забыть об использовании DOM-изображения, эти действия почти не отличаются от создания OpenGL-текстуры. Самое главное отличие — настройка формата хранения пикселя с помощью метода `gl.pixelStorei()`. Константа `gl.UNPACK_FLIP_Y_WEBGL` уникальна для WebGL, и ее нужно использовать в большинстве ситуаций при загрузке изображений одного из популярных в интернете форматов. Это объясняется тем, что системы координат GIF-, JPEG- и PNG-изображений отличаются от внутренней системы координат в WebGL. Без этой константы изображение будет показано вверх ногами.

Текстуры должны находиться в том же домене, что и страница-контейнер, или храниться на сервере, на котором для изображений включен обмен ресурсами с запросом происхождения (Cross-Origin Resource Sharing, CORS).

> **ПРИМЕЧАНИЕ** Текстуры могут быть основаны на изображениях, на видеороликах, загруженных в элемент `<video>`, и даже на других элементах `<canvas>`. На видео распространяются те же ограничения относительно исходных доменов.

Чтение пикселей

Как и при работе с двухмерным контекстом, вы можете читать пиксели из WebGL-контекста. Метод `readPixels()` принимает те же аргументы, что и в OpenGL, только его последним аргументом должен быть типизированный массив. Данные пикселей считываются из буфера кадров в типизированный массив. Аргументами метода `readPixels()` являются координаты x и y, ширина, высота, формат изображения, тип и типизированный массив. Первые четыре аргумента показывают расположение считываемых пикселей. Форматом изображения почти всегда является `gl.RGBA`. Тип определяет тип данных, которые будут сохранены в типизированном массиве, и имеет следующие ограничения:

- если типом является `gl.UNSIGNED_BYTE`, типизированным массивом должен быть `Uint8Array`;
- если типом является `gl.UNSIGNED_SHORT_5_6_5`, `gl.UNSIGNED_SHORT_4_4_4_4` или `gl.UNSIGNED_SHORT_5_5_5_1`, типизированным массивом должен быть `Uint16Array`.

Вот простой пример:

```
let pixels = new Uint8Array(25*25);
gl.readPixels(0, 0, 25, 25, gl.RGBA, gl.UNSIGNED_BYTE, pixels);
```

Этот код читает из буфера кадров область размерами 25 × 25, сохраняя данные пикселей в массиве `pixels`. Цвет каждого пикселя представляется массивом из четырех элементов — по одному для красного, зеленого, синего компонентов и прозрачности.

Значениями элементов могут быть числа от 0 до 255 включительно. Не забудьте инициализировать типизированный массив с учетом ожидаемого объема данных.

Вызов метода `readPixels()` до рисования обновленного WebGL-изображения выполняется без сюрпризов. По завершении рисования кадровый буфер переходит в первоначальное очищенное состояние; если после этого вызвать метод `readPixels()`, он возвратит данные очищенного буфера. Если требуется считать данные пикселей после рисования, при инициализации WebGL-контекста нужно указать уже упоминавшийся параметр `preserveDrawingBuffer`:

```
let gl = drawing.getContext("experimental-webgl",
                            { preserveDrawingBuffer: true; });
```

Этот параметр предписывает буферу кадров оставаться в последнем состоянии до следующей операции рисования. Это снижает быстродействие, так что лучше не использовать его без необходимости.

Сравнение WebGL1 и WebGL2

Код, написанный для WebGL1, почти на 100 процентов совместим с WebGL2. При использовании контекста `webgl2` единственное необходимое изменение кода для обеспечения совместимости — обработка расширений. В WebGL2 многие расширения стали функциями по умолчанию.

Например, чтобы использовать буферы отрисовки в WebGL1, перед использованием нужно проверить расширение следующим образом:

```
let ext = gl.getExtension('WEBGL_draw_buffers');

if (!ext) {
    // обработка пропущенных расширений
} else {
    ext.drawBuffersWEBGL([...])
}
```

В WebGL2 это больше не требуется, поскольку функция доступна непосредственно как метод объекта контекста:

```
gl.drawBuffers([...]);
```

Все следующие расширения стали стандартными функциями:

- `ANGLE_instanced_arrays`
- `EXT_blend_minmax`
- `EXT_frag_depth`
- `EXT_shader_texture_lod`
- `OES_element_index_uint`
- `OES_standard_derivatives`
- `OES_texture_float`

- `OES_texture_float_linear`
- `OES_vertex_array_object`
- `WEBGL_depth_texture`
- `WEBGL_draw_buffers`
- `Доступ к текстуре Vertex Shader`

ПРИМЕЧАНИЕ Есть отличный пост, посвященный основам обновления WebGL, который можно найти по адресу https://webgl2fundamentals.org/webgl/lessons/webgl1-to-webgl2.html.

ИТОГИ

`requestAnimationFrame` — это простой, но элегантный инструмент, который позволяет JavaScript подключаться к циклу отрисовки браузера, чтобы эффективно выполнять визуальные манипуляции со страницей.

Элемент `<canvas>` из HTML5 предоставляет API, предназначенный для динамического создания графики в одном из двух специальных контекстов. Двухмерный контекст поддерживает следующие примитивные операции:

- настройка цветов и узоров заливки и контуров;
- рисование прямоугольников;
- рисование путей;
- рисование текста;
- создание градиентов и узоров.

Второй контекст, трехмерный, называется WebGL. WebGL — это браузерная версия языка OpenGL ES 2.0, который разработчики игр часто используют для программирования компьютерной графики. WebGL-контекст поддерживает гораздо более мощную функциональность, чем двухмерный контекст, в том числе:

- вершинные и фрагментные шейдеры, создаваемые на языке GLSL (OpenGL Shading Language);
- типизированные массивы, которые могут содержать только числа определенных типов;
- текстуры и различные операции над ними.

19

Работа с формами

- Общие сведения о формах
- Работа с текстовыми полями и проверка их содержимого
- Использование других элементов управления форм

Загрузки с сайта wrox.com для этой главы

Обратите внимание, что все примеры кода для этой главы доступны на веб-сайте книги по ссылке www.wrox.com/go/projavascript4e на вкладке Downloads.

Одной из первоначальных целей применения JavaScript было разделение ответственности за обработку форм между сервером и браузером. В отличие от веб-технологий и JavaScript, веб-формы с тех пор изменились незначительно. К сожалению, стандартной функциональности веб-форм оказалось недостаточно для решения типичных проблем, и со временем разработчики начали расширять возможности форм с помощью JavaScript.

ОБЩИЕ СВЕДЕНИЯ О ФОРМАХ

Веб-форма представляется HTML-элементом `<form>` в HTML и типом `HTMLFormElement` в JavaScript. Тип `HTMLFormElement` наследуется от типа `HTMLElement`, от которого получает все стандартные свойства HTML-элементов. К ним он добавляет следующие свойства и методы:

- `acceptCharset` — кодировки, которые может обрабатывать сервер (эквивалент HTML-атрибута `accept-charset`);
- `action` — URL-адрес для отправки запроса (эквивалент HTML-атрибута `action`);

- `elements` — коллекция `HTMLCollection`, содержащая все элементы управления формы;
- `enctype` — тип кодировки запроса (эквивалент HTML-атрибута `enctype`);
- `length` — количество элементов управления формы;
- `method` — тип отправляемого HTTP-запроса, обычно `"get"` или `"post"` (эквивалент HTML-атрибута `method`);
- `name` — имя формы (эквивалент HTML-атрибута `name`);
- `reset()` — сбрасывает все поля формы, восстанавливая значения, предлагаемые по умолчанию;
- `submit()` — отправляет данные формы;
- `target` — имя окна, используемого для отправки запроса и получения ответа (эквивалент HTML-атрибута `target`).

Ссылку на элемент `<form>` можно получить разными способами. Чаще всего ему назначают атрибут `id` подобно другим элементам, что позволяет использовать метод `getElementById()`, например:

```
let form = document.getElementById("form1");
```

Все формы на странице содержатся в коллекции `document.forms`. Каждая форма в ней доступна по числовому индексу и по имени:

```
// получение первой формы на странице
let firstForm = document.forms[0];

// получение формы с именем "form2"
let myForm = document.forms["form2"];
```

Старые браузеры и браузеры со строгой обратной совместимостью добавляют также каждую форму с именем к объекту `document` в качестве его свойства, например, форма `"form2"` доступна как `document.form2`. Использовать этот формат доступа не рекомендуется, потому что он подвержен ошибкам и в будущем его поддержка браузерами может быть прекращена.

Имейте в виду, что формы могут иметь и идентификатор, и имя (`id` и `name`), которые могут быть разными.

Отправка данных формы

Данные формы отправляются серверу, когда пользователь щелкает на кнопке отправки или на графической кнопке. Кнопку отправки представляет элемент `<input>` или `<button>`, у которого атрибут `type` имеет значение `"submit"`, а графическую кнопку — элемент `<input>`, у которого атрибут `type` имеет значение `"image"`. Вот три примера кнопок, каждая из которых отправляет свою форму серверу:

```
<!-- обобщенная кнопка отправки -->
<input type="submit" value="Submit Form">
```

```
<!-- пользовательская кнопка отправки -->
<button type="submit">Submit Form</button>

<!-- графическая кнопка -->
<input type="image" src="graphic.gif">
```

Кнопка отправки может передать данные формы при нажатии клавиши Enter, если фокус принадлежит одному из элементов управления формы (исключение — поле `textarea`, для которого при нажатии клавиши Enter выполняется перевод строки). Данные формы без кнопки отправки при нажатии клавиши Enter не передаются.

Когда данные формы отправляется таким способом, непосредственно перед отправкой запроса серверу генерируется событие `submit`. В его обработчике можно проверить введенные в форме данные и при необходимости заблокировать их отправку, отменив для события действие, предлагаемое по умолчанию, например:

```
let form = document.getElementById("myForm");

form.addEventListener("submit", (event) => {
    // отмена отправки данных формы
    event.preventDefault();
});
```

Вызов его метода `preventDefault()` останавливает отправку данных формы. Как правило, так делают, если форма содержит недопустимые данные, которые не имеет смысла отправлять серверу.

Кроме того, в любой момент можно отправить данные формы программно, вызвав ее метод `submit()`, который не требует наличия кнопки отправки на форме, например:

```
let form = document.getElementById("myForm");

// отправка данных формы
form.submit();
```

При такой отправке данных формы событие `submit` не генерируется, так что не забудьте проверить данные до вызова метода `submit()`.

Одна из главных проблем с отправкой данных форм — многократная отправка. Иногда пользователи от нетерпения щелкают на кнопке отправки несколько раз, что в лучшем случае просто нагружает сервер (которому приходится обрабатывать несколько идентичных запросов), а в худшем может привести к убыткам и другим неприятностям (например, если пользователь оформит несколько заказов вместо одного). Эта проблема имеет два решения: можно отключать кнопку отправки после отправки данных формы или отменять последующие попытки отправки данных в обработчике события `submit`.

Сброс формы

Сбросить форму можно нажав на кнопку сброса — элемент `<input>` или `<button>`, у которого атрибут `type` имеет значение `"reset"`, например:

```
<!-- обобщенная кнопка сброса -->
<input type="reset" value="Reset Form">

<!-- пользовательская кнопка сброса -->
<button type="reset">Reset Form</button>
```

Обе эти кнопки сбрасывают форму, при этом в ее полях восстанавливаются значения, которые имели место при начальной визуализации страницы. Если поле первоначально было пустым, оно снова становится пустым.

При сбросе формы с помощью кнопки сброса генерируется событие `reset`, позволяющее отменить сброс, например:

```
let form = document.getElementById("myForm");

form.addEventListener("reset", (event) => {
    event.preventDefault();
});
```

Кроме того, сбросить форму можно программно, вызвав метод `reset()`:

```
let form = document.getElementById("myForm");

// сброс формы
form.reset();
```

В отличие от метода `submit()`, метод `reset()` генерирует событие `reset`, как если бы был выполнен щелчок на кнопке сброса.

> **ПРИМЕЧАНИЕ** Многие разработчики плохо относятся к сбросу веб-форм. Он часто дезориентирует пользователя и при случайном выполнении раздражает. Сброс формы почти никогда не требуется – как правило, достаточно предоставить вместо кнопки сброса кнопку возврата на предыдущую страницу.

Поля форм

Как и другие элементы страницы, элементы форм доступны с помощью встроенных DOM-методов. Кроме того, все элементы любой формы содержатся в ее коллекции `elements` — упорядоченном списке ссылок на все поля формы и все элементы `<input>`, `<textarea>`, `<button>`, `<select>` и `<fieldset>`. Поля формы хранятся в коллекции `elements` в том порядке, в котором они расположены в разметке, и индексируются по позиции и имени, например:

```
let form = document.getElementById("form1");

// получение первого поля формы
let field1 = form.elements[0];

// получение поля с именем "textbox1"
let field2 = form.elements["textbox1"];

// получение количества полей
let fieldCount = form.elements.length;
```

Если одно имя идентифицирует несколько элементов управления формы, как в случае переключателей, все такие элементы возвращаются в коллекции `HTMLCollection`. Возьмем для примера следующий HTML-код:

```
<form method="post" id="myForm">
    <ul>
        <li><input type="radio" name="color" value="red">Red</li>
        <li><input type="radio" name="color" value="green">Green</li>
        <li><input type="radio" name="color" value="blue">Blue</li>
    </ul>
</form>
```

Эта форма содержит три переключателя с именем `"color"`, которое связывает их вместе. В этом случае выражение `elements["color"]` возвращает коллекцию `NodeList`, содержащую все три элемента, тогда как выражение `elements[0]` возвращает только первый элемент:

```
let form = document.getElementById("myForm");

let colorFields = form.elements["color"];
console.log(colorFields.length);    // 3

let firstColorField = colorFields[0];
let firstFormField = form.elements[0];
console.log(firstColorField === firstFormField);    // true
```

Этот код показывает, что первое поле формы, доступное как `form.elements[0]`, совпадает с первым элементом коллекции `form.elements["color"]`.

> **ПРИМЕЧАНИЕ** Элементы формы доступны также как ее свойства, например, свойство `form[0]` представляет первое поле формы, а `form["color"]` – именованное поле. Эти свойства всегда возвращают те же значения, что и их эквиваленты в коллекции `elements`. Этот подход поддерживается ради сохранения обратной совместимости со старыми браузерами, и вместо него следует использовать коллекцию `elements`.

Общие свойства полей форм

За исключением элемента `<fieldset>`, все поля форм имеют несколько общих свойств. Так как многие поля форм представляет тип `<input>`, одни свойства используются только с полями некоторых типов, а другие доступны независимо от типа поля. Общие свойства и методы полей форм таковы:

- `disabled` — логическое значение, указывающее, отключено ли поле;
- `form` — указатель на форму, к которой относится поле (это свойство доступно только для чтения);
- `name` — имя поля;
- `readOnly` — логическое значение, указывающее, доступно ли поле только для чтения;

- `tabIndex` — порядок перехода по нажатию клавиши табуляции;
- `type` — тип поля (`"checkbox"`, `"radio"` и т. д.);
- `value` — значение поля, отправляемое серверу; у полей добавления файлов это свойство доступно только для чтения и содержит путь к файлу на компьютере.

Все свойства, кроме `form`, можно изменять динамически, например:

```
let form = document.getElementById("myForm");
let field = form.elements[0];

// изменение значения
field.value = "Another value";

//проверка значения form
console.log(field.form === form);    // true

// установка фокуса для поля
field.focus();

// отключение поля
field.disabled = true;

// изменение типа поля (не рекомендуется, но возможно для элементов <input>)
field.type = "checkbox";
```

Возможность динамически изменять свойства полей форм позволяет в любой момент модифицировать форму самыми разными способами. Например, при работе с веб-формами пользователи иногда щелкают на кнопке отправки несколько раз. В приложениях электронной коммерции это может привести к выставлению нескольких счетов, что крайне нежелательно. Для решения этой проблемы можно отключить кнопку отправки в обработчике события `submit`:

```
// предотвращение многократной отправки данных формы
let form = document.getElementById("myForm");
form.addEventListener("submit", (event) => {
    let target = EventUtil.getTarget(event);

    // получение кнопки отправки
    let btn = target.elements["submit-btn"];

    // отключение кнопки отправки
    btn.disabled = true;
});
```

Этот код подключает к форме обработчик события `submit`, который получает кнопку отправки и присваивает ее свойству `disabled` значение `true`. Событие `click` кнопки отправки в этом случае бесполезно, потому что одни браузеры генерируют его раньше, чем событие `submit`, а другие наоборот. Если браузер сначала генерирует событие `click`, кнопка будет отключена до отправки данных формы, которые вообще нельзя будет отправить. По этой причине кнопку отправки следует отключать в обработчике события `submit`. Имейте в виду, что этот подход не годится, если данные

формы отправляются без использования кнопки отправки, потому что событие `submit` генерируется только этой кнопкой.

Свойство `type` есть у всех полей формы, кроме `<fieldset>`. У элементов `<input>` оно имеет то же значение, что и HTML-атрибут `type`. Значения свойства `type` у других элементов представлены в таблице.

ОПИСАНИЕ	ПРИМЕР HTML-КОДА	ЗНАЧЕНИЕ СВОЙСТВА TYPE
Список с возможностью одиночного выбора	`<select>...</select>`	`"select-one"`
Список с возможностью множественного выбора	`<select multiple>...</select>`	`"select-multiple"`
Пользовательская кнопка	`<button>...</button>`	`"submit"`
Пользовательская кнопка (не отправки)	`<button type="button">...</button>`	`"button"`
Пользовательская кнопка сброса	`<button type="reset">...</button>`	`"reset"`
Пользовательская кнопка отправки	`<button type="submit">...</button>`	`"submit"`

У элементов `<input>` и `<button>` свойство `type` можно изменять динамически, а у элемента `<select>` оно доступно только для чтения.

Общие методы полей форм

У каждого поля формы есть методы `focus()` и `blur()`. Метод `focus()` назначает фокус полю формы, то есть делает его активным, после чего поле начинает реагировать на события клавиатуры. Например, в текстовом поле, получившем фокус, появляется курсор, показывающий, что оно готово принимать ввод. Метод `focus()` чаще всего используется для привлечения внимания к какой-то части страницы. Например, при загрузке формы фокус часто назначают ее первому полю. Это можно сделать, вызвав метод `focus()` для первого поля в обработчике события `load`:

```
window.addEventListener("load", (event) => {
    document.forms[0].elements[0].focus();
});
```

При выполнении этого кода возникнет ошибка, если первым полем формы является элемент `<input>` типа `"hidden"` или если поле было скрыто с помощью CSS-свойства `display` или `visibility`.

В спецификации HTML5 для полей формы предлагается атрибут `autofocus`, который в поддерживающих его браузерах автоматически назначает фокус элементу без использования JS-кода, например:

```
<input type="text" autofocus>
```

Чтобы предыдущий пример правильно работал с атрибутом `autofocus`, нужно сначала определить, задан ли он, и если да, не вызывать метод `focus()`:

```
window.addEventListener("load", (event) => {
    let element = document.forms[0].elements[0];

    if (element.autofocus !== true) {
        element.focus();
        console.log("JS focus");
    }
});
```

Поскольку `autofocus` является логическим атрибутом, в поддерживающих его браузерах свойство `autofocus` равно `true` (если атрибут не поддерживается, оно содержит пустую строку). Таким образом, этот код вызывает метод `focus()`, только если свойство `autofocus` не равно `true`, что обеспечивает совместимость с последующими версиями браузеров. Свойство `autofocus` поддерживается в большинстве современных браузеров. Добавлена ограниченная поддержка в браузерах iOS Safari, Opera Mini и Internet Explorer 10 и более ранних версий.

> **ПРИМЕЧАНИЕ** По умолчанию фокус можно назначать только элементам форм. Чтобы назначить фокус любому другому элементу, нужно присвоить его свойству `tabIndex` значение `-1` и вызвать метод `focus()`. Этот прием поддерживают все браузеры, кроме Opera.

Метод `blur()` противоположен методу `focus()`: он отменяет для элемента фокус, но не назначает его никакому другому элементу. Когда не было атрибута `readonly`, этот метод использовался для создания полей, доступных только для чтения, а сейчас он применяется редко. Вот пример его вызова:

```
document.forms[0].elements[0].blur();
```

Общие события полей форм

В дополнение к событиям мыши, клавиатуры, изменения DOM-структуры и HTML-событиям все поля форм поддерживают следующие три события:

- `blur` — генерируется при утрате фокуса полем;
- `change` — генерируется для элементов `<input>` и `<textarea>` при утрате фокуса, если свойство `value` было изменено, а также для элементов `<select>` при выборе другого элемента списка;
- `focus` — генерируется при получении фокуса полем.

События `blur` и `focus` возникают в результате действий пользователя и при вызове методов `blur()` и `focus()` соответственно. Эти события одинаковы у всех полей формы, тогда как событие `change` генерируется для разных элементов управления в разное время. Для элементов `<input>` и `<textarea>` оно возникает при утрате фокуса, если за время нахождения элемента в фокусе у него изменилось свойство `value`. Для

элементов <select> событие change возникает, когда пользователь выбирает другой элемент списка, при этом не требуется, чтобы элемент утратил фокус.

События focus и blur обычно применяют для изменения пользовательского интерфейса, вывода визуальных подсказок или доступа к дополнительной функциональности элемента (например, для вывода раскрывающегося меню с возможными значениями текстового поля). С помощью события change обычно проверяют данные, введенные в поле. Например, если текстовое поле должно принимать только числа, событие focus можно использовать для изменения фонового цвета поля при его активации, событие blur — для восстановления цвета, предлагаемого по умолчанию, а событие change — для изменения фонового цвета на красный при вводе нецифровых символов:

```
let textbox = document.forms[0].elements[0];

textbox.addEventListener("focus", (event) => {
    let target = event.target;
    if (target.style.backgroundColor != "red") {
        target.style.backgroundColor = "yellow";
    }
});

textbox.addEventListener("blur", (event) => {
    let target = event.target;
    target.style.backgroundColor = /[^\d]/.test(target.value) ? "red" : "";
});

textbox.addEventListener("change", (event) => {
    let target = event.target;
    target.style.backgroundColor = /[^\d]/.test(target.value) ? "red" : "";
});
```

Обработчик события focus просто изменяет фоновый цвет текстового поля на желтый, чтобы сразу было ясно, что оно активно. Обработчики событий blur и change изменяют фоновый цвет поля на красный при обнаружении в нем нецифровых символов. Для проверки символов значение текстового поля сопоставляется с простым регулярным выражением. Проверка выполняется в обоих обработчиках событий, чтобы поведение текстового поля оставалось согласованным независимо от изменений его значения.

> **ПРИМЕЧАНИЕ** Отношения между событиями blur и change не формализованы. В некоторых браузерах событие blur генерируется раньше, чем change, в других — наоборот, так что при их обработке будьте внимательны.

РАБОТА С ТЕКСТОВЫМИ ПОЛЯМИ

Однострочное текстовое поле представляется HTML-элементом <input>, а многострочное — элементом <textarea>. Эти два элемента управления очень похожи и в большинстве случаев работают одинаково, но между ними есть важные различия.

По умолчанию элемент `<input>` отображает текстовое поле, даже если его атрибут `type` опущен (по умолчанию он имеет значение `"text"`). Атрибут `size` определяет ширину текстового поля в видимых символах. Атрибут `value` указывает первоначальное значение текстового поля, а атрибут `maxlength` — максимально допустимое количество символов в поле. Например, следующий код создает текстовое поле, которое может показывать 25 символов, но поддерживает значения длиной до 50 символов:

```
<input type="text" size="25" maxlength="50" value="initial value">
```

Элемент `<textarea>` всегда выводит на экран многострочное текстовое поле. Для указания его размеров можно использовать атрибуты `rows` и `cols`, которые определяют соответственно высоту и ширину текстового поля в символах. В отличие от элемента `<input>`, первоначальное значение `<textarea>` должно быть указано между тегами `<textarea>` и `</textarea>`:

```
<textarea rows="25" cols="5">initial value</textarea>
```

Кроме того, для элемента `<textarea>` нельзя ограничить количество символов в HTML-коде.

Несмотря на различия разметки, содержимое текстовых полей обоих типов хранится в свойстве `value`, которое можно использовать для чтения и задания значения текстового поля, например:

```
let textbox = document.forms[0].elements["textbox1"];
console.log(textbox.value);

textbox.value = "Some new value";
```

Читать и задавать значения текстовых полей с помощью свойства `value` предпочтительнее, чем использовать для этого стандартные DOM-методы. Например, не следует задействовать метод `setAttribute()` для установки атрибута `value` элемента `<input>` или изменять первый дочерний узел элемента `<textarea>`. Изменения свойства `value` также не всегда отражаются в DOM-структуре, так что для работы со значениями текстовых полей DOM-методы лучше не использовать.

Выделение текста

У текстовых полей обоих типов есть метод `select()`, который выделяет все содержимое поля. Большинство браузеров автоматически назначают фокус текстовому полю при вызове его метода `select()` (Opera не назначает). Этот метод не принимает аргументов и его можно вызвать в любое время, например:

```
let textbox = document.forms[0].elements["textbox1"];
textbox.select();
```

Разработчики часто выделяют весь текст в текстовом поле, когда оно получает фокус, особенно если у него есть значение, предлагаемое по умолчанию. Идея в том, что так пользователь может сразу удалить весь текст, если нужно ввести другое значение. Вот как это можно сделать:

```
textbox.addEventListener("focus", (event) => {
    event.target.select();
});
```

Этот код выделяет весь текст в текстовом поле, как только оно получает фокус. Это делает работу с формами гораздо удобнее.

Событие select

Методу `select()` сопутствует событие `select`, которое генерируется при выделении текста в текстовом поле. Точный момент возникновения этого события зависит от браузера. В Internet Explorer 9+, Opera, Firefox, Chrome и Safari оно генерируется, когда пользователь завершает выделение текста, а в Internet Explorer 8 и более ранних версиях — при выделении одной буквы. Событие `select` также возникает при вызове метода `select()`. Вот простой пример:

```
let textbox = document.forms[0].elements["textbox1"];

textbox.addEventListener("select", (event) => {
    console.log('Text selected: ${textbox.value}');
});
```

Получение выделенного текста

Событие `select` информирует о выделении текста, но не сообщает, какой текст выделен, поэтому в HTML5 для получения выделенного текста к текстовым полям были добавлены свойства `selectionStart` и `selectionEnd`. Они содержат отсчитываемые от нуля числа, указывающие границы выделенного текста (смещения его начала и конца). Для получения текста, выделенного в текстовом поле, можно использовать следующий код:

```
function getSelectedText(textbox) {
    return textbox.value.substring(textbox.selectionStart,
                                   textbox.selectionEnd);
}
```

Поскольку метод `substring()` работает со смещениями строк, значения `selectionStart` и `selectionEnd` можно передавать в него напрямую.

Это решение работает в Internet Explorer 9+, Firefox, Safari, Chrome и Opera. В Internet Explorer 8 и более ранних версиях эти свойства не поддерживаются, так что нужен другой подход.

В старых версиях Internet Explorer доступен объект `document.selection`, который содержит сведения о выделении текста во всем документе, при этом неизвестно, где на странице находится этот текст. Однако при использовании этого объекта вместе с событием `select` выделенный текст наверняка находится в текстовом поле, которое сгенерировало событие. Чтобы получить выделенный текст, нужно сначала создать диапазон, а затем извлечь из него текст:

```
function getSelectedText(textbox) {
    if (typeof textbox.selectionStart == "number") {
```

```
        return textbox.value.substring(textbox.selectionStart,
                                       textbox.selectionEnd);
    } else if (document.selection) {
        return document.selection.createRange().text;
    }
}
```

Новая версия функции возвращает выделенный текст независимо от браузера. Заметьте, что для использования объекта `document.selection` аргумент `textbox` не требуется.

Частичное выделение текста

В HTML5 разрешается выделять фрагменты текстовых полей. Для этого используется метод `setSelectionRange()`, первоначально реализованный в Firefox, а теперь доступный для всех текстовых полей в дополнение к методу `select()`. Как и метод `substring()` строки, он принимает два аргумента: индекс первого выделяемого символа и индекс конца выделения, например:

```
textbox.value = "Hello world!"

// выделение всего текста
textbox.setSelectionRange(0, textbox.value.length);    // "Hello world!"

// выделение трех первых символов
textbox.setSelectionRange(0, 3);    // "Hel"

//выделение символов с 4 по 6
textbox.setSelectionRange(4, 7);    // "o w"
```

Чтобы показать выделение, нужно назначить фокус текстовому полю или непосредственно перед вызовом метода `setSelectionRange()` или после него. Этот подход работает в Internet Explorer 9, Firefox, Safari, Chrome и Opera.

В Internet Explorer 8 и более ранних версий частично выделять текст можно с помощью диапазонов. Чтобы выделить часть текста в текстовом поле, нужно сначала создать диапазон методом `createTextRange()`, который в Internet Explorer доступен для текстовых полей, свернуть диапазон к началу поля с помощью метода `collapse()`, а затем воспользоваться методами `moveStart()` и `moveEnd()` для настройки границ диапазона. Метод `moveStart()` перемещает начальную и конечную точки диапазона в одно место, а метод `moveEnd()` задает общее количество выделяемых символов. Наконец, для выделения текста нужно вызывать метод `select()` диапазона:

```
textbox.value = "Hello world!";

var range = textbox.createTextRange();

// выделение всего текста
range.collapse(true);
range.moveStart("character", 0);
range.moveEnd("character", textbox.value.length);    // "Hello world!"
range.select();
```

```
// выделение первых трех символов
range.collapse(true);
range.moveStart("character", 0);
range.moveEnd("character", 3);
range.select();    // "Hel"

// выделение символов с 4 по 6
range.collapse(true);
range.moveStart("character", 4);
range.moveEnd("character", 3);
range.select();    // "o w"
```

Как и в других браузерах, чтобы выделение было видно, фокус должен принадлежать текстовому полю.

Частичное выделение текста полезно при реализации полей для ввода текста с расширенными возможностями, например полей с автозавершением ввода.

Фильтрация ввода

Часто требуется, чтобы текстовые поля принимали только данные определенного типа или в определенном формате, например данные должны содержать какие-то символы или соответствовать некоторому шаблону. По умолчанию возможности проверки контента текстовых полей весьма ограничены, поэтому для *фильтрации ввода* (input filtering) необходимо задействовать JavaScript. Используя события и другие возможности DOM, можно превратить обычное текстовое поле в элемент, по-настоящему «понимающий» свои данные.

Блокировка символов

Некоторые типы значений никогда не могут содержать определенные символы или наоборот, требуют их наличия. Скажем, текстовое поле для ввода номера телефона должно принимать только цифры. Для вставки символов в текстовое поле используется событие `keypress`, отменив для которого действие, предлагаемое по умолчанию, можно заблокировать ввод символов. Например, следующий код блокирует нажатия всех клавиш:

```
textbox.addEventListener("keypress", (event) => {
    event.preventDefault();
});
```

По сути, этот код делает текстовое поле доступным только для чтения, блокируя нажатия всех клавиш. Можно также блокировать только определенные символы по их кодам. Например, следующий код блокирует все символы, кроме цифр:

```
textbox.addEventListener("keypress", (event) => {
    if (!/\d/.test(String.fromCharCode(event.charCode))){
        event.preventDefault();
    }
})
```

В этом примере метод `String.fromCharCode()` преобразует код символа в строку, которая сопоставляется с регулярным выражением `/\d/`, определяющим все цифры. Если символ не является цифрой, событие блокируется методом `EventUtil.preventDefault()`.

Хотя событие `keypress` должно генерироваться только при нажатии клавиши ввода символа, некоторые браузеры генерируют его и для других клавиш. Firefox и Safari (до версии 3.1) генерируют событие `keypress` для клавиш со стрелками, а также клавиш Backspace и Delete, а Safari 3.1 и более поздних версий — нет. Это означает, что просто заблокировать все нецифровые клавиши нельзя, потому что иначе будут заблокированы и эти полезные клавиши. К счастью, можно легко определить, когда нажата одна из них. В Firefox все несимвольные клавиши, которые генерируют событие `keypress`, имеют код 0, тогда как Safari до версии 3 назначает всем им код 8. В общем, чтобы не блокировать никакие коды символов, меньшие 10, функцию можно переписать следующим образом:

```
textbox.addEventListener("keypress", (event) => {
    if (!/\d/.test(String.fromCharCode(event.charCode)) &&
        event.charCode > 9){
        event.preventDefault();
    }
});
```

Этот обработчик правильно работает во всех браузерах, блокируя нецифровые символы, но разрешая нажатия всех основных клавиш, которые также генерируют событие `keypress`.

Нам осталось решить проблему с копированием, вставкой и любыми другими действиями с нажатием клавиши Ctrl. Во всех браузерах, кроме Internet Explorer, предыдущий код блокирует сочетания Ctrl+C, Ctrl+V и любые другие сочетания с клавишей Ctrl, поэтому перед блокировкой мы должны убедиться в том, что она не нажата:

```
textbox.addEventListener("keypress", (event) => {
    if (!/\d/.test(String.fromCharCode(event.charCode)) &&
        event.charCode > 9 &&
        !event.ctrlKey){
        event.preventDefault();
    }
});
```

Теперь все варианты поведения, доступные для текстовых полей по умолчанию, будут работать. Слегка изменив этот код, можно разрешить или запретить ввод любых символов в текстовом поле.

Работа с буфером обмена

Поддержка событий, связанных с буфером обмена, и поддержка доступа к нему из JS-кода впервые была реализована в Internet Explorer. Эта реализация стала стандартом де-факто, на основе которого в Safari 2, Chrome и Firefox 3 были реализованы

похожие возможности. Еще позже события буфера обмена были добавлены в спецификацию HTML5. Этих событий шесть:

- `beforecopy` — генерируется непосредственно перед копированием;
- `copy` — генерируется при копировании;
- `beforecut` — генерируется непосредственно перед вырезанием;
- `cut` — генерируется при вырезании;
- `beforepaste` — генерируется непосредственно перед вставкой;
- `paste` — генерируется при вставке.

Поскольку это довольно новый стандарт доступа к буферу обмена, соответствующие события и объекты работают по-разному в зависимости от браузера. В Safari, Chrome и Firefox события `beforecopy`, `beforecut` и `beforepaste` генерируются только при выводе контекстного меню для текстового поля (в предвидении возможного события буфера обмена), а Internet Explorer генерирует их также непосредственно перед событиями `copy`, `cut` и `paste`. События `copy`, `cut` и `paste` возникают во всех браузерах без сюрпризов: при выборе соответствующих команд в контекстном меню и нажатии требуемых сочетаний клавиш.

События `beforecopy`, `beforecut` и `beforepaste` позволяют изменить данные, отправляемые в буфер обмена или получаемые из него, до фактического выполнения операции. Однако отмена одного из этих событий не отменяет операцию с буфером обмена — для этого нужно отменить событие `copy`, `cut` или `paste`.

Данные в буфере обмена доступны через объект `clipboardData`, который принадлежит либо объекту `window` (в Internet Explorer), либо объекту `event` (в Firefox 4+, Safari и Chrome). В Firefox, Safari и Chrome объект `clipboardData` доступен только при обработке событий буфера обмена, что предотвращает несанкционированный доступ к буферу обмена; в Internet Explorer он доступен все время. Чтобы код поддерживали все браузеры, лучше использовать этот объект только при обработке событий буфера обмена.

У объекта `clipboardData` есть три метода: `getData()`, `setData()` и `clearData()`. Метод `getData()` получает строковые данные из буфера обмена, принимая в качестве аргумента формат этих данных. Internet Explorer поддерживает два формата: `"text"` и `"URL"`. Firefox, Safari и Chrome вместо этого используют MIME-типы, но поддерживают и значение `"text"` как эквивалент `"text/plain"`.

Метод `setData()` принимает тип данных и текст, который нужно поместить в буфер обмена. В этом случае Internet Explorer также поддерживает типы `"text"` и `"URL"`, а Safari и Chrome ожидают MIME-тип, но, в отличие от метода `getData()`, они не распознают тип `"text"` и игнорируют вызов `setData()` с таким аргументом. Чтобы нивелировать особенности браузеров, можно добавить следующие кроссбраузерные методы:

```
function getClipboardText(event) {
    var clipboardData = (event.clipboardData || window.clipboardData);
    return clipboardData.getData("text");
}
```

```
function setClipboardText (event, value) {
    if (event.clipboardData) {
        return event.clipboardData.setData("text/plain", value);
    } else if (window.clipboardData){
        return window.clipboardData.setData("text", value);
    }
}
```

Метод `getClipboardText()` сравнительно прост. Он лишь выясняет способ доступа к объекту `clipboardData`, а затем вызывает его метод `getData()` с аргументом `"text"`. Второй метод, `setClipboardText()`, чуть сложнее. Определив способ доступа к объекту `clipboardData`, он вызывает метод `setData()` с типом данных, зависящим от реализации (`"text/plain"` для Firefox, Safari и Chrome; `"text"` для Internet Explorer).

Чтение текста из буфера обмена полезно, если текстовое поле принимает только определенные символы или текст в определенном формате. Например, если текстовое поле принимает только числа, желательно проверять также значения, вставляемые в него из буфера обмена. В обработчике события `paste` можно выяснить, соответствует ли требованиям содержимое буфера обмена, и если нет, отменить действие, предлагаемое по умолчанию:

```
textbox.addEventListener("paste", (event) => {
    let text = getClipboardText(event);

    if (!/^\d*$/.test(text)){
        event.preventDefault();
    }
});
```

Этот обработчик события `paste` позволяет вставлять в текстовое поле только числовые значения. Если содержимое буфера обмена не соответствует шаблону, операция отменяется. В Firefox, Safari и Chrome доступ к методу `getData()` возможен только в обработчике события `paste`.

Поскольку не все браузеры поддерживают доступ к буферу обмена, часто проще заблокировать одну или несколько операций с ним. В браузерах, которые поддерживают события `copy`, `cut` и `paste` (Internet Explorer, Safari, Chrome и Firefox), отменить для них действие, предлагаемое по умолчанию, легко. В Opera для этого нужно заблокировать соответствующие нажатия клавиш и запретить вывод контекстного меню.

Автоматический переход по нажатию клавиши табуляции

JavaScript предоставляет несколько способов упростить работу с полями форм. Один из наиболее популярных — это автоматическое перемещение фокуса к следующему полю после заполнения текущего. Так часто делают, если длина данных известна, например, если поле служит для ввода номера телефона. В США номера телефонов обычно состоят из трех частей: кода области, номера АТС и номера абонента. На веб-страницах для их ввода часто предлагается три поля:

```
<input type="text" name="tel1" id="txtTel1" maxlength="3">
<input type="text" name="tel2" id="txtTel2" maxlength="3">
<input type="text" name="tel3" id="txtTel3" maxlength="4">
```

Чтобы упростить и ускорить ввод этих данных, можно автоматически перемещать фокус к следующему полю, как только в текущее введено максимальное количество знаков. Иначе говоря, после ввода трех символов в первое поле можно назначить фокус второму, а после ввода очередных трех символов — третьему. Это можно сделать следующим образом:

```
<script>
    function tabForward(event){
        let target = event.target;

        if (target.value.length == target.maxLength){
            let form = target.form;

            for (let i = 0, len = form.elements.length; i < len; i++) {
                if (form.elements[i] == target) {
                    if (form.elements[i+1]) {
                        form.elements[i+1].focus();
                    }
                    return;
                }
            }
        }
    }

    let inputIds = ["txtTel1", "txtTel2", "txtTel3"];

    for (let id of inputIds) {
        let textbox = document.getElementById(id);
        textbox.addEventListener("keyup", tabForward);
    }

    let textbox1 = document.getElementById("txtTel1");
    let textbox2 = document.getElementById("txtTel2");
    let textbox3 = document.getElementById("txtTel3");
</script>
```

Главную роль в этом фрагменте играет функция `tabForward()`. Она проверяет, достигнута ли максимальная длина содержимого текстового поля, сравнивая его текущую длину с атрибутом `maxlength`. Если они равны (а больше символов браузер ввести не позволит), функция находит в коллекции элементов текущее текстовое поле и назначает фокус следующему элементу. Далее эта функция назначается каждому текстовому полю как обработчик события `keyup`. Поскольку оно генерируется после вставки в текстовое поле каждого нового символа, это идеальная возможность для проверки длины содержимого поля. В результате при заполнении этой простой формы пользователю никогда не придется нажимать клавишу табуляции для перемещения между полями и отправки данных формы.

Имейте в виду, что этот код специфичен для приведенной разметки и не учитывает возможное наличие скрытых полей.

API проверки ограничений в HTML5

HTML5 предлагает механизм проверки данных формы перед их отправкой серверу. Это позволяет выполнять несложную проверку данных, даже если JS-сценарии недоступны или не могут быть загружены. Браузер сам проверяет данные на основе правил, а затем показывает соответствующие сообщения об ошибках (без дополнительного JS-кода). Это возможно только в браузерах, которые поддерживают данную часть спецификации HTML5, то есть во всех современных браузерах (кроме Safari) и IE 10+.

Браузер автоматически проверяет только те поля формы, для которых заданы ограничения в HTML-разметке.

Обязательные поля

Первое ограничение, которое мы обсудим, задается с помощью атрибута `required`, например:

```
<input type="text" name="username" required>
```

Поле формы, помеченное атрибутом `required` как обязательное, не может быть пустым, иначе отправить данные формы не удастся. Этот атрибут можно задать для полей `<input>`, `<textarea>` и `<select>` (в Opera до версии 11 включительно его не поддерживает поле `<select>`). В JS-коде можно проверить, обязательно ли поле формы, используя свойство `required` элемента:

```
let isUsernameRequired = document.forms[0].elements["username"].required;
```

Можно также проверить, поддерживает ли браузер атрибут `required`:

```
let isRequiredSupported = "required" in document.createElement("input");
```

Этот код проверяет, есть ли у нового элемента `<input>` свойство `required`, используя простое распознавание возможностей.

При отправке формы с пустым обязательным полем браузеры ведут себя по-разному. Firefox, Chrome, IE и Opera блокируют отправку данных формы и выводят ниже поля всплывающую подсказку, а Safari ничего не делает и отправляет данные формы, как ни в чем не бывало.

Альтернативные типы ввода

В HTML5 определено несколько новых значений атрибута `type` элемента `<input>`. Они не только предоставляют дополнительные сведения о типе ожидаемых данных, но и по умолчанию обеспечивают некоторую проверку. Лучше всего поддерживаются типы `"email"` и `"url"`, каждый из которых применяет к данным специальные правила проверки:

```
<input type="email" name="email">
<input type="url" name="homepage">
```

Тип "email" проверяет введенный текст на соответствие шаблону адреса электронной почты, а тип "url" сопоставляет его с шаблоном URL-адреса. Имейте в виду, что в браузерах, упомянутых ранее в этом разделе, сопоставлением с шаблоном работает не идеально, например "-@-" окажется допустимым адресом электронной почты. Производители браузеров все еще работают над решением этих проблем.

Чтобы узнать, поддерживает ли браузер эти новые типы, можно создать элемент на JS-коде, присвоить его свойству type значение "email" или "url", а затем прочитать его. Старые браузеры автоматически заменяют неизвестные им типы значением "text", а браузеры, поддерживающие новые типы, возвращают правильные значения, например:

```
let input = document.createElement("input");
input.type = "email";

let isEmailSupported = (input.type == "email");
```

Помните, что пустое поле также считается допустимым, если для него не задан атрибут required. Кроме того, указание специального типа ввода не мешает пользователю ввести недопустимое значение, а только включает некоторые правила проверки, применяемые по умолчанию.

Числовые диапазоны

Кроме "email" и "url", HTML5 определяет еще несколько новых типов элементов ввода: "number", "range", "datetime", "datetime-local", "date", "month", "week" и "time". Все они являются числовыми. Эти типы не поддерживаются современными браузерами, поэтому использовать их следует осторожно, пока производители браузеров работают над логикой функционала и взаимной совместимостью. Сведения в этом разделе скорее приведены с расчетом на будущее.

Для каждого из этих числовых типов можно задать атрибуты min (наименьшее возможное значение), max (наибольшее возможное значение) и step (разница между отдельными значениями от min до max). Например, следующий код принимает только числа от 0 до 100 с интервалом 5:

```
<input type="number" min="0" max="100" step="5" name="count">
```

Некоторые браузеры для этого кода выводят счетчик с кнопками вверх и вниз для изменения значения счетчика.

Каждому из атрибутов соответствуют свойства элемента, которые можно читать и изменять с помощью JS-кода. Кроме того, у них есть методы stepUp() и stepDown(), которые принимают число, вычитаемое из текущего значения или добавляемое к нему (по умолчанию оно увеличивается или уменьшается на единицу). Эти методы еще не реализованы в браузерах, но будут использоваться следующим образом:

```
input.stepUp();        // увеличение на 1
input.stepUp(5);       // увеличение на 5
input.stepDown();      // уменьшение на 1
input.stepDown(10);    // уменьшение на 10
```

Шаблоны ввода

В HTML5 для текстовых полей представлен атрибут `pattern`, указывающий регулярное выражение, с которым должно быть сопоставлено введенное значение. Например, следующее текстовое поле принимает только числа:

```
<input type="text" pattern="\d+" name="count">
```

Предполагается, что в начале и в конце шаблона есть знаки `^` и `$` соответственно. Это означает, что ввод должен точно соответствовать шаблону от начала до конца.

Как и в коде с альтернативными типами ввода, шаблон не мешает пользователю ввести недопустимый текст, а просто указывает браузеру, допустимо значение или нет. Прочитать шаблон можно с помощью свойства `pattern`:

```
let pattern = document.forms[0].elements["count"].pattern;
```

Проверить, поддерживает ли браузер атрибут `pattern`, можно следующим образом:

```
let isPatternSupported = "pattern" in document.createElement("input");
```

Проверка допустимости

С помощью метода `checkValidity()` можно проверить, допустимо ли значение конкретного поля формы. Он доступен для всех элементов и возвращает `true`, если значение поля допустимо, и `false` в противном случае. При проверке используются условия, описанные ранее в этом разделе, так что обязательное поле без значения или поле со значением, которое не соответствует шаблону, считаются недопустимыми, например:

```
if (document.forms[0].elements[0].checkValidity()) {
    // значение поля допустимо, продолжаем
} else {
    // поле недопустимо
}
```

Вызвав метод `checkValidity()` для самой формы, можно проверить все ее поля. Если все они допустимы, метод возвратит `true`, а если хотя бы одно из полей недопустимо, — `false`:

```
if(document.forms[0].checkValidity()) {
    // форма допустима, продолжаем
} else {
    // поле формы недопустимо
}
```

В то время как метод `checkValidity()` просто сообщает, допустимо ли значение поля, свойство `validity` указывает точную причину, почему оно допустимо или нет. У этого объекта есть следующие свойства логического типа:

- `customError` — `true`, если с помощью метода `setCustomValidity()` было задано пользовательское сообщение, иначе `false`;

- `patternMismatch` — `true`, если значение не соответствует заданному атрибуту `pattern`;
- `rangeOverflow` — `true`, если значение больше, чем значение `max`;
- `rangeUnderflow` — `true`, если значение меньше, чем значение `min`;
- `stepMisMatch` — `true`, если значение не соответствует атрибуту `step` с учетом значений `min` и `max`;
- `tooLong` — `true`, если значение содержит больше символов, чем допускает свойство `maxlength` (некоторые браузеры, такие как Firefox 4, автоматически ограничивают количество символов, так что это значение может всегда быть равно `false`);
- `typeMismatch` — `true`, если значение не соответствует требуемому формату `"email"` или `"url"`;
- `valid` — `true`, если все остальные свойства равны `false` (это то же значение, которое возвращает метод `checkValidity()`);
- `valueMissing` — `true`, если поле, отмеченное как обязательное, пусто.

С помощью свойства `validity` можно получить более конкретные сведения о том, что не так с формой, например:

```
if (input.validity && !input.validity.valid) {
    if (input.validity.valueMissing) {
        console.log("Please specify a value.")            // Укажите значение
    } else if (input.validity.typeMismatch) {
        console.log("Please enter an email address.");    // Укажите адрес
                                                          // электронной почты
    } else {
        console.log("Value is invalid.");                 // Значение недопустимо
    }
}
```

Отключение проверки

С помощью атрибута `novalidate` можно отключить для формы все виды проверки:

```
<form method="post" action="signup.php" novalidate>
    <!-- код элементов формы -->
</form>
```

Этот атрибут можно также получить или задать, используя свойство `noValidate`, которое равно `true`, если атрибут задан, и `false`, если он отсутствует:

```
document.forms[0].noValidate = true;    // отключение проверки
```

Если у формы несколько кнопок отправки, можно отключить проверку формы для конкретной кнопки, добавив к ней атрибут `formnovalidate`:

```
<form method="post" action="foo.php">
    <!-- код элементов формы -->
```

```
    <input type="submit" value="Regular Submit">
    <input type="submit" formnovalidate name="btnNoValidate"
           value="Non-validating Submit">
</form>
```

В этом примере форма проверяется при щелчке на первой кнопке отправки, но не второй. То же самое можно сделать и в JS-коде:

```
// отключение проверки
document.forms[0].elements["btnNoValidate"].formNoValidate = true;
```

РАБОТА СО СПИСКАМИ

Списки создают с помощью элементов `<select>` и `<option>`. Чтобы упростить работу с ними, тип `HTMLSelectElement` предоставляет следующие свойства и методы в дополнение к тем, которые доступны для всех полей форм:

- `add(новыйЭлемент, связанныйЭлемент)` — добавляет новый элемент (`<option>`) перед связанным элементом;
- `multiple` — логическое значение, указывающее, разрешен ли множественный выбор (эквивалент HTML-атрибута `multiple`);
- `options` — коллекция `HTMLCollection` элементов `<option>` в элементе управления;
- `remove(индекс)` — удаляет элемент в указанной позиции;
- `selectedIndex` — отсчитываемый от нуля индекс выбранного элемента или значение –1, если никакой элемент не выбран (если список поддерживает множественный выбор, это всегда первый выбранный элемент);
- `size` — количество видимых строк списка (эквивалент HTML-атрибута `size`).

Свойство `type` списка может иметь значение `"select-one"` или `"select-multiple"` в зависимости от того, задан ли атрибут `multiple`. Значение свойства `value` определяется для списка на основе выбранного в нем элемента по следующим правилам:

- если никакой элемент не выбран, значением списка является пустая строка;
- если выбран элемент, для которого задан атрибут `value`, значением списка является атрибут `value` выбранного элемента (это верно, даже если атрибут `value` является пустой строкой);
- если выбран элемент, для которого не задан атрибут `value`, значением списка является текст элемента;
- если выбрано несколько эдементов, значение списка определяется по первому выбранному элементу с использованием двух предыдущих правил.

Рассмотрим следующий список:

```
<select name="location" id="selLocation">
    <option value="Sunnyvale, CA">Sunnyvale</option>
```

```
    <option value="Los Angeles, CA">Los Angeles</option>
    <option value="Mountain View, CA">Mountain View</option>
    <option value="">China</option>
    <option>Australia</option>
</select>
```

Если выбрать первый элемент этого списка, значением списка будет `"Sunnyvale, CA"`. Если выбрать элемент с текстом `"China"`, значением списка будет пустая строка, потому что атрибут `value` пуст. Наконец, если выбрать последний элемент, значением списка будет `"Australia"`, потому что у этого элемента `<option>` нет атрибута `value`.

Каждому элементу `<option>` соответствует DOM-объект `HTMLOptionElement`, который предоставляет следующие дополнительные свойства, позволяющий упростить доступ к данным:

- `index` — индекс элемента в коллекции `options`;
- `label` — надпись элемента (эквивалент HTML-атрибута `label`);
- `selected` — логическое значение, указывающее, выбран ли элемент (чтобы выбрать элемент, присвойте этому свойству значение `true`);
- `text` — текст элемента;
- `value` — значение элемента (эквивалент HTML-атрибута `value`).

Большинство свойств элемента `<option>` просто ускоряют доступ к данным элементов списка. Вместо них можно использовать и обычные возможности DOM, но это менее эффективно, например:

```
let selectbox = document.forms[0].elements["location"];

// не рекомендуется
let text = selectbox.options[0].firstChild.nodeValue;      // текст элемента
let value = selectbox.options[0].getAttribute("value");    // значение элемента
```

Этот код получает текст и значение первого элемента списка с помощью стандартных средств DOM. Сравните его с использованием специальных свойств:

```
let selectbox = document.forms[0].elements["location"];

// предпочтительный вариант
let text = selectbox.options[0].text;      // текст элемента
let value = selectbox.options[0].value;    // значение элемента
```

При работе с элементами списков лучше использовать специальные свойства, которые хорошо поддерживаются всеми браузерами, тогда как взаимодействия элементов управления форм при манипулировании DOM-узлами могут зависеть от браузера. Изменять текст или значения элементов `<option>` с помощью стандартных средств модели DOM не рекомендуется.

Использование события `change` со списками также имеет особенности. В отличие от других полей форм, которые генерируют его после изменения значения и утраты фокуса, для списка оно генерируется при выборе одного из его элементов.

ПРИМЕЧАНИЕ Значение, возвращаемое свойством `value`, зависит от браузера. Во всех браузерах свойство `value` всегда равно атрибуту `value`. Если он не указан, Internet Explorer 8 и более ранних версий возвращают пустую строку, тогда как Internet Explorer 9+, Safari, Firefox, Chrome и Opera возвращают значение свойства `text`.

Выбор элементов списка

Самый простой способ получить выбранный элемент списка с одиночным выбором — воспользоваться свойством `selectedIndex`, например:

```
let selectedOption = selectbox.options[selectbox.selectedIndex];
```

Этот код можно задействовать для вывода всех сведений о выбранном элементе:

```
let selectedIndex = selectbox.selectedIndex;
let selectedOption = selectbox.options[selectedIndex];
console.log('Selected index: $[selectedIndex}\n' +
            'Selected text: ${selectedOption.text}\n' +
            'Selected value: ${selectedOption.value}');
```

Этот код выводит в консоль сообщение с индексом, текстом и значением выбранного элемента.

Если список поддерживает множественный выбор, свойство `selectedIndex` все равно работает так, как если бы можно было выбрать только один элемент. При установке свойства `selectedIndex` прежний выбор отменяется и выбирается единственный указанный элемент, а при чтении этого свойства возвращается индекс только первого выбранного элемента.

Элемент также можно выбрать, получив ссылку на него и присвоив его свойству `selected` значение `true`. Например, следующий код выбирает первый элемент списка:

```
selectbox.options[0].selected = true;
```

В отличие от `selectedIndex`, установка свойства `selected` для элемента не отменяет выбор других элементов списка с множественным выбором, что позволяет динамически выбирать любое количество элементов. При изменении свойства `selected` для элемента списка с одиночным выбором выбор другого элемента отменяется. Присвоение значения `false` свойству `selected` не влияет на список с одиночным выбором.

С помощью свойства `selected` можно узнать, какие элементы списка выбраны. Чтобы получить все выбранные элементы, можно перебрать набор элементов в цикле, проверяя их свойство `selected`:

```
function getSelectedOptions(selectbox) {
    let result = new Array();

    for (let option of selectbox.options) {
        if (option.selected) {
```

```
            result.push(option);
        }
    }

    return result;
}
```

Эта функция возвращает массив элементов, выбранных в переданном ей списке. Сначала она создает массив для хранения результатов, а затем в цикле `for` перебирает все элементы списка, проверяя у каждого из них свойство `selected`. Если элемент выбран, он добавляется в массив `result`, который в конце возвращается из функции. Использовать ее для получения сведений о выбранных элементах можно следующим образом:

```
let selectbox = document.getElementById("selLocation");
let selectedOptions = getSelectedOptions(selectbox);
let message = "";

for (let option of selectedOptions) {
    message += 'Selected index: ${option.index}\n' +
               'Selected text: ${option.text}\n' +
               'Selected value: ${option.value}\n'
}

console.log(message);
```

Этот пример составляет в цикле `for` сообщение со сведениями о выбранных элементах, а затем выводит его на экран. Сообщение включает индекс, текст и значение каждого выбранного элемента. Этот код можно использовать со списками, поддерживающими как одиночный, так и множественный выбор.

Добавление элементов в список

JavaScript поддерживает несколько способов динамического создания элементов и их добавления в списки. Первый основан на использовании DOM:

```
let newOption = document.createElement("option");
newOption.appendChild(document.createTextNode("Option text"));
newOption.setAttribute("value", "Option value");

selectbox.appendChild(newOption);
```

Этот код создает элемент `<option>`, добавляет в него текстовый узел, задает атрибут `value` и присоединяет новый элемент к списку, при этом список сразу же обновляется.

Новые элементы также можно создавать с помощью конструктора `Option`, который использовался еще до появления DOM. Он принимает два аргумента, `text` и `value`, хотя второй аргумент не обязателен. Несмотря на то что этот конструктор создает экземпляр `Object`, браузеры, соответствующие DOM, возвращают элемент `<option>`. Это означает, что для добавления элемента в список можно использовать метод `appendChild()`, например:

```
let newOption = new Option("Option text", "Option value");
selectbox.appendChild(newOption);    // проблемы в IE до версии 8 включительно
```

Этот подход нормально работает во всех браузерах, кроме Internet Explorer 8 и более ранних версий, где текст нового элемента задается неправильно.

Другой способ добавить новый элемент в список — использовать метод `add()` списка. В спецификации DOM сказано, что этот метод принимает два аргумента: новый элемент и элемент, перед которым его нужно вставить. Чтобы добавить элемент в конец списка, нужно передать в качестве второго аргумента значение `null`. В Internet Explorer 8 и более ранних версий реализация метода `add()` отличается тем, что второй аргумент не обязателен и представляет индекс элемента, перед которым нужно вставить новый элемент. В браузерах, соответствующих модели DOM, второй аргумент обязателен, так что в кроссбраузерном коде нельзя просто использовать один аргумент (Internet Explorer 9 соответствует DOM). Если передать в качестве второго аргумента значение `undefined`, элемент будет без проблем добавлен в конец списка во всех браузерах, например:

```
let newOption = new Option("Option text", "Option value");
selectbox.add(newOption, undefined);    // оптимальное решение
```

Этот код работает надлежащим образом во всех версиях Internet Explorer и всех браузерах, соответствующих DOM. Если нужно вставить новый элемент не в конец списка, следует использовать DOM-подход с методом `insertBefore()`.

> **ПРИМЕЧАНИЕ** Как и в HTML, задавать значение элемента списка не требуется. Конструктор `Option` работает и с одним аргументом (текст элемента).

Удаление элементов списка

Удалить элемент списка можно несколькими способами. Во-первых, можно вызвать DOM-метод `removeChild()`, передав ему элемент, который нужно удалить:

```
selectbox.removeChild(selectbox.options[0]);    // удаление первого элемента
```

Во-вторых, можно использовать метод `remove()` списка, который принимает индекс удаляемого элемента:

```
selectbox.remove(0);    // удаление первого элемента
```

В-третьих, можно просто присвоить элементу значение `null`. Этот способ использовался еще до появления DOM:

```
selectbox.options[0] = null;    // удаление первого элемента
```

Чтобы полностью очистить список, нужно перебрать все элементы и удалить каждый из них:

```
function clearSelectbox(selectbox) {
    for(let option of selectbox.options) {
        selectbox.remove(0);
    }
}
```

Эта функция просто циклически удаляет первый элемент списка. Так как при этом все остальные элементы автоматически смещаются на одну позицию, в итоге получается пустой список.

Перемещение и переупорядочение элементов списка

До появления DOM переместить элемент из одного списка в другой было непросто. Для этого требовалось удалить элемент из первого списка, создать новый одноименный элемент с таким же значением и добавить его во второй список. В DOM для перемещения элемента в другой список достаточно воспользоваться методом `appendChild()`, который удаляет полученный элемент из родительского элемента и помещает его в конец другого элемента. Например, следующий код перемещает первый элемент исходного списка в конец второго списка:

```
let selectbox1 = document.getElementById("selLocations1");
let selectbox2 = document.getElementById("selLocations2");

selectbox2.appendChild(selectbox1.options[0]);
```

Перемещение элементов похоже на удаление в том смысле, что свойство `index` каждого элемента при этом сбрасывается.

Переупорядочение элементов выполняется очень похоже, и DOM-методы идеально подходят для этого. Чтобы переместить элемент в конкретное место списка, лучше всего использовать метод `insertBefore()`, хотя переместить элемент в последнюю позицию можно также с помощью метода `appendChild()`. Поднять элемент списка на одну позицию можно следующим образом:

```
let optionToMove = selectbox.options[1];
selectbox.insertBefore(optionToMove,
                       selectbox.options[optionToMove.index-1]);
```

Этот код выбирает элемент, который нужно переместить, и вставляет его перед предыдущим элементом. Вторую инструкцию можно использовать с любым элементом списка, кроме первого. Переместить элемент на одну позицию вниз можно аналогичным образом:

```
let optionToMove = selectbox.options[1];
selectbox.insertBefore(optionToMove,
                       selectbox.options[optionToMove.index+2]);
```

Этот код работает со всеми элементами списка, включая последний.

СЕРИАЛИЗАЦИЯ ФОРМ

С появлением Ajax (см. главу 21) одним из стандартных требований к веб-приложениям стала поддержка *сериализации форм* (form serialization). Сериализовать

форму в JavaScript можно, используя свойства `type`, `name` и `value` ее полей. При отправке данных формы серверу браузер соблюдает определенные правила.

- Имена и значения полей кодируются в формате URL, а их пары разделяются амперсандами.
- Отключенные поля не отправляются.
- Флажки и переключатели отправляются, только если они заданы.
- Кнопки типов `"reset"` и `"button"` не отправляются.
- У полей с множественным выбором отправляется запись для каждого выбранного значения.
- Кнопка отправки отправляется только в том случае, если она была использована для отправки данных формы. Любые элементы `<input>` типа `"image"` обрабатываются так же, как кнопки отправки.
- Значением элемента `<select>` является атрибут `value` выбранного элемента `<option>`. Если у элемента `<option>` нет атрибута `value`, значением является текст элемента `<option>`.

В сериализованное представление формы обычно не включаются никакие поля типа `button`, потому что итоговая строка, вероятно, будет отправлена другим способом. Все остальные правила должны быть соблюдены. Код сериализации формы таков:

```
function serialize(form) {
    let parts = [];
    let optValue;

    for (let field of form.elements) {
        switch(field.type) {
            case "select-one":
            case "select-multiple":

                if (field.name.length) {
                    for (let option of field.options) {
                        if (option.selected) {
                            optValue = "";
                            if (option.hasAttribute) {
                                optValue = (option.hasAttribute("value") ?
                                            option.value : option.text);
                            } else {
                                optValue =
                                   (option.attributes["value"].specified ?
                                    option.value : option.text);
                            }
                            parts.push(encodeURIComponent(field.name) +
                                       "=" + encodeURIComponent(optValue));
                        }
                    }
                }
                break;

            case undefined:        // коллекция полей
```

```
            case "file":            // поле добавления файлов
            case "submit":          // кнопка отправки
            case "reset":           // кнопка сброса
            case "button":          // пользовательская кнопка
                break;

            case "radio":           // переключатель
            case "checkbox":        // флажок
                if (!field.checked) {
                    break;
                }
            default:
                // поля формы без имен не сериализуются
                if (field.name.length) {
                    parts.push('${encodeURIComponent(field.name)}=' +
                            '${encodeURIComponent(field.value)}');
                }
        }
    }
    return parts.join("&");
}
```

Функция `serialize()` начинается с определения массива `parts` для хранения частей итоговой строки. Затем цикл `for` перебирает все поля формы, сохраняя их по очереди в переменной `field`. Как только получена ссылка на поле, его тип проверяется с помощью инструкции `switch`. Сложнее всего сериализовать элемент `<select>` с одиночным или множественным выбором. Для этого мы перебираем все элементы списка, добавляя в массив те из них, которые выделены. В списках с одиночным выбором может быть выбран только один элемент, а в списках с множественным выбором — сколько угодно (в том числе ни одного), но код подходит для списков обоих типов, потому что браузер сам контролирует количество выбранных элементов. Обнаружив выбранный элемент, мы должны выяснить, какое значение следует использовать. Если атрибут `value` отсутствует, нужно сериализовать текст элемента, хотя атрибут `value` с пустой строкой допустим. Чтобы проверить его наличие, мы используем метод `hasAttribute()` в браузерах, соответствующих DOM, и свойство `specified` атрибута в Internet Explorer 8 и более ранних версий.

Если форма содержит элемент `<fieldset>`, он доступен в коллекции элементов, но не имеет свойства `type`. Если свойство `type` равно `undefined`, сериализация не требуется. Это верно для всех типов кнопок и полей добавления файлов (поля добавления файлов содержат при отправке файла его содержимое, но их нельзя воспроизвести, так что они обычно опускаются при сериализации). У переключателей и флажков проверяется свойство `checked`, и, если оно имеет значение `false`, выполняется выход из инструкции `switch`. Если свойство `checked` имеет значение `true`, мы переходим к разделу, предлагаемому по умолчанию, который кодирует имя и значение поля и добавляет их в массив `parts`. Заметьте, что мы не сериализуем поля формы без имен, чтобы имитировать отправку данных формы браузером. В конце функции вызывается метод `join()` для составления строки с амперсандами между полями.

Функция `serialize()` возвращает строку в формате строки запроса, но ее можно легко адаптировать для сериализации формы в другом формате.

РЕДАКТИРОВАНИЕ ФОРМАТИРОВАННОГО ТЕКСТА

Одной из наиболее востребованных возможностей веб-приложений в свое время была возможность редактировать форматированный текст на веб-странице, что иногда называли WYSIWYG-редактированием (What You See Is What You Get — *что видишь, то и получаешь*). На основе этого функционала, представленного в Internet Explorer и теперь поддерживаемого в Opera, Safari, Chrome и Firefox, возник стандарт де-факто, который, однако, не описан ни в какой спецификации. Методика основана на добавлении в страницу встроенного фрейма (`iframe`), содержащего пустой HTML-файл. С помощью свойства `designMode` можно включить редактирование этого пустого документа, то есть HTML-кода элемента `<body>` страницы. Свойство `designMode` может иметь значение `"off"` (по умолчанию) или `"on"`. Если оно равно `"on"`, можно редактировать весь документ так же, как в текстовом редакторе, используя сочетания клавиш, которые изменяют начертание на полужирное, курсивное и т. д.

В качестве источника встроенного фрейма можно использовать совсем простую пустую HTML-страницу, например:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Blank Page for Rich Text Editing</title>
    </head>
    <body>
    </body>
</html>
```

Эта страница загружается во встроенный фрейм как любая другая. Чтобы ее можно было редактировать, нужно присвоить свойству `designMode` значение `"on"`, но это возможно только после полной загрузки документа. Обработчик события `load` страницы-контейнера подходит как нельзя лучше:

```
<iframe name="richedit" style="height: 100px; width: 100px">
</iframe>

<script>
window.addEventListener("load", () => {
    frames["richedit"].document.designMode = "on";
});
</script>
```

Как только этот код загрузится, вы увидите некоторое подобие текстового поля. Оно по умолчанию будет иметь такой же стиль, что и любая веб-страница, хотя его можно изменить, применив к пустой странице CSS-стиль.

Атрибут contenteditable

Другой способ работы с форматированным текстом, также впервые реализованный в Internet Explorer, включает использование специального атрибута `contenteditable`.

Применив его к любому элементу на странице, можно разрешить редактирование этого элемента пользователем. Многие предпочитают этот подход предыдущему, потому что он не требует использования встроенного фрейма, пустой страницы и JS-кода. Вместо этого можно просто добавить атрибут к элементу:

```
<div class="editable" id="richedit" contenteditable></div>
```

Любой текст, уже содержащийся в элементе, при этом автоматически становится редактируемым подобно элементу `<textarea>`. Включать и отключать режим редактирования можно динамически, используя свойство `contentEditable` элемента:

```
let div = document.getElementById("richedit");
richedit.contentEditable = "true";
```

Свойство `contentEditable` поддерживает три значения: `"true"` включает режим редактирования, `"false"` отключает, а `"inherit"` указывает, что нужно наследовать параметр родительского элемента (требуется потому, что в элементе со свойством `contenteditable` можно создавать другие элементы и уничтожать их). Атрибут `contentEditable` поддерживается в Internet Explorer, Firefox, Chrome, Safari, Opera и всех современных мобильных браузерах.

> **ПРИМЕЧАНИЕ** `contenteditable` – чрезвычайно универсальный атрибут. Например, можно преобразовать окно браузера в блокнот, посетив псевдо-URL `data:text/html, <html contenteditable>`. Это создает специальную модель DOM со всем документом, доступным для редактирования.

Работа с форматированным текстом

Основным механизмом взаимодействия с редактором форматированного текста является метод `document.execCommand()`, который выполняет для документа именованные команды, поддерживая большинство изменений формата. Метод `document.execCommand()` имеет три аргумента: имя команды, которую нужно выполнить, логическое значение, указывающее, должен ли браузер предоставить пользовательский интерфейс для команды, и значение, необходимое команде для работы (или `null`, если оно не требуется). Второй аргумент в кроссбраузерном коде всегда должен быть равен `false`, потому что Firefox генерирует ошибку, если он равен `true`.

Каждый браузер поддерживает свой набор команд. Команды с наиболее широкой поддержкой указаны в таблице.

КОМАНДА	ЗНАЧЕНИЕ (ТРЕТИЙ АРГУМЕНТ)	ОПИСАНИЕ
`backcolor`	Строка цвета	Задает фоновый цвет документа
`bold`	`null`	Включает и отключает полужирное начертание для выделенного текста

КОМАНДА	ЗНАЧЕНИЕ (ТРЕТИЙ АРГУМЕНТ)	ОПИСАНИЕ
`copy`	`null`	Копирует выделенный текст в буфер обмена
`createlink`	Строка URL-адреса	Преобразует выделенный текст в ссылку с указанным URL-адресом
`cut`	`null`	Вырезает выделенный текст в буфер обмена
`delete`	`null`	Удаляет выделенный текст
`fontname`	Название шрифта	Изменяет шрифт для выделенного текста
`fontsize`	Число от 1 до 7	Изменяет размер шрифта для выделенного текста
`forecolor`	Строка цвета	Изменяет цвет выделенного текста
`formatblock`	HTML-элемент, в который должен быть заключен блок, например `<h1>`	Форматирует все текстовое поле с выделенным текстом, используя указанный HTML-элемент
`indent`	`null`	Задает отступ для текста
`inserthorizontalrule`	`null`	Вставляет элемент `<hr>` в позиции курсора
`insertimage`	URL-адрес изображения	Вставляет изображение в позиции курсора
`insertorderedlist`	`null`	Вставляет элемент `<ol>` в позиции курсора
`insertparagraph`	`null`	Вставляет элемент `<p>` в позиции курсора
`insertunorderedlist`	`null`	Вставляет элемент `<ul>` в позиции курсора
`italic`	`null`	Включает и отключает курсивное начертание для выделенного текста
`justifycenter`	`null`	Центрирует блок текста, в котором находится курсор
`justifyleft`	`null`	Выравнивает по левому краю блок текста, в котором находится курсор
`outdent`	`null`	Отменяет отступ для текста
`paste`	`null`	Вставляет выделенный текст из буфера обмена

КОМАНДА	ЗНАЧЕНИЕ (ТРЕТИЙ АРГУМЕНТ)	ОПИСАНИЕ
`removeformat`	`null`	Отменяет форматирование блока, в котором находится курсор. Эта команда противоположна команде `formatblock`
`selectall`	`null`	Выделяет весь текст в документе
`underline`	`null`	Включает и отключает подчеркивание выделенного текста
`unlink`	`null`	Удаляет текстовую ссылку. Эта команда противоположна команде `createlink`

Реализация команд для работы с буфером обмена во многом зависит от браузера. Хотя не все эти команды доступны через метод `document.execCommand()`, соответствующие сочетания клавиш все же работают.

С помощью этих команд можно в любое время изменить вид форматированного текста во встроенном фрейме, например:

```
// переключение полужирного начертания текста во встроенном фрейме
frames["richedit"].document.execCommand("bold", false, null);

// переключение курсивного начертания текста во встроенном фрейме
frames["richedit"].document.execCommand("italic", false, null);

// создание ссылки на сайт www.wrox.com во встроенном фрейме
frames["richedit"].document.execCommand("createlink", false,
                                        "http://www.wrox.com");

// создание заголовка первого уровня во встроенном фрейме
frames["richedit"].document.execCommand("formatblock", false, "<h1>");
```

С помощью этих же команд можно форматировать текст в разделе `contenteditable` страницы, но тогда вместо встроенного фрейма нужно использовать объект `document` текущего окна:

```
// переключение полужирного начертания текста
document.execCommand("bold", false, null);

// переключение курсивного начертания текста
document.execCommand("italic", false, null);

// создание ссылки на сайт www.wrox.com
document.execCommand("createlink", false, "http://www.wrox.com");

// создание заголовка первого уровня
document.execCommand("formatblock", false, "<h1>");
```

Даже если команда поддерживается во всех браузерах, HTML-код, который она генерирует, может сильно различаться. Например, команда `bold` применяет к тексту

элемент в Internet Explorer и Opera, элемент <b> в Safari и Chrome, элемент в Firefox. Из-за различий в реализации команд и в преобразовании HTML-кода свойством innerHTML полагаться на единообразие HTML-кода, генерируемого редактором форматированного текста, не следует.

Есть несколько других методов, связанных с командами. Метод queryCommandEnabled() определяет, можно ли выполнить команду при текущем выделенном тексте или положении курсора. Он принимает имя проверяемой команды и возвращает true, если она допустима при текущем состоянии области редактирования, или false в противном случае, например:

```
let result = frames["richedit"].document.queryCommandEnabled("bold");
```

Если команду "bold" можно выполнить для текущего выделенного текста, этот код возвратит true. Метод queryCommandEnabled() не указывает, разрешено ли выполнение команды, а только сообщает, допустимо ли это при текущем выделенном тексте. Например, в Firefox вызов queryCommandEnabled("cut") возвращает true, хотя по умолчанию эта команда запрещена.

С помощью метода queryCommandState() можно узнать, была ли выполнена конкретная команда для выделенного текста. Например, следующий код определяет, является ли текущий выделенный текст полужирным:

```
let isBold = frames["richedit"].document.queryCommandState("bold");
```

Если команда "bold" была применена к выделенному тексту, этот код возвратит true. Так редакторы форматированного текста обновляют кнопки задания полужирного и курсивного начертания и т. д.

Наконец, метод queryCommandValue() возвращает значение, с которым была выполнена команда (третий аргумент метода execCommand). Например, для диапазона текста, к которому была применена команда "fontsize" со значением 7, следующий код возвращает "7":

```
let fontSize = frames["richedit"].document.queryCommandValue("fontsize");
```

С помощью этого метода можно определить, как команда была выполнена для выделенного текста, чтобы выяснить, допустима ли следующая команда.

Выделение форматированного текста

Получить текст, выделенный в редакторе форматированного текста, можно с помощью метода getSelection() встроенного фрейма. Этот метод, доступный для объектов document и window, возвращает объект Selection, у которого есть следующие свойства:

- anchorNode — узел, в котором начинается выделенная область;
- anchorOffset — количество знаков в узле anchorNode, пропущенных перед началом выделенной области;
- focusNode — узел, в котором завершается выделенная область;

- `focusOffset` — количество выделенных символов в узле `focusNode`;
- `isCollapsed` — логическое значение, указывающее, совпадают ли начало и конец выделенной области;
- `rangeCount` — количество DOM-диапазонов в выделенной области.

Свойства объекта `Selection` не особо полезны. К счастью, у него также доступны следующие методы, позволяющие получить дополнительные сведения о выделенной области и изменить ее:

- `addRange(`*диапазон*`)` — добавляет указанный DOM-диапазон в выделенную область;
- `collapse(`*узел*, *смещение*`)` — свертывает выделенную область к указанному смещению в указанном узле;
- `collapseToEnd()` — свертывает выделенную область к ее концу;
- `collapseToStart()` — свертывает выделенную область к ее началу;
- `containsNode(`*узел*`)` — определяет, содержится ли указанный узел в выделенной области;
- `deleteFromDocument()` — удаляет выделенный текст из документа (то же самое, что и вызов `execCommand("delete", false, null)`);
- `extend(`*узел*, *смещение*`)` — расширяет выделенную область, перемещая `focusNode` и `focusOffset` к указанным значениям;
- `getRangeAt(`*индекс*`)` — возвращает DOM-диапазон по указанному индексу в выделенной области;
- `removeAllRanges()` — удаляет все DOM-диапазоны из выделенной области (по сути, выделенная область при этом удаляется, потому что она должна содержать хотя бы один диапазон);
- `removeRange(`*диапазон*`)` — удаляет указанный DOM-диапазон из выделенной области;
- `selectAllChildren(`*узел*`)` — отменяет выделение и выделяет все дочерние узлы указанного узла;
- `toString()` — возвращает текстовое содержимое выделенной области.

Методы объекта `Selection` очень эффективны и широко используют DOM-диапазоны для управления выделением (см. главу 12). Доступ к DOM-диапазонам обеспечивает даже больший контроль над форматированием текста, чем метод `execCommand()`, позволяя работать непосредственно DOM-структурой выделенного текста, например:

```
let selection = frames["richedit"].getSelection();

// получение выделенного текста
let selectedText = selection.toString();

// получение диапазона, представляющего выделенную область
let range = selection.getRangeAt(0);
```

```
// закрашивание фона выделенного текста
let span = frames["richedit"].document.createElement("span");
span.style.backgroundColor = "yellow";
range.surroundContents(span);
```

Этот код закрашивает фон выделенного текста желтым цветом. Для этого метод `surroundContents()` заключает DOM-диапазон выделенной области в элемент `<span>` с желтым фоном.

Метод `getSelection()` определен в HTML5 и реализован в Internet Explorer 9 и во всех современных версиях Firefox, Safari, Chrome и Opera.

Internet Explorer 8 и более ранних версий не поддерживают DOM-диапазоны, но позволяют взаимодействовать с выделенным текстом с помощью фирменного объекта `selection`, являющегося свойством объекта `document`. Чтобы получить текст, выделенный в редакторе форматированного текста, нужно сначала создать текстовый диапазон (см. главу 12), а затем прочитать его свойство `text`:

```
let range = frames["richedit"].document.selection.createRange();
let selectedText = range.text;
```

Манипулировать HTML-кодом с помощью текстовых диапазонов в Internet Explorer не так безопасно, как использовать DOM-диапазоны, хотя и возможно. Например, для закрашивания фона выделенной области как в предыдущем фрагменте можно использовать свойство `htmlText` в сочетании с методом `pasteHTML()`:

```
let range = frames["richedit"].document.selection.createRange();
range.pasteHTML(
    '<span style="background-color:yellow">${range.htmlText}</span>');
```

Этот код получает HTML-код текущей выделенной области с помощью свойства `htmlText`, а затем заключает его в элемент `<span>` и вставляет обратно методом `pasteHTML()`.

Форматированный текст в формах

Поскольку редактирование форматированного текста выполняется с помощью встроенного фрейма или элемента `contenteditable`, а не элемента управления формы, редактор форматированного текста технически не является частью формы. Это означает, что для отправки HTML-кода серверу нужно извлечь его вручную и отправить самостоятельно. Для этого обычно используют скрытое поле формы, обновляя его HTML-кодом из встроенного фрейма или элемента `contenteditable`. Непосредственно перед отправкой формы HTML-код извлекают из встроенного фрейма или элемента и вставляют в скрытое поле. Например, если используется встроенный фрейм, в обработчике события `submit` формы можно сделать следующее:

```
form.addEventListener("submit", (event) => {
    let target = event.target;

    target.elements["comments"].value =
    frames["richedit"].document.body.innerHTML;
});
```

Здесь мы получаем HTML-код из встроенного фрейма с помощью свойства `innerHTML` тела документа и вставляем его в поле формы с именем `"comments"`, заполняя поле непосредственно перед отправкой данных формы. Не забудьте сделать это, если вы отправляете форму вручную методом `submit()`. С элементом `contenteditable` можно поступить аналогичным образом:

```
form.addEventListener("submit", (event) => {
    let target = event.target;

    target.elements["comments"].value =
        document.getElementById("richedit").innerHTML;
});
```

ИТОГИ

В отличие от HTML и веб-приложений, веб-формы за время их существования изменились незначительно. Используя в JS-коде свойства, методы и события форм и их полей, можно расширять возможности форм и делать работу с ними более удобной. Перечислим некоторые из концепций, рассмотренных в этой главе.

- Используя различные стандартные и нестандартные методы, можно выделять содержимое текстовых полей полностью или частично.
- Для взаимодействия с выделенным текстом во всех браузерах имеются стандартные средства, первоначально реализованные в Firefox.
- Прослушивая события клавиатуры и проверяя вводимые символы, можно блокировать добавление определенных символов в текстовое поле.

Все браузеры поддерживают события буфера обмена, включая `copy`, `cut` и `paste`, но реализации этих событий сильно различаются в зависимости от поставщика браузера.

Заблокировав событие `paste` для буфера обмена, можно предотвратить вставку определенных символов в текстовое поле.

Благодаря DOM управлять списками стало гораздо проще. Используя стандартные DOM-приемы, можно добавлять элементы в списки, удалять их, изменять их порядок и перемещать из одного списка в другой.

Для редактирования форматированного текста можно использовать встроенный фрейм с пустым HTML-документом. Чтобы включить для страницы такой же режим редактирования, как в текстовом редакторе, нужно присвоить свойству `designMode` документа значение `"on"` или задать для элемента атрибут `contenteditable`. По умолчанию во время редактирования можно переключать стили шрифтов, такие как полужирный и курсивный, работать с буфером обмена. Для доступа к некоторым из этих возможностей средствами JS-кода можно использовать метод `execCommand()`, а получить сведения о выделенном тексте можно с помощью методов `queryCommandEnabled()`, `queryCommandState()` и `queryCommandValue()`. Поле формы для редактора форматированного текста не создается, поэтому для отправки такого текста серверу нужно предварительно скопировать его HTML-код из встроенного фрейма или элемента `contenteditable` в поле формы.

20

API в JavaScript

- Atomics и SharedArrayBuffer
- Кроссконтекстная отправка сообщений
- Encoding API
- File и Blob API
- Перетаскивание
- Notifications API
- Page Visibility API
- Streams API
- Timing APIs
- Веб-компоненты
- Web Cryptography API

Растущая универсальность веб-браузеров сопровождается головокружительным ростом их сложности. Во многих отношениях современный веб-браузер стал перочинным ножом из различных API, подробно описанных в обширном наборе спецификаций. Эта экосистема спецификации браузера беспорядочна и нестабильна. Некоторые спецификации, такие как HTML5, представляют собой набор API и функций браузера, которые улучшают существующий стандарт. Другие спецификации определяют API для отдельной функции, такой как Web Cryptography API или Notifications API. В зависимости от браузера принятие этих более новых API может иногда быть частичным или вообще отсутствовать.

В конечном счете, решение использовать более новые API предполагает компромисс между поддержкой большего количества браузеров и предоставлением более современных функций. Некоторые API могут эмулироваться с использованием заменителей, но они могут часто снижать производительность или увеличивать полезную нагрузку JS сайта.

ПРИМЕЧАНИЕ Количество веб-API невероятно велико (https://developer.Mozilla.org/en-US/docs/Web/API). Эта глава посвящена API, которыми пользуется большинство разработчиков, которые поддерживаются несколькими поставщиками браузеров и не рассматриваются в других разделах этой книги.

ATOMICS И SHAREDARRAYBUFFER

Когда `SharedArrayBuffer` доступен из нескольких контекстов, может возникнуть состояние гонки, когда несколько операций над буфером выполняется одновременно. Atomics API позволяет нескольким контекстам безопасно читать и записывать данные в один `SharedArrayBuffer`, заставляя операции буфера выполняться только по одной за раз. Atomics API был определен в спецификации ES2017.

Вы заметите, что Atomics API во многом напоминает упрощенную архитектуру набора команд (ISA) — это не случайно. Природа атомарных операций исключает некоторые оптимизации, которые операционная система или компьютерное оборудование обычно выполняют автоматически (например, переупорядочение команд). Атомарные операции также делают невозможным одновременный доступ к памяти, что может замедлить выполнение программы при неправильном применении. Поэтому Atomics API был разработан для того, чтобы позволить сложным многопоточным программам JavaScript быть спроектированными из минимальной, но надежной коллекции атомарного поведения.

SharedArrayBuffer

`SharedArrayBuffer` имеет API-интерфейс, идентичный `ArrayBuffer`. Основное отличие состоит в том, что, хотя ссылка на `ArrayBuffer` должна передаваться между контекстами выполнения, ссылка на `SharedArrayBuffer` может использоваться одновременно любым числом контекстов выполнения.

Совместное использование памяти между несколькими контекстами выполнения означает, что становятся возможными параллельные операции потока. Традиционные операции JavaScript не защищают от состязаний, возникающих в результате одновременного доступа к памяти. В следующем примере демонстрируется состояние гонки между четырьмя выделенными рабочими потоками, обращающимися к одному `SharedArrayBuffer`:

```
const workerScript = `
self.onmessage = ({data}) => {
    const view = new Uint32Array(data);
```

```
    // Выполнение 1000000 операций добавления
    for (let i = 0; i < 1E6; ++i) {
        // Операции добавления, небезопасные для потоков, вызывают состояние гонки
        view[0] += 1;
    }

    self.postMessage(null);
};
`;

const workerScriptBlobUrl = URL.createObjectURL(new Blob([workerScript]));

// Создание пула потоков размером 4
const workers = [];
for (let i = 0; i < 4; ++i) {
    workers.push(new Worker(workerScriptBlobUrl));
}

// Запись конечного значения после завершения последнего рабочего потока
let responseCount = 0;
for (const worker of workers) {
    worker.onmessage = () => {
        if (++responseCount == workers.length) {
            console.log(`Final buffer value: ${view[0]}`);
        }
    };
}

// Инициализация SharedArrayBuffer
const sharedArrayBuffer = new SharedArrayBuffer(4);
const view = new Uint32Array(sharedArrayBuffer);
view[0] = 1;

// Send the SharedArrayBuffer to each worker
for (const worker of workers) {
    worker.postMessage(sharedArrayBuffer);
}

// (Ожидаемый результат — 4000001. Действительно выведется что-то вроде:)
// Final buffer value: 2145106
```

Для решения этой проблемы был введен Atomics API для обеспечения поточно-ориентированных операций JavaScript в `SharedArrayBuffer`.

> **ПРИМЕЧАНИЕ** `SharedArrayBuffer` API идентичен `ArrayBuffer` API, описанному в главе 6 «Ссылочные типы коллекций». Подробнее об использовании `SharedArrayBuffer` в нескольких контекстах см. в главе 27 «Рабочие потоки».

Основы использования Atomics

Объект `Atomics` существует во всех глобальных контекстах и предоставляет набор статических методов для выполнения поточно-ориентированных операций.

Большинство из этих методов принимают экземпляр `TypedArray` (ссылающийся на `SharedArrayBuffer`) в качестве первого аргумента, а соответствующие операнды — в качестве последующих аргументов.

Арифметика и побитовые методы в Atomics

Atomics API предлагает простой набор методов для выполнения модификации на месте. В спецификации ECMA эти методы определены как операции `AtomicReadModifyWrite`. В рамках каждого из этих методов выполняется чтение из местоположения в `SharedArrayBuffer`, арифметическая или побитовая операция и запись в то же местоположение. Атомарная природа этих операторов означает, что эти три операции будут выполняться последовательно и без прерывания другим потоком.

Все арифметические методы демонстрируются здесь:

```
// Создание буфера размером 1
let sharedArrayBuffer = new SharedArrayBuffer(1);

// Создание Uint8Array из буфера
let typedArray = new Uint8Array(sharedArrayBuffer);

// Все ArrayBuffer инициализируются со значением 0
console.log(typedArray);    // Uint8Array[0]

const index = 0;
const increment = 5;

// Atomic добавляет 5 к значению по индексу 0
Atomics.add(typedArray, index, increment);

console.log(typedArray);    // Uint8Array[5]

// Atomic вычитает 5 из значения по индексу 0
Atomics.sub(typedArray, index, increment);
console.log(typedArray);    // Uint8Array[0]
```

Все побитовые методы демонстрируются здесь:

```
// Создание буфера размером 1
let sharedArrayBuffer = new SharedArrayBuffer(1);

// Создание Uint8Array из буфера
let typedArray = new Uint8Array(sharedArrayBuffer);

// Все ArrayBuffer инициализируются со значением 0
console.log(typedArray);    // Uint8Array[0]

const index = 0;

// Atomic ИЛИ с 0b1111 для значения по индексу 0
Atomics.or(typedArray, index, 0b1111);

console.log(typedArray);    // Uint8Array[15]
```

```
// Atomic И с 0b1100 для значения по индексу 0
Atomics.and(typedArray, index, 0b1100);

console.log(typedArray);    // Uint8Array[12]

// Atomic исключающее ИЛИ с 0b1111 для значения по индексу 0
Atomics.xor(typedArray, index, 0b1111);

console.log(typedArray);    // Uint8Array[3]
```

Предыдущий пример небезопасного потока может быть исправлен следующим образом:

```
const workerScript = `
self.onmessage = ({data}) => {
    const view = new Uint32Array(data);

    // выполнение 1000000 операций добавления
    for (let i = 0; i < 1E6; ++i) {
        // Операция добавления, безопасная для потока
        Atomics.add(view, 0, 1);
    }

    self.postMessage(null);
};
`;

const workerScriptBlobUrl = URL.createObjectURL(new Blob([workerScript]));

// Создание пула потоков размером 4
const workers = [];
for (let i = 0; i < 4; ++i) {
    workers.push(new Worker(workerScriptBlobUrl));
}

// Запись конечного значения после завершения последнего рабочего потока
let responseCount = 0;
for (const worker of workers) {
    worker.onmessage = () => {
        if (++responseCount == workers.length) {
            console.log(`Final buffer value: ${view[0]}`);
        }
    };
}

// Инициализация SharedArrayBuffer
const sharedArrayBuffer = new SharedArrayBuffer(4);
const view = new Uint32Array(sharedArrayBuffer);
view[0] = 1;

// Отправка SharedArrayBuffer каждому рабочему потоку
for (const worker of workers) {
    worker.postMessage(sharedArrayBuffer);
}

// (Ожидаемый результат — 4000001)
// Final buffer value: 4000001
```

Атомарные чтение и запись

И компилятору JavaScript браузера, и самой архитектуре ЦПУ предоставляется разрешение на изменение порядка команд, если они обнаружат, что это увеличит общую пропускную способность выполнения программы. Обычно однопотоковая природа JavaScript означает, что эту оптимизацию следует приветствовать с распростертыми объятиями. Однако переупорядочение команд в нескольких потоках может привести к состоянию гонки, которое чрезвычайно сложно отладить.

Atomics API решает эту проблему двумя основными способами:

- Все инструкции Atomics никогда не переупорядочиваются друг относительно друга.
- Использование атомарного чтения или записи гарантирует, что все инструкции (как атомарные, так и неатомарные) никогда не будут переупорядочены относительно этого атомарного чтения/записи. Это означает, что все инструкции перед атомным чтением/записью завершатся до того, как произойдет атомное чтение/запись, и все инструкции после них не начнутся, пока атомное чтение/запись не завершится.

В дополнение к чтению и записи значений в буфер, `Atomics.load()` и `Atomics.store()` ведут себя как «ограждения кода». Механизм JavaScript гарантирует, что, хотя неатомарные инструкции могут быть *локально* переупорядочены относительно `load()` или `store()`, переупорядочение никогда не нарушит границы чтения и записи Atomic. Следующий код комментирует это поведение:

```
const sharedArrayBuffer = new SharedArrayBuffer(4);
const view = new Uint32Array(sharedArrayBuffer);

// Выполнение неатомарной записи
view[0] = 1;

// Неатомарная запись гарантированно выполнится до этого чтения,
// поэтому гарантированно будет прочитан 1
console.log(Atomics.load(view, 0)); // 1

// Атомарная запись
Atomics.store(view, 0, 2);

// Неатомарное чтение гарантированно выполнится после атомарной записи,
// поэтому гарантированно будет прочитан 2
console.log(view[0]); // 2
```

Атомарный обмен

Atomics API предлагает два типа методов, которые гарантируют последовательное и непрерывное чтение-запись: `exchange()` и `compareExchange()`. `Atomics.exchange()` выполняет простой обмен, гарантируя, что никакие другие потоки не прервут обмен значениями:

```
const sharedArrayBuffer = new SharedArrayBuffer(4);
const view = new Uint32Array(sharedArrayBuffer);

// Запись 3 по индексу 0
Atomics.store(view, 0, 3);

// Чтение значения по индексу 0 и запись 4 по индексу 0
console.log(Atomics.exchange(view, 0, 4));    // 3

// Чтение значения по индексу 0
console.log(Atomics.load(view, 0));    // 4
```

Один поток в многопоточной программе может захотеть выполнить запись в общий буфер, *только* если другой поток не изменил определенное значение с момента его последнего чтения. Если значение не изменилось, он может безопасно записать обновленное значение. Если значение *изменилось*, выполнение записи уничтожит значение, вычисленное другим потоком. Для этой задачи в Atomics API есть метод `compareExchange()`. Он выполняет запись, только если значение по заданному индексу соответствует ожидаемому. Рассмотрим следующий пример:

```
const sharedArrayBuffer = new SharedArrayBuffer(4);
const view = new Uint32Array(sharedArrayBuffer);

// Запись 5 по индексу 0
Atomics.store(view, 0, 5);
// Чтение значения из буфера
let initial = Atomics.load(view, 0);

// выполнение неатомарной операции с этим значением
let result = initial ** 2;

// Запись этого значения обратно в буфер, только если буфер не был изменен
Atomics.compareExchange(view, 0, initial, result);

// Проверка успешности записи
console.log(Atomics.load(view, 0));    // 25
```

Если значение не совпадает, вызов `compareExchange()` будет просто проходным:

```
const sharedArrayBuffer = new SharedArrayBuffer(4);
const view = new Uint32Array(sharedArrayBuffer);

// Запись 5 по индексу 0
Atomics.store(view, 0, 5);

// Чтение значения из буфера
let initial = Atomics.load(view, 0);

// выполнение неатомарной операции с этим значением
let result = initial ** 2;

// Запись этого значения обратно в буфер, только если буфер не был изменен
Atomics.compareExchange(view, 0, -1, result);

// Проверка ошибки при записи
console.log(Atomics.load(view, 0));    // 5
```

Операции и блокировки futex в Atomics

Многопоточные программы не будут иметь большого значения без какой-либо конструкции блокировки. Чтобы удовлетворить эту потребность, Atomics API предлагает несколько методов, смоделированных на Linux futex (упаковка *быстрого мьютекса пространства пользователя*). Методы довольно просты, но они предназначены для использования в качестве основных строительных блоков для более сложных конструкций блокировки.

> **ПРИМЕЧАНИЕ** Все futex операции в Atomics работают только с представлением `Int32Array`. Кроме того, они могут использоваться только внутри рабочих потоков.

То, как работают `Atomics.wait()` и `Atomics.notify()`, лучше всего понять на примере. Следующий примитивный пример создает четыре рабочих потока для работы с `Int32Array` длины 1. Потоки по очереди получают блокировку и выполняют операцию добавления:

```
const workerScript = `
self.onmessage = ({data}) => {
    const view = new Int32Array(data);

    console.log('Waiting to obtain lock');
    // Остановка при обнаружении начального значения, тайм-аут на 10000 мс
    Atomics.wait(view, 0, 0, 1E5);

    console.log('Obtained lock');

    // Добавление 1 к индексу данных
    Atomics.add(view, 0, 1);

    console.log('Releasing lock');

    // Разрешение продолжить работу только одному потоку
    Atomics.notify(view, 0, 1);

    self.postMessage(null);
};
`;

const workerScriptBlobUrl = URL.createObjectURL(new Blob([workerScript]));

const workers = [];
for (let i = 0; i < 4; ++i) {
    workers.push(new Worker(workerScriptBlobUrl));
}

// Запись конечного значения после завершения работы последним потоком
let responseCount = 0;
for (const worker of workers) {
    worker.onmessage = () => {
        if (++responseCount == workers.length) {
            console.log(`Final buffer value: ${view[0]}`);
        }
```

```
    };
}
// Инициализация SharedArrayBuffer
const sharedArrayBuffer = new SharedArrayBuffer(8);
const view = new Int32Array(sharedArrayBuffer);

// Отправка SharedArrayBuffer каждому рабочему потоку
for (const worker of workers) {
    worker.postMessage(sharedArrayBuffer);
}

// Выполнение первой блокировки через 1000 мс
setTimeout(() => Atomics.notify(view, 0, 1), 1000);

// Waiting to obtain lock
// Waiting to obtain lock
// Waiting to obtain lock
// Waiting to obtain lock
// Obtained lock
// Releasing lock
// Obtained lock
// Releasing lock
// Obtained lock
// Releasing lock
// Obtained lock
// Releasing lock
// Final buffer value: 4
```

Поскольку `SharedArrayBuffer` инициализируется со значением 0, каждый рабочий поток прибудет в `Atomics.wait()` и остановит выполнение. В остановленном состоянии поток выполнения будет находиться в очереди ожидания, оставаясь приостановленным до истечения указанного времени ожидания или до вызова `Atomics.notify()` для этого индекса. Спустя 1000 миллисекунд контекст исполнения верхнего уровня вызовет `Atomics.notify()`, чтобы освободить ровно один из ожидающих потоков. Этот поток завершит выполнение и снова вызовет `Atomics.notify()`, освобождая еще один поток. Это продолжается до тех пор, пока все потоки не завершат выполнение и не передадут свой последний `postMessage()`.

В Atomics API также есть метод `Atomics.isLockFree()`. Почти наверняка вам никогда не понадобится его использовать, так как он предназначен для высокопроизводительных алгоритмов, чтобы решить, нужно ли получение блокировки или нет. Спецификация предлагает такое описание:

> `Atomics.isLockFree()` *является примитивом оптимизации. Интуиция подсказывает, что если атомарный шаг атомарного примитива (*`compareExchange`, `load`, `store`, `add`, `sub`, `and`, `or`, `xor` *или* `exchange`*) для данных размером n байтов будет выполнен без вызова вызывающим агентом внешней блокировки n байтов, составляющих элемент данных, тогда* `Atomics.isLockFree(n)` *вернет* `true`. *Высокопроизводительные алгоритмы будут использовать* `Atomics.isLockFree`, *чтобы определить, использовать ли блокировки или атомарные операции в критических секциях. Если атомарный примитив не свободен от блокировки, то для алгоритма часто является более эффективным обеспечение собственной блокировки.*

`Atomics.isLockFree(4)` *всегда возвращает* `true`*, поскольку это может поддерживаться на всех известных соответствующих аппаратных средствах. Можно предположить, что это в целом упростит программы.*

КРОСС-КОНТЕКСТНЫЙ ОБМЕН СООБЩЕНИЯМИ

Кросс-документный обмен сообщениями (cross-document messaging), иногда сокращенно называемый XDM, — это возможность передавать информацию между различными контекстами выполнения, такими как рабочие потоки или страницы из разных источников. Например, страница на www.wrox.com хочет связаться со страницей из p2p.wrox.com, которая располагается в фрейме. До появления XDM безопасное взаимодействие с ним требовало много работы. XDM формализует эту функциональность безопасным и простым в использовании способом.

> **ПРИМЕЧАНИЕ** Кросс-контекстный обмен сообщениями используется для связи между окнами и общения с рабочими потоками. Этот раздел посвящен использованию `postMessage()` для взаимодействия с другими окнами. Для получения информации о сообщениях рабочих потоков, `MessageChannel` и `BroadcastChannel`, см. главу 27 «Рабочие потоки».

В основе XDM лежит метод `postMessage()`. Это имя метода используется во многих частях HTML5 в дополнение к XDM и всегда используется для одной и той же цели: для передачи данных из одного места в другое.

Метод `postMessage()` принимает три аргумента: сообщение, строку, указывающую на предполагаемое происхождение получателя, и необязательный массив переносимых объектов (относится только к рабочим потокам). Второй аргумент очень важен по соображениям безопасности и ограничивает места, куда браузер может доставить сообщение. Рассмотрим этот пример:

```
let iframeWindow = document.getElementById("myframe").contentWindow;
iframeWindow.postMessage("A secret", "http://www.wrox.com");
```

Последняя строка пытается отправить сообщение в фрейм и указывает, что источником должно быть `http://www.wrox.com`. Если источник совпадает, то сообщение будет доставлено в фрейм; в противном случае `postMessage()` ничего не делает. Это ограничение защищает информацию в случае изменения местоположения окна без вашего ведома. Можно разрешить отправку сообщений любому источнику, передав `*` в качестве второго аргумента функции `postMessage()`, но это не рекомендуется.

Событие `message` запускается в `window` при получении сообщения XDM. Это сообщение запускается асинхронно, поэтому может иметь место задержка между временем отправки сообщения и моментом запуска события `message` в принимающем окне. Объект события, который передается обработчику события `onmessage`, содержит три важных элемента информации:

- `data` — строковые данные, которые были переданы в качестве первого аргумента в `postMessage()`.

- `origin` — источник документа, отправившего сообщение, например `http://www.wrox.com`.
- `source` — прокси для объекта `window` документа, который отправил сообщение. Этот прокси-объект используется главным образом для выполнения метода `postMessage()` в окне, которое отправило последнее сообщение. Если окно отправки имеет тот же источник, это может быть фактический объект окна.

При получении сообщения очень важно проверить происхождение окна отправки. Точно так же, как указание второго аргумента для `postMessage()` гарантирует, что данные не будут непреднамеренно переданы на неизвестную страницу, проверка источника во время `onmessage` гарантирует, что передаваемые данные поступают из правильного места. Основной шаблон выглядит следующим образом:

```
window.addEventListener("message", (event) => {
    // убедитесь, что отправитель – тот же, что ожидался
    if (event.origin == "http://www.wrox.com") {
        // обработайте информацию
        processMessage(event.data);
        // необязательно: отправьте письмо назад в начальное окно
        event.source.postMessage("Received!", "http://p2p.wrox.com");
    }
});
```

Имейте в виду, что в большинстве случаев `event.source` является прокси для `window`, а не фактическим объектом `window`, поэтому нельзя получить доступ ко всей информации об окне. Лучше всего использовать `postMessage()`, который всегда присутствует и всегда может быть вызван.

Есть несколько странностей, связанных с XDM. Во-первых, первый аргумент `postMessage()` изначально был реализован как строка. Его определение было изменено, чтобы разрешить передачу любых структурированных данных; однако не все браузеры реализовали это изменение. По этой причине лучше всегда передавать строку, используя `postMessage()`. Если нужно передать структурированные данные, то лучший способ — вызвать `JSON.stringify()` для данных, передав строку в `postMessage()`, а затем вызвать `JSON.parse()` в обработчике события `onmessage`.

XDM чрезвычайно полезен при попытке поместить содержимое в песочницу с помощью фрейма в другой домен. Этот подход часто используется в гибридных приложениях и приложениях социальных сетей. Содержащая страница способна защитить себя от вредоносного контента, передавая информацию только во встроенный фрейм через XDM. XDM также можно использовать со страницами из того же домена.

ENCODING API

Encoding API позволяет конвертировать строки и типизированные массивы. Спецификация представляет четыре глобальных класса для выполнения этих преобразований: `TextEncoder`, `TextEncoderStream`, `TextDecoder` и `TextDecoderStream`.

ПРИМЕЧАНИЕ Поддержка потокового кодирования/декодирования намного уже, чем массового кодирования/декодирования.

Кодировка текста

EncodingAPI предоставляет два способа преобразования строки в двоичный эквивалент ее типизированного массива: *массовое* кодирование и *потоковое* кодирование. При переходе от строки к типизированному массиву кодировщик всегда будет использовать UTF-8.

Массовое кодирование

Обозначение «*массовое*» означает, что механизм JavaScript будет синхронно кодировать всю строку. Для очень длинных строк это может быть затратной операцией. Массовое кодирование выполняется с использованием экземпляра `TextEncoder`:

```
const textEncoder = new TextEncoder();
```

Этот экземпляр предоставляет метод `encode()`, который принимает строку и возвращает кодировку UTF-8 каждого символа в только что созданном `Uint8Array`:

```
const textEncoder = new TextEncoder();
const decodedText = 'foo';
const encodedText = textEncoder.encode(decodedText);

// f, закодированный в utf-8, — это 0x66 (102 в десятичной системе)
// o, закодированный в utf-8, — это 0x6F (111 в десятичной системе)
console.log(encodedText);    // Uint8Array(3) [102, 111, 111]
```

Кодировщик может обрабатывать символы, которые будут принимать несколько индексов в конечном массиве, таких как эмодзи:

```
const textEncoder = new TextEncoder();
const decodedText = ' ';
const encodedText = textEncoder.encode(decodedText);
// ☺, закодированный в UTF-8 — это 0xF0 0x9F 0x98 0x8A
// (240, 159, 152, 138 в десятичной системе)
console.log(encodedText);    // Uint8Array(4) [240, 159, 152, 138]
```

Экземпляр также предоставляет метод `encodeInto()`, который принимает строку *и* целевой `Uint8Array`. Этот метод возвращает словарь, содержащий свойства `read` и `written`, указывающие, сколько символов было успешно прочитано из исходной строки и записано в целевой массив соответственно. Если в типизированном массиве недостаточно места, кодирование прекратится и словарь укажет этот результат:

```
const textEncoder = new TextEncoder();
const fooArr = new Uint8Array(3);
const barArr = new Uint8Array(2);
```

```
const fooResult = textEncoder.encodeInto('foo', fooArr);
const barResult = textEncoder.encodeInto('bar', barArr);

console.log(fooArr);       // Uint8Array(3) [102, 111, 111]
console.log(fooResult);    // { read: 3, written: 3 }

console.log(barArr);       // Uint8Array(2) [98, 97]
console.log(barResult);    // { read: 2, written: 2 }
```

`encode()` должен выделить новый `Uint8Array`, тогда как `encodeInto()` — нет. Для чувствительных к производительности приложений это различие может иметь значительные последствия.

> **ПРИМЕЧАНИЕ** Кодировка текста всегда будет использовать формат UTF-8 и должна записываться в экземпляр `Uint8Array`. Попытка использовать другой типизированный массив при вызове `encodeInto()` приведет к ошибке.

Потоковое кодирование

`TextEncoderStream` — это просто `TextEncoder` в форме `TransformStream`. Передача потока декодированного текста через кодировщик потока вернет поток кодированных фрагментов текста:

```
async function* chars() {
    const decodedText = 'foo';
    for (let char of decodedText) {
        yield await new Promise((resolve) => setTimeout(resolve, 1000, char));
    }
}

const decodedTextStream = new ReadableStream({
    async start(controller) {
        for await (let chunk of chars()) {
            controller.enqueue(chunk);
        }

        controller.close();
    }
});

const encodedTextStream = decodedTextStream.pipeThrough(new TextEncoderStream());

const readableStreamDefaultReader = encodedTextStream.getReader();

(async function() {
    while(true) {
        const { done, value } = await readableStreamDefaultReader.read();

        if (done) {
            break;
        } else {
            console.log(value);
        }
    }
})();
```

```
// Uint8Array[102]
// Uint8Array[111]
// Uint8Array[111]
```

Декодирование текста

EncodingAPI предоставляет два способа преобразования типизированного массива в его строковый эквивалент: *массовое* декодирование и *потоковое* декодирование. В отличие от классов кодировщика, при переходе от типизированного массива к строке декодер поддерживает большое количество строковых кодировок, перечисленных здесь: https://encoding.spec.whatwg.org/#names-and-labels.

Кодировка символов по умолчанию — UTF-8.

Массовое декодирование

Обозначение «*массовое*» означает, что механизм JavaScript будет синхронно декодировать всю строку. Для очень длинных строк это может быть затратной операцией. Массовое декодирование выполняется с использованием экземпляра `DecoderEncoder`:

```
const textDecoder = new TextDecoder();
```

Этот экземпляр предоставляет метод `decode()`, который принимает типизированный массив и возвращает декодированную строку:

```
const textDecoder = new TextDecoder();

// f, закодированный в utf-8, — это 0x66 (102 в десятичной системе)
// o, закодированный в utf-8, - это 0x6F (111 в десятичной системе)
const encodedText = Uint8Array.of(102, 111, 111);
const decodedText = textDecoder.decode(encodedText);

console.log(decodedText);    // foo
```

Декодеру все равно, какой типизированный массив он передает, поэтому он покорно декодирует все двоичное представление. В этом примере 32-битные значения, содержащие только 8-битные символы, декодируются как UTF-8, получая дополнительные пустые символы:

```
const textDecoder = new TextDecoder();

// f, закодированный в utf-8, — это 0x66 (102 в десятичной системе)
// o, закодированный в utf-8, — это 0x6F (111 в десятичной системе)
const encodedText = Uint32Array.of(102, 111, 111);
const decodedText = textDecoder.decode(encodedText);

console.log(decodedText);    // "f o o "
```

Декодер оборудован для обработки символов, которые охватывают несколько индексов в набираемом массиве, таких как эмодзи:

```
const textDecoder = new TextDecoder();

// ☺, закодированный в UTF-8, — это 0xF0 0x9F 0x98 0x8A
```

```
// (240, 159, 152, 138 в десятичной системе)
const encodedText = Uint8Array.of(240, 159, 152, 138);
const decodedText = textDecoder.decode(encodedText);

console.log(decodedText); // ☺
```

В отличие от `TextEncoder`, `TextDecoder` совместим с большим числом символьных кодировок. Взгляните на следующий пример, где вместо стандартной UTF-8 используется UTF-16:

```
const textDecoder = new TextDecoder('utf-16');

// f, закодированный в utf-8 — это 0x0066 (102 в десятичной системе)
// o, закодированный в utf-8 — это 0x006F (111 в десятичной системе)
const encodedText = Uint16Array.of(102, 111, 111);
const decodedText = textDecoder.decode(encodedText);

console.log(decodedText); // foo
```

Потоковое декодирование

`TextDecoderStream` — это просто `TextDecoder` в форме `TransformStream`. Передача потока закодированного текста через декодер потока вернет поток фрагментов декодированного текста:

```
async function* chars() {
    // Каждый фрагмент должен существовать как типизированный массив
    const encodedText = [102, 111, 111].map((x) => Uint8Array.of(x));
    for (let char of encodedText) {
        yield await new Promise((resolve) => setTimeout(resolve, 1000, char));
    }
}

const encodedTextStream = new ReadableStream({
    async start(controller) {
        for await (let chunk of chars()) {
            controller.enqueue(chunk);
        }

        controller.close();
    }
});

const decodedTextStream = encodedTextStream.pipeThrough(new TextDecoderStream());

const readableStreamDefaultReader = decodedTextStream.getReader();

(async function() {
    while(true) {
        const { done, value } = await readableStreamDefaultReader.read();

        if (done) {
            break;
        } else {
            console.log(value);
        }
    }
})();
```

```
// f
// o
// o
```

Потоки текстового декодера неявно понимают, что суррогатные пары могут быть разделены между фрагментами. Поток декодера будет сохранять фрагментированный текст до тех пор, пока не будет сформирован полный символ. Рассмотрим следующий пример, где потоковый декодер будет ожидать прохождения всех четырех фрагментов, прежде чем декодированный поток выдаст один символ:

```
async function* chars() {
    // ☺, закодированный в UTF-8, — это 0xF0 0x9F 0x98 0x8A
    // (240, 159, 152, 138 в десятичной системе)
    const encodedText = [240, 159, 152, 138].map((x) => Uint8Array.of(x));

    for (let char of encodedText) {
        yield await new Promise((resolve) => setTimeout(resolve, 1000, char));
    }
}

const encodedTextStream = new ReadableStream({
    async start(controller) {
        for await (let chunk of chars()) {
            controller.enqueue(chunk);
        }

        controller.close();
    }
});

const decodedTextStream = encodedTextStream.pipeThrough(new TextDecoderStream());

const readableStreamDefaultReader = decodedTextStream.getReader();

(async function() {
    while(true) {
        const { done, value } = await readableStreamDefaultReader.read();

        if (done) {
            break;
        } else {
            console.log(value);
        }
    }
})();

// ☺
```

Потоки текстового декодера чаще всего используются вместе с `fetch()`, поскольку тело ответа может быть обработано как `ReadableStream`. Это можно сделать следующим образом:

```
const response = await fetch(url);
const stream = response.body.pipeThrough(new TextDecoderStream());

for await (let decodedChunk of decodedStream) {
    console.log(decodedChunk);
}
```

BLOB И FILE API

Одной из основных проблем веб-приложений является невозможность взаимодействия с файлами на компьютере пользователя. До 2000 г. единственным способом работы с файлами было помещение `<input type="file">` в форму — и на этом все. API Blob и File разработаны для того, чтобы предоставить веб-разработчикам доступ к файлам на компьютере клиента безопасным способом, обеспечивающим лучшее взаимодействие с этими файлами.

Тип File

File API по-прежнему основан на поле ввода файла формы, но добавляет возможность прямого доступа к информации о файле. HTML5 добавляет коллекцию `files` в DOM для элемента ввода файла. Если в поле выбран один или несколько файлов, коллекция `files` будет содержать последовательность объектов `File`, представляющих каждый файл. Каждый объект `File` имеет несколько свойств только для чтения, в том числе:

- `name` — имя файла в локальной системе.
- `size` — размер файла в байтах.
- `type` — строка, содержащая MIME-тип файла.
- `lastModifiedDate` — строка, представляющая время последнего изменения файла. Это свойство было реализовано только в Chrome.

Например, можно получить информацию о каждом выбранном файле, прослушивая событие `change` и затем просматривая коллекцию `files`:

```
let filesList = document.getElementById("files-list");
filesList.addEventListener("change", (event) => {
    let files = event.target.files,
    i = 0,
    len = files.length;

    while (i < len) {
        const f = files[i];
        console.log(`${f.name} (${f.type}, ${f.size} bytes)`);
        i++;
    }
});
```

Этот пример просто выводит информацию о каждом файле в консоль. Эта возможность сама по себе является большим шагом вперед для веб-приложений, но File API идет дальше, позволяя фактически читать данные из файлов через тип `FileReader`.

Тип FileReader

Тип `FileReader` представляет собой асинхронный механизм чтения файлов. Можно считать `FileReader` похожим на `XMLHttpRequest`, только он используется для чтения

файлов из файловой системы, а не для чтения данных с сервера. Тип `FileReader` предлагает несколько методов для чтения данных файла:

- `readAsText(file, encoding)` — считывает файл в виде простого текста и сохраняет текст в свойстве результата. Второй аргумент, тип кодировки, является необязательным.
- `readAsDataURL(file)` — читает файл и сохраняет URI данных, представляющий файлы в свойстве `result`.
- `readAsBinaryString(file)` — читает файл и сохраняет строку, где каждый символ представляет байт в свойстве `result`.
- `readAsArrayBuffer(file)` — читает файл и сохраняет `ArrayBuffer`, содержащий содержимое файла, в свойстве `result`.

Эти различные способы чтения файла обеспечивают максимальную гибкость при работе с данными файла. Например, можно прочитать изображение как URI данных, чтобы отобразить его обратно пользователю, или же можно прочитать файл в виде текста, чтобы проанализировать его.

Поскольку чтение происходит асинхронно, каждый `FileReader` публикует несколько событий. Три наиболее полезных события — это `progress`, `error` и `load`, которые указывают, когда доступно больше данных, когда произошла ошибка и когда файл полностью прочитан соответственно.

Событие `progress` срабатывает примерно каждые 50 миллисекунд и имеет ту же информацию, что и событие `progress` в XHR: `lengthComputable`, `loaded` и `total`. Кроме того, свойство `result` объекта `FileReader` доступно для чтения во время события `progress`, хотя оно может еще не содержать все данные.

Событие `error` срабатывает, если файл не может быть прочитан по какой-либо причине. Когда происходит событие `error`, заполняется свойство `error` объекта `FileReader`. Этот объект имеет единственное свойство `code`, которое представляет собой код ошибки 1 (файл не найден), 2 (ошибка безопасности), 3 (чтение было прервано) , 4 (файл не читается) или 5 (ошибка кодирования).

Событие `load` срабатывает, когда файл был успешно загружен; оно не сработает, если сработало событие `error`. Вот пример использования всех трех событий:

```
let filesList = document.getElementById("files-list");
filesList.addEventListener("change", (event) => {
    let info = "",
        output = document.getElementById("output"),
        progress = document.getElementById("progress"),
        files = event.target.files,
        type = "default",
        reader = new FileReader();

    if (/image/.test(files[0].type)) {
        reader.readAsDataURL(files[0]);
        type = "image";
    } else {
```

```
        reader.readAsText(files[0]);
        type = "text";
    }

    reader.onerror = function() {
        output.innerHTML = "Could not read file, error code is " +
        reader.error.code;
    };

    reader.onprogress = function(event) {
        if (event.lengthComputable) {
            progress.innerHTML = `${event.loaded}/${event.total}`;
        }
    };

    reader.onload = function() {
        let html = "";

        switch(type) {
            case "image":
                html = `<img src="${reader.result}">`;
                break;
            case "text":
                html = reader.result;
                break;
        }
        output.innerHTML = html;
    };
});
```

Этот код читает файл из поля формы и отображает его на странице. Если файл имеет тип MIME, указывающий, что это изображение, то запрашивается URI данных, и при загрузке этот URI вставляется в виде изображения на страницу. Если файл не является изображением, он читается как строка и выводится на страницу как есть. Событие `progress` используется для отслеживания и отображения байтов читаемых данных, а событие error отслеживает любые ошибки.

Можно остановить текущее чтение, вызвав метод `abort()`, и в этом случае вызовется событие `abort`. После срабатывания `load`, `error` или `abort` запускается событие с именем `loadend`. Событие `loadend` указывает, что все чтение завершено по любой из трех причин. Методы `readAsText()` и `readAsDataURL()` поддерживаются во всех реализующих их браузерах.

Тип FileReaderSync

Тип `FileReaderSync`, как следует из его названия, является *синхронной* (sync — синхронный) версией `FileReader`. Он использует те же методы, что и FileReader, но выполняет блокировку чтения файла, только продолжая выполнение после загрузки всего файла в память. `FileReaderSync` доступен только внутри рабочих потоков, поскольку крайне медленный процесс чтения всего файла никогда не будет практичным для использования в среде выполнения верхнего уровня.

Предположим, что рабочему потоку отправляется объект `File` через `postMessage()`. Следующий код указывает потоку синхронно прочитать весь файл в память и отправить обратно URL-адрес данных файла:

```
// worker.js

self.omessage = (messageEvent) => {
    const syncReader = new FileReaderSync();
    console.log(syncReader);    // FileReaderSync {}

    // Блокировка рабочего потока на время чтения файла
    const result = syncReader.readAsDataUrl(messageEvent.data);

    // Пример ответа для PDF-файла
    console.log(result);    // data:application/pdf;base64,JVBERi0xLjQK...

    // Отправка URL назад
    self.postMessage(result);
};
```

Blobs и частичное чтение

В некоторых случаях можно прочитать только части файла, а не весь файл. Для этого объект `File` предоставляет метод `slice()`. Он принимает два аргумента: начальный байт и количество байтов для чтения. Этот метод возвращает экземпляр `Blob`, который на самом деле является супертипом `File`.

«Blob», сокращение от «*binary large object*, большой двоичный объект», — это JavaScript-оболочка для неизменяемых двоичных данных. Blob-объекты могут быть созданы из массива, содержащего строки, `ArrayBuffers`, `ArrayBufferViews` или даже другие `Blob`-объекты. Для конструктора `Blob` необязательно может быть предоставлен тип MIME как часть его параметра `options`:

```
console.log(new Blob(['foo']));
// Blob {size: 3, type: ""}

console.log(new Blob(['{"a": "b"}'], { type: 'application/json' }));
// {size: 10, type: "application/json"}

console.log(new Blob(['<p>Foo</p>', '<p>Bar</p>'], { type: 'text/html' }));
// {size: 20, type: "text/html"}
```

`Blob` также имеет свойства `size` и `type` и метод `slice()` для дальнейшего сокращения данных. Также можно читать из `Blob` с помощью `FileReader`. Этот пример читает только первые 32 байта из файла:

```
let filesList = document.getElementById("files-list");
filesList.addEventListener("change", (event) => {
    let info = "",
        output = document.getElementById("output"),
        progress = document.getElementById("progress"),
        files = event.target.files,
        reader = new FileReader(),
```

```
        blob = blobSlice(files[0], 0, 32);

    if (blob) {
        reader.readAsText(blob);


        reader.onerror = function() {
            output.innerHTML = "Could not read file, error code is " +
            reader.error.code;
        };
        reader.onload = function() {
            output.innerHTML = reader.result;
        };
    } else {
        console.log("Your browser doesn't support slice().");
    }
});
```

Чтение только части файла может сэкономить время, особенно когда вы просто ищете определенный фрагмент данных, например заголовок файла.

URL объекта и Blob-объекты

URL-адреса объектов, также называемые URL-адресами *Blob-объектов*, представляют собой URL-адреса, которые ссылаются на данные, хранящиеся в `File` или `Blob`. Преимущество объектных URL заключается в том, что не нужно считывать содержимое файла в JavaScript для их использования. Вместо этого вы просто указываете URL объекта в соответствующем месте. Чтобы создать URL объекта, используйте метод `window.URL.createObjectURL()` и передайте объект `File` или `Blob`. Возвращаемое значение этой функции — строка, которая указывает на адрес памяти. Поскольку строка является URL, ее можно использовать в DOM. Например, следующий фрагмент кода отображает файл изображения на странице:

```
let filesList = document.getElementById("files-list");
filesList.addEventListener("change", (event) => {
    let info = "",
        output = document.getElementById("output"),
        progress = document.getElementById("progress"),
        files = event.target.files,
        reader = new FileReader(),
        url = window.URL.createObjectURL(files[0]);
    if (url) {
        if (/image/.test(files[0].type)) {
            output.innerHTML = `<img src="${url}">`;
        } else {
            output.innerHTML = "Not an image.";
        }
    } else {
        output.innerHTML = "Your browser doesn't support object URLs.";
    }
});
```

Благодаря передаче URL объекта непосредственно в тег `<img>` нет необходимости сначала читать данные в JavaScript. Вместо этого тег `<img>` идет прямо в область памяти и считывает данные на страницу.

Когда данные больше не нужны, лучше всего освободить память, связанную с ними. Память не может быть освобождена, пока используется URL объекта. Можно указать, что URL объекта больше не нужен, передав его в `window.URL.revokeObjectURL()`. Все URL объекта удаляются из памяти автоматически при выгрузке страницы. Тем не менее лучше всего освободить URL каждого объекта вручную, если он больше не нужен, чтобы обеспечить минимальное использование памяти на странице.

Перетаскивание файла чтения

Объединение HTML5 Drag-and-Drop API с File API позволяет создавать интересные интерфейсы для чтения информации о файле. После создания настраиваемой цели перетаскивания на странице можно перетаскивать файлы с рабочего стола в эту цель. При перетаскивании изображения или ссылки вызывается событие `drop`. Удаляемые файлы доступны в `event.dataTransfer.files`, который представляет собой список объектов `File`, аналогичных доступным в поле ввода файла.

В следующем примере выводится информация о файлах, которые отбрасываются на настраиваемую цель перетаскивания на странице:

```
let droptarget = document.getElementById("droptarget");
function handleEvent(event) {
    let info = "",
        output = document.getElementById("output"),
        files, i, len;
    event.preventDefault();

    if (event.type == "drop") {
        files = event.dataTransfer.files;
        i = 0;
        len = files.length;

        while (i < len) {
            info += `${files[i].name} (${files[i].type},
                     ${files[i].size} bytes)<br>`;
            i++;
        }

        output.innerHTML = info;
    }
}
droptarget.addEventListener("dragenter", handleEvent);
droptarget.addEventListener("dragover", handleEvent);
droptarget.addEventListener("drop", handleEvent);
```

Как и в предыдущих примерах с перетаскиванием, вы должны отменить поведение по умолчанию событий `dragenter`, `dragover` и `drop`. Во время события удаления файлы становятся доступными в `event.dataTransfer.files`, и их информацию можно прочитать в это время.

МЕДИА-ЭЛЕМЕНТЫ

С появлением взрывной популярности встроенного аудио и видео в интернете большинство производителей контента были вынуждены использовать Flash для оптимальной кросс-браузерной совместимости. HTML5 вводит два элемента, связанных с мультимедиа, для включения кросс-браузерного аудио- и видео-встраивания в базовую линию браузера без каких-либо плагинов: `<audio>` и `<video>`.

Оба эти элемента позволяют веб-разработчикам легко встраивать мультимедийные файлы в страницу, а также предоставляют JavaScript-хуки для общей функциональности, позволяя создавать собственные элементы управления для мультимедиа. Они используются следующим образом:

```
<!-- встроенное видео -->
<video src="conference.mpg" id="myVideo">Video player not available.</video>
<!-- встроенный аудиофайл -->
<audio src="song.mp3" id="myAudio">Audio player not available.</audio>
```

Каждый из этих элементов требует как минимум наличия атрибута `src`, указывающего медиа-файл для загрузки. Можно также указать атрибуты `width` и `height`, чтобы указать предполагаемые размеры видеопроигрывателя, и атрибут `poster`, который представляет собой URI изображения, отображаемого во время загрузки видеосодержимого. Атрибут `controls`, если он присутствует, указывает, что браузер должен отображать пользовательский интерфейс, позволяющий пользователю напрямую взаимодействовать с мультимедиа. Любой контент между открывающим и закрывающим тегами считается альтернативным контентом, отображаемым, если медиаплеер недоступен.

При желании можно указать несколько различных источников мультимедиа, поскольку не все браузеры поддерживают все форматы мультимедиа. Для этого пропустите атрибут `src` элемента и вместо этого добавьте один или несколько элементов `<source>`, как в этом примере:

```
<!-- встроенное видео -->
<video id="myVideo">
    <source src="conference.webm" type="video/webm; codecs='vp8, vorbis'">
    <source src="conference.ogv" type="video/ogg; codecs='theora, vorbis'">
    <source src="conference.mpg">
    Video player not available.
</video>
<!-- встроенный аудиофайл -->
<audio id="myAudio">
    <source src="song.ogg" type="audio/ogg">
    <source src="song.mp3" type="audio/mpeg">
    Audio player not available.
</audio>
```

Обсуждение различных кодеков, используемых с видео и аудио, выходит за рамки этой книги, но достаточно сказать, что браузеры поддерживают различный диапазон кодеков, поэтому обычно требуется несколько исходных файлов.

Свойства

Элементы `<video>` и `<audio>` обеспечивают надежные интерфейсы JavaScript. Существует множество свойств, совместно используемых обоими элементами, которые можно оценить для определения текущего состояния носителя, как описано в следующей таблице.

ИМЯ СВОЙСТВА	ТИП ДАННЫХ	ОПИСАНИЕ
`autoplay`	Логический	Получает или устанавливает флаг `autoplay`
`buffered`	TimeRanges	Объект, указывающий буферизованные диапазоны времени, которые уже были загружены
`bufferedBytes`	ByteRanges	Объект, указывающий буферизованные диапазоны байтов, которые уже были загружены
`bufferingRate`	Целое число	Среднее количество бит в секунду, полученное при загрузке
`bufferingThrottled`	Логический	Указывает, была ли буферизация остановлена браузером
`controls`	Логический	Получает или задает атрибут элементов `controls`, который отображает или скрывает встроенные элементы управления браузера
`currentLoop`	Целое число	Количество циклов, воспроизведенных носителем
`currentSrc`	Строка	URL-адрес для текущего воспроизводимого мультимедиа
`currentTime`	Число с плавающей точкой	Количество секунд воспроизведения
`defaultPlaybackRate`	Число с плавающей точкой	Получает или задает скорость воспроизведения по умолчанию. По умолчанию это 1,0 секунды
`duration`	Число с плавающей точкой	Общее количество секунд для носителя
`ended`	Логический	Указывает, полностью ли воспроизведен носитель
`loop`	Логический	Получает или задает необходимость возвращения к началу после завершения воспроизведения
`muted`	Логический	Получает или задает отключение носителя

ИМЯ СВОЙСТВА	ТИП ДАННЫХ	ОПИСАНИЕ
`networkState`	Целое число	Указывает текущее состояние сетевого подключения для носителя: 0 для пустого, 1 для начала загрузки, 2 для загрузки метаданных, 3 для загрузки первого кадра и 4 для полностью загруженного носителя
`paused`	Логический	Указывает, приостановлен ли проигрыватель
`playbackRate`	Число с плавающей точкой	Получает или задает текущую скорость воспроизведения. Она может зависеть от того, что пользователь заставляет мультимедиа воспроизводиться быстрее или медленнее, в отличие от `defaultPlaybackRate`, который остается неизменным до тех пор, пока разработчик не изменит его
`played`	TimeRanges	Диапазон времени пройденного воспроизведения
`readyState`	Целое число	Указывает, готов ли носитель к воспроизведению. Значения равны 0, если данные недоступны, 1 – если текущий кадр может быть отображен, 2 – если медиа может начать воспроизведение, и 3 – если медиа может воспроизводиться от начала до конца
`seekable`	TimeRanges	Диапазоны времени, доступные для поиска
`seeking`	Логический	Указывает, был ли проигрыватель перемещен на новую позицию в медиафайле
`src`	Строка	Источник медиафайла. Может быть перезаписан в любое время
`start`	Число с плавающей точкой	Получает или задает местоположение в медиафайле в секундах, с которого должно начинаться воспроизведение
`totalBytes`	Целое число	Общее количество байтов, необходимых для ресурса (если известно)
`videoHeight`	Целое число	Возвращает высоту видео (не обязательно элемента). Только для `<video>`
`videoWidth`	Целое число	Возвращает ширину видео (не обязательно элемента). Только для `<video>`
`volume`	Число с плавающей точкой	Получает или задает текущую громкость в виде значения от 0,0 до 1,0

Многие из этих свойств также могут быть указаны как атрибуты для элементов `<audio>` или `<video>`.

События

В дополнение к многочисленным свойствам существуют также многочисленные события, которые запускают эти элементы мультимедиа. События отслеживают все различные свойства, которые изменяются из-за воспроизведения мультимедиа и взаимодействия пользователя с проигрывателем. Эти события перечислены в следующей таблице.

ИМЯ СОБЫТИЯ	ЗАПУСКАЕТСЯ, КОГДА
`abort`	Загрузка была прервана
`canplay`	Воспроизведение может начаться; `readyState` равен 2
`canplaythrough`	Воспроизведение может продолжаться и должно быть непрерывным; `readyState` равен 3
`canshowcurrentframe`	Текущий кадр был загружен; `readyState` равен 1
`dataunavailable`	Воспроизведение не может начаться, потому что нет данных; `readyState` равен 0
`durationchange`	Значение свойства `duration` изменилось
`emptied`	Сетевое соединение было прервано
`empty`	Произошла ошибка, препятствующая загрузке носителя
`ended`	Медиафайл полностью проигран и остановлен
`error`	Произошла ошибка сети во время загрузки
`load`	Все медиа были загружены. Это событие считается устаревшим; используйте вместо него `canplaythrough`
`loadeddata`	Первый кадр медиафайла был загружен
`loadedmetadata`	Метаданные носителя были загружены
`loadstart`	Загрузка началась
`pause`	Воспроизведение было приостановлено
`play`	Медиафайл был запрошен для начала воспроизведения
`playing`	Медиафайл фактически начал проигрываться
`progress`	Идет загрузка
`ratechange`	Скорость воспроизведения мультимедиа изменилась
`seeked`	Поиск закончился
`seeking`	Поиск переместился на новую позицию
`stalled`	Браузер пытается загрузить медиафайл, но данные не принимаются
`timeupdate`	Текущее время обновляется нерегулярным или неожиданным образом
`volumechange`	Значение свойства `volume` или `muted` изменилось
`waiting`	Воспроизведение приостановлено для загрузки дополнительных данных

Эти события должны быть как можно более конкретными, чтобы позволить веб-разработчикам создавать собственные аудио/видеопроигрыватели, используя что-то чуть большее, чем HTML и JavaScript (в отличие от создания нового Flash-фильма).

Пользовательские медиапроигрыватели

Можно вручную управлять воспроизведением медиафайла, используя методы `play()` и `pause()`, доступные как в `<audio>`, так и в `<video>`. Объединение свойств, событий и этих методов позволяет легко создать собственный медиапроигрыватель, как показано в следующем примере:

```
<div class="mediaplayer">
    <div class="video">
        <video id="player" src="movie.mov" poster="mymovie.jpg"
            width="300" height="200">
            Video player not available.
        </video>
    </div>
    <div class="controls">
        <input type="button" value="Play" id="video-btn">
        <span id="curtime">0</span>/<span id="duration">0</span>
    </div>
</div>
```

Затем этот базовый HTML-код можно реализовать с помощью JavaScript для создания простого видеопроигрывателя, как показано здесь:

```
// получение ссылок на элементы
let player = document.getElementById("player"),
    btn = document.getElementById("video-btn"),
    curtime = document.getElementById("curtime"),
    duration = document.getElementById("duration");
// обновление длительности
duration.innerHTML = player.duration;

// прикрепление обработчика события к кнопке
btn.addEventListener( "click", (event) => {
    if (player.paused) {
        player.play();
        btn.value = "Pause";
    } else {
        player.pause();
        btn.value = "Play";
    }
});

// периодическое обновление текущего времени воспроизведения
setInterval(() => {
    curtime.innerHTML = player.currentTime;
}, 250);
```

Код JavaScript здесь просто присоединяет обработчик событий к кнопке, которая либо приостанавливает, либо воспроизводит видео, в зависимости от его текущего

состояния. Затем для события `load` элемента `<video>` устанавливается обработчик события, чтобы можно было отобразить его продолжительность. Наконец, повторяющийся таймер установлен для обновления отображения текущего времени. Можно расширить поведение этого пользовательского видеопроигрывателя, прослушивая больше событий и используя больше свойств. Точно такой же код можно также использовать с элементом `<audio>` для создания пользовательского аудиопроигрывателя.

Обнаружение поддержки кодека

Как упоминалось ранее, не все браузеры поддерживают все кодеки для `<video>` и `<audio>`, что часто означает, что нужно предоставить более одного источника мультимедиа. Существует также JavaScript API для определения, поддерживается ли данный формат и кодек браузером. Оба медиаэлемента имеют метод `canPlayType()`, который принимает строку формата/кодека и возвращает строковое значение `"probably"`, `"maybe"` или `""` (пустая строка). Пустая строка является ложным значением, что означает, что все еще можно использовать `canPlayType()` в выражении `if`, например:

```
if (audio.canPlayType("audio/mpeg")) {
    // какой-то код
}
```

И `"probably"`, и `"maybe"` являются истинными значениями, и поэтому они приводятся к истине в контексте оператора `if`.

Когда `canPlayType()` предоставляется только MIME-тип, наиболее вероятными возвращаемыми значениями являются `"maybe"` и пустая строка, потому что файл на самом деле является просто контейнером для аудио- или видеоданных; именно кодировка действительно определяет, может ли файл воспроизводиться. Когда указан тип MIME и кодек, увеличивается вероятность получения `"probably"` в качестве возвращаемого значения. Пара примеров:

```
let audio = document.getElementById("audio-player");
// вероятнее всего "maybe"
if (audio.canPlayType("audio/mpeg")) {
    // какой-то код
}
// может быть "probably"
if (audio.canPlayType("audio/ogg; codecs=\"vorbis\"")) {
    // какой-то код
}
```

Обратите внимание, что список кодеков всегда должен быть заключен в кавычки для правильной работы. Также можно определить форматы видео, используя `canPlayType()` для любого элемента видео.

Тип аудио

Элемент `<audio>` также имеет собственный конструктор JavaScript под названием `Audio`, позволяющий воспроизводить звук в любой момент. Тип `Audio` похож на `Image` в том смысле, что он эквивалентен элементу DOM, но не требует вставки

в документ для работы. Просто создайте новый экземпляр и передайте исходный файл аудио:

```
let audio = new Audio("sound.mp3");
EventUtil.addHandler(audio, "canplaythrough", function(event) {
    audio.play();
});
```

Создание нового экземпляра `Audio` начинает процесс загрузки указанного поля. Когда все будет готово, можно вызвать `play()`, чтобы начать воспроизведение аудио.

Вызов метода `play()` в iOS возвращает всплывающее диалоговое окно с запросом разрешения пользователя на воспроизведение звука. Чтобы воспроизводить один аудиофайл за другим, нужно немедленно вызвать `play()` в обработчике события `onfinish`.

ВСТРОЕННОЕ ПЕРЕТАСКИВАНИЕ

В Internet Explorer 4 впервые появилась поддержка JavaScript для функции перетаскивания на веб-страницах. В то время только два элемента на веб-странице могли инициировать системное перетаскивание: изображение или какой-либо текст. При перетаскивании изображения вы просто удерживали кнопку мыши и затем перемещали ее; с текстом вы сначала выделите некоторый текст, а затем вы можете перетащить его так же, как изображение. В Internet Explorer 4 единственной допустимой целью перетаскивания было текстовое поле. В версии 5 Internet Explorer расширил возможности перетаскивания, добавив новые события и позволив почти всему на веб-странице стать целью для перетаскивания. Версия 5.5 пошла немного дальше, позволив практически чему-либо стать перетаскиваемым. (Internet Explorer 6 также поддерживает эту функцию.) HTML5 использует реализацию перетаскивания в Internet Explorer как основу для своей спецификации перетаскивания. Все основные браузеры реализовали встроенное перетаскивание в соответствии со спецификацией HTML5.

Пожалуй, самое интересное в поддержке перетаскивания — это то, что элементы можно перетаскивать по фреймам, окнам браузера и иногда по другим приложениям. Поддержка перетаскивания в браузере позволяет использовать эту функцию.

События перетаскивания

События, предусмотренные для перетаскивания, позволяют контролировать практически каждый аспект операции перетаскивания. Самая сложная часть — определить, где происходит каждое событие: одни вызываются на перетаскиваемом элементе, другие — на цели. При перетаскивании элемента запускаются следующие события (в указанном порядке):

1. `dragstart`
2. `drag`
3. `dragend`

В тот момент, когда вы удерживаете кнопку мыши нажатой и начинаете двигать мышью, событие `dragstart` запускается для перетаскиваемого элемента. Курсор изменится на символ запрета отпускания (кружок, перечеркнутый линией), указывающий, что элемент не может быть отпущен сам по себе. Можно использовать обработчик события `ondragstart` для запуска кода JavaScript в начале перетаскивания.

После срабатывания события `dragstart` срабатывает событие `drag` и продолжает работать, пока объект перетаскивается. Оно похоже на `mousemove`, которое также срабатывает несколько раз при перемещении мыши. Когда перетаскивание прекращается (поскольку вы перетаскиваете элемент на действительную или недействительную цель), происходит событие `dragend`.

Целью всех трех событий является перетаскиваемый элемент. По умолчанию браузер не изменяет внешний вид перетаскиваемого элемента во время перетаскивания, поэтому можно изменить его внешний вид самостоятельно. Однако большинство браузеров создают полупрозрачную копию перетаскиваемого элемента, которая всегда остается непосредственно под курсором.

Когда элемент перетаскивается через допустимую цель, запускается следующая последовательность событий:

1. `dragenter`
2. `dragover`
3. `dragleave` или `drop`

Событие `dragenter` (аналогичное событию `mouseover`) запускается, как только элемент перетаскивается на цель. Сразу после срабатывания события `dragenter` событие `dragover` срабатывает и продолжает работать, когда элемент перетаскивается в пределах границ целевого объекта. Когда элемент перетаскивается за пределы цели, `dragover` перестает работать, и запускается событие `dragleave` (аналогично событию `mouseout`). Если перетаскиваемый объект фактически отпускается на цель, событие `drop` срабатывает вместо `dragleave`. Целью этих событий является целевой элемент перетаскивания.

Пользовательские цели перетаскивания

Когда вы пытаетесь перетащить что-то на недопустимую цель, появится специальный курсор (кружок, перечеркнутый линией), указывающий, что нельзя отпустить элемент. Несмотря на то что все элементы поддерживают события цели перетаскивания, по умолчанию отпускание перетаскиваемого элемента запрещается. Если вы перетаскиваете элемент поверх чего-либо, что не допускает отпускания, событие `drop` никогда не сработает независимо от действий пользователя. Однако можно превратить любой элемент в допустимую цель для перетаскивания, переопределив поведение по умолчанию событий `dragenter` и `dragover`. Например, если у вас есть элемент `<div>` с идентификатором «`droptarget`», можно использовать следующий код, чтобы превратить его в цель перетаскивания:

```
let droptarget = document.getElementById("droptarget");

droptarget.addEventListener("dragover", (event) => {
    event.preventDefault();
});

droptarget.addEventListener("dragenter", (event) => {
    event.preventDefault();
});
```

После внесения этих изменений вы заметите, что курсор теперь указывает на то, что при перетаскивании элемента допускается перетаскивание на целевой объект. Кроме того, будет срабатывать событие `drop`.

В Firefox поведение по умолчанию для события перетаскивания заключается в переходе к URL-адресу, который был сброшен на цель перетаскивания. Это означает, что перетаскивание изображения на цель приведет к переходу страницы к файлу изображения, а текст, сброшенный на цель перетаскивания, приведет к ошибке с недопустимым URL. Для поддержки Firefox также нужно отменить поведение по умолчанию для события drop, чтобы предотвратить это перенаправление:

```
droptarget.addEventListener("drop", (event) => {
    event.preventDefault();
});
```

Объект dataTransfer

Простое перетаскивание элементов не имеет смысла, если не затрагиваются данные. Чтобы облегчить передачу данных с помощью операции перетаскивания, Internet Explorer 5 представил объект `dataTransfer`, который существует как свойство `event` и используется для передачи строковых данных из перетаскиваемого элемента в целевой объект. Поскольку это свойство объекта `event`, объект `dataTransfer` не существует, кроме как в области действия обработчика события перетаскивания. В обработчике событий можно использовать свойства и методы объекта для работы с функциями перетаскивания. Объект `dataTransfer` теперь является частью рабочего проекта HTML5.

Объект `dataTransfer` имеет два основных метода: `getData()` и `setData()`. Как и следовало ожидать, `getData()` способен получить значение, сохраненное в `setData()`. Первый аргумент для `setData()` и единственный аргумент `getData()` — это строка, указывающая тип устанавливаемых данных: `"text"` или `"URL"`, как показано здесь:

```
// работа с текстом
event.dataTransfer.setData("text", "some text");
let text = event.dataTransfer.getData("text");

// работа с URL
event.dataTransfer.setData("URL", "http://www.wrox.com/");
let url = event.dataTransfer.getData("URL");
```

Несмотря на то что Internet Explorer начинал с введения только “`text`” и `"URL"` в качестве допустимых типов данных, HTML5 расширяет это, позволяя указывать любой тип MIME. Значения `"text"` и `"URL"` будут поддерживаться HTML5 для обратной совместимости, но они сопоставляются с `"text/plain"` и `"text/uri-list"`.

Объект `dataTransfer` может содержать ровно одно значение каждого типа MIME, что означает, что можно хранить как текст, так и URL-адрес одновременно, без перезаписи. Данные, хранящиеся в объекте `dataTransfer`, доступны только до события перетаскивания. Если вы не получите данные в обработчике события `ondrop`, объект `dataTransfer` будет уничтожен, а данные потеряны.

При перетаскивании текста из текстового поля браузер вызывает `setData()` и сохраняет перетаскиваемый текст в формате `"text"`. Аналогично при перетаскивании ссылки или изображения вызывается `setData()` и сохраняется URL. Можно получить эти значения при отпускании данных над целевым объектом, используя `getData()`. Также можно вызвать `setData()` вручную во время события `dragstart`, чтобы сохранить пользовательские данные, которые, возможно, понадобятся позже.

Существует разница между данными, которые рассматриваются как текст, и данными, которые рассматриваются как URL. Когда вы указываете данные для хранения в виде текста, они не получают никакой специальной обработки. Однако при указании данных, которые должны храниться как URL, они обрабатываются как ссылка на веб-странице, а это означает, что если вы перетащите их в другое окно браузера, браузер перейдет по этому URL.

Firefox до версии 5 не соотносит в точности псевдоним `"url"` с `"text/uri-list"` или `"text"` с `"text/plain"`. Однако он создает псевдоним `"Text"` (с заглавной буквой T) для `"text/plain"`. Для лучшей кросс-браузерной совместимости извлечения данных из `dataTransfer` нужно будет проверить два значения для URL и использовать `"Text"` для простого текста:

```
let dataTransfer = event.dataTransfer;
// чтение URL
let url = dataTransfer.getData("url") || dataTransfer.getData("text/uri-list");
// чтение текста
let text = dataTransfer.getData("Text");
```

Важно, чтобы сокращенное имя данных было опробовано первым, поскольку Internet Explorer до версии 10 не поддерживает расширенные имена, а также выдает ошибку, когда имя данных не распознается.

dropEffect и effectAllowed

Объект `dataTransfer` может использоваться не только для передачи данных туда и обратно; его также можно использовать для определения того, какие действия можно выполнить с перетаскиваемым элементом и целевым объектом. Это достигается с помощью двух свойств: `dropEffect` и `effectAllowed`.

Свойство `dropEffect` используется, чтобы сообщить браузеру, какой тип поведения перетаскивания разрешен. Это свойство имеет следующие четыре возможных значения:

- `"none"` — перетаскиваемый элемент не может быть здесь отпущен. Это значение по умолчанию применяется для всего, кроме текстовых полей.
- `"move"` — перетаскиваемый элемент должен быть перемещен к цели.
- `"copy"` — перетаскиваемый элемент должен быть скопирован в целевой объект.
- `"link"` — указывает, что целевой объект будет перемещаться к перетаскиваемому элементу (но только если это URL-адрес).

Каждое из этих значений приводит к отображению другого курсора, когда элемент перетаскивается на целевой объект. Тем не менее действительный вызов действий, указанных курсором, зависит от пользователя. Другими словами, ничто не может быть автоматически перемещено, скопировано или связано без вашего непосредственного вмешательства. Единственное, что вы получаете просто так, — это смена курсора. Чтобы использовать свойство `dropEffect`, необходимо установить его в обработчике события `ondragenter` для целевого объекта.

Свойство `dropEffect` бесполезно без установки `effectAllowed`. Это свойство указывает, какой `dropEffect` разрешен для перетаскиваемого элемента. Возможные значения:

- `"uninitialized"` — для перетаскиваемого элемента не задано никаких действий.
- `"none"` — в перетаскиваемом элементе не разрешается никаких действий.
- `"copy"` — допускается только `"copy"` в `dropEffect`.
- `"link"` — допускается только `"link"` в `dropEffect`.
- `"move"` — допускается только `"move"` в `dropEffect`.
- `"copyLink"` — допускаются `"copy"` и `"link"` в `dropEffect`.
- `"copyMove"` — допускаются `"copy"` и `"move"` в `dropEffect`.
- `"linkMove"` — допускаются `"link"` и `"move"` в `dropEffect`.
- `"all"` — допускаются все значения `dropEffect`.

Это свойство должно быть установлено внутри обработчика события `ondragstart`.

Предположим, нужно разрешить пользователю перемещать текст из текстового поля в `<div>`. Для этого необходимо установить `dropEffect` и `effectAllowed` в `"move"`. Текст не будет автоматически перемещаться сам по себе, потому что стандартное поведение для события `drop` для `<div>` — ничего не делать. Если переопределить поведение по умолчанию, текст автоматически удаляется из текстового поля. Затем можно будет вставить его в `<div>` для завершения действия. При изменении `dropEffect` и `effectAllowed` на `"copy"` текст в текстовом поле не будет автоматически удален.

Возможность перетаскивания

По умолчанию изображения, ссылки и текст можно перетаскивать, а это означает, что не требуется никакого дополнительного кода, чтобы пользователь мог начать это делать. Текст можно перетаскивать только после выделения раздела, а изображения и ссылки можно перетаскивать в любой момент.

Можно сделать другие элементы перетаскиваемыми. HTML5 определяет свойство `draggable` на всех HTML-элементах, указывающее, можно ли перетаскивать элемент. Для изображений и ссылок `draggable` автоматически устанавливается в значение `true`, тогда как для всех остальных значений по умолчанию установлено значение `false`. Это свойство может быть установлено, чтобы позволить другим элементам перетаскиваться или гарантировать, что изображение или ссылка не будут перетаскиваемыми. Например:

```
<!-- запрет на перетаскивание для этого изображения -->
<img src="smile.gif" draggable="false" alt="Smiley face">
<!—разрешение перетаскивания для этого элемента -->
<div draggable="true">...</div>
```

Дополнительные члены

Спецификация HTML5 указывает следующие дополнительные методы для объекта `dataTransfer`:

- `addElement(`*`element`*`)` — добавляет элемент к операции перетаскивания. Он создан исключительно для целей данных и не влияет на внешний вид операции перетаскивания. На момент написания этой статьи ни один браузер не реализовывал этот метод.
- `clearData (`*`format`*`)` — очищает данные, хранящиеся в определенном формате.
- `setDragImage(`*`element, x, y`*`)` — позволяет указать изображение, которое будет отображаться под курсором при перетаскивании. Этот метод принимает три аргумента: отображаемый элемент HTML и координаты *x* и *y* на изображении, где должен располагаться курсор. Элемент HTML может быть изображением, в этом случае отображается само изображение, или любым другим элементом, и в этом случае отображается отрисовка элемента.
- `types` — список типов данных, которые хранятся в данный момент. Эта коллекция действует как массив и хранит типы данных в виде строк, таких как «`text`».

NOTIFICATIONS API

Notifications API, как следует из его названия (notification — уведомление), используется для отображения уведомлений пользователю. Во многих отношениях уведомления аналогичны диалоговым окнам `alert()`: оба используют JavaScript API для запуска поведения браузера за пределами самой страницы и оба позволяют странице обрабатывать различные способы взаимодействия пользователей

с диалоговыми окнами или плитками уведомлений. Уведомления, однако, предлагают гораздо большую степень настраиваемости.

Notifications API особенно полезен в контексте служебных рабочих потоков. Он позволяет прогрессивному веб-приложению (progressive web application, PWA) вести себя больше как собственное приложение, вызывая отображение уведомлений, даже когда страница браузера не активна.

Разрешения для уведомлений

Notification API потенциально может быть уязвим, поэтому по умолчанию он обеспечивает две функции безопасности:

- Уведомления могут запускаться только при выполнении кода в безопасном контексте.
- Уведомления должны быть явно разрешены пользователем для каждого источника.

Пользователь предоставляет разрешение уведомления источнику в диалоговом окне браузера. Если пользователь не откажется явно разрешить или запретить уведомления, этот запрос на разрешение может быть выполнен только один раз для домена: браузер запомнит выбор пользователя, и если ему будет отказано, возмещения не будет.

Страница может запросить разрешение на уведомление, используя глобальный объект `Notification`. Этот объект имеет метод `requestPemission()`, возвращающий обещание, которое определяется, когда пользователь выполняет действие в диалоговом окне с запросом разрешения.

```
Notification.requestPermission()
    .then((permission) => {
        console.log('User responded to permission request:', permission);
    });
```

Значение `granted` означает, что пользователь явно предоставил разрешение на показ уведомлений. Любое другое значение указывает, что попытки показать уведомление будут молча проваливаться. Если пользователь отказывает в разрешении, будет получено значение `denied`. Для этого не предусмотрено никакого программного исправления, поскольку невозможно повторно вызвать запрос на разрешение.

Отображение и скрытие уведомлений

Конструктор `Notification` используется для создания и отображения уведомлений. Самая простая форма уведомления — только строка заголовка, которая передается в качестве первого обязательного параметра в конструктор.

Когда конструктор вызывается таким образом, уведомление немедленно отобразится:

```
new Notification('Title text!');
```

Уведомления легко настраиваются с помощью параметра `options`. Такие параметры, как тело уведомления, изображения и вибрация, управляются с помощью этого объекта:

```
new Notification('Title text!', {
    body: 'Body text!',
    image: 'path/to/image.png',
    vibrate: true
});
```

Объект `Notification`, возвращаемый конструктором, можно использовать для закрытия активного уведомления с помощью метода `close()`. В следующем примере уведомление открывается, а затем закрывается через 1000 миллисекунд:

```
const n = new Notification('I will close in 1000ms');
setTimeout(() => n.close(), 1000);
```

Обратные вызовы жизненного цикла уведомлений

Уведомления не всегда используются только для отображения текстовых строк; они также могут быть интерактивными. Notification API предлагает четыре перехвата жизненного цикла для присоединения обратных вызовов:

- `onshow` запускается при отображении уведомления.
- `onclick` срабатывает при нажатии на уведомление.
- `onclose` запускается, когда уведомление закрывается или закрывается с помощью `close()`.
- `onerror` срабатывает при возникновении ошибки, препятствующей отображению уведомления.

Следующее уведомление регистрирует сообщение при каждом событии жизненного цикла:

```
const n = new Notification('foo');

n.onshow = () => console.log('Notification was shown!');
n.onclick = () => console.log('Notification was clicked!');
n.onclose = () => console.log('Notification was closed!');
n.onerror = () => console.log('Notification experienced an error!');
```

PAGE VISIBILITY API

Основной проблемой для веб-разработчиков является знание того, когда пользователи на самом деле взаимодействуют со страницей. Если страница свернута или скрыта за другой вкладкой, функционирование некоторых возможностей страницы может не иметь смысла, например опрос сервера на наличие обновлений или

выполнение анимации. Page Visibility API предназначен для предоставления разработчикам информации о том, является ли страница видимой для пользователя.

Сам API очень прост и состоит из трех частей:

- `document.visibilityState` — значение, указывающее одно из четырех состояний:
  - Страница находится на фоновой вкладке или браузер свернут.
  - Страница находится на вкладке переднего плана.
  - Фактическая страница скрыта, но предварительный просмотр страницы виден (например, в Windows 7 при наведении указателя мыши на значок на панели задач отображается предварительный просмотр).
  - Страница отображается вне экрана.
- событие `visibilitychange` — это событие возникает, когда документ изменяется со скрытого на видимый или наоборот.
- `document.hidden` — логическое значение, указывающее, является ли страница скрытой от просмотра. Это может означать, что страница находится на фоновой вкладке или что браузер свернут. Это значение поддерживается для обратной совместимости: `document.visibilityState` используется, чтобы оценить, является ли страница видимой или нет.

Чтобы получить уведомление, когда страница изменит состояние с видимой на скрытую или со скрытой на видимую, можно прослушать событие `visibilitychange`.

`document.visibilityState` может иметь одно из трех возможных строковых значений:

- `hidden`
- `visible`
- `prerender`

STREAMS API

Streams API является ответом на простой, но фундаментальный вопрос: *как веб-приложение может использовать информацию в виде последовательных фрагментов, а не в больших количествах?* Эта возможность чрезвычайно полезна в двух основных случаях:

- **Блок данных может быть доступен не сразу**. Прекрасным примером этого является ответ на сетевой запрос. Сетевые данные доставляются в виде последовательности пакетов, и потоковая обработка может позволить приложению использовать доставленные по сети данные, когда они становятся доступными, а не ожидать завершения полной загрузки данных.
- **Блок данных может обрабатываться небольшими порциями**. Обработка видео, распаковка данных, декодирование изображений и анализ JSON — все это примеры вычислений, которые локализованы для части блока данных и для которых не требуется, чтобы они были в памяти все сразу.

Глава 24 «Сетевые запросы и удаленные ресурсы» описывает, как Streams API связан с `fetch()`, но Streams API является полностью обобщаемым. Библиотеки JavaScript, которые реализуют `Observables`, разделяют многие фундаментальные концепции с потоками.

ПРИМЕЧАНИЕ Хотя Fetch API хорошо поддерживается основными браузерами, поддержка Streams API значительно отстает.

Введение в потоки

Когда мы думаем о потоках, представление данных как жидкости, протекающей по трубам, является подходящей мысленной структурой. Потоки JavaScript в значительной степени заимствованы из лексикона сантехники из-за их существенного концептуального совпадения. Согласно спецификации Streams, «эти API были разработаны для эффективного отображения на примитивы ввода/вывода низкого уровня, включая специализации для байтовых потоков, где это уместно». Двумя общими задачами, с которыми непосредственно связывается Streams API, является обработка сетевых запросов и чтение/запись на диск.

Streams API имеет три типа потоков:

- **Читаемые потоки** — это потоки, из которых можно читать фрагменты через открытый интерфейс. Данные поступают в поток из *основного источника* и обрабатываются *потребителем*.
- **Доступные для записи потоки** — это потоки, в которых фрагменты могут быть записаны через открытый интерфейс. *Производитель* записывает данные в поток, и эти данные внутренним образом передаются в *базовый приемник*.
- **Потоки преобразования** состоят из двух потоков: потока с возможностью записи для приема входных данных (*сторона с возможностью записи*) и потока с возможностью чтения для вывода выходных данных (*сторона с возможностью чтения*). Между этими двумя потоками находится *трансформатор*, который можно использовать для проверки и изменения данных потока по мере необходимости.

Фрагменты, внутренние очереди и противодавление

Основной единицей в потоках является *фрагмент*. Фрагмент может быть любого типа данных, но часто он принимает форму типизированного массива. Каждый фрагмент является отдельным сегментом потока, который может быть обработан полностью. Важно отметить, что фрагменты не имеют фиксированного размера и поступают не через фиксированные интервалы. В *идеальном* потоке фрагменты обычно имеют *примерно* одинаковый размер и достигают цели *примерно* через регулярные интервалы, но любая хорошая реализация потока должна быть подготовлена для обработки крайних случаев.

Для всех типов потоков существует общая концепция входа и выхода в поток. Иногда может появляться несоответствие между скоростью ввода и вывода данных. Этот баланс потока может принимать одну из трех форм:

- **Выход из потока может обрабатывать данные быстрее, чем данные предоставляются на входе**. Выход из потока часто будет простаивающим (что может указывать на потенциальную неэффективность на входе в поток), но при этом затрачивается мало памяти или вычислений, поэтому этот дисбаланс потока является приемлемым.
- **Вход и выход из потока находятся в равновесии**. Этот баланс идеален.
- **Вход потока может предоставить данные быстрее, чем выход может обработать их.** Этот дисбаланс потока по своей сути проблематичен. Где-то обязательно будет отставание в данных, и потоки должны обрабатывать это соответствующим образом.

Дисбаланс потоков является распространенной проблемой, но потоки спроектированы для возможности ее решения. Все потоки поддерживают *внутреннюю очередь* фрагментов, которые вошли в поток, но еще не вышли из него. Для потока, находящегося в равновесии, внутренняя очередь будет иметь ноль или небольшое количество фрагментов внутри себя, потому что выход потока удаляет фрагменты примерно с той же скоростью, с которой новые фрагменты передаются в очередь. Объем памяти внутренней очереди такого потока останется относительно небольшим.

Когда фрагменты ставятся в очередь быстрее, чем удаляются из нее, размер внутренней очереди будет расти. Поток не может позволить своей внутренней очереди расти бесконечно, поэтому он использует *противодавление*, чтобы сигнализировать входу потока о прекращении отправки данных до тех пор, пока размер очереди не опустится ниже заданного порогового значения. Этот порог определяется *стратегией очередей*, которая определяет *верхнюю отметку*, максимальный объем памяти внутренней очереди.

Читаемые потоки

Читаемые потоки являются оболочкой для основного источника данных. Этот базовый источник способен передавать свои данные в поток и позволяет считывать эти данные из открытого интерфейса потока.

Использование ReadableStreamDefaultController

Рассмотрим следующий генератор, который выдает увеличенное целое число каждые 1000 миллисекунд:

```
async function* ints() {
    // возврат увеличенного числа каждые 1000мс
    for (let i = 0; i < 5; ++i) {
        yield await new Promise((resolve) => setTimeout(resolve, 1000, i));
    }
}
```

Эти значения могут быть переданы в читаемый поток через его контроллер. Самый простой способ получить доступ к контроллеру — создать новый экземпляр `ReadableStream`, определить метод `start()` внутри параметра `basicSource` конструктора и использовать параметр контроллера, передаваемый этому методу. По умолчанию параметр контроллера является экземпляром `ReadableStreamDefaultController`:

```
const readableStream = new ReadableStream({
    start(controller) {
        console.log(controller); // ReadableStreamDefaultController {}
    }
});
```

Используйте метод `enqueue()` для передачи значений в контроллер. Как только все значения пройдены, поток закрывается с помощью `close()`:

```
async function* ints() {
    // возврат увеличенного числа каждые 1000мс
    for (let i = 0; i < 5; ++i) {
        yield await new Promise((resolve) => setTimeout(resolve, 1000, i));
    }
}

const readableStream = new ReadableStream({
    async start(controller) {
        for await (let chunk of ints()) {
            controller.enqueue(chunk);
        }

        controller.close();
    }
});
```

Использование ReadableStreamDefaultReader

Этот пример успешно помещает в очередь пять значений в потоке, но ничто не считывает их из этой очереди. Для этой задачи можно получить из потока экземпляр `ReadableStreamDefaultReader` с помощью `getReader()`. Это обеспечит блокировку потока, гарантируя, что только этот читатель может читать значения из этого потока:

```
async function* ints() {
    // возврат увеличенного числа каждые 1000мс

    for (let i = 0; i < 5; ++i) {
        yield await new Promise((resolve) => setTimeout(resolve, 1000, i));
    }
}

const readableStream = new ReadableStream({
    async start(controller) {
        for await (let chunk of ints()) {
            controller.enqueue(chunk);
        }

        controller.close();
```

```
    }
});

console.log(readableStream.locked);    // false
const readableStreamDefaultReader = readableStream.getReader();
console.log(readableStream.locked);    // true
```

Потребитель может получить значения из этого экземпляра читателя, используя метод `read()`:

```
async function* ints() {
    // возврат увеличенного числа каждые 1000мс
    for (let i = 0; i < 5; ++i) {
        yield await new Promise((resolve) => setTimeout(resolve, 1000, i));
    }
}

const readableStream = new ReadableStream({
    async start(controller) {
        for await (let chunk of ints()) {
            controller.enqueue(chunk);
        }

        controller.close();
    }
});

console.log(readableStream.locked); // false
const readableStreamDefaultReader = readableStream.getReader();
console.log(readableStream.locked); // true

// Потребитель
(async function() {
    while(true) {
        const { done, value } = await readableStreamDefaultReader.read();

        if (done) {
            break;
        } else {
            console.log(value);
        }
    }
})();

// 0
// 1
// 2
// 3
// 4
```

Записываемые потоки

Записываемые потоки являются оболочкой для базового приемника данных. Этот базовый приемник обрабатывает данные из открытого интерфейса потока.

Создание WriteableStream

Рассмотрим следующий генератор, который выдает увеличенное целое число каждые 1000 миллисекунд:

```
async function* ints() {
    // возврат увеличенного числа каждые 1000мс
    for (let i = 0; i < 5; ++i) {
        yield await new Promise((resolve) => setTimeout(resolve, 1000, i));
    }
}
```

Эти значения могут быть записаны в доступный для записи поток через его открытый интерфейс. Когда вызывается открытый метод `write()`, также вызывается метод `write()`, определенный для объекта basicSink, переданного в конструктор:

```
const readableStream = new ReadableStream({
    write(value) {
        console.log(value);
    }
});
```

Использование WritableStreamDefaultWriter

Для записи значений в этот поток экземпляр `WritableStreamDefaultWriter` можно получить из потока с помощью `getWriter()`. Это обеспечит блокировку потока, гарантируя, что только этот писатель может записывать значения в поток:

```
async function* ints() {
    // возврат увеличенного числа каждые 1000мс
    for (let i = 0; i < 5; ++i) {
        yield await new Promise((resolve) => setTimeout(resolve, 1000, i));
    }
}

const writableStream = new WritableStream({
    write(value) {
        console.log(value);
    }
});

console.log(writableStream.locked);    // false
const writableStreamDefaultWriter = writableStream.getWriter();
console.log(writableStream.locked);    // true
```

Перед записью значений в поток производитель должен убедиться, что записывающее устройство может принимать значения. `WritableStreamDefaultWriter.ready` возвращает промис, которое разрешается, когда модуль записи готов к записи значений в поток. После этого значения можно передавать с помощью `write()` до тех пор, пока поток данных не будет завершен, после чего поток может быть закрыт с помощью `close()`:

```
async function* ints() {
    // возврат увеличенного числа каждые 1000мс
    for (let i = 0; i < 5; ++i) {
        yield await new Promise((resolve) => setTimeout(resolve, 1000, i));
    }
}

const writableStream = new WritableStream({
    write(value) {
        console.log(value);
    }
});

console.log(writableStream.locked);    // false
const writableStreamDefaultWriter = writableStream.getWriter();
console.log(writableStream.locked);    // true

// Производитель
(async function() {
    for await (let chunk of ints()) {
        await writableStreamDefaultWriter.ready;
        writableStreamDefaultWriter.write(chunk);
    }

    writableStreamDefaultWriter.close();
})();
```

Потоки преобразования

Потоки преобразования объединяют читаемый и записываемый потоки. Между двумя потоками находится метод `transform()`, который является промежуточной точкой, в которой происходит преобразование фрагмента.

Рассмотрим следующий генератор, который выдает увеличенное целое число каждые 1000 миллисекунд:

```
async function* ints() {
    // возврат увеличенного числа каждые 1000мс
    for (let i = 0; i < 5; ++i) {
        yield await new Promise((resolve) => setTimeout(resolve, 1000, i));
    }
}
```

`TransformStream`, который удваивает значения, выдаваемые этим генератором, может быть определен следующим образом:

```
async function* ints() {
    // возврат увеличенного числа каждые 1000мс
    for (let i = 0; i < 5; ++i) {
        yield await new Promise((resolve) => setTimeout(resolve, 1000, i));
    }
}

const { writable, readable } = new TransformStream({
```

```
    transform(chunk, controller) {
        controller.enqueue(chunk * 2);
    }
});
```

Передача и получение данных в потоке преобразования могут быть выполнены идентично предыдущим разделам о читаемом и записываемом потоках в этой главе:

```
async function* ints() {
    // возврат увеличенного числа каждые 1000мс
    for (let i = 0; i < 5; ++i) {
        yield await new Promise((resolve) => setTimeout(resolve, 1000, i));
    }
}

const { writable, readable } = new TransformStream({
    transform(chunk, controller) {
        controller.enqueue(chunk * 2);
    }
});

const readableStreamDefaultReader = readable.getReader();
const writableStreamDefaultWriter = writable.getWriter();

// Потребитель
(async function() {
    while (true) {
        const { done, value } = await readableStreamDefaultReader.read();

        if (done) {
            break;
        } else {
            console.log(value);
        }
    }
})();

// Производитель
(async function() {
    for await (let chunk of ints()) {
        await writableStreamDefaultWriter.ready;
        writableStreamDefaultWriter.write(chunk);
    }

    writableStreamDefaultWriter.close();
})();
```

Соединение потоков

Потоки могут быть соединены друг с другом в цепочку. Одной из распространенных форм этого является передача `ReadableStream` в `TransformStream` с использованием метода `pipeThrough()`. Под капотом начальный `ReadableStream` передает свои значения в `WritableStream` внутри `TransformStream`, поток выполняет преобразование,

и преобразованные значения передаются из новой конечной точки `ReadableStream`. Рассмотрим следующий пример, где `ReadableStream` целых чисел передается через `TransformStream`, который удваивает каждое значение:

```
async function* ints() {
    // возврат увеличенного числа каждые 1000мс
    for (let i = 0; i < 5; ++i) {
        yield await new Promise((resolve) => setTimeout(resolve, 1000, i));
    }
}

const integerStream = new ReadableStream({
    async start(controller) {
        for await (let chunk of ints()) {
            controller.enqueue(chunk);
        }

        controller.close();
    }
});

const doublingStream = new TransformStream({
    transform(chunk, controller) {
        controller.enqueue(chunk * 2);
    }
});

// Соединение потоков
const pipedStream = integerStream.pipeThrough(doublingStream);

// Получение читателя вывода соединенных потоков
const pipedStreamDefaultReader = pipedStream.getReader();

// Потребитель
(async function() {
    while(true) {
        const { done, value } = await pipedStreamDefaultReader.read();

        if (done) {
            break;
        } else {
            console.log(value);
        }
    }
})();

// 0
// 2
// 4
// 6
// 8
```

Также можно передать `ReadableStream` в `WritableStream` с помощью метода `pipeTo()`. Он ведет себя аналогично `pipeThrough()`:

```
async function* ints() {
    // возврат увеличенного числа каждые 1000мс
    for (let i = 0; i < 5; ++i) {
        yield await new Promise((resolve) => setTimeout(resolve, 1000, i));
    }
}

const integerStream = new ReadableStream({
    async start(controller) {
        for await (let chunk of ints()) {
            controller.enqueue(chunk);
        }

        controller.close();
    }
});

const writableStream = new WritableStream({
    write(value) {
        console.log(value);
    }
});

const pipedStream = integerStream.pipeTo(writableStream);

// 0
// 1
// 2
// 3
// 4
```

Обратите внимание, что операция соединения неявно получает читателя из `ReadableStream` и передает полученные значения в `WritableStream`.

API ПРОИЗВОДИТЕЛЬНОСТИ

Производительность страниц всегда вызывает беспокойство у веб-разработчиков. Интерфейс Performance изменяет это, предоставляя внутренние метрики браузера через JavaScript API, что позволяет разработчикам напрямую получать доступ к этой информации и делать с ней все, что необходимо. Этот интерфейс доступен через объект `window.performance`. Все метрики, относящиеся к странице, как уже определенные, так и в будущем, существуют для этого объекта.

Интерфейс Performance состоит из нескольких API, большинство из которых имеют два уровня спецификации:

- **High Resolution Time API**

  Уровень 1: https://www.w3.org/TR/hr-time-1/#dom-performance-now

  Уровень 2: https://www.w3.org/TR/hr-time/

- **Performance Timeline API**

  Уровень 1: https://www.w3.org/TR/performance-timeline/#sec-window

  .performance-attribute

  Уровень 2: https://w3c.github.io/performance-timeline/#extensions-to-theperformance-interface

- **Navigation Timing API**

  Уровень 1: https://www.w3.org/TR/navigation-timing/

  Уровень 2: https://w3c.github.io/navigation-timing/

- **User Timing API**

  Уровень 1: https://www.w3.org/TR/user-timing/#extensions-performance-interface

  Уровень 2: https://w3c.github.io/user-timing/#extensions-performance-interface

- **Resource Timing API**

  Уровень 1: https://www.w3.org/TR/resource-timing-1/#extensions-performanceinterface

  Уровень 2: https://www.w3.org/TR/resource-timing-2/#extensions-performanceinterface

- **Paint Timing API**

  https://w3c.github.io/paint-timing/#sec-PerformancePaintTiming

> **ПРИМЕЧАНИЕ** Браузеры обычно поддерживают устаревшую версию 1 и заменяющую версию 2. Этот раздел посвящен спецификации уровня 2 во всех случаях, где это применимо.

High Resolution Time API

Метод `Date.now()` полезен только для операций `datetime`, которые не требуют точного хронометража. В следующем примере метка времени записывается до и после вызова функции `foo()`:

```
const t0 = Date.now();
foo();
const t1 = Date.now();
const duration = t1 – t0;

console.log(duration);
```

Рассмотрим следующие сценарии, в которых `duration` имеет неожиданное значение:

- **duration равен 0**. `Date.now()` имеет точность только в миллисекундах, и обе метки времени будут захватывать одно и то же значение, если `foo()` выполняется достаточно быстро.
- **duration отрицательный или огромный**. Если системные часы спешат или отстают во время выполнения функции `foo()` (например, во время перехода на

летнее время), полученные метки времени не будут учитывать это, и разница будет включать в себя настройку.

По этим причинам для точного измерения времени необходимо использовать другой API для измерения времени. Чтобы удовлетворить эти потребности, High Resolution Time API определяет `window.performance.now()`, который возвращает число с плавающей запятой с точностью до микросекунды. В результате гораздо менее вероятно, чтобы последовательно захваченные метки времени будут идентичными. Этот метод также гарантирует монотонно увеличивающиеся временные метки.

```
const t0 = performance.now();
const t1 = performance.now();

console.log(t0);    // 1768.625000026077
console.log(t1);    // 1768.6300000059418

const duration = t1 – t0;

console.log(duration);    // 0.004999979864805937
```

Таймер `performance.now()` является *относительным* измерением. Он начинает отсчет с 0, когда создается его контекст выполнения: например, когда страница открывается или когда создается рабочий поток. Поскольку инициализация таймера будет смещена между контекстами, прямое сравнение значений `performance.now()` по контекстам выполнения невозможно без общей контрольной точки. Свойство `performance.timeOrigin` возвращает значение глобальных системных часов при инициализации таймера.

```
const relativeTimestamp = performance.now();

const absoluteTimestamp = performance.timeOrigin + relativeTimestamp;

console.log(relativeTimestamp);    // 244.43500000052154
console.log(absoluteTimestamp);    // 1561926208892.4001
```

ПРИМЕЧАНИЕ Эксплойты безопасности, такие как Spectre, могут выполнять атаки на кеш с помощью `performance.now()` для измерения задержки между кешем L1 и основной памятью. Чтобы устранить эту уязвимость, все основные браузеры либо снизили точность `performance.now()`, либо включили некоторую случайность в метку времени. В блоге WebKit есть отличная статья на эту тему по адресу https://webkit.org/blog/8048/what-spectre-and-meltdown-mean-for-webkit/.

Performance Timeline API

Performance Timeline API расширяет интерфейс Performance набором инструментов, предназначенных для измерения задержки на стороне клиента. Измерения производительности почти всегда принимают форму расчета разницы между временем окончания и начала. Это время начала и окончания записывается в виде значений

DOMHighResTimeStamp, а объекты, заключающие эти метки времени, являются экземплярами PerformanceEntry.

Браузер автоматически записывает различные объекты PerformanceEntry, а также предоставляет возможность записывать свои собственные с помощью performance.mark(). Все записи, созданные в контексте выполнения, могут быть доступны с помощью performance.getEntires():

```
console.log(performance.getEntries());

// [PerformanceNavigationTiming, PerformanceResourceTiming, ... ]
```

Эта коллекция представляет *график производительности* браузера. Каждый объект PerformanceEntry имеет свойства name, entryType, startTime и duration:

```
const entry = performance.getEntries()[0];

console.log(entry.name);         // "https://foo.com"
console.log(entry.entryType);    // navigation
console.log(entry.startTime);    // 0
console.log(entry.duration);     // 182.36500001512468
```

Однако PerformanceEntry по сути является абстрактным базовым классом, поскольку записи всегда будут наследоваться от PerformanceEntry, но в конечном итоге будут существовать как один из следующих классов:

- PerformanceMark
- PerformanceMeasure
- PerformanceFrameTiming
- PerformanceNavigationTiming
- PerformanceResourceTiming
- PerformancePaintTiming

Каждый из этих типов добавляет *значительное* количество свойств, которые описывают метаданные, включающие то, что представляет запись. Свойство name и entryType для экземпляра будут различаться в зависимости от его типа.

User Timing API

User Timing API позволяет записывать и анализировать пользовательские записи производительности. Пользовательская запись производительности создается с помощью performance.mark():

```
performance.mark('foo');

console.log(performance.getEntriesByType('mark')[0]);
// PerformanceMark {
//    name: "foo",
//    entryType: "mark",
//    startTime: 269.8800000362098,
```

```
//    duration: 0
// }
```

Создание двух записей производительности по обе стороны вычислений позволяет рассчитать разницу времени. Самые новые метки помещаются в начало массива, возвращаемого из `getEntriesByType()`:

```
performance.mark('foo');
for (let i = 0; i < 1E6; ++i) {}
performance.mark('bar');

const [endMark, startMark] = performance.getEntriesByType('mark');
console.log(startMark.startTime - endMark.startTime);    // 1.3299999991431832
```

Также можно создать запись `PerformanceMeasure`, которая соответствует разнице времени между двумя метками, указанными с помощью их имен. Это достигается с использованием `performance.measure()`:

```
performance.mark('foo');
for (let i = 0; i < 1E6; ++i) {}
performance.mark('bar');

performance.measure('baz', 'foo', 'bar');

const [differenceMark] = performance.getEntriesByType('measure');

console.log(differenceMark);
// PerformanceMeasure {
// name: "baz",
// entryType: "measure",
// startTime: 298.9800000214018,
// duration: 1.349999976810068
// }
```

Navigation Timing API

Navigation Timing API предлагает высокоточные метки времени для метрик, определяющих скорость загрузки текущей страницы. Браузер автоматически создает запись `PerformanceNavigationTiming` при возникновении события навигации. Этот объект охватывает широкий диапазон временных меток, описывающих, как и когда страница загружена.

В следующем примере вычисляется количество времени между отметками времени `loadEventStart` и `loadEventEnd`:

```
const [performanceNavigationTimingEntry] = performance.
getEntriesByType('navigation');

console.log(performanceNavigationTimingEntry);
// PerformanceNavigationTiming {
// connectEnd: 2.259999979287386
// connectStart: 2.259999979287386
// decodedBodySize: 122314
// domComplete: 631.9899999652989
```

```
// domContentLoadedEventEnd: 300.92499998863786
// domContentLoadedEventStart: 298.8950000144541
// domInteractive: 298.88499999651685
// domainLookupEnd: 2.259999979287386
// domainLookupStart: 2.259999979287386
// duration: 632.819999998901
// encodedBodySize: 21107
// entryType: "navigation"
// fetchStart: 2.259999979287386
// initiatorType: "navigation"
// loadEventEnd: 632.819999998901
// loadEventStart: 632.0149999810383
// name: "https://foo.com"
// nextHopProtocol: "h2"
// redirectCount: 0
// redirectEnd: 0
// redirectStart: 0
// requestStart: 7.7099999762140214
// responseEnd: 130.50999998813495
// responseStart: 127.16999999247491
// secureConnectionStart: 0
// serverTiming: []
// startTime: 0
// transferSize: 21806
// type: "navigate"
// unloadEventEnd: 132.73999997181818
// unloadEventStart: 132.41999997990206
// workerStart: 0
// }
console.log(performanceNavigationTimingEntry.loadEventEnd –
 performanceNavigationTimingEntry.loadEventStart);
// 0.805000017862767
```

Resource Timing API

Resource Timing API предлагает высокоточные временные метки для метрик, определяющих, насколько быстро загружаются ресурсы для текущей страницы. Браузер автоматически создает запись `PerformanceResourceTiming` при загрузке ресурса. Этот объект захватывает широкий диапазон временных меток, описывающих, как быстро загружается этот ресурс.

В следующем примере вычисляется время, необходимое для загрузки определенного ресурса:

```
const performanceResourceTimingEntry = performance.getEntriesByType('resource')[0];
console.log(performanceResourceTimingEntry);
// PerformanceResourceTiming {
// connectEnd: 138.11499997973442
// connectStart: 138.11499997973442
// decodedBodySize: 33808
// domainLookupEnd: 138.11499997973442
// domainLookupStart: 138.11499997973442
// duration: 0
// encodedBodySize: 33808
```

```
// entryType: "resource"
// fetchStart: 138.11499997973442
// initiatorType: "link"
// name: "https://static.foo.com/bar.png",
// nextHopProtocol: "h2"
// redirectEnd: 0
// redirectStart: 0
// requestStart: 138.11499997973442
// responseEnd: 138.11499997973442
// responseStart: 138.11499997973442
// secureConnectionStart: 0
// serverTiming: []
// startTime: 138.11499997973442
// transferSize: 0
// workerStart: 0
// }
console.log(performanceResourceTimingEntry.responseEnd –
 performanc eResourceTimingEntry.requestStart);
// 493.9600000507198
```

Использование разницы между различными метками времени может дать хорошее представление о том, как страница загружается в браузер и где скрываются потенциальные узкие места.

ВЕБ-КОМПОНЕНТЫ

Термин «веб-компоненты» относится к нескольким инструментам, разработанным для улучшения поведения DOM: теневой DOM, пользовательские элементы и шаблоны HTML. Эта коллекция браузерных API особенно запутанна:

- Нет единой спецификации «Веб-компоненты»: каждый веб-компонент определен в отдельной спецификации.
- Несколько веб-компонентов, таких как теневой DOM и пользовательские элементы, подверглись обратно-несовместимому версионированию.
- Поддержка среди поставщиков браузеров крайне противоречива.

Из-за этих проблем для принятия веб-компонентов часто требуется библиотека веб-компонентов, такая как Polymer (https://www.polymer-project.org/), чтобы заменить и эмулировать отсутствующие веб-компоненты в браузере.

> **ПРИМЕЧАНИЕ** В этой главе рассматриваются только самые последние версии веб-компонентов.

Шаблоны HTML

До веб-компонентов не существовало особенно хорошего способа написания HTML, который позволял бы браузеру создавать поддерево DOM из проанализированного

HTML, но отказывался отображать это поддерево до получения конкретного указания. Одним из обходных путей было использование innerHTML для преобразования строки разметки в элементы DOM, но эта стратегия имеет серьезные последствия для безопасности. Другим обходным решением было создание каждого элемента с использованием `document.createElement()` и постепенное присоединение их к осиротевшему корневому узлу (не присоединенному к DOM), но это довольно трудоемко и обходится без использования разметки вообще.

Вместо этого было бы гораздо лучше написать специальную разметку на странице, которую браузер автоматически анализирует в поддереве DOM, но пропускает саму отрисовку. Это основная идея шаблонов HTML, которые используют тег `<template>` именно для такой цели. Простой пример шаблона HTML выглядит следующим образом:

```
<template id="foo">
    <p>I'm inside a template!</p>
</template>
```

Использование DocumentFragment

При отрисовке внутри браузера вы не увидите этот отрисованный текст на странице. Поскольку содержимое `<template>` не считается частью активного документа, методы сопоставления DOM, такие как `document.querySelector()`, не смогут найти тег `<p>`. Это потому, что он существует внутри нового подкласса `Node`, добавленного как часть шаблонов HTML: `DocumentFragment`.

`DocumentFragment` внутри `<template>` виден при проверке внутри браузера:

```
<template id="foo">
    #document-fragment
    <p>I'm inside a template!</p>
</template>
```

Ссылку на этот `DocumentFragment` можно получить с помощью свойства `content` элемента `<template>`:

```
console.log(document.querySelector('#foo').content);    // #document-fragment
```

`DocumentFragment` ведет себя как минимальный объект документа для этого поддерева. Например, методы сопоставления DOM в `DocumentFragment` *могут* найти узлы в своем поддереве:

```
const fragment = document.querySelector('#foo').content;

console.log(document.querySelector('p'));    // null
console.log(fragment.querySelector('p'));    // <p>...<p>
```

`DocumentFragment` также невероятно полезен для массового добавления HTML. Рассмотрим сценарий, в котором желательно максимально эффективно добавить несколько дочерних элементов в элемент HTML. Использование последовательных вызовов `document.appendChild()` для каждого дочернего элемента является

кропотливым и может привести к многократному повторному использованию. Использование `DocumentFragment` позволяет пакетировать эти дочерние дополнения, гарантируя не более одного перекомпонования:

```
// Начальное состояние:
// <div id="foo"></div>
//
// Желаемое конечное состояние:
// <div id="foo">
//    <p></p>
//    <p></p>
//    <p></p>
// </div>

// Также можно использовать document.createDocumentFragment()
const fragment = new DocumentFragment();

const foo = document.querySelector('#foo');

// Добавление дочерних элементов в DocumentFragment не вызывает перекомпоновку
fragment.appendChild(document.createElement('p'));
fragment.appendChild(document.createElement('p'));
fragment.appendChild(document.createElement('p'));

console.log(fragment.children.length); // 3

foo.appendChild(fragment);

console.log(fragment.children.length); // 0

console.log(document.body.innerHTML);
// <div id="foo">
//    <p></p>
//    <p></p>
//    <p></p>
// </div>
```

Использование тегов <template>

Обратите внимание в предыдущем примере, как дочерние узлы `DocumentFragment` эффективно переносятся в элемент `foo`, оставляя `DocumentFragment` пустым. Эту же процедуру можно повторить с помощью `<template>`:

```
const fooElement = document.querySelector('#foo');
const barTemplate = document.querySelector('#bar');
const barFragment = barTemplate.content;

console.log(document.body.innerHTML);
// <div id="foo">
// </div>
// <template id="bar">
//    <p></p>
//    <p></p>
//    <p></p>
```

```
// </template>

fooElement.appendChild(barFragment);

console.log(document.body.innerHTML);
// <div id="foo">
//    <p></p>
//    <p></p>
//    <p></p>
// </div>
// <tempate id="bar"></template>
```

Если нужно вместо этого скопировать шаблон, можно использовать простой `importNode()` для клонирования `DocumentFragment`:

```
const fooElement = document.querySelector('#foo');
const barTemplate = document.querySelector('#bar');
const barFragment = barTemplate.content;

console.log(document.body.innerHTML);
// <div id="foo">
// </div>
// <template id="bar">
//    <p></p>
//    <p></p>
//    <p></p>
// </template>

fooElement.appendChild(document.importNode(barFragment, true));

console.log(document.body.innerHTML);
// <div id="foo">
//    <p></p>
//    <p></p>
//    <p></p>
// </div>
// <template id="bar">
//    <p></p>
//    <p></p>
//    <p></p>
// </template>
```

Сценарии шаблонов

Выполнение сценария будет отложено до тех пор, пока `DocumentFragment` не будет добавлен в реальное дерево DOM. Это демонстрируется здесь:

```
// HTML страницы:
//
// <div id="foo"></div>
// <template id="bar">
//    <script>console.log('Template script executed');</script>
// </template>

const fooElement = document.querySelector('#foo');
```

```
const barTemplate = document.querySelector('#bar');
const barFragment = barTemplate.content;

console.log('About to add template');
fooElement.appendChild(barFragment);
console.log('Added template');

// About to add template
// Template script executed
// Added template
```

Это полезно в ситуациях, когда добавление новых элементов требует некоторой инициализации.

Теневая DOM

Концептуально веб-компонент теневой DOM довольно прост: он позволяет присоединить совершенно отдельное дерево DOM в качестве узла к родительскому дереву DOM. Это позволяет инкапсулировать DOM, что означает, что такие вещи, как стилизация CSS и селекторы CSS, могут быть ограничены поддеревом теневой DOM вместо всего дерева DOM верхнего уровня.

Теневая DOM похожа на HTML-шаблоны в том, что оба они представляют собой документоподобную структуру, обеспечивающую степень отделения от DOM верхнего уровня. Однако теневая DOM отличается от HTML-шаблонов тем, что контент теневой DOM фактически отображается на странице, тогда как контент HTML-шаблона — нет.

Введение в теневую DOM

Рассмотрим сценарий, в котором у вас есть несколько аналогично структурированных поддеревьев DOM:

```
<div>
    <p>Make me red!</p>
</div>
<div>
    <p>Make me blue!</p>
</div>
<div>
    <p>Make me green!</p>
</div>
```

Как вы, вероятно, догадались по текстовым элементам, каждому из этих трех поддеревьев DOM должны быть назначены разные цвета (*red* — красный, *blue* — синий, *green* — зеленый). Обычно, чтобы применить стиль уникальным образом к каждому из них, не прибегая к атрибуту стиля, вы, вероятно, примените уникальное имя класса к каждому поддереву и определите стиль внутри соответствующего селектора:

```
<div class="red-text">
    <p>Make me red!</p>
</div>
```

```
<div class="green-text">
    <p>Make me green!</p>
</div>
<div class="blue-text">
    <p>Make me blue!</p>
</div>

<style>
.red-text {
    color: red;
}
.green-text {
    color: green;
}
.blue-text {
    color: blue;
}
</style>
```

Конечно, это далеко не идеальное решение. Это не сильно отличается от определения переменных в глобальном пространстве имен; этот CSS будет применяться ко всему DOM, даже если вы точно знаете, что эти определения стилей больше нигде не нужны. Можно продолжать добавлять специфичность CSS-селекторов, чтобы эти стили не всплывали в других местах, но это немногим больше, чем полумера. В идеале предпочтительно было бы ограничить CSS только частью DOM: в этом заключается полезность теневой DOM.

Создание теневой DOM

По причинам, связанным с безопасностью или предотвращением столкновений теневой DOM, она может быть прикреплена не ко всем типам элементов. Попытка присоединить теневую DOM к недопустимому типу элемента или элементу с уже прикрепленной теневой DOM приведет к ошибке.

Ниже приведен список элементов, способных разместить теневую DOM:

- Любой автономный пользовательский элемент с допустимым именем (как определено в спецификации HTML: https://html.spec.whatwg.org/multipage/custom-elements.html#valid-custom-element-name)
- `<article>`
- `<aside>`
- `<blockquote>`
- `<body>`
- `<div>`
- `<footer>`
- `<h1>`
- `<h2>`
- `<h3>`
- `<h4>`

- `<h5>`
- `<h6>`
- `<header>`
- `<main>`
- `<nav>`
- `<p>`
- `<section>`
- `<span>`

Теневая DOM создается путем присоединения ее к допустимому элементу HTML с помощью метода `attachShadow()`. Элемент, к которому присоединена теневая DOM, называется *теневым хостом*. Корневой узел теневой DOM называется *теневым корнем*.

Метод `attachShadow()` ожидает требуемый объект `shadowRootInit` и возвращает экземпляр теневой DOM. Объект `shadowRootInit` должен содержать одно свойство `mode`, определяющее либо `"open"`, либо `"closed"`. Ссылка на `open` теневую DOM может быть получена для HTML-элемента через свойство `shadowRoot`; это невозможно с закрытой теневой DOM.

Разница между `mode` демонстрируется здесь:

```
document.body.innerHTML = `
    <div id="foo"></div>
    <div id="bar"></div>
`;

const foo = document.querySelector('#foo');
const bar = document.querySelector('#bar');

const openShadowDOM = foo.attachShadow({ mode: 'open' });
const closedShadowDOM = bar.attachShadow({ mode: 'closed' });

console.log(openShadowDOM);    // #shadow-root (open)
console.log(closedShadowDOM);    // #shadow-root (closed)

console.log(foo.shadowRoot);    // #shadow-root (open)
console.log(bar.shadowRoot);    // null
```

В общем случае редко возникает ситуация, когда необходимо создать закрытую теневую DOM. Хотя это дает возможность ограничивать программный доступ к теневой DOM с теневого хоста, существует множество способов обойти этот вредоносный код и восстановить доступ к теневой DOM. Короче говоря, создание закрытой теневой DOM *не* должно использоваться в целях безопасности.

> **ПРИМЕЧАНИЕ** Если вы хотите защитить отдельное дерево DOM от ненадежного кода, теневая DOM не подходит для такого требования. Ограничения перекрестного происхождения, введенные для `<iframe>`, гораздо более надежны.

Использование теневой DOM

После подключения к элементу теневая DOM может использоваться как обычный DOM. Рассмотрим следующий пример, который воссоздает красный/зеленый/синий пример, показанный ранее:

```
for (let color of ['red', 'green', 'blue']) {
    const div = document.createElement('div');
    const shadowDOM = div.attachShadow({ mode: 'open' });

    document.body.appendChild(div);

shadowDOM.innerHTML = `
    <p>Make me ${color}</p>

    <style>
        p {
        color: ${color};
    }
 </style>
 `;
}
```

Хотя есть три идентичных селектора, применяющих три разных цвета, они будут применены только к теневой DOM, в которой они определены. Таким образом, три элемента `<p>` будут отображаться в трех разных цветах.

Вы можете проверить, что эти элементы существуют в их собственной теневой DOM, следующим образом:

```
for (let color of ['red', 'green', 'blue']) {
    const div = document.createElement('div');
    const shadowDOM = div.attachShadow({ mode: 'open' });

    document.body.appendChild(div);

    shadowDOM.innerHTML = `
        <p>Make me ${color}</p>

        <style>
            p {
                color: ${color};
            }
        </style>
    `;
}

function countP(node) {
    console.log(node.querySelectorAll('p').length);
}

countP(document); // 0

for (let element of document.querySelectorAll('div')) {
```

```
    countP(element.shadowRoot);
}

// 1
// 1
// 1
```

Инструменты анализа браузера выяснят, где существует теневая DOM. Например, предыдущий пример будет выглядеть следующим образом в инструменте анализа:

```
<body>
<div>
    #shadow-root (open)
    <p>Make me red!</p>

    <style>
        p {
            color: red;
        }
    </style>
</div>
<div>
    #shadow-root (open)
    <p>Make me green!</p>

    <style>
        p {
            color: green;
        }
    </style>
</div>
<div>
    #shadow-root (open)
    <p>Make me blue!</p>

    <style>
        p {
            color: blue;
        }
    </style>
</div>
</body>
```

Теневые DOM не являются непроницаемой границей. HTML-элемент можно перемещать между деревьями DOM без ограничений.

```
document.body.innerHTML = `
<div></div>
<p id="foo">Move me</p>
`;

const divElement = document.querySelector('div');
const pElement = document.querySelector('p');

const shadowDOM = divElement.attachShadow({ mode: 'open' });
```

```
// Удаление элемента из родительской DOM
divElement.parentElement.removeChild(pElement);

// Добавление элемента к теневой DOM
shadowDOM.appendChild(pElement);

// Проверка перемещения элемента
console.log(shadowDOM.innerHTML);    // <p id="foo">Move me</p>
```

Композиция и слоты теневой DOM

Теневая DOM предназначена для добавления настраиваемых компонентов, и для этого требуется способность обрабатывать вложенные фрагменты DOM. С концептуальной точки зрения это относительно просто: HTML-код внутри элемента теневого хоста нуждается в способе отрисовки внутри теневой DOM, фактически не являясь частью дерева теневой DOM.

По умолчанию вложенный контент будет скрыт. Рассмотрим следующий пример, где текст становится скрытым через 1000 миллисекунд:

```
document.body.innerHTML = `
<div>
    <p>Foo</p>
</div>
`;

setTimeout(() => document.querySelector('div').attachShadow({ mode: 'open' }),
                 1000);
```

После подключения теневой DOM браузер отдает приоритет теневой DOM и отображает ее содержимое вместо текста. В этом примере теневая DOM пуста, поэтому в свою очередь `<div>` будет казаться пустым.

Чтобы показать этот контент, можно использовать тег `<slot>`, чтобы указать, где браузер должен разместить HTML. В следующем коде предыдущий пример переработан, поэтому текст снова появляется внутри теневой DOM:

```
document.body.innerHTML = `
<div id="foo">
    <p>Foo</p>
</div>
`;

document.querySelector('div')
    .attachShadow({ mode: 'open' })
    .innerHTML = `<div id="bar">
                 <slot></slot>
                 <div>`
```

Теперь проецируемое содержимое будет вести себя так, как будто оно существует внутри теневой DOM. Изучение страницы показывает, что содержимое действительно заменяет `<slot>`:

```
<body>
    <div id="foo">
        #shadow-root (open)
        <div id="bar">
            <p>Foo</p>
        </div>
    </div>
</body>
```

Обратите внимание, что несмотря на появление элемента в инструменте анализа страниц, это только *проекция* содержимого DOM. Элемент остается подключенным к внешнему DOM:

```
document.body.innerHTML = `
    <div id="foo">
        <p>Foo</p>
    </div>
`;

document.querySelector('div')
    .attachShadow({ mode: 'open' })
    .innerHTML = `
        <div id="bar">
            <slot></slot>
        </div>`

console.log(document.querySelector('p').parentElement);
// <div id="foo"></div>
```

Красно-зелено-синий пример, приведенный ранее, может быть переписан для использования слотов следующим образом:

```
for (let color of ['red', 'green', 'blue']) {
    const divElement = document.createElement('div');
    divElement.innerText = `Make me ${color}`;
    document.body.appendChild(divElement)

    divElement
        .attachShadow({ mode: 'open' })
        .innerHTML = `
        <p><slot></slot></p>

        <style>
            p {
                color: ${color};
            }
        </style>
    `;
}
```

Также можно использовать *именованные слоты* для создания нескольких проекций. Это достигается с помощью соответствующих пар атрибутов `slot`/`name`. Элемент, обозначенный атрибутом `slot="foo"`, будет проецироваться в `<slot>` с `name="foo"`.

Следующий пример демонстрирует это путем переключения порядка отрисовки дочерних элементов теневого узла:

```
document.body.innerHTML = `
    <div>
        <p slot="foo">Foo</p>
        <p slot="bar">Bar</p>
    </div>
`;

document.querySelector('div')
    .attachShadow({ mode: 'open' })
    .innerHTML = `
    <slot name="bar"></slot>
    <slot name="foo"></slot>
    `;

// Renders:
// Bar
// Foo
```

Перенацеливание событий

Если событие браузера, такое как `click`, происходит внутри теневой DOM, браузеру нужен способ, чтобы родительский DOM обработал это событие. Однако реализация должна также учитывать границу теневой DOM. Чтобы решить эту проблему, события, которые избегают теневой DOM и обрабатываются снаружи, подвергаются *перенацеливанию*. После экранирования это событие, по-видимому, было сгенерировано самим теневым хостом вместо истинно инкапсулированного элемента. Такое поведение демонстрируется здесь:

```
// Создание элемента для теневого хоста
document.body.innerHTML = `
<div onclick="console.log('Handled outside:', event.target)"></div>
`;

// Прикрепление теневой DOM и добавление в нее HTML
document.querySelector('div')
    .attachShadow({ mode: 'open' })
    .innerHTML = `
        <button onclick="console.log('Handled inside:', event.target)">Foo</button>
    `;

// При нажатии кнопки:
// Handled inside: <button onclick="..."></button>
// Handled outside: <div onclick="..."></div>
```

Обратите внимание, что перенацеливание происходит только для элементов, которые действительно существуют внутри теневой DOM. Для элементов, спроецированных извне с использованием тега <slot>, события не будут перенацелены, поскольку они все еще технически существуют вне теневой DOM.

Пользовательские элементы

Если вы использовали JavaScript-фреймворк, то, вероятно, знакомы с концепцией пользовательских элементов, поскольку все основные фреймворки предоставляют эту функцию в той или иной форме. Пользовательские элементы привносят вкус объектного программирования в элементы HTML. С их помощью можно создавать пользовательские, сложные и повторно используемые элементы и создавать экземпляры с помощью простого HTML-тега или атрибута.

Определение пользовательского элемента

Браузеры уже попытаются включить нераспознанные элементы в DOM как общие элементы. Конечно, по умолчанию они не делают ничего особенного, чего не делал бы обычный элемент HTML. Рассмотрим следующий пример, где бессмысленный HTML-тег становится экземпляром `HTMLElement`:

```
document.body.innerHTML = `
<x-foo >I'm inside a nonsense element.</x-foo >
`;

console.log(document.querySelector('x-foo') instanceof HTMLElement); // true
```

Пользовательские элементы шагают дальше. Они позволяют определять сложное поведение всякий раз, когда появляется тег `<x-foo>`, а также подключаться к жизненному циклу элемента относительно DOM. Определение пользовательского элемента выполняется с помощью глобального свойства `customElements`, которое возвращает объект `CustomElementRegistry`.

```
console.log(customElements);    // CustomElementRegistry {}
```

Определение пользовательского элемента выполняется с помощью метода `define()`. Следующий код создает простой пользовательский элемент, который наследуется от обычного HTMLElement:

```
class FooElement extends HTMLElement {}
customElements.define('x-foo', FooElement);

document.body.innerHTML = `
<x-foo >I'm inside a nonsense element.</x-foo >
`;

console.log(document.querySelector('x-foo') instanceof FooElement); // true
```

ПРИМЕЧАНИЕ У пользовательских имен элементов должен быть хотя бы один дефис в имени, но строка имени не должна начинаться или заканчиваться дефисом, а тег элемента не должен закрываться самостоятельно.

Сила пользовательских элементов находится в определении класса. Например, теперь каждый экземпляр этого класса в DOM будет вызывать управляемый конструктор:

```
class FooElement extends HTMLElement {
    constructor() {
        super();
        console.log('x-foo')
    }
}
customElements.define('x-foo', FooElement);

document.body.innerHTML = `
    <x-foo></x-foo>
    <x-foo></x-foo>
    <x-foo></x-foo>
`;

// x-foo
// x-foo
// x-foo
```

> **ПРИМЕЧАНИЕ** `super()` всегда должен вызываться первым в конструкторе пользовательских элементов. Если элемент наследуется от `HTMLElement` или подобного без переопределения конструктора, вызывать `super()` не нужно, так как конструктор прототипа сделает это по умолчанию. Необходимость определить пользовательский элемент, который не должен наследоваться от `HTMLElement`, появляется очень редко.

Если пользовательский элемент наследует от класса элемента, тег можно указать как экземпляр этого пользовательского элемента, используя атрибут `is` и параметр `extends`:

```
class FooElement extends HTMLDivElement {
    constructor() {
        super();
        console.log('x-foo')
    }
}
customElements.define('x-foo', FooElement, { extends: 'div' });

document.body.innerHTML = `
<div is="x-foo"></div>
<div is="x-foo"></div>
<div is="x-foo"></div>
`;

// x-foo
// x-foo
// x-foo
```

Добавление содержимого веб-компонентов

Поскольку конструктор класса пользовательского элемента вызывается каждый раз, когда элемент добавляется в DOM, легко заполнить пользовательский элемент содержимым дочерней DOM. Хотя запрещено добавлять дочерние элементы DOM в конструктор (исключение `DOMException`), можно прикрепить теневую DOM и поместить содержимое внутрь:

```
class FooElement extends HTMLElement {
    constructor() {
        super();

        // 'this' обращается к узлу веб-компонента
        this.attachShadow({ mode: 'open' });

        this.shadowRoot.innerHTML = `
            <p>I'm inside a custom element!</p>
        `;
    }
}
customElements.define('x-foo', FooElement);

document.body.innerHTML += `<x-foo></x-foo`;

// DOM в результате:
// <body>
// <x-foo>
//    #shadow-root (open)
//        <p>I'm inside a custom element!</p>
// <x-foo>
// </body>
```

Во избежание скверности строковых шаблонов и `innerHTML` этот пример можно реорганизовать для использования HTML-шаблонов и `document.createElement()`:

```
// (Начальный HTML)
// <template id="x-foo-tpl">
//    <p>I'm inside a custom element template!</p>
// </template>

const template = document.querySelector('#x-foo-tpl');

class FooElement extends HTMLElement {
    constructor() {
        super();

        this.attachShadow({ mode: 'open' });

        this.shadowRoot.appendChild(template.content.cloneNode(true));
    }
}
customElements.define('x-foo', FooElement);

document.body.innerHTML += `<x-foo></x-foo`;
```

```
// DOM в результате:
// <body>
// <template id="x-foo-tpl">
//    <p>I'm inside a custom element template!</p>
// </template>
// <x-foo>
//    #shadow-root (open)
//        <p>I'm inside a custom element template!</p>
// <x-foo>
// </body>
```

Эта практика обеспечивает высокую степень повторного использования HTML и кода, а также инкапсуляцию DOM внутри пользовательского элемента. С его помощью можно создавать многократно используемые виджеты, не опасаясь, что внешний CSS может испортить заданный стиль.

Использование ловушек жизненного цикла пользовательских элементов

Возможно выполнение кода в различных точках жизненного цикла пользовательского элемента. Методы экземпляра в классе пользовательских элементов с соответствующим именем будут вызываться на этом этапе жизненного цикла.

Существует пять доступных ловушек:

- `constructor()` вызывается при создании экземпляра элемента или при обновлении существующего элемента DOM до пользовательского элемента.
- `connectedCallback()` вызывается каждый раз, когда этот экземпляр пользовательского элемента добавляется в DOM.
- `disconnectedCallback()` вызывается каждый раз, когда этот экземпляр пользовательского элемента удаляется из DOM.
- `attributeChangedCallback()` вызывается каждый раз, когда изменяется значение *наблюдаемого атрибута*. При создании экземпляра элемента определение начального значения считается изменением.
- `acceptCallback()` вызывается каждый раз, когда этот экземпляр перемещается в новый объект документа с помощью `document.adoptNode()`.

В следующем примере демонстрируются обратные вызовы при срабатывании событий создания, подключения и отключения элемента:

```
class FooElement extends HTMLElement {
    constructor() {
        super();
        console.log('ctor');
    }

    connectedCallback() {
        console.log('connected');
    }
```

```
    disconnectedCallback() {
        console.log('disconnected');
    }
}

customElements.define('x-foo', FooElement);

const fooElement = document.createElement('x-foo');
// ctor

document.body.appendChild(fooElement);
// connected

document.body.removeChild(fooElement);
// disconnected
```

Отражение атрибутов пользовательского элемента

Поскольку элемент существует и как объект DOM, и как объект JavaScript, общий шаблон должен отражать изменения между ними. Другими словами, изменение в DOM должно отражать изменение в объекте, и наоборот. Для отражения от объекта к DOM распространенной стратегией является использование методов чтения и записи свойств. В следующем примере отражается свойство `bar` от объекта к DOM:

```
document.body.innerHTML = `<x-foo></x-foo>`;

class FooElement extends HTMLElement {
    constructor() {
        super();

        this.bar = true;
    }

    get bar() {
        return this.getAttribute('bar');
    }

    set bar(value) {
        this.setAttribute('bar', value)
    }
}
customElements.define('x-foo', FooElement);
console.log(document.body.innerHTML);
// <x-foo bar="true"></x-foo>
```

Отражение в обратном направлении — от DOM к объекту — требует установки наблюдателя для этого атрибута. Для этого можно указать пользовательскому элементу вызывать `attributeChangedCallback()` каждый раз, когда значение атрибута изменяется, используя метод чтения наблюдаемого атрибута `observedAttributes()`:

```
class FooElement extends HTMLElement {
    static get observedAttributes() {
        // Перечисление атрибутов, изменение которых должно
        // вызывать attributeChangedCallback()
        return ['bar'];
```

```
    }

    get bar() {
        return this.getAttribute('bar');
    }

    set bar(value) {
        this.setAttribute('bar', value)
    }

    attributeChangedCallback(name, oldValue, newValue) {
        if (oldValue !== newValue) {
            console.log(`${oldValue} -> ${newValue}`);

            this[name] = newValue;
        }
    }
}
customElements.define('x-foo', FooElement);

document.body.innerHTML = `<x-foo bar="false"></x-foo>`;
// null -> false

document.querySelector('x-foo').setAttribute('bar', true);
// false -> true
```

Обновление пользовательских элементов

Не всегда есть возможность определить пользовательский элемент до появления его тега в DOM. Веб-компоненты решают эту проблему упорядочения, предоставляя несколько дополнительных методов в `CustomElementRegistry`, которые позволяют определить, когда пользовательский элемент в конечном итоге будет определен, и обновить существующие элементы.

Метод `CustomElementRegistry.get()` возвращает класс пользовательского элемента, если он уже был определен. Аналогичным образом метод `CustomElementRegistry.whenDefined()` возвращает обещание, которое вычисляется при определении пользовательского элемента:

```
customElements.whenDefined('x-foo').then(() => console.log('defined!'));

console.log(customElements.get('x-foo'));
// undefined

customElements.define('x-foo', class {});
// определен!

console.log(customElements.get('x-foo'));
// class FooElement {}
```

Элементы, подключенные к DOM, будут *автоматически* обновлены после определения пользовательского элемента. Если необходимо принудительно обновить элемент до его подключения к DOM, это можно сделать с помощью `CustomElementRegistry.upgrade()`:

```
// Создание объекта HTMLUnknownElement до определения пользовательского элемента
const fooElement = document.createElement('x-foo');

// Определение пользовательского элемента
class FooElement extends HTMLElement {}
customElements.define('x-foo', FooElement);

console.log(fooElement instanceof FooElement);    // false

// Принудительное обновление
customElements.upgrade(fooElement);

console.log(fooElement instanceof FooElement);    // true
```

ПРИМЕЧАНИЕ Существует также веб-компонент импорта HTML, но его спецификация остается рабочим проектом, и ни один из основных браузеров не поддерживает его. Остается неясным, добавят ли какие-либо браузеры поддержку этого веб-компонента.

WEB CRYPTOGRAPHY API

Web Cryptography API (https://www.w3.org/TR/WebCryptoAPI) описывает набор инструментов криптографии, которые стандартизируют то, как JavaScript может использовать криптографическое поведение безопасным и идиоматическим образом. Эти инструменты включают в себя возможность генерировать, использовать и применять пары криптографических ключей; шифровать и дешифровать сообщения; надежно генерировать случайные числа.

ПРИМЕЧАНИЕ Организация криптографии несколько странная – с внешним объектом `Crypto` и внутренним объектом `SubtleCrypto`. До стандартизации API веб-криптографии свойство `window.crypto` было сильно фрагментировано во всех браузерах. Чтобы обеспечить кросс-браузерную совместимость, основная часть API предоставляется объекту `SubtleCrypto`.

Генерация случайных чисел

Когда задается задача генерации случайных значений, большинство разработчиков используют `Math.random()`. Этот метод реализован в браузерах как *генератор псевдослучайных чисел* (pseudorandom number generator, PRNG). Обозначение «псевдо» связано с природой генерации в том смысле, что она не является действительно случайной. Значения, получаемые из PRNG, только *эмулируют* свойства, связанные со случайностью. Это появление случайности стало возможным благодаря некоторой умной инженерии. PRNG браузера не использует никаких реальных источников случайности — это чисто фиксированный алгоритм, применяемый к герметичному внутреннему состоянию. Каждый раз, когда вызывается `Math.random()`, алгоритм

изменяет внутреннее состояние, а результат преобразуется в новое случайное значение. Например, механизм V8 использует алгоритм под названием xorshift128+ для выполнения этого изменения.

Поскольку этот алгоритм является фиксированным и его входом является *только* предыдущее состояние, порядок случайных чисел является детерминированным. xorshift128+ использует 128 бит внутреннего состояния, и алгоритм спроектирован таким образом, что любое начальное состояние будет создавать последовательность из $2^{128}-1$ псевдослучайных значений перед повторением. Такое циклическое поведение называется *циклом перестановок*, а продолжительность этого цикла называется *периодом*. Последствия этого очевидны: если злоумышленник знает внутреннее состояние PRNG, он может предсказать псевдослучайные значения, которые он впоследствии сгенерирует. Если ничего не подозревающий разработчик должен использовать PRNG для генерации закрытого ключа в целях шифрования, злоумышленник может использовать свойства PRNG для определения закрытого ключа.

Генераторы псевдослучайных чисел предназначены для быстрого вычисления значений, которые кажутся случайными. Однако они не подходят для целей криптографических вычислений. Чтобы решить эту проблему, *криптографически безопасный генератор псевдослучайных чисел* (cryptographically secure pseudorandom number generator, CSPRNG) дополнительно включает в себя источник энтропии в качестве входных данных, такой как измерение аппаратных таймингов или других системных свойств, которые проявляют непредсказуемое поведение. Это намного медленнее, чем обычный PRNG, но значения, генерируемые CSPRNG, являются достаточно непредсказуемыми для криптографических целей.

Web Cryptography API представляет CSPRNG, к которому можно получить доступ в глобальном объекте Crypto через `crypto.getRandomValues()`. В отличие от `Math.random()`, который возвращает число с плавающей точкой от 0 до 1, `getRandomValues()` записывает случайные числа в типизированный массив, предоставленный в качестве параметра. Класс типизированного массива неважен, поскольку основной буфер заполняется случайными двоичными битами.

Следующий пример генерирует пять 8-битных случайных значений:

```
const array = new Uint8Array(1);

for (let i=0; i<5; ++i) {
    console.log(crypto.getRandomValues(array));
}

// Uint8Array [41]
// Uint8Array [250]
// Uint8Array [51]
// Uint8Array [129]
// Uint8Array [35]
```

`getRandomValues()` будет генерировать до 216 байтов; выше этого значения он выдаст ошибку:

```
const fooArray = new Uint8Array(2 ** 16);
console.log(window.crypto.getRandomValues(fooArray));    // Uint32Array(16384)
[...]

const barArray = new Uint8Array((2 ** 16) + 1);
console.log(window.crypto.getRandomValues(barArray));    // Ошибка
```

Переопределение `Math.random()` с использованием CSPRNG может быть выполнено путем генерации одного случайного 32-битного числа и деления его на максимальное возможное значение, `0xFFFFFFFF`. Это позволяет получить значение от 0 до 1:

```
function randomFloat() {
    // Генерация 32 рандомных битов
    const fooArray = new Uint32Array(1);

    // Максимальное значение — 2^32 - 1
    const maxUint32 = 0xFFFFFFFF;

    // Деление на максимальное возможное значение
    return crypto.getRandomValues(fooArray)[0]    / maxUint32;
}
console.log(randomFloat());    // 0.5033651619458955
```

Использование объекта SubtleCrypto

Подавляющее большинство функционала Web Cryptography API находится внутри объекта `SubtleCrypto`, доступного через `window.crypto.subtle`:

```
console.log(crypto.subtle);    // SubtleCrypto {}
```

Этот объект содержит набор методов для выполнения общих криптографических функций, таких как шифрование, хеширование, подписывание и генерация ключей. Поскольку все криптографические операции выполняются с необработанными двоичными данными, каждый метод `SubtleCrypto` имеет дело с типами `ArrayBuffer` и `ArrayBufferView`. Из-за того, что строки так часто становятся предметом криптографических операций, классы `TextEncoder` и `TextDecoder` будут часто использоваться вместе с `SubtleCrypto` для преобразования в строки и обратно.

> **ПРИМЕЧАНИЕ** Объект `SubtleCrypto` доступен только в безопасном контексте (https). В небезопасном контексте свойство `subtle` будет иметь значение `undefined`.

Создание криптографических дайджестов

Одной из наиболее распространенных операций криптографии является расчет криптографического дайджеста данных. Спецификация поддерживает четыре алгоритма для этого — SHA-1 и три разновидности SHA-2:

- **Безопасный алгоритм хеширования 1 (SHA-1)** — хеш-функция с архитектурой, аналогичной MD5. Она принимает ввод любого размера и генерирует 160-битный

дайджест сообщения. Этот алгоритм больше не считается безопасным, поскольку он уязвим к атакам столкновений.

- **Безопасный алгоритм хеширования 2 (SHA-2)** — семейство хеш-функций, построенных на одной и той же функции одностороннего сжатия, устойчивой к столкновениям. Спецификация поддерживает три члена этого семейства: SHA-256, SHA-384 и SHA-512. Размер сгенерированного дайджеста сообщения может составлять 256 бит (SHA-256), 384 бит (SHA-384) или 512 бит (SHA-512). Этот алгоритм считается безопасным и широко используется во многих приложениях и протоколах, включая TLS, PGP и криптовалюты, такие как биткойн.

Метод `SubtleCrypto.digest()` используется для создания дайджеста сообщения. Алгоритм хеширования указывается с использованием строки: SHA-1, SHA-256, SHA-384 или SHA-512. В следующем примере демонстрируется простое применение SHA-256 для генерации дайджеста сообщения строки `foo`:

```
(async function() {
    const textEncoder = new TextEncoder();
    const message = textEncoder.encode('foo');
    const messageDigest = await crypto.subtle.digest('SHA-256', message);

    console.log(new Uint32Array(messageDigest));
})();

// Uint32Array(8) [1806968364, 2412183400, 1011194873, 876687389,
//                 1882014227, 2696905572, 2287897337, 2934400610]
```

Обычно двоичный файл дайджеста сообщения будет использоваться в шестнадцатеричном формате. Преобразование буфера массива в этот формат выполняется путем разделения буфера на 8-битные фрагменты и преобразования с использованием `toString()` в обычные 16-битные:

```
(async function() {
    const textEncoder = new TextEncoder();
    const message = textEncoder.encode('foo');
    const messageDigest = await crypto.subtle.digest('SHA-256', message);

    const hexDigest = Array.from(new Uint8Array(messageDigest))
        .map((x) => x.toString(16).padStart(2, '0'))
        .join('');

    console.log(hexDigest);
})();

// 2c26b46b68ffc68ff99b453c1d30413413422d706483bfa0f98a5e886266e7ae
```

Компании-разработчики программного обеспечения обычно публикуют дайджест своих двоичных файлов установки, чтобы люди, желающие безопасно установить их программное обеспечение, могли проверить, что загружаемый ими двоичный файл — это версия, которую фактически опубликовала компания (а не версия с внедренным вредоносным ПО). В следующем примере загружается Firefox v67.0, хешируется с SHA-512, загружается двоичная проверка SHA-512 и проверяется соответствие двух шестнадцатеричных строк:

```
(async function() {
    const mozillaCdnUrl = '//downloadorigin.cdn.mozilla.net/pub/firefox/
releases/67.0/';
    const firefoxBinaryFilename = 'linux-x86_64/en-US/firefox-67.0.tar.bz2';
    const firefoxShaFilename = 'SHA512SUMS';

    console.log('Fetching Firefox binary...');
    const fileArrayBuffer = await (await fetch(mozillaCdnUrl +
firefoxBinaryFilename))
        .arrayBuffer();

    console.log('Calculating Firefox digest...');
    const firefoxBinaryDigest = await crypto.subtle.digest('SHA-512',
fileArrayBuffer);

    const firefoxHexDigest = Array.from(new Uint8Array(firefoxBinaryDigest))
        .map((x) => x.toString(16).padStart(2, '0'))
        .join('');

    console.log('Fetching published binary digests...');
    // Файл SHA содержит дайджесты всех двоичных файлов firefox binary в этом
    // выпуске, поэтому здесь выполняется некоторая организация.
    const shaPairs = (await (await fetch(mozillaCdnUrl + firefoxShaFilename)).
text())
        .split(/\n/).map((x) => x.split(/\s+/));

    let verified = false;

    console.log('Checking calculated digest against published digests...');
    for (const [sha, filename] of shaPairs) {
        if (filename === firefoxBinaryFilename) {
            if (sha === firefoxHexDigest) {
                verified = true;
                break;
            }
        }
    }

    console.log('Verified:', verified);
})();

// Fetching Firefox binary...
// Calculating Firefox digest...
// Fetching published binary digests...
// Checking calculated digest against published digests...
// Verified: true
```

Криптоключи и алгоритмы

Криптография была бы бессмысленной без секретных ключей, и объект `SubtleCrypto` использует экземпляры класса `CryptoKey` для хранения этих секретов. Класс `CryptoKey` поддерживает несколько типов алгоритмов шифрования и позволяет контролировать извлечение и использование ключей.

Класс `CryptoKey` поддерживает следующие алгоритмы, классифицированные по их родительской криптосистеме:

- **RSA (Rivest-Shamir-Adleman)** — криптосистема с открытым ключом, в которой два больших простых числа используются для получения пары открытых и закрытых ключей, которые можно использовать подписывать/проверять или шифровать/дешифровать сообщения. Односторонняя функция с потайным входом для RSA называется *проблемой факторинга*.
- **RSASSA-PKCS1-v1_5** — приложение RSA, используемое для подписи сообщений закрытым ключом и проверки этой подписи открытым ключом.
  - SSA обозначает *схемы подписи с приложением*, указывая, что алгоритм поддерживает операции генерации и проверки подписи.
  - PKCS1 обозначает *стандарты криптографии с открытым ключом #1*, указывая, что алгоритм обладает математическими свойствами, которые должны иметь ключи RSA.
  - RSASSA-PKCS1-v1_5 является *детерминированным*, то есть одно и то же сообщение и ключ будут выдавать одинаковую подпись при каждом выполнении.
- **RSA-PSS** — еще одно приложение RSA, используемое для подписи и проверки сообщений.
  - PSS обозначает *вероятностную схему подписи*, указывая на то, что генерация подписи включает возможность для рандомизации подписи.
  - В отличие от RSASSA-PKCS1-v1_5, одно и то же сообщение и ключ будут выдавать разную подпись при каждом выполнении.
  - В отличие от RSASSA-PKCS1-v1_5, RSA-PSS доказуемо сводится к суровости проблемы факторинга RSA.
  - В целом, хотя RSASSA-PKCS1-v1_5 по-прежнему считается безопасным, RSA-PSS следует использовать в качестве замены RSASSA-PKCS1-v1_5.
- **RSA-OAEP** — приложение RSA, используемое для шифрования сообщений с открытым ключом и расшифровки их с помощью закрытого ключа.
  - OAEP обозначает «*Оптимальное асимметричное заполнение шифрования* (Optimal Asymmetric Encryption Padding)», указывая, что алгоритм использует сеть Фейстеля для обработки незашифрованного сообщения перед шифрованием.
  - OAEP служит для преобразования детерминированной схемы шифрования RSA в вероятностную схему шифрования.
- **ECC (криптография с эллиптической кривой)** — криптосистема с открытым ключом, в которой простое число и эллиптическая кривая используются для получения пары открытых и закрытых ключей, которые можно использовать для подписи/проверки сообщений. Односторонняя функция с потайным входом для ECC называется *задачей дискретного логарифма эллиптической кривой*. ECC считается превосходящим RSA: хотя и RSA, и ECC криптографически

надежны, ключи ECC короче, чем ключи RSA, а криптографические операции ECC быстрее, чем операции RSA.

- **ECDSA (алгоритм цифровой подписи эллиптической кривой)** — приложение ECC, используемое для подписи и проверки сообщений. Этот алгоритм представляет собой вариант *алгоритма цифровой подписи* (DSA, Digital Signature Algorithm) с примесью эллиптических кривых.
- **ECDH (эллиптическая кривая Диффи-Хеллмана)** — приложение ECC для генерации и согласования ключей, которое позволяет двум сторонам устанавливать общий секрет по общему каналу связи. Этот алгоритм представляет собой вариант протокола *обмена ключами Диффи-Хеллмана* (DH) с примесью эллиптических кривых.
- **AES (расширенный стандарт шифрования)** — криптосистема с симметричным ключом, которая шифрует/дешифрует данные с использованием блочного шифра, полученного из сети с подстановочной перестановкой. AES используется в разных *режимах*, которые меняют характеристики алгоритма.
- **AES-CTR** — *режим счетчика* AES. Этот режим ведет себя как потоковый шифр, используя счетчик приращения для генерации своего потока ключей. Также должен быть предоставлен одноразовый номер, который эффективно используется в качестве вектора инициализации. Шифрование/дешифрование AES-CTR может быть распараллелено.
- **AES-CBC** — *режим цепочки блоков шифрования* AES. Перед шифрованием каждого блока открытого текста он проводит операцию исключающего ИЛИ с предыдущим блоком зашифрованного текста — отсюда и название «цепочки». Вектор инициализации используется как вход с исключающим ИЛИ для первого блока.
- **AES-GCM** — *режим счетчика с аутентификацией Галуа* AES. Этот режим использует счетчик и вектор инициализации для генерации значения, которое выполняет операцию исключающего ИЛИ с открытым текстом каждого блока. В отличие от CBC, входы исключающего ИЛИ не зависят от шифрования предыдущего блока, и поэтому режим GCM может быть распараллелен. Из-за его превосходных эксплуатационных характеристик AES-GCM используется во многих сетевых протоколах безопасности.
- **AES-KW** — *режим оборачивания ключей* AES. Этот алгоритм оборачивает секретный ключ в переносимый и зашифрованный формат, который безопасен для передачи по ненадежному каналу. После передачи принимающая сторона может развернуть ключ. В отличие от других режимов AES, AES-KW не требует вектора инициализации.
- **HMAC (код аутентификации сообщений на основе хеша)** — алгоритм, генерирующий *коды аутентификации сообщений*, используемые для проверки того, что сообщение поступает без изменений при отправке по ненадежной сети. Две стороны используют хеш-функцию и общий закрытый ключ для подписи и проверки сообщений.
- **KDF (функции получения ключа)** — алгоритмы, которые могут получить один или несколько ключей из мастер-ключа, используя хеш-функцию. KDF

способны генерировать ключи различной длины или преобразовывать ключи в разные форматы.

- **HKDF (основанная на HMAC функция получения ключа)** — функция получения ключа, предназначенная для использования с входом с высокой энтропией, таким как существующий ключ.
- **PBKDF2 (функция получения ключа на основе пароля 2)** — функция получения ключа, предназначенная для использования с входом с низкой энтропией, например строкой пароля.

> **ПРИМЕЧАНИЕ** `CryptoKey` поддерживает большое количество алгоритмов, но только некоторые алгоритмы применимы к некоторым методам `SubtleCrypto`. Обратитесь к спецификации для обзора того, какие алгоритмы поддерживаются для каких методов: https://www.w3.org/TR/WebCryptoAPI/#algorithm-overview.

Генерация криптоключей

Генерация случайного `CryptoKey` выполняется с помощью метода `SubtleCrypto.generateKey()`, который возвращает промис, разрешаемый в одном или нескольких экземплярах `CryptoKey`. В этот метод передается объект `params`, указывающий целевой алгоритм, логическое значение, указывающее, должен ли ключ извлекаться из объекта `CryptoKey`, и массив строк — `keyUsages`, указывающий, с какими методами `SubtleCrypto` можно использовать ключ.

Поскольку разные криптосистемы требуют разных входных данных для генерации ключей, объект params предоставляет необходимые входные данные для каждой криптосистемы:

- Криптосистема RSA использует объект `RsaHashedKeyGenParams`.
- Криптосистема ECC использует объект `EcKeyGenParams`.
- Криптосистема HMAC использует объект `HmacKeyGenParams`.
- Криптосистема AES использует объект `AesKeyGenParams`.

Объект `keyUsages` описывает, с какими алгоритмами можно использовать ключ. Ожидается хотя бы одна из следующих строк:

- `encrypt`
- `decrypt`
- `sign`
- `verify`
- `deriveKey`
- `deriveBits`
- `wrapKey`
- `unwrapKey`

Предположим, нужно сгенерировать симметричный ключ со следующими свойствами:

- Поддержка алгоритма AES-CTR.
- Длина ключа 128 бит.
- Не может быть извлечен из объекта CryptoKey.
- Может использоваться с методами `encrypt()` и `decrypt()`.

Генерация этого ключа может быть выполнена следующим образом:

```
(async function() {
    const params = {
        name: 'AES-CTR',
        length: 128
    };

    const keyUsages = ['encrypt', 'decrypt'];

    const key = await crypto.subtle.generateKey(params, false, keyUsages);

    console.log(key);
    // CryptoKey {type: "secret", extractable: true, algorithm: {...},
       usages: Array(2)}
})();
```

Предположим, нужно сгенерировать пару асимметричных ключей со следующими свойствами:

- Поддержка алгоритма ECDSA.
- Использование эллиптической кривой P-256.
- Возможность извлечения из объекта `CryptoKey`.
- Возможность использования с методами `sign()` и `verify()`.

Генерация этого ключа может быть выполнена следующим образом:

```
(async function() {
    const params = {
        name: 'ECDSA',
        namedCurve: 'P-256'
    };

    const keyUsages = ['sign', 'verify'];

    const {publicKey, privateKey} = await crypto.subtle.generateKey(params, true,
 keyUsages);

    console.log(publicKey);
    // CryptoKey {type: "public", extractable: true, algorithm: {...},
       usages: Array(1)}

    console.log(privateKey);
    // CryptoKey {type: "private", extractable: true, algorithm: {...},
       usages: Array(1)}
})();
```

Экспорт и импорт ключей

Если ключ является извлекаемым, можно открыть простой двоичный файл ключа изнутри объекта `CryptoKey`. Метод `exportKey()` позволяет сделать это, одновременно указав целевой формат (`raw`, `pkcs8`, `spki` или `jwk`). Метод возвращает промис, который разрешается в `ArrayBuffer`, содержащий ключ:

```
(async function() {
    const params = {
        name: 'AES-CTR',
        length: 128
    };
    const keyUsages = ['encrypt', 'decrypt'];
    const key = await crypto.subtle.generateKey(params, true, keyUsages);

    const rawKey = await crypto.subtle.exportKey('raw', key);

    console.log(new Uint8Array(rawKey));
    // Uint8Array[93, 122, 66, 135, 144, 182, 119, 196, 234, 73, 84, 7, 139, 43, 238,
    // 110]
})();
```

Обратной операцией `exportKey()` является `importKey()`. Сигнатура этого метода по сути является комбинацией `generateKey()` и `exportKey()`. Следующий метод генерирует ключ, экспортирует его и импортирует еще раз:

```
(async function() {
    const params = {
        name: 'AES-CTR',
        length: 128
    };
    const keyUsages = ['encrypt', 'decrypt'];
    const keyFormat = 'raw';
    const isExtractable = true;
    const key = await crypto.subtle.generateKey(params, isExtractable, keyUsages);

    const rawKey = await crypto.subtle.exportKey(keyFormat, key);

    const importedKey = await crypto.subtle.importKey(keyFormat, rawKey,
          params.name, isExtractable, keyUsages);

    console.log(importedKey);
    // CryptoKey {type: "secret", extractable: true, algorithm: {...},
       usages: Array(2)}
})();
```

Получение ключей из мастер-ключей

Объект `SubtleCrypto` позволяет получить новые ключи с настраиваемыми свойствами из существующего секрета. Он поддерживает метод `deriveKey()`, который возвращает промис, разрешенный в `CryptoKey`, и метод `deriveBits()`, который возвращает промис, разрешенный в `ArrayBuffer`.

ПРИМЕЧАНИЕ Разница между `deriveKey()` и `deriveBits()` является тривиальной, так как вызов `deriveKey()` является практически таким же, как и вызов `deriveBits()` и передачи результата в `importKey()`.

Функция `deriveBits()` принимает объект параметров алгоритма, главный ключ и длину выходных данных в битах. Она может быть использована в ситуациях, когда два человека, каждый со своими собственными парами ключей, хотят получить общий секретный ключ. В следующем примере алгоритм ECDH используется для генерации обратных ключей из двух пар ключей и обеспечивает получение одинаковых битов ключа:

```
(async function() {
    const ellipticCurve = 'P-256';
    const algoIdentifier = 'ECDH';
    const derivedKeySize = 128;

    const params = {
        name: algoIdentifier,
        namedCurve: ellipticCurve
    };

    const keyUsages = ['deriveBits'];

    const keyPairA = await crypto.subtle.generateKey(params, true, keyUsages);
    const keyPairB = await crypto.subtle.generateKey(params, true, keyUsages);

    // Получение битов ключа из открытого ключа A и закрытого ключа B
    const derivedBitsAB = await crypto.subtle.deriveBits(
        Object.assign({ public: keyPairA.publicKey }, params),
        keyPairB.privateKey,
        derivedKeySize);

    // Получение битов ключа из открытого ключа B и закрытого ключа A
    const derivedBitsBA = await crypto.subtle.deriveBits(
        Object.assign({ public: keyPairB.publicKey }, params),
        keyPairA.privateKey,
        derivedKeySize);

    const arrayAB = new Uint32Array(derivedBitsAB);
    const arrayBA = new Uint32Array(derivedBitsBA);

    // Проверка идентичности массивов ключей
    console.log(
        arrayAB.length === arrayBA.length &&
        arrayAB.every((val, i) => val === arrayBA[i])); // true
})();
```

Метод `deriveKey()` ведет себя аналогично, возвращая экземпляр `CryptoKey` вместо `ArrayBuffer`. В следующем примере берется простая строка, применяется алгоритм PBKDF2 для импорта ее в простой главный ключ и выводится новый ключ в формате AES-GCM:

```
(async function() {
    const password = 'foobar';
    const salt = crypto.getRandomValues(new Uint8Array(16));
    const algoIdentifier = 'PBKDF2';
    const keyFormat = 'raw';
    const isExtractable = false;

    const params = {
        name: algoIdentifier
    };

    const masterKey = await window.crypto.subtle.importKey(
        keyFormat,
        (new TextEncoder()).encode(password),
        params,
        isExtractable,
        ['deriveKey']
    );

    const deriveParams = {
        name: 'AES-GCM',
        length: 128
    };

    const derivedKey = await window.crypto.subtle.deriveKey(
        Object.assign({salt, iterations: 1E5, hash: 'SHA-256'}, params),
        masterKey,
        deriveParams,
        isExtractable,
        ['encrypt']
    );

    console.log(derivedKey);
    // CryptoKey {type: "secret", extractable: false, algorithm: {...},
       usages: Array(1)}
})();
```

Подписание и проверка сообщений с помощью асимметричных ключей

Объект `SubtleCrypto` позволяет использовать алгоритмы с открытым ключом для генерации подписей с использованием закрытого ключа или для проверки подписей с использованием открытого ключа. Они выполняются с использованием методов `SubtleCrypto.sign()` и `SubtleCrypto.verify()` соответственно.

Для подписания сообщения требуется объект `params`, чтобы указать алгоритм и любые необходимые значения, частный `CryptoKey` и `ArrayBuffer` или `ArrayBufferView` для подписи. В следующем примере генерируется пара ключей с эллиптической кривой и используется закрытый ключ для подписи сообщения:

```
(async function() {
    const keyParams = {
        name: 'ECDSA',
```

```
        namedCurve: 'P-256'
    };

    const keyUsages = ['sign', 'verify'];

    const {publicKey, privateKey} = await crypto.subtle.generateKey(keyParams,
           true, keyUsages);

    const message = (new TextEncoder()).encode('I am Satoshi Nakamoto');

    const signParams = {
        name: 'ECDSA',
        hash: 'SHA-256'
    };

    const signature = await crypto.subtle.sign(signParams, privateKey, message);

    console.log(new Uint32Array(signature));
    // Uint32Array(16) [2202267297, 698413658, 1501924384, 691450316, 778757775,
       ... ]
})();
```

Человек, желающий проверить это сообщение по подписи, может использовать открытый ключ и метод `SubtleCrypto.verify()`. Подпись этого метода почти идентична `sign()`, за исключением того, что ему должны быть предоставлены как открытый ключ, так и подпись. Следующий пример расширяет предыдущий пример, проверяя сгенерированную подпись:

```
async function() {
    const keyParams = {
        name: 'ECDSA',
        namedCurve: 'P-256'
    };
    const keyUsages = ['sign', 'verify'];

    const {publicKey, privateKey} = await crypto.subtle.generateKey(keyParams,
           true, keyUsages);

    const message = (new TextEncoder()).encode('I am Satoshi Nakamoto');

    const signParams = {
        name: 'ECDSA',
        hash: 'SHA-256'
    };

    const signature = await crypto.subtle.sign(signParams, privateKey, message);

    const verified = await crypto.subtle.verify(signParams, publicKey, signature,
 message);

    console.log(verified);    // true
})();
```

Шифрование и дешифрование с помощью симметричных ключей

Объект `SubtleCrypto` позволяет использовать как открытый ключ, так и симметричные алгоритмы для шифрования и дешифрования сообщений. Это может быть выполнено с использованием методов `SubtleCrypto.encrypt()` и `SubtleCrypto.decrypt()` соответственно.

Для шифрования сообщения требуется объект `params`, указывающий алгоритм и любые необходимые значения, ключ шифрования и данные, которые должны быть зашифрованы. В следующем примере генерируется симметричный ключ AES-CBC, зашифровывается и, наконец, расшифровывается сообщение:

```
(async function() {
    const algoIdentifier = 'AES-CBC';

    const keyParams = {
        name: algoIdentifier,
        length: 256
    };

    const keyUsages = ['encrypt', 'decrypt'];

    const key = await crypto.subtle.generateKey(keyParams, true,
 keyUsages);

    const originalPlaintext = (new TextEncoder()).encode('I am Satoshi Nakamoto');

    const encryptDecryptParams = {
        name: algoIdentifier,
        iv: crypto.getRandomValues(new Uint8Array(16))
    };

    const ciphertext = await crypto.subtle.encrypt(encryptDecryptParams, key,
 originalPlaintext);

    console.log(ciphertext);
    // ArrayBuffer(32) {}

    const decryptedPlaintext = await crypto.subtle.decrypt(encryptDecryptParams,
                               key, ciphertext);

    console.log((new TextDecoder()).decode(decryptedPlaintext));
    // I am Satoshi Nakamoto
})();
```

Упаковка и распаковка ключа

Объект `SubtleCrypto` позволяет упаковывать и распаковывать ключи, чтобы обеспечить передачу по ненадежному каналу. Это выполняется с использованием методов `SubtleCrypto.wrapKey()` и `SubtleCrypto.unwrapKey()` соответственно.

Для переноса ключа требуется строка форматирования, экземпляр `CryptoKey` для переноса, `CryptoKey` для выполнения переноса и объект `params` для указания алгоритма и любых необходимых значений. В следующем примере генерируется симметричный ключ AES-GCM, упаковывается с помощью AES-KW и, наконец, распаковывается ключ:

```
(async function() {
    const keyFormat = 'raw';
    const extractable = true;

    const wrappingKeyAlgoIdentifier = 'AES-KW';
    const wrappingKeyUsages = ['wrapKey', 'unwrapKey'];
    const wrappingKeyParams = {
        name: wrappingKeyAlgoIdentifier,
        length: 256
    };

    const keyAlgoIdentifier = 'AES-GCM';
    const keyUsages = ['encrypt'];
    const keyParams = {
        name: keyAlgoIdentifier,
        length: 256
    };

    const wrappingKey = await crypto.subtle.generateKey(wrappingKeyParams,
                        extractable, wrappingKeyUsages);

    console.log(wrappingKey);
    // CryptoKey {type: "secret", extractable: true, algorithm: {...}, usages:
       Array(2)}

    const key = await crypto.subtle.generateKey(keyParams, extractable, keyUsages);

    console.log(key);
    // CryptoKey {type: "secret", extractable: true, algorithm: {...}, usages:
       Array(1)}

    const wrappedKey = await crypto.subtle.wrapKey(keyFormat, key, wrappingKey,
 wrappingKeyAlgoIdentifier);

    console.log(wrappedKey);
    // ArrayBuffer(40) {}

    const unwrappedKey = await crypto.subtle.unwrapKey(keyFormat, wrappedKey,
 wrappingKey, wrappingKeyParams, keyParams, extractable, keyUsages);

    console.log(unwrappedKey);
    // CryptoKey {type: "secret", extractable: true, algorithm: {...}, usages:
       Array(1)}
})()
```

ИТОГИ

HTML5, помимо определения новых правил разметки, также определяет несколько API JavaScript. Эти API разработаны для обеспечения лучших веб-интерфейсов, которые могут конкурировать с возможностями приложений для ПК.

Ниже приведены API-интерфейсы, описанные в этой главе:

- Atomics API позволяет защитить код от состояний гонки, возникающих в результате использования многопоточных шаблонов доступа к памяти.
- `postMessage()` API предоставляет возможность отправлять сообщения по документам из разных источников, сохраняя при этом безопасность политики одного и того же источника.
- Encoding API позволяет беспрепятственно преобразовывать строки и буферы — все более распространенный шаблон.
- File API предоставляет надежные инструменты для отправки, получения и чтения больших двоичных объектов.
- Медиаэлементы `<audio>` и `<video>` имеют свои собственные API для взаимодействия с аудио и видео. Не все форматы мультимедиа поддерживаются всеми браузерами, поэтому используйте метод `canPlayType()` для правильного определения поддержки браузера.
- Drag-and-Drop API позволяет вам легко указать, что элемент является перетаскиваемым и реагирует на перетаскивание так же, как операционная система. Можно создавать собственные перетаскиваемые элементы и целевые объекты.
- Notifications API предоставляет независимый от браузера способ представления интерактивных плиток для пользователя.
- Streams API предоставляет совершенно новый способ поэтапного чтения, записи и обработки данных.
- Timing API обеспечивает надежный способ измерения задержки в браузере и вокруг него.
- Web Components API представляет собой гигантский скачок вперед для повторного использования и инкапсуляции элементов.
- Web Cryptography API выполняет криптографические операции, такие как генерация случайных чисел, шифрование и подпись сообщений объектами первого класса.

21

Обработка ошибок и отладка

- Уведомления об ошибках
- Обработка ошибок
- Отладка JS-кода

Загрузки с сайта wrox.com для этой главы

Обратите внимание, что все примеры кода для этой главы доступны на веб-сайте книги по ссылке www.wrox.com/go/projavascript4e на вкладке Downloads.

Из-за динамической типизации языка и многолетнего отсутствия полноценных средств разработки отладка JS-кода традиционно требовала немалых усилий. Когда возникали ошибки, браузеры обычно отделывались туманными сообщениями, и хорошо, если при этом удавалось получить хоть какие-то сведения о контексте проблемы. Чтобы помочь разработчикам обрабатывать ошибки при их возникновении, в третью редакцию ECMAScript были добавлены инструкции `try-catch` и `throw`, а также разные типы ошибок. Несколькими годами позже начали появляться отладчики и отладочные средства JavaScript для браузеров, и к 2008 г. большинство браузеров поддерживали те или иные возможности отладки JS-кода.

Благодаря адекватной поддержке со стороны языка и средствам разработки теперь можно правильно обрабатывать ошибки и эффективно искать источники проблем.

УВЕДОМЛЕНИЯ ОБ ОШИБКАХ

Все основные веб-браузеры для ПК — Internet Explorer/Edge, Firefox, Safari, Chrome и Opera — поддерживают тот или иной способ уведомления о JavaScript-ошибках. По умолчанию эта информация скрывается, потому что она полезна только разработчикам и потому что в этом состоит характер веб-страниц — выбрасывать ошибки во время нормальной работы.

Консоли браузеров для ПК

Все современные веб-браузеры для ПК выявляют ошибки через веб-консоль. Эти ошибки могут быть обнаружены в консоли инструментов разработчика. Все ранее упомянутые браузеры имеют общий путь доступа к веб-консоли. Возможно, самый простой способ просмотреть ошибки — щелкнуть правой кнопкой мыши веб-страницу, выбрать Проверка или Проверка элемента и перейти на вкладку Консоль.

Чтобы перейти непосредственно к консоли, разные операционные системы и браузеры поддерживают разные комбинации клавиш.

БРАУЗЕР	WINDOWS/LINUX	MAC
Chrome	Ctrl+Shift+J	Cmd+Opt+J
Firefox	Ctrl+Shift+K	Cmd+Opt+K
Internet Explorer/Edge	F12, then Ctrl+2	NA
Opera	Ctrl+Shift+I	Cmd+Opt+I
Safari	NA	Cmd+Opt+C

Консоли мобильных браузеров

По сути, мобильные телефоны не будут предлагать консольный интерфейс непосредственно на устройстве. Тем не менее есть несколько вариантов, которые можно использовать в ситуациях, когда нужно проверить ошибки, возникающие на мобильном устройстве.

Chrome для мобильных устройств и Safari на iOS поставляются в комплекте с утилитами, которые позволяют подключать устройство к операционной системе хоста, на которой работает тот же браузер, и затем вы можете просматривать ошибки, возникающие в браузере сопряженного рабочего стола. Это включает физическое подключение устройства и следование инструкциям по установке для Chrome (https://developers.google.com/web/tools/chrome-devtools/remote-debugging/) или Safari (https://developer.apple.com/library/content/documentation/AppleApplications/Conceptual/Safari_Developer_Guide/GettingStarted/GettingStarted.html).

Также можно использовать стороннюю утилиту для отладки непосредственно на мобильном устройстве. Обычно используемой утилитой для отладки Firefox является

Firebug Lite (https://getfirebug.com/firebuglite_mobile). Это работает путем добавления скрипта Firebug на текущую веб-страницу с помощью букмарклета JavaScript. После запуска сценария можно будект открыть интерфейс отладки прямо в мобильном браузере. Firebug Lite также имеет версии для разных браузеров, к примеру, Chrome (http://getfirebug.com/releases/lite/chrome/).

ОБРАБОТКА ОШИБОК

Важность обработки ошибок в программировании несомненна. Каждое серьезное веб-приложение должно следовать строгому протоколу обработки ошибок, который обычно реализуют на стороне сервера. Как правило, разработчики серверной части приложения уделяют большое внимание механизму регистрации ошибок, реализуя в нем сортировку ошибок по типу, частоте возникновения и любым другим важным критериям. Это позволяет быстро проверить работу приложения, сгенерировав отчет или выполнив запрос к базе данных.

Обработка ошибок в браузере не менее важна, хотя на осознание этого потребовалось больше времени. Нельзя забывать, что большинство посетителей веб-сайтов мало что понимают в технологиях — многие из них даже не знают, что такое веб-браузер, не говоря уж о том, какой браузер они используют. Как уже отмечалось, браузеры реагируют на JavaScript-ошибки по-разному. Некоторые браузеры вообще не уведомляют пользователей об ошибках или имеют неудобный интерфейс для этого. Столкнувшись с проблемой, типичный пользователь в лучшем случае просто обновит страницу, а в худшем никогда больше не вернется на ваш веб-сайт. Грамотная стратегия обработки ошибок должна информировать пользователей о том, что происходит, не отпугивая их, но для реализации такой процедуры нужно уметь применять разные способы обработки JavaScript-ошибок по мере их возникновения.

Инструкция try-catch

В третьей редакции ECMA-262 для обработки исключений в JavaScript была представлена инструкция `try-catch` с таким же синтаксисом, как в Java:

```
try {
    // код, который может привести к ошибке
} catch (error) {
    // действия при возникновении ошибки
}
```

Любой код, который может привести к ошибке, следует помещать в раздел `try`, а код обработки ошибки — в раздел `catch`, например:

```
try {
    window.someNonexistentFunction();
} catch (error) {
    console.log("An error happened!"); // Оповещение об ошибке
}
```

Если в разделе `try` происходит ошибка, выполнение кода немедленно прекращается и возобновляется с раздела `catch`, в который передается объект со сведениями об ошибке. В отличие от других языков, вы должны назначить объекту ошибки имя, даже если не собираетесь его использовать. Содержимое этого объекта зависит от браузера, но включает как минимум свойство `message` с сообщением об ошибке. В ECMA-262 также определено свойство `name`, которое задает тип ошибки и доступно во всех современных браузерах. При необходимости можно показать фактическое сообщение браузера об ошибке:

```
try {
    window.someNonexistentFunction();
} catch (error) {
    console.log(error.message);
}
```

Этот код выводит в качестве сообщения об ошибке значение свойства `message`. Оно является единственным, которое гарантированно имеется в Internet Explorer, Firefox, Safari, Chrome и Opera, хотя каждый браузер добавляет к нему другие сведения. Internet Explorer добавляет свойство `description`, которое всегда совпадает с `message`, а также свойство `number`, содержащее внутренний номер ошибки. Firefox добавляет свойства `fileName`, `lineNumber` и `stack` (которое содержит трассу стека), а Safari — `line` (номер строки), `sourceId` (внутренний код ошибки) и `sourceURL`. В кроссбраузерном коде лучше использовать только свойство `message`.

Предложение finally

Необязательное предложение `finally` инструкции `try-catch` выполняется после нее в любом случае независимо от того, возникла ли ошибка. Ничто в разделе `try` или `catch`, даже инструкция `return`, не может предотвратить выполнение кода в разделе `finally`. Рассмотрим следующую функцию:

```
function testFinally() {
    try {
        return 2;
    } catch (error) {
        return 1;
    } finally {
        return 0;
    }
}
```

Эта функция содержит только инструкцию `try-catch`, каждый раздел которой возвращает число. На первый взгляд, функция должна возвратить значение 2, потому что в разделе `try` нет никаких ошибок. Однако из-за наличия предложения `finally` эта инструкция `return` игнорируется, и функция всегда возвращает 0. Если удалить предложение `finally`, функция возвратит 2.

При наличии предложения `finally` раздел `catch` не обязателен (достаточно чего-то одного).

ПРИМЕЧАНИЕ Не забывайте, что при наличии предложения `finally` любые инструкции `return` в разделах `try` и `catch` игнорируются. Проверяйте такие фрагменты кода с особым вниманием.

Типы ошибок

JavaScript-ошибки делятся на несколько категорий, каждой из которых соответствует особый объект, генерируемый при ошибке. В ECMA-262 определены семь типов ошибок:

- `Error`;
- `InternalError`;
- `EvalError`;
- `RangeError`;
- `ReferenceError`;
- `SyntaxError`;
- `TypeError`;
- `URIError`;

Тип `Error` — это базовый тип, от которого наследуются все остальные типы ошибок. Таким образом, все они имеют набор общих свойств (что касается методов, то у них есть только стандартные методы объекта). Ошибки типа `Error` почти никогда не генерируются браузерами; этот тип предоставляется в основном для того, чтобы разработчики могли генерировать собственные пользовательские ошибки.

Тип `InternalError` генерируется, когда исключение генерирует базовый движок JavaScript, например, когда переполнение стека происходит из-за слишком большой рекурсии. Это не тот тип ошибки, который был бы явно обработан внутри кода; если выдается эта ошибка, очень велики шансы, что код делает что-то неправильное или опасное, и это должно быть исправлено.

Ошибка `EvalError` генерируется, когда возникает исключение при использовании функции `eval()`. В ECMA-262 сказано, что она генерируется, «если значение свойства `eval` используется каким-либо образом, отличным от непосредственного вызова (то есть отличным от явного использования имени в качестве идентификатора, который представляет собой `MemberExpression` в `CallExpression`), или если свойству `eval` присваивается значение». По сути, речь идет о ситуациях, когда `eval()` используется не как вызов функции, например:

```
new eval();        // генерирует ошибку EvalError
eval = foo;        // генерирует ошибку EvalError
```

На практике браузеры генерируют ошибку `EvalError` не всегда, когда того требует спецификация. Например, в Firefox 4+ и Internet Explorer 8 в первом случае возникает ошибка `TypeError`, а вторая строка выполняется без ошибок. По этой причине

и из-за того, что такой код используется редко, едва ли вы столкнетесь с ошибками данного типа.

Ошибка `RangeError` генерируется, если число не попадает в диапазон. Например, она может возникнуть при попытке определить массив с недопустимым количеством элементов, таким как –20 или `Number.MAX_VALUE`:

```
let items1 = new Array(-20);                  // генерирует ошибку RangeError
let  items2 = new Array(Number.MAX_VALUE);    // генерирует ошибку RangeError
```

Ошибки диапазонов редко встречаются в JavaScript.

Ошибка `ReferenceError` генерируется, если не удается получить ожидаемый объект (это причина знаменитой ошибки `"object expected"`). Она обычно возникает при попытке доступа к несуществующей переменной, например:

```
let obj = x;    // генерирует ошибку ReferenceError, если
                // переменная x не объявлена
```

Наиболее частая причина ошибки `SyntaxError` — передача строки кода с неправильным синтаксисом в функцию `eval()`, например:

```
eval("a ++ b");    // генерирует ошибку SyntaxError
```

Вне функции `eval()` этот тип используется редко, потому что синтаксические ошибки в JS-коде немедленно останавливают его выполнение.

Чаще всего JavaScript-ошибки имеют тип `TypeError`. Такие ошибки возникают, если переменная имеет недопустимый тип или предпринимается попытка вызвать несуществующий метод. Наиболее частая причина этого — использование переменной неправильного типа в типизированной операции, например:

```
let o = new 10;                              // генерирует ошибку TypeError
console.log("name" in true);                 // генерирует ошибку TypeError
Function.prototype.toString.call("name");    // генерирует ошибку TypeError
```

Очень много ошибок типа `TypeError` возникает при использовании аргументов функций без предварительной проверки их типа.

Ошибки последнего типа, `URIError`, возникают только при использовании функции `encodeURI()` или `decodeURI()` с URI неправильного формата. Наверное, эти JavaScript-ошибки встречаются реже всего, потому что эти функции очень надежны.

В объектах ошибок разных типов можно передавать дополнительные сведения для дифференцированной обработки исключений. Определить тип ошибки в разделе `catch` можно с помощью оператора `instance`:

```
try {
    someFunction();
} catch (error) {
    if (error instanceof TypeError) {
        // обработка ошибок типа TypeError
    } else if (error instanceof ReferenceError) {
```

```
        // обработка ошибок типа ReferenceError
    } else {
        // обработка всех остальных ошибок
    }
}
```

Проверка типа ошибки — самый простой кроссбраузерный способ выбора алгоритма ее обработки. Свойство `message` плохо подходит для этого, потому что его значение зависит от браузера.

Использование инструкции try-catch

Если ошибка возникает в инструкции `try-catch`, браузер не уведомляет о ней, потому что считает, что она обрабатывается должным образом. Это идеально подходит для веб-приложений, ориентированных на пользователей без технической подготовки, которым не имеет смысла сообщать об ошибках. С помощью `try-catch` можно реализовать собственный механизм обработки ошибок конкретных типов.

Инструкцию `try-catch` лучше всего использовать, когда у вас нет контроля над возможными ошибками, например, если код включает вызов функции из JavaScript-библиотеки, которую вы не можете изменить. В этом случае уместно заключить вызов функции в блок `try` и обработать возможную ошибку в блоке `catch`.

Использовать инструкцию `try-catch` для обработки ошибок в собственном коде не следует. Например, если функция принимает число и выдает ошибку, когда получает вместо него строку, нужно просто проверить в ней тип данных аргумента. Инструкция `try-catch` в этой ситуации не требуется.

Генерирование ошибок

Инструкцию `try-catch` дополняет оператор `throw`, с помощью которого в любое время можно сгенерировать собственную ошибку. Он применяется со значением, но не налагает никаких ограничений на тип значения. Все следующие варианты допустимы:

```
throw 12345;
throw "Hello world!";
throw true;
throw { name: "JavaScript"};
```

Оператор `throw` немедленно останавливает выполнение кода, которое возобновляется только в том случае, если инструкция `try-catch` перехватывает сгенерированное им значение.

Ошибки браузера можно имитировать, используя один из встроенных типов. Конструктор каждого типа ошибки принимает как единственный аргумент сообщение об ошибке, например:

```
throw new Error("Something bad happened.");
```

Этот код генерирует обобщенную ошибку, но с нашим собственным сообщением. Она обрабатывается браузером, как если бы была сгенерирована им самим, то есть браузер уведомляет о ней обычным способом, выводя указанное сообщение. То же самое возможно и с ошибками других типов, например:

```
throw new SyntaxError("I don't like your syntax.");
throw new InternalError("I can't do that, Dave.");
throw new TypeError("What type of variable do you take me for?");
throw new RangeError("Sorry, you just don't have the range.");
throw new EvalError("That doesn't evaluate.");
throw new URIError("Uri, is that you?");
throw new ReferenceError("You didn't cite your references properly.");
```

Чаще всего разработчики используют собственные сообщения об ошибках `Error`, `RangeError`, `ReferenceError` и `TypeError`.

Вы также можете создавать собственные типы ошибок, наследуя их от типа `Error` (см. главу 6). Для собственного типа ошибки нужно задать свойства `name` и `message`, например:

```
class CustomError extends Error {
    constructor(message) {
        super(message);
        this.name = "CustomError";
           this.message = message;
    }
}

throw new CustomError("My message");
```

Ошибки собственных типов, унаследованных от типа `Error`, обрабатываются браузером, как и любые другие ошибки. Собственные типы ошибок полезны, если вам нужно отличать свои ошибки от ошибок, сгенерированных браузером.

Ситуации для генерирования ошибок

Генерирование собственных ошибок — прекрасный способ предоставить дополнительные сведения о том, почему функция завершилась сбоем. Ошибки следует генерировать, если известно, что при определенных условиях функция может выполняться неправильно. Например, следующая функция не сможет решить свою задачу, если получит аргумент, отличный от массива:

```
function process(values) {
    values.sort();

    for (let value of values) {
        if (value > 100) {
            return value;
        }
    }

    return -1;

}
```

Если передать этой функции строку, вызов `sort()` приведет к ошибке. Как указано далее, каждый браузер сообщит о ней по-своему.

- **Internet Explorer** — «property or method doesn't exist» (свойство или метод не существует).
- **Firefox** — «`values.sort()` is not a function» (`values.sort()` не является функцией).
- **Safari** — «value undefined (result of expression `values.sort`) is not an object» (неопределенное значение (результат выражения `values.sort`) не является объектом).
- **Chrome** — «object name has no method `'sort'`» (у объекта нет метода `'sort'`).
- **Opera** — «type mismatch (usually a non-object value used where an object is required)» (несоответствие типов (скорее всего, там, где требуется объект, используется значение, отличное от объекта)).

Хотя Firefox, Chrome и Safari указывают, в каком месте кода произошла ошибка, ни одно из этих сообщений не описывает достаточно ясно, что случилось или как можно устранить проблему. Если нужно отладить всего одну функцию, как в этом примере, это не беда, но при работе над сложными веб-приложениями с тысячами строк кода выяснить причину ошибки по таким сообщениям гораздо труднее.

В подобной ситуации генерирование собственной ошибки с точным описанием проблемы может значительно упростить обслуживание кода, например:

```
function process(values) {

    if (!(values instanceof Array)) {
        throw new Error("process(): Argument must be an array.");
    }

    values.sort();

    for (let value of values){
        if (value > 100) {
            return value;
        }
    }

    return -1;
}
```

Эта версия функции явно генерирует ошибку, если аргумент `values` не является массивом. Сообщение об ошибке содержит имя функции и ясно описывает проблему: «*Argument must be an array*» (аргумент должен быть массивом). Если бы эта ошибка произошла в сложном веб-приложении, вам было бы гораздо проще понять, что пошло не так.

При написании каждой JavaScript-функции нужно учитывать условия, в которых она может завершиться ошибкой. Грамотная организация обработки ошибок позволит вам работать только с теми ошибками, которые вы генерируете сами.

Генерирование и перехват ошибок

Часто спрашивают, когда следует генерировать ошибки, а когда перехватывать их с помощью инструкции `try-catch`. Вообще говоря, ошибки генерируются на низких уровнях архитектуры приложения, где мало известно о происходящем и адекватно обработать ошибку трудно. Если вы пишете JavaScript-библиотеку, предполагая, что ее будут использовать в самых разных приложениях, или вспомогательную функцию, которая будет вызываться в разных местах одного приложения, следует всерьез задуматься о генерировании ошибок с подробными описаниями проблем. Тогда при необходимости можно будет перехватывать ошибки в приложении и обрабатывать их должным образом.

Различие между генерированием и перехватом ошибок лучше всего выразить следующим образом: перехватывать ошибки следует, только если вы точно знаете, что делать дальше. Цель перехвата ошибки — блокировать действия, которые браузер выполняет по умолчанию. Цель генерирования ошибки — предоставить сведения о том, почему она произошла.

Событие error

Любая ошибка, не обработанная с помощью инструкции `try-catch`, генерирует событие `error` для объекта `window`. Это событие было реализовано в браузерах одним из первых, и ради обратной совместимости его формат был оставлен неизменным во всех современных браузерах. Вместо объекта `event` в обработчик события `error` передаются три аргумента: сообщение об ошибке, URL-адрес страницы и номер строки с ошибкой. В большинстве случаев важно только сообщение об ошибке, потому что URL-адрес совпадает с расположением документа, а номер строки может указывать место во встроенном JS-сценарии или во внешнем файле. Обработчик события `error` не соответствует стандартной спецификации DOM Level 2 Events, поэтому его нужно назначать, используя DOM Level 0:

```
window.onerror = (message, url, line) => {
     console.log(message);
};
```

Любая ошибка (независимо от того, генерирует ли ее браузер или JS-код) инициирует событие `error` и запускает этот обработчик. Затем браузер выполняет действия, предлагаемые по умолчанию, выводя сообщение об ошибке. Эти действия можно заблокировать, возвратив из обработчика значение `false`:

```
window.onerror = (message, url, line) => {
    console.log(message);
    return false;
};
```

По сути, возврат значения `false` превращает эту функцию в глобальную инструкцию `try-catch`, которая перехватывает в документе все необработанные ошибки времени выполнения. Такой обработчик — последнее средство блокирования уведомлений

браузера, которое в идеале никогда не должно использоваться. При грамотном применении инструкции `try-catch` никакие ошибки не должны достигать уровня браузера, а значит, и событие `error` никогда не будет генерироваться.

> **ПРИМЕЧАНИЕ** Браузеры обрабатывают событие `error` по-разному. Internet Explorer при ошибке продолжает обычное выполнение кода, при этом все переменные и данные доступны в обработчике `onerror`. В то же время в Firefox обычное выполнение кода при этом событии завершается, а все переменные и данные уничтожаются, что затрудняет анализ ошибки.

Изображения также поддерживают событие `error`, которое генерируется, если при доступе к атрибуту `src` не удается получить изображение в известном формате. Это событие соответствует формату DOM и возвращает в качестве целевого элемента событий объект `event` с изображением, например:

```
const image = new Image();

image.addEventListener("load", (event) => {
    console.log("Image loaded!");
});

image.addEventListener("error", (event) => {
    console.log("Image not loaded!");
});

image.src = "doesnotexist.gif";    // изображение не существует
```

Этот код выводит оповещение, если загрузка изображения завершается ошибкой. К тому времени, когда генерируется событие `error`, загрузка изображения уже завершена и возобновить ее нельзя.

Стратегии обработки ошибок

Стратегию обработки ошибок в веб-приложениях традиционно реализуют на сервере, при этом часто тратят много времени и усилий на разработку систем регистрации и мониторинга ошибок. Эти системы нужны для анализа паттернов возникновения ошибок, чтобы можно было отследить их причины и понять, как ошибки влияют на обслуживание пользователей.

Не менее важно продумать аналогичную стратегию и для уровня JS-сценариев. Любая ошибка в них может сделать работу с веб-страницей невозможной, поэтому необходимо ясно понимать, когда и почему возникают ошибки. Большинство пользователей веб-приложений не имеют технической подготовки и быстро теряются, если что-то не работает. По сути, все, что они могут сделать, — это перезагрузить страницу. Как разработчик вы должны хорошо понимать, где и как могут возникнуть сбои, и эффективно их отслеживать.

Идентификация потенциальных источников ошибок

Самое важное при обработке ошибок — это определить места, где они могут возникнуть. Поскольку JavaScript слабо типизирован и аргументы функций в нем не проверяются, ошибки часто обнаруживают себя только при выполнении кода. Можно выделить три категории ошибок:

- ошибки приведения типов;
- ошибки типов данных;
- ошибки обмена данными.

Любые из этих ошибок чаще всего возникают в специфичных ситуациях, которые можно отслеживать.

Статическое средство анализа кода

Было бы безответственно не упомянуть здесь, что подавляющее количество ошибок, с которыми вы можете столкнуться, может быть упреждающим образом обработано с помощью статического средства анализа кода как части процесса сборки приложения. Потрясающий список ресурсов представлен на https://gist.github.com/listochkin/6250151. Наиболее часто используемые средства анализа — это JSHint, JSLint, Google Closure и TypeScript.

Статические средства анализа кода требуют аннотировать ваш JavaScript типами, сигнатурами функций и другими директивами, которые описывают, как программа будет работать вне базового исполняемого кода. Средство анализа сравнит аннотации с различными частями кодовой базы JavaScript, которые используют друг друга, и расскажет вам о любых потенциальных несовместимостях, которые могут проявиться при реальном выполнении.

> **ПРИМЕЧАНИЕ** По мере роста кодовой базы вы обнаружите, что статические средства анализа становятся все более необходимыми, особенно с увеличением числа разработчиков, работающих над одним и тем же кодом. Все крупные технологические компании, которые поддерживают чрезвычайно большие кодовые базы JavaScript, используют надежную форму статического анализа как часть процессов сборки.

Ошибки приведения типов

Ошибки приведения типов (type coercion errors) возникают при использовании операторов или других конструкций языка, которые автоматически изменяют тип данных значения. Две наиболее частые ошибки приведения типов возникают при сравнении значений с помощью оператора равенства (==) или неравенства (!=)

и использовании неуместных значений в условиях управляющих инструкций, таких как `if`, `for` и `while`.

Операторы равенства и неравенства, которые были описаны в главе 3, перед сравнением автоматически преобразуют типы значений. Поскольку во многих нединамических языках такие же операторы выполняют обычное сравнение, разработчики часто ошибочно используют их в JavaScript аналогичным образом. Чтобы избежать приведения типов, вместо них лучше использовать операторы строгого равенства (`===`) и строгого неравенства (`!==`):

```
console.log(5 == "5");      // true
console.log(5 === "5");     // false
console.log(1 == true);     // true
console.log(1 === true);    // false
```

Этот код сначала сравнивает число 5 и строку `"5"`, используя операторы равенства и строгого равенства. Оператор равенства преобразует строку `"5"` в число 5, а затем сравнивает его с другим числом 5, что дает в результате `true`. Оператор строгого равенства учитывает, что типы двух значений различаются, и возвращает `false`. То же самое происходит со значениями 1 и `true`: оператор строгого равенства не признает их равными из-за разницы типов. Операторы строгого равенства и строгого неравенства предотвращают ошибки приведения типов, а потому предпочтительнее, чем обычные операторы равенства и неравенства.

> **ПРИМЕЧАНИЕ** Руководства по стилю кода часто утверждают, как следует применять `===` и `==`. Некоторые руководства поддерживают идею, что если для сравнения всегда используется `===`, приведение типов не является проблемой. Другие диктуют, что неуклонное использование `===` излишне во всех местах, кроме тех, в которых возможно приведение строки к логическому значению.

Ошибки приведения типов также часто возникают в управляющих инструкциях. Дело в том, что инструкции вроде `if` перед выбором следующего действия автоматически преобразуют любое значение в логическое, например:

```
function concat(str1, str2, str3) {
    let result = str1 + str2;
    if (str3) {    // не делайте так!!!
        result += str3;
    }
    return result;
}
```

Предполагаемое назначение этой функции — конкатенация двух или трех строк. Третья строка является необязательным аргументом, поэтому инструкция `if` проверяет, передан ли он в функцию. Как отмечалось в главе 3, если именованная переменная не используется, ей автоматически присваивается значение `undefined`. Ему соответствует логическое значение `false`, так что инструкция `if` присоединяет третий аргумент к строке, только если он задан. Проблема в том, что `undefined` — не единственное

значение, которое преобразуется в `false`, а строка — не единственное значение, которое преобразуется в `true`. Например, если третьим аргументом будет число 0, условие `if` окажется ложным, а если 1 — истинным.

Очень много ошибок возникает из-за того, что в условиях управляющих инструкций используются значения, типы которых отличаются от логического. Чтобы таких ошибок не было, условия должны иметь логический тип, чего можно добиться с помощью операторов равенства. Например, предыдущую функцию можно переписать следующим образом:

```
function concat(str1, str2, str3) {
    let result = str1 + str2;
    if (typeof str3 === "string") {    // правильное сравнение
        result += str3;
    }
    return result;
}
```

В этой версии функции условие `if` возвращает логическое значение. Эта функция гораздо безопаснее и меньше зависит от неправильных значений.

Ошибки типов данных

Поскольку JavaScript типизирован слабо, типы переменных и аргументов функций не проверяются автоматически — ответственность за это лежит на разработчике. Ошибки типов данных чаще всего возникают при передаче неправильных значений в функции.

В предыдущем примере проверяется тип данных третьего аргумента, но не первых двух. Если функция должна возвратить строку, то передача двух чисел без третьего аргумента вызовет ошибку. Похожая проблема имеет место в следующей функции:

```
// небезопасная функция – значение, отличное от строки, приводит к ошибке
function getQueryString(url) {
    const pos = url.indexOf("?");
    if (pos > -1) {
        return url.substring(pos +1);
    }
    return "";
}
```

Эта функция извлекает строку запроса из полученного URL-адреса. Для этого она сначала ищет в адресе вопросительный знак с помощью метода `indexOf()`, и если его удается обнаружить, возвращает часть адреса после него методом `substring()`. Два метода, которые используются в этом примере, специфичны для строк, так что передача значения любого другого типа в функцию приведет к ошибке. Следующая простая проверка типа делает функцию более устойчивой к ошибкам:

```
function getQueryString(url) {
    if (typeof url === "string") {    // более безопасный вариант
```

```
        let pos = url.indexOf("?");
        if (pos > -1) {
            return url.substring(pos +1);
        }
    }
    return "";
}
```

Эта версия функции первым делом проверяет, действительно ли в нее передана строка, что предотвращает ошибки при передаче значений других типов.

Вы уже знаете, что использование значений, отличных от логических, в условиях управляющих инструкций может приводить к ошибкам приведения типов. Это плохо еще и потому, что может вызывать ошибки типов данных. Рассмотрим следующую функцию:

```
// небезопасная функция – значение, отличное от массива, приводит к ошибке
function reverseSort(values) {
    if (values) {    // не делайте так!!!
        values.sort();
        values.reverse();
    }
}
```

Функция `reverseSort()` сортирует массив в обратном порядке, используя методы `sort()` и `reverse()`. К сожалению, при текущем условии инструкции `if` любое значение, которое не является массивом, но преобразуется в `true`, приведет к ошибке. Другой частой ошибкой является сравнение аргумента со значением `null`:

```
// небезопасная функция – значение, отличное от массива, приводит к ошибке
function reverseSort(values) {
    if (values != null) {    // не делайте так!!!
        values.sort();
        values.reverse();
    }
}
```

Сравнение аргумента с `null` защищает функцию только от значений `null` и `undefined` (что эквивалентно использованию операторов равенства и неравенства). Однако это не гарантирует, что значение приемлемо в других отношениях, поэтому использовать этот прием не следует. Сравнивать переменную со значением `undefined` не рекомендуется по той же причине.

Еще один источник ошибок — распознавание только одной из нужных возможностей, например:

```
// небезопасная функция – значение, отличное от массива, приводит к ошибке
function reverseSort(values) {
    if (typeof values.sort === "function") {    // не делайте так!!!
        values.sort();
        values.reverse();
    }
}
```

Этот код проверяет, есть ли у аргумента метод `sort()`, однако в функцию может быть передан объект, который содержит метод `sort()`, но не является массивом. В этом случае вызов `reverse()` приведет к ошибке. Если вы точно знаете, какого типа объект вам нужен, лучше использовать для его проверки оператор `instanceof`:

```
// безопасная функция – значение, отличное от массива, игнорируется
function reverseSort(values) {
    if (values instanceof Array) {    // исправлено
        values.sort();
        values.reverse();
    }
}
```

Эта версия функции `reverseSort()` безопасна. Она проверяет, является ли аргумент `values` экземпляром `Array`, и игнорирует любые значения, отличные от массива.

Вообще говоря, если значение должно быть примитивным типом, его следует проверять с помощью оператора `typeof`, а если объектом — с помощью оператора `instanceof`. В зависимости от использования функции проверка типа данных всех аргументов может быть излишней, но и пренебрегать ею не следует, особенно в общедоступных API.

Ошибки обмена данными

Благодаря Ajax (см. главу 21) веб-приложения смогли динамически загружать информацию или код в течение всего жизненного цикла, но любой обмен данными между браузером и сервером — это еще один потенциальный источник ошибок.

Ошибки обмена данными могут возникать из-за неправильного формата URL-адресов или отправляемых данных. Это обычно происходит, если данные перед отправкой серверу не кодируются с помощью метода `encodeURIComponent()`. Например, формат следующего URL-адреса недопустим:

```
http://www.yourdomain.com/?redir=http://www.someotherdomain.com?a=b&c=d
```

Этот URL-адрес можно исправить, вызвав метод `encodeURIComponent()` для части после `"redir="`. В результате получается следующее:

```
http://www.yourdomain.com/?redir=http%3A%2F%2Fwww.someotherdomain.
com%3Fa%3Db%26c%3Dd
```

Аргументы строки запроса всегда нужно обрабатывать методом `encodeURIComponent()`. Чтобы гарантировать это, можно создать для составления строки запроса специальную функцию:

```
function addQueryStringArg(url, name, value) {
    if (url.indexOf("?") == -1) {
        url += "?";
    } else {
        url += "&";
```

```
    }
    url += '${encodeURIComponent(name)=${encodeURIComponent(value)}';
    return url;
}
```

Эта функция принимает три аргумента: URL-адрес, к которому нужно добавить аргумент строки запроса, имя аргумента и его значение. Если в URL-адресе нет вопросительного знака, функция добавляет его; если он есть, это означает, что строка запроса уже содержит аргументы, так что добавляется амперсанд. После этого функция кодирует имя и значение аргумента и добавляет их к URL-адресу. Использовать ее можно следующим образом:

```
const url = "http://www.somedomain.com";
const newUrl = addQueryStringArg(url, "redir",
                                 "http://www.someotherdomain.com?a=b&c=d");
console.log(newUrl);
```

Использование этой функции вместо составления URL-адресов вручную обеспечивает правильное кодирование запросов, предотвращая возможные ошибки.

Ошибки обмена данными также происходят, если сервер возвращает неожиданный ответ. Например, при динамической загрузке сценариев или стилей запрошенный ресурс иногда оказывается недоступен. В этом случае одни браузеры не делают ничего, а другие возвращают ошибку. К сожалению, при использовании таких приемов узнать об ошибке почти невозможно, хотя иногда с помощью Ajax можно получить сведения о проблеме.

Различение критичных и некритичных ошибок

Любая стратегия обработки ошибок должна давать ответ на вопрос, критична ли та или иная ошибка. Для некритичной ошибки характерно следующее:

- она не мешает пользователю решать основные задачи;
- она затрагивает только часть страницы;
- восстановление возможно;
- повторение действия может решить проблему.

Некритичные ошибки не являются серьезным поводом для беспокойства. Например, в Gmail (https://mail.google.com) есть инструмент, позволяющий пользователям отправлять SMS-сообщения. Если по какой-либо причине он не работает, это некритичная ошибка, потому что отправка SMS не входит в число основных функций приложения Gmail. Оно предназначено для чтения и написания сообщений электронной почты, и пока пользователь может это делать, незачем его беспокоить. Некритичная ошибка — не повод отвлекать его от работы, хотя при желании вы можете заменить нерабочую область страницы каким-либо уведомлением.

В то же время для критичной ошибки характерно следующее:

- приложение не может продолжить работу;

- ошибка мешает пользователю решать основные задачи;
- ошибка влечет за собой другие ошибки.

Важно понимать, когда в JavaScript происходит критичная ошибка, чтобы можно было принять адекватные меры. Следует сразу же отправить пользователям уведомление об ошибке, чтобы дать им понять, что продолжение работы невозможно. Если для восстановления работоспособности приложения нужно перезагрузить страницу, следует сообщить об этом и предоставить кнопку перезагрузки.

Код не должен диктовать, что является, а что не является критичной ошибкой — серьезность ошибки зависит в основном от того, как она влияет на работу пользователя. В грамотно спроектированном приложении ошибка в одной части не должна влиять на части приложения, которые напрямую с ней не связаны. Например, персонализированная домашняя страница вроде Gmail (https://mail.google.com) может содержать много независимых модулей. Для инициализации JavaScript-модулей иногда используют такой код:

```
for (let mod of mods) {
    mod.init();        // возможна критичная ошибка
}
```

На первый взгляд, все в порядке: метод `init()` просто вызывается для каждого модуля. Проблема в том, что ошибка в методе `init()` любого из модулей помешает инициализировать все последующие модули в массиве. Если ошибка произойдет на первой же итерации, ни один модуль на странице не будет инициализирован. Это не имеет никакого смысла, потому что все модули не зависят друг от друга. Чтобы эти ошибки инициализации перестали быть критичными, можно переписать код следующим образом:

```
for (let mod of mods) {
    try {
        mod.init();
    } catch (ex) {
        // обработка ошибки
    }
}
```

Теперь ошибка в каком-либо из модулей не помешает инициализировать другие модули. Ее можно будет обработать отдельно, не прерывая работу пользователя.

Протоколирование ошибок на сервере

Для записи и отслеживания ошибок в веб-приложениях часто используют централизованный журнал. Протоколирование и систематизация ошибок базы данных и сервера в таком журнале с помощью некоего API — уже общепринятая практика, но в сложных веб-приложениях JavaScript-ошибки также следует протоколировать на сервере. Идея в том, чтобы сохранять сведения о них в той же системе, где хранятся ошибки серверной части приложения, но помечать, что они возникли на стороне клиента. Такой подход позволяет унифицировать анализ ошибок.

Для протоколирования JavaScript-ошибок на сервере нужна страница, которая будет принимать данные из строки запроса и сохранять их в журнале ошибок. Для отправки сведений об ошибке можно использовать следующую функцию:

```
function logError(sev, msg) {
    let img = new Image();
           encodedSev = encodeURIComponent(sev),
        encodedMsg = encodeURIComponent(msg);
    img.src = 'log.php?sev=${encodedSev}&msg=${encodedMsg}';
}
```

Функция `logError()` принимает индикатор серьезности ошибки и сообщение об ошибке. Серьезность ошибки можно обозначать самыми разными строками в зависимости от того, какую систему вы используете. Объект `Image` выбран для отправки запроса из следующих соображений:

- Он доступен во всех браузерах, даже тех, которые не поддерживают объект `XMLHttpRequest`.
- На него не распространяются ограничения взаимодействия между доменами. Если один сервер должен принимать сведения об ошибках от многих других серверов, объект `XMLHttpRequest` не будет работать.
- Это снижает вероятность сбоя во время протоколирования ошибки. Как правило, обмен данными по технологии Ajax осуществляется с помощью оболочек из JavaScript-библиотек. Если в коде библиотеки, которую вы пытаетесь использовать для протоколирования ошибки, случится сбой, сообщение об ошибке может быть утеряно.

Если в коде используется инструкция `try-catch`, вероятно, ошибку следует запротоколировать. Это можно сделать следующим образом:

```
for (let mod of mods) {
    try {
        mod.init();
    } catch (ex) {
        logError("nonfatal", 'Module init failed: ${ex.message}');
    }
}
```

При неудачной инициализации модуля здесь вызывается функция `logError()`. Для указания серьезности ошибки в функцию передается значение `"nonfatal"`, а ее второй аргумент состоит из описания контекста и подлинного сообщения о JavaScript-ошибке. В сообщения, протоколируемые на сервере, желательно добавлять максимально подробные сведения о контексте ошибки, чтобы было проще определить ее причину.

ПРИЕМЫ ОТЛАДКИ

Когда JavaScript-отладчиков еще не было, разработчики были вынуждены добавлять в свой код фрагменты, специально предназначенные для вывода отладочной

информации. Чаще всего для этого использовались оповещения с присущими им недостатками. Так, чтобы удалить их из кода после отладки, приходилось тратить драгоценное время, а оповещения, по недосмотру оставленные в коде, раздражали пользователей. К счастью, теперь доступны другие более элегантные решения.

Вывод сообщений на консоль

Во всех основных браузерах доступна JavaScript-консоль, которую можно использовать для просмотра ошибок, а сведения об ошибках можно выводить на JavaScript-консоль с использованием объекта `console`, у которого есть следующие методы:

- `error(`*сообщение*`)` — выводит на консоль сообщение об ошибке;
- `info(`*сообщение*`)` — выводит на консоль информационное сообщение;
- `log(`*сообщение*`)` — выводит на консоль сообщение общего характера;
- `warn(`*сообщение*`)` — выводит на консоль предупреждение.

Вид сообщения на консоли ошибок зависит от того, какой метод был использован для его вывода. Сообщения об ошибках содержат красный значок, а предупреждения — желтый. Выводить сообщения на консоль можно следующим образом:

```
function sum(num1, num2) {
    console.log('Entering sum(), arguments are ${num1},${num2}');

    console.log("Before calculation");
    const result = num1 + num2;
    console.log("After calculation");

    console.log("Exiting sum()");
    return result;
}
```

При вызове этой функции `sum()` в JavaScript-консоль выводятся несколько отладочных сообщений.

Регистрация сообщений в консоли JavaScript полезна при отладке кода, но все сообщения должны быть удалены, когда код будет переведен в окончательную версию. Это можно сделать автоматически, используя шаг обработки кода при развертывании, или вручную.

> **ПРИМЕЧАНИЕ** Ведение журнала сообщений считается лучшим способом отладки, чем использование предупреждений, поскольку оповещения прерывают выполнение программы, что может повлиять на результат кода из-за влияния времени асинхронных процессов. Ведение журнала также позволяет выводить произвольное количество аргументов и проверять экземпляры объекта (оповещение объекта сериализует его в строку перед оповещением, что приводит к тому, что часто показывается `Object [Object]` как предупреждающее сообщение).

Выполнение в консоли

Консоль браузера представляет собой REPL (read-eval-print loop, цикл чтения-вычисления-печати), который совпадает с выполнением JavaScript-страницы. Он ведет себя так же, как если бы браузер обрабатывал недавно обнаруженный тег `<script>` внутри DOM. Команды, выполняемые из консоли, могут обращаться к глобальным переменным и различным API так же, как и JavaScript на уровне страницы. Произвольное количество кода может быть вычислено в консоли; как и в случае с любым другим кодом уровня страницы, который он блокирует. Изменения, объекты и обратные вызовы будут сохраняться в DOM и/или в среде выполнения.

Среда выполнения JavaScript будет ограничивать доступ к различным окнам, и поэтому во всех основных браузерах можно выбрать, в каком окне должны выполняться входы консоли JavaScript. Исполняемый код не выполняется с повышенными привилегиями — он все еще подвержен ограничениям происхождения и любым другим элементам управления, которые применяются браузером.

Среда выполнения консоли также имеет интеграцию с инструментом разработчика, который предлагает несколько контекстных бонусных инструментов, помогающих в отладке, недоступных в обычном JavaScript. Одним из наиболее полезных инструментов является средство выбора с учетом последнего нажатия, которое доступно в той или иной форме во всех основных браузерах. На вкладке Элемент в инструментах разработчика при нажатии на узел в дереве DOM вы получаете ссылку на экземпляр JavaScript этого узла на вкладке Консоль, используя `$0`. Он ведет себя как обычный экземпляр JavaScript, поэтому чтение свойств, таких как `$0.scrollWidth`, и вызов методов-членов, таких как `$0.remove()`, разрешены.

Использование средства отладки JavaScript

Во всех основных браузерах также доступно средство отладки JavaScript. Как часть спецификации ECMAScript 5.1, ключевое слово `debugger` попытается вызвать любые доступные функции отладки. Если связанного поведения нет, этот оператор будет молча пропущен как неактивный. Выражение может быть использовано следующим образом:

```
function pauseExecution(){
    console.log("Will print before breakpoint");
    debugger;
    console.log("Will not print until breakpoint continues");
}
```

Когда среда выполнения встречает ключевое слово, во всех основных браузерах открывается панель инструментов разработчика с точкой останова, установленной именно в этом месте. После этого можно будет использовать отдельную консоль браузера, которая выполняет код в определенной лексической области, где в данный момент остановлена точка останова. Кроме того, вы сможете выполнять стандартные операции отладчика кода (шаг вперед, пропустить шаг, продолжить и т. д.).

Браузеры также обычно позволяют устанавливать точки останова вручную (без использования ключевого слова `debugger`), проверяя фактический загруженный код JavaScript в инструментах разработчика и выбирая строку, где нужно установить точку останова. Эта точка останова будет вести себя так же, но она не будет сохраняться в сеансах браузера.

Вывод сообщений на страницу

Другой популярный прием отладки — это вывод отладочных сообщений на страницу. Можно использовать для этого отдельную область, всегда доступную на странице, или создавать соответствующий элемент только при необходимости. Например, функцию `log()` можно изменить так:

```
function log(message) {
    // Лексическая область этой функции будет использовать следующий экземпляр
    // вместо window.console
    const console = document.getElementById("debuginfo");
    if (console === null) {
        console = document.createElement("div");
        console.id = "debuginfo";
        console.style.background = "#dedede";
        console.style.border = "1px solid silver";
        console.style.padding = "5px";
        console.style.width = "400px";
        console.style.position = "absolute";
        console.style.right = "0px";
        console.style.top = "0px";
        document.body.appendChild(console);
    }
    console.innerHTML += '<p> ${message}</p>';
}
```

Эта версия функции `log()` сначала проверяет, существует ли элемент для вывода отладочных сведений. Если нет, она создает элемент `<div>` и назначает ему специфический стиль, отличный от остальной страницы. После этого переданное в функцию сообщение присваивается свойству `innerHTML` элемента `<div>`. В результате оно выводится на странице в небольшой специальной области. Этот подход может быть полезен при отладке кода в Internet Explorer 7 и более ранних версий или в других браузерах, не поддерживающих JavaScript-консоль.

> **ПРИМЕЧАНИЕ** В окончательной программе отладочные данные не должны выводиться ни на консоль, ни на страницу.

Заменяющие методы консоли

Можно легко забыть о возможности использовать два разных типа операторов журнала — собственный `console.log()` и отдельно определенный пользователем `log()`. Поскольку `console` — это глобальный объект с доступными для записи

методами-членами, вполне возможно перезаписать его методы-члены с помощью специального поведения и позволить операторам журнала, разбросанным по всей базе кода, с радостью регистрировать все, что было определено.

Можно определить замену следующим образом:

```
// Сбор всех аргументов в строку и вывод результата
console.log = function() {
    // 'arguments' не имеет метода для объединения в строку, поэтому
    // сначала аргументы переводятся в массив
    const args = Array.prototype.slice.call(arguments);
    console.log(args.join(', '));
}
```

Теперь это будет вызываться вместо обычного поведения журнала. Эта модификация не сохранится при перезагрузке страницы, поэтому она является полезной и легкой стратегией для отладки или перехвата журнала.

Генерирование ошибок

Как уже отмечалось, генерирование ошибок — прекрасный способ отладки кода. Если сообщения об ошибках достаточно конкретны, беглого взгляда на уведомление может хватить для определения источника ошибки. Хорошие сообщения об ошибках должны предоставлять точные сведения о проблеме, чтобы дополнительные действия по отладке можно было свести к минимуму. Возьмем для примера следующую функцию:

```
function divide(num1, num2) {
    return num1 / num2;
}
```

Эта простая функция делит одно число на другое, но возвращает `NaN`, если один из аргументов не является числом. Неожиданное получение `NaN` в результате простых вычислений часто приводит к проблемам в веб-приложениях. Чтобы подстраховаться, можно проверять перед вычислением тип каждого аргумента:

```
function divide(num1, num2) {
    if (typeof num1 != "number" || typeof num2 != "number") {
        throw new Error("divide(): Both arguments must be numbers.");
    }
    return num1 / num2;
}
```

Если какой-либо из аргументов имеет нечисловой тип, этот код генерирует ошибку, указывая в сообщении имя функции и точную причину ошибки. Увидев это сообщение, вы сразу узнаете, где начать поиск, и получите общее представление о проблеме. Это гораздо лучше, чем получить от браузера неконкретное сообщение об ошибке.

В сложных приложениях для генерирования пользовательских ошибок разработчики обычно задействуют функцию `assert()`. Она принимает условие, которое должно быть истинным, и генерирует ошибку, если оно ложно. Вот совсем простой пример:

```
function assert(condition, message) {
    if (!condition) {
        throw new Error(message);
    }
}
```

Функция `assert()` позволяет заменить несколько инструкций `if` и хорошо подходит для вывода сведений об ошибках, например:

```
function divide(num1, num2) {
    assert(typeof num1 == "number" && typeof num2 == "number",
           "divide(): Both arguments must be numbers.");
    return num1 / num2;
}
```

Применение функции `assert()` сокращает объем кода, требуемого для генерирования пользовательских ошибок, и упрощает его чтение в сравнении с предыдущим примером.

ЧАСТЫЕ УСТАРЕВШИЕ ОШИБКИ INTERNET EXPLORER

Отлаживать JavaScript-ошибки в браузере Internet Explorer традиционно непросто. Многие сообщения об ошибках в устаревших его версиях коротки, непонятны и ничего не говорят о контексте. В последующих разделах описано несколько часто встречающихся нетривиальных JavaScript-ошибок, которые могут возникнуть в устаревших версиях Internet Explorer. Поскольку эти браузеры не поддерживают ES6, код не будет обратно совместимым.

Недопустимый символ

Синтаксис JavaScript содержит не все символы. Если при обработке JS-файла обнаруживается неподдерживаемый символ, IE генерирует ошибку *invalid character* (недопустимый символ). Недопустимым считается любой символ, который не используется в синтаксисе. Например, символ, который выглядит как «минус», но представляется в Юникоде значением 8211 (`\u2013`), нельзя применять вместо обычного минуса (ASCII-код 45). Этот специальный символ часто автоматически вставляется в документы Microsoft Word, и если запустить код с ним в Internet Explorer, возникнет ошибка недопустимого символа. Другие браузеры ведут себя подобным образом. Firefox генерирует ошибку *illegal character*, Safari сообщает о синтаксической ошибке, а Opera возвращает ошибку `ReferenceError`, потому что интерпретирует недопустимый символ как неопределенный идентификатор.

Член группы не найден

Как уже отмечалось, все DOM-объекты в Internet Explorer реализованы с помощью модели COM, а не как встроенные JavaScript-объекты. При сборке мусора

это может вызывать очень странные эффекты. Ошибка *member not found* (член группы не найден) — прямое следствие несоответствия алгоритмов сборки мусора в Internet Explorer.

Эта ошибка обычно возникает при попытке присвоить значение свойству уже уничтоженного объекта. Чтобы появилось это сообщение об ошибке, объект должен быть COM-объектом. Самый показательный пример этого имеет место при работе с объектом `event`. В Internet Explorer он является свойством объекта `window`, которое создается при возникновении события и уничтожается после выполнения последнего обработчика события. Если объект `event` используется в замыкании, которое должно быть выполнено позднее, любая попытка задать свойство этого объекта приведет к этой ошибке, например:

```
document.onclick = function() {
    var event = window.event;
    setTimeout(function() {
        event.returnValue = false;    // ошибка "member not found"
    }, 1000);
};
```

Этот код назначает документу обработчик события `click`, который сохраняет ссылку на объект `window.event` в локальной переменной `event`. Затем эта переменная используется в замыкании, которое передается в функцию `setTimeout()`. При завершении обработчика события `click` объект `event` уничтожается, в результате замыкание ссылается на объект, члены которого больше не существуют. Присвоение значения свойству `returnValue` вызывает ошибку, потому что запись в COM-объект, члены которого уже уничтожены, невозможна.

Неизвестная ошибка выполнения

Неизвестная ошибка выполнения возникает при установке свойства `innerHTML` или `outerHTML`, если блочный элемент вставляется в встроенный элемент или доступ к любому из этих свойств осуществляется в какой-либо части таблицы (`<table>`, `<tbody>` и т. д.). Например, тег `<p>` технически не может содержать элемент блочного уровня вроде `<div>`, так что следующий код вызывает неизвестную ошибку выполнения:

```
p.innerHTML = "<div>Привет</div>";    // где p содержит элемент <p>
```

Другие браузеры пытаются исправлять ошибки при вставке блочных элементов в недопустимые места, но Internet Explorer гораздо строже в этом отношении.

Синтаксическая ошибка

Большинство синтаксических ошибок в Internet Explorer возникают по банальным причинам вроде отсутствия точки с запятой или скобки. Однако иногда причина ошибки далеко не так очевидна.

Если запрошенный внешний JS-файл по какой-либо причине возвращает не JS-код, а что-то другое, Internet Explorer генерирует синтаксическую ошибку. Например, она возникает, если атрибут `src` элемента `<script>` ссылается на HTML-файл. В качестве источника синтаксической ошибки обычно указывается первый символ в первой строке сценария. Opera и Safari при этом также указывают на файл с ошибкой. Internet Explorer этого не делает, так что вы сами должны проверить все используемые в коде внешние JS-файлы. Firefox просто игнорирует любые ошибки синтаксического анализа файлов, добавленных как JS-файлы, но содержащих что-то другое.

Ошибки этого типа обычно возникают, если JS-код динамически генерируется серверным компонентом. Многие серверные языки при ошибке времени выполнения автоматически вставляют HTML-код в выводимые данные, что нарушает синтаксис JavaScript. Если вам трудно отследить синтаксическую ошибку, проверьте каждый внешний JS-файл на предмет того, нет ли в нем HTML-кода, добавленного сервером из-за ошибки.

Не удается найти указанный ресурс

Иногда при ошибке выводится практически бесполезное сообщение *The system cannot locate the resource specified* (Системе не удается найти указанный ресурс). Эта ошибка возникает, если JS-код запрашивает ресурс по URL-адресу, длина которого превышает 2083 символа (максимум в Internet Explorer). Это ограничение длины URL-адреса распространяется не только на JS-код, но и вообще на браузер Internet Explorer (другие браузеры не ограничивают длину URL-адреса так строго). Дополнительно путь, указанный в URL-адресе, ограничен 2048 символами. Эта ошибка возникает в следующем примере:

```
function createLongUrl(url) {
    var s = "?";
    for (var i=0, len=2500; i < len; i++) {
        s += "a";
    }

    return url + s;
}

var x = new XMLHttpRequest();
x.open("get", createLongUrl("http://www.somedomain.com/"), true);
x.send(null);
```

Здесь объект `XMLHttpRequest` пытается запросить URL-адрес, длина которого превышает максимально допустимую, из-за чего при вызове метода `open()` возникает ошибка. Одно из обходных решений этой проблемы — сократить именованные аргументы в строке запроса или удалить из нее ненужные данные. Другое решение — изменить метод запроса на `POST` и отправлять данные в теле запроса, а не в URL-адресе. Технология Ajax, объект `XMLHttpRequest` и проблемы вроде этой подробно обсуждаются в главе 21.

ИТОГИ

Обработка JavaScript-ошибок — обязательное требование к современным сложным веб-приложениям. Если приложение не может предвосхитить возможные ошибки и восстановиться после них, пользователи могут остаться разочарованными. По умолчанию большинство браузеров не уведомляют о JavaScript-ошибках, так что при разработке и отладке вы должны вручную включать режим протоколирования ошибок. Тем не менее в окончательном коде никакие отладочные сообщения об ошибках выводить не следует.

Чтобы не заставлять браузер отвлекаться на JavaScript-ошибки, можно использовать следующие способы:

- В предполагаемых местах возникновения ошибок можно задействовать инструкцию `try-catch`, которая позволяет обработать ошибку надлежащим образом вместо того, чтобы доверять это браузеру.
- Другой вариант — использовать обработчик `window.onerror`, который получает все ошибки, не обработанные с помощью инструкций `try-catch` (относится только к Internet Explorer, Firefox и Chrome).

В каждом веб-приложении следует анализировать источники ошибок и выбирать в зависимости от этого оптимальные способы их обработки.

- Необходимо заблаговременно решить, какие ошибки считать критичными и некритичными.
- После этого можно определить в коде наиболее вероятные места возникновения ошибок. Некоторые частые причины JavaScript-ошибок таковы:
  - приведение типов;
  - неадекватная проверка типов данных;
  - отправка или получение неправильных данных при взаимодействии с сервером.

Для Internet Explorer, Firefox, Chrome, Opera и Safari доступны отладчики JS-кода, интегрированные в браузер или загружаемые в виде дополнений. Все они поддерживают установку точек прерывания, контроль выполнения кода и просмотр значений переменных во время выполнения.

22

XML в JavaScript

- Поддержка XML DOM в браузерах
- XPath в JavaScript
- Использование XSLT-процессоров

Когда-то XML был стандартным форматом хранения структурированных данных и их передачи через интернет. Развитие XML шло в ногу с эволюцией веб-технологий, в частности DOM, которая использовалась для работы со структурами XML-данных не только в веб-браузерах, но и в приложениях для настольных компьютеров и серверов. Из-за отсутствия готовых решений многие разработчики писали собственные синтаксические XML-анализаторы на JavaScript, но с тех пор поддержка XML, XML DOM и многих связанных технологий была встроена во все браузеры.

ПОДДЕРЖКА XML DOM В БРАУЗЕРАХ

Поскольку производители браузеров начали работать над поддержкой XML еще до создания формальных стандартов, XML-реализации в браузерах различаются. Так, в DOM Level 2 была представлена концепция динамического создания DOM-документа с XML-кодом, которая в DOM Level 3 была дополнена синтаксическим анализом и сериализацией. Однако ко времени окончания работы над DOM Level 3 большинство производителей браузеров уже реализовали в них собственные решения.

DOM Level 2 Core

Как отмечено в главе 12, DOM Level 2 у объекта `document.implementation` есть метод `createDocument()`. Вы можете создать пустой XML-документ, используя следующий синтаксис:

```
let xmldom = document.implementation.createDocument(namespaceUri, root, doctype);
```

При работе с XML в JavaScript обычно используется только аргумент `root`, который определяет имя тега DOM-элемента `document` с XML-кодом. Аргумент `namespaceUri` применяется не часто, потому что управлять пространствами имен в JS-коде сложно, что касается аргумента `doctype`, то он требуется совсем редко.

Создать XML-документ с тегом `<root>` в качестве элемента `document` можно следующим образом:

```
let xmldom = document.implementation.createDocument("", "root", null);

console.log(xmldom.documentElement.tagName);    // "root"

let child = xmldom.createElement("child");
xmldom.documentElement.appendChild(child);
```

Этот код создает документ XML DOM без пространства имен, предлагаемого по умолчанию, и без типа документа. Даже если пространство имен и тип документа не требуются, аргументы все же должны быть переданы в метод `createDocument()`. В качестве URI пространства имен в него передается пустая строка, а в качестве типа документа — значение `null`. Переменной `xmldom` назначается экземпляр типа `Document` согласноDOM Level 2 со всеми методами и свойствами DOM, описанными в главе 12. Далее этот код выводит имя тега элемента `document`, а затем создает и добавляет к нему дочерний элемент.

Проверить, включена ли в браузере поддержка DOM Level 2 XML, можно следующим образом:

```
let hasXmlDom = document.implementation.hasFeature("XML", "2.0");
```

На практике разработчики редко создают XML-документ «с нуля», наращивая его с помощью DOM-методов. Гораздо чаще приходится преобразовывать XML-документ в структуру DOM или наоборот. DOM Level 2 не предлагает такого функционала, но со временем некоторые способы решения этой задачи стали стандартами де-факто.

Тип DOMParser

В Firefox для синтаксического анализа XML-кода с преобразованием его в DOM-документ был введен тип `DOMParser`, который позднее был реализован во всех остальных поставщиках браузеров. Чтобы использовать его, нужно создать экземпляр `DOMParser` и вызвать метод `parseFromString()`. Он принимает исходную строку

XML-кода и тип содержимого, которым всегда должен быть "text/xml", а возвращает экземпляр типа Document, например:

```
let parser = new DOMParser();
let xmldom = parser.parseFromString("<root><child/></root>", "text/xml");

console.log(xmldom.documentElement.tagName);    // "root"
console.log(xmldom.documentElement.firstChild.tagName);    // "child"

let anotherChild = xmldom.createElement("child");
xmldom.documentElement.appendChild(anotherChild);

let children = xmldom.getElementsByTagName("child");
alert(children.length);        // 2
```

Этот код выполняет синтаксический анализ простой строки XML-кода, преобразуя ее в DOM-документ с узлом <root> в качестве элемента document и единственным дочерним элементом <child>. С возвращенным документом можно работать, используя DOM-методы.

Тип DOMParser выполняет синтаксический анализ XML-кода только правильного формата, а потому не способен преобразовать HTML-код в HTML-документ. При ошибке синтаксического анализа браузеры ведут себя по-разному. В Firefox, Opera, Safari и Chrome метод parseFromString() при ошибке все же возвращает объект Document, но его элементом document является <parsererror>, а содержимым элемента — описание ошибки, например:

```
<parsererror xmlns="http://www.mozilla.org/newlayout/xml/parsererror.xml">
Parsing Error: no element found Location
Адрес: file:///I:/My%20Writing/My%20Books/Professional%20JavaScript/
Second%20Edition/Examples/Ch15/DOMParserExample2.htm
Строка 1, символ 7:<sourcetext>&lt;root&gt;
------^</sourcetext></parsererror>
```

Firefox и Opera возвращают документы в этом формате, а в Safari и Chrome возвращаемый документ содержит элемент <parsererror> там, где произошла ошибка. Ранние версии Internet Explorer генерируют ошибку синтаксического анализа при вызове метода parseFromString(). Из-за этих различий для выявления ошибки синтаксического анализа лучше всего воспользоваться блоком try-catch, а при отсутствии ошибки попытаться найти элемент <parsererror> в документе методом getElementsByTagName():

```
let parser = new DOMParser(),
    xmldom,
    errors;
try {
    xmldom = parser.parseFromString("<root>", "text/xml");
    errors = xmldom.getElementsByTagName("parsererror");
    if (errors.length > 0) {
        throw new Error("Parsing error!"); // ошибка синтаксического анализа
    }
} catch (ex) {
    console.log("Parsing error!");
}
```

В этом примере в исходной строке отсутствует закрывающий тег `</root>`. При ее обработке Internet Explorer 9+ генерирует ошибку. В Firefox и Opera элемент `<parsererror>` становится элементом `document`, а в Chrome и Safari — первым дочерним элементом узла `<root>`. Оба эти случая распознаются при вызове метода `getElementsByTagName("parsererror")`. Если он возвращает какие-либо элементы, значит, произошла ошибка, поэтому отображается оповещение о ней. При желании можно извлечь из элемента сведения об ошибке.

Тип XMLSerializer

Тип `XMLSerializer` противоположен типу `DOMParser`: он сериализует DOM-документ в XML-строку. Этот тип появился в Firefox и позднее был реализован в во всех основных браузерах.

Чтобы сериализовать DOM-документ, нужно создать экземпляр `XMLSerializer`, а затем вызвать метод `serializeToString()`, передав в него документ:

```
let serializer = new XMLSerializer();
let xml = serializer.serializeToString(xmldom);
console.log(xml);
```

Метод `serializeToString()` возвращает строку без какого-либо форматирования, так что читать ее трудно.

Тип `XMLSerializer` поддерживает сериализацию любых допустимых DOM-объектов, в том числе отдельных узлов и HTML-документов. HTML-документ, переданный в метод `serializeToString()`, обрабатывается как XML-документ, так что формат итогового кода получается правильным.

> **ПРИМЕЧАНИЕ** Если передать в метод `serializeToString()` что-либо, отличное от DOM-объекта, произойдет ошибка.

ПОДДЕРЖКА XPATH В БРАУЗЕРАХ

XPath был создан для доступа к конкретным узлам в DOM-документе, так что его важность для обработки XML очевидна. API для XPath впервые появился в рекомендации DOM Level 3 XPath. Эта спецификация реализована во всех основных браузерах, кроме Internet Explorer.

DOM Level 3 XPath

Спецификация DOM Level 3 XPath определяет интерфейсы для обработки XPath-выражений в DOM. Определить, поддерживает ли браузер DOM Level 3 XPath, можно следующим образом:

```
let supportsXPath = document.implementation.hasFeature("XPath", "3.0");
```

Из определенных в спецификации типов наиболее важны `XPathEvaluator` и `XPathResult`. Тип `XPathEvaluator` используется для оценки XPath-выражений в конкретном контексте и имеет три метода:

- `createExpression(`*выражение*, *объектРазрешенияПИ*`)` — вычисляет XPath-выражение и сопутствующие сведения о пространстве имен, возвращая объект `XPathExpression` — скомпилированную версию запроса. Этот метод полезен, если один запрос будет выполняться несколько раз.
- `createNSResolver(`*узел*`)` — создает объект `XPathNSResolver` на основе сведений о пространстве имен полученного узла. Объект `XPathNSResolver` требуется при оценке выражений для XML-документа, в котором используются пространства имен.
- `evaluate(`*выражение*, *контекст*, *объектРазрешенияПИ*, *тип*, *результат*`)` — оценивает XPath-выражение в указанном контексте с конкретным пространством имен. Дополнительные аргументы указывают, как должен быть возвращен результат.

Тип `Document` обычно реализован с помощью интерфейса `XPathEvaluator`. Таким образом, вы можете либо создать новый экземпляр `XPathEvaluator`, либо использовать методы экземпляра `Document` (и с XML-, и с HTML-документами).

Из этих трех методов более всего востребован `evaluate()`. Он принимает пять аргументов: XPath-выражение, узел контекста, объект разрешения пространства имен, тип возвращаемого результата и объект `XPathResult` для представления результата (он обычно имеет значение `null`, потому что результат также возвращается как значение функции). Третий аргумент, объект разрешения пространства имен, необходим, только если в XML-коде используется XML-пространство имен. В противном случае он должен иметь значение `null`. Типом возвращаемого результата может быть одна из десяти констант:

- `XPathResult.ANY_TYPE` — возвращает тип данных, соответствующий XPath-выражению;
- `XPathResult.NUMBER_TYPE` — возвращает числовое значение;
- `XPathResult.STRING_TYPE` — возвращает строковое значение;
- `XPathResult.BOOLEAN_TYPE` — возвращает логическое значение;
- `XPathResult.UNORDERED_NODE_ITERATOR_TYPE` — возвращает множество узлов, соответствующих XPath-выражению, хотя их порядок может не соответствовать порядку узлов в документе;
- `XPathResult.ORDERED_NODE_ITERATOR_TYPE` — возвращает множество узлов, соответствующих XPath-выражению, в том порядке, в котором они располагаются в документе. Этот тип результата используется чаще всего;
- `XPathResult.UNORDERED_NODE_SNAPSHOT_TYPE` — возвращает снимок множества узлов, сохраняя их вне документа, чтобы последующие изменения документа не влияли на множество узлов. Порядок узлов в множестве может отличаться от их расположения в документе;

- `XPathResult.ORDERED_NODE_SNAPSHOT_TYPE` — возвращает снимок множества узлов, сохраняя их вне документа, чтобы последующие изменения документа не влияли на множество результатов. Узлы в множестве результатов располагаются в том же порядке, что и в документе;
- `XPathResult.ANY_UNORDERED_NODE_TYPE` — возвращает множество узлов, соответствующих XPath-выражению, хотя их порядок может не соответствовать порядку узлов в документе;
- `XPathResult.FIRST_ORDERED_NODE_TYPE` — возвращает множество только с одним узлом, являющимся первым узлом в документе, соответствующим XPath-выражению.

От указанного типа результата зависит способ его получения. Вот типичный пример:

```
let result = xmldom.evaluate("employee/name", xmldom.documentElement, null,
                             XPathResult.ORDERED_NODE_ITERATOR_TYPE, null);

if (result !== null) {
    let element = result.iterateNext();
    while(element) {
        console.log(element.tagName);
        node = result.iterateNext();
    }
}
```

Этот код запрашивает тип результата `XPathResult.ORDERED_NODE_ITERATOR_TYPE`, который используется чаще всего. Если ни один узел не соответствует XPath-выражению, метод `evaluate()` возвращает `null`, иначе возвращается объект `XPathResult`, у которого есть свойства и методы получения результатов конкретных типов. Если результатом является итератор узлов (неважно, упорядоченных или нет), для получения каждого соответствующего выражению узла нужно использовать метод `iterateNext()`. По исчерпании таких узлов метод `iterateNext()` возвращает `null`.

Если запрошен снимок множества узлов (упорядоченных или неупорядоченных), для доступа к ним требуются метод `snapshotItem()` и свойство `snapshotLength`, например:

```
let result = xmldom.evaluate("employee/name", xmldom.documentElement, null,
                             XPathResult.ORDERED_NODE_SNAPSHOT_TYPE, null);

if (result !== null) {
    for (let i=0, len=result.snapshotLength; i < len; i++) {
        console.log(result.snapshotItem(i).tagName);
    }
}
```

Здесь свойство `snapshotLength` возвращает количество узлов в снимке, а метод `snapshotItem()` — узел в конкретной позиции (подобно свойству `length` и методу `item()` объекта `NodeList`).

Результат из одного узла

Аргумент `XPathResult.FIRST_ORDERED_NODE_TYPE` позволяет запросить первый узел, соответствующий выражению, который доступен в результате через свойство `singleNodeValue`, например:

```
let result = xmldom.evaluate("employee/name", xmldom.documentElement, null,
                             XPathResult.FIRST_ORDERED_NODE_TYPE, null);

if (result !== null) {
    console.log(result.singleNodeValue.tagName);
}
```

Как и в других запросах, при отсутствии узлов, соответствующих выражению, метод `evaluate()` возвращает `null`. Если узел возвращается, он доступен через свойство `singleNodeValue`.

Результаты простых типов

С помощью XPath-выражений можно получать простые типы данных (не узлы), используя логический, числовой и строковый типы `XPathResult`. Если запросить результат одного из этих типов, будет возвращено одно значение в свойстве `booleanValue`, `numberValue` или `stringValue`. В случае логического типа обработка XPath-выражения обычно дает `true`, если хотя бы один узел соответствует выражению, и `false` в противном случае, например:

```
let result = xmldom.evaluate("employee/name", xmldom.documentElement, null,
                             XPathResult.BOOLEAN_TYPE, null);
console.log(result.booleanValue);
```

Если какие-либо узлы в этом примере соответствуют выражению `"employee/name"`, свойство `booleanValue` возвратит `true`.

Если запрашивается числовой тип, XPath-выражением должна быть XPath-функция, возвращающая число, например функция `count()`, которая подсчитывает все узлы, соответствующие указанному шаблону:

```
let result = xmldom.evaluate("count(employee/name)", xmldom.documentElement,
                             null, XPathResult.NUMBER_TYPE, null);
console.log(result.numberValue);
```

Этот код выводит количество узлов, которые соответствуют выражению `"employee/name"`. Если попытаться использовать этот метод без какой-либо из специальных XPath-функций, свойство `numberValue` будет равно `NaN`.

При запросе результата строкового типа метод `evaluate()` находит первый узел, соответствующий XPath-выражению, и возвращает значение его первого дочернего узла, если он является текстовым, или пустую строку в противном случае, например:

```
let result = xmldom.evaluate("employee/name", xmldom.documentElement, null,
                             XPathResult.STRING_TYPE, null);
console.log(result.stringValue);
```

Этот код выводит содержимое первого текстового узла, дочернего по отношению к первому элементу, который соответствует строке `"element/name"`.

Тип результата по умолчанию

Все XPath-выражения автоматически сопоставляются с тем или иным типом результата. Указание конкретного типа ограничивает возможные результаты, но вы можете автоматически получить предлагаемый по умолчанию тип, указав константу `XPathResult.ANY_TYPE`. В этом случае результатом обычно оказывается логическое значение, число, строка или итератор неупорядоченных узлов. Узнать тип возвращенного результата можно с помощью его свойства `resultType`, например:

```
let result = xmldom.evaluate("employee/name", xmldom.documentElement, null,
                             XPathResult.ANY_TYPE, null);

if (result !== null) {
    switch(result.resultType) {
        case XPathResult.STRING_TYPE:
            // обработка результата строкового типа
            break;

        case XPathResult.NUMBER_TYPE:
            // обработка результата числового типа
            break;

        case XPathResult.BOOLEAN_TYPE:
            // обработка результата логического типа
            break;

        case XPathResult.UNORDERED_NODE_ITERATOR_TYPE:
            // обработка итератора неупорядоченных узлов
            break;

        default:
            // обработка результатов других возможных типов

    }
}
```

Константа `XPathResult.ANY_TYPE` позволяет использовать XPath более естественным образом, но может требовать дополнительной обработки результата.

Поддержка пространств имен

Если в XML-документе используются пространства имен, для правильной оценки XPath-выражения нужно сообщить объекту `XPathEvaluator` пространство имен. Есть несколько способов сделать это. Рассмотрим следующий XML-код:

```
<?xml version="1.0" ?>
<wrox:books xmlns:wrox="http://www.wrox.com/">
    <wrox:book>
        <wrox:title>Professional JavaScript for Web Developers</wrox:title>
```

```
        <wrox:author>Nicholas C. Zakas</wrox:author>
    </wrox:book>
    <wrox:book>
        <wrox:title>Professional Ajax</wrox:title>
        <wrox:author>Nicholas C. Zakas</wrox:author>
        <wrox:author>Jeremy McPeak</wrox:author>
        <wrox:author>Joe Fawcett</wrox:author>
    </wrox:book>
</wrox:books>
```

В этом XML-документе все элементы относятся к пространству имен `http://www.wrox.com/` с префиксом `wrox`. Чтобы использовать XPath-выражение с этим документом, нужно определить упоминаемые в нем пространства имен; в противном случае оценить выражение не удастся.

Первый способ обработать пространства имен — создать объект `XPathNSResolver` методом `createNSResolver()`, который принимает узел документа, содержащий определение пространства имен. В предыдущем примере это элемент документа `<wrox:books>` с атрибутом `xmlns`, который определяет пространство имен. Вы можете передать этот узел в метод `createNSResolver()` и использовать полученный результат в методе `evaluate()`:

```
let nsresolver = xmldom.createNSResolver(xmldom.documentElement);

let result = xmldom.evaluate("wrox:book/wrox:author",
                             xmldom.documentElement, nsresolver,
                             XPathResult.ORDERED_NODE_SNAPSHOT_TYPE, null);

console.log(result.snapshotLength);
```

Передача объекта `nsresolver` в метод `evaluate()` гарантирует, что префикс `wrox` в XPath-выражении будет обработан правильно. Попытка использовать это же выражение без объекта `XPathNSResolver` приведет к ошибке.

Второй способ обработать пространства имен — определить функцию, которая принимает префикс пространства имен и возвращает соответствующий URI, например:

```
let nsresolver = function(prefix) {
    switch(prefix) {
        case "wrox": return "http://www.wrox.com/";
        // другие варианты
    }
};

let result = xmldom.evaluate("count(wrox:book/wrox:author)",
        xmldom.documentElement, nsresolver, XPathResult.NUMBER_TYPE, null);

console.log(result.numberValue);
```

Функция разрешения пространств имен помогает, если вы не знаете, какой узел документа содержит определения пространств имен. Если вам известны префиксы и URI, вы можете определить функцию для возвращения этой информации и передать ее в качестве третьего аргумента в метод `evaluate()`.

ПОДДЕРЖКА XSLT В БРАУЗЕРАХ

XSLT — это вспомогательная технология для XML, которая использует XPath для преобразования одного документа из одного представления в другое. В отличие от XML и XPath, технология XSLT не имеет формального API и совсем не представлена в формальной DOM, из-за чего производители браузеров могли реализовать ее по своему усмотрению. Впервые поддержка XSLT в JavaScript появилась в Internet Explorer.

Тип XSLTProcessor

Разработчики Firefox реализовали поддержку XSLT в JavaScript, создав новый тип `XSLTProcessor`. Он позволяет преобразовывать XML-документы, используя XSLT подобно XSL-процессору в Internet Explorer. Со временем объект `XSLTProcessor` был скопирован во всех основных браузерах, что сделало его стандартом де-факто для XSLT-преобразований в JavaScript.

Как и в Internet Explorer, первым делом нужно загрузить два DOM-документа: один с XML и один с XSLT. После этого следует создать объект `XSLTProcessor` и назначить ему таблицу XSLT-стилей с помощью метода `importStylesheet()`:

```
let processor = new XSLTProcessor()
processor.importStylesheet(xsltdom);
```

Наконец, нужно выполнить преобразование. Это можно сделать двумя способами. Если требуется возвратить полный DOM-документ, вызовите метод `transformToDocument()`. Также можно получить объект фрагмента документа, вызвав метод `transformToFragment()`. Вообще говоря, единственный разумный способ использования метода `transformToFragment()` — это добавление его результатов в другой DOM-документ.

При вызове метода `transformToDocument()` просто передайте в него документ XML DOM, а возвращенный им результат используйте как совершенно другой DOM-документ, например:

```
let result = processor.transformToDocument(xmldom);
console.log(serializeXml(result));
```

Метод `transformToFragment()` принимает два аргумента: документ XML DOM, который нужно преобразовать, и документ, которому должен принадлежать итоговый фрагмент. Это гарантирует, что новый фрагмент документа будет действителен в целевом документе. Например, вы можете создать фрагмент и добавить его к странице, передав в метод объект `document` в качестве второго параметра:

```
let fragment = processor.transformToFragment(xmldom, document);
let div = document.getElementById("divResult");
div.appendChild(fragment);
```

Здесь процессор создает фрагмент, принадлежащий объекту `document`. Это позволяет добавить фрагмент в имеющийся на странице элемент `<div>`.

Если у таблицы XSLT-стилей форматом вывода является `"xml"` или `"html"`, создание документа или фрагмента документа вполне обоснованно, но если форматом вывода является `"text"`, обычно требуется просто текстовый результат преобразования.

К сожалению, метода, возвращающего непосредственно текст, не существует. Метод `transformToDocument()` при формате вывода `"text"` возвращает полный XML-документ, но его содержимое зависит от браузера. Например, Safari возвращает весь HTML-документ, а Opera и Firefox — документ с одним элементом, текстом которого является вывод.

Решением этой проблемы является вызов метода `transformToFragment()`, который возвращает фрагмент документа с единственным дочерним узлом, содержащим текст результата. Следовательно, текст можно получить следующим образом:

```
let fragment = processor.transformToFragment(xmldom, document);
let text = fragment.firstChild.nodeValue;
console.log(text);
```

Этот код работает одинаково во всех поддерживающих его браузерах, возвращая только текстовый вывод преобразования.

Использование параметров

Объект `XSLTProcessor` позволяет также задать XSLT-параметры с помощью метода `setParameter()`, который принимает три аргумента: URI пространства имен, локальное имя параметра и задаваемое значение. Обычно URI пространства имен имеет значение `null`, а локальным именем является имя параметра. Этот метод нужно вызывать до метода `transformToDocument()` или `transformToFragment()`, например:

```
let processor = new XSLTProcessor()
processor.importStylesheet(xsltdom);
processor.setParameter(null, "message", "Hello World!");
let result = processor.transformToDocument(xmldom);
```

Для работы с параметрами используются также методы `getParameter()` и `removeParameter()`, которые получают текущее значение параметра и удаляют параметр соответственно. Оба они принимают URI пространства имен (обычно `null`) и локальное имя параметра, например:

```
let processor = new XSLTProcessor()
processor.importStylesheet(xsltdom);
processor.setParameter(null, "message", "Hello World!");

console.log(processor.getParameter(null, "message"));    // "Hello World!"
processor.removeParameter(null, "message");

let result = processor.transformToDocument(xmldom);
```

Эти методы применяются нечасто и предоставляются в основном для удобства.

Сброс процессора

Каждый экземпляр `XSLTProcessor` можно использовать многократно для преобразований с разными таблицами XSLT-стилей. Метод `reset()` удаляет из процессора все параметры и таблицы стилей, позволяя снова вызвать метод `importStylesheet()` для загрузки другой таблицы XSLT-стилей, например:

```
let processor = new XSLTProcessor()
processor.importStylesheet(xsltdom);

// какие-то преобразования

processor.reset();
processor.importStylesheet(xsltdom2);

// другие преобразования
```

Повторное использование экземпляра `XSLTProcessor` экономит память, если в преобразованиях задействовано несколько таблиц стилей.

ИТОГИ

В JavaScript доступна развитая поддержка XML и связанных технологий. К сожалению, из-за отсутствия спецификаций она реализована в браузерах по-разному. Спецификация DOM Level 2 определила API для создания пустых XML-документов, но не для синтаксического анализа или сериализации.

В браузерах для синтаксического анализа и сериализации XML-кода были добавлены два объекта:

- `DOMParser` — это простой объект, который преобразует строку XML в DOM-документ;
- Объект `XMLSerializer` выполняет противоположную операцию, сериализуя DOM-документ в XML-строку.

В DOM Level 3 представлена спецификация XPath API, которая реализована в Firefox, Safari, Chrome и Opera. Этот API позволяет выполнять в JS-коде любые XPath-запросы для DOM-документа и получать результат независимо от его типа данных.

У XSLT нет API с открытой спецификацией. В Firefox для выполнения преобразований с помощью JavaScript был создан тип `XSLTProcessor`.

23

JSON

- Общие сведения о синтаксисе JSON
- Синтаксический анализ JSON
- Сериализация JSON

Загрузки с сайта wrox.com для этой главы

Обратите внимание, что все примеры кода для этой главы доступны на веб-сайте книги по ссылке www.wrox.com/go/projavascript4e на вкладке Downloads.

Когда-то XML был де-факто стандартом передачи структурированных данных через интернет. Большинство веб-сервисов первого поколения были основаны на XML, что подчеркивало их предназначение — обеспечение взаимодействия серверов. Однако XML не лишен недостатков. Многим казалось, что он слишком многословен и избыточен. Было предложено несколько решений этих проблем, но развитие веб-технологий пошло по другому пути.

Дуглас Крокфорд (Douglas Crockford) впервые описал нотацию JavaScript-объектов (JavaScript Object Notation, JSON) в 2006 г. в IETF RFC 4627, хотя она использовалась уже в 2001 г. JSON — это строгое подмножество JavaScript, использующее несколько его паттернов для представления структурированных данных. Крокфорд указал, что JSON лучше XML подходит для доступа к структурированным данным в JavaScript, потому что позволяет напрямую передавать их в метод `eval()` и не требует создания DOM.

Важно понимать, что JSON — это формат данных, а не язык программирования. JSON не входит в JavaScript, хотя имеет такой же синтаксис, и, будучи форматом данных, используется не только в JavaScript. Синтаксические анализаторы и средства сериализации JSON доступны для многих языков программирования.

СИНТАКСИС

JSON поддерживает значения трех типов:

- **Простые значения** — строки, числа, логические значения и значения `null` можно представлять в JSON, используя тот же синтаксис, что и в JavaScript. Специальное значение `undefined` не поддерживается.
- **Объекты** — первый сложный тип данных, служащий для хранения упорядоченных пар ключей и значений. Каждое значение может быть примитивным или сложным типом.
- **Массивы** — второй сложный тип данных, который представляет упорядоченный список значений, доступных по числовому индексу. Значениями массивов могут быть данные любого типа, в том числе простые значения, объекты и даже другие массивы.

В JSON нет переменных, функций или экземпляров объектов. Этот формат предназначен для работы со структурированными данными, и хотя его синтаксис заимствован из JavaScript, его не следует воспринимать как одну из базовых составляющих языка.

Простые значения

Простейший JSON-код — это примитивное значение, например:

```
5
```

Это просто число 5. А следующий JSON-код представляет строку:

```
"Hello world!"
```

Важное различие между JavaScript- и JSON-строками заключается в том, что последние нужно заключать в двойные кавычки (использование одинарных кавычек приводит к синтаксической ошибке).

Логические значения и значение `null` также допустимы в JSON, но на практике этот формат чаще всего используется для представления более сложных структур данных, в которых простые значения — только часть всей информации.

Объекты

Объекты представляются в JSON с помощью слегка измененной нотации литералов объектов, которые в JavaScript выглядят следующим образом:

```
let person = {
    name: "Nicholas",
    age: 29
};
```

Это стандартный способ создания литералов объектов, но в JSON имена свойств заключаются в кавычки. Следующий код эквивалентен предыдущему:

```
let object = {
    "name": "Nicholas",
    "age": 29
};
```

В JSON тот же объект представляется так:

```
{
    "name": "Nicholas",
    "age": 29
}
```

Можно заметить два отличия этого кода от примера JS-кода. Во-первых, в нем нет объявления переменной (переменные в JSON отсутствуют). Во-вторых, в этом коде отсутствует заключительная точка с запятой (она не требуется, потому что это не JavaScript-инструкция). Чтобы JSON-код был синтаксически правильным, имена свойств должны быть заключены в кавычки. Свойства могут содержать любые простые и сложные значения, что позволяет создавать вложенные объекты, например:

```
{
    "name": "Nicholas",
    "age": 29,
    "school": {
        "name": "Merrimack College",
        "location": "North Andover, MA"
    }
}
```

В этом примере объект со сведениями об учебном заведении вложен в объект верхнего уровня. Хотя код содержит два свойства с именем `"name"`, это нормально, потому что они принадлежат разным объектам. Однако в отдельном объекте одноименных свойств быть не должно.

В отличие от JavaScript, имена свойств-объектов в JSON всегда нужно заключать в двойные кавычки. При написании JSON-кода вручную отсутствие двойных кавычек или использование одинарных кавычек — частая ошибка.

Массивы

Второй сложный тип данных в JSON — массив. Массивы представляются в JSON с использованием нотации литералов массивов из JavaScript. Например, рассмотрим JavaScript-массив:

```
let values = [25, "hi", true];
```

Этот же массив в JSON будет выглядеть так:

```
[25, "hi", true]
```

Обратите внимание еще раз на отсутствие переменной и точки с запятой. Массивы и объекты можно использовать вместе для представления более сложных коллекций данных, например:

```
[
    {
        "title": "Professional JavaScript",
        "authors": [
            "Nicholas C. Zakas",
            "Matt Frisbie"
        ],
        "edition": 4,
        "year": 2017
    },
    {
        "title": "Professional JavaScript",
        "authors": [
            "Nicholas C. Zakas",
            "Nicholas C. Zakas"
        ],
        edition: 3,
        year: 2011
    },
    {
        "title": "Professional JavaScript",
        "authors": [
            "Nicholas C. Zakas"
        ],
        edition: 2,
        year: 2009
    },
    {
        "title": "Professional Ajax",
        "authors": [
            "Nicholas C. Zakas",
            "Jeremy McPeak",
            "Joe Fawcett"
        ],
        edition: 2,
        year: 2008
    },
    {
        "title": "Professional Ajax",
        "authors": [
            "Nicholas C. Zakas",
            "Jeremy McPeak",
            "Joe Fawcett"
        ],
        edition: 1,
        year: 2007
    },
    {
        "title": "Professional JavaScript",
        "authors": [
            "Nicholas C. Zakas"
        ],
        edition: 1,
        year: 2006
    }
]
```

Этот массив содержит несколько объектов, представляющих книги. Каждый объект включает несколько ключей, одним из которых является другой массив `"authors"`. Объекты и массивы обычно находятся на верхних уровнях JSON-иерархии (хотя такого требования нет) и используются для создания самых разных структур данных.

СИНТАКСИЧЕСКИЙ АНАЛИЗ И СЕРИАЛИЗАЦИЯ

Рост популярности JSON объясняется не только привычным синтаксисом. Еще важнее то, что он позволяет преобразовать данные в объект, который можно использовать в JavaScript. Это резко контрастирует с XML-кодом, который преобразуется в DOM-документ, что затрудняет извлечение данных. Например, получить название третьей книги из предыдущего фрагмента можно следующим образом:

```
books[2].title
```

Здесь предполагается, что структура данных содержится в переменной `books`. Сравните это выражение с типичным способом просмотра структуры DOM:

```
doc.getElementsByTagName("book")[2].getAttribute("title")
```

Неудивительно, что JSON приобрел невероятную популярность среди JavaScript-разработчиков и стал стандартом де-факто в веб-сервисах.

Объект JSON

Возможности ранних синтаксических JSON-анализаторов практически ограничивались вызовом `eval()` из JavaScript. Поскольку JSON является подмножеством синтаксиса JavaScript, функция `eval()` может анализировать, интерпретировать и возвращать данные как объекты и массивы JavaScript. В ECMAScript 5 средства синтаксического анализа JSON формализованы в виде встроенного глобального объекта `JSON`. Он поддерживается в основных браузерах, а JSON-прокладка (shim) для старых браузеров доступна по адресу https://github.com/douglascrockford/JSON-js. Из-за риска столкнуться с исполняемым кодом обрабатывать JSON-код в старых браузерах лишь с помощью функции `eval()` опасно. Использование JSON-прокладки — оптимальный вариант для браузеров без встроенных JSON-средств синтаксического анализа.

У объекта `JSON` есть два метода: `stringify()` и `parse()`. В простых сценариях они просто сериализуют объект JavaScript в строку JSON и преобразуют ее в значение JavaScript соответственно. Вот пример:

```
let book = {
               title: "Professional JavaScript",
                authors: [
                    "Nicholas C. Zakas",
                   "Matt Frisbie"
                ],
```

```
        edition: 4,
        year: 2017
    };

let jsonText = JSON.stringify(book);
```

Этот код сериализует JavaScript-объект в JSON-строку методом `JSON.stringify()` и сохраняет ее в переменной `jsonText`. По умолчанию метод `JSON.stringify()` возвращает JSON-строку без дополнительных пробелов или отступов, так что в переменной `jsonText` сохраняется следующее значение:

```
{"title":"Professional JavaScript","authors":["Nicholas C. Zakas"],
"edition":3,"year":2011}
```

При сериализации JavaScript-объекта функции и члены прототипа специально не включаются в результат. Кроме того, пропускаются любые свойства со значением `undefined`. Итоговая строка содержит только свойства экземпляра, относящиеся к одному из JSON-типов данных.

Метод `JSON.parse()` создает из JSON-строки соответствующее JavaScript-значение. Например, так можно создать объект, аналогичный объекту `book`:

```
let bookCopy = JSON.parse(jsonText);
```

Переменные `book` и `bookCopy` являются отдельными объектами, которые не связаны друг с другом, хотя и имеют общие свойства.

Если текст, переданный в метод `JSON.parse()`, не является допустимой JSON-строкой, генерируется ошибка.

Параметры сериализации

Кроме сериализуемого JavaScript-объекта метод `JSON.stringify()` принимает еще два аргумента, с помощью которых можно указать альтернативные способы сериализации объекта. Первым аргументом является фильтр, которым может быть массив или функция, а вторым — параметр отступа для итоговой JSON-строки. Используя их по отдельности или вместе, можно задействовать очень полезные механизмы управления JSON-сериализацией.

Фильтрация результатов

Если вторым аргументом метода `JSON.stringify()` является массив, метод сериализует только свойства объекта, указанные в массиве, например:

```
let book = {
    title: "Professional JavaScript",
    authors: [
        "Nicholas C. Zakas",
        "Matt Frisbie"
    ],
    edition: 4,
```

```
    year: 2017
};

let jsonText = JSON.stringify(book, ["title", "edition"]);
```

Здесь в метод `JSON.stringify()` в качестве второго аргумента передается массив со строками `"title"` и `"edition"`. Они соответствуют свойствам сериализуемого объекта, так что итоговая JSON-строка будет содержать только эти свойства:

```
{"title":"Professional JavaScript","edition":4}
```

Если вторым аргументом метода является функция, его поведение немного отличается. Эта функция должна принимать два аргумента: имя ключа свойства и значение свойства. Ключ указывает, что нужно сделать со свойством. Он всегда является строкой, но может быть пустой строкой, если значение не соответствует ключу.

Чтобы изменить способ сериализации объекта, возвратите из функции значение, которое нужно сериализовать при указанном ключе. Помните, что возвращение значения `undefined` приводит к тому, что свойство не включается в результат, например:

```
let book = {
    title: "Professional JavaScript",
    authors: [
        "Nicholas C. Zakas",
        "Matt Frisbie"
    ],
    edition: 4,
    year: 2017
};

let jsonText = JSON.stringify(book, (key, value) => {
    switch(key) {
        case "authors":
            return value.join(",")

        case "year":
            return 5000;

        case "edition":
            return undefined;

        default:
            return value;
    }
});
```

Эта функция выполняет фильтрацию на основе ключа. Для ключа `"authors"` значение преобразуется из массива в строку, для ключа `"year"` возвращается значение 5000, а ключ `"edition"` вообще опускается — для него возвращается значение `undefined`. Важно указать вариант, предлагаемый по умолчанию и просто возвращающий переданное в функцию значение, чтобы в результат были добавлены все остальные

значения. При первом вызове этой функции аргумент `key` равен пустой строке, а `value` ссылается на объект `book`. Итоговая JSON-строка такова:

```
{"title":"Professional JavaScript","authors":"Nicholas C. Zakas", "Matt Frisbie"
"year":5000}
```

Не забывайте, что фильтры применяются ко всем объектам, которые содержатся в сериализуемом объекте, так что если в первом примере из этого раздела передать в метод массив с несколькими объектами, каждый объект в итоге будет содержать только свойства `"title"` и `"edition"`.

В Firefox 3.5–3.6 в методе `JSON.stringify()` есть дефект, который проявляется, если вторым аргументом является функция. Она может работать только как фильтр: возвращение значения `undefined` приводит к тому, что свойство пропускается, а при любом другом значении оно добавляется в результат. Этот дефект исправлен в Firefox 4.

Отступы строк

Третий аргумент метода `JSON.stringify()` определяет отступы и свободное пространство. Если им является число, оно представляет количество пробелов, используемых в качестве отступа на каждом уровне. Например, задать для каждого уровня отступ из четырех пробелов можно следующим образом:

```
let book = {
    title: "Professional JavaScript",
    authors: [
        "Nicholas C. Zakas",
        "Matt Frisbie"
    ],
    edition: 4,
    year: 2017
};

let jsonText = JSON.stringify(book, null, 4);
```

В результате переменной `jsonText` будет присвоена следующая строка:

```
{
    "title": "Professional JavaScript",
    "authors": [
        "Nicholas C. Zakas",
        "Matt Frisbie"
    ],
    "edition": 4,
    "year": 2017
}
```

Возможно, вы заметили, что метод `JSON.stringify()` вставляет в JSON-код символы перевода строки, чтобы упростить его чтение. Это выполняется для всех допустимых значений аргумента отступа (отступы без перевода строки едва ли полезны). Максимальное значение отступа — 10. Если указать большее число, оно автоматически уменьшается до 10.

Если в качестве отступа указана строка, а не число, она используется в JSON-строке как символ отступа вместо пробела. Это позволяет сделать символом отступа знак табуляции или что-нибудь совершенно произвольное, например два дефиса:

```
let jsonText = JSON.stringify(book, null, "--");
```

В этом случае значение `jsonText` будет таким:

```
{
--"title": "Professional JavaScript",
--"authors": [
----"Nicholas C. Zakas"
----"Matt Frisbie"
--],
--"edition": 4,
--"year": 2017
}
```

Длина строки отступа ограничена десятью символами. Если указать строку длиннее десяти символов, она обрезается.

Метод toJSON()

Иногда возможностей метода `JSON.stringify()` недостаточно для сериализации некоторых объектов. В этих случаях можно добавить к объекту метод `toJSON()`, чтобы объект возвращал правильное представление себя в формате JSON. Между прочим, у встроенного типа `Date` есть метод `toJSON()`, который автоматически преобразует объекты `Date` из JavaScript в строки дат стандарта ISO 8601 (по сути, это то же самое, что вызвать метод `toISOString()` объекта `Date`).

Метод `toJSON()` можно добавить к любому объекту, например:

```
let book = {
    title: "Professional JavaScript",
    authors: [
        "Nicholas C. Zakas",
        "Matt Frisbie"
    ],
    edition: 4,
    year: 2017,
    toJSON: function() {
        return this.title;
    }
};

let jsonText = JSON.stringify(book);
```

Здесь метод `toJSON()`, определенный для объекта `book`, просто возвращает название книги. Подобно объекту `Date`, это значение сериализуется в простую строку, а не в объект. Из метода `toJSON()` можно возвратить любое сериализованное значение, и код будет работать правильно. Возвращение значения `undefined` приводит к установке значения в `null`, если объект вложен в другой объект, а если объект относится к верхнему уровню, просто возвращается `undefined`.

Обратите внимание, что для метода `toJSON()` не используется стрелочная функция. Это в первую очередь потому, что лексическая область действия стрелочной функции будет глобальной областью действия, что в данном примере бесполезно.

Метод `toJSON()` можно использовать вместе с функцией фильтрации, поэтому важно понимать, в каком порядке осуществляется сериализация. Когда объект передается в метод `JSON.stringify()`, выполняются следующие действия.

1. Для получения фактического значения вызывается метод `toJSON()`, если он доступен, в противном случае используется способ сериализации, предлагаемый по умолчанию.
2. Если в метод передан второй аргумент, применяется фильтр. В функцию фильтра передается значение, возвращенное на первом этапе.
3. Каждое значение, полученное на втором этапе, сериализуется надлежащим образом.
4. Если указан третий аргумент метода, выполняется форматирование.

Важно понимать этот порядок, когда требуется выбрать для сериализации метод `toJSON()`, функцию фильтра или комбинированный подход.

Параметры синтаксического анализа

Метод `JSON.parse()` тоже принимает дополнительный аргумент — функцию, которая вызывается для каждой пары ключа и значения. Ее называют *функцией восстановления* (reviver function), чтобы отличать от *функции замены* (replacer function), или фильтра, который передается в метод `JSON.stringify()`. Функция восстановления также принимает ключ и значение в качестве аргументов и должна возвращать значение.

Если функция восстановления возвращает значение `undefined`, ключ удаляется из результата; если она возвращает любое другое значение, оно вставляется в результат. Эту функцию часто используют для преобразования строк дат в объекты `Date`, например:

```
let book = {
    title: "Professional JavaScript",
    authors: [
        "Nicholas C. Zakas",
        "Matt Frisbie"
    ],
    edition: 4,
    year: 2017,
    releaseDate: new Date(2017, 11, 1)
};

let jsonText = JSON.stringify(book);

let bookCopy = JSON.parse(jsonText, (key, value) => key == "releaseDate"
               ? new Date(value) : value);

alert(bookCopy.releaseDate.getFullYear());
```

Этот код добавляет к объекту `book` свойство `releaseDate` типа `Date`. Далее объект сериализуется для получения допустимой JSON-строки, а затем преобразуется обратно в объект `bookCopy`. Функция восстановления ищет ключ `"releaseDate"` и в случае его обнаружения создает из его значения объект `Date`. Итоговое свойство `bookCopy.releaseDate` содержит объект `Date`, так что для него можно вызвать метод `getFullYear()`.

ИТОГИ

JSON — это компактный формат, с помощью которого можно легко представлять сложные структуры данных. Используемое в нем подмножество синтаксиса JavaScript поддерживает объекты, массивы, строки, числа, логические значения и значения `null`. Хотя XML предоставляет те же возможности, JSON-код более лаконичен и лучше поддерживается в JavaScript. Более того, встроенный объект JSON имеет хорошую поддержку во всех браузерах.

В ECMAScript 5 определен встроенный объект `JSON`, который служит для сериализации объектов в формат JSON и для преобразования данных JSON в JavaScript-объекты с помощью методов `JSON.stringify()` и `JSON.parse()` соответственно. Используя параметры этих методов, можно фильтровать значения или иным образом менять преобразование.

24

Сетевые запросы и удаленные ресурсы

- Использование объекта XMLHttpRequest
- Работа с событиями XMLHttpRequest
- Ограничения на использование Ajax между доменами
- Fetch API
- Streams API

Загрузки с сайта wrox.com для этой главы

Обратите внимание, что все примеры кода для этой главы доступны на веб-сайте книги по ссылке www.wrox.com/go/projavascript4e на вкладке Downloads.

В 2005 г. Джесс Джеймс Гарретт (Jesse James Garrett) опубликовал в интернете статью «Ajax: A New Approach to Web Applications» («Ajax: новый подход к программированию веб-приложений»). В этой статье он описал технологию, которую назвал *Ajax* (Asynchronous JavaScript+XML — «асинхронный JavaScript+XML»). Суть предложения заключалась в том, чтобы для улучшения пользовательского опыта запрашивать дополнительные данные у сервера без выгрузки веб-страницы. Гарретт объяснил, как эта технология может изменить традиционный подход «щелкни и жди».

Ключевым элементом Ajax является объект `XMLHttpRequest` (XHR), представленный корпорацией Microsoft, а затем продублированный другими производителями браузеров. До XHR для обмена данными в стиле Ajax приходилось прибегать

к различным хитростям — как правило, для этого использовали скрытые или встроенные фреймы. Объект XHR предоставил оптимизированный интерфейс для отправки запросов серверу и обработки его ответов. Это сделало возможным асинхронное получение дополнительной информации от сервера, то есть больше не нужно обновлять страницу. Вместо этого можно получить данные с помощью объекта XHR, а затем вставить их в страницу, используя DOM. Несмотря на то, что в полном названии Ajax упоминается XML, обмен данными с помощью Ajax не зависит от формата; сутью технологии является не использование XML, а получение данных от сервера без обновления страницы.

На самом деле эта технология уже была известна. Этот режим взаимодействия браузера и сервера, который называется *удаленным выполнением сценариев* (remote scripting), использовался в том или ином виде с 1998 г. Изначально запрашивать сервер из JavaScript можно было с помощью промежуточного компонента, такого как Java-апплет или Flash-ролик. Объект XHR предоставил разработчикам доступ к естественным средствам обмена данными браузера, что позволило работать эффективнее.

API объекта XHR считается сложным в использовании, и Fetch API, со времени его появления, расцвел в качестве модернизированной замены XHR. Поддержка промисов и служебных рабочих потоков сделала его невероятно мощным инструментом веб-разработки.

ПРИМЕЧАНИЕ Эта глава полностью охватывает `XMLHttpRequest`, но в целом это артефакт устаревших спецификаций JavaScript, и его следует использовать только для поддержки устаревших браузеров – по возможности используйте `fetch()`.

ОБЪЕКТ XMLHTTPREQUEST

Объект XHR был представлен в Internet Explorer 5 в виде AcitveX-объекта из MSXML-библиотеки. В браузере доступны три его версии: `MSXML2.XMLHttp`, `MSXML2.XMLHttp.3.0` и `MXSML2.XMLHttp.6.0`.

Все современные браузеры поддерживают встроенный объект XHR, который можно создать с помощью конструктора `XMLHttpRequest`:

```
let xhr = new XMLHttpRequest();
```

Использование объекта XHR

Чтобы приступить к работе с объектом XHR, нужно сначала вызвать метод `open()`, который принимает три аргумента: тип отправляемого запроса (`"get"`, `"post"` и т. п.), URL-адрес запроса и логическое значение, указывающее, нужно ли отправить запрос асинхронно, например:

```
xhr.open("get", "example.php", false);
```

Эта строка создает синхронный запрос GET страницы example.php. Имейте в виду, что URL-адрес интерпретируется относительно страницы, с которой вызывается код, хотя можно указать и абсолютный путь. Также учтите, что вызов open() не отправляет запрос, а только готовит его к отправке.

> **ПРИМЕЧАНИЕ** Доступны URL-адреса только из того же источника, то есть из того же домена с такими же номером порта и протоколом. Если какой-либо из этих элементов задан в URL-адресе иначе, чем на странице, с которой совершается запрос, возникнет ошибка безопасности.

Чтобы отправить указанный запрос, необходимо вызвать метод send():

```
xhr.open("get", "example.txt", false);
xhr.send(null);
```

Метод send() принимает в качестве аргумента данные, которые нужно отправить как тело запроса. Если тело отправлять не нужно, вы должны передать в метод значение null, потому что в некоторых браузерах этот аргумент обязателен. Как только метод send() вызван, запрос направляется серверу.

Поскольку запрос выполняется синхронно, выполнение JS-кода приостанавливается, пока не будет возвращен ответ. При получении ответа содержащиеся в нем данные назначаются свойствам объекта XHR. Нас интересуют следующие свойства:

- responseText — текст, возвращенный как тело ответа;
- responseXML — DOM-документ с данными ответа, если ответ содержит контент типа "text/xml" или "application/xml";
- status — HTTP-состояние ответа;
- statusText — описание HTTP-состояния.

При получении ответа первым делом нужно проверить его свойство status, чтобы убедиться, что он был возвращен успешно. Как правило, успешно возвращенные ответы имеют 200-е коды HTTP-состояния, и если тип контента правилен, в этом случае какой-то контент будет содержаться в свойстве responseText и, возможно, в responseXML. Код состояния 304 указывает, что ресурс не был изменен и выдается из кеша браузера; это также означает, что ответ доступен. Чтобы убедиться в получении правильного ответа, следует проверить все эти состояния:

```
xhr.open("get", "example.txt", false);
xhr.send(null);

if ((xhr.status >= 200 && xhr.status < 300) || xhr.status == 304) {
    alert(xhr.responseText);
} else {
    alert("Request was unsuccessful: " + xhr.status);
}
```

Этот код выводит на экран либо контент, возвращенный сервером, либо сообщение об ошибке, в зависимости от возвращенного кода состояния. Рекомендуется

всегда проверять свойство `status` для определения оптимальных действий и не использовать для этого свойство `statusText`, потому что в некоторых браузерах оно работает ненадежно. Свойству `responseText` независимо от типа контента всегда назначается тело ответа, а свойство `responseXML` имеет значение `null`, если тип данных отличен от XML.

Вместо синхронных запросов, наподобие предыдущего, обычно лучше использовать асинхронные, потому что они не требуют приостанавливать выполнение JS-кода до получения ответа. У объекта XHR есть свойство `readyState`, которое указывает текущий этап цикла «запрос-ответ». Оно может иметь следующие значения:

- 0 — состояние не инициализировано. Метод `open()` еще не вызывался.
- 1 — запрос создан. Метод `open()` был вызван, а метод `send()` — нет.
- 2 — запрос отправлен. Метод `send()` был вызван, но ответ еще не получен.
- 3 — идет получение ответа. Некоторые данные ответа получены.
- 4 — запрос выполнен. Все данные ответа получены и доступны.

Когда изменяется значение свойства `readyState`, генерируется событие `readystatechange`, в обработчике которого можно проверить значение `readyState`. Вообще говоря, нас интересует только значение `4`, которое указывает, что все данные готовы. Обработчик события `readystatechange` необходимо задать до вызова метода `open()`, чтобы код работал во всех браузерах. Рассмотрим пример:

```
let xhr = new XMLHttpRequest();
xhr.onreadystatechange = function() {
    if (xhr.readyState == 4) {
        if ((xhr.status >= 200 && xhr.status < 300) || xhr.status == 304) {
            alert(xhr.responseText);
        } else {
            alert("Request was unsuccessful: " + xhr.status);
        }
    }
};
xhr.open("get", "example.txt", true);
xhr.send(null);
```

Обратите внимание, что этот код подключает обработчик события к объекту `xhr` в стиле DOM Level 0, потому что не все браузеры поддерживают подход DOM Level 2. В отличие от других обработчиков событий, в обработчик `onreadystatechange` объект `event` не передается. Вместо этого для определения дальнейших действий нужно использовать сам объект XHR.

ПРИМЕЧАНИЕ В этом примере в обработчике события `readystatechange` используется объект XHR, а не `this` из-за проблем с областью видимости обработчика. Использование `this` может вызвать в некоторых браузерах сбой в работе функции, так что безопаснее использовать переменную экземпляра XHR.

С помощью метода `abort()` можно отменить асинхронный запрос до получения ответа:

```
xhr.abort();
```

При вызове этого метода объект XHR прекращает генерировать события; кроме того, при этом блокируется доступ к любым его свойствам, связанным с ответом. Как только запрос отменен, следует присвоить ссылке на объект XHR значение `null`. Из-за возможных проблем с памятью повторно использовать объект XHR не рекомендуется.

Заголовки HTTP

Вместе с каждым HTTP-запросом и HTTP-ответом отправляется заголовочная информация, которая может быть полезна разработчику. Для доступа к заголовкам запроса и ответа используются методы объекта XHR.

По умолчанию при отправке XHR-запроса отправляются следующие заголовки:

- `Accept` — типы контента, которые браузер может обработать.
- `Accept-Charset` — кодировки, поддерживаемые браузером.
- `Accept-Encoding` — способы сжатия, поддерживаемые браузером.
- `Accept-Language` — языки, поддерживаемые браузером.
- `Connection` — тип подключения браузера к серверу.
- `Cookie` — cookie-файлы страницы.
- `Host` — домен страницы, которая инициирует запрос.
- `Referer` — URI страницы, инициирующей запрос. В спецификации HTTP имя этого заголовка содержит ошибку, которую приходится воспроизводить для обеспечения совместимости (правильное написание — «referrer»).
- `User-Agent` — строка пользовательского агента браузера.

Хотя набор отправляемых заголовков запроса зависит от браузера, некоторые заголовки отправляются в большинстве случаев. Дополнительные заголовки запроса можно задать с помощью метода `setRequestHeader()`, который принимает имя заголовка и его значение. Для отправки заголовков запроса нужно вызвать метод `setRequestHeader()` после метода `open()`, но до `send()`, например:

```
let xhr = new XMLHttpRequest();
xhr.onreadystatechange = function() {
    if (xhr.readyState == 4) {
        if ((xhr.status >= 200 && xhr.status < 300) || xhr.status == 304) {
            alert(xhr.responseText);
        } else {
            alert("Request was unsuccessful: " + xhr.status);
        }
    }
};
```

```
xhr.open("get", "example.php", true);
xhr.setRequestHeader("MyHeader", "MyValue");
xhr.send(null);
```

С помощью пользовательских заголовков запроса можно указывать серверу правильный план действий. Рекомендуется всегда указывать собственные имена заголовков вместо тех, что обычно отправляет браузер, потому что заголовки, предлагаемые по умолчанию, могут влиять на ответ сервера. Одни браузеры позволяют перезаписывать заголовки, предлагаемые по умолчанию, а другие — нет.

Получить заголовок ответа из объекта XHR можно методом `getResponseHeader()`, передав в него имя нужного заголовка. Также можно получить все заголовки как одну строку с помощью метода `getAllResponseHeaders()`. Вот пример с обоими методами:

```
let myHeader = xhr.getResponseHeader("MyHeader");
let allHeaders = xhr.getAllResponseHeaders();
```

Заголовки можно использовать для передачи дополнительных структурированных данных от сервера браузеру. Метод `getAllResponseHeaders()` обычно возвращает данные вроде следующих:

```
Date: Sun, 14 Nov 2004 18:04:03 GMT
Server: Apache/1.3.29 (Unix)
Vary: Accept
X-Powered-By: PHP/4.3.8
Connection: close
Content-Type: text/html; charset=iso-8859-1
```

Получив эти данные, можно выполнить их синтаксический анализ и найти все имена отправленных заголовков вместо того, чтобы проверять наличие каждого заголовка по отдельности.

Запросы GET

Из разных типов запросов наиболее востребованы запросы GET, которые обычно используются, если нужно запросить у сервера ту или иную информацию. При необходимости к URL-адресу в запросе можно добавить аргументы строки запроса для передачи информации серверу. При работе с XHR строка запроса обязательна и должна быть правильно закодирована в URL-адресе, который передается в метод `open()`.

Одной из наиболее частых ошибок при работе с запросами GET является неправильное форматирование строки запроса. Все имена и значения в строке запроса должны быть закодированы с помощью метода `encodeURIComponent()` перед их добавлением к URL-адресу, а все пары имен и значений должны быть разделены амперсандом, например:

```
xhr.open("get", "example.php?name1=value1&name2=value2", true);
```

Для добавления аргументов строки запроса к существующему URL-адресу можно использовать следующую вспомогательную функцию:

```
function addURLParam(url, name, value) {
    url += (url.indexOf("?") == -1 ? "?" : "&");
    url += encodeURIComponent(name) + "=" + encodeURIComponent(value);
    return url;
}
```

Функция `addURLParam()` принимает три аргумента: URL-адрес, к которому нужно добавить параметр, имя параметра и его значение. Первым делом функция проверяет, содержит ли URL-адрес вопросительный знак (чтобы определить, есть ли другие параметры в URL-адресе). Если нет, функция присоединяет к адресу вопросительный знак, в противном случае добавляется амперсанд. Затем имя и значение кодируются и добавляются к концу URL-адреса. Наконец, функция возвращает обновленный URL-адрес.

Эту функцию можно использовать для составления URL-адреса запроса следующим образом:

```
let url = "example.php";

// добавление аргументов
url = addURLParam(url, "name", "Nicholas");
url = addURLParam(url, "book", "Professional JavaScript");

// инициирование запроса
xhr.open("get", url, false);
```

Функция `addURLParam()` гарантирует, что строка запроса будет правильно отформатирована для использования с объектом XHR.

Запросы POST

Вторым по популярности типом запросов является запрос POST. Он обычно используется, если нужно отправить серверу данные, которые должны быть сохранены. Предполагается, что тело каждого запроса POST содержит данные, тогда как запросы GET обычно их не содержат. Тело запроса POST может содержать большой объем данных любого формата. Инициировать запрос POST можно, указав `"post"` в качестве первого аргумента метода `open()`, например:

```
xhr.open("post", "example.php", true);
```

Далее нужно передать некоторые данные в метод `send()`. Поскольку объект XHR был разработан для работы с XML, вы можете передать в метод XML-документ DOM, который будет сериализован и отправлен как тело запроса. Кроме того, можно передать в метод любую строку, которую требуется отправить серверу.

По умолчанию запрос POST не воспринимается сервером как отправка веб-формы. Чтобы получить переданные данные, сервер должен прочитать необработанные данные запроса. Тем не менее вы можете имитировать отправку формы с помощью объекта XHR. Сначала для этого нужно присвоить заголовку `Content-Type` значение `application/x-www-form-urlencoded`, которое представляет тип контента, задаваемый при отправке формы. Затем нужно создать строку соответствующего

формата. Данные запроса POST отправляются в том же формате, что и строка запроса. Если форму, уже имеющуюся на странице, нужно сериализовать и отправить серверу с помощью объекта XHR, для создания соответствующей строки можно использовать функцию `serialize()`, упоминавшуюся в главе 14:

```
function submitData() {
    let xhr = new XMLHttpRequest();
    xhr.onreadystatechange = function() {
        if (xhr.readyState == 4) {
            if ((xhr.status >= 200 && xhr.status < 300) ||
                     xhr.status == 304) {
                alert(xhr.responseText);
            } else {
                alert("Request was unsuccessful: " + xhr.status);
            }
        }
    };

    xhr.open("post", "postexample.php", true);
    xhr.setRequestHeader("Content-Type",
                         "application/x-www-form-urlencoded");
    let form = document.getElementById("user-info");
    xhr.send(serialize(form));
}
```

Эта функция сериализует данные формы с идентификатором `"user-info"` и отправляет их серверу. После этого отправленные данные можно получить в PHP-файле `postexample.php` с помощью переменной `$_POST`. Рассмотрим пример:

```
<?php
    header("Content-Type: text/plain");
    echo <<<EOF
Имя: {$_POST['user-name']}
Эл. почта: {$_POST['user-email']}
EOF;
?>
```

Если бы мы не добавили заголовок `Content-Type`, данные не были бы доступны в суперглобальной переменной `$_POST` и для доступа к ним пришлось бы использовать переменную `$HTTP_RAW_POST_DATA`.

> **ПРИМЕЧАНИЕ** Выполнение запросов POST требует больше ресурсов в сравнении с запросами GET. В плане быстродействия запросы GET могут выполняться до двух раз быстрее, чем запросы POST, отправляющие тот же объем данных.

XMLHTTPREQUEST LEVEL 2

Популярность объекта XHR в качестве стандарта подтолкнула консорциум W3C к созданию официальных спецификаций для регламентации его поведения. В спецификации XMLHttpRequest Level 1 были определены детали уже существующей

реализации объекта XHR, а в XMLHttpRequest Level 2 он был расширен. Не во всех браузерах спецификация Level 2 реализована полностью, но в той или иной степени ее поддерживает каждый браузер.

Тип FormData

В современных веб-приложениях часто требуется сериализовать данные формы, и для этого в спецификации XMLHttpRequest Level 2 представлен тип `FormData`. Он позволяет с легкостью выполнять сериализацию существующих форм и создавать данные в том же формате, что и в форме, для их передачи с помощью объекта XHR. Следующий код создает объект `FormData` и заполняет его некоторыми данными:

```
let data = new FormData();
data.append("name", "Nicholas");
```

Метод `append()` принимает в качестве аргументов ключ и значение — по сути, имя поля формы и содержащееся в нем значение. Вы можете добавить к объекту любое количество таких пар. Кроме того, можно предварительно заполнить пары ключей и значений данными элемента формы, передав элемент в конструктор `FormData`:

```
let data = new FormData(document.forms[0]);
```

Как только у вас есть экземпляр `FormData`, его можно передать непосредственно в метод `send()` объекта XHR, например:

```
let xhr = new XMLHttpRequest();
xhr.onreadystatechange = function() {
    if (xhr.readyState == 4) {
        if ((xhr.status >= 200 && xhr.status < 300) || xhr.status == 304) {
            alert(xhr.responseText);
        } else {
            alert("Request was unsuccessful: " + xhr.status);
        }
    }
};

xhr.open("post", "postexample.php", true);
let form = document.getElementById("user-info");
xhr.send(new FormData(form));
```

Тип `FormData` удобен тем, что вам не нужно явно задавать заголовки запроса для объекта XHR. Объект XHR распознает тип переданных данных как экземпляр `FormData` и настраивает заголовки соответствующим образом.

Тип `FormData` поддерживается в Firefox 4+, Safari 5+, Chrome и WebKit для Android 3+.

Тайм-ауты

В Internet Explorer 8 объект XHR был расширен свойством `timeout`, которое указывает время ожидания ответа на запрос перед его отменой. С тех пор все браузеры

приняли это свойство в своих реализациях XHR. Если свойство `timeout` задано, и ответ не получен в течение указанного количества миллисекунд, генерируется событие `timeout` и вызывается обработчик `ontimeout`. Позднее этот функционал был добавлен в спецификацию XMLHttpRequest Level 2. Рассмотрим пример:

```
let xhr = new XMLHttpRequest();
xhr.onreadystatechange = function() {
    if (xhr.readyState == 4) {
        try {
            if ((xhr.status >= 200 && xhr.status < 300) ||
                    xhr.status == 304) {
                alert(xhr.responseText);
            } else {
                alert("Request was unsuccessful: " + xhr.status);
            }
        } catch (ex) {
            // предполагается, что исключения обрабатываются
            // в обработчике ontimeout
        }
    }
};

xhr.open("get", "timeout.php", true);
xhr.timeout = 1000;        // тайм-аут — 1 секунда
xhr.ontimeout = function() {
    alert("Request did not return in a second.");
};
xhr.send(null);
```

Этот пример поясняет использование свойства `timeout`. Присвоение ему значения в 1000 миллисекунд означает, что если ответ на запрос не возвратится за 1 секунду или раньше, запрос отменяется, при этом вызывается обработчик события `timeout`. Чтобы можно было сравнить свойство `readyState` со значением `4`, обработчик события `readystatechange` все же должен быть вызван, однако при попытке доступа к свойству `status` после тайм-аута возникнет ошибка. Для предотвращения сбоя следует инкапсулировать код, проверяющий свойство `status`, в инструкцию `try-catch`.

Метод overrideMimeType()

Метод `overrideMimeType()` был введен в Firefox для переопределения MIME-типа запроса XHR, позднее он был добавлен в спецификацию XMLHttpRequest Level 2. Поскольку от MIME-типа ответа зависит способ обработки ответа объектом XHR, возможность переопределить тип, возвращенный сервером, бывает весьма кстати.

Рассмотрим ситуацию, когда сервер отправляет MIME-тип `text/plain`, который на самом деле содержит XML-код. В этом случае свойство `responseXML` имеет значение `null`, хотя ответ содержит XML-код. Вызвав метод `overrideMimeType()`, вы можете гарантировать, что ответ будет обработан как XML-данные, а не как обычный текст:

```
let xhr = createXHR();
xhr.open("get", "text.php", true);
xhr.overrideMimeType("text/xml");
xhr.send(null);
```

Этот код указывает объекту XHR обработать ответ как XML-данные, а не как обычный текст. Чтобы правильно переопределить MIME-тип ответа, нужно вызвать метод `overrideMimeType()` до метода `send()`.

СОБЫТИЯ ХОДА ОБМЕНА ДАННЫМИ

Спецификация Progress Events от W3C — это рабочий проект, определяющий события обмена данными между клиентом и сервером. Сначала эти события были явно ориентированы на XHR, но теперь входят и в другие похожие API. Событий обмена данными шесть:

- `loadstart` — генерируется при получении первого байта ответа;
- `progress` — многократно генерируется во время получения ответа;
- `error` — генерируется, если при попытке выполнить запрос происходит ошибка;
- `abort` — генерируется при завершении подключения с помощью метода `abort()`;
- `load` — генерируется, когда ответ полностью получен;
- `loadend` — генерируется при завершении обмена данными и после событий `error`, `abort` и `load`.

Каждый запрос начинается с события `loadstart`, за которым следует одно или несколько событий `progress`. После этого генерируется событие `error`, `abort` или `load`, и наконец, все завершается событием `loadend`.

Большинство этих событий просты, но два из них имеют нюансы, заслуживающие отдельного внимания.

Событие load

Объект XHR был реализован в Firefox, чтобы упростить модель взаимодействия. В связи с этим было представлено событие `load` для замены события `readystatechange`. Событие `load` генерируется, когда ответ полностью получен, и устраняет необходимость проверки свойства `readyState`. Обработчик события `load` получает объект `event`, у которого свойство `target` содержит экземпляр объекта XHR со всеми свойствами и методами. Не во всех браузерах объект `event` этого события реализован правильно, из-за чего необходимо использовать саму переменную объекта XHR, например:

```
let xhr = new XMLHttpRequest();
xhr.onload = function() {
    if ((xhr.status >= 200 && xhr.status < 300) || xhr.status == 304) {
```

```
        alert(xhr.responseText);
    } else {
        alert("Request was unsuccessful: " + xhr.status);
    }
};
xhr.open("get", "altevents.php", true);
xhr.send(null);
```

При получении ответа от сервера событие `load` генерируется независимо от состояния. Это означает, что вы должны проверить свойство `status`, чтобы определить, доступны ли нужные данные. Событие `load` поддерживается в Firefox, Opera, Chrome и Safari.

Событие progress

Другой инновацией, реализованной в объекте XHR компанией Mozilla, является событие `progress`, которое периодически генерируется, когда браузер получает новые данные. В обработчик события `progress` передается объект `event`, который содержит объект XHR в свойстве `target` и имеет три дополнительных свойства:

- `lengthComputable` — логическое значение, которое указывает, доступна ли информация о ходе выполнения операции;
- `position` — количество уже полученных байтов;
- `totalSize` — общее количество ожидаемых байтов согласно заголовку ответа `Content-Length`.

Имея на руках эту информацию, можно вывести для пользователя индикатор обмена данными:

```
let xhr = new XMLHttpRequest();
xhr.onload = function(event) {
    if ((xhr.status >= 200 && xhr.status < 300) || xhr.status == 304) {
        alert(xhr.responseText);
    } else {
        alert("Request was unsuccessful: " + xhr.status);
    }
};
xhr.onprogress = function(event) {
    let divStatus = document.getElementById("status");
    if (event.lengthComputable) {
        divStatus.innerHTML = "Received " + event.position + " of " +
            event.totalSize + " bytes";
    }
};

xhr.open("get", "altevents.php", true);
xhr.send(null);
```

Для правильного выполнения обработчика `onprogress` нужно назначить его до вызова метода `open()`. В предыдущем примере HTML-элемент заполняется информацией о состоянии запроса каждый раз, когда генерируется событие `progress`. Если ответ

содержит заголовок `Content-Length`, вы также можете использовать эту информацию для вычисления доли полученных данных относительно полного ответа.

ОБМЕН РЕСУРСАМИ С ЗАПРОСОМ ПРОИСХОЖДЕНИЯ

Одним из главных ограничений взаимодействия в стиле Ajax с помощью объекта XHR является политика безопасности, регламентирующая доступ к разным источникам. По умолчанию объектам XHR доступны ресурсы только в том домене, к которому относится содержащая их веб-страница. Этот механизм защиты предотвращает некоторые злонамеренные действия. Однако потребность в правомочном доступе к ресурсам из разных доменов была настолько острой, что разработчики начали предлагать соответствующие решения для браузеров.

Обмен ресурсами с запросом происхождения (Cross-Origin Resource Sharing, CORS) определяет способ взаимодействия браузера и сервера при доступе к источникам из других доменов. В основе CORS лежит применение пользовательских HTTP-заголовков с той целью, чтобы браузер и сервер могли получить друг о друге достаточно информации и выяснить, должен ли запрос или ответ завершиться успехом или неудачей.

Простой запрос GET или POST без пользовательских заголовков и с телом формата `text/plain` отправляется с дополнительным заголовком `Origin`. Он содержит источник (протокол, имя домена и порт) страницы, инициирующей запрос, чтобы сервер мог легко определить, должен ли он возвратить ответ. Заголовок `Origin` может выглядеть так:

```
Origin: http://www.nczonline.net
```

Если сервер решает, что запрос допустим, он отправляет браузеру заголовок `Access-Control-Allow-Origin`, дублируя информацию об источнике или возвращая `"*"`, если ресурс общедоступен, например:

```
Access-Control-Allow-Origin: http://www.nczonline.net
```

Если этот заголовок отсутствует или источники в запросе и ответе не соответствуют друг другу, браузер блокирует запрос, в противном случае запрос обрабатывается. Ни запросы, ни ответы не включают файлы cookie.

В современных браузерах поддержка CORS реализована в виде объекта `XMLHttpRequest`. При попытке открыть ресурс из другого источника этот объект автоматически задействуется без дополнительного кода. Чтобы запросить ресурс из другого домена, следует использовать стандартный объект XHR и передать в метод `open()` абсолютный URL-адрес, например:

```
let xhr = new XMLHttpRequest();
xhr.onreadystatechange = function() {
```

```
    if (xhr.readyState == 4) {
        if ((xhr.status >= 200 && xhr.status < 300) || xhr.status == 304) {
            alert(xhr.responseText);
        } else {
            alert("Request was unsuccessful: " + xhr.status);
        }
    }
};
xhr.open("get", "http://www.somewhere-else.com/page/", true);
xhr.send(null);
```

Кроссдоменный объект XHR обеспечивает доступ к свойствам `status` и `statusText` и поддерживает синхронные запросы. Однако у него есть некоторые дополнительные ограничения, служащие для обеспечения безопасности:

- пользовательские заголовки нельзя задать с помощью метода `setRequestHeader()`;
- cookie-файлы не отправляются и не принимаются;
- метод `getAllResponseHeaders()` всегда возвращает пустую строку.

Поскольку и у обычных, и у кроссдоменных запросов один интерфейс, лучше всегда использовать относительный URL-адрес для доступа к локальному ресурсу и абсолютный — для доступа к удаленному. Это делает код однозначным и помогает предотвращать такие проблемы, как ограничение доступа к заголовкам и (или) cookie-файлам для локальных ресурсов.

Предварительные запросы

CORS позволяет применять пользовательские заголовки, методы, отличные от GET или POST, и разные типы контента в теле запроса. Для этого служит прозрачный механизм верификации сервера, который называется *предварительными запросами* (preflighted requests). Предварительный запрос сервера получается, когда вы пытаетесь выполнить запрос с одним из расширенных параметров. Такой запрос осуществляется по методу OPTIONS и содержит следующие заголовки:

- `Origin` — то же, что и в простых запросах;
- `Access-Control-Request-Method` — метод выполнения запроса;
- `Access-Control-Request-Headers` (необязательный заголовок) — список пользовательских заголовков, разделенных запятыми.

Вот пример запроса POST с пользовательским заголовком `NCZ`:

```
Origin: http://www.nczonline.net
Access-Control-Request-Method: POST
Access-Control-Request-Headers: NCZ
```

Получив такой запрос, сервер может определить, следует ли разрешить запросы этого типа, а затем сообщает о принятом решении браузеру, отправляя в ответе следующие заголовки:

- `Access-Control-Allow-Origin` — то же, что и в простых запросах;
- `Access-Control-Allow-Methods` — список разрешенных методов, разделенных запятыми;
- `Access-Control-Allow-Headers` — список разрешенных сервером заголовков, разделенных запятыми;
- `Access-Control-Max-Age` — время в секундах, в течение которого предварительный запрос должен находиться в кеше.

Вот пример:

```
Access-Control-Allow-Origin: http://www.nczonline.net
Access-Control-Allow-Methods: POST, GET
Access-Control-Allow-Headers: NCZ
Access-Control-Max-Age: 1728000
```

Когда предварительный запрос обработан, результат кешируется на время, указанное в ответе; то есть накладные расходы, связанные с дополнительным HTTP-запросом, имеют место только при первом запросе данного типа.

Запросы с учетными данными

По умолчанию запросы ресурсов из других доменов не предоставляют учетные данные (cookie-файлы, данные для проверки подлинности HTTP или клиентские SSL-сертификаты). Вы можете указать, что в запросе нужно отправить учетные данные, присвоив свойству `withCredentials` значение `true`. Если сервер поддерживает запросы с учетными данными, он возвратит ответ со следующим HTTP-заголовком:

```
Access-Control-Allow-Credentials: true
```

Если в ответе на запрос с учетными данными нет этого заголовка, браузер не передает ответ JS-сценарию (при этом свойство `responseText` является пустой строкой, свойство `status` равно 0, кроме того, вызывается обработчик `onerror()`). Имейте в виду, что сервер также может отправить этот HTTP-заголовок в ответе на предварительный запрос, показывая, что источнику разрешено отправлять запросы с учетными данными.

АЛЬТЕРНАТИВНЫЕ МЕТОДИКИ КРОССДОМЕННОГО ВЗАИМОДЕЙСТВИЯ

До появления CORS реализовать кроссдоменное взаимодействие в стиле Ajax было сложнее, поэтому для выполнения некоторых кроссдоменных запросов без объекта XHR разработчики использовали возможности DOM. Несмотря на повсеместное

распространение CORS, эти методики все еще популярны, потому что они не требуют внесения изменений на стороне сервера.

Проверка связи с помощью изображения

Одна из первых методик кроссдоменного взаимодействия была основана на использовании тега `<img>`. С любой страницы можно загружать изображения из других доменов, не беспокоясь об ограничениях, например, это основной способ отслеживания показов рекламных объявлений. Вы также можете динамически создавать изображения и использовать их обработчики `onload` и `onerror` для регистрации получения ответов.

Динамическое создание изображений часто используется для *проверки связи с помощью изображения* (image ping). Так называют простое кроссдоменное однонаправленное взаимодействие с сервером. Данные при этом отправляются с помощью аргументов строки запроса, а ответ может быть любым, хотя обычно им является пиксельное изображение или ответ 204. При использовании этого приема браузер не может получить никакие специфические данные, зато он может узнать о получении ответа по событиям `load` и `error`. Вот простой пример:

```
let img = new Image();
img.onload = img.onerror = function() {
    alert("Done!");
};
img.src = "http://www.example.com/test?name=Nicholas";
```

Этот код создает экземпляр `Image`, а затем задает одну функцию в качестве обработчика событий `load` и `error`. Это гарантирует, что независимо от ответа вы будете уведомлены о завершении запроса. Запрос начинается при установке свойства `src`, при этом в данном примере с ним отправляется параметр `name`.

Проверка связи с помощью изображения обычно применяется для отслеживания щелчков пользователя на странице или для динамического показа рекламы. Два главных недостатка этого приема заключаются в том, что вы можете отправлять только запросы GET и не можете получить доступ к тексту ответа с сервера. По этим причинам данный прием лучше всего подходит для однонаправленного взаимодействия браузера с сервером.

JSONP

JSONP (JSON with padding — JSON с набивкой) называют специальный вариант формата JSON, который стал популярен в веб-сервисах. JSONP-код выглядит как JSON-код, за исключением того, что данные заключены в синтаксический аналог вызова функции, например:

```
callback({ "name": "Nicholas" });
```

Синтаксис JSONP состоит из двух частей: функции обратного вызова и данных. Функция обратного вызова выполняется для страницы при получении ответа. Обычно имя этой функции указывается как часть запроса. Данные — это просто JSON-данные, передаваемые функции. Типичный JSONP-запрос выглядит так:

```
http://freegeoip.net/json/?callback=handleResponse
```

Это URL-адрес JSONP-сервиса геолокации. Функцию обратного вызова в JSONP-сервисах часто указывают как аргумент строки запроса; в данном случае она называется `handleResponse()`.

JSONP используется с динамическими элементами `<script>`, при этом атрибут `src` назначается кроссдоменному URL-адресу. Подобно элементу `<img>`, элемент `<script>` может загружать ресурсы из других доменов без ограничений. Благодаря тому, что JSONP является допустимым JS-кодом, ответ JSONP извлекается на страницу и незамедлительно выполняется по завершении запроса. Вот пример:

```
function handleResponse(response) {
    console.log(`
                You're at IP address ${response.ip}, which is in
                ${response.city}, ${response.region_name}`);
}

let script = document.createElement("script");
script.src = "http://freegeoip.net/json/?callback=handleResponse";
document.body.insertBefore(script, document.body.firstChild);
```

Этот код выводит на экран ваш IP-адрес и сведения о расположении, полученные от сервиса геолокации.

Формат JSONP очень популярен среди веб-разработчиков благодаря простоте и удобству его использования. Он привлекательнее проверки связи с помощью изображения тем, что текст ответа доступен напрямую, что делает возможным двунаправленное взаимодействие между браузером и сервером. Однако JSONP имеет и недостатки.

Первый недостаток заключается в том, что вы извлекаете исполняемый код на свою страницу из другого документа. Если этот домен не является доверенным, он может легко заменить ответ вредоносным кодом, и у вас не будет иного выхода, кроме как удалить JSONP-вызов. При использовании веб-сервиса, которым управляете не вы, убедитесь, что его источник заслуживает доверия.

Вторым недостатком является то, что нет простого способа определить, что JSONP-запрос завершился неудачей. Хотя в HTML5 определен обработчик события `error` для элементов `<script>`, он еще не реализован в браузерах. Разработчики часто использовали таймеры, чтобы узнать, получен ли ответ за указанное время, но этот способ недостаточно хорош, потому что подключения различаются по скорости и пропускной способности.

FETCH API

Fetch API может выполнять все те же задачи, что и объект `XMLHttpRequest`, но его гораздо проще использовать, он имеет более современный интерфейс и может использоваться современными веб-инструментами, такими как рабочие потоки. Принимая во внимание, что `XMLHttpRequest` может быть асинхронным, все запросы, отправляемые Fetch API, являются строго асинхронными. Fetch API — это спецификация стандарта WHATWG, который можно найти по адресу https://fetch.spec.whatwg.org/. В спецификации определение звучит лучше всего: *«Стандарт Fetch определяет запросы, ответы и процесс, который их связывает: получение данных»*.

Fetch API сам по себе является превосходным инструментом для запроса ресурсов в JavaScript, но этот API также важен для рабочих потоков служб в том смысле, что он обеспечивает интерфейс для перехвата, перенаправления и изменения запросов, выполняемых посредством метода `fetch()`.

Основы использования API

Метод `fetch()` доступен в любой глобальной области видимости, как при выполнении кода основной страницы, в модулях, так и внутри рабочих потоков. Запустив его, браузер отправит запрос на указанный URL-адрес.

Отправка запроса

Метод `fetch()` имеет только один обязательный параметр `input`, который в большинстве случаев будет URL-адресом ресурса, который нужно получить. Этот метод возвращает промис:

```
let r = fetch('/bar');
console.log(r); // Promise <pending>
```

Интерпретация этого URL (относительные пути, абсолютные пути и т. д.) выполняется идентично интерпретации запроса XHR.

Когда запрос завершится и ресурс станет доступен, промис преобразуется в объект `Response`, который используется в качестве оболочки API для любого ресурса, который был выбран. Этот объект предоставляет ряд свойств и методов для проверки ответа и преобразования полезной нагрузки в форму, как показано здесь:

```
fetch('bar.txt')
    .then((response) => {
        console.log(response);
    });

// Response { type: "basic", url: ... }
```

Чтение ответа

Простейший способ прочитать содержимое ответа — использование простого текстового формата, доступ к которому осуществляется с помощью метода `text()`.

Этот метод возвращает промис, который возвращается с полным содержимым извлеченного ресурса:

```
fetch('bar.txt')
    .then((response) => {
        response.text().then((data) => {
            console.log(data);
        });
    });

// Содержимое bar.txt!
```

Эта структура промиса обычно сглаживается:

```
fetch('bar.txt')
    .then((response) => response.text())
    .then((data) => console.log(data));

// Содержимое bar.txt!
```

Обработка кодов состояния и ошибок запросов

Fetch API позволяет проверять код состояния и текст состояния объекта `Response`, доступные через свойства `status` и `statusText` соответственно. Успешная загрузка ресурса обычно приводит к коду ответа 200, как показано в этом примере:

```
fetch('/bar')
    .then((response) => {
        console.log(response.status);        // 200
        console.log(response.statusText);    // OK
    });
```

Запрос несуществующего ресурса обычно приводит к коду ответа 404:

```
fetch('/does-not-exist')
    .then((response) => {
        console.log(response.status);        // 404
        console.log(response.statusText);    // Not Found
    });
```

Запрос URL-адреса ресурса, который выдает ошибку сервера, обычно выдает код ответа 500:

```
fetch('/throw-server-error')
    .then((response) => {
        console.log(response.status);        // 500
        console.log(response.statusText);    // Internal Server Error
    });
```

Поведение `fetch()` по отношению к перенаправлениям может быть задано явно (как описано далее в этой главе), но поведение по умолчанию — следовать по адресу перенаправления и возвращать ответ, который не имеет кода ответа 300–399. При извлечении ответа свойство `redirected` устанавливается равным `true` для объекта ответа, но все равно будет содержать код ответа 200:

```
fetch('/permanent-redirect')
    .then((response) => {
        // Поведение по умолчанию состоит в том, чтобы следовать перенаправлениям
        // до достижения конечного URL.
        // Этот пример будет вызывать как минимум два запроса туда и обратно:
        // <origin url>/permanent-redirect -> <redirect url>
        console.log(response.status); // 200
        console.log(response.statusText); // OK
        console.log(response.redirected); // true
    });
```

Во всех этих примерах обратите внимание, что *разрешенный* обработчик промиса выполняется, даже если запрос может рассматриваться как сбой, например, состояние 500. Если сервер отправит ответ любого рода, промис `fetch()` разрешится. Такое поведение должно иметь смысл: сетевой протокол системного уровня завершил успешную передачу сообщения в оба конца. То, что квалифицируется как «успешный» запрос, должно быть определено в том, как обрабатывается этот ответ.

Обычно ответ в 200 считается успешным, а все остальное считается неудачным. Чтобы различать их, свойство `ok` объекта `Response` определяет, когда код ответа находится между 200–299.

```
fetch('/bar')
    .then((response) => {
    console.log(response.status); // 200
    console.log(response.ok); // true
});
fetch('/does-not-exist')
    .then((response) => {
    console.log(response.status); // 404
    console.log(response.ok); // false
 });
```

Истинная ошибка `fetch()`, например тайм-аут браузера при отсутствии ответа от сервера, будет отклонена:

```
fetch('/hangs-forever')
    .then((response) => {
        console.log(response);
    }, (err) => {
        console.log(err);
    });

// (после тайм-аута браузера)
// TypeError: "NetworkError when attempting to fetch resource."
```

Промис запроса отклоняется по таким причинам, как нарушения CORS, отсутствие подключения к сети, нарушения HTTPS и другие нарушения политики браузера или сети.

Можно проверить полный URL-адрес, используемый `fetch()` при отправке запроса с помощью свойства `url`:

```
// Запрос, сделанный с foo.com/bar/baz
console.log(window.location.href); // https://foo.com/bar/baz

fetch('qux').then((response) => console.log(response.url));
// https://foo.com/bar/qux

fetch('/qux').then((response) => console.log(response.url));
// https://foo.com/qux

fetch('//qux.com').then((response) => console.log(response.url));
// https://qux.com

fetch('https://qux.com').then((response) => console.log(response.url));
// https://qux.com
```

Пользовательские параметры fetch()

При использовании только с URL-адресом `fetch()` отправляет запрос GET с минимальным набором заголовков запроса. Чтобы настроить способ отправки запроса, объект `init` может быть передан в качестве необязательного второго аргумента функции `fetch()`. Объект `init` должен быть заполнен любым количеством ключей и соответствующих значений, перечисленных в следующей таблице.

КЛЮЧ	ЗНАЧЕНИЕ
`body`	Используется для указания поля тела для запросов, которые используют тело. Должен принадлежать одному из следующих типов: `Blob`, `BufferSource`, `FormData`, `URLSearchParams`, `ReadableStream` или `string`

`cache`	Используется для управления взаимодействием браузера с кешем HTTP при выполнении получения данных. Чтобы кешированные перенаправления выполнялись, в качестве значения `follow` должно быть указано свойство `redirect`, а также должны соблюдаться ограничения того же источника. Должен иметь одно из следующих `string`-значений:
	`default`
	• Из `fetch()` возвращается новое попадание в кеш. Запрос не отправлен.
	• Устаревшее попадание в кеш отправит условный запрос. Если ответ изменился, кешированное значение обновляется. Кешированное значение затем возвращается из `fetch()`.
	• Отсутствие кеша отправит запрос, а ответ будет кеширован. Ответ возвращается из `fetch()`.
	`no-store`
	• Браузер отправляет запрос без проверки кеша.
	• Ответ не кешируется и возвращается из `fetch()`.

КЛЮЧ	ЗНАЧЕНИЕ

	`reload`
	• Браузер отправляет запрос без проверки кеша.
	• Ответ кешируется и возвращается из `fetch()`.
	`no-cache`
	• Как новое, так и устаревшее попадание в кеш отправит условный запрос. Если ответ изменился, кешированное значение обновляется. Кешированное значение затем возвращается из `fetch()`.
	• Отсутствие кеша отправит запрос, а ответ будет кеширован. Ответ возвращается из `fetch()`.
	`force-cache`
	• Из `fetch()` будет возвращено как новое, так и устаревшее попадание в кеш. Запрос не отправится.
	• Отсутствие кеша отправит запрос, а ответ будет кеширован. Ответ возвращается из `fetch()`.
	`only-if-cached`
	• Может использоваться только в том случае, если режим запроса имеет значение `same-origin`.
	• Из `fetch()` будет возвращено как новое, так и устаревшее попадание в кеш. Запрос не отправится.
	• Отсутствие кеша вернет ответ со статусом 504 (время ожидания).
	По умолчанию имеет значение `default`

`credentials`	Используется для указания того, должны ли файлы cookie включаться в исходящий запрос и если да, то каким образом. Похож на флаг `XMLHttpRequest withCredentials`. Должен иметь одно из следующих `string`-значений: i • `omit`: Файлы cookie не отправляются. • `same-origin`: Файлы cookie отправляются только тогда, когда источник URL запроса совпадает с источником сценария, выполняющего `fetch`. • `include`: Файлы cookie включаются в запросы как одного источника, так и перекрестных. Также может быть экземпляром `FederatedCredential` или `PasswordCredential` в браузерах, которые поддерживают Credential Management API. По умолчанию имеет значение `same-origin`
`headers`	Используется для указания заголовков запроса. Должен быть экземпляром объекта `Headers` или обычным объектом, содержащим пары ключ–значение заголовка типа `string`.

КЛЮЧ	ЗНАЧЕНИЕ
	По умолчанию используется объект `Headers` без пар ключ–значение. Это не означает, что запрос будет отправлен без заголовков; браузер может все еще добавить заголовки после формальной отправки запроса. Это несоответствие будет невидимым для JavaScript, но все еще может наблюдаться в инспекторе сети браузера
`integrity`	Используется для обеспечения целостности подресурса. Должен быть экземпляром типа `string`, содержащим идентификатор целостности подресурса. По умолчанию имеет значение «пустая строка»
`keepalive`	Используется, чтобы указать браузеру, чтобы запрос существовал после срока жизни страницы. Это полезно для сообщения о событиях или аналитических метриках для сервера, когда выгрузка страницы может произойти вскоре после отправки `fetch`. Выборка с флагом keepalive может использоваться вместо `Navigator.sendBeacon()`. Должен быть логическим значением. По умолчанию имеет значение `false`

`method`	Используется для указания метода HTTP запроса.
	Почти всегда будет одним из следующих значений типа `string`:
	`GET`
	`POST`
	`PUT`
	`PATCH`
	`DELETE`
	`HEAD`
	`OPTIONS`
	`CONNECT`
	`TRACE`
	По умолчанию имеет значение `GET`

`mode`	Используется для указания режима запроса. Режим определяет, является ли ответ на запрос кросс-доменного источника действительным и какая часть ответа может быть прочитана клиентом. Запросы, нарушающие указанный режим, приведут к ошибке. Должен быть экземпляром типа `string`: • `cors`: Разрешены перекрестные запросы, соответствующие протоколу CORS. Ответом будет «CORS-фильтрованный ответ», означающий, что заголовки, доступные в ответе, фильтруются с помощью белого списка, поддерживаемого браузером.

КЛЮЧ	ЗНАЧЕНИЕ
	• `no-cors`: Запросы из разных источников, которые не требуют предварительного запроса (HEAD, GET и POST только с заголовками CORS-фильтрованных запросов) разрешены. Тип ответа будет `opaque`, что означает, что содержимое ответа не может быть прочитано. • `same-origin`: Перекрестные запросы не допускаются. • `navigate`: Предназначен для поддержки HTML-навигации, создается только при навигации между документами. Скорее всего, вам никогда не понадобится использовать этот режим. Когда экземпляр `Request` создается вручную через конструктор, по умолчанию используется `cors`. В противном случае по умолчанию используется значение `no-cors`
`redirect`	Используется для указания способа обработки перенаправленных ответов (определенных как код состояния ответа 301, 302, 303, 307 или 308). Должен быть одним из перечисленных экземпляров типа `string`: • `follow`: Означает, что будет выполнено перенаправление и в качестве окончательного ответа будет возвращен возможный URL-адрес, имеющий не перенаправленный ответ. • `error`: Означает, что перенаправление вызовет ошибку. • `manual`: Означает, что запрос не будет следовать за перенаправлением, а вместо этого будет возвращать ответ с типом `opaqueredirect`, в то же время предоставляя URL предполагаемого перенаправления. Это позволяет выполнить перенаправление вручную. По умолчанию имеет значение `follow`
`referrer`	Используется для указания того, что следует отправлять в качестве заголовка HTTP `Referer`. Должен быть одним из перечисленных экземпляров типа `string`: • `no-referrer`: Отправить no-referrer в качестве значения HTTP `Referer`. • `client/about:client`: Отправить текущий URL-адрес или `no-referrer` (определяется политикой реферера) в качестве фактического значения HTTP `Referer`. • <URL>: Подменить URL-адрес, отправленный в качестве HTTP `Referer`. Происхождение подмененного URL должно соответствовать источнику исполняемого сценария. По умолчанию имеет значение `client/about:client`

КЛЮЧ	ЗНАЧЕНИЕ

`referrer-Policy`	Используется для указания заголовка HTTP `Referer`.
	Должен быть одним из перечисленных экземпляров типа `string`:
	`no-referrer`
	• Заголовок `Referer` полностью исключен из запроса.
	`no-referrer-when-downgrade`:
	• Для запросов, отправляемых из безопасного HTTPS-контекста на HTTP-URL, заголовок `Referer` опущен.
	• Для всех других запросов в заголовке `Referer` указывается полный URL-адрес.
	`origin`
	• Для всех запросов заголовок `Referer` установлен только на источник.
	`same-origin`
	• Для кросс-доменных запросов заголовок `Referer` опущен.
	• Для обычных запросов в заголовке `Referer` указывается полный URL-адрес.
	`strict-origin`
	• Для запросов, отправленных из безопасного контекста HTTPS к HTTP-URL, заголовок `Referer` опущен.
	• Для всех других запросов заголовок `Referer` установлен только на источник.
	`origin-when-cross-origin`
	• Для кросс-доменных запросов заголовок `Referer` устанавливается только на источник.
	• Для обычных запросов в заголовке `Referer` указывается полный URL-адрес.
	`strict-origin-when-cross-origin`
	• Для кросс-доменных запросов, отправляемых из безопасного HTTPS-контекста к HTTP URL, заголовок `Referer` опущен.
	• Для всех других кросс-доменных запросов заголовок `Referer` установлен только на источник.
	• Для обычных запросов в заголовке Referer указывается полный URL-адрес.
	`unsafe-url`
	• Для всех запросов в заголовке `Referer` задан полный URL-адрес.
	По умолчанию – `no-referer-when-downgrade`

КЛЮЧ	ЗНАЧЕНИЕ
signal	Используется для включения возможности отмены fetch на лету через связанный AbortController. Должен быть экземпляром AbortSignal. По умолчанию используется неассоциированный экземпляр AbortSignal

Общие паттерны Fetch

Как и в случае с XMLHttpRequest, fetch() используется как для извлечения данных, так и для их отправки. Используя объект init, можно настроить fetch() для отправки ассортимента сериализуемых типов данных в теле запроса.

Отправка данных JSON

Простая строка JSON может быть отправлена на сервер следующим образом:

```
let payload = JSON.stringify({
    foo: 'bar'
});

let jsonHeaders = new Headers({
    'Content-Type': 'application/json'
});

fetch('/send-me-json', {
    method: 'POST', // Должен использовать HTTP метод, посылающий тело
    body: payload,
    headers: jsonHeaders
});
```

Отправка параметров в теле запроса

Поскольку тело запроса поддерживает любые строковые значения, так же легко отправить параметры в виде сериализованной строки:

```
let payload = 'foo=bar&baz=qux';

let paramHeaders = new Headers({
    'Content-Type': 'application/x-www-form-urlencoded; charset=UTF-8'
});

fetch('/send-me-params', {
    method: 'POST', // Должен использовать HTTP метод, посылающий тело
    body: payload,
    headers: paramHeaders
});
```

Отправка файлов

Поскольку тело поддерживает экземпляры `FormData`, `fetch()` будет успешно сериализован и отправит файл, извлеченный из формы ввода средства выбора файлов:

```
let imageFormData = new FormData();
let imageInput = document.querySelector("input[type='file']");

imageFormData.append('image', imageInput.files[0]);

fetch('/img-upload', {
    method: 'POST',
    body: imageFormData
});
```

Такая реализация `fetch()` также может поддерживать несколько файлов:

```
let imageFormData = new FormData();
let imageInput = document.querySelector("input[type='file'][multiple]");

for (let i = 0; i < imageInput.files.length; ++i) {
    imageFormData.append('image', imageInput.files[i]);
}

fetch('/img-upload', {
    method: 'POST',
    body: imageFormData
});
```

Загрузка файлов как Blob-объектов

Fetch API может предоставить ответ в виде Blob-объекта, который, в свою очередь, совместим с несколькими API браузера. Одним из распространенных проявлений этого является явная загрузка файла изображения в память и присоединение его к элементу изображения HTML. Для этого объект ответа предоставляет метод `blob()`, который возвращает промис, разрешаемый в экземпляре Blob. Он, в свою очередь, может быть передан в `URL.createObjectUrl()`, чтобы сгенерировать допустимое значение для атрибута `src` элемента изображения:

```
const imageElement = document.querySelector('img');

fetch('my-image.png')
    .then((response) => response.blob())
    .then((blob) => {
        imageElement.src = URL.createObjectURL(blob);
    });
```

Отправка кросс-доменного запроса

Для кросс-доменного запроса ресурса ответу нужно иметь заголовки CORS, чтобы браузер мог его принять. Без заголовков кросс-доменный запрос завершится и выдаст ошибку.

```
fetch('//cross-origin.com');
// TypeError: Failed to fetch
// No 'Access-Control-Allow-Origin' header is present on the requested resource.
```

Если коду не нужен доступ к ответу, можно отправить `no-cors` запрос. В этом случае свойство `type` ответа будет иметь значение `opaque`, поэтому его нельзя будет проверить. Эта стратегия может быть полезна для отправки пингов или в случаях, когда ответ может быть просто кеширован для последующего использования.

```
fetch('//cross-origin.com', { method: 'no-cors' })
    .then((response) => console.log(response.type));

// opaque
```

Прерывание запроса

Fetch API поддерживает прерывание запроса через пару `AbortController/AbortSignal`. Вызов `AbortController.abort()` прекращает всю передачу по сети, поэтому это особенно полезно, если нужно остановить передачу большой полезной нагрузки. Прерывание `fetch()` приведет к его отклонению с ошибкой.

```
let abortController = new AbortController();

fetch('wikipedia.zip', { signal: abortController.signal })
    .catch(() => console.log('aborted!');

// Прерывание fetch спустя 10мс
setTimeout(() => abortController.abort(), 10);

// aborted!
```

Объекты Headers

Объект `Headers` используется в качестве контейнера для всех заголовков исходящих запросов и входящих ответов. Каждый исходящий экземпляр запроса включает в себя пустой экземпляр заголовка, доступный через `Request.prototype.headers` и каждый входящий экземпляр `Response` включает заполненный экземпляр `Headers`, доступный через `Response.prototype.headers` — оба они являются изменяемыми свойствами. Можно создать свежий экземпляр через конструктор с помощью `new Headers()`.

Изучение схожести Headers и Map

Объект `Headers` имеет высокую степень схожести с объектом `Map`. Это должно иметь смысл, так как заголовки HTTP по существу являются сериализованными парами ключ–значение, а их JavaScript-представление является промежуточным интерфейсом. Типы `Headers` и `Map` совместно используют несколько методов экземпляра: `get()`, `set()`, `has()` и `delete()`, как показано здесь:

```
let h = new Headers();
let m = new Map();

// Задание ключа
h.set('foo', 'bar');
m.set('foo', 'bar');

// Проверка ключа
console.log(h.has('foo'));    // true
console.log(m.has('foo'));    // true
console.log(h.has('qux'));    // false
console.log(m.has('qux'));    // false

// Получение значения
console.log(h.get('foo'));    // bar
console.log(m.get('foo'));    // bar

// Замена значения
h.set('foo', 'baz');
m.set('foo', 'baz');

// Получение замененного значения
console.log(h.get('foo'));    // baz
console.log(m.get('foo'));    // baz

// Удаление значения
h.delete('foo');
m.delete('foo');

// Проверка удаления значения
console.log(h.get('foo'));    // undefined
console.log(m.get('foo'));    // undefined
```

Эти типы могут быть оба инициализированы с помощью итерируемого объекта, как показано здесь:

```
let seed = [['foo', 'bar']];

let h = new Headers(seed);
let m = new Map(seed);

console.log(h.get('foo')); // bar
console.log(m.get('foo')); // bar
```

Они также имеют идентичные интерфейсы итераторов `keys()`, `values()` и `records()`:

```
let seed = [['foo', 'bar'], ['baz', 'qux']];

let h = new Headers(seed);
let m = new Map(seed);

console.log(...h.keys()); // foo, baz
console.log(...m.keys()); // foo, baz

console.log(...h.values()); // bar, qux
console.log(...m.values()); // bar, qux

console.log(...h.entries()); // ['foo', 'bar'], ['baz', 'qux']
console.log(...m.entries()); // ['foo', 'bar'], ['baz', 'qux']
```

Уникальные возможности объекта Headers

Объект `Headers` не является точным факсимиле `Map`. При инициализации объект `Headers` можно инициализировать с помощью объекта пар ключ–значение, тогда как `Map` нельзя:

```
let seed = {foo: 'bar'};

let h = new Headers(seed);
console.log(h.get('foo')); // bar

let m = new Map(seed);
// TypeError: object is not iterable
```

Одному заголовку HTTP может быть назначено несколько значений, и объект `Headers` поддерживает это с помощью метода `append()`. При использовании с заголовком, который еще не существует в экземпляре Headers, `append()` ведет себя идентично `set()`. Последующее использование объединит значение заголовка, разделенное запятой:

```
let h = new Headers();

h.append('foo', 'bar');
console.log(h.get('foo'));    // "bar"

h.append('foo', 'baz');
console.log(h.get('foo'));    // "bar, baz"
```

Охранники заголовков

В некоторых случаях не все заголовки HTTP могут изменяться клиентом, и объект `Headers` использует средства защиты для обеспечения этого. Различные настройки защиты изменят поведение методов `set()`, `append()` и `delete()`. Нарушение ограничений охраны вызовет ошибку TypeError.

Экземпляр `Headers` будет вести себя по-разному в зависимости от его происхождения; это поведение регулируется охранником. Невозможно определить настройку защиты экземпляра `Headers` в JavaScript. В следующей таблице описаны различные возможные параметры защиты и поведенческие последствия каждого из них.

ОХРАННИК	ПРИМЕНИМЫЙ СЦЕНАРИЙ	ОГРАНИЧЕНИЯ
`none`	Активен, когда экземпляр `Headers` создается через конструктор	Нет
`request`	Активен, когда объект `Request` создается через конструктор в любом режиме, кроме `no-cors`	Не допускаются изменения заголовков с запрещенными именами (https://developer.mozilla.org/en-US/docs/Glossary/Forbidden_header_name)

ОХРАННИК	ПРИМЕНИМЫЙ СЦЕНАРИЙ	ОГРАНИЧЕНИЯ
`request-no-cors`	Активен, когда объект `Request` создается через конструктор в режиме `no-cors`	Не допускаются изменения заголовков, которые не являются простыми (https://developer.mozilla.org/en-US/docs/Glossary/simple_header)
`response`	Активен, когда объект `Response` создается через конструктор	Не допускаются изменения заголовков с запрещенными именами заголовков ответов (https://developer.mozilla.org/en-US/docs/Glossary/Forbidden_response_header_name)
`immutable`	Активен, когда объект `Response` создается с помощью статических методов `error()` или `redirect()`	Никакие модификации заголовка не разрешены

Объект Request

Как указывает его имя, объект `Request` (Запрос) является интерфейсом для запроса на извлеченный ресурс. Этот интерфейс предоставляет информацию о характере запроса, а также о различных способах использования тела запроса.

> **ПРИМЕЧАНИЕ** Свойства и методы, касающиеся тела, описаны в разделе «Запросы, ответы и описание тела» в этой главе.

Создание объектов Request

Объект `Request` может быть создан с помощью конструктора. Требуется входной аргумент, который чаще всего будет URL-строкой:

```
let r = new Request('https://foo.com');
console.log(r);
// Request {...}
```

Конструктор `Request` также принимает второй необязательный аргумент, объект `init`. Этот объект `init` идентичен объекту `fetch()`, как описано ранее в разделе «Пользовательские параметры `fetch`». Значениям, не указанным в `init`, будут присвоены значения по умолчанию в экземпляре `Request`:

```
// Создание объекта Request со всеми значениями по умолчанию:
console.log(new Request(''));

// Request {
//    bodyUsed: false
//    cache: "default"
```

```
//    credentials: "same-origin"
//    destination: ""
//    headers: Headers {}
//    integrity: ""
//    keepalive: false
//    method: "GET"
//    mode: "cors"
//    redirect: "follow"
//    referrer: "about:client"
//    referrerPolicy: ""
//    signal: AbortSignal {aborted: false, onabort: null}
//    url: "<current URL>"
// }

// Создание объекта Request с указанными значениями init:
console.log(new Request('https://foo.com',
    { method: 'POST' }));

// Request {
//    bodyUsed: false
//    cache: "default"
//    credentials: "same-origin"
//    destination: ""
//    headers: Headers {}
//    integrity: ""
//    keepalive: false
//    method: "POST"
//    mode: "cors"
//    redirect: "follow"
//    referrer: "about:client"
//    referrerPolicy: ""
//    signal: AbortSignal {aborted: false, onabort: null}
//    url: "https://foo.com/"
// }
```

Клонирование объекта Request

Fetch API предлагает два немного отличающихся способа создания копий объекта `Request`: с помощью конструктора `Request` и метода `clone()`.

Передача экземпляра `Request` в качестве входного аргумента в конструктор `Request` создаст копию этого запроса:

```
let r1 = new Request('https://foo.com');
let r2 = new Request(r1);

console.log(r2.url);    // https://foo.com/
```

Значения внутри объекта `init` переопределяют значения исходного объекта:

```
let r1 = new Request('https://foo.com');
let r2 = new Request(r1, {method: 'POST'});

console.log(r1.method);    // GET
console.log(r2.method);    // POST
```

Эта стратегия копирования не всегда получает в результате точную копию. В частности, она будет помечать первое тело запроса как использованное:

```
let r1 = new Request('https://foo.com',
    { method: 'POST', body: 'foobar' });
let r2 = new Request(r1);

console.log(r1.bodyUsed); // true
console.log(r2.bodyUsed); // false
```

Если исходный объект имеет происхождение, отличное от того, где создается новый объект, свойство `referrer` очищается. Кроме того, если исходный объект имеет значение режима `navigate`, оно будет преобразовано в `same-origin`.

Второй способ клонирования объекта `Request` заключается в использовании метода `clone()`, который создает точную копию без возможности переопределения каких-либо значений. В отличие от первого метода он не помечает тело запроса как использованное:

```
let r1 = new Request('https://foo.com', { method: 'POST', body: 'foobar' });
let r2 = r1.clone();

console.log(r1.url); // https://foo.com/
console.log(r2.url); // https://foo.com/

console.log(r1.bodyUsed); // false
console.log(r2.bodyUsed); // false
```

Клонирование `Request` с использованием любого из методов допускается, если его свойство `bodyUsed` имеет значение `false`, то есть тело еще не прочитано. Как только тело будет прочитано, попытка клонирования вызовет ошибку `TypeError`.

```
let r = new Request('https://foo.com');
r.clone();
new Request(r);
// Без ошибок

r.text();    // установка поля bodyUsed в значение false

r.clone();
// TypeError: Failed to execute 'clone' on 'Request': Request body is already used

new Request(r);
// TypeError: Failed to construct 'Request': Cannot construct a Request with a
// Request object that has already been used.
```

Использование объекта Request с fetch()

Факт, что `fetch()` и конструктор `Request` имеют идентичные сигнатуры функций, не случаен. При вызове `fetch()` можно передать уже созданный экземпляр `Request` вместо URL. Как и в конструкторе `Request`, значения, предоставленные в объекте инициализации `fetch()`, будут переопределять предоставленные значения запроса:

```
let r = new Request('https://foo.com');

// отправка GET запроса к foo.com
fetch(r);

// отправка POST запроса к foo.com
fetch(r, { method: 'POST' });
```

Технически `fetch` клонирует предоставленный объект `Request`. Как и при клонировании `Request`, `fetch` не может быть отправлен с `Request`, тело которого использовано:

```
let r = new Request('https://foo.com',
    { method: 'POST', body: 'foobar' });

r.text();

fetch(r);
// TypeError: Cannot construct a Request with a Request object that has already
// been used.
```

Важно отметить, что использование `Request` в `fetch` также помечает тело как использованное. Следовательно, с `Request`, имеющим тело, можно выполнить только один `fetch()`. (Запросы, которые не включают тело, не подпадают под это ограничение.) Это демонстрируется здесь:

```
let r = new Request('https://foo.com',
    { method: 'POST', body: 'foobar' });

fetch(r);

fetch(r);
// TypeError: Cannot construct a Request with a Request object that has already
// been used.
```

Для вызова `fetch()` несколько раз с одним и тем же объектом `Request`, который содержит тело, `clone()` должен быть вызван перед отправкой первого `fetch()`:

```
let r = new Request('https://foo.com',
    { method: 'POST', body: 'foobar' });

// Все три вызова пройдут успешно
fetch(r.clone());
fetch(r.clone());
fetch(r);
```

Объект Response

Как указывает его имя, объект `Response` (Ответ) является интерфейсом к ответу из извлеченного ресурса. Этот интерфейс предоставляет информацию о природе ответа, а также о различных способах использования тела ответа.

> **ПРИМЕЧАНИЕ** Свойства и методы, касающиеся тела, описаны в разделе «Запросы, ответы и описание тела» в этой главе.

Создание объекта Response

Объект `Response` может быть создан с помощью конструктора — этот способ не требует никаких аргументов. Его свойства будут заполнены значениями по умолчанию, так как этот экземпляр не представляет фактический ответ HTTP:

```
let r = new Response();
console.log(r);
// Response {
//    body: (...)
//    bodyUsed: false
//    headers: Headers {}
//    ok: true
//    redirected: false
//    status: 200
//    statusText: "OK"
//    type: "default"
//    url: ""
// }
```

Конструктор `Response` принимает первый необязательный аргумент, `body`. Этот `body`, который может иметь значение `null`, идентичен `init body`, как описано ранее в разделе «Пользовательские параметры `fetch()`». Второй необязательный аргумент, объект `init`, должен заполняться любым количеством ключей и соответствующими значениями в следующей таблице.

КЛЮЧ	ЗНАЧЕНИЕ
`headers`	Должен быть экземпляром объекта `Headers` или обычным объектом, содержащим пары ключ–значение заголовка типа `string`. По умолчанию используется объект `Headers` без пар ключ–значение
`status`	Целое число, указывающее код состояния ответа HTTP. По умолчанию имеет значение 200
`statusText`	Строка, описывающая статус ответа HTTP. По умолчанию используется пустая строка

`body` и `init` могут быть использованы для создания `Response` следующим образом:

```
let r = new Response('foobar', {
    status: 418,
    statusText: 'I\'m a teapot'
});
console.log(r);

// Response {
//    body: (...)
//    bodyUsed: false
//    headers: Headers {}
//    ok: false
//    redirected: false
//    status: 418
```

```
//    statusText: "I'm a teapot"
//    type: "default"
//    url: ""
// }
```

Для большинства приложений наиболее распространенным способом создания объекта `Response` является вызов `fetch()`; он возвращает промис, который разрешается в объекте `Response`, который представляет фактический HTTP-ответ. В следующем коде показан пример объекта `Response`, который можно ожидать:

```
fetch('https://foo.com')
    .then((response) => {
        console.log(response);
    });
```

```
// Response {
//    body: (...)
//    bodyUsed: false
//    headers: Headers {}
//    ok: true
//    redirected: false
//    status: 200
//    statusText: "OK"
//    type: "basic"
//    url: "https://foo.com/"
// }
```

Класс `Response` также имеет два статических метода для генерации объектов `Response`, `Response.redirect()` и `Response.error()`. `Response.redirect()` принимает URL-адрес и код состояния перенаправления (301, 302, 303, 307 или 308) и возвращает перенаправленный объект `Response`:

```
console.log(Response.redirect('https://foo.com', 301));
// Response {
//    body: (...)
//    bodyUsed: false
//    headers: Headers {}
//    ok: false
//    redirected: false
//    status: 301
//    statusText: ""
//    type: "default"
//    url: ""
// }
```

Предоставленный код состояния должен квалифицироваться как перенаправление; в противном случае выдается ошибка:

```
Response.redirect('https://foo.com', 200);
// RangeError: Failed to execute 'redirect' on 'Response': Invalid status code
```

Также доступен для использования `Response.error()`. Этот статический метод создает ответ, который вы ожидаете получить при ошибке сети, что приведет к отклонению промиса `fetch()`.

```
console.log(Response.error());
// Response {
//    body: (...)
//    bodyUsed: false
//    headers: Headers {}
//    ok: false
//    redirected: false
//    status: 0
//    statusText: ""
//    type: "error"
//    url: ""
// }
```

Чтение информации из статуса Response

Объект `Response` предлагает набор свойств только для чтения, описывающих, как был завершен запрос, что показано в следующей таблице.

СВОЙСТВО	ЗНАЧЕНИЕ
`headers`	Объект `Headers`, связанный с ответом
`ok`	Логическое значение, указывающее природу кода состояния HTTP. Код состояния 200–299 возвращает значение `true`, остальные коды состояния – `false`
`redirected`	Логическое значение, указывающее, подвергался ли ответ хотя бы одному перенаправлению
`status`	Целое число, указывающее код состояния HTTP ответа
`statusText`	Строка, содержащая каноническое описание, связанное с кодом статуса HTTP. Это значение получено из необязательного поля HTTP Reason-Phrase, поэтому это поле может быть пустой строкой, если сервер отказывается ответить с Reason-Phrase
`type`	Строка, содержащая тип ответа. Она будет содержать одно из следующих строковых значений: • `basic`: Обозначает стандартный ответ того же источника. • `cors`: Указывает на стандартный ответ о происхождении. • `error`: Указывает, что объект ответа был создан с помощью `Response.error()`. • `opaque`: Указывает на перекрестный ответ на `no-cors fetch()`. • `opaqueredirect`: Указывает на ответ на запрос со свойством `redirect`, установленным в значение `manual`.
`url`	Строка, содержащая URL-адрес ответа. Для перенаправленных ответов это будет последний URL-адрес, по которому был получен ответ без перенаправления

Следующий пример демонстрирует типичное содержание ответа для URL, которые возвращают 200, 302, 404 и 500:

```
fetch('//foo.com').then(console.log);
// Response {
//    body: (...)
//    bodyUsed: false
//    headers: Headers {}
//    ok: true
//    redirected: false
//    status: 200
//    statusText: "OK"
//    type: "basic"
//    url: "https://foo.com/"
// }

fetch('//foo.com/redirect-me').then(console.log);
// Response {
//    body: (...)
//    bodyUsed: false
//    headers: Headers {}
//    ok: true
//    redirected: true
//    status: 200
//    statusText: "OK"
//    type: "basic"
//    url: "https://foo.com/redirected-url/"
// }

fetch('//foo.com/does-not-exist').then(console.log);
// Response {
//    body: (...)
//    bodyUsed: false
//    headers: Headers {}
//    ok: false
//    redirected: true
//    status: 404
//    statusText: "Not Found"
//    type: "basic"
//    url: "https://foo.com/does-not-exist/"
// }

fetch('//foo.com/throws-error').then(console.log);
// Response {
//    body: (...)
//    bodyUsed: false
//    headers: Headers {}
//    ok: false
//    redirected: true
//    status: 500
//    statusText: "Internal Server Error"
//    type: "basic"
//    url: "https://foo.com/throws-error/"
// }
```

Клонирование объекта Response

Основным способом клонирования объекта `Response` является использование метода `clone()`, который создает точную копию без возможности переопределения каких-либо значений. Он не помечает тело запроса как использованное:

```
let r1 = new Response('foobar');
let r2 = r1.clone();

console.log(r1.bodyUsed);    // false
console.log(r2.bodyUsed);    // false
```

Клонирование ответа допускается, если свойство запроса `bodyUsed` имеет значение `false`, то есть тело еще не прочитано. Как только тело будет прочитано, попытка клонирования вызовет ошибку `TypeError`.

```
let r = new Response('foobar');
r.clone();

// Без ошибок
r.text();    // установка поля bodyUsed в значение false

r.clone();
// TypeError: Failed to execute 'clone' on 'Response': Response body is
// already used
```

Чтение может быть выполнено только с `Response`, у которого есть тело. (Ответы, которые не включают тело, не подпадают под это ограничение.) Это продемонстрировано здесь:

```
let r = new Response('foobar');

r.text().then(console.log);    // foobar

r.text().then(console.log);
// TypeError: Failed to execute 'text' on 'Response': body stream is locked
```

Чтобы прочитать тело несколько раз с одним и тем же объектом `Response`, который включает тело, `clone()` должен быть вызван перед выполнением первого чтения:

```
let r = new Response('foobar');

r.clone().text().then(console.log);    // foobar
r.clone().text().then(console.log);    // foobar
r.text().then(console.log);            // foobar
```

В качестве альтернативы можно выполнить операцию псевдоклонирования, создав новый экземпляр `Response` с исходным телом. Важно отметить, что эта стратегия не помечает первый ответ как прочитанный, но тело *разделяется* между двумя ответами:

```
let r1 = new Response('foobar');
let r2 = new Response(r1.body);

console.log(r1.bodyUsed);    // false
```

```
console.log(r2.bodyUsed);    // false

r2.text().then(console.log); // foobar
r1.text().then(console.log);
// TypeError: Failed to execute 'text' on 'Response': body stream is locked
```

Запросы, ответы и описание тела

И `Request`, и `Response` включают в себя описания `Body` Fetch API, чтобы учесть характер переноса полезной нагрузки обоих типов. Это описание присваивает каждому типу свойство `body` только для чтения (реализовано как `ReadableStream`), логическое свойство `bodyUsed` только для чтения, указывающее, был ли прочитан поток тела, и несколько методов, которые читают поток до завершения и преобразуют результат в определенный тип объекта JavaScript.

Как правило, есть две основные причины использовать тело запроса или ответа в виде потока. Либо сетевая задержка является фактором, обусловленным размером полезной нагрузки, либо сам API потока по своей природе пригодится для обработки полезной нагрузки. Почти во всех других случаях тело извлеченного ресурса будет наиболее полезным, если оно будет использовано одновременно.

Описание `Body` предоставляет пять различных методов, которые сбрасывают `ReadableStream` в один буфер в памяти, приводят буфер к определенному типу объекта JavaScript и создают его внутри промиса. Этот промис будет ждать до тех пор, пока основной поток не сообщит о завершении, а буфер не будет проанализирован перед разрешением. Это означает, что нужно будет дождаться полной загрузки выбранного ресурса на клиенте перед получением доступа к его содержимому.

Body.text()

Метод `Body.text()` возвращает промис, который разрешается с очищенным буфером, декодированным как строка UTF-8. Использование `Body.text()` показано здесь с объектом `Response`:

```
fetch('https://foo.com')
    .then((response) => response.text())
    .then(console.log);

// <!doctype html><html lang="en">
// <head>
// <meta charset="utf-8">
// ...
```

Использование `Body.text()` с объектом `Request`:

```
let request = new Request('https://foo.com',
    { method: 'POST', body: 'barbazqux' });

request.text()
    .then(console.log);

// barbazqux
```

Body.json()

Метод `Body.json()` возвращает промис, который разрешается с очищенным буфером, декодированным как JSON. Использование `Body.json()` показано здесь с объектом `Response`:

```
fetch('https://foo.com/foo.json')
    .then((response) => response.json())
    .then(console.log);

// {"foo": "bar"}
```

Использование `Body.json()` с объектом `Request`:

```
let request = new Request('https://foo.com',
    { method:'POST', body: JSON.stringify({ bar: 'baz' }) });

request.json()
    .then(console.log);

// {bar: 'baz'}
```

Body.formData()

Браузеры могут сериализовать/десериализовать объекты `FormData` как тело. Например, рассмотрим следующий экземпляр `FormData`:

```
let myFormData = new FormData();
myFormData.append('foo', 'bar');
```

При передаче через HTTP браузер WebKit может сериализовать это следующим образом:

```
------WebKitFormBoundarydR9Q2kOzE6nbN7eR
Content-Disposition: form-data; name="foo"
bar
------WebKitFormBoundarydR9Q2kOzE6nbN7eR-
```

Метод `Body.formData()` возвращает промис, разрешаемый с очищенным буфером, декодированным как экземпляр `FormData`. Использование `Body.formData()` показано здесь с объектом `Response`:

```
fetch('https://foo.com/form-data')
    .then((response) => response.formData())
    .then((formData) => console.log(formData.get('foo'));

// bar
```

Использование `Body.formData()` с объектом `Request`:

```
let myFormData = new FormData();
myFormData.append('foo', 'bar');

let request = new Request('https://foo.com',
    { method:'POST', body: myFormData });
```

```
request.formData()
    .then((formData) => console.log(formData.get('foo'));

// bar
```

Body.arrayBuffer()

Вам может понадобиться проверять и изменять полезную нагрузку тела как двоичный файл. Для такой задачи тело можно преобразовать в экземпляр `ArrayBuffer` с помощью `Body.arrayBuffer()`. Этот метод возвращает промис, который разрешается с очищенным буфером, выставленным как `ArrayBuffer`. Использование `Body.arrayBuffer()` показано здесь с объектом `Response`:

```
fetch('https://foo.com')
    .then((response) => response.arrayBuffer())
    .then(console.log);

// ArrayBuffer(...) {}
```

Использование `Body.arrayBuffer()` с объектом `Request`:

```
let request = new Request('https://foo.com',
    { method:'POST', body: 'abcdefg' });

// Запись закодированных строковых двоичных значений как целых чисел
request.arrayBuffer()
    .then((buf) => console.log(new Int8Array(buf)));

// Int8Array(7) [97, 98, 99, 100, 101, 102, 103]
```

Body.blob()

Вам может понадобиться использовать полезную нагрузку тела как двоичный файл без проверок и модификаций. Для такой задачи тело можно использовать как экземпляр `Blob` с помощью `Body.blob()`. Этот метод возвращает промис, который разрешается с очищенным буфером, выставленным как `Blob`. Использование `Body.blob()` показано здесь с объектом `Response`:

```
fetch('https://foo.com')
    .then((response) => response.blob())
    .then(console.log);

// Blob(...) {size:..., type: "..."}
```

Использование `Body.arrayBuffer()` с объектом `Request`:

```
let request = new Request('https://foo.com',
    { method:'POST', body: 'abcdefg' });

request.blob()
    .then(console.log);

// Blob(7) {size: 7, type: "text/plain;charset=utf-8"}
```

Одноразовые потоки

Поскольку описание `Body` построено поверх `ReadableStream`, это означает, что поток тела может быть прочитан только один раз. Смысл этого в том, что все методы описания `Body` можно вызывать только один раз; последующие попытки вызвать метод описания приведут к ошибке.

```
fetch('https://foo.com')
    .then((response) => response.blob().then(() => response.blob()));

// TypeError: Failed to execute 'blob' on 'Response': body stream is locked
let request = new Request('https://foo.com',
    { method: 'POST', body: 'foobar' });

request.blob().then(() => request.blob());
// TypeError: Failed to execute 'blob' on 'Request': body stream is locked
```

Даже если поток находится в процессе чтения, все эти методы будут блокировать `ReadableStream` сразу после вызова и не позволят второму считывателю получить доступ к потоку:

```
fetch('https://foo.com')
    .then((response) => {
        response.blob(); // First call locks the stream
        response.blob(); // Second call attempts to lock the stream and fails
 });

// TypeError: Failed to execute 'blob' on 'Response': body stream is locked
let request = new Request('https://foo.com',
    { method: 'POST', body: 'foobar' });

request.blob(); // First call locks the stream
request.blob(); // Second call attempts to lock the stream and fails
// TypeError: Failed to execute 'blob' on 'Request': body stream is locked
```

Как часть описания `Body`, логическое свойство `bodyUsed` указывает, *прерван* ли `ReadableStream`, что означает, что читатель уже установил блокировку в потоке. Это не обязательно означает, что поток полностью опустошен. Данное свойство демонстрируется здесь:

```
let request = new Request('https://foo.com',
    { method: 'POST', body: 'foobar' });
let response = new Response('foobar');

console.log(request.bodyUsed);        // false
console.log(response.bodyUsed);       // false

request.text().then(console.log);     // foobar
response.text().then(console.log);    // foobar

console.log(request.bodyUsed);        // true
console.log(response.bodyUsed);       // true
```

Использование тела ReadableStream

Большая часть программирования на JavaScript рассматривает сети как атомарные операции; запросы создаются и отправляются одновременно, а ответы представляются как единая полезная нагрузка данных, которая становится доступной сразу. Это соглашение скрывает основную путаницу, делая код, связанный с сетью, приятным для написания.

По самой природе TCP/IP передаваемые данные поступают в конечную точку порциями и только так быстро, как сеть может доставить эти порции. Принимающая конечная точка выделяет память и записывает то, что получено по сети, по мере поступления. Fetch API позволяет читать и манипулировать этими данными, когда они поступают в режиме реального времени через `ReadableStream`.

> **ПРИМЕЧАНИЕ** Примеры в этом разделе извлекают HTML-код спецификации Fetch, которую можно найти по адресу https://fetch.spec.whatwg.org/. Эта страница имеет примерно 1 МБ разметки, что является достаточно большой полезной нагрузкой, поэтому примеры потоков в этом разделе будут разбиты на несколько частей.

`ReadableStream`, как определено в Stream API, предоставляет метод `getReader()`, который создает `ReadableStreamDefaultReader` — его можно использовать для асинхронного получения фрагментов тела по мере их поступления. Каждый фрагмент потока тела предоставляется как `Uint8Array`.

Следующий пример кода вызывает `read()` для читателя для регистрации первого доступного фрагмента:

```
fetch('https://fetch.spec.whatwg.org/')
    .then((response) => response.body)
    .then((body) => {
        let reader = body.getReader();

        console.log(reader);    // ReadableStreamDefaultReader {}

        reader.read()
            .then(console.log);
        });

// { value: Uint8Array{}, done: false }
```

Чтобы получить всю полезную нагрузку, когда она станет доступной, метод `read()` может быть вызван рекурсивно:

```
fetch('https://fetch.spec.whatwg.org/')
    .then((response) => response.body)
    .then((body) => {
        let reader = body.getReader();

        function processNextChunk({value, done}) {
```

```
            if (done) {
                return;
            }

            console.log(value);

            return reader.read()
                .then(processNextChunk);
            }

            return reader.read()
                .then(processNextChunk);
            });

// { value: Uint8Array{}, done: false }
// { value: Uint8Array{}, done: false }
// { value: Uint8Array{}, done: false }
// ...
```

Асинхронные функции очень подходят для использования в операциях `fetch()`. Эта рекурсивная реализация может быть сокращена с помощью `async/await`:

```
fetch('https://fetch.spec.whatwg.org/')
    .then((response) => response.body)
    .then(async function(body) {
        let reader = body.getReader();

        while(true) {
            let { value, done } = await reader.read();
            if (done) {
                break;
            }

            console.log(    value);
        }
    });

// { value: Uint8Array{}, done: false }
// { value: Uint8Array{}, done: false }
// { value: Uint8Array{}, done: false }
// ...
```

С другой стороны, метод `read()` достаточно близок к интерфейсу `Iterable`, поэтому его просто преобразовать в цикл `for-await-of`:

```
fetch('https://fetch.spec.whatwg.org/')
    .then((response) => response.body)
    .then(async function(body) {
        let reader = body.getReader();

        let asyncIterable = {
            [Symbol.asyncIterator]() {
                return {
                    next() {
                        return reader.read();
```

```
                    }
                };
            }
        };

        for await (chunk of asyncIterable) {
            console.log(chunk);
        }
    });

// { value: Uint8Array{}, done: false }
// { value: Uint8Array{}, done: false }
// { value: Uint8Array{}, done: false }
// ...
```

Этот код может быть уменьшен до более чистой функции генератора. Кроме того, эту реализацию можно сделать более надежной за счет частичного чтения потока. Если поток завершается либо из-за истощения, либо из-за ошибки, считыватель должен снять блокировку, чтобы позволить другому считывателю потока определить, где он остановился:

```
async function* streamGenerator(stream) {
    const reader = stream.getReader();

    try {
        while (true) {
            const { value, done } = await reader.read();
            if (done) {
                break;
            }

            yield value;
        }
    } finally {
        reader.releaseLock();
    }
}

fetch('https://fetch.spec.whatwg.org/')
    .then((response) => response.body)
    .then(async function(body) {
        for await (chunk of streamGenerator(body)) {
            console.log(chunk);
        }
    });
```

В этих примерах, после того как текущий фрагмент `Uint8Array` выходит из области видимости, браузер помечает его как подходящий для сборки мусора. Это обеспечивает потенциально значительную экономию памяти в сценариях, где целесообразно исследовать большую полезную нагрузку последовательно и в отдельных сегментах.

Размер буфера и то, ожидает ли браузер его заполнения, прежде чем помещать его в поток, зависит от реализации среды выполнения JavaScript. Браузер чувствителен

к тому факту, что он идеально подходит для ожидания и заполнения выделенного буфера, когда это возможно, но в то же время он старается поддерживать поток заполненным, посылая (иногда незаполненные) буферы как можно чаще.

Браузеры могут изменять размер буфера фрагмента в зависимости от таких факторов, как пропускная способность или задержка сети. Кроме того, браузер может решить отправить частично заполненный буфер в поток, если он решит не ждать сеть. В конечном счете код должен быть подготовлен для обработки следующего:

- Фрагменты `Uint8Array` переменного размера;
- Фрагменты `Uint8Array` заполнены частично;
- Фрагменты поступают с непредсказуемыми интервалами.

По умолчанию фрагменты поступают в формате `Uint8Array`. Поскольку завершение фрагмента не учитывает закодированный контент, могут быть значения, такие как многобайтовые символы, разделенные между двумя отдельными последовательными фрагментами. Существуют грязные способы учета этого вручную, но для многих случаев существуют решения в стиле «включай и работай» из Encoding API.

Чтобы преобразовать `Uint8Array` в читаемый текст, можно передать `TextDecoder` в буфер и вернуть преобразованное значение. Установка конфигурации `stream:true` позволяет ему сохранить предыдущий буфер в памяти, чтобы содержимое, соединенное между двумя блоками, могло быть правильно декодировано:

```
let decoder = new TextDecoder();

async function* streamGenerator(stream) {
    const reader = stream.getReader();

    try {
        while (true) {
            const { value, done } = await reader.read();

            if (done) {
                break;
            }

            yield value;
        }
    } finally {
        reader.releaseLock();
    }
}

fetch('https://fetch.spec.whatwg.org/')
    .then((response) => response.body)
    .then(async function(body) {
        for await (chunk of streamGenerator(body)) {
            console.log(decoder.decode(chunk, { stream: true }));
```

```
        }
    });

// <!doctype html><html lang="en"> ...
// любой <a data-link-type="dfn" href="#concept-header" ...
// превращается в результате в <var>rangeValue</var>. ...
// ...
```

Поскольку объект `Response` может быть создан с использованием `ReadableStream`, можно прочитать поток, передать его во вновь созданный вторичный поток и использовать этот вторичный поток для методов `Body`, таких как `text()`. Это позволяет проверять и управлять содержимым потока, когда оно становится доступным. Техника использования двойного потока показана здесь:

```
fetch('https://fetch.spec.whatwg.org/')
    .then((response) => response.body)
    .then((body) => {
        const reader = body.getReader();

        // создание вторичного потока
        return new ReadableStream({
            async start(controller) {
                try {
                    while (true) {
                        const { value, done } = await reader.read();

                        if (done) {
                            break;
                        }

                        // Передача фрагмента тела потока во
                        // вторичный поток
                        controller.enqueue(value);
                    }
                } finally {
                    controller.close();
                    reader.releaseLock();
                }
            }
        })
    })
    .then((secondaryStream) => new Response(secondaryStream))
    .then(response => response.text())
    .then(console.log);

// <!doctype html><html lang="en"><head><meta charset="utf-8"> ...
```

BEACON API

Чтобы максимизировать объем информации, передаваемой о странице, многим аналитическим инструментам необходимо отправлять данные телеметрии или аналитики на сервер как можно позже в жизненном цикле страницы. В результате

оптимальным вариантом является отправка сетевого запроса на событие `unload` браузера. Это событие сигнализирует о том, что происходит отправка страницы и что на этой странице больше не будет генерироваться никакой полезной информации.

Когда происходит событие `unload`, аналитические инструменты хотят прекратить сбор информации и попытаться отправить то, что у них есть, на сервер. Это создает проблему, так как событие `unload` означает для браузера, что нет особых оснований для отправки любых ожидающих сетевых запросов (поскольку страница все равно удаляется). Например, любые асинхронные запросы, созданные в обработчике выгрузки, будут отменены браузером. Поэтому асинхронный `XMLHttpRequest` или `fetch()` не подходит для этой задачи. Аналитические инструменты могут использовать синхронный `XMLHttpRequest` для принудительной доставки запроса, но это вызывает проблемы с пользовательским интерфейсом. Поскольку браузер приостанавливает ожидание возврата запроса, переход к следующей странице задерживается, пока браузер ожидает завершения работы обработчика выгрузки внутри обработчика `unload`.

Для решения этой проблемы W3C представил дополнительный API — Beacon. API добавляет единственный метод `sendBeacon()` к объекту `navigator`. Этот простой метод принимает URL-адрес и полезную нагрузку данных и отправляет запрос POST. Дополнительной полезной нагрузкой данных может быть экземпляр `ArrayBufferView`, `Blob`, `DOMString` или `FormData`. Метод возвращает `true`, если запрос был успешно поставлен в очередь для возможной передачи, иначе `false`.

Его можно использовать следующим образом:

```
// Sends POST request
// URL: 'https://example.com/analytics-reporting-url'
// Request Payload: '{foo: "bar"}'

navigator.sendBeacon('https://example.com/analytics-reporting-url',
                     '{foo: "bar"}');
```

Этот метод может показаться просто синтаксическим сахаром для запроса POST, но у него есть несколько примечательных особенностей:

- `sendBeacon()` не ограничивается концом жизненного цикла страницы, его можно использовать в любое время.
- После вызова `sendBeacon()` браузер добавляет запросы во внутреннюю очередь запросов. Браузер будет усиленно пытаться отправить запросы в очереди.
- Браузер гарантирует, что попытается отправить запрос, даже после уничтожения исходной страницы.
- Коды ответов, тайм-ауты и любые другие сбои в сети полностью непрозрачны и не могут быть обработаны программно.
- Запрос beacon отправляется со всеми соответствующими файлами cookie в момент первоначального вызова `sendBeacon()`.

ВЕБ-СОКЕТЫ

Веб-сокеты обеспечивают полнодуплексное двунаправленное взаимодействие с сервером по одному длительному соединению. При создании веб-сокета в JS-коде серверу отправляется HTTP-запрос для инициирования подключения. Когда сервер отвечает, подключение использует обновление HTTP для переключения с HTTP на протокол Web Socket. Это означает, что веб-сокеты невозможно реализовать на стандартном HTTP-сервере — для правильной работы требуется специализированный сервер, поддерживающий этот протокол.

Поскольку веб-сокеты используют собственный протокол, их схема URL немного отличается. Вместо схем `http://` и `https://` применяются схемы `ws://` для небезопасных подключений и `wss://` для безопасных. При указании URL-адреса веб-сокета необходимо задавать схему, так как в будущем может быть реализована поддержка других схем.

Преимущество использования специализированного протокола вместо HTTP состоит в том, что между клиентом и сервером можно передавать совсем небольшие объемы данных без служебных HTTP-данных. Применение меньших пакетов данных делает веб-сокеты идеальным решением для мобильных приложений, в которых особенно важны такие факторы, как пропускная способность и время задержки. Недостаток специализированного протокола состоит в том, что для его определения потребовалось больше времени в сравнении с JavaScript API. Веб-сокеты поддерживаются во всех современных браузерах.

API

Чтобы создать веб-сокет, создайте объект `WebSocket`, передав в конструктор URL-адрес подключения:

```
let socket = new WebSocket("ws://www.example.com/server.php");
```

Конструктор `WebSocket` принимает абсолютный URL-адрес. Политика одинакового источника не применяется к веб-сокетам, так что вы можете открыть подключение к любому сайту. Сервер сам решает, будет ли он взаимодействовать со страницей из конкретного источника (он может определить источник запроса по информации, полученной в ходе установления подключения).

Когда объект `WebSocket` создан, браузер пытается создать подключение. Подобно XHR, у объекта `WebSocket` есть свойство `readyState`, которое указывает текущее состояние. Однако его значения отличаются от значений у XHR:

- `WebSocket.OPENING` (0) — подключение устанавливается;
- `WebSocket.OPEN` (1) — подключение установлено;
- `WebSocket.CLOSING` (2) — начинается закрытие подключения;
- `WebSocket.CLOSE` (3) — подключение закрыто.

У объекта `WebSocket` нет события `readystatechange`, однако есть другие события, соответствующие различным состояниям. Первоначальное значение `readyState` всегда равно 0.

Подключение через веб-сокет можно закрыть в любой момент методом `close()`:

```
socket.close();
```

При вызове метода `close()` значение `readyState` немедленно изменяется на 2 (закрытие), а по завершении операции — на 3.

Отправка и получение данных

Когда веб-сокет открыт, вы можете отправлять и получать данные через соединение. Чтобы отправить данные серверу, вызовите метод `send()`, передав в него нужную строку, `ArrayBuffer` или `Blob`, например:

```
let socket = new WebSocket("ws://www.example.com/server.php");

let stringData = "Hello world!";
let arrayBufferData = Uint8Array.from(['f', 'o' 'o']);
let blobData = new Blob(['f', 'o' 'o']);

socket.send(stringData);
socket.send(arrayBufferData.buffer);
socket.send(blobData);
```

Когда сервер отправляет сообщение клиенту, для объекта `WebSocket` генерируется событие `message`. Оно работает так же, как и в других протоколах обмена сообщениями, при этом данные доступны через свойство `event.data`:

```
socket.onmessage = function(event) {
    let data = event.data;

    // какие-то действия с данными
};
```

Подобно данным, которые отправляются на сервер с помощью `send()`, данные, возвращаемые в `event.data`, могут быть получены как `ArrayBuffer` или `Blob`. Это определяется свойством `binaryType` объекта `WebSocket`, который может быть «`blob`» или «`arraybuffer`».

Другие события

У объекта `WebSocket` есть еще три события, которые генерируются во время существования подключения:

- `open` — генерируется при успешном подключении;
- `error` — генерируется при возникновении ошибки, при этом сохранить подключение не удается;
- `close` — генерируется при закрытии подключения.

Объект `WebSocket` не поддерживает слушатели событий DOM Level 2, так что обрабатывать эти события нужно в стиле DOM Level 0:

```
let socket = new WebSocket("ws://www.example.com/server.php");

socket.onopen = function() {
    alert("Connection established.");
};

socket.onerror = function() {
    alert("Connection error.");
};

socket.onclose = function() {
    alert("Connection closed.");
};
```

Из этих трех событий только у события `close` объект `event` содержит дополнительную информацию. У него есть три дополнительных свойства:

- `wasClean` — логическое значение, указывающее, правильно ли было закрыто подключение;
- `code` — числовой код состояния, отправленный сервером;
- `reason` — строковое сообщение, отправленное сервером.

Вы можете показать эту информацию пользователю или применить ее для анализа:

```
socket.onclose = function(event) {
    console.log(`as clean? ${event.wasClean} Code=${event.code} Reason=${
                event.reason}`);
};
```

БЕЗОПАСНОСТЬ

На тему безопасности Ajax и Comet опубликовано много статей и даже целые книги. Обсуждать безопасность сложных Ajax-приложений можно очень долго, но есть некоторые базовые принципы безопасности Ajax, заслуживающие особого внимания.

Прежде всего, любой URL-адрес, доступный с помощью XHR, также доступен браузеру или серверу. Возьмем для примера следующий URL-адрес:

```
/getuserinfo.php?id=23
```

Предположим, что при запросе этого URL-адреса возвращаются некоторые данные о пользователе с идентификатором 23. Ничто не мешает кому-нибудь изменить идентификатор в URL-адресе на 24, 56 или любое другое значение. Файлу getuserinfo.php должно быть известно, действительно ли у инициатора запроса есть

доступ к запрошенным данным, в противном случае сервер будет возвращать их кому угодно.

Такой несанкционированный доступ к ресурсу называется межсайтовой подделкой сценариев (CSRF), при этом неавторизованная система выдает себя серверу, который обрабатывает запрос, за уполномоченную. И крупные, и небольшие Ajax-приложения подвержены CSRF-атакам, проводимым с самыми разными целями: от проверки на уязвимость для совершенствования защиты до вредоносных атак, имеющих целью похищение или уничтожение данных.

Для защиты доступа к URL-адресам с помощью XHR обычно проверяют наличие прав на доступ у отправителя запроса. Это можно сделать следующим образом:

- потребовать использовать SSL для доступа к ресурсам, которые могут быть запрошены с помощью XHR;
- потребовать отправлять вычисляемый маркер с каждым запросом.

Имейте в виду, что следующие меры не защищают от CSRF-атак:

- требование использовать запрос POST вместо GET — это легко изменить;
- использование источника ссылки для определения происхождения запроса — его легко подделать;
- проверка прав с помощью cookie-файла — его также легко подделать.

ИТОГИ

Ajax — это технология получения данных с сервера без обновления текущей страницы. Перечислим ее ключевые характеристики:

- Главный объект, которому технология Ajax обязана своей популярностью, называется `XMLHttpRequest` (XHR).
- Этот объект был разработан корпорацией Microsoft и представлен в Internet Explorer 5 для получения XML-данных с сервера с помощью JavaScript.
- С тех пор объект XHR был продублирован в Firefox, Safari, Chrome и Opera, а консорциум W3C издал спецификацию его поведения, сделав XHR веб-стандартом.
- Несмотря на некоторые различия реализаций, основы работы с объектом XHR мало чем отличаются в разных браузерах, поэтому его можно безопасно использовать в веб-приложениях.

Одним из основных ограничений XHR является политика одинакового источника, которая требует использовать для взаимодействия один домен, один порт и один протокол. Любая попытка доступа к ресурсам в обход этого ограничения приводит к ошибке безопасности, если не используется одобренное кроссдоменное решение. Это решение называется обменом ресурсами с запросом происхождения (CORS);

оно реализовано посредством объекта XHR. Проверка связи с помощью изображения и JSONP — две другие технологии обмена данными с разными доменами, но они менее надежны, чем CORS.

Fetch API был представлен как сквозная замена существующего объекта XHR. API предлагает превосходную структуру, основанную на промисах, более интуитивно понятный интерфейс и первоклассную поддержку Stream API.

Веб-сокеты обеспечивают полнодуплексное двунаправленное взаимодействие с сервером. В отличие от других решений, веб-сокеты используют не HTTP, а специализированный протокол, оптимизированный для быстрой доставки небольших блоков данных. Это требует применения другого веб-сервера, но обеспечивает преимущество в скорости.

25

Клиентское хранилище

- Cookie-файлы
- API хранилища браузера
- IndexedDB

Загрузки с сайта wrox.com для этой главы

Обратите внимание, что все примеры кода для этой главы доступны на веб-сайте книги по ссылке www.wrox.com/go/projavascript4e на вкладке Downloads.

С появлением веб-приложений появилась необходимость хранить пользовательскую информацию непосредственно на клиентском компьютере. Это логично: информация, относящаяся к конкретному пользователю, должна находиться на его компьютере, будь то сведения для входа в систему, предпочтения или другие данные. Соответственно поставщики веб-приложений начали искать способы сохранения данных на стороне клиента. Первое решение поступило от корпорации Netscape, которая предложила использовать для этого cookie-файлы, описанные в спецификации *Persistent Client State*: *HTTP Cookies* («Хранение состояния клиента: HTTP-cookie»), доступной по ссылке http://curl.haxx.se/rfc/cookie_spec.html. Сегодня cookie-файлы — это лишь одна из технологий сохранения данных на стороне клиента.

COOKIE-ФАЙЛЫ

Cookie-файлы изначально были предназначены для сохранения сведений о сеансе на клиентском компьютере. Спецификация предписывала серверу отправлять

в любом ответе на HTTP-запрос HTTP-заголовок `Set-Cookie` со сведениями о сеансе. Например, заголовки ответа сервера могут быть такими:

```
HTTP/1.1 200 OK
Content-type: text/html
Set-Cookie: name=значение
Other-header: значение_другого_заголовка
```

Этот HTTP-ответ задает cookie-файл с именем `"name"` и значением `"значение"`. И имя, и значение при отправке кодируются в формате URL-адреса. Браузер сохраняет эти сведения о сеансе и при каждом последующем запросе отправляет их серверу в HTTP-заголовке `Cookie`, например:

```
GET /index.jsl HTTP/1.1
Cookie: name=значение
Other-header: значение_другого_заголовка
```

Эти дополнительные сведения сервер может использовать для идентификации клиента, который отправил запрос.

Ограничения

Cookie-файлы по природе связаны с конкретным доменом. Заданный cookie-файл прилагается к запросам, отправляемым в тот домен, в котором файл был создан. Это ограничение гарантирует, что информация, сохраненная в cookie-файле, будет доступна только одобренным получателям и не сможет попасть в другие домены.

Поскольку cookie-файлы хранятся на клиентском компьютере, к ним применяются ограничения, гарантирующие, что их невозможно будет использовать со злым умыслом и что они не будут занимать много места на диске.

В целом, если вы используете следующие приблизительные ограничения, у вас не возникнет проблем со всем трафиком браузера:

- 300 cookie-файлов всего.
- 4096 байт на cookie-файл.
- 20 cookie-файлов на домен.
- 81 920 байт на домен.

Общее количество cookie-файлов на домен ограничено и зависит от браузера:

- последние версии Internet Explorer и Edge поддерживают 50 cookie-файлов на домен;
- последние версии Firefox поддерживают 150 cookie-файлов на домен;
- последние версии Opera поддерживают 180 cookie-файлов на домен;
- в Safari и Chrome количество cookie-файлов на домен строго не ограничено.

При создании cookie-файлов сверх этих ограничений браузер начинает удалять имеющиеся cookie-файлы. Internet Explorer и Opera начинают с удаления cookie-файла,

который не использовался дольше всего. Firefox, по всей видимости, выбирает удаляемый cookie-файл случайно, поэтому во избежание непредвиденных последствий важно соблюдать ограничения на количество cookie-файлов.

Размер cookie-файла в браузерах также ограничен. В большинстве браузеров это ограничение составляет около 4096 байт, но ради совместимости кода с разными браузерами лучше ограничивать размер cookie-файла значением в 4095 байт или менее. Это ограничение применяется ко всем cookie-файлам в домене, а не к отдельным.

Если попытаться создать cookie-файл с размером, превышающим ограничение, запрос будет проигнорирован. Имейте в виду, что один знак обычно занимает один байт, если не используются многобайтовые знаки — например, некоторые символы Unicode UTF-8, которые могут содержать до 4 байтов на символ.

Части cookie-файла

Cookie-файлы состоят из перечисленных далее частей, хранящихся в браузере.

- **Имя** — уникальное имя, идентифицирующее cookie-файл. Имена cookie нечувствительны к регистру, например имена `myCookie` и `MyCookie` считаются одинаковыми. Тем не менее на практике лучше предполагать, что имена cookie чувствительны к регистру, потому что так их обрабатывают некоторые серверные программы. Имена cookie-файлов нужно кодировать в формате URL-адресов.
- **Значение** — строковое значение, хранящееся в cookie-файле. Его необходимо кодировать в формате URL-адресов.
- **Домен** — домен, для которого действителен cookie-файл. Все запросы, отправляемые в этот домен, будут содержать данные из cookie. Это значение может включать поддомен (например, www.wrox.com) или исключать его (например, значение .wrox.com действительно для всех поддоменов wrox.com). Если домен не задан явно, предполагается тот домен, в котором был задан cookie-файл.
- **Путь** — путь в указанном домене, для которого cookie-файл следует отправить серверу. Например, вы можете указать, что cookie-файл доступен только по адресу http://www.wrox.com/books/, и тогда страницы на сайте http://www.wrox.com не будут отправлять cookie несмотря на то, что запрос исходит из того же домена.
- **Дата истечения срока действия** — метка времени, указывающая, когда cookie-файл должен быть удален (то есть когда следует прекратить отправлять его серверу). По умолчанию все cookie-файлы удаляются при завершении сеанса браузера, но можно задать и другое время удаления. Это значение задается как дата в формате GMT (Нед, ДД-Мес-ГГГГ ЧЧ:ММ:СС GMT) и указывает точное время, когда cookie-файл должен быть удален. Таким образом, cookie-файл может оставаться на компьютере пользователя даже после закрытия браузера. Если задать уже прошедшую дату, cookie-файл будет удален незамедлительно.

- **Флаг безопасности** — если этот флаг указан, информация из cookie отправляется серверу, только если используется SSL-соединение. Например, при запросе https://www.wrox.com она отправляется, а при запросе http://www.wrox.com — нет.

Каждый из этих элементов данных указывается в составе заголовка `Set-Cookie`, при этом в качестве разделителя используется сочетание точки с запятой и пробела, например:

```
HTTP/1.1 200 OK
Content-type: text/html
Set-Cookie: name=значение; expires=Mon, 22-Jan-07 07:10:24 GMT;
domain=.wrox.com
Other-header: значение_другого_заголовка
```

Этот заголовок определяет cookie-файл с именем `"name"`, который должен устареть 22 января 2007 г. в 7:10:24 по GMT и действителен для домена www.wrox.com и любых других поддоменов wrox.com, таких как p2p.wrox.com.

Флаг безопасности — единственный элемент cookie-файла, который не является парой из имени и значения, это просто слово `"secure"`. Рассмотрим пример:

```
HTTP/1.1 200 OK
Content-type: text/html
Set-Cookie: name=значение; domain=.wrox.com; path=/; secure
Other-header: значение_другого_заголовка
```

В этом фрагменте создается cookie-файл, действительный для всех поддоменов wrox.com и всех страниц в этом домене (что указано с помощью аргумента `path`). Поскольку задан флаг `secure`, этот cookie-файл может передаваться только по SSL-соединению.

Имейте в виду, что домен, путь, дата истечения срока действия и флаг безопасности лишь указывают браузеру, когда следует отправлять cookie-файл вместе с запросом. Эти аргументы не отправляются серверу с информацией cookie-файла; отправляются только пары имен и значений.

Cookie-файлы в JavaScript

Работать с cookie-файлами в JavaScript не просто из-за на редкость неудачного интерфейса — BOM-свойства `document.cookie`. Оно уникально тем, что работает совершенно по-разному в зависимости от того, как используется. При чтении свойство `document.cookie` возвращает строку, содержащую все cookie-файлы, доступные странице (на основе домена, пути, даты истечения срока действия и параметров безопасности). Эта строка представляет собой набор пар имен и значений, разделенных точками с запятой, например:

```
имя1=значение1;имя2=значение2;имя3=значение3
```

Все имена и значения кодируются в формате URL-адресов, поэтому их нужно декодировать с помощью метода `decodeURIComponent()`.

При записи можно присвоить свойству `document.cookie` новую cookie-строку, которая будет интерпретирована и добавлена в существующий набор cookie-файлов. При установке свойства `document.cookie` никакие cookie-файлы не перезаписываются, если задаваемое имя cookie-файла не используется. Для задания cookie-файла используется такой же формат, что и в заголовке `Set-Cookie`:

```
name=значение; expires=срок_действия; path=путь_домена;
domain=имя_домена; secure
```

Из этих параметров обязательны только имя и значение cookie-файла. Вот простой пример:

```
document.cookie = "name=Nicholas";
```

Этот код создает cookie-файл сеанса с именем `"name"` и значением `"Nicholas"`. Этот cookie-файл отправляется серверу при каждом клиентском запросе и удаляется при закрытии браузера. Хотя такой способ вполне приемлем, потому что никакие знаки в имени или значении кодировать не нужно, при задании cookie-файла лучше всегда использовать метод `encodeURIComponent()`, например:

```
document.cookie = encodeURIComponent("name") + "=" +
                  encodeURIComponent("Nicholas");
```

Чтобы указать дополнительные сведения о создаваемом cookie-файле, просто добавьте их к строке в том же формате, что и в заголовке `Set-Cookie`, например:

```
document.cookie = encodeURIComponent("name") + "=" +
                  encodeURIComponent("Nicholas") + "; domain=.wrox.com;
                  path=/";
```

Поскольку читать и записывать cookie-файлы в JavaScript непросто, для этого часто используют вспомогательные функции. Базовых операций над cookie-файлами три: чтение, запись и удаление. Они представлены в объекте `CookieUtil` следующим образом:

```
class CookieUtil {

    static get(name) {
        let cookieName = `${encodeURIComponent(name)}=`,
            cookieStart = document.cookie.indexOf(cookieName),
            cookieValue = null;

        if (cookieStart > -1) {
            let cookieEnd = document.cookie.indexOf(";", cookieStart);
            if (cookieEnd == -1) {
                cookieEnd = document.cookie.length;
            }
            cookieValue = decodeURIComponent(document.cookie.substring(
                cookieStart + cookieName.length, cookieEnd));
        }

        return cookieValue;
    },
```

```
    static set(name, value, expires, path, domain, secure) {
        let cookieText =
            `${encodeURIComponent(name)}=${encodeURIComponent(value)}`

        if (expires instanceof Date) {
            cookieText += `; expires=${expires.toGMTString()}`;
        }

        if (path) {
            cookieText += `; path=${path}`;
        }

        if (domain) {
            cookieText += `; domain=${domain}`;
        }

        if (secure) {
            cookieText += "; secure";
        }

        document.cookie = cookieText;
    },

    static unset(name, path, domain, secure) {
        CookieUtil.set(name, "", new Date(0), path, domain, secure);
    }

};
```

Метод `CookieUtil.get()` получает значение cookie-файла с указанным именем. Для этого он ищет в свойстве `document.cookie` имя cookie-файла, за которым следует знак равенства. Если ему удается найти этот шаблон, вызывается метод `indexOf()` для поиска ближайшей точки с запятой (которая обозначает конец cookie). Если точка с запятой не обнаруживается, это означает, что cookie является последним в строке и всю оставшуюся строку следует считать значением cookie. Это значение декодируется с помощью метода `decodeURIComponent()` и возвращается. Если обнаружить cookie не удается, возвращается значение `null`.

Метод `CookieUtil.set()` задает cookie для страницы. В качестве аргументов он принимает имя cookie, значение cookie, необязательный объект `Date`, указывающий, когда нужно удалить cookie, необязательный URL-путь для cookie, необязательный домен cookie и необязательное логическое значение, указывающее, следует ли добавить флаг `secure`. Аргументы указываются в том порядке, в котором они используются чаще всего, при этом обязательны только первые два. Внутри этого метода имя и значение cookie кодируются в формате URL-адресов с помощью метода `encodeURIComponent()`, после чего проверяются другие параметры. Если аргументом `expires` является объект `Date`, к строке добавляется параметр `expires`, при этом для правильного форматирования даты вызывается метод `toGMTString()` объекта `Date`. В оставшейся части метода мы просто составляем строку cookie и присваиваем ее свойству `document.cookie`.

Непосредственного способа удалить существующий cookie-файл не существует. Вместо этого нужно еще раз задать cookie с теми же путем, доменом и параметрами

безопасности и назначить в качестве даты истечения его срока действия какой-то момент в прошлом. Это и делает метод `CookieUtil.unset()`. Он принимает четыре аргумента: имя удаляемого cookie, необязательный путь, необязательный домен и необязательный аргумент безопасности.

Эти аргументы передаются в метод `CookieUtil.set()` вместе с пустой строкой и датой истечения срока действия, равной 1 января 1970 г. (объект `Date` инициализируется нулевым значением счетчика миллисекунд). В результате cookie-файл удаляется.

Эти методы можно использовать следующим образом:

```
// задание cookie-файлов
CookieUtil.set("name", "Nicholas");
CookieUtil.set("book", "Professional JavaScript");

// чтение значений
alert(CookieUtil.get("name"));    // "Nicholas"
alert(CookieUtil.get("book"));    // "Professional JavaScript"

// удаление cookie-файлов
CookieUtil.unset("name");
CookieUtil.unset("book");

// задание cookie-файла с путем, доменом и датой истечения срока действия
CookieUtil.set("name", "Nicholas", "/books/projs/", "www.wrox.com",
               new Date("January 1, 2010"));

// удаление того же cookie-файла
CookieUtil.unset("name", "/books/projs/", "www.wrox.com");

// задание защищенного cookie-файла
CookieUtil.set("name", "Nicholas", null, null, null, true);
```

Эти методы упрощают использование cookie-файлов для хранения данных на клиенте, обеспечивая синтаксический анализ и конструирование строки cookie.

Вложенные cookie-файлы

Чтобы обойти действующее во многих браузерах ограничение на количество cookie-файлов на домен, некоторые разработчики используют *вложенные cookie* (subcookies). Так называют меньшие фрагменты данных, хранящиеся в одном cookie. Идея в том, чтобы хранить в одном cookie несколько пар имен и значений. Наиболее популярный формат вложенных cookie таков:

```
name=имя1=значение1&имя2=значение2&имя3=значение3&имя4=значение4
```

Вложенные cookie обычно форматируют как строку запроса. Затем эти значения можно сохранить в единственном cookie, а не использовать отдельный cookie для хранения каждой пары имени и значения. Благодаря этому разработчики веб-сайтов и веб-приложений могут хранить больше структурированных данных, не достигая предельного количества cookie на домен.

Для работы с вложенными cookie нужен другой набор методов. Синтаксический анализ и сериализация вложенных cookie выполняются иначе и более сложны из-за предполагаемого способа использования таких cookie. Например, чтобы получить вложенный cookie, требуется выполнить те же базовые действия, но перед декодированием значения нужно найти информацию вложенного cookie:

```
class SubCookieUtil {

    static get(name, subName) {
        let subCookies = thisSubCookieUtil.getAll(name);
        return subCookies ? subCookies[subName] : null;
    },

    static getAll(name) {
        let cookieName = encodeURIComponent(name) + "=",
            cookieStart = document.cookie.indexOf(cookieName),
            cookieValue = null,
            cookieEnd,
            subCookies,
            parts,
            result = {};

        if (cookieStart > -1) {
            cookieEnd = document.cookie.indexOf(";", cookieStart);
            if (cookieEnd == -1) {
                cookieEnd = document.cookie.length;
            }
            cookieValue = document.cookie.substring(cookieStart +
                          cookieName.length, cookieEnd);

            if (cookieValue.length > 0) {
                subCookies = cookieValue.split("&");

                for (let i=0, len=subCookies.length; i < len; i++) {
                    parts = subCookies[i].split("=");
                    result[decodeURIComponent(parts[0])] =
                         decodeURIComponent(parts[1]);
                }

                return result;
            }
        }

        return null;
    },

    // другой код
};
```

Для получения вложенных cookie в этом коде используются методы `get()` и `getAll()`. Первый из них получает значение одного вложенного cookie, тогда как второй получает все вложенные cookie, возвращая их в виде объекта, свойствами которого являются имена cookie, а значениями — значения cookie. Метод `get()` принимает два аргумента: имя cookie и имя вложенного cookie. Он просто вызывает метод

getAll() для получения всех вложенных cookie, а затем возвращает только нужный cookie (или null, если cookie не существует).

Метод SubCookieUtil.getAll() очень похож на CookieUtil.get() тем, как он разбирает значение cookie. Его отличие состоит в том, что значение cookie не декодируется немедленно. Вместо этого оно разделяется по знаку амперсанда для сохранения всех вложенных cookie в массиве. Затем каждый вложенный cookie разделяется по знаку равенства, чтобы первым элементом в массиве parts было имя вложенного cookie, а вторым — его значение. Оба элемента декодируются с помощью метода decodeURIComponent() и назначаются объекту result, который возвращается как значение метода. Если cookie-файл не существует, возвращается значение null.

Эти методы можно использовать следующим образом:

```
// предполагается, что
// document.cookie=data=name=Nicholas&book=Professional%20JavaScript

// получение всех вложенных cookie-фрагментов
let data = SubCookieUtil.getAll("data");
alert(data.name);    // "Nicholas"
alert(data.book);    // "Professional JavaScript"

// индивидуальное получение вложенных cookie-фрагментов
alert(SubCookieUtil.get("data", "name"));    // "Nicholas"
alert(SubCookieUtil.get("data", "book"));    // "Professional JavaScript"
```

Для записи вложенных cookie-фрагментов хорошо подходят методы set() и setAll():

```
class SubCookieUtil {

    // предыдущий код
    static set(name, subName, value, expires, path, domain, secure) {
        let subcookies = SubCookieUtil.getAll(name) || {};
        subcookies[subName] = value;
        SubCookieUtil.setAll(name, subcookies, expires, path, domain, secure);
    },

    static setAll(name, subcookies, expires, path, domain, secure) {

        let cookieText = encodeURIComponent(name) + "=",
            subcookieParts = new Array(),
            subName;

        for (subName in subcookies) {
            if (subName.length > 0 && subcookies.hasOwnProperty(subName)) {
                subcookieParts.push(
                    '${encodeURIComponent(subName)}=${encodeURIComponent(subcookies
                                                    [subName])}');
            }
        }

        if (cookieParts.length > 0) {
            cookieText += subcookieParts.join("&");

            if (expires instanceof Date) {
                cookieText += `; expires=${expires.toGMTString()}`;
```

```
            }

            if (path) {
                cookieText += `; path=${path}`;
            }

            if (domain) {
                cookieText += `; domain=${domain}`;
            }

            if (secure) {
                cookieText += "; secure";
            }
        } else {
            cookieText += `; expires=${(new Date(0)).toGMTString()}`;
        }
        document.cookie = cookieText;
    },

    // другой код
};
```

Метод `set()` принимает семь аргументов: имя cookie, имя вложенного cookie, значение вложенного cookie, необязательный объект `Date`, содержащий дату и время истечения срока действия cookie, необязательный путь для cookie, необязательный домен cookie и необязательный логический флаг безопасности. Все необязательные аргументы относятся к самому cookie, а не к вложенному cookie. Чтобы можно было сохранить несколько вложенных cookie в одном cookie, их путь, домен и флаг безопасности должны быть одинаковыми. Дата истечения срока действия относится ко всему cookie и может быть задана при записи отдельного вложенного cookie. В теле метода мы первым делом получаем все вложенные cookie для указанного имени cookie. Если метод `getAll()` возвращает `null`, переменной `subcookies` с помощью логического оператора ИЛИ назначается новый объект. После этого объекту `subcookies` присваивается значение вложенного cookie, и этот объект передается в метод `setAll()`.

Метод `setAll()` принимает шесть аргументов: имя cookie, объект, содержащий все вложенные cookie, и остальные необязательные аргументы, используемые в методе `set()`. В теле метода свойства второго аргумента перебираются в цикле `for-in`. Для сохранения нужных данных служит метод `hasOwnProperty()`, который гарантирует, что во вложенных cookie будут сериализованы только свойства экземпляра. Поскольку именем свойства может быть пустая строка, перед добавлением значения к результату также проверяется длина имени свойства. Все пары из имен и значений вложенных cookie добавляются в массив `subcookieParts`, чтобы позднее их можно было легко объединить с помощью амперсанда, используя метод `join()`. Остальной код не отличается от метода `CookieUtil.set()`.

Эти методы можно использовать следующим образом:

```
//предполагается, что
// document.cookie=data=name=Nicholas&book=Professional%20JavaScript
```

```
// задание двух вложенных cookie
SubCookieUtil.set("data", "name", "Nicholas");
SubCookieUtil.set("data", "book", "Professional JavaScript");

// задание всех вложенных cookie с датой истечения срока действия
SubCookieUtil.setAll("data", { name: "Nicholas",
    book: "Professional JavaScript" }, new Date("January 1, 2010"));

// изменение значения name и даты истечения срока действия cookie
SubCookieUtil.set("data", "name", "Майкл", new Date("February 1, 2010"));
```

Нам осталось обсудить методы удаления вложенных cookie. Для удаления обычного cookie достаточно задать уже прошедшую дату истечения срока действия, но вложенный cookie удалить сложнее. Для этого нужно получить все вложенные cookie, содержащиеся в обычном cookie, удалить тот из них, который не нужен, а затем сохранить оставшиеся значения вложенных cookie. Рассмотрим следующий код:

```
class SubCookieUtil = {

    // предыдущий код

    static unset(name, subName, path, domain, secure) {
        let subcookies = SubCookieUtil.getAll(name);
        if (subcookies) {
            delete subcookies[subName];
            SubCookieUtil.setAll(name, subcookies, null, path, domain, secure);
        }
    },

    static unsetAll(name, path, domain, secure) {
        SubCookieUtil.setAll(name, null, new Date(0), path, domain, secure);
    }

};
```

Метод `unset()` из этого фрагмента удаляет один вложенный cookie, оставляя остальное содержимое неизменным, а метод `unsetAll()` эквивалентен методу `CookieUtil.unset()`, который удаляет cookie полностью. Как и в методах `set()` и `setAll()`, путь, домен и флаг безопасности должны соответствовать параметрам, с которыми cookie был создан. Эти методы можно использовать следующим образом:

```
// удаление вложенного cookie с именем "name"
SubCookieUtil.unset("data", "name");

// удаление всего cookie
SubCookieUtil.unsetAll("data");
```

Если вас беспокоит ограничение количества cookie на домен, вложенные cookie могут оказаться привлекательной альтернативой, но вам придется более тщательно отслеживать размеры cookie-файлов, чтобы не превзойти предельный размер отдельного cookie.

Замечания по поводу cookie-файлов

Существуют также *cookie-файлы, используемые только в HTTP* (HTTP-only cookie). Такие cookie можно задать и в браузере, и с сервера, а прочитать — только с сервера, потому что получить их значения с помощью JavaScript нельзя.

Поскольку все cookie отправляются браузером как заголовки запроса, хранение большого объема информации в cookie может сказаться на скорости обработки запросов, адресованных в конкретный домен. Чем больше данных содержит cookie, тем дольше обрабатывается запрос. Несмотря на то что браузер налагает ограничения на размер cookie, во избежание проблем с быстродействием имеет смысл хранить в cookie как можно меньше информации.

Из-за целого ряда ограничений cookie-файлы плохо подходят для хранения больших объемов данных. К счастью, существуют более эффективные альтернативы.

ПРИМЕЧАНИЕ Хранить в cookie важные или конфиденциальные данные вроде номеров кредитных карт или личных адресов не рекомендуется, потому что содержимое cookie может быть доступно другим пользователям.

ВЕБ-ХРАНИЛИЩЕ

Впервые веб-хранилище было описано в спецификации Web Applications 1.0, подготовленной рабочей группой Web Hypertext Application Technical Working Group (WHAT-WG). Первоначальное описание из этой спецификации позднее было включено в HTML5, а в итоге стало отдельной спецификацией. Веб-хранилище устраняет некоторые ограничения cookie в тех ситуациях, когда данные нужны только на стороне клиента и их не требуется многократно отправлять серверу. Самая современная версия веб-хранилища — второй выпуск. Основные цели использования веб-хранилища таковы:

- хранение данных сеанса вне cookie-файлов;
- хранение больших объемов данных между сеансами.

Второй выпуск спецификации Web Storage включает определения двух объектов: `localStorage`, постоянный способ хранения, и `sessionStorage`, ограниченный временем сессии. Оба этих API хранилища браузера предоставляют два различных способа хранения данных в браузере, которые могут выдержать перезагрузку страницы. И `localStorage`, и `sessionStorage` доступны как свойство `window` во всех версиях браузеров основных поставщиков, выпущенных с 2009 г.

ПРИМЕЧАНИЕ `globalStorage` ранее был частью первой редакции спецификации Web Storage и с тех пор устарел.

Тип Storage

Тип `Storage` позволяет хранить пары имен и значений, при этом максимальный объем хранилища определяется браузером. Экземпляр `Storage` работает подобно любым другим объектам и имеет следующие дополнительные методы:

- `clear()` — удаляет все значения (не реализован в Firefox);
- `getItem(имя)` — получает значение, соответствующее указанному имени;
- `key(индекс)` — получает имя значения в указанной позиции;
- `removeItem(имя)` — удаляет пару имени и значения, указанную по имени;
- `setItem(имя, значение)` — задает значение для указанного имени.

Методы `getItem()`, `removeItem()` и `setItem()` можно вызывать напрямую или косвенно, манипулируя объектом `Storage`. Поскольку элементы хранятся в этом объекте как свойства, вы можете читать их и задавать их значения, используя точечную или скобочную нотацию, а также удалять их с помощью оператора `delete`. Тем не менее рекомендуется применять для этого методы, чтобы не перезаписать один из уже доступных членов объекта с ключом.

Узнать количество пар имен и значений в объекте `Storage` можно с помощью свойства `length`. Объем всех данных в объекте определить невозможно, но в Internet Explorer 8 доступно свойство `remainingSpace`, которое возвращает объем доступного в хранилище места в байтах.

> **ПРИМЕЧАНИЕ** Объект `Storage` может хранить только строки. Нестроковые данные перед сохранением в нем автоматически преобразуются в строку. Учитывайте, что это преобразование не отменяется при выгрузке.

Объект sessionStorage

Объект `sessionStorage` хранит данные только в течение сеанса, то есть до закрытия браузера (в этом смысле он похож на сеансовый cookie). Данные, сохраненные в объекте `sessionStorage`, остаются в нем при обновлениях страницы и даже могут оставаться нетронутыми при сбое и перезапуске браузера (в Firefox и WebKit, но не в Internet Explorer).

Поскольку объект `sessionStorage` связан с сеансом сервера, он недоступен, если файл используется локально. Данные, сохраненные в объекте `sessionStorage`, доступны только со страницы, которая первоначально поместила их в него, из-за чего в многостраничных приложениях от него мало пользы.

Объект `sessionStorage` является экземпляром типа `Storage`, поэтому вы можете назначить ему данные, вызвав метод `setItem()` или напрямую задав новое свойство. Вот соответствующие примеры:

```
// сохранение данных с помощью метода
sessionStorage.setItem("name", "Nicholas");
```

```
// сохранение данных с помощью свойства
sessionStorage.book = "Professional JavaScript";
```

Все современные браузеры реализуют запись в хранилище как блокирующее синхронное действие, поэтому данные, добавляемые в хранилище, фиксируются немедленно. Реализация API может не сразу записать значение на диск (и предпочтет изначально использовать другое физическое хранилище), но это различие невидимо на уровне JavaScript, и любые записи, использующие какую-либо форму веб-хранилища, могут быть немедленно прочитаны.

В старых версиях Internet Explorer 8 можно форсировать запись на диск, вызвав метод `begin()` перед назначением любых новых данных свойствам и метод `commit()` после, например:

```
// только для IE8
sessionStorage.begin();
sessionStorage.name = "Nicholas";
sessionStorage.book = "Professional JavaScript";
sessionStorage.commit();
```

В этом фрагменте значения свойств `name` и `book` записываются, как только вызывается метод `commit()`. Вызов метода `begin()` гарантирует, что во время выполнения этого кода данные не будут записываться на диск. Если данных немного, такой транзакционный подход не требуется, но при записи больших объемов данных он приобретает смысл.

При наличии данных в объекте `sessionStorage` их можно получить, вызвав метод `getItem()` или прочитав свойство напрямую:

```
// получение данных с помощью метода
let name = sessionStorage.getItem("name");

// получение данных с помощью свойства
let book = sessionStorage.book;
```

Перебрать значения в объекте `sessionStorage` можно с помощью свойства `length` и метода `key()`:

```
for (let i=0, len = sessionStorage.length; i < len; i++) {
    let key = sessionStorage.key(i);
    let value = sessionStorage.getItem(key);
    alert(`${key}=`${value}`);}
```

Для последовательного доступа к парам имен и значений в объекте `sessionStorage` можно сначала получить имя элемента данных в указанной позиции методом `key()`, а затем вызвать для получения значения метод `getItem()`, передав в него имя элемента.

Вы также можете перебрать значения в объекте `sessionStorage` в цикле `for-in`:

```
for (let key in sessionStorage) {
    let value = sessionStorage.getItem(key);
    alert(`${key}=${value}`);
}
```

На каждой итерации цикла переменной `key` назначается очередное имя из объекта `sessionStorage`, но встроенные методы, как и свойство `length`, не возвращаются.

Удалить данные из объекта `sessionStorage` можно, применив оператор `delete` к соответствующему свойству или вызвав метод `removeItem()`:

```
// удаление значения с помощью оператора delete
delete sessionStorage.name;

// удаление значения с помощью метода
sessionStorage.removeItem("book");
```

Объект `sessionStorage` рекомендуется использовать для хранения небольших фрагментов данных, действительных только в течение сеанса. Для хранения данных между сеансами лучше подходит объект `globalStorage` или `localStorage`.

Объект localStorage

Переработанная спецификация HTML5 предписывает использовать для хранения клиентских данных объект `localStorage` вместо `globalStorage`. Чтобы можно было получить доступ к тому же объекту `localStorage`, страницы должны быть выданы из того же домена (поддомены не допускаются) с использованием того же протокола и того же порта.

Поскольку объект `localStorage` является экземпляром типа `Storage`, его можно использовать так же, как и объект `sessionStorage`. Вот несколько примеров:

```
// сохранение данных с помощью метода
localStorage.setItem("name", "Nicholas");

// сохранение данных с помощью свойства
localStorage.book = "Professional JavaScript";

// получение данных с помощью метода
let name = localStorage.getItem("name");

// получение данных с помощью свойства
let book = localStorage.book;
```

Разница между этими двумя способами хранения заключается в том, что данные, хранящиеся в `localStorage`, сохраняются до тех пор, пока они не будут специально удалены с помощью JavaScript или пользователь не очистит кеш браузера. Данные `localStorage` будут существовать после перезагрузки страницы, закрытия окон и вкладок и перезапуска браузера.

Событие storage

При изменении объекта `Storage` для документа генерируется событие `storage`. Это происходит при задании значений с помощью свойств или метода `setItem()`, при удалении значений с помощью оператора `delete` или метода `removeItem()` и при каждом вызове метода `clear()`. Объект `event` этого события имеет четыре свойства:

- `domain` — домен, для которого было изменено хранилище;
- `key` — заданный или измененный ключ;
- `newValue` — значение, присвоенное ключу, или `null`, если ключ был удален;
- `oldValue` — значение до изменения ключа.

Для прослушивания события `storage` можно использовать следующий код:

```
window.addEventListener("storage",
    (event) => alert('Storage changed for ${event.domain}'));
```

Событие `storage` генерируется при любых изменениях объектов `sessionStorage` и `localStorage`, не делая различий между ними.

Пределы и ограничения

Как и другие решения для хранения данных на стороне клиента, веб-хранилище имеет ограничения, которые специфичны для браузера. Вообще говоря, ограничение размера клиентских данных задается по отдельности для каждого источника (протокол, домен и порт), так что каждому источнику выделяется фиксированный объем места для хранения данных. Ограничение применяется на основе анализа источника страницы с данными.

Ограничения хранилища для `localStorage` и `sessionStorage` несовместимы во всех браузерах, но большинство из них ограничат объем хранилища для источника до 5 МБ. Таблицу, содержащую современные ограничения хранения для каждого носителя, можно найти по адресу https://www.html5rocks.com/en/tutorials/offline/quota-research/.

Дополнительные сведения об этих ограничениях см. на странице тестирования поддержки веб-хранилища по адресу http://dev-test.nemikor.com/web-storage/support-test/.

INDEXEDDB

Indexed Database API (IndexedDB) служит для хранения структурированных данных в браузере. Он был разработан как альтернатива Web SQL Database API, который уже признан устаревшим. Целью его разработчиков было создание API, позволяющего легко сохранять и получать JavaScript-объекты и при этом поддерживающего запросы и поиск данных.

IndexedDB почти полностью асинхронен. Большинство операций представляют собой запросы, которые выполняются по прошествии некоторого времени и завершаются успехом или ошибкой. Почти каждая операция IndexedDB требует подключения обработчиков событий `error` и `success`, чтобы можно было определить ее результат.

Начиная с 2017 г. последние версии браузеров большинства крупных поставщиков (Chrome, Firefox, Opera, Safari) полностью поддерживают IndexedDB. Браузеры Internet Explorer 10/11 и Edge частично поддерживают IndexedDB.

Базы данных

IndexedDB — это база данных, похожая на базы данных, которые вы, вероятно, уже использовали, такие как MySQL или Web SQL Database. Важное отличие IndexedDB состоит в том, что в ней для хранения данных используются не таблицы, а хранилища объектов. База данных IndexedDB — это просто набор хранилищ объектов, объединенных общим именем — в стиле NoSQL.

Чтобы использовать базу данных, нужно сначала открыть ее методом `indexedDB.open()`, передав в него ее имя. Если база данных с этим именем уже существует, она будет открыта; если такой базы данных нет, она будет создана и открыта. Метод `indexDB.open()` возвращает экземпляр `IDBRequest`, к которому можно подключить обработчики событий `error` и `success`. Вот пример:

```
let db,
    request,
    version = 1;

request = indexedDB.open("admin", version);
request.onerror = (event) =>
    alert(`Failed to open: ${event.target.errorCode}`);

request.onsuccess = (event) => {
    db = event.target.result;
};
```

Ранее в IndexedDB использовался метод `setVersion()` для указания, к какой версии следует обращаться. Этот метод теперь устарел; как показано здесь, версия теперь указывается при открытии базы данных. Номера версий будут преобразованы в длинное число без знака, поэтому используйте целые числа вместо десятичных точек.

В обоих обработчиках событий свойство `event.target` указывает на переменную `request`, так что они взаимозаменяемы. Если вызывается обработчик события `success`, объект базы данных (`IDBDatabase`) доступен как свойство `event.target.result`, и мы сохраняем его в переменной `database`. С этого момента все запросы базы данных выполняются через переменную `database`. Если происходит ошибка, суть проблемы можно узнать по коду ошибки, сохраненному в свойстве `event.target.errorCode`.

> **ПРИМЕЧАНИЕ** Ранее для указания ошибки в `IndexedDB` использовался `IDBDatabaseException`, который был заменен стандартными `DOMExceptions`.

Хранилища объектов

Как только подключение к базе данных установлено, можно взаимодействовать с хранилищами объектов. Если версия базы данных отличается от ожидаемой,

вероятно, вам придется создать такое хранилище, но перед этим важно подумать о том, какие данные вам нужно в нем хранить.

Предположим, что вам нужно хранить пользовательские записи, содержащие имя пользователя, пароль и т. д. Объект для хранения одной записи может быть таким:

```
let user = {
    username: "007",
    firstName: "James",
    lastName: "Bond",
    password: "foo"
};
```

С одного взгляда на этот объект ясно, что ключом в хранилище следует сделать свойство `username`. Имя пользователя должно быть глобально уникальным, и в большинстве случаев именно его вы будете использовать для доступа к данным. Это важно потому, что ключ нужно указать при создании хранилища объектов.

Версия базы данных определяет схему, которая состоит из хранилищ объектов в базе данных и структур этих хранилищ объектов. Если база данных еще не существует, операция `open()` создает ее; затем запускается событие `anupgradeneeded`. Вы можете установить обработчик для этого события и создать схему базы данных в обработчике. Если база данных существует, но был указан обновленный номер версии, немедленно запускается событие `anupgradeneeded`, что позволяет предоставить обновленную схему в обработчике событий.

Вот как можно создать хранилище объектов для этих пользователей:

```
request.onupgradeneeded = (event) => {
    const db = event.target.result;
    // Удаление текущего существующего хранилища объектов. Это полезно для
    // тестирования, но стирает все данные каждый раз при вызове обработчика.
    if (db.objectStoreNames.contains("users")) {
        db.deleteObjectStore("users");
    }

    db.createObjectStore("users", { keyPath: "username" });
};
```

Свойство `keyPath` второго аргумента указывает имя свойства хранимых объектов, которое следует использовать в качестве ключа.

Транзакции

После создания хранилища объектов все последующие операции с ним выполняются как *транзакции*. Чтобы создать транзакцию, нужно вызвать метод `transaction()` для объекта базы данных. Каждый раз, когда требуется прочитать или изменить данные, все изменения следует сгруппировать в транзакцию. Создать простейшую транзакцию можно так:

```
let transaction = db.transaction();
```

Если аргументы не указаны, все хранилища объектов в базе данных доступны только для чтения. Для оптимизации можно указать имена одного или нескольких хранилищ объектов, к которым требуется получить доступ:

```
let    transaction = db.transaction("users");
```

Этот код загружает для выполнения транзакции только сведения о хранилище объектов `users`. Если вам нужен доступ к нескольким хранилищам объектов, можно передать в метод `transaction()` массив строк:

```
let transaction = db.transaction(["users", "anotherStore"]);
```

Как уже отмечалось, каждая из этих транзакций получает доступ к базе данных в режиме только для чтения. Изменить режим доступа можно с помощью второго аргумента — одной из строк `"readonly"`, `"readwrite"` или `"versionchange"`. Второй аргумент в `transaction()` можно определить следующим образом:

```
let transaction = db.transaction("users", "readwrite");
```

Эта транзакция может читать хранилище объектов `users` и записывать данные в него.

Когда у вас есть ссылка на транзакцию, доступ к конкретному хранилищу объектов обеспечивается методом `objectStore()`, которому передают имя хранилища. После этого можно использовать методы `add()` и `put()`, как было показано, а также метод `get()` для получения значений, `delete()` для удаления объекта и `clear()` для удаления всех объектов. Методы `get()` и `delete()` принимают в качестве аргумента ключ объекта, и все эти пять методов создают объект запроса, например:

```
const transaction = db.transaction("users"),
    store = transaction.objectStore("users"),
    request = store.get("007");
request.onerror = (event) => alert("Did not get the object!");
request.onsuccess = (event) => alert(event.target.result.firstName);
```

Поскольку в рамках одной транзакции можно выполнить любое количество запросов, у самого объекта транзакции есть обработчики событий `error` и `complete`. Они используются для получения сведений о состоянии транзакции:

```
transaction.onerror = (event) => {
    // транзакция была отменена
};

transaction.oncomplete = (event) => {
    // транзакция выполнена успешно
};
```

Помните, что объект `event` события `complete` не предоставляет доступ к данным, возвращенным запросом `get()`, так что для таких запросов вам все равно нужно обрабатывать событие `success`.

Вставка данных

Поскольку теперь у вас есть ссылка на хранилище объектов, можно заполнить хранилище объектов данными, используя `add()` или `put()`. Оба эти метода принимают

один аргумент — объект для хранения и сохраняют объект в хранилище объектов. Разница между ними возникает только тогда, когда объект с таким же ключом уже существует в хранилище объектов. В этом случае `add()` выдаст ошибку, а `put()` просто перезапишет объект. Проще говоря, представьте, что `add()` используется для вставки новых значений, а `put()` используется для обновления значений. Итак, чтобы инициализировать хранилище объектов в первый раз, можно сделать что-то вроде этого:

```
// users – массив новых пользователей
for (let user of users) {
    store.add(user);
}
```

Каждый вызов `add()` или `put()` создает новый запрос на обновление хранилища объектов. Если нужно убедиться, что запрос успешно выполнен, сохраните объект запроса в переменной и назначьте обработчики событий `onerror` и `onsuccess`:

```
// users –массив новых пользователей
let request,
    requests = [];
for (let user of users) {
    request = store.add(user);
    request.onerror = () => {
        // обработка ошибки
    };
    request.onsuccess = () => {
        // обработка успеха
    };
    requests.push(request);
}
```

Как только хранилище объектов будет создано и заполнено данными, наступит пора делать запросы.

Запросы с курсорами

С помощью транзакции можно напрямую получить один элемент с известным ключом. Если требуется получить несколько элементов, нужно создать в транзакции *курсор* (cursor) — указатель в наборе результатов. В отличие от обычных запросов баз данных, курсор не собирает весь набор результатов, а указывает на первый результат и не ищет следующий, пока не получит команду это сделать.

Чтобы создать курсор, нужно вызвать метод `openCursor()` для хранилища объектов. Как и другие IndexedDB-операции, этот метод возвращает запрос, так что вы должны назначить обработчики событий `success` и `error`, например:

```
const transaction = db.transaction("users"),
    store = transaction.objectStore("users"),
    request = store.openCursor();

request.onsuccess = (event) => {
    // код, выполняемый при успешном завершении операции
};
```

```
request.onfailure = (event) => {
    // обработка сбоя
};
```

Когда вызывается обработчик события `success`, следующий элемент в хранилище объектов доступен с помощью свойства `event.target.result`, которое содержит экземпляр `IDBCursor` при наличии следующего элемента или значение `null`, если элементов больше нет. Перечислим свойства экземпляра `IDBCursor`:

- `direction` — число, указывающее направление, в котором должен перемещаться курсор, и задающее необходимость пересекать все повторяющиеся значения. Существует четыре возможных строковых значения: `"next"`, `"nextunique"`, `"prev"` и `"prevunique"`.
- `key` — ключ объекта.
- `value` — фактический объект.
- `primaryKey` — ключ, используемый курсором. Может быть ключом объекта или индексным ключом (см. далее).

Получить информацию об одном результате можно следующим образом:

```
request.onsuccess = (event) => {
    const cursor = event.target.result;
    if (cursor) {    // проверка обязательна
        console.log(`Key: ${cursor.key}, Value: ${JSON.stringify(cursor.value)}`);
    }
};
```

Значение `cursor.value` в этом примере является объектом, поэтому перед отображением оно кодируется в формате JSON.

С помощью курсора можно обновить отдельную запись. Метод `update()` обновляет текущее значение курсора указанным объектом. Как и другие подобные операции, метод `update()` создает новый запрос, поэтому, чтобы узнать результат, нужно назначить обработчики событий `success` и `error`:

```
request.onsuccess = (event) => {
    const cursor = event.target.result;
    let value,
        updateRequest;

    if (cursor) {    // проверка обязательна
        if (cursor.key == "foo") {
            value = cursor.value;          // получение текущего значения
            value.password = "magic!";     // обновление пароля

            // запрос на сохранение обновления
            updateRequest = cursor.update(value);
            updateRequest.onsuccess = () => {
                // код, выполняемый при успешном завершении операции
            };
            updateRequest.onfailure = () => {
                // обработка сбоя
```

```
            };
        }
    }
};
```

Вы также можете удалить элемент в текущей позиции с помощью метода `delete()`. Как и метод `update()`, он тоже создает запрос:

```
request.onsuccess = (event) => {
    const cursor = event.target.result;
    let value,
        deleteRequest;

    if (cursor) {    // проверка обязательна
        if (cursor.key == "foo") {

            // запрос на удаление значения
            deleteRequest = cursor.delete();
            deleteRequest.onsuccess = () => {
                // код, выполняемый при успешном завершении операции
            };
            deleteRequest.onfailure = () => {
                // обработка сбоя
            };
        }
    }
};
```

Если у транзакции нет разрешения на изменение хранилища объектов, методы `update()` и `delete()` генерируют ошибки.

Каждый курсор выполняет по умолчанию только один запрос. Чтобы выполнить другой запрос, необходимо вызвать один из следующих методов:

- `continue(`*ключ*`)` — перемещает курсор к следующему элементу в наборе результатов. Аргумент *ключ* не обязателен. Если он не указан, курсор просто перемещается к следующему элементу; если он указан, курсор перемещается к указанному ключу.
- `advance(`*количество*`)` — перемещает курсор вперед на указанное количество элементов.

Каждый из этих методов предписывает курсору повторно использовать тот же запрос, так что при этом задействуются те же самые обработчики событий `success` и `failure`, пока они не станут ненужными. Например, следующий код перебирает все элементы в хранилище объектов:

```
request.onsuccess = (event) => {
    const cursor = event.target.result;
    if (cursor) {    // проверка обязательна
        console.log(`Key: ${cursor.key}, Value: ${JSON.stringify(cursor.value)}`);
        cursor.continue();    // переход к следующему значению
    } else {
        console.log("Готово!");
    }
};
```

Вызов `continue()` инициирует другой запрос, в результате которого снова вызывается обработчик события `success`. По исчерпании элементов этот обработчик вызывается в последний раз, при этом свойство `event.target.result` равно `null`.

Диапазоны ключей

Курсоры могут показаться недостаточно эффективными, потому что они ограничивают способы получения данных. Чтобы сделать работу с курсорами немного более управляемой, используют *диапазоны ключей* (key ranges), которые представляются экземплярами типа `IDBKeyRange`.

Указать диапазон ключей можно четырьмя разными способами. Первый — вызвать метод `only()`, передав ему ключ, который нужно получить:

```
const onlyRange = IDBKeyRange.only("007");
```

В данном случае будет получено только значение с ключом `"007"`. Создание курсора с таким диапазоном аналогично непосредственному доступу к хранилищу объектов и вызову `get("007")`.

Диапазон второго типа определяет нижнюю границу набора результатов, то есть указывает для курсора начальный элемент. Например, если указать следующий диапазон ключей, курсор переберет элементы до конца, начиная с ключа `"007"`:

```
// проход к концу, начиная с элемента "007"
const lowerRange = IDBKeyRange.lowerBound("007");
```

Если нужно начать с элемента, следующего за значением с ключом `"007"`, можно передать в метод значение `true` в качестве второго аргумента:

```
// проход к концу, начиная после элемента "007"
const lowerRange = IDBKeyRange.lowerBound("007", true);
```

Диапазон третьего типа определяет верхнюю границу набора результатов, то есть ключ конечного значения. Такой диапазон задается с помощью метода `upperBound()`. В следующем фрагменте курсор остановится, когда дойдет до значения с ключом `"ace"`:

```
// проход с начала до элемента "ace"
const upperRange = IDBKeyRange.upperBound("ace");
```

Если включать указанный ключ в набор результатов не нужно, передайте в метод в качестве второго аргумента значение `true`:

```
// проход с начала до элемента перед "ace"
const upperRange = IDBKeyRange.upperBound("ace", true);
```

Указать нижнюю и верхнюю границы можно с помощью метода `bound()`. Он принимает четыре аргумента: ключ нижней границы, ключ верхней границы, необязательное логическое значение, предписывающее пропустить нижнюю границу, и необязательное логическое значение, предписывающее пропустить верхнюю границу. Вот несколько примеров:

```
// проход до элемента "ace", начиная с элемента "007"
const boundRange = IDBKeyRange.bound("007", "ace");

// проход до элемента "ace", начиная с элемента после "007"
const boundRange = IDBKeyRange.bound("007", "ace", true);

// проход до элемента перед "ace", начиная с элемента после "007"
const boundRange = IDBKeyRange.bound("007", "ace", true, true);

// проход до элемента перед "ace", начиная с элемента "007"
const boundRange = IDBKeyRange.bound("007", "ace", false, true);
```

Если затем передать определенный диапазон в метод `openCursor()`, будет создан курсор, остающийся в заданных пределах:

```
const store = db.transaction("users").objectStore("users"),
    range = IDBKeyRange.bound("007", "ace");
    request = store.openCursor(range);

request.onsuccess = function(event) {
    const cursor = event.target.result;
    if (cursor) {    // проверка обязательна
        console.log(`Key: ${cursor.key}, Value: ${JSON.stringify(cursor.value)}`);
        cursor.continue();    // переход к следующему элементу
    } else {
        console.log("Готово!");
    }
};
```

Этот код выводит значения между ключами `"007"` и `"ace"`. Очевидно, что их будет меньше, чем в примере из предыдущего раздела.

Указание направления перемещения курсора

На самом деле у метода `openCursor()` два аргумента. Первым является экземпляр `IDBKeyRange`, а вторым — строка, указывающая направление перемещения курсора.

Обычно курсор начинает обработку с первого элемента в хранилище объектов и продвигается к последнему с каждым вызовом метода `continue()` или `advance()`. По умолчанию такие курсоры используют константу направления `"next"`. При наличии дубликатов в хранилище может потребоваться пропускающий их курсор. Его можно реализовать, передав в метод `openCursor()` в качестве второго аргумента значение `"nextunique"`:

```
const transaction = db.transaction("users"),
    store = transaction.objectStore("users"),
    request = store.openCursor(null, "nextunique");
```

Обратите внимание, что первым аргументом здесь является значение `null`, которое предписывает использовать предлагаемый по умолчанию диапазон ключей, охватывающий все значения. Этот курсор перебирает элементы в хранилище объектов от первого до последнего, пропуская при этом дубликаты.

Вы также можете создать курсор, который перемещается по хранилищу от последнего элемента к первому. Для этого следует передать в метод `openCursor()` значение `"prev"` или `"prevunique"` (последнее указывает пропускать дубликаты):

```
const transaction = db.transaction("users"),
    store = transaction.objectStore("users"),
    request = store.openCursor(null, "prevunique");
```

Если курсор открыт с помощью значения `"prev"` или `"prevunique"`, при каждом вызове метода `continue()` или `advance()` курсор будет перемещаться по хранилищу объектов назад, а не вперед.

Индексы

Возможно, в некоторых наборах данных вы захотите указать более одного ключа для хранилища объектов. Например, если вы отслеживаете пользователей по идентификатору и имени, может потребоваться получать доступ к записям с помощью любого из этих двух элементов. Для этого имеет смысл использовать идентификатор пользователя как первичный ключ и создать индекс для имени пользователя.

Чтобы создать индекс, сначала следует получить ссылку на хранилище объектов, а затем вызвать метод `createIndex()`, например:

```
const transaction = db.transaction("users"),
    store = transaction.objectStore("users"),
    index = store.createIndex("username", "username", { unique: true });
```

Первым аргументом метода `createIndex()` является имя индекса, вторым — имя свойства для индексации, третьим — объект `options`, содержащий ключ `unique`. Этот параметр нужно указывать всегда, чтобы было ясно, уникален ли ключ среди всех записей. Поскольку имя пользователя дублироваться не может, этот индекс уникален.

Метод `createIndex()` возвращает экземпляр `IDBIndex`. Этот же экземпляр можно получить, вызвав для хранилища объектов метод `index()`. Например, использовать уже существующий индекс `"username"` можно так:

```
const transaction = db.transaction("users"),
    store = transaction.objectStore("users"),
    index = store.index("username");
```

Индекс во многом похож на хранилище объектов. Вы можете создать для индекса новый курсор методом `openCursor()`, который отличается от одноименного метода хранилища объектов лишь тем, что свойству `result.key` присваивается индексный, а не первичный ключ, например:

```
const transaction = db.transaction("users"),
    store = transaction.objectStore("users"),
    index = store.index("username"),
    request = index.openCursor();
request.onsuccess = (event) => {
    // код, выполняемый при успешном завершении операции
};
```

Для индекса также можно создать специальный курсор, возвращающий для каждой записи только первичный ключ, используя метод `openKeyCursor()`, который принимает те же аргументы, что и метод `openCursor()`. Имейте в виду, что свойство `event.result.key` представляет индексный ключ, а `event.result.value` — первичный ключ, а не всю запись.

```
const transaction = db.transaction("users"),
    store = transaction.objectStore("users"),
    index = store.index("username"),
    request = index.openKeyCursor();

request.onsuccess = (event) => {
    // код, выполняемый при успешном завершении операции
    // event.result.key — индексный ключ
    // event.result.value — первичный ключ
};
```

Вы также можете получить из индекса одно значение, вызвав метод `get()` и передав в него индексный ключ; в результате будет создан новый запрос:

```
const transaction = db.transaction("users"),
    store = transaction.objectStore("users"),
    index = store.index("username"),
    request = index.get("007");

request.onsuccess = (event) => {
    // код, выполняемый при успешном завершении операции
};
request.onfailure = (event) => {
    // обработка сбоя
};
```

Чтобы получить только первичный ключ для указанного индексного ключа, используйте метод `getKey()`. Он также создает новый запрос, но свойство `result.value` при этом содержит первичный ключ, а не всю запись:

```
const transaction = db.transaction("users"),
    store = transaction.objectStore("users"),
    index = store.index("username"),
    request = index.getKey("007");

request.onsuccess = (event) => {
    // код, выполняемый при успешном завершении операции
    // event.result.key — индексный ключ
    // event.result.value — первичный ключ
};
```

В обработчике события `success` в этом примере значением `event.result.value` был бы идентификатор пользователя.

Информацию об индексе можно получить в любой момент с помощью свойств объекта `IDBIndex`:

- `name` — имя индекса;
- `keyPath` — пусть к свойству, переданный в метод `createIndex()`;

- `objectStore` — хранилище объектов, с которым работает данный индекс;
- `unique` — логическое значение, указывающее, уникален ли индексный ключ.

Само хранилище объектов также отслеживает индексы по имени с помощью свойства `indexNames`. Это позволяет легко узнать, какие индексы уже есть у объекта:

```
const transaction = db.transaction("users"),
    store = transaction.objectStore("users"),
    indexNames = store.indexNames
for (let indexName in indexNames) {
    const index = store.index(indexName);
    console.log(`Index name: ${index.name}
                 KeyPath: ${index.keyPath}
                 Unique: ${index.unique}`);
}
```

Этот код перебирает все индексы и выводит информацию о них на консоль.

Индекс можно удалить, вызвав для хранилища объектов метод `deleteIndex()` с именем индекса:

```
const transaction = db.transaction("users"),
    store = transaction.objectStore("users"),
    store.deleteIndex("username");
```

Поскольку удаление индекса не влияет на данные в хранилище объектов, эта операция выполняется без каких-либо функций обратного вызова.

Проблемы параллельного доступа

Хотя IndexedDB является асинхронным API, ему все равно присущи проблемы параллельного доступа. Если одна веб-страница одновременно открыта в двух разных вкладках браузера, какой-то из ее экземпляров может попытаться обновить базу данных, когда второй экземпляр не готов к этому. Проблема связана с заданием новой версии базы данных, так что метод `setVersion()` может быть выполнен, только если базу данных использует лишь одна вкладка браузера.

При открытии базы данных в первый раз важно назначить обработчик события `versionchange`. Эта функция обратного вызова выполняется, когда открывается другая вкладка из того же источника с новой версией базы данных. В ответ на это событие лучше всего немедленно закрыть базу данных, чтобы можно было обновить ее версию, например:

```
let request, database;

request = indexedDB.open("admin", 1);
request.onsuccess = (event) => {
    database = event.target.result;
    database.onversionchange = () => database.close();
};
```

Обработчик события `versionchange` следует назначать после каждого успешного открытия базы данных. Помните, что обработчик события `versionchange` вызывается и для других вкладок.

Если всегда назначать эти обработчики событий, веб-приложение сможет эффективнее обрабатывать проблемы параллельного доступа, связанные с IndexedDB.

Пределы и ограничения

У IndexedDB много тех же ограничений, что и у веб-хранилища. Так, базы данных IndexedDB привязаны к источнику (протокол, домен и порт) страницы, из-за чего информация не может быть общей для нескольких доменов. Это означает, например, что с поддоменами www.wrox.com и p2p.wrox.com используются разные хранилища данных.

Объем хранимых данных на источник также ограничен. В Firefox это ограничение сейчас составляет 50 Мбайт, а в Chrome — 5 Мбайт. Firefox для мобильных устройств ограничивает объем данных значением 5 Мбайт, а при превышении этой квоты запрашивает у пользователя разрешение на сохранение дополнительных данных.

Кроме того, в Firefox запрещен доступ к базам данных IndexedDB для локальных файлов, тогда как в Chrome такого ограничения нет. Таким образом, для запуска примеров из этой книги на локальном компьютере используйте Chrome.

ИТОГИ

Можно сохранять данные в веб-хранилище, используя объекты `sessionStorage` и `localStorage`. Первый из них хранит данные в течение сеанса браузера, по завершении которого данные удаляются. Второй используется для сохранения данных между сеансами.

IndexedDB — это механизм хранения структурированной информации. Она напоминает базу данных SQL-типа, но в IndexedDB данные хранятся не в таблицах, а в хранилищах объектов. Чтобы создать такое хранилище, нужно сначала определить ключ, а уже затем можно добавлять данные. Для запроса конкретных элементов данных из хранилищ объектов используются курсоры, а для ускорения просмотра конкретных свойств можно создавать индексы.

Таким образом, с помощью JavaScript можно хранить на клиентском компьютере значительный объем данных. Будьте, однако, внимательны, чтобы не сохранить по неосторожности конфиденциальную информацию, потому что кеш данных не шифруется.

26

Модули

- Паттерн Модуль
- Импровизированные модульные системы
- Загрузчики модулей до ES6
- Модули в ES6

Загрузки с сайта wrox.com для этой главы

Обратите внимание, что все примеры кода для этой главы доступны на веб-сайте книги по ссылке www.wrox.com/go/projavascript4e на вкладке Downloads.

Написание современного JavaScript по сути гарантирует, что вы будете работать с большими кодовыми базами и использовать сторонние ресурсы. Следствием этого является то, что вы в конечном итоге будете использовать код, разбитый на разные части, и каким-то образом соединять их вместе.

До спецификации модулей в ECMAScript 6 существовала острая необходимость в модульном поведении, даже если браузеры изначально его не поддерживали. ECMAScript никак не поддерживал модули, поэтому библиотеки и кодовые базы, которые хотели использовать модульный паттерн, должны были ловко использовать конструкции JavaScript и лексические функции для «подделки» поведения, подобного модулю.

Поскольку JavaScript является асинхронно загруженным интерпретируемым языком, реализации модулей, которые появились и получили широкое распространение, приняли несколько различных форм. Эти различные формы перевоплотились для достижения разных результатов, но в конечном итоге все они были реализациями канонического паттерна Модуль.

ПАТТЕРН МОДУЛЬ

Разделение кода на независимые части и соединение этих частей вместе может быть надежно реализовано с помощью паттерна Модуль. Основные идеи для этого паттерна просты: разбить логику на части, которые полностью инкапсулированы в остальной части кода, позволяют каждому фрагменту явно определять, какие части самого себя доступны внешним элементам, и явно определять, какие внешние части нужно выполнить. Существуют различные реализации и функции, которые усложняют концепции, но эти фундаментальные идеи являются основой для всех модульных систем в JavaScript.

Идентификаторы модулей

Общей для всех модульных систем является концепция идентификаторов модулей. Модульные системы — это сущности ключ–значение, где каждый модуль имеет идентификатор, который можно использовать для ссылки на него. Этот токен иногда будет строкой в случаях, когда модульная система эмулируется, или может быть реальным путем к файлу модуля в тех случаях, когда модульная система реализована изначально.

Некоторые модульные системы позволяют явно объявлять идентификатор модуля, а некоторые неявно используют имя файла в качестве токена идентификации модуля. В любом случае правильно построенная модульная система не будет иметь конфликтов идентичности модулей, и любой модуль в системе должен иметь возможность ссылаться на другой модуль в системе без двусмысленности.

Как именно идентификатор модуля будет преобразован в фактический модуль, будет зависеть от реализации идентификатора в любой модульной системе. Родные идентификаторы модуля браузера должны указывать путь к реальному файлу JavaScript. В дополнение к путям к файлам NodeJS будет выполнять поиск совпадений модулей в каталоге `node_modules`, а также может сопоставлять идентификатор с каталогом, содержащим `index.js`.

Зависимости модуля

Реальный смысл модульных систем вступает в игру при рассмотрении того, как нужно управлять зависимостями. Модуль, определяющий зависимость, заключает договор с окружающей средой. Локальный модуль объявляет системе модулей список внешних модулей — «зависимостей», — о которых известно, что они существуют и необходимы для правильной работы локального модуля. Модульная система проверяет зависимости и, в свою очередь, гарантирует, что эти модули будут загружены и инициализированы к моменту выполнения локального модуля.

Каждый модуль также связан с некоторым уникальным токеном, который можно использовать для извлечения модуля. Часто это путь к файлу JavaScript, но в некоторых модульных системах это также может быть строка пути пространства имен, объявленная внутри самого модуля.

Загрузка модулей

Концепция «загрузки» модулей вытекает из требований договора о зависимостях. Когда в качестве зависимости указан внешний модуль, локальный модуль ожидает, что при его выполнении зависимости будут готовы и инициализированы.

В контексте браузеров загрузка модуля состоит из нескольких компонентов. Загрузка модуля включает выполнение кода внутри него, но оно не может начаться, пока не будут загружены и выполнены все зависимости. Зависимость модуля, если это код, который еще не был отправлен в браузер, должна быть запрошена и доставлена по сети. После того как полезная нагрузка кода будет получена браузером, браузер должен определить, есть ли у недавно загруженного внешнего модуля свои собственные зависимости, и он будет рекурсивно оценивать эти зависимости и загружать их по очереди, пока все зависимые модули не будут загружены. Только после загрузки всего графа зависимостей первоначальный модуль может запустить выполнение.

Точки входа

Сеть модулей, зависящих друг от друга, должна указывать один модуль в качестве «точки входа», где начинается путь выполнения. Это должно иметь смысл, так как среда выполнения JavaScript выполняется последовательно и однопоточна, поэтому код должен где-то начинаться. Этот модуль точки входа, вероятно, будет иметь зависимости, а некоторые из этих зависимостей, в свою очередь, будут иметь свои зависимости. В результате этого все модули модульного приложения JavaScript будут формировать граф зависимостей.

Зависимости между модулями в приложении могут быть представлены в виде ориентированного графа. Предположим, что граф зависимостей показан на рис. 26.1, представляя воображаемое приложение:

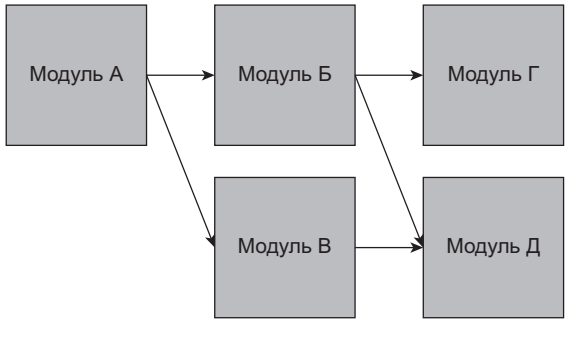

Рис. 26.1

Стрелки представляют поток зависимостей модуля: модуль А зависит от модуля Б и модуля В, модуль Б зависит от модуля Г и модуля Д и т. д. Поскольку модуль

не может быть загружен до тех пор, пока не будут загружены его зависимости, из этого следует, что модуль A, точка входа в это воображаемое приложение, должен выполняться только после загрузки остальной части приложения.

В JavaScript концепция «загрузки» может принимать различные формы. Поскольку модули реализованы в виде файла, содержащего код JavaScript, который будет выполняться немедленно, можно запросить отдельные сценарии в порядке, который будет соответствовать графу зависимостей. Для предыдущего приложения следующий порядок сценариев удовлетворял бы графу зависимостей:

```
<script src="moduleE.js"></script>
<script src="moduleD.js"></script>
<script src="moduleC.js"></script>
<script src="moduleB.js"></script>
<script src="moduleA.js"></script>
```

Загрузка модуля является «блокирующей», что означает, что дальнейшее выполнение не может продолжаться до завершения операции. Каждый модуль постепенно загружается после того, как его сценарий загружен в браузер, и все его зависимости уже загружены и инициализированы. Однако эта стратегия имеет ряд последствий для производительности и сложности: последовательная загрузка пяти файлов JavaScript для выполнения работы одного приложения не идеальна, а управление правильным порядком загрузки — непростая задача, которую можно выполнить вручную.

Асинхронные зависимости

Поскольку JavaScript является асинхронным языком, это также может быть полезно для загрузки модулей по требованию, позволяя коду JavaScript указывать системе модулей загрузить новый модуль и предоставлять модуль для обратного вызова, как только он будет готов. Псевдокод для этого может выглядеть следующим образом:

```
// определение модуля A
load('moduleB').then(function(moduleB) {
    moduleB.doStuff();
});
```

Код модуля A использует токен `moduleB`, чтобы указать модульной системе загрузить модуль Б и вызвать обратный вызов с модулем Б, предоставленным в качестве параметра. Модуль Б, возможно, уже был загружен, или его, возможно, придется заново запрашивать и инициализировать, но код для этого не имеет значения — эти обязанности делегируются загрузчику модуля.

Если бы вам пришлось переделывать предыдущее приложение, чтобы использовать только программную загрузку модуля, нужно было бы использовать только один тег `<script>` для загрузки модуля A, а модуль A запрашивал бы файлы модуля по мере необходимости, не создавая упорядоченный список требуемых зависимостей. Это имеет ряд преимуществ, одним из которых является производительность,

поскольку при загрузке страницы требуется синхронная загрузка только одного файла.

Также было бы возможно разделить эти сценарии, применить атрибут `defer` или `async` к тегам `<script>` и добавить логику, которая может различать, когда асинхронный сценарий загружается и инициализируется. Такое поведение будет эмулировать то, что реализовано в спецификации модуля ES6, которая будет рассмотрена позже в этой главе.

Программные зависимости

Некоторые модульные системы требуют указания всех зависимостей в начале модуля, но некоторые позволят динамически добавлять зависимости внутри структуры программы. Это отличается от обычных зависимостей, перечисленных в начале модуля, которые необходимо загрузить до того, как модуль может начать выполняться.

Ниже приведен пример загрузки программных зависимостей.

```
if (loadCondition) {
    require('./moduleA');
}
```

Внутри этого модуля она определяется только во время выполнения, если загружен `moduleA`. Возможно, загрузка `moduleA` блокируется или может привести к выполнению и продолжится только после загрузки модуля. В любом случае выполнение внутри этого модуля не может продолжаться до тех пор, пока не будет загружен `moduleA`, поскольку предполагается, что наличие `moduleA` является критическим для последующего поведения модуля.

Программные зависимости допускают более сложное использование зависимостей, но это затрудняет статический анализ модуля.

Статический анализ

Встроенный в модули и добавленный в браузер JavaScript часто подвергается статическому анализу, где инструменты проверяют структуру кода и выясняют, как он будет себя вести без выполнения программы. Модульная система, которая удобна для статического анализа, позволит системам связывания модулей легче находить способы объединения кода в меньшее количество файлов. Он также предложит возможность выполнять интеллектуальное автозаполнение в умном редакторе.

Более сложное поведение модуля, такое как программные зависимости, усложнит процесс статического анализа. Различные модульные системы и загрузчики модулей будут предлагать разные уровни сложности. Что касается зависимостей модуля, то из-за дополнительной сложности инструменту будет сложнее предсказать, какие именно зависимости понадобятся модулю при выполнении.

Циклические зависимости

Почти невозможно создать приложение JavaScript без циклов зависимостей, и поэтому все модульные системы, включая CommonJS, AMD и ES6, поддерживают циклические зависимости. В приложении с циклами зависимостей порядок загрузки модулей может отличаться от ожидаемого. Однако, если вы правильно структурировали свои модули так, чтобы не было побочных эффектов, порядок загрузки не должен наносить ущерб общему приложению.

В следующем модуле (который использует независимый от модуля псевдокод) любой из модулей может использоваться в качестве модуля точки входа, даже если в графе зависимостей есть циклы:

```
require('./moduleD');
require('./moduleB');

console.log('moduleA');
require('./moduleA');
require('./moduleC');

console.log('moduleB');
require('./moduleB');
require('./moduleD');

console.log('moduleC');

require('./moduleA');
require('./moduleC');

console.log('moduleD');
```

Изменение модуля, используемого в качестве основного модуля, изменит порядок загрузки зависимостей. Если `moduleA` должен был быть загружен первым, в консоли было бы выведено следующее, что указывает на абсолютный порядок завершения загрузки модуля:

```
moduleB
moduleC
moduleD
moduleA
```

Порядок загрузки можно визуализировать с помощью графика зависимостей на рис. 26.2, где загрузчик будет выполнять загрузку зависимостей по направлению вглубь.

Если вместо этого модуль C должен быть загружен первым, будет выведено следующее, что указывает на абсолютный порядок загрузок модуля:

```
moduleD
moduleA
moduleB
moduleC
```

Порядок загрузки можно визуализировать с помощью графика зависимостей на рис. 26.3, где загрузчик будет выполнять загрузку зависимостей по направлению вглубь:

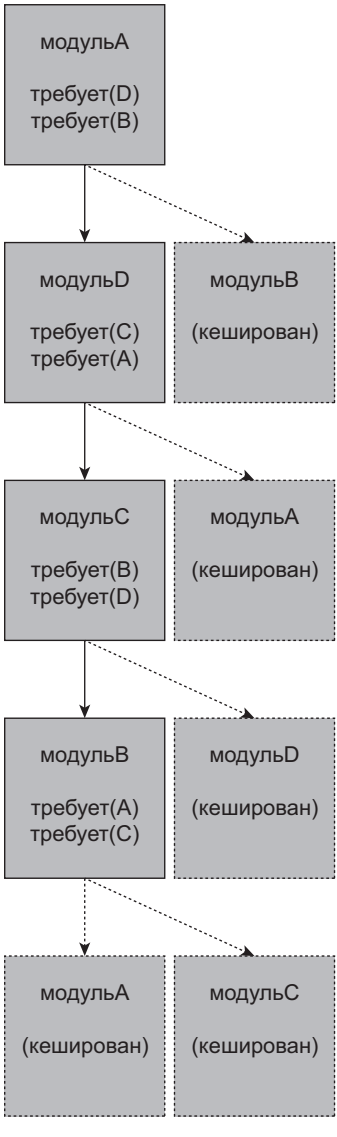

Рис. 26.2

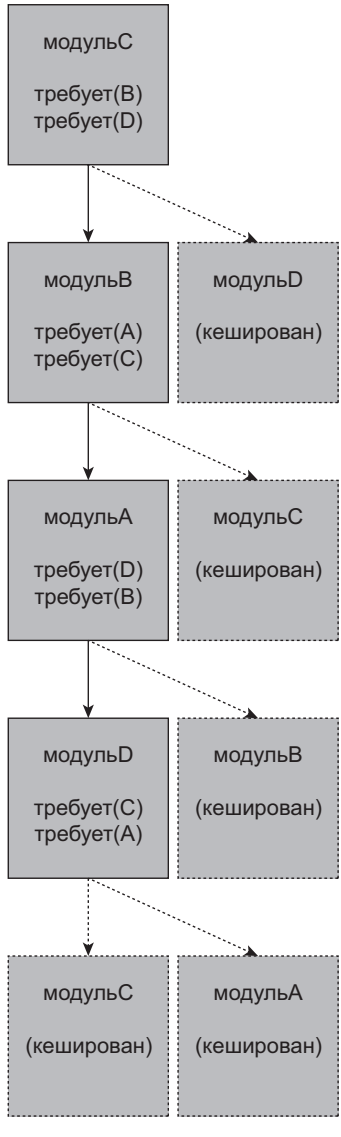

Рис. 26.3

ИМПРОВИЗИРОВАННЫЕ МОДУЛЬНЫЕ СИСТЕМЫ

Чтобы предложить инкапсуляцию, требуемую паттерном Модуль, модули до ES6 иногда использовали область действия функции и немедленно вызывали функции-выражения (IIFE), чтобы обернуть определение модуля в анонимное замыкание. Определение модуля выполняется немедленно, как показано здесь:

```
(function() {
    // закрытый код модуля Foo
    console.log('bar');
})();

// 'bar'
```

Когда возвращаемое значение этого модуля было присвоено переменной, это фактически создало пространство имен для модуля:

```
var Foo = (function() {
    console.log('bar');
})();

'bar'
```

Чтобы предоставить открытый API, модуль IIFE должен вернуть объект, свойства которого будут открытыми членами внутри пространства имен модуля:

```
var Foo = (function() {
    return {
        bar: 'baz',
        baz: function() {
            console.log(this.bar);
        }
    };
})();

console.log(Foo.bar);    // 'baz'
Foo.baz();    // 'baz'
```

Шаблон, похожий на предыдущий, называемый «Шаблон модуля раскрытия», возвращает только объект, свойства которого являются ссылками на закрытые данные и элементы:

```
var Foo = (function() {
    var bar = 'baz';
    var baz = function() {
        console.log(bar);
    };

    return {
        bar: bar,
        baz: baz
    };
})();

console.log(Foo.bar);    // 'baz'
Foo.baz();    // 'baz'
```

Также возможно определить модули внутри модулей, что может быть полезно для целей вложения пространства имен:

```
var Foo = (function() {
    return {
        bar: 'baz'
```

```
    };
})();

Foo.baz = (function() {
    return {
        qux: function() {
            console.log('baz');
        }
    };
})();

console.log(Foo.bar);    // 'baz'
Foo.baz.qux();    // 'baz'
```

Чтобы модуль правильно использовал внешние значения, их можно передать в качестве параметров в IIFE:

```
var globalBar = 'baz';

var Foo = (function(bar) {
    return {
        bar: bar,
        baz: function() {
        console.log(bar);
    }
 };
})(globalBar);

console.log(Foo.bar);    // 'baz'
Foo.baz();    // 'baz'
```

Поскольку реализации модуля здесь просто создают экземпляр объекта JavaScript, вполне возможно дополнить модуль после его определения:

```
// Оригинальный Foo
var Foo = (function(bar) {
    var bar = 'baz';

    return {
        bar: bar
    };
})();

// Дополненный Foo
var Foo = (function(FooModule) {
    FooModule.baz = function() {
        console.log(FooModule.bar);
    }

return FooModule;
})(Foo);

console.log(Foo.bar);    // 'baz'
Foo.baz();    // 'baz'
```

Также может быть полезно настроить расширение модуля для дополнения независимо от того, присутствует модуль или нет:

```
// Дополнение Foo для добавления метода alert
var Foo = (function(FooModule) {
    FooModule.baz = function() {
        console.log(FooModule.bar);
    }

    return FooModule;
})(Foo || {});

// Дополнение Foo для добавления данных
var Foo = (function(FooModule) {
    FooModule.bar = 'baz';

    return FooModule;
})(Foo || {});

console.log(Foo.bar);    // 'baz'
Foo.baz();    // 'baz'
```

Как вы можете подозревать, проектирование вашей собственной модульной системы — это интересное упражнение, но его не рекомендуется применять в реальных проектах, поскольку результат слишком хрупок. Предыдущие примеры не имеют хорошего способа программной загрузки зависимостей, кроме использования порочного `eval`. Зависимости должны управляться и упорядочиваться вручную. Чрезвычайно сложно добавить асинхронную загрузку и циклические зависимости. Наконец, выполнение статического анализа на такой системе было бы довольно сложно.

ЗАГРУЗЧИКИ МОДУЛЕЙ ДО ES6

До появления собственной поддержки модулей ES6 кодовые базы JavaScript, использующие модули, в основном хотели использовать языковую функцию, которая не была доступна по умолчанию. Следовательно, они должны быть написаны в синтаксисе модуля, который соответствует определенной спецификации, а отдельные инструменты модуля будут служить для устранения разрыва между синтаксисом модуля и средой выполнения JavaScript. Синтаксис модуля и проектирование мостов принимают различные формы, обычно это либо дополнительная библиотека в браузере, либо предварительная обработка во время сборки.

CommonJS

Спецификация CommonJS описывает соглашение для определения модуля, которое использует синхронные декларативные зависимости. Эта спецификация в первую очередь предназначена для организации модулей на сервере, но ее также можно использовать для определения зависимостей для модулей, которые будут использоваться в браузере. Синтаксис модуля CommonJS не будет естественным образом работать в браузере.

ПРИМЕЧАНИЕ Часто NodeJS и CommonJS будут описаны как использующие один и тот же стиль модульных систем, и это не совсем так. NodeJS использует слегка модифицированную версию CommonJS, которая подходит для серверной среды, поскольку это не требует решения проблемы задержки в сети. Для согласованности в этом разделе будет использоваться синтаксис определения модуля со вкусом NodeJS.

Определение модуля в CommonJS обозначит его зависимости с помощью `require()`, и определит его публичный API с помощью объекта `exports`. Простое определение модуля может выглядеть следующим образом:

```
var moduleB = require('./moduleB');

module.exports = {
    stuff: moduleB.doStuff();
};
```

`moduleA` определяет свою зависимость от `moduleB`, используя относительный путь к определению модуля. То, что считается «определением модуля», и то, как строка ссылается на этот модуль, полностью зависит от реализации модульной системы. Например, в NodeJS идентификатор модуля может указывать на один файл или каталог с файлом `index.js`.

Запрашивание модуля загрузит его, и, хотя назначение модуля в переменную является очень распространенным явлением, оно не потребуется. Вызов `require()` означает, что модуль будет загружаться одинаково.

```
console.log('moduleA');
require('./moduleA');    // "moduleA"
```

Модули всегда являются одиночными, независимо от того, сколько раз на модуль ссылаются внутри `require()`. В следующем примере `moduleA` будет напечатан только один раз, потому что он загружается только один раз, даже если запрашивается несколько раз.

```
console.log('moduleA');
var a1 = require('./moduleA');
var a2 = require('./moduleA');

console.log(a1 === a2);    // true
```

Модули кешируются после первой загрузки; последующие попытки загрузить модуль вернут кешированный модуль. Порядок загрузки модуля определяется графом зависимостей.

```
console.log('moduleA');
require('./moduleA');
require('./moduleB');    // "moduleA"
require('./moduleA');
```

В CommonJS загрузка модуля — это синхронная операция, выполняемая модульной системой, поэтому `require()` может быть программно вызван как внутри модуля, так и условно:

```
console.log('moduleA');
if (loadCondition) {
    require('./moduleA');
}
```

Здесь, `moduleA` будет загружаться, только если `loadCondition` будет вычислен как `true`. Загрузка является синхронной, поэтому любой код, который предшествует блоку `if()`, будет выполнен до загрузки `moduleA`, а любой код, следующий за блоком `if()`, будет выполнен после загрузки `moduleA`. Применяются все те же правила порядка загрузки, поэтому, если `moduleA` был загружен ранее в другом месте в графе, этот условный `require()` будет служить только для разрешения использовать пространство имен `moduleA`.

В этих примерах модульная система реализована внутри NodeJS, поэтому `./moduleB` — это относительный путь к целевому модулю в том же каталоге, что и этот модуль. NodeJS будет использовать строку идентификатора модуля в вызове `require()` для разрешения зависимости от ссылки на модуль. NodeJS может использовать абсолютные или относительные пути к модулям или идентификаторы для зависимостей, установленных в каталоге `node_modules`. Эти детали не относятся к предмету этой книги, но важно знать, что строковая ссылка на модуль может быть по-разному реализована в разных реализациях CommonJS. Однако общим для всех реализаций в стиле CommonJS является то, что модули не будут указывать свой идентификатор; это происходит из их расположения в файловой иерархии модуля.

Путь к определению модуля может ссылаться на каталог или один файл JavaScript — в любом случае, этот локальный модуль не имеет отношения к реализации модуля, и модуль Б загружается в локальную переменную. Модуль А, в свою очередь, определяет свой открытый интерфейс, свойство `foo`, в объекте `module.exports`.

Если другой модуль хочет использовать этот интерфейс, он может импортировать модуль следующим образом:

```
var moduleA = require('./moduleA');

console.log(moduleA.stuff);
```

Обратите внимание, что этот модуль ничего не экспортирует. Даже если у него нет открытого интерфейса, если модуль требуется в приложении, он все равно будет выполнять тело модуля под нагрузкой.

Объект экспорта чрезвычайно гибок и может принимать несколько форм. Если нужно экспортировать только одну сущность, вы можете выполнить прямое назначение для `module.exports`:

```
module.exports = 'foo';
```

Таким образом, весь интерфейс модуля является строкой, которую можно использовать следующим образом:

```
var moduleA = require('./moduleB');

console.log(moduleB);    // 'foo'
```

Также очень распространено объединение нескольких значений в экспорт, что может быть сделано либо с помощью литерала объекта, либо одноразового назначения свойства:

```
// Эквивалентны:

module.exports = {
    a: 'A',
    b: 'B'
};

module.exports.a = 'A';
module.exports.b = 'B';
```

Одним из основных применений модулей является размещение определений классов (показано здесь с использованием определения класса в стиле ES6, но определение класса в стиле ES5 также совместимо):

```
class A {}

module.exports = A;

var A = require('./moduleA');
var a = new A();
```

Также можно назначить экземпляр класса в качестве экспортируемого значения:

```
class A {}

module.exports = new A();
```

Кроме того, CommonJS поддерживает программные зависимости:

```
if (condition) {
    var A = require('./moduleA');
}
```

CommonJS использует несколько глобальных переменных, таких как `require` и `module.exports` для работы. Для того чтобы модули CommonJS могли использоваться в браузере, необходим некий мост между его сторонним модульным синтаксисом. Также должен быть какой-то барьер между кодом уровня модуля и выполнением браузера, так как код CommonJS, выполняемый без инкапсуляции, будет объявлять глобальные переменные в браузере — поведение, нежелательное в модульном паттерне.

Распространенное решение состоит в том, чтобы заранее объединить файлы модулей, преобразовать глобальные переменные в собственные конструкции JavaScript, инкапсулировать код модуля в замыканиях функций и обслуживать только один

файл. Понимание графа зависимостей необходимо для объединения модулей в правильном порядке.

Асинхронное определение модулей

Принимая во внимание, что CommonJS нацелен на серверную модель исполнения — где нет штрафов за загрузку всего в память сразу, — система асинхронного определения модулей (Asynchronous Module Definition, AMD) специально нацелена на модель исполнения браузера, где *есть* штрафы от увеличенной задержки сети. Общая стратегия AMD заключается в том, чтобы модули объявляли свои зависимости, а модульная система, работающая в браузере, извлекала их по требованию и запускала модуль, который зависит от них, после их загрузки.

Ядром реализации модуля AMD является функциональная оболочка вокруг определения модуля. Это предотвращает объявление глобальных переменных и позволяет библиотеке загрузчика контролировать, когда именно загружать модуль. Оболочка функции также обеспечивает превосходную переносимость кода модуля, поскольку весь код модуля внутри обертки функции использует собственные конструкции JavaScript. Эта функция-обертка является аргументом для `define`, которая определяется реализацией библиотеки загрузчика AMD.

Модуль AMD может указать свои зависимости с помощью строковых идентификаторов, а загрузчик AMD вызовет функцию фабрики модулей после загрузки всех зависимых модулей. В отличие от CommonJS, AMD позволяет дополнительно указать идентификатор строки для модуля.

```
// Определение для модуля с идентификатором 'moduleA'. moduleA зависит от moduleB,
// который будет загружен асинхронно.
define('moduleA', ['moduleB'], function(moduleB) {
    return {
        stuff: moduleB.doStuff();
    };
});
```

AMD также поддерживает объекты `require` и `exports`, которые позволяют создавать модули CommonJSstyle внутри функции фабрики модулей AMD. Они запрашиваются так же, как и модули, но загрузчик AMD распознает их как собственные конструкции AMD, а не как определения модулей:

```
define('moduleA', ['require', 'exports'], function(require, exports) {
    var moduleB = require('moduleB');

    exports.stuff = moduleB.doStuff();
});
```

Программные зависимости поддерживаются с помощью этого стиля:

```
define('moduleA', ['require'], function(require) {
    if (condition) {
        var moduleB = require('moduleB');
    }
});
```

Универсальное определение модулей

В попытке объединить экосистемы CommonJS и AMD было введено соглашение об универсальном определении модулей (Universal Module Definition, UMD) для создания кода модуля, который может использоваться обеими системами. По сути, паттерн определяет модули таким способом, который определяет, какая модульная система используется при запуске, настраивает ее соответствующим образом и оборачивает все это в немедленно вызванную функцию-выражение. Это несовершенная комбинация, но при объединении двух экосистем она подходит для неожиданно большого числа сценариев.

Пример модуля с одной зависимостью (на основе хранилища UMD на GitHub) выглядит следующим образом:

```
(function (root, factory) {
    if (typeof define === 'function' && define.amd) {
        // AMD. Регистрация в качестве анонимного модуля.
        define(['moduleB'], factory);
    } else if (typeof module === 'object' && module.exports) {
        // Node. Работает не со строгим режимом CommonJS, а
        // только со средами, похожими на CommonJS, котрые поддерживают
        // module.exports like подобно Node.
        module.exports = factory(require(' moduleB '));
    } else {
        // Глобальные переменные браузера (корневой модуль — window)
        root.returnExports = factory(root. moduleB);
 }
}(this, function (moduleB) {
    // использование moduleB каким-то образом.

    // Только возврат значения для определения экспорта модуля.
    // Этот пример возвращает объект, но модуль
    // может вернуть функцию как возвращаемое значение.
    return {};
}));
```

Существуют варианты этого паттерна, которые позволяют поддерживать строгие глобальные переменные CommonJS и браузера. Никогда не следует ожидать точного создания этой оболочки вручную — она должна автоматически генерироваться инструментом сборки. Ваша цель — заботиться о содержании модулей, а не о паттерне, который их всех соединяет.

Устаревший модуль загрузчика

В конечном итоге паттерны, показанные в этом разделе, будут становиться все более устаревшими по мере расширения поддержки спецификации модулей в ECMAScript 6. Таким образом, все еще весьма полезно знать, из чего выросла спецификация модулей ES6, чтобы узнать, почему были выбраны проектные решения. Интенсивный конфликт между CommonJS и AMD привел к появлению спецификации модуля ECMAScript 6, которая нам сейчас нравится.

МОДУЛИ В ES6

Одним из наиболее значительных введений в ECMAScript 6 была спецификация модулей. Спецификация во многих отношениях проще, чем предшествующие загрузчики модулей, а встроенная поддержка браузера означает, что библиотеки загрузчиков и другая предварительная обработка не требуются. Во многом модульная система ES6 объединяет лучшие характеристики AMD и CommonJS в единую спецификацию.

Маркировка и определение модулей

Модули ECMAScript 6 существуют как монолитный кусок JavaScript. Тег скрипта с `type="module"` будет сигнализировать браузеру, что связанный код должен быть выполнен как модуль, а не как традиционный сценарий. Модули могут быть определены внутри или во внешнем файле:

```
<script type="module">
    // код модуля
</script>

<script type="module" src="path/to/myModule.js"></script>
```

Даже если они обрабатываются не так, как обычно загружаемый файл JavaScript, файлы модуля JavaScript не имеют специального типа содержимого.

В отличие от своих традиционных аналогов сценариев, все модули будут выполняться в том же порядке, в котором будет выполняться `<script defer>`. Загрузка файла модуля начинается сразу после анализа тега `<script type="module">`, но выполнение задерживается до тех пор, пока документ не будет полностью проанализирован. Это относится как к встроенным модулям, так и к модулям, определенным во внешних файлах. Порядок, в котором код `<script type="module">` появляется на странице, является порядком, в котором он будет выполняться. Как и в случае с `<script defer>`, изменение расположения тегов модуля — в `<head>` или `<body>` — будет контролироваться только при загрузке файлов, а не модулей.

Ниже приведен порядок выполнения встроенного модуля:

```
<!-- Выполняется 2-м -->
<script type="module"></script>

<!-- Выполняется 3-м -->
<script type="module"></script>

<!-- Выполняется 1-м -->
<script></script>
```

Кроме того, этот код может быть переработан с помощью определения внешнего модуля JS:

```
<!-- Выполняется 2-м -->
<script type="module" src="module.js"></script>

<!-- Выполняется 3-м -->
<script type="module" src="module.js"></script>
```

```
<!-- Выполняется 1-м -->
<script><script>
```

Также можно добавить атрибут `async` в теги модуля. Эффект этого двоякий: порядок выполнения модуля больше не привязан к порядку тегов скрипта на странице, и модуль не будет ждать окончания анализа документа перед началом выполнения. Вводной модуль все еще должен ждать загрузки своих зависимостей.

Модули ES6, связанные с тегом `<script type="module">`, считаются вводными для графа модулей. Нет ограничений относительно того, сколько вводных модулей может быть на странице, и нет ограничений на перекрытие модулей. Независимо от того, сколько раз модуль загружается на страницу, независимо от того, как происходит эта загрузка, он загрузится только один раз, как показано здесь:

```
<!-- moduleA загрузится только один раз на этой странице -->

<script type="module">
    import './moduleA.js'
<script>

<script type="module">
    import './moduleA.js'
<script>

<script type="module" src="./moduleA.js"></script>
<script type="module" src="./moduleA.js"></script>
```

Модули, определенные как встроенные, не могут быть загружены в другие модули с помощью `import`. С его помощью могут быть загружены только модули из внешнего файла. Поэтому встроенные модули полезны только в качестве модуля точки входа.

Загрузка модулей

Модули ECMAScript 6 уникальны своей способностью загружаться как из браузера, так и в сочетании со сторонними загрузчиками и инструментами сборки. Некоторые браузеры по-прежнему не поддерживают модули ES6, поэтому могут потребоваться сторонние инструменты. Во многих случаях предпочтительнее использовать сторонние инструменты.

Браузер, который предлагает полную поддержку модуля ECMAScript 6, сможет загружать весь граф зависимостей из модуля верхнего уровня, и он будет делать это асинхронно. Браузер интерпретирует вводной модуль, идентифицирует его зависимости и отправляет запросы для зависимых модулей. Когда эти файлы возвращаются по сети, браузер проанализирует их содержимое, определит их зависимости и отправит больше запросов на зависимости второго порядка, если они еще не загружены.

Этот рекурсивный асинхронный процесс будет продолжаться до тех пор, пока не будет разрешен весь граф зависимостей приложения. Как только зависимости разрешены, приложение может начать загружаться формально.

Этот процесс очень похож на стиль загрузки модулей в AMD. Файлы модулей загружаются по требованию, и последующие раунды запросов к файлам модулей задерживаются из-за задержки в сети каждого файла модуля зависимостей. То есть если вводной `moduleA` зависит от `moduleB`, а `moduleB` зависит от `moduleC`, браузер не будет знать, отправлять ли запрос C, пока запрос B не будет выполнен первым. Этот стиль загрузки эффективен и не требует внешних инструментов, но загрузка большого приложения с графом с глубокими зависимостями может занять слишком много времени.

Модульное поведение

Модули ECMAScript 6 позаимствовали многие из лучших возможностей предшественников CommonJS и AMD. Вот пара из них:

- Код модуля выполняется только после загрузки.
- Модуль будет загружен только один раз.
- Модули являются одиночками.
- Модули могут определять открытый интерфейс, с которым другие модули могут наблюдать и взаимодействовать.
- Модули могут запросить загрузку других модулей.
- Поддерживаются циклические зависимости.

Модульная система ES6 также вводит новое поведение:

- Модули ES6 по умолчанию выполняются в строгом режиме.
- Модули ES6 не разделяют глобальное пространство имен.
- Значение параметра `this` на верхнем уровне модуля — `undefined` (в отличие от `window` в случае обычных сценариев).
- Объявления `var` не будут добавлены к объекту `window`.
- Модули ES6 загружаются и выполняются асинхронно.

Описанное здесь поведение, характеризующее модуль ECMAScript 6, обусловлено условием времени выполнения браузера, когда он знает, что определенный файл является модулем. Файл JavaScript обозначается как модуль либо когда он связан с `<script type="module">`, либо когда он загружается с помощью оператора `import`.

Экспортирование модулей

Публичная система экспорта для модулей ES6 очень похожа на CommonJS. Ключевое слово `export` используется для управления тем, какие части модуля видны внешним модулям. Существует два типа экспорта в модули ES6: именованные экспорты и экспорт по умолчанию. Различные типы экспорта означают, что они импортируются по-разному — это рассматривается в следующем разделе.

Ключевое слово `export` используется для объявления значения как именованного экспорта. Экспорт должен происходить на верхнем уровне модуля; они не могут быть вложены в блоки:

```
// Допустимо
export ...

// Недопустимо
if (condition) {
    export ...
}
```

Экспорт значения не оказывает прямого влияния на выполнение JavaScript внутри модуля, поэтому нет ограничений по расположению оператора `export` относительно того, что экспортируется, или в каком порядке ключевое слово `export` должно появляться в модуле. `export` может даже предшествовать объявлению значения, которое он экспортирует:

```
// Допустимо
const foo = 'foo';
export { foo };

// Допустимо
export const foo = 'foo';

// Допустимо, но лучше этого избегать
export { foo };
const foo = 'foo';
```

Именованный экспорт ведет себя так, как будто модуль является контейнером для экспортируемых значений. Встроенный именованный экспорт, как следует из названия, может выполняться в той же строке, что и объявление переменной. В следующем примере объявление переменной связано со встроенным экспортом. Внешний модуль может импортировать этот модуль, и значение `foo` будет доступно внутри него как свойство этого модуля:

```
export const foo = 'foo';
```

Объявление не должно происходить в той же строке, что и экспорт; можно выполнить объявление и экспортировать идентификатор в другом месте в модуле внутри *предложения экспорта*:

```
const foo = 'foo';
export { foo };
```

Также возможно предоставить псевдоним при экспорте. Псевдоним должен встречаться в синтаксисе скобки предложения экспорта; следовательно, объявление значения, его экспорт и предоставление псевдонима не могут быть представлены в одной строке. В следующем примере внешний модуль получит доступ к этому значению, импортировав этот модуль и используя экспорт `myFoo`:

```
const foo = 'foo';
export { foo as myFoo };
```

Поскольку именованные экспорты ES6 позволяют рассматривать модуль как контейнер, можно объявить несколько именованных экспортов в одном модуле. Значения могут быть объявлены внутри оператора экспорта, или они могут быть объявлены до указания его как экспорта:

```
export const foo = 'foo';
export const bar = 'bar';
export const baz = 'baz';
```

Поскольку экспорт нескольких значений является обычным явлением, поддерживается группировка объявлений экспорта, а также замена псевдонимом некоторых или всех этих экспортов:

```
const foo = 'foo';
const bar = 'bar';
const baz = 'baz';
export { foo, bar as myBar, baz };
```

Экспорт по умолчанию ведет себя так, как будто модуль является той же сущностью, что и экспортированное значение. Модификатор ключевого слова `default` используется для объявления значения в качестве экспорта по умолчанию — может существовать только один экспорт по умолчанию. Попытка указать дубликаты экспорта по умолчанию приведет к ошибке `SyntaxError`.

В следующем примере внешний модуль может импортировать этот модуль, а сам модуль будет иметь значение `foo`:

```
const foo = 'foo';
export default foo;
```

С другой стороны, модульная система ES6 распознает ключевое слово `default`, если оно предоставляется в качестве псевдонима, и применит экспорт по умолчанию к значению, даже если оно использует именованный синтаксис экспорта:

```
const foo = 'foo';

// Поведение, идентичное "export default foo;"
export { foo as default };
```

Поскольку нет несовместимости между именованным экспортом и экспортом по умолчанию, ES6 позволяет использовать оба в одном и том же модуле:

```
const foo = 'foo';
const bar = 'bar';

export { bar };
export default foo;
```

Два оператора `export` могут быть объединены в одну строку:

```
const foo = 'foo';
const bar = 'bar';

export { foo as default, bar };
```

Спецификация ES6 ограничивает то, что можно и нельзя делать в различных формах оператора экспорта. Некоторые формы допускают объявление и присваивание, некоторые допускают только выражения, а некоторые — только простые идентификаторы. Обратите внимание, что одни формы используют точки с запятой, а другие нет.

```
// Именованные встроенные экспорты
export const baz = 'baz';
export const foo = 'foo', bar = 'bar';
export function foo() {}
export function* foo() {}
export class Foo {}

// Именованные предложения экспорта
export { foo };
export { foo, bar };
export { foo as myFoo, bar };

// Экспорты по умолчанию
export default 'foo';
export default 123;
export default /[a-z]*/;
export default { foo: 'foo' };
export { foo, bar as default };
export default foo
export default function() {}
export default function foo() {}
export default function*() {}
export default class {}

// Различные недопустимые формы, которые приведут к ошибкам:

// Объявления переменных не могут появляться внутри встроенных
// экспортов по умолчанию
export default const foo = 'bar';

// Только идентификаторы могут использоваться в предложениях экспорта
export { 123 as foo }'

// Замена псевдонимами может использоваться только в предложениях экспорта
export const foo = 'foo' as myFoo;
```

ПРИМЕЧАНИЕ Правила того, что может и не может отображаться в одной строке с ключевым словом `export`, могут быть трудны для запоминания. Как правило, рекомендуется выполнять объявления и присваивания отдельно, а затем экспортировать идентификатор. Это позволяет легко следовать правильному синтаксису экспорта, а также сохранять операторы `export` сгруппированными.

Импорт модулей

Модули могут использовать экспорт из других модулей, используя ключевое слово `import`. Как и `export`, `import` должен отображаться на верхнем уровне модуля:

```
// Допустимо
Import ...

// Недопустимо
if (condition) {
    import ...
}
```

Операторы `import` выводятся наверх модуля. Следовательно, как и ключевое слово экспорта, `export`, в котором отображаются операторы `import` относительно использования импортированных значений, не имеет значения. Тем не менее рекомендуется сохранять импорт в верхней части модуля.

```
// Допустимо
import { foo } from './fooModule.js';
console.log(foo);    // 'foo'

// Допустимо, но лучше этого избегать
console.log(foo);    // 'foo'
import { foo } from './fooModule.js';
```

Идентификатор модуля может быть либо относительным путем к этому файлу модуля из текущего модуля, либо абсолютным путем к этому файлу модуля из базового пути. Это должна быть простая строка; идентификатор не может быть вычислен динамически, например путем объединения строк.

Если модули изначально загружаются в браузер по пути, указанному в их идентификаторе, для ссылки на правильный файл требуется расширение `.js`. Однако, если модули ES6 собираются или интерпретируются с помощью средства сборки или стороннего загрузчика модулей, вам может не потребоваться включать расширение файла модуля в его идентификатор.

```
// Разрешается к /components/bar.js
import ... from './bar.js';

// Разрешается к /bar.js
import ... from '../bar.js';

// Разрешается к /bar.js
import ... from '/bar.js';
```

Модули не обязательно импортировать через экспортированных членов. Если вам не нужны определенные экспортированные привязки из модуля, но вы все еще загружаете его для загрузки и выполнения модуля для его побочных эффектов, можно загрузить его только с его путем:

```
import './foo.js';
```

Импорты обрабатываются как доступные только для чтения представления модуля, фактически так же, как объявленные переменные `const`. При выполнении массового импорта с использованием * коллекция псевдонимов именованных экспортов ведет себя так, как если бы она была обработана с помощью `Object.freeze()`. Прямое манипулирование экспортируемыми значениями невозможно, хотя все еще можно

изменить свойства экспортируемого объекта. Добавление или удаление экспортированных свойств экспортированной коллекции также запрещено. Мутация экспортируемых значений должна происходить с использованием экспортированных методов, которые имеют доступ к внутренним переменным и свойствам.

```
import foo, * as Foo './foo.js';

foo = 'foo';        // Ошибка

Foo.foo = 'foo';    // Ошибка

foo.bar = 'bar';    // Допустимо
```

Различие между именованным экспортом и экспортом по умолчанию отражается в способе их импорта. Именованные экспорты могут быть получены оптом без указания их точного идентификатора с помощью * и предоставления идентификатора для коллекции экспорта:

```
const foo = 'foo', bar = 'bar', baz = 'baz';
export { foo, bar, baz }
import * as Foo from './foo.js';

console.log(Foo.foo);    // foo
console.log(Foo.bar);    // bar
console.log(Foo.baz);    // baz
```

Для выполнения явного импорта идентификаторы могут быть помещены в *предложение импорта*. Использование предложения импорта также позволяет указать псевдонимы для импорта:

```
import { foo, bar, baz as myBaz } from './foo.js';

console.log(foo);      // foo
console.log(bar);      // bar
console.log(myBaz);    // baz
```

Экспорты по умолчанию ведут себя так, как будто целью модуля является экспортированное значение. Их можно импортировать, используя ключевое слово `default` и предоставляя псевдоним; альтернативно они могут быть импортированы без использования фигурных скобок, а указанный идентификатор фактически будет являться псевдонимом для экспорта по умолчанию:

```
// Эквивалентны
import { default as foo } from './foo.js';
import foo from './foo.js';
```

Если модуль экспортирует как именованные экспорты, так и экспорт по умолчанию, их можно извлечь в одном и том же операторе импорта. Это извлечение может быть выполнено путем перечисления определенных экспортов или использования *:

```
import foo, { bar, baz } from './foo.js';

import { default as foo, bar, baz } from './foo.js';

import foo, * as Foo from './foo.js';
```

ПРИМЕЧАНИЕ В настоящее время на этапе 3 существует предложение для динамического импорта модулей: https://tc39.github.io/proposal-dynamic-import/.

Сквозной экспорт модулей

Импортированные значения могут быть переданы напрямую в экспорт. Можно также конвертировать экспорты по умолчанию в именованные, и наоборот. Если нужно добавить все именованные экспорты из одного модуля в другой, это можно сделать с помощью *:

```
export * from './foo.js';
```

Все именованные экспорты в `foo.js` будут доступны при импорте `bar.js`. Этот синтаксис будет игнорировать значение по умолчанию для `foo.js`, если оно есть; он также требует осторожности при конфликтах имен экспорта. Если `foo.js` экспортирует `baz` и `bar.js` также экспортирует `baz`, конечное значение экспорта будет тем, которое указано в `bar.js`. Эта «перезапись» произойдет незаметно:

```
foo.js
    export const baz = 'origin:foo';

bar.js
    export * from './foo.js';
    export const baz = 'origin:bar';
    import { baz } from './bar.js';

main.js
    console.log(baz);    // origin:bar
```

Также возможно перечислить, какие значения из внешнего модуля передаются в локальный экспорт. Этот синтаксис поддерживает псевдонимы:

```
export { foo, bar as myBar } from './foo.js';
```

Точно так же экспорт по умолчанию импортированного модуля может быть повторно использован и экспортирован как экспорт по умолчанию текущего модуля:

```
export { default } from './foo.js';
```

Никакая копия экспорта при этом не создается; он просто распространяет импортированную ссылку на исходный модуль. Заданное значение этого импорта все еще сохраняется в исходном модуле, и те же ограничения, связанные с изменением импорта, применяются к переэкспортированному импорту.

При выполнении переэкспорта также можно изменить обозначение «именованный/по умолчанию» из импортированного модуля. Именованный импорт может быть указан как экспорт по умолчанию следующим образом:

```
export { foo as default } from './foo.js';
```

Модули рабочих потоков

Модули ECMAScript 6 полностью совместимы с экземплярами `Worker`. При создании экземпляра `Worker` можно передать путь к файлу модуля так же, как вы передаете обычный файл сценария. Конструктор `Worker` принимает второй аргумент, позволяющий сообщить ему, что вы передаете файл модуля.

Создание экземпляра рабочего потока выглядит следующим образом:

```
// Второй аргумент имеет значение по умолчанию { type: 'classic' }
const scriptWorker = new Worker('scriptWorker.js');

const moduleWorker = new Worker('moduleWorker.js', { type: 'module' });
```

Внутри модульного рабочего потока метод `self.importScripts()`, который обычно используется для загрузки внешних сценариев внутри рабочего потока, определенного сценарием, выдаст ошибку. Это связано с тем, что поведение при импорте модуля соответствует поведению `importScripts()`.

Обратная совместимость

Поскольку обеспечение совместимости модулей ECMAScript будет постепенным, для начинающих пользователей важно иметь возможность разрабатывать как для браузеров, которые поддерживают модули, так и для браузеров, которые этого не делают. Для пользователей, которые хотят использовать модули ECMAScript 6 в браузере, когда это возможно, решения будут включать обслуживание двух версий кода — версию на основе модулей и версию на основе сценариев. Если это нежелательно, лучше использовать сторонние модульные системы, такие как SystemJS, или переносить модули ES6 во время сборки.

Первая стратегия заключается в проверке пользовательского агента браузера на сервере, сопоставлении его с известным списком браузеров, поддерживающих модули, и использовании его для определения того, какие файлы JS следует обслуживать. Этот метод хрупок и сложен, и его не рекомендуется использовать. Лучшее и более изящное решение — использование атрибутов сценария `type` и `nomodule`.

Если браузер не распознает значение атрибута типа элемента `<script>`, он не выполнит его содержимое. Для устаревших браузеров, которые не поддерживают модули, это означает, что `<script type="module">` никогда не будет выполняться, поэтому можно разместить запасной тег `<script>` рядом с тегом `<script type="module">`:

```
// Устаревшие браузеры не выполнят этот сценарий
<script type="module" src="module.js"></script>

// Устаревшие браузеры выполнят этот сценарий
<script src="script.js"></script>
```

Это, конечно, оставляет проблему браузеров, которые *поддерживают* модули. В этом случае предыдущий код будет выполнен дважды — очевидно, что это нежелательный результат. Для предотвращения этого браузеры, которые изначально поддерживают

модули ECMAScript 6, также распознают атрибут `nomodule`. Этот атрибут сообщает браузерам, поддерживающим модули ES6, о невозможности выполнения сценария. Устаревшие браузеры не распознают атрибут и игнорируют его.

Поэтому следующая конфигурация приведет к установке, в которой как современные, так и устаревшие браузеры будут выполнять ровно один из этих сценариев:

```
// Современные браузеры выполнят этот сценарий
// Устаревшие браузеры не выполнят этот сценарий
<script type="module" src="module.js"></script>

// Современные браузеры не выполнят этот сценарий
// Устаревшие браузеры выполнят этот сценарий
<script nomodule src="script.js"></script>
```

ИТОГИ

Паттерн Модуль остается вневременным инструментом для управления сложностью. Он позволяет разработчикам создавать сегменты изолированной логики, объявлять зависимости между этими сегментами и связывать их вместе. Более того, этот шаблон элегантно масштабируется до произвольной сложности на разных платформах.

В течение многих лет экосистема росла вокруг спорной дихотомии между CommonJS модульной системой, ориентированной на серверные среды, и AMD модульной системой, направленной на ограниченность клиентских сред. Обе системы получили взрывной рост, но код, написанный для каждой из них, во многих отношениях отличался друг от друга и часто приводил к ужасному образцу. Более того, ни одна из систем не была изначально реализована браузерами, и в результате этой несовместимости возникло множество инструментов, которые позволили использовать шаблон модуля в браузерах.

В спецификацию ECMAScript 6 включена совершенно новая концепция для модулей браузера, которая использует лучшее из обоих миров и объединяет их в более простой декларативный синтаксис. Браузеры все чаще предлагают поддержку использования собственных модулей, но также предоставляют надежные инструменты для преодоления разрыва между незначительной и полной поддержкой модулей ES6.

27

Рабочие потоки

- Введение в рабочие потоки
- Запуск фоновых задач с выделенными рабочими потоками
- Использование общих рабочих потоков
- Управление запросами с рабочими потоками служб

Утверждение «JavaScript является однопоточным» — практически мантра для сообщества фронтенд-разработчиков. Это утверждение, хотя оно и делает некоторые упрощающие предположения, эффективно описывает, как среда JavaScript обычно ведет себя внутри браузера. Поэтому оно полезно в качестве педагогического инструмента, помогающего веб-разработчикам понимать JavaScript.

Эта однопоточная парадигма по своей природе ограничительна, поскольку она предотвращает паттерны программирования, которые в других случаях осуществимы в языках, способных делегировать работу отдельным потокам или процессам. JavaScript связан с этой однопоточной парадигмой, чтобы сохранить совместимость с различными API-интерфейсами браузера, с которыми он должен взаимодействовать. Такие конструкции, как объектная модель документа, могут столкнуться с проблемами, если подвергнутся одновременным изменениям через несколько потоков JavaScript. Поэтому традиционные конструкции параллелизма, такие как потоки POSIX или класс Thread в Java, не являются началом расширения JavaScript.

В этом заключается основное ценностное предложение рабочих потоков: разрешить первичному потоку выполнения делегировать работу отдельной сущности без изменения существующей однопоточной модели. Хотя различные типы рабочих потоков, описанные в этой главе, имеют разные формы и функции, они едины в своем отделении от основной среды JavaScript.

ВВЕДЕНИЕ В РАБОЧИЕ ПОТОКИ

Единая среда JavaScript — это виртуализированная среда, работающая внутри операционной системы хоста. Каждой открытой странице в браузере выделяется своя среда. Это обеспечивает каждой странице собственную память, цикл обработки событий, DOM и т. д. Каждая страница более или менее изолирована и не может мешать другим страницам. Обязательным условием для браузера является одновременное управление многими средами, причем все они выполняются *параллельно*.

Используя *рабочие потоки*, браузеры могут выделить вторую дочернюю среду, которая полностью отделена от исходной среды страниц. Эта дочерняя среда не может взаимодействовать с однопоточными зависимыми конструкциями, такими как DOM, но в противном случае она может выполнять код параллельно с родительской средой.

Сравнение рабочих потоков и потоков выполнения

Вводные ресурсы обычно проводят сравнение между рабочими потоками и потоками выполнения. Во многих отношениях это удачное сравнение, поскольку рабочие потоки и потоки выполнения действительно имеют много общих характеристик:

- **Рабочие потоки реализованы в виде реальных потоков выполнения**. Например, механизм браузера Blink реализует рабочие потоки с `WorkerThread`, который соответствует базовому потоку платформы.
- **Рабочие потоки выполняются параллельно**. Несмотря на то, что страница и рабочий объект реализуют однопоточную среду JavaScript, инструкции в каждой среде могут выполняться параллельно.
- **Рабочие потоки могут разделять *часть* памяти**. Рабочие потоки могут использовать `SharedArrayBuffer` для разделения памяти между несколькими средами, в то время как потоки выполнения будут использовать блокировки для реализации управления параллелизмом. JavaScript использует интерфейс `Atomics` для реализации управления параллелизмом.

Рабочие потоки и потоки выполнения сильно перекрываются, но между ними есть некоторые важные различия:

- **Рабочие потоки не разделяют *всю* память**. В традиционной модели потоков несколько потоков имеют возможность чтения и записи в общую область памяти. За исключением `SharedArrayBuffer`, перемещение данных в и из рабочих потоков требует их копирования или переноса.
- **Рабочие потоки не обязательно являются частью одного и того же процесса**. Как правило, один процесс может порождать несколько потоков внутри него. В зависимости от реализации движка браузера рабочий поток может быть или не быть частью того же процесса, что и страница. Например, движок Chrome Blink использует отдельный процесс для общих рабочих и служебных потоков.

- **Рабочие потоки выполнения дороже создавать**. Рабочие потоки выполнения включают в себя собственный отдельный цикл событий, глобальные объекты, обработчики событий и другие функции, которые являются неотъемлемой частью среды JavaScript. Расходы на их создание не следует упускать из виду.

Как по форме, так и по функциям рабочие потоки не заменяют потоки выполнения.

В спецификации HTML Web Worker указано следующее:

> *Рабочие потоки имеют относительно большой вес и не предназначены для использования в больших количествах. Например, было бы неуместно запускать один рабочий поток на каждый пиксель изображения размером в четыре мегапикселя. Как правило, ожидается, что рабочие потоки будут долгоживущими и будут иметь высокую стоимость запуска и высокую стоимость памяти для каждого экземпляра.*

Типы рабочих потоков

Существует три основных типа рабочих потоков, определенных в спецификации Web Worker: *выделенный рабочий поток*, *общий рабочий поток* и *рабочий поток служб (служебный рабочий поток*). Все они широко доступны в современных веб-браузерах.

ПРИМЕЧАНИЕ Спецификацию Web Worker можно найти по адресу https://html.spec.whatwg.org/multipage/workers.html.

Выделенный рабочий поток

Выделенный рабочий поток, обычно называемый выделенным потоком или просто рабочим потоком, — это универсальная утилита, которая позволяет сценариям создавать отдельный поток JavaScript и делегировать ему задачи. Как следует из названия, выделенный рабочий поток может быть доступен только на той странице, которая его выделила.

Общий рабочий поток

Общий рабочий поток ведет себя очень похоже на выделенный. Основное отличие состоит в том, что к общему рабочему потоку можно обращаться в разных контекстах, включая разные страницы. Любые сценарии, выполняющиеся в том же источнике, что и сценарий, который первоначально выделил общий рабочий поток, могут отправлять и получать сообщения для общего потока.

Служебный рабочий поток

Служебный рабочий поток полностью отличается от выделенного или общего. Его основная цель — выступать в качестве арбитра для сетевых запросов, способного перехватывать, перенаправлять и изменять запросы, отправляемые страницей.

ПРИМЕЧАНИЕ Существует несколько других рабочих спецификаций, таких как ChromeWorker или Web Audio API. Они широко не поддерживаются или предназначены для нишевых приложений и поэтому не включены в книгу.

WorkerGlobalScope

На веб-странице объект `window` предоставляет широкий набор глобальных переменных для сценариев, выполняющихся внутри нее. Внутри рабочего потока понятие «окна» не имеет смысла, и поэтому его глобальный объект является экземпляром `WorkerGlobalScope`. Вместо `window` этот глобальный объект доступен через ключевое слово `self`.

Свойства и методы WorkerGlobalScope

Свойства, доступные для `self`, являются строгим подмножеством свойств, доступных для `window`. Некоторые свойства возвращают версию объекта в стиле рабочего потока:

- **navigator** — возвращает `WorkerNavigator`, связанный с этим рабочим потоком.
- **self** — возвращает объект `WorkerGlobalScope`.
- **location** — возвращает `WorkerLocation`, связанный с этим рабочим потоком.
- **performance** — возвращает объект `Performance` (с сокращенным набором свойств и методов).
- **console** — возвращает объект `Console`, связанный с этим рабочим потоком. Нет ограничений по API.
- **caches** — возвращает объект `CacheStorage`, связанный с этим рабочим потоком. Нет ограничений по API.
- **indexedDB** — возвращает объект `IDBFactory`.
- **isSecureContext** — возвращает логическое значение, указывающее, является ли контекст рабочего потока безопасным.
- **origin** — возвращает источник объекта `WorkerGlobalScope`.

Точно так же некоторые методы, доступные для `self`, являются подмножеством методов, доступных для `window`. Методы `self` работают идентично своим аналогам в `window`:

- `atob()`
- `btoa()`
- `clearInterval()`
- `clearTimeout()`
- `createImageBitmap()`
- `fetch()`

- `setInterval()`
- `setTimeout()`

`WorkerGlobalScope` также представляет новый глобальный метод `importScripts()`, который доступен только внутри рабочего потока. Этот метод описан ниже в этой главе.

Подклассы WorkerGlobalScope

`WorkerGlobalScope` фактически нигде не реализуется. Каждый тип рабочего потока использует свой собственный вид глобального объекта, который наследуется от `WorkerGlobalScope`.

- Выделенный рабочий поток использует `DedicatedWorkerGlobalScope`.
- Общий рабочий поток использует `SharedWorkerGlobalScope`.
- Служебный рабочий поток использует `ServiceWorkerGlobalScope`.

Различия между этими глобальными объектами обсуждаются в соответствующих разделах этой главы.

ВЫДЕЛЕННЫЕ РАБОЧИЕ ПОТОКИ

Выделенный рабочий поток — самый простой тип рабочих потоков. Выделенные рабочие потоки создаются веб-страницей для выполнения сценариев вне потока выполнения страницы. Эти рабочие потоки способны обмениваться информацией с родительской страницей, отправлять сетевые запросы, выполнять файловый ввод-вывод, выполнять интенсивные вычисления, массово обрабатывать данные или любое другое количество вычислительных задач, которые не подходят для потока выполнения страницы (где они могли бы стать причиной проблем с задержкой).

ПРИМЕЧАНИЕ При работе с рабочими потоками важны понятия, где выполняется сценарий и откуда он был загружен. Если не указано иное, в этой главе предполагается, что main.js – это скрипт верхнего уровня, загружаемый из корневого пути домена https://example.com и выполняющийся на нем.

Основные сведения о выделенных рабочих потоках

Выделенные рабочие потоки могут быть точно описаны как *фоновые* сценарии. Характеристики рабочего потока JavaScript, включая управление жизненным циклом, путь к коду и ввод/вывод, регулируются единственным сценарием, предоставляемым при инициализации потока. Этот сценарий может, в свою очередь, запрашивать дополнительные сценарии, но рабочий поток всегда начинается с одного исходного сценария.

Создание выделенного рабочего потока

Наиболее распространенный способ создания выделенного рабочего потока — через загруженный файл JavaScript. Путь к файлу предоставляется конструктору `Worker`, который, в свою очередь, асинхронно загружает сценарий в фоновом режиме и создает экземпляр рабочего потока. Конструктор запрашивает путь к файлу, хотя этот путь может принимать несколько различных форм.

В следующем простом примере создается пустой выделенный рабочий поток:

EMPTYWORKER.JS

```
  // пустой JS-файл рабочего потока
```

MAIN.JS

```
  console.log(location.href);    // "https://example.com/"
  const worker = new Worker(location.href + 'emptyWorker.js');
  console.log(worker);           // Worker {}
```

Эта демонстрация проста, но включает несколько основополагающих концепций:

- Файл `emptyWorker.js` загружается по абсолютному пути. В зависимости от структуры приложения использование абсолютного URL-адреса часто будет излишним.
- Этот файл загружается в фоновом режиме, и инициализация рабочего потока происходит в потоке, полностью отделенном от `main.js`.
- Сам рабочий поток существует в отдельной среде JavaScript, поэтому `main.js` должен использовать объект `Worker` как прокси для связи с этим потоком. В приведенном выше примере этот объект присвоен `worker` переменной.
- Хотя сам рабочий поток может еще не существовать, этот объект `Worker` доступен сразу в исходной среде.

Предыдущий пример может быть изменен для использования относительного пути; однако для этого требуется, чтобы `main.js` выполнялся по тому же пути, из которого можно загрузить `emptyWorker.js`:

```
const worker = new Worker('./emptyWorker.js');
console.log(worker);    // Worker {}
```

Ограничения безопасности рабочего потока

Файлы сценариев рабочих потоков могут быть загружены только из того же источника, что и родительская страница. Попытки загрузить файл сценария потока из удаленного источника приведут к ошибке при попытке создать поток, как показано здесь:

```
// Попытка создать рабочий поток из сценария по адресу https://example.com/worker.js
const sameOriginWorker = new Worker('./worker.js');

// Попытка создать рабочй поток из сценария по адресу
// https://untrusted.com/worker.js
const remoteOriginWorker = new Worker('https://untrusted.com/worker.js');
```

```
// Error: Uncaught DOMException: Failed to construct 'Worker':
// Script at https://untrusted.com/main.js cannot be accessed
// from origin https://example.com
```

ПРИМЕЧАНИЕ Ограничение источника рабочего потока не препятствует выполнению кода из удаленного источника. Он может быть выполнен внутри потока с помощью `importScripts()`, который будет описан ниже в этой главе.

Рабочие потоки, созданные из загруженного сценария, не подчиняются политике безопасности содержимого документа, поскольку они выполняются в контексте, отдельном от родительского документа. Однако если поток загружается из сценария с глобально уникальным идентификатором, как в случае, когда он загружается из большого двоичного объекта, он *будет* подчиняться политике безопасности содержимого родительского документа.

ПРИМЕЧАНИЕ Создание рабочих потоков из Blob-объектов описано в разделе «Создание рабочего потока из встроенного JavaScript».

Использование объекта Worker

Объект `Worker`, возвращаемый конструктором `Worker()`, используется в качестве единой точки связи с вновь созданным выделенным рабочим потоком. Он может использоваться для передачи информации между рабочим потоком и родительским контекстом, а также для отслеживания событий, генерируемых выделенным потоком.

ПРИМЕЧАНИЕ Тщательно отслеживайте ссылки на объекты `Worker`, связанные с каждым создаваемым рабочим потоком. Пока рабочий поток не завершен, он не может быть собран мусором, и нет никакого программного инструмента, который можно использовать для восстановления ссылки на объект `worker`.

Объект `Worker` поддерживает следующие свойства обработчика событий:

- **onerror** — может быть назначен обработчик событий, который будет вызываться всякий раз, когда `ErrorEvent` типа `error` всплывает от рабочего потока.
  - Это событие происходит, когда в рабочем потоке возникает ошибка.
  - Это событие также может быть обработано с помощью `worker.addEventListener('error', handler)`.
- **onmessage** — может быть назначен обработчик события, который будет вызываться всякий раз, когда сообщение `MessageEvent` типа `message` всплывает от рабочего потока.
  - Это событие происходит, когда сценарий потока отправляет сообщение обратно в родительский контекст.

  - Это событие также может быть обработано с помощью `worker.addEventListener('message', handler)`.
- **onmessageerror** — может быть назначен обработчик события, который будет вызываться всякий раз, когда `MessageEvent` типа `messageerror` всплывает от рабочего потока.
  - Это событие происходит при получении сообщения, которое не может быть десериализовано.
  - Это событие также может быть обработано с помощью `worker.addEventListener('messageerror', handler)`.

Объект `Worker` также поддерживает следующие методы:

- **postMessage()** — используется для отправки информации рабочему потоку через асинхронные события сообщений.
- **terminate()** — используется для немедленного прекращения работы потока. Рабочему потоку не предоставляется никакой возможности для очистки, и сценарий внезапно завершается.

DedicatedWorkerGlobalScope

Внутри выделенного рабочего потока глобальная область видимости является экземпляром `DedicatedWorkerGlobalScope`. Это наследуется от `WorkerGlobalScope` и, следовательно, включает в себя все его свойства и методы. Рабочий поток может получить доступ к этой глобальной области видимости через `self`:

GLOBALSCOPEWORKER.JS
```
  console.log('inside worker:', self);
```

MAIN.JS
```
  const worker = new Worker('./globalScopeWorker.js');

  console.log('created worker:', worker);

  // created worker: Worker {}
  // inside worker: DedicatedWorkerGlobalScope {}
```

Как показано здесь, объект `console` и в сценарии верхнего уровня, и в рабочем потоке будет записывать информацию в консоль браузера, что полезно для отладки. Поскольку рабочий поток имеет незначительную задержку запуска, сообщение журнала рабочего потока печатается после сообщения журнала основного потока, даже если объект `Worker` существует.

ПРИМЕЧАНИЕ Обратите внимание, что два отдельных потока JavaScript отправляют сообщения в единый объект console, который впоследствии сериализует сообщения и печатает их в консоли браузера. Браузер получает сообщения от двух разных потоков JavaScript и отвечает за их чередование по своему усмотрению. По этой причине использование сообщений журнала из нескольких потоков для определения порядка работы должно выполняться с осторожностью.

`DedicatedWorkerGlobalScope` расширяет `WorkerGlobalScope` следующими свойствами и методами:

- **name** — необязательный идентификатор строки, который может быть предоставлен конструктору `Worker`.
- **postMessage()** — аналог `worker.postMessage()`. Он используется для отправки сообщений обратно из рабочего потока в родительский контекст.
- **close()** — аналог `worker.terminate()`. Он используется для немедленного завершения рабочего потока. Потоку не предоставляется никакой возможности для очистки; сценарий внезапно завершается.
- **importScripts()** — используется для импорта произвольного количества сценариев в рабочий поток.

Выделенные рабочие потоки и явные MessagePort

Вы заметите, что выделенный рабочий поток и объект `DedicatedWorkerGlobalScope` совместно используют несколько частей интерфейсных обработчиков и методов с `MessagePort`: `onmessage`, `onmessageerror`, `close()` и `postMessage()`. Это не случайно: выделенные рабочие потоки неявно используют `MessagePort` для связи между контекстами.

Реализация такова, что объект в родительском контексте и `DedicatedWorkerGlobalScope` эффективно поглощают `MessagePort` и предоставляют его обработчики и методы как часть их собственных интерфейсов. Другими словами, вы все еще отправляете сообщения через `MessagePort`; вы просто не имеете доступа к самому порту.

Существуют некоторые несоответствия, такие как соглашения для `start()` и `close()`. Порты для выделенных рабочих потоков автоматически начнут отправку сообщений в очереди, поэтому запуск `start()` не требуется. Кроме того, метод `close()` не имеет смысла в контексте выделенного рабочего потока, так как закрытие порта фактически лишило бы потока труда. Следовательно, метод `close()`, вызываемый изнутри рабочего потока (или `terminate()` извне), не только закрывает порт, но и завершает рабочий поток.

Жизненный цикл выделенного рабочего потока

Конструктор `Worker()` — это начало жизни выделенного рабочего потока. После вызова он инициирует запрос сценария рабочего потока и возвращает объект `Worker` в родительский контекст. Несмотря на то, что объект `Worker` сразу доступен для использования в родительском контексте, связанный поток, возможно, еще не был создан из-за задержек в сети сценария рабочего потока или задержки инициализации.

Вообще говоря, выделенные рабочие потоки можно неофициально охарактеризовать как существующие в одном из трех состояний: *инициализирующийся*, *активный* и *завершенный*. Состояние выделенного рабочего потока непрозрачно для любых

других контекстов. Хотя объект `Worker` может существовать в родительском контексте, он не может быть установлен, если выделенный поток инициализируется, активен или завершен. Другими словами, объект `Worker`, связанный с активным выделенным рабочим потоком, неотличим от объекта `Worker`, связанного с завершенным выделенным рабочим потоком.

Во время инициализации, хотя сценарий рабочего потока еще не начал выполняться, можно ставить в очередь сообщения для него. Эти сообщения будут ожидать, пока поток станет активным, и впоследствии будут добавлены в его очередь сообщений. Такое поведение демонстрируется здесь:

INITIALIZINGWORKER.JS
```
self.addEventListener('message', ({data}) => console.log(data));
```

MAIN.JS
```
const worker = new Worker('./initializingWorker.js');

// Рабочий поток может еще инициализироваться,
// но информация из postMessage будет обработана корректно.
worker.postMessage('foo');
worker.postMessage('bar');
worker.postMessage('baz');

// foo
// bar
// baz
```

После создания выделенный рабочий поток будет работать в течение всей жизни страницы, если он не будет явно прерван либо через самоуничтожение (`self.close()`), либо через внешнее завершение (`worker.terminate()`). Даже после завершения рабочего сценария рабочая среда сохранится. Пока поток еще жив, связанный с ним объект `Worker` не будет собран как мусор.

Самоуничтожение и внешнее завершение в конечном счете выполняют одну и ту же процедуру завершения рабочего потока. Рассмотрим следующий пример, где рабочий поток самоуничтожается между отправкой сообщений:

CLOSEWORKER.JS
```
self.postMessage('foo');
self.close();
self.postMessage('bar');
setTimeout(() => self.postMessage('baz'), 0);
```

MAIN.JS
```
const worker = new Worker('./worker.js');
worker.onmessage = ({data}) => console.log(data);

// foo
// bar
```

Хотя вызывается `close()`, очевидно, что выполнение рабочего потока не прекращается немедленно. Метод `close()` только дает указание потоку отказаться от всех задач в цикле событий и предотвратить добавление дальнейших задач. Вот почему

«baz» никогда не выведется в консоль. Остановка синхронного выполнения *не* требуется, и, следовательно, «bar» по-прежнему выводится в консоль, поскольку это обрабатывается в цикле событий родительского контекста.

Теперь рассмотрим следующий пример внешнего завершения:

TERMINATEWORKER.JS
```
self.onmessage = ({data}) => console.log(data);
```

MAIN.JS
```
const worker = new Worker('./worker.js');

// Дает 1000мс на инициализацию worker
setTimeout(() => {
    worker.postMessage('foo');
    worker.terminate();
    worker.postMessage('bar');
    setTimeout(() => worker.postMessage('baz'), 0);
}, 1000);

// foo
```

Здесь рабочий поток сначала отправляет сообщение `postMessage` с «foo», которое он может обработать до внешнего завершения. После вызова `terminate()` очередь сообщений потока очищается и блокируется — поэтому печатается только «foo».

> **ПРИМЕЧАНИЕ** И `close()`, и `terminate()` являются идемпотентными операциями; они могут быть безвредно вызваны несколько раз. Эти методы просто служат для того, чтобы пометить рабочий поток как подлежащий завершению, поэтому многократный вызов его не будет иметь вредных последствий.

В течение своего жизненного цикла выделенный рабочий поток связан только с одной веб-страницей (упоминаемой в спецификации Web Worker как *документ*). Если не было вызвано явное завершение, выделенный рабочий поток будет сохраняться, пока существует связанный документ. Если браузер закрывает страницу (возможно, с помощью навигации или закрытия вкладки или окна), браузер помечает рабочие потоки, связанные с этим документом, как подлежащие завершению, и их выполнение немедленно останавливается.

Настройка параметров рабочих потоков

Конструктор `Worker()` допускает использование необязательного объекта настройки в качестве второго аргумента. Объект настройки поддерживает следующие свойства:

- `name` — строковый идентификатор, который может быть прочитан изнутри потока через `self.name`.
- `type` — указывает, как должен выполняться загруженный сценарий, `classic` или `module`. `classic` выполняет сценарий как обычный сценарий; `module` выполняет сценарий как модульный.

- `credentials` — когда для типа задано значение «module», указывается, как следует извлекать модульные сценарии рабочего потока в отношении передачи учетных данных. Может иметь значение `omit`, `same-origin` или `include`. Эти параметры работают идентично опции учетных данных `fetch()`. Когда тип установлен на `classic`, по умолчанию имеет значение `omit`.

ПРИМЕЧАНИЕ Некоторые современные браузеры все еще не полностью поддерживают модульные рабочие потоки или могут потребовать использования флага, чтобы включить их поддержку.

Создание рабочего потока из встроенного JavaScript

Рабочие потоки должны быть созданы из файла сценария, но это не означает, что сценарий должен быть загружен из удаленного ресурса. Выделенный рабочий поток также может быть создан из встроенного скрипта через Blob-объект URL. Это позволяет быстрее инициализировать рабочий поток благодаря устранению задержки в двусторонней сети.

В следующем примере создается рабочий поток из встроенного скрипта:

```
// Создание строки JS-кода для выполнения
const workerScript = `
    self.onmessage = ({data}) => console.log(data);
`;

// Генерация экземпляра blob из строки сценария
const workerScriptBlob = new Blob([workerScript]);

// Создание объекта URL для экземпляра blob
const workerScriptBlobUrl = URL.createObjectURL(workerScriptBlob);

// Создание выделенного рабочего потока из blob
const worker = new Worker(workerScriptBlobUrl);

worker.postMessage('blob worker script');
// blob worker script
```

В этом примере строка сценария передается в экземпляр Blob-объекта, которому затем присваивается URL-адрес объекта, который, в свою очередь, передается в конструктор `Worker()`. Конструктор создает выделенный рабочий поток как обычно.

Вкратце этот же пример может выглядеть следующим образом:

```
const worker = new Worker(URL.createObjectURL(new Blob([`self.onmessage =
({data}) => console.log(data);`])));

worker.postMessage('blob worker script');
// blob worker script
```

Рабочие потоки также могут воспользоваться сериализацией функций с помощью встроенной инициализации сценария. Поскольку метод `toString()` функции возвращает фактический код функции, функция может быть определена в родительском контексте, но выполнена в дочернем. Рассмотрим следующий простой пример:

```
function fibonacci(n) {
    return n < 1 ? 0
        : n <= 2 ? 1
        : fibonacci(n - 1) + fibonacci(n - 2);
}

const workerScript = `
    self.postMessage(
        (${fibonacci.toString()})(9)
    );
`;

const worker = new Worker(URL.createObjectURL(new Blob([workerScript])));

worker.onmessage = ({data}) => console.log(data);

// 34
```

Здесь приведена намеренно дорогостоящая реализация последовательности Фибоначчи, которая сериализуется и передается рабочему потоку. Он вызывается как выражение немедленного вызова функции (IIFE) и передает параметр, а результат передается обратно в основной поток. Несмотря на то, что вычисления Фибоначчи здесь довольно затратны, все вычисления делегируются рабочему потоку и, следовательно, не влияют на производительность родительского контекста.

> **ПРИМЕЧАНИЕ** Важно отметить, что этот метод сериализации функции требует, чтобы переданная функция не использовала никаких ссылок, полученных через замыкание, включая глобальные переменные, такие как `window`, так как они будут ломаться при выполнении внутри рабочего потока.

Динамическое выполнение сценария внутри рабочего потока

Сценарии рабочего потока не обязательно должны быть монолитными объектами. Можно программно загрузить и выполнить произвольное количество сценариев с помощью метода `importScripts()`, который доступен для глобального объекта рабочего потока. Этот метод будет загружать сценарии и синхронно выполнять их по порядку. Рассмотрим следующий пример, который загружает и выполняет два сценария:

MAIN.JS

```
  const worker = new Worker('./worker.js');

  // importing scripts
  // scriptA executes
```

```
  // scriptB executes
  // scripts imported
```

SCRIPTA.JS

```
  console.log('scriptA executes');
```

SCRIPTB.JS

```
  console.log('scriptB executes');
```

WORKER.JS

```
  console.log('importing scripts');

  importScripts('./scriptA.js');
  importScripts('./scriptB.js');

  console.log('scripts imported');
```

`importScripts()` принимает произвольное количество аргументов сценария. Браузер может загружать их в любом порядке, но сценарии будут выполняться строго в порядке следования параметров. Следовательно, следующий сценарий рабочего потока будет эквивалентен предыдущему:

```
console.log('importing scripts');

importScripts('./scriptA.js', './scriptB.js');

console.log('scripts imported');
```

Загрузка сценариев подчиняется обычным ограничениям CORS, но в противном случае рабочие потоки могут свободно запрашивать сценарии из других источников. Эта стратегия импорта аналогична динамической загрузке сценария с помощью генерации тега `<script>`. В этом духе область видимости используется совместно с импортированными сценариями. Такое поведение демонстрируется здесь:

MAIN.JS

```
  const worker = new Worker('./worker.js', {name: 'foo'});

  // importing scripts in foo with bar
  // scriptA executes in foo with bar
  // scriptB executes in foo with bar
  // scripts imported
```

SCRIPTA.JS

```
  console.log(`scriptA executes in ${self.name} with ${globalToken}`);
```

SCRIPTB.JS

```
  console.log(`scriptB executes in ${self.name} with ${globalToken}`);
```

WORKER.JS

```
  const globalToken = 'bar';

  console.log(`importing scripts in ${self.name} with ${globalToken}`);

  importScripts('./scriptA.js', './scriptB.js');

  console.log('scripts imported');
```

Передача задач вложенным рабочим потокам

Вы можете обнаружить, что рабочие потоки должны порождать свои «вложенные потоки». Это может быть полезно в тех случаях, когда в вашем распоряжении несколько ядер ЦП для распараллеливания вычислений. Выбор модели подчиненного рабочего потока должен быть сделан только после тщательного рассмотрения проекта: запуск нескольких потоков может повлечь за собой значительные вычислительные затраты и должен быть сделан только в том случае, если выгоды от распараллеливания перевешивают затраты.

Создание вложенного рабочего потока работает почти так же, как и создание обычного, за исключением разрешения пути: путь сценария вложенного потока будет разрешаться относительно его родительского рабочего потока, а не главной страницы. Это демонстрируется следующим образом (обратите внимание на добавление каталога сценариев):

MAIN.JS

```
const worker = new Worker('./js/worker.js');

// worker
// subworker
```

JS/WORKER.JS

```
console.log('worker');

const worker = new Worker('./subworker.js');
```

JS/SUBWORKER.JS

```
console.log('subworker');
```

> **ПРИМЕЧАНИЕ** И сценарии рабочих потоков верхнего уровня, и сценарии вложенных рабочих потоков должны загружаться из того же источника, что и главная страница.

Обработка ошибок рабочих потоков

Если в сценарии рабочего потока выдается ошибка, отделение рабочего потока будет препятствовать прерыванию родительского потока выполнения. Это демонстрируется здесь, где блок `try/catch` не улавливает выданную ошибку:

MAIN.JS

```
try {
    const worker = new Worker('./worker.js');
    console.log('no error');
} catch(e) {
    console.log('caught error');
}

// no error
```

WORKER.JS

```
throw Error('foo');
```

Тем не менее это событие все еще будет всплывать в глобальном рабочем контексте, и к нему можно получить доступ, установив наблюдатель за событием ошибки на объект `Worker`. Это показано здесь:

MAIN.JS
```
  const worker = new Worker('./worker.js');
  worker.onerror = console.log;

  // ErrorEvent {message: "Uncaught Error: foo"}
```

WORKER.JS
```
  throw Error('foo');
```

Общение с выделенным рабочим потоком

Все общение с рабочим потоком происходит с помощью асинхронных сообщений, но эти сообщения могут принимать несколько различных форм.

Связь с помощью postMessage()

Самая простая и распространенная форма — это использование `postMessage()` для передачи сериализованных сообщений туда и обратно. Простой пример с факториалом показан ниже:

FACTORIALWORKER.JS
```
  function factorial(n) {
      let result = 1;
      while(n) { result *= n--; }
      return result;
  }

  self.onmessage = ({data}) => {
      self.postMessage(`${data}! = ${factorial(data)}`);
  };
```

```
  MAIN.JS
  const factorialWorker = new Worker('./factorialWorker.js');

  factorialWorker.onmessage = ({data}) => console.log(data);

  factorialWorker.postMessage(5);
  factorialWorker.postMessage(7);
  factorialWorker.postMessage(10);

  // 5! = 120
  // 7! = 5040
  // 10! = 3628800
```

Для простой передачи сообщений использование `postMessage()` для связи между окном и потоком очень похоже на передачу сообщений между двумя окнами. Основное отличие состоит в том, что отсутствует концепция ограничения `targetOrigin`, которое присутствует для `Window.prototype.postMessage`, но не для `WorkerGlobalScope.prototype.postMessage` или `Worker.prototype.postMessage`. Причина такого соглашения

проста: источник сценария рабочего потока ограничен источником главной страницы, поэтому механизм фильтрации не используется.

Общение с помощью MessageChannel

Как для основного, так и для рабочего потока обмен данными через `postMessage()` включает в себя вызов метода для глобального объекта и определение в нем специального протокола передачи. Его можно заменить API обмена сообщениями каналов, который позволяет создать явный канал связи между двумя контекстами.

Экземпляр `MessageChannel` имеет два порта, представляющих две конечные точки связи. Чтобы родительская страница и рабочий поток могли обмениваться данными по каналу, один порт можно передать потоку, как показано здесь:

WORKER.JS

```
// Хранение messagePort глобально внутри наблюдателя
let messagePort = null;

function factorial(n) {
    let result = 1;
    while(n) { result *= n--; }
    return result;
}

// Установка обработчика сообщений на глобальный объект
self.onmessage = ({ports}) => {
    // Установка порта проходит только один раз
    if (!messagePort) {
        // Первое сообщение передает порт,
        // назначает его переменной и снимает наблюдатель
        messagePort = ports[0];
        self.onmessage = null;

        // Установка обработчика сообщений на глобальный объект
        messagePort.onmessage = ({data}) => {
            // Последовательная передача данных в сообщениях
            messagePort.postMessage(`${data}! = ${factorial(data)}`);
        };
    }
};
```

MAIN.JS

```
const channel = new MessageChannel();
const factorialWorker = new Worker('./worker.js');

// Отправка объекта MessagePort рабочему потоку.
// Worker может корректно это обработать
factorialWorker.postMessage(null, [channel.port1]);

// Отправка сообщения по каналу связи
channel.port2.onmessage = ({data}) => console.log(data);

// Worker может отвечать через канал
channel.port2.postMessage(5);

// // 5! = 120
```

В этом примере родительская страница делит `MessagePort` с рабочим потоком через `postMessage`. Нотация массива должна передавать отправляемый объект между контекстами, такая концепция рассматривается позже в этой главе. Рабочий поток поддерживает ссылку на этот порт и использует ее для передачи сообщений вместо передачи их через глобальный объект. Конечно, этот формат все еще использует своего рода специальный протокол: рабочий поток написан так, чтобы ожидать, что первое сообщение отправит порт, а последующие — отправят данные.

Использование экземпляра `MessageChannel` для связи с родительской страницей в значительной степени избыточно, поскольку глобальные возможности `postMessage` по существу выполняют ту же задачу, что и `channel.postMessage` (не включая дополнительные функции интерфейса `MessageChannel`). `MessageChannel` действительно становится полезным в ситуации, когда два потока хотели бы напрямую общаться друг с другом. Это может быть достигнуто путем передачи одного порта каждому рабочему потоку. Рассмотрим следующий пример, в котором массив передается одному потоку, затем другому и возвращается на главную страницу:

MAIN.JS

```
  const channel = new MessageChannel();
  const workerA = new Worker('./worker.js');
  const workerB = new Worker('./worker.js');

  workerA.postMessage('workerA', [channel.port1]);
  workerB.postMessage('workerB', [channel.port2]);

  workerA.onmessage = ({data}) => console.log(data);
  workerB.onmessage = ({data}) => console.log(data);

  workerA.postMessage(['page']);

  // ['page', 'workerA', 'workerB']

  workerB.postMessage(['page'])

  // ['page', 'workerB', 'workerA']
```

WORKER.JS

```
  let messagePort = null;
  let contextIdentifier = null;

  function addContextAndSend(data, destination) {
      // Добавление идентификатора для отслеживания момента, когда он достигнет
      // рабочего потока
      data.push(contextIdentifier);

      // Отправка данных к следующему месту назначения
      destination.postMessage(data);
  }

  self.onmessage = ({data, ports}) => {
      // Если в сообщении передан порт,
      // настраивается рабочий поток
      if (ports.length) {
```

```
        // Запись идентификатора
        contextIdentifier = data;

        // Захват MessagePort
        messagePort = ports[0];

        // Добавление обработчика для отправки полученных данных
        // обратно родителю
        messagePort.onmessage = ({data}) => {
            addContextAndSend(data, self);
        }
    } else {
        addContextAndSend(data, messagePort);
    }
};
```

В этом примере на каждой части пути массива в массив будет добавляться строка, чтобы пометить момент прибытия. Массив передается с родительской страницы в рабочий поток, который добавляет свой контекстный идентификатор. Потом он передается от одного потока к другому, что добавляет второй идентификатор контекста. Затем он передается обратно на главную страницу, где записывается массив. Обратите внимание в этом примере на то, как, поскольку два рабочих потока совместно используют общий сценарий, эта схема передачи массива работает в двух направлениях.

Общение с помощью BroadcastChannel

Сценарии, работающие в одном источнике, могут отправлять и получать сообщения в общем `BroadcastChannel`. Этот тип канала проще в настройке и не требует беспорядочной передачи портов, требуемой с `MessageChannel`. Это можно сделать следующим образом:

MAIN.JS

```
const channel = new BroadcastChannel('worker_channel');
const worker = new Worker('./worker.js');

channel.onmessage = ({data}) => {
    console.log(`heard ${data} on page`);
}

setTimeout(() => channel.postMessage('foo'), 1000);

// heard foo in worker
// heard bar on page
```

WORKER.JS

```
const channel = new BroadcastChannel('worker_channel');

channel.onmessage = ({data}) => {
    console.log(`heard ${data} in worker`);
    channel.postMessage('bar');
}
```

Обратите внимание, что страница ожидает 1000 миллисекунд перед отправкой начального сообщения в `BroadcastChannel`. Поскольку в этом типе канала отсутствует понятие владения портом, передаваемые сообщения не будут обрабатываться, если в канале нет другого объекта, прослушивающего канал. В этом случае без `setTimeout` задержка инициализации рабочего потока достаточно велика, чтобы предотвратить установку обработчика сообщений потока до его фактической отправки.

Передача данных рабочего потока

Рабочие потоки часто должны быть обеспечены данными в той или иной форме. Поскольку рабочие потоки выполняются в отдельном контексте, при передаче фрагмента данных из одного контекста в другой возникают накладные расходы. В языках, которые поддерживают традиционные модели многопоточности, в вашем распоряжении есть такие понятия, как блокировки, мьютексы и изменчивые переменные. В JavaScript существует три способа перемещения информации между контекстами: *алгоритм структурированного клонирования*, *переносимые объекты* и *совместно используемые буферы массивов*.

Алгоритм структурированного клонирования

Алгоритм структурированного клонирования можно использовать для совместного использования фрагмента данных между двумя отдельными контекстами выполнения. Этот алгоритм реализован браузером за кулисами, но он не может быть вызван явно.

Когда объект передается в `postMessage()`, браузер подхватывает объект и делает копию в контексте назначения. Следующие типы полностью поддерживаются алгоритмом структурированного клонирования:

- Все типы примитивов, кроме `Symbol`.
- Объект `Boolean`.
- Объект `String`.
- `BDate`.
- `RegExp`.
- `Blob`.
- `File`.
- `FileList`.
- `ArrayBuffer`.
- `ArrayBufferView`.
- `ImageData`.
- `Array`.
- `Object`.
- `Map`.
- `Set`.

Несколько замечаний о поведении алгоритма структурированного клонирования:

- После копирования изменения объекта в исходном контексте не будут распространяться на целевой объект контекста.
- Алгоритм структурированного клонирования распознает, когда объект содержит цикл, и не будет бесконечно перемещаться по объекту.
- Попытка клонировать объект `Error`, объект `Function` или узел DOM вызовет ошибку.
- Алгоритм структурированного клонирования не всегда создает точную копию.
- Дескрипторы свойств объекта, методы чтения и записи свойств не клонируются и будут возвращаться к значениям по умолчанию, где это применимо.
- Цепочки прототипов не клонируются.
- Свойство `RegExp.prototype.lastIndex` не клонируется.

ПРИМЕЧАНИЕ Алгоритм структурированного клонирования может быть вычислительно затратным, когда копируемый объект велик. По возможности избегайте большого или чрезмерного копирования.

Переносимые объекты

Можно передать право собственности из одного контекста в другой, используя *переносимые объекты*. Это особенно полезно в тех случаях, когда нецелесообразно копировать большие объемы данных между контекстами. Передается только несколько типов:

- `ArrayBuffer`
- `MessagePort`
- `ImageBitmap`
- `OffscreenCanvas`

Вторым необязательным аргументом для `postMessage()` является массив, указывающий, какие объекты следует перенести в контекст назначения. При обходе объекта полезной нагрузки сообщения браузер проверяет ссылки на объекты в массиве передаваемых объектов и выполняет передачу этих объектов вместо их копирования. Это означает, что переданные объекты могут быть отправлены в копируемой полезной нагрузке сообщения, такой как объект или массив.

В следующем примере демонстрируется обычное структурированное клонирование `ArrayBuffer` в рабочий поток. Здесь передача объекта не происходит:

MAIN.JS

```
const worker = new Worker('./worker.js');

// Создание буфера из 32 байтов
```

```
  const arrayBuffer = new ArrayBuffer(32);

  console.log(`page's buffer size: ${arrayBuffer.byteLength}`);    // 32

  worker.postMessage(arrayBuffer);

  console.log(`page's buffer size: ${arrayBuffer.byteLength}`);    // 32
```

WORKER.JS

```
  self.onmessage = ({data}) => {
      console.log(`worker's buffer size: ${data.byteLength}`);     // 32
  };
```

Когда `ArrayBuffer` указан в качестве передаваемого объекта, ссылка на буферную память стирается в родительском контексте и выделяется рабочему контексту. Это демонстрируется здесь, где память, выделенная внутри `ArrayBuffer`, удаляется из родительского контекста:

MAIN.JS

```
  const worker = new Worker('./worker.js');

  // Создание буфера из 32 байтов
  const arrayBuffer = new ArrayBuffer(32);

  console.log(`page's buffer size: ${arrayBuffer.byteLength}`);    // 32

  worker.postMessage(arrayBuffer);

  console.log(`page's buffer size: ${arrayBuffer.byteLength}`);    // 0
```

WORKER.JS

```
  self.onmessage = ({data}) => {
      console.log(`worker's buffer size: ${data.byteLength}`);     // 32
  };
```

Вполне допустимо вкладывать переносимые объекты в другие типы объектов. Внешний объект будет скопирован, а вложенный объект — передан:

MAIN.JS

```
  const worker = new Worker('./worker.js');

  // Создание буфера из 32 байтов
  const arrayBuffer = new ArrayBuffer(32);

  console.log(`page's buffer size: ${arrayBuffer.byteLength}`);    // 32

  worker.postMessage({foo: {bar: arrayBuffer}}, [arrayBuffer]);

  console.log(`page's buffer size: ${arrayBuffer.byteLength}`);    // 0
```

WORKER.JS

```
  self.onmessage = ({data}) => {
      console.log(`worker's buffer size: ${data.foo.bar.byteLength}`);    // 32
  };
```

SharedArrayBuffer

ПРИМЕЧАНИЕ SharedArrayBuffer был отключен во всех основных браузерах в январе 2018 г. из-за Spectre (https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-5753) и Meltdown (https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-5754) уязвимостей. С 2019 г. некоторые браузеры начали постепенно включать эту функцию.

Вместо того чтобы быть клонированным или перенесенным, SharedArrayBuffer представляет собой ArrayBuffer, который разделяется между контекстами браузера. При передаче SharedArrayBuffer внутри postMessage() браузер передает только ссылку на исходный буфер. В результате два разных контекста JavaScript будут поддерживать свои собственные ссылки на один и тот же блок памяти. Каждый контекст свободен для изменения буфера, как это было бы с обычным ArrayBuffer. Такое поведение демонстрируется здесь:

MAIN.JS

```
const worker = new Worker('./worker.js');

// Создание буфера из одного байта
const sharedArrayBuffer = new SharedArrayBuffer(1);

// Создание представления для буфера из 1 байта
const view = new Uint8Array(sharedArrayBuffer);

// Назначение значения 1 родительским контекстом
view[0] = 1;

worker.onmessage = () => {
    console.log(`buffer value after worker modification: ${view[0]}`);
};

// Отправка ссылки sharedArrayBuffer
worker.postMessage(sharedArrayBuffer);

// buffer value before worker modification: 1
// buffer value after worker modification: 2
```

WORKER.JS

```
self.onmessage = ({data}) => {
    const view = new Uint8Array(data);

    console.log(`buffer value before worker modification: ${view[0]}`);

    // Назначение рабочим потоком нового значения совместно используемому буферу
    view[0] += 1;

    // Отправка назад пустого postMessage, чтобы указать,
    // что назначение завершено
    self.postMessage(null);
};
```

Разумеется, совместное использование блока памяти между двумя параллельными потоками создает риск возникновения гонки. Другими словами, экземпляр `SharedArrayBuffer` эффективно обрабатывается как энергозависимая память. Эта проблема демонстрируется в следующем примере:

MAIN.JS

```
// Создание пула рабочих потоков размером 4
const workers = [];
for (let i = 0; i < 4; ++i) {
    workers.push(new Worker('./worker.js'));
}

// Запись конечного значения после завершения работы последнего потока
let responseCount = 0;
for (const worker of workers) {
    worker.onmessage = () => {
        if (++responseCount == workers.length) {
            console.log(`Final buffer value: ${view[0]}`);
        }
    };
}

// Инициализация SharedArrayBuffer
const sharedArrayBuffer = new SharedArrayBuffer(4);
const view = new Uint32Array(sharedArrayBuffer);
view[0] = 1;

// Отправка SharedArrayBuffer каждому рабочему потоку
for (const worker of workers) {
    worker.postMessage(sharedArrayBuffer);
}

// (Ожидаемый результат — 4000001. Действительный результат выведет что-то вроде:)
// Final buffer value: 2145106
```

WORKER.JS

```
self.onmessage = ({data}) => {
    const view = new Uint32Array(data);

    // Выполнение 1000000 операций добавления
    for (let i = 0; i < 1E6; ++i) {
        view[0] += 1;
    }

    self.postMessage(null);
};
```

Здесь каждый рабочий поток выполняет 1 000 000 последовательных операций, которые читают данные из индекса совместно используемого массива, выполняют сложение и записывают это значение обратно в индекс массива. Состояние гонки возникает, когда операции чтения/записи в потоке чередуются. Например:

1. Рабочий поток A читает значение 1.
2. Рабочий поток B читает значение 1.
3. Рабочий поток A добавляет 1 и записывает 2 обратно в массив.

4. Рабочий поток B, все еще использующий устаревшее значение массива 1, записывает 2 обратно в массив.

Для решения этой проблемы глобальный объект `Atomics` позволяет рабочему потоку эффективно получить блокировку экземпляра `SharedArrayBuffer` и выполнить всю последовательность чтения/добавления/записи, прежде чем разрешить другому потоку выполнять какие-либо операции. Включение `Atomics.add()` в этот пример дает правильное окончательное значение:

MAIN.JS

```
// Создание пула рабочих потоков размером 4
const workers = [];
for (let i = 0; i < 4; ++i) {
    workers.push(new Worker('./worker.js'));
}

// Запись конечного значения после завершения работы последнего потока
let responseCount = 0;
for (const worker of workers) {
    worker.onmessage = () => {
        if (++responseCount == workers.length) {
            console.log(`Final buffer value: ${view[0]}`);
        }
    };
}

// Инициализация SharedArrayBuffer
const sharedArrayBuffer = new SharedArrayBuffer(4);
const view = new Uint32Array(sharedArrayBuffer);
view[0] = 1;

// Отправка SharedArrayBuffer каждому рабочему потоку
for (const worker of workers) {
    worker.postMessage(sharedArrayBuffer);
}

// (Ожидаемый результат — 4000001.)
// Final buffer value: 4000001
```

WORKER.JS

```
self.onmessage = ({data}) => {
    const view = new Uint32Array(data);

    // Выполнение 1000000 операций добавления
    for (let i = 0; i < 1E6; ++i) {
        Atomics.add(view, 0, 1);
    }

    self.postMessage(null);
};
```

ПРИМЕЧАНИЕ `SharedArrayBuffer` и Atomics API полностью описаны в главе 20 «API в JavaScript».

Пулы рабочих потоков

Поскольку запуск рабочего потока — довольно затратная операция, могут быть ситуации, когда более эффективно поддерживать в живых определенное количество рабочих потоков и при необходимости направлять им работу. Когда поток выполняет вычисления, он помечается как занятый и будет готов выполнить другое задание только после того, как он сообщит пулу, что снова доступен. Обычно это называется «пул потоков» или «пул рабочих потоков».

Определение идеального числа потоков в пуле не является точной наукой, но свойство `navigator.hardwareConcurrency` возвращает количество ядер, доступных в системе. Поскольку вы, скорее всего, не сможете определить многопоточность каждого ядра, лучше всего рассматривать это число как верхнюю границу размера пула.

Один сценарий, с которым можно столкнуться, включает в себя фиксированный набор рабочих потоков в пуле, выполняющих одну и ту же задачу, которая управляется небольшим набором входных параметров. Используя пул потоков для конкретной задачи, можно выделить фиксированное количество рабочих потоков и передать их параметры по требованию. Рабочий поток примет эти параметры, выполнит длительные вычисления и вернет значение в пул. В свою очередь, пул затем отправит потоку дополнительную работу для выполнения. Этот пример создаст относительно упрощенный пул потоков, но он покроет все основные требования для этой концепции.

Начните с определения `TaskWorker`, который расширяет класс `Worker`. У этого класса есть две задачи: отслеживать, занят он работой или нет, и управлять информацией и событиями, приходящими и выходящими из рабочего потока. Кроме того, задачи, переданные этому потоку, будут заключены в промис и будут соответствующим образом разрешаться/отклоняться. Класс может быть определен следующим образом:

```
class TaskWorker extends Worker {
    constructor(notifyAvailable, ...workerArgs) {
        super(...workerArgs);

        // Инициализация в качестве недоступного
        this.available = false;
        this.resolve = null;
        this.reject = null;

        // Пул потоков передаст обратный вызов, чтобы
        // рабочий поток мог сигнализировать о необходимости получения задач
        this.notifyAvailable = notifyAvailable;

        // Сценарий рабочего потока отправит postmessage о готовности
        // сразу после инициализации
        this.onmessage = () => this.setAvailable();
    }

    // Вызов пулом потоков для начала работы над задачами
    dispatch({ resolve, reject, postMessageArgs }) {
        this.available = false;
        this.onmessage = ({ data }) => {
```

```
        resolve(data);
        this.setAvailable();
    };

    this.onerror = (e) => {
        reject(e);
        this.setAvailable();
    };

    this.postMessage(...postMessageArgs);
 }

    setAvailable() {
        this.available = true;
        this.resolve = null;
        this.reject = null;
        this.notifyAvailable();
    }
}
```

Затем определению класса `WorkerPool` нужно использовать этот класс `TaskWorker`. Данный класс также должен поддерживать очередь задач, которые еще предстоит назначить рабочему потоку. Два события могут сигнализировать о необходимости отправки новой задачи: новая задача добавляется в очередь, или поток завершает задачу, и ему следует отправить другую. Класс может быть определен следующим образом:

```
class WorkerPool {
    constructor(poolSize, ...workerArgs) {
        this.taskQueue = [];
        this.workers = [];

        // Инициализация пула потоков
        for (let i = 0; i < poolSize; ++i) {
            this.workers.push(
                new TaskWorker(() => this.dispatchIfAvailable(), ...workerArgs));
            }
        }

        // Добавление задачи в очередь
        enqueue(...postMessageArgs) {
            return new Promise((resolve, reject) => {
                this.taskQueue.push({ resolve, reject, postMessageArgs });

                this.dispatchIfAvailable();
            });
        }

        // Отправка задачи следующему потоку, если он существует
        dispatchIfAvailable() {
            if (!this.taskQueue.length) {
                return;
            }
            for (const worker of this.workers) {
                if (worker.available) {
```

```
                    let a = this.taskQueue.shift();
                    worker.dispatch(a);
                    break;
                }
            }
        }

        // Завершение всех рабочих потоков
        close() {
            for (const worker of this.workers) {
                worker.terminate();
            }
        }
}
```

Определив эти два класса, можно легко отправлять задачи в пул потоков и выполнять их по мере доступности рабочих потоков. В этом примере предположим, что необходимо сложить 10 миллионов чисел с плавающей запятой. Чтобы сэкономить на затратах на передачу, будем использовать `SharedArrayBuffer`. Определение рабочего потока может выглядеть следующим образом:

```
self.onmessage = ({data}) => {
    let sum = 0;
    let view = new Float32Array(data.arrayBuffer)

    // Выполнение сложения
    for (let i = data.startIdx; i < data.endIdx; ++i) {
        // В Atomics нет необходимости при выполнении операций чтения
        sum += view[i];
    }

    // Отправка результата потоку
    self.postMessage(sum);
};

// Отправка сообщения TaskWorker чтобы сигнализировать о том, что поток
// готов получать задачи.
self.postMessage('ready');
```

С учетом всего этого код, использующий пулы потоков, может выглядеть следующим образом:

```
Class TaskWorker {
    ...
]

Class WorkerPool {
    ...
}

const totalFloats = 1E8;
const numTasks = 20;
const floatsPerTask = totalFloats / numTasks;
const numWorkers = 4;

// Создание пула
```

```
const pool = new WorkerPool(numWorkers, './worker.js');

// Заполнение массива числами с плавающей точкой
let arrayBuffer = new SharedArrayBuffer(4 * totalFloats);
let view = new Float32Array(arrayBuffer);
for (let i = 0; i < totalFloats; ++i) {
    view[i] = Math.random();
}

let partialSumPromises = [];
for (let i = 0; i < totalFloats; i += floatsPerTask) {
    partialSumPromises.push(
        pool.enqueue({
            startIdx: i,
            endIdx: i + floatsPerTask,
            arrayBuffer: arrayBuffer
        })
    );
}

// Выполнение сложения после завершения всех промисов
Promise.all(partialSumPromises)
    .then((partialSums) => partialSums.reduce((x, y) => x + y))
    .then(console.log);

// (В этом примере сумма должна быть приблизительно равна 1E8/2)
// 49997075.47203197
```

ПРИМЕЧАНИЕ Слепое внедрение распараллеливания не является универсальным улучшением. Настройка производительности для пулов потоков будет зависеть от того, что включает вычисление задачи и какое системное оборудование используется.

ОБЩИЕ РАБОЧИЕ ПОТОКИ

Общий рабочий поток, или *общий поток*, ведет себя как выделенный рабочий поток, но доступен через несколько доверенных контекстов выполнения. Например, две разные вкладки в одном источнике смогут получить доступ к одному рабочему потоку. `SharedWorker` и `Worker` имеют несколько разные интерфейсы обмена сообщениями, как внешние, так и внутренние.

Общий рабочий поток полезен в ситуациях, когда разработчик желает уменьшить вычислительные затраты, позволяя нескольким контекстам выполнения совместно использовать поток. Примером этого может быть один общий рабочий, управляющий веб-сокетом для отправки и получения сообщений для нескольких страниц одного и того же происхождения. Совместно используемые рабочие потоки также полезны, когда контексты одного и того же происхождения хотят общаться через совместно используемый поток.

Основы использования общих рабочих потоков

Говоря о поведении, общие рабочие потоки могут считаться продолжением выделенных потоков. Создание потока, опции потока, ограничения безопасности и `importScripts()` ведут себя одинаково. Как и в случае с выделенным рабочим потоком, общий поток также работает в отдельном контексте выполнения и может взаимодействовать только асинхронно с другими контекстами.

Создание общего рабочего потока

Как и в случае с выделенными рабочими потоками, наиболее распространенный способ создания общего потока — через загружаемый JS-файл. Путь к файлу предоставляется конструктору `SharedWorker`, который, в свою очередь, асинхронно загружает сценарий в фоновом режиме и создает экземпляр рабочего потока.

В следующем простом примере создается пустой общий поток по абсолютному пути:

EMPTYSHAREDWORKER.JS

```
  // empty JS worker file
```

MAIN.JS

```
  console.log(location.href);    // "https://example.com/"
  const sharedWorker = new SharedWorker(
      location.href + 'emptySharedWorker.js');
  console.log(sharedWorker);    // SharedWorker {}
```

Предыдущий пример может быть изменен для использования относительного пути; однако при этом требуется, чтобы `main.js` выполнялся по тому же пути, из которого можно загрузить `emptySharedWorker.js`:

```
const worker = new Worker('./emptyWorker.js');
console.log(worker);    // Worker {}
```

Общие рабочие потоки также могут быть созданы из встроенного сценария, но в этом нет особого смысла: каждому Blob-объекту, созданному из строки встроенного сценария, назначается собственный уникальный URL-адрес в браузере, поэтому общие потоки, созданные из встроенных сценариев, всегда будут уникальными. Причины этого раскрыты в следующем разделе.

Идентичность и единовременное владение SharedWorker

Важное различие между общим и выделенным рабочими потоками заключается в том, что, в то время как конструктор `Worker()` всегда создает новый экземпляр потока, конструктор `SharedWorker()` создает новый экземпляр только в том случае, если еще не существует такой же идентификатор. Если общий рабочий поток, соответствующий идентификатору, *существует*, будет создано новое соединение с существующим общим потоком.

Идентичность общего рабочего потока определяется на основе URL-адреса разрешенного сценария, имени потока и источника документа. Например, следующий скрипт будет создавать экземпляр одного общего рабочего потока и добавлять два последующих подключения:

```
// Создание одного экземпляра общего рабочего потока
// - Все конструкторы вызываются из одного источника
// - Все сценарии ссылаются на один URL
// - Все рабочие потоки имеют одинаковое имя
new SharedWorker('./sharedWorker.js');
new SharedWorker('./sharedWorker.js');
new SharedWorker('./sharedWorker.js');
```

Точно так же, поскольку все три из следующих строк сценария разрешаются по одному и тому же URL-адресу, создается только один общий рабочий поток:

```
// Создание одного экземпляра общего рабочего потока
// - Все конструкторы вызываются из одного источника
// - Все сценарии ссылаются на один URL
// - Все рабочие потоки имеют одинаковое имя
new SharedWorker('./sharedWorker.js');
new SharedWorker('sharedWorker.js');
new SharedWorker('https://www.example.com/sharedWorker.js');
```

Поскольку необязательное имя рабочего потока является частью его общей идентификации, использование разных имен потоков заставит браузер создать несколько общих потоков — один с именем «`foo`» и один с именем «`bar`», — даже если они имеют одинаковое происхождение и URL сценария:

```
// Создание одного экземпляра общего рабочего потока
// - Все конструкторы вызываются из одного источника
// - Все сценарии ссылаются на один URL
// - Один общий поток имеет имя 'foo', второй – 'bar'
new SharedWorker('./sharedWorker.js', {name: 'foo'});
new SharedWorker('./sharedWorker.js', {name: 'foo'});
new SharedWorker('./sharedWorker.js', {name: 'bar'});
```

Как видно из названия, общие рабочие потоки являются общими для вкладок, окон, фреймов или других потоков, работающих в одном источнике. Следовательно, следующий сценарий, запущенный на нескольких вкладках, будет создавать поток только при первом его запуске, и каждый последующий запуск будет подключаться к одному и тому же рабочему потоку:

```
// Создание одного экземпляра общего рабочего потока
// - Все конструкторы вызываются из одного источника
// - Все сценарии ссылаются на один URL
// - Все рабочие потоки имеют одинаковое имя
new SharedWorker('./sharedWorker.js');
```

Аспект сценария идентификации общего рабочего потока ограничен только URL-адресом, поэтому следующий код создаст два общих рабочих потока даже при загрузке одного и того же сценария:

```
// Создание одного экземпляра общего рабочего потока
// - Все конструкторы вызываются из одного источника
// - токен '?' делает URL различными
// - Все рабочие потоки имеют одинаковое имя
new SharedWorker('./sharedWorker.js');
new SharedWorker('./sharedWorker.js?');
```

Если бы этот сценарий запускался на двух разных вкладках, все равно было бы создано только два общих рабочих потока. Каждый конструктор будет проверять существование общего потока и просто подключится к нему, если он существует.

Использование объекта SharedWorker

Объект `SharedWorker`, возвращаемый конструктором `SharedWorker()`, используется в качестве единой точки связи с вновь созданным выделенным рабочим потоком. Он может использоваться для передачи информации между потоком и родительским контекстом через `MessagePort`, а также для отслеживания событий `error`, отправляемых выделенным потоком.

Объект `SharedWorker` поддерживает следующие свойства:

- **onerror** — может быть назначен обработчик событий, который будет вызываться всякий раз, когда `ErrorEvent` типа `error` всплывает от рабочего потока.
  - Это событие происходит, когда в рабочем потоке возникает ошибка.
  - Это событие также можно обработать с помощью `sharedWorker.addEventListener ('error', handler)`.
- **port** — выделенный `MessagePort` для связи с общим рабочим потоком.

SharedWorkerGlobalScope

Внутри общего потока глобальная область является экземпляром `SharedWorkerGlobalScope`. Он наследуется от `WorkerGlobalScope` и, следовательно, включает в себя все его свойства и методы. Как и в случае с выделенными потоками, общий рабочий поток может получить доступ к этой глобальной области действия через `self`.

`SharedWorkerGlobalScope` расширяет `WorkerGlobalScope` следующими свойствами и методами:

- **name** — необязательный идентификатор строки, который может быть предоставлен конструктору `SharedWorker`.
- **importScripts()** — используется для импорта произвольного количества сценариев в рабочий поток.
- **close()** — аналог `worker.terminate()`. Он используется для немедленного завершения рабочего потока. Потоку не предоставляется никакой возможности для очистки; сценарий внезапно завершается.

- **onconnect** — должен быть установлен в качестве обработчика, когда устанавливается новое подключение к общему рабочему потоку. События `connect` включают в себя массив `ports` экземпляров `MessagePort`, который можно использовать для отправки сообщений обратно в родительский контекст.
  - Событие `connect` наступает, когда с общим потоком устанавливается соединение через `worker.port.onmessage` или `worker.port.start()`.
  - Это событие также может быть обработано с использованием `sharedWorker.addEventListener ('connect', handler)`.

ПРИМЕЧАНИЕ В зависимости от реализации браузера вывод в `console` внутри `SharedWorker` может не происходить в представлении консоли браузера по умолчанию.

Жизненный цикл общих рабочих потоков

Жизненный цикл общих рабочих потоков имеет те же стадии и особенности, что и жизненный цикл выделенных потоков. Разница в том, что выделенный поток неразрывно связан с одной страницей, а общий будет сохраняться до тех пор, пока остается подключенный к нему контекст.

Рассмотрим следующий фрагмент, который создает выделенный рабочий поток при каждом запуске:

```
new Worker('./worker.js');
```

В следующей таблице подробно описано, что происходит, когда три вкладки, выделяющие рабочие потоки, открываются и закрываются последовательно.

СОБЫТИЕ	РЕЗУЛЬТАТ	КОЛИЧЕСТВО ПОТОКОВ ПОСЛЕ СОБЫТИЯ
Вкладка 1 запускает `main.js`	Порождение выделенного потока 1	1
Вкладка 2 запускает `main.js`	Порождение выделенного потока 2	2
Вкладка 3 запускает `main.js`	Порождение выделенного потока 3	3
Вкладка 1 закрывается	Завершение выделенного потока 1	2
Вкладка 2 закрывается	Завершение выделенного потока 2	1
Вкладка 3 закрывается	Завершение выделенного потока 3	0

Как показано в таблице, существует соответствие между количеством выполнений сценария, количеством открытых вкладок и количеством запущенных рабочих потоков. Далее рассмотрим следующий простой сценарий, который создает новый или подключается к существующему рабочему потоку при каждом запуске:

```
new SharedWorker('./sharedWorker.js');
```

В следующей таблице подробно описано, что происходит, когда три вкладки открываются и закрываются последовательно.

СОБЫТИЕ	РЕЗУЛЬТАТ	КОЛИЧЕСТВО ПОТОКОВ ПОСЛЕ СОБЫТИЯ
Вкладка 1 запускает `main.js`	Порождение общего потока 1	1
Вкладка 2 запускает `main.js`	Подключение к общему потоку 1	1
Вкладка 3 запускает `main.js`	Подключение к общему потоку 1	1
Вкладка 1 закрывается	Отключение от общего потока 1	1
Вкладка 2 закрывается	Отключение от общего потока 1	1
Вкладка 3 закрывается	Отключение от общего потока 1. Соединений больше не остается, поэтому рабочий поток 1 завершается	0

Как показано в этой таблице, последовательные вызовы `new SharedWorker()` на вкладках 2 и 3 подключатся к существующему рабочему потоку. Поскольку соединения добавляются и удаляются из потока, общее количество соединений отслеживается. Когда количество подключений становится равным нулю, рабочий поток завершается.

Важно отметить, что нет способа программно завершить общий рабочий поток. Вы заметите, что метод `terminate()` отсутствует в объекте `SharedWorker`. Кроме того, вызов `close()` для общего рабочего порта (обсуждается далее в этой главе) не вызовет завершение потока, даже если к потоку подключен только один порт.

«Соединение» `SharedWorker` не связано с состоянием соединения соответствующего `MessagePort` или `MessageChannel`. Как только соединение с общим рабочим потоком установлено, браузер отвечает за управление этим соединением. Установленное соединение будет сохраняться в течение всего срока жизни страницы, и только в том случае, если страница будет закрыта и не будет никаких дальнейших подключений к общему рабочему потоку, браузер решит прекратить работу этого потока.

Подключение к общему рабочему потоку

Событие подключения вызывается внутри общего рабочего потока каждый раз, когда вызывается конструктор `SharedWorker`, независимо от того, был ли создан новый рабочий поток. Это продемонстрировано в следующем примере, где конструктор вызывается внутри цикла:

SHAREDWORKER.JS

```
let i = 0;
self.onconnect = () => console.log(`connected ${++i} times`);
```

MAIN.JS

```
for (let i = 0; i < 5; ++i) {
    new SharedWorker('./sharedWorker.js');
}

// connected 1 times
// connected 2 times
// connected 3 times
// connected 4 times
// connected 5 times
```

После события `connect` конструктор `SharedWorker` неявно создает `MessageChannel` и передает владение `MessagePort`, уникальным для этого экземпляра `SharedWorker`. Этот `MessagePort` доступен внутри объекта события `connect` в виде массива `ports`. Поскольку событие `connect` будет когда-либо представлять только одно соединение, можно смело предполагать, что длина массива `ports` будет равна 1.

Ниже показано, как получить доступ к массиву `ports` события. Здесь `Set` используется для обеспечения отслеживания только уникальных экземпляров объекта:

SHAREDWORKER.JS

```
const connectedPorts = new Set();

self.onconnect = ({ports}) => {
    connectedPorts.add(ports[0]);

    console.log(`${connectedPorts.size} unique connected ports`);
};
```

MAIN.JS

```
for (let i = 0; i < 5; ++i) {
    new SharedWorker('./sharedWorker.js');
}

// 1 unique connected ports
// 2 unique connected ports
// 3 unique connected ports
// 4 unique connected ports
// 5 unique connected ports
```

Важно отметить, что совместно общие рабочие потоки ведут себя асимметрично с точки зрения настройки и завершений. Каждое новое соединение `SharedWorker` вызывает событие, но нет соответствующего события, когда экземпляр `SharedWorker` отключается (например, при закрытии страницы).

В предыдущем примере, когда страницы подключаются и отключаются от одного и того же общего потока, коллекция `connectedPorts` будет загрязнена мертвыми портами без возможности их идентификации. Одним из решений этой проблемы является отправка явного сообщения об удалении прямо перед тем, как страница будет уничтожена при событии `beforeunload`, и разрешение совместному рабочему потоку выполнить очистку.

СЛУЖЕБНЫЕ РАБОЧИЕ ПОТОКИ

Служебный рабочий поток — это тип рабочего потока, который ведет себя как прокси-сервер внутри браузера. Служебные потоки позволяют перехватывать исходящие запросы и кешировать ответ. Это позволяет веб-странице работать без подключения к сети, поскольку некоторые или все страницы могут потенциально обслуживаться из кеша служебного рабочего потока. Служебный поток также может использовать Notifications API, Push API, Background Sync API и Channel Messaging API.

Как и общие рабочие потоки, несколько страниц в одном домене будут взаимодействовать с единственным экземпляром служебного рабочего потока. Однако для включения таких функций, как Push API, служебные потоки также могут пережить закрытие соответствующей вкладки или браузера и ожидать получения push-события.

ПРИМЕЧАНИЕ Служебные рабочие потоки – это невероятно широкая тема, которая может заполнить почти всю книгу. Чтобы расширить понимание за пределы этой главы, подумайте о прохождении курса на Udacity «Оффлайн-приложения» (https://www.udacity.com/course/offline-web-applications--ud899). Кроме того, Mozilla поддерживает сайт с рецептами служебных потоков (https://serviceworke.rs/), который является отличным справочником по общим паттернам служебных потоков.

ПРИМЕЧАНИЕ Жизненный цикл служебного потока сильно зависит от количества открытых вкладок одного и того же источника (называемых «клиентами»), от того, получила ли страница событие навигации и изменился ли сценарий служебного рабочего потока (среди многих других факторов). Некоторые примеры в разделе «Служебные рабочие потоки» могут работать не так, как ожидалось, если вы плохо понимаете жизненный цикл служебных потоков. Раздел «Жизненный цикл служебного рабочего потока» проливает свет на то, что происходит под капотом.

Кроме того, будьте осторожны с использованием функции жесткого обновления браузера (Ctrl + Shift + R) при работе со служебными потоками. Жесткое обновление заставит браузер игнорировать все сетевые кеши, а служебный поток в большинстве браузеров считается сетевым кешем.

В конечном итоге большинство разработчиков обнаружат, что служебные потоки наиболее полезны для двух основных задач: выступать в качестве слоя кеширования для сетевых запросов и включать push-уведомления. В этом смысле служебный

рабочий поток — это инструмент, разработанный для того, чтобы веб-страницы могли вести себя как встроенные приложения.

Основы использования служебных рабочих потоков

Как класс рабочих потоков, служебный рабочий поток демонстрирует многие из тех же правил, что и выделенный или общий потоки. Он существует в совершенно отдельном контексте выполнения и может взаимодействовать с чем-либо только через асинхронный обмен сообщениями. Однако существует ряд принципиальных различий между служебным и выделенным/общим рабочими потоками.

ServiceWorkerContainer

Служебные рабочие потоки отличаются от выделенных и общих тем, что у них нет глобального конструктора. Вместо этого служебный поток управляется через `ServiceWorkerContainer`, доступный через `navigator.serviceWorker`. Этот объект является интерфейсом верхнего уровня, который позволяет указывать браузеру на создание, обновление, уничтожение или взаимодействовие со служебным рабочим потоком.

```
console.log(navigator.serviceWorker);
// ServiceWorkerContainer { ... }
```

Создание служебного рабочего потока

Служебные рабочие потоки похожи на общие в том, что если поток еще не существует, то будет создан новый экземпляр; в противном случае происходит соединение с существующим потоком. Вместо создания с помощью глобального конструктора `ServiceWorkerContainer` предоставляет метод `register()`, которому передается URL-адрес сценария таким же образом, как в конструкторы `Worker` или `SharedWorker`:

EMPTYSERVICEWORKER.JS

```
  // пустой сценарий служебного рабочего потока
```

MAIN.JS

```
  navigator.serviceWorker.register('./emptyServiceWorker.js');
```

Метод `register()` возвращает промис, который разрешается в объекте `ServiceWorkerRegistration` или отклоняется в случае сбоя регистрации.

EMPTYSERVICEWORKER.JS

```
  // пустой сценарий служебного рабочего потока
```

MAIN.JS

```
  // Успешная регистрация служебного потока, разрешение промиса
  navigator.serviceWorker.register('./emptyServiceWorker.js')
      .then(console.log, console.error);

  // ServiceWorkerRegistration { ... }
```

```
// Попытка зарегистрировать поток из несуществующего файла, отклонение промиса
navigator.serviceWorker.register('./doesNotExist.js')
    .then(console.log, console.error);

// TypeError: Failed to register a ServiceWorker:
// A bad HTTP response code (404) was received when fetching the script.
```

Природа служебных рабочих потоков предоставляет некоторую гибкость в отношении выбора, когда начинать регистрацию. Как только служебный поток активируется после первоначального `register()`, последующие вызовы метода `register()` на той же странице и с тем же URL-адресом фактически становятся запретными. Более того, даже несмотря на то, что служебные рабочие потоки не поддерживаются браузерами глобально, они должны быть фактически невидимыми для страницы, потому что их прокси-подобное поведение означает, что действия, которые в противном случае были бы обработаны, будут просто отправляться в сеть как обычно.

Из-за вышеупомянутых свойств чрезвычайно распространенный паттерн регистрации служебного рабочего потока заключается в том, чтобы скрыть его от обнаружения функций и события загрузки страницы. Это часто выглядит следующим образом:

```
if ('serviceWorker' in navigator) {
    window.addEventListener('load', () => {
        navigator.serviceWorker.register('./serviceWorker.js');
    });
}
```

Без обработки событий `load` регистрация служебного потока будет пересекаться с загрузкой ресурсов страницы, что может замедлить общую исходную отрисовку страницы. Если служебный рабочий поток не отвечает за управление поведением кеша, которое *должно* произойти как можно раньше в процессе настройки страницы (например, при использовании `clients.claim()`, обсуждаемого далее в этой главе), ожидание события `load` обычно целесообразно Выбор, который все еще позволяет странице пользоваться всеми преимуществами использования служебных потоков.

Использование объекта ServiceWorkerContainer

Интерфейс `ServiceWorkerContainer` — это оболочка верхнего уровня для экосистемы рабочего потока браузера. Он предоставляет средства для управления состоянием и жизненным циклом служебного потока.

`ServiceWorkerContainer` всегда доступен в контексте клиента:

```
console.log(navigator.serviceWorker);

// ServiceWorkerContainer { ... }
```

`ServiceWorkerContainer` поддерживает следующие обработчики событий:

- **oncontrollerchange** — может быть назначен обработчик событий, который будет вызываться каждый раз, когда событие `controllerchange` отправляется из `ServiceWorkerContainer`.

- Это событие происходит при получении новой активированной регистрации `ServiceWorkerRegistration`.
- Это событие также можно обработать с помощью `navigator.serviceWorker.addEventListener('controllerchange', handler)`.

- **onerror** — может быть назначен обработчик события, который будет вызываться всякий раз, когда `ErrorEvent` типа `error` всплывает от любого связанного служебного рабочего потока.
  - Это событие происходит, когда в любом связанном рабочем потоке возникает ошибка.
  - Это событие также можно обработать с помощью `navigator.serviceWorker.addEventListener ('error', handler)`.
- **onmessage** — может быть назначен обработчик события, который будет вызываться всякий раз, когда сообщение `ServiceEvent` типа отправляется от сервисного рабочего потока.
  - Это событие происходит, когда сценарий служебного рабочего потока отправляет событие сообщения обратно в родительский контекст.
  - Это событие также можно обработать с помощью `navigator.serviceWorker.addEventListener('message', handler)`.

`ServiceWorkerContainer` поддерживает следующие свойства:

- **ready** — возвращает промис, который может разрешиться с помощью активированного объекта `ServiceWorkerRegistration`. Этот промис никогда не будет отклонен.
- **controller** — возвращает активированный объект `ServiceWorker`, связанный с текущей страницей, или `null`, если активного служебного потока нет.

`ServiceWorkerContainer` поддерживает следующие методы:

- **register()** — создает или обновляет `ServiceWorkerRegistration`, используя предоставленный объект URL-адреса и параметров.
- **getRegistration()** — возвращает промис, который разрешается с помощью объекта `ServiceWorkerRegistration`, соответствующего предоставленной области, или разрешается с undefined, если нет соответствующего служебного рабочего потока.
- **getRegistration()** — возвращает промис, который разрешается с помощью массива всех объектов `ServiceWorkerRegistration`, связанных с `ServiceWorkerContainer`, или пустого массива, если нет связанных служебных потоков.
- **startMessage()** — запускает передачу сообщений через `Client.postMessage()`.

Использование объекта ServiceWorkerRegistration

Объект `ServiceWorkerRegistration` представляет успешную регистрацию служебного рабочего потока. Объект доступен в обработчике разрешенных промисов,

возвращаемом из `register()`. Этот объект позволяет определить статус жизненного цикла связанного рабочего потока, используя несколько свойств.

Объект регистрации предоставляется внутри промиса после вызова `navigator.serviceWorker.register()`. Несколько вызовов на одной странице с одним и тем же URL вернут один и тот же объект регистрации.

```
navigator.serviceWorker.register('./serviceWorker.js')
    .then((registrationA) => {
        console.log(registrationA);

navigator.serviceWorker.register('./serviceWorker2.js')
    .then((registrationB) => {
        console.log(registrationA === registrationB);
    });
});
```

`ServiceWorkerRegistration` поддерживает следующий обработчик событий:

- **onupdatefound** — может быть назначен обработчик событий, который будет вызываться всякий раз, когда событие типа `updatefound` запускается из служебного потока.
  - Это событие происходит, когда начинается установка новой версии этого рабочего потока, о чем свидетельствует `ServiceWorkerRegistration.installing` при получении нового служебного потока.
  - Это событие также можно обработать с помощью `serv serviceWorkerRegistration.addEventListener('updatefound', handler)`.

`ServiceWorkerRegistration` поддерживает следующие общие свойства:

- **scope** — возвращает полный путь URL-адреса области видимости служебного потока. Это значение является производным от пути, из которого был извлечен сценарий потока, и/или области, предоставленной `в register()`.
- **navigationPreload** — возвращает экземпляр `NavigationPreloadManager`, связанный с этим объектом регистрации.
- **pushManager** — возвращает экземпляр `PushManager`, связанный с этим объектом регистрации.

`ServiceWorkerRegistration` также поддерживает следующие свойства, которые можно использовать для проверки служебных рабочих потоков на различных этапах их жизненных циклов:

- **installing** — возвращает служебный поток с состоянием `installing`, если в данный момент он существует, иначе — `null`.
- **waiting** — возвращает рабочий поток с состоянием `waiting`, если оно есть в данный момент, иначе — `null`.
- **active** — возвращает служебный поток с состоянием `activating` или `active`, если оно есть в данный момент, иначе — `null`.

Обратите внимание, что эти свойства являются одноразовым захватом состояния служебного потока. Они подходят для большинства случаев использования, так как активный служебный поток не будет изменять состояние в течение всего срока жизни страницы, если его не заставят делать это с помощью чего-то вроде `ServiceWorkerGlobalScope.skipWaiting()`.

`ServiceWorkerRegistration` поддерживает следующие методы:

- **getNotifications()** — возвращает промис, который разрешается с помощью массива объектов `Notification`.
- **showNotifications()** — отображает уведомление, которое можно настроить с помощью аргументов `title` и `options`.
- **update()** — повторно запрашивает сценарий служебного рабочего потока непосредственно с сервера и запускает новую установку, если новый сценарий отличается от предыдущего.
- **unregister()** — попытается отменить регистрацию потока. Это позволяет завершить выполнение служебного потока перед отменой регистрации.

Использование объекта ServiceWorker

Объект `ServiceWorker` может быть получен одним из двух способов: через свойство контроллера на объекте `ServiceWorkerController` и активное свойство объекта `ServiceWorkerRegistration`. Этот объект наследуется от прототипа `Worker` и поэтому получает все его свойства и методы, но, в частности, не имеет метода `terminate()`.

`ServiceWorker` поддерживает следующий обработчик событий:

- **onstatechange** — может быть назначен обработчик событий, который будет вызываться всякий раз, когда событие `Servicechange` отправляется из `ServiceWorker`.
  - Это событие происходит при изменении `ServiceWorker.state`.
  - Это событие также может быть обработано с помощью `serviceWorker.addEventListener ('statechange', handler)`.

`ServiceWorker` поддерживает следующие свойства:

- **scriptURL** — разрешенный URL-адрес, используемый для регистрации служебного рабочего потока. Например, если поток был создан с относительным путем «`./serviceWorker.js`», то если он был зарегистрирован на https://www.example.com, свойство `scriptURL` вернет https://www.example.com/serviceWorker.js.
- **state** — возвращает строку, идентифицирующую состояние служебного потока. Возможны следующие состояния:
  - `installing`
  - `installed`
  - `activating`
  - `activated`
  - `redundant`

Ограничения безопасности служебного потока

Как класс рабочего потока, служебные потоки подчиняются обычным ограничениям в отношении соответствия происхождения загруженного сценария. (Подробнее об этом см. в разделе «Ограничения безопасности для рабочих потоков» ранее в этой главе.) Кроме того, поскольку служебным потокам предоставляется практически неограниченная возможность изменять и перенаправлять сетевые запросы и загруженные статические ресурсы, API служебного потока доступен только в безопасном `https` контексте; в контексте `http` `navigator.serviceWorker` будет иметь значение `undefined`. Для упрощения разработки браузеры делают исключение из правила безопасного контекста для страниц, загружаемых локально, через `localhost` или `127.0.0.1`.

> **ПРИМЕЧАНИЕ** Удобным инструментом для оценки того, является ли текущий контекст безопасным, является `window.isSecureContext`.

ServiceWorkerGlobalScope

Внутри служебного потока глобальная область является экземпляром `ServiceWorkerGlobalScope`. Она наследуется от `WorkerGlobalScope` и, следовательно, включает в себя все его свойства и методы. Служебный рабочий поток может получить доступ к этой глобальной области через `self`.

`ServiceWorkerGlobalScope` расширяет `WorkerGlobalScope` следующими свойствами и методами:

- **caches** — возвращает объект `CacheStorage` служебного потока.
- **clients** — возвращает интерфейс клиентов служебного потока. Используется для доступа к базовым объектам `Client`.
- **registration** — возвращает объект `ServiceWorkerRegistration` служебного потока.
- **skipWaiting()** — переводит служебный поток в активное состояние. Это используется вместе с `Clients.claim()`.
- **fetch()** — выполняет обычное извлечение из служебного потока. Это используется, когда служебный поток определяет, что должен быть сделан фактический исходящий сетевой запрос (вместо возврата кешированного значения).

В то время как выделенные или общие рабочие потоки имеют в качестве входных данных только событие `message`, служебные потоки могут использовать большое количество событий, которые инициируются действиями на странице, действиями по уведомлению или push-событиями.

> **ПРИМЕЧАНИЕ** В зависимости от реализации браузера запись в `console` внутри `SharedWorker` может не выводиться в представлении консоли браузера по умолчанию.

Глобальная область видимости служебного рабочего потока может прослушивать следующие события, разбитые здесь по категориям:

Состояние служебного потока

- **install** запускается, когда служебный поток входит в состояние *installing* (отображается на клиенте через `ServiceWorkerRegistration.installing`). Также можно установить обработчик для этого события в `self.oninstall`.
  - Это первое событие, полученное служебным потоком, и оно запускается, как только начинается выполнение потока.
  - Вызывается только один раз на поток.
- **activate** запускается, когда служебный поток входит в состояние *activating* или *activated* (отображается на клиенте через `ServiceWorkerRegistration.active`). Также можно установить обработчик для этого события в `self.onactivate`.
  - Это событие вызывается, когда служебный поток готов обрабатывать функциональные события и управлять клиентами.
  - Это событие *не* означает, что служебный поток контролирует клиента, только то, что он готов сделать это.

Fetch API

- **fetch** запускается, когда служебный поток перехватывает функцию `fetch()`, вызываемую на главной странице. Обработчик события `fetch` потока имеет доступ к `FetchEvent` и может корректировать результат по своему усмотрению. Также можно установить обработчик для этого события в `self.onfetch`.

Message API

- **message** запускается, когда служебный поток получает данные через `postMesssage()`. Также можно установить обработчик для этого события в `self.onmessage`.

Notification API

- **notificationclick** срабатывает, когда система сообщает браузеру, что было щелкнуто мышью уведомление, порожденное `ServiceWorkerRegistration.showNotification()`. Также можно установить обработчик для этого события на `self.onnotificationclick`.
- **notificationclose** запускается, когда система сообщает браузеру, что уведомление, порожденное `ServiceWorkerRegistration.showNotification()`, было закрыто или отклонено. Также можно установить обработчик для этого события в `self.onnotificationclose`.

Push API

- **push** запускается, когда служебный поток получает push-сообщение. Также можно установить обработчик для этого события в `self.onpush`.

- **pushsubscriptionchange** запускается при изменении состояния принудительной подписки, которое произошло вне контроля приложения (явно не в JavaScript). Также можно установить обработчик для этого события в `self.onpushsubscriptionchange`.

> **ПРИМЕЧАНИЕ** Некоторые браузеры также поддерживают событие sync, которое является частью Background Sync API. Этот API не стандартизирован и поддерживается только в Chrome и Opera, поэтому он не включен в эту книгу.

Ограничения области действия служебных потоков

Служебные потоки будут перехватывать только запросы от клиентов, которые находятся в их области действия. Область действия определяется относительно пути, по которому был обработан сценарий служебного потока. Если не указано иное внутри `register()`, область действия становится путем к сценарию служебного рабочего потока. (Все регистрации служебных потоков в этих примерах используют абсолютные URL-адреса для сценария, чтобы избежать путаницы с путями.) В этом первом примере демонстрируется корневая область по умолчанию для рабочего сценария, обрабатываемого из корневого пути:

```
navigator.serviceWorker.register('/serviceWorker.js')
    .then((serviceWorkerRegistration) => {
        console.log(serviceWorkerRegistration.scope);
        // https://example.com/
});

// Все перечисленное будет перехвачено:
// fetch('/foo.js');
// fetch('/foo/fooScript.js');
// fetch('/baz/bazScript.js');
```

В следующем примере демонстрируется область действия того же каталога для сценария рабочего потока, обрабатываемого из корневого пути:

```
navigator.serviceWorker.register('/serviceWorker.js', {scope: './'})
    .then((serviceWorkerRegistration) => {
        console.log(serviceWorkerRegistration.scope);
        // https://example.com/
});

// Все перечисленное будет перехвачено:
// fetch('/foo.js');
// fetch('/foo/fooScript.js');
// fetch('/baz/bazScript.js');
```

В следующем примере демонстрируется ограниченная область действия для сценария рабочего потока, обрабатываемого из корневого пути:

```
navigator.serviceWorker.register('/serviceWorker.js', {scope: './foo'})
    .then((serviceWorkerRegistration) => {
    console.log(serviceWorkerRegistration.scope);
    // https://example.com/foo/
});

// Все перечисленное будет перехвачено:
// fetch('/foo/fooScript.js');

// Все перечисленное не будет перехвачено:
// fetch('/foo.js');
// fetch('/baz/bazScript.js');
```

В следующем примере демонстрируется область действия того же каталога для сценария рабочего потока, обрабатываемого по вложенному пути:

```
navigator.serviceWorker.register('/foo/serviceWorker.js')
    .then((serviceWorkerRegistration) => {
        console.log(serviceWorkerRegistration.scope);
        // https://example.com/foo/
});

// Все перечисленное будет перехвачено:
// fetch('/foo/fooScript.js');

// Все перечисленное не будет перехвачено:
// fetch('/foo.js');
// fetch('/baz/bazScript.js');
```

Область действия служебного потока фактически следует модели разрешений каталогов в том смысле, что можно только уменьшить область действия относительно того, откуда был получен файл. Попытка расширить область следующим образом выдает ошибку:

```
navigator.serviceWorker.register('/foo/serviceWorker.js', {scope: '/'});

// Error: The path of the provided scope 'https://example.com/'
// is not under the max scope allowed 'https://example.com/foo/'
```

Как правило, область действия служебного потока будет определяться как абсолютный путь с косой чертой, как показано ниже:

```
navigator.serviceWorker.register('/serviceWorker.js', {scope: '/foo/'})
```

Этот стиль определения пути области действия выполняет две задачи: он отделяет относительный путь файла сценария от относительного пути области действия и предотвращает включение самого пути в область. Например, в предыдущем фрагменте кода, вероятно, нежелательно, чтобы путь `/foo` был включен в область действия служебного потока; добавление завершающего `/` явно исключит путь `/foo`. Конечно, это требует, чтобы абсолютный путь области действия не выходил за пределы пути служебного потока.

Если нужно расширить область действия служебного потока, есть два основных способа сделать это:

- Обрабатывать сценарий служебного потока по пути, который охватывает желаемую область действия.
- Добавить заголовок `Service-Worker-Allowed` в ответ сценария служебного потока со значением, установленным в желаемой области. Это значение области действия должно соответствовать значению области внутри `register()`.

Кеш служебного рабочего потока

До появления служебных рабочих потоков на веб-страницах отсутствовал надежный механизм кеширования сетевых запросов. Браузеры всегда использовали HTTP-кеш, но внутри него нет программного интерфейса JavaScript, и его поведение регулируются вне среды выполнения JavaScript. Можно было разработать специальный механизм кеширования, который кешировал бы строку ответа или большой двоичный объект, но такие стратегии были грязными и неэффективными.

Реализации JavaScript-кеша были опробованы ранее. Документы MDN описывают это удивительно:

> *Предыдущая попытка — AppCache — казалась хорошей идеей, поскольку она позволяла очень легко указывать ресурсы для кеширования. Тем не менее он выдавал много предположений о том, что вы пытались сделать, а затем ломался, когда приложение не следовало этим предположениям в точности.*

Одной из основных особенностей служебных потоков является настоящий механизм кеширования сетевых запросов, которым можно программно управлять. В отличие от кеша HTTP или кеша ЦП, кеш служебного потока довольно примитивен:

- **Кеш служебного потока не кеширует никаких запросов автоматически.** Все записи кеша должны быть добавлены явно.
- **Кеш служебного потока не имеет понятия истечения времени.** Запись в кеше останется в кеше, если только явно ее не удалить.
- **Записи кеша служебного потока должны обновляться и удаляться вручную.**
- **Кеши должны получать версии вручную.** Каждый раз, когда служебный поток обновляется, новый поток отвечает за предоставление нового ключа кеша для хранения новых записей кеша.
- **Единственная политика вытеснения, основанная на браузере, основана на хранилище, доступном для использования кешем служебного потока.** Служебный поток отвечает за управление объемом пространства, используемого его кешем. Когда размер кеша превышает ограничения браузера, браузер будет использовать *политику высвобождения* (LRU), которая используется не так давно, чтобы освободить место для новых записей в кеше.

По своей сути механизм кеширования служебных потоков представляет собой двухуровневый словарь, в котором каждая запись в словаре верхнего уровня сопоставляется со вторым вложенным словарем. Словарь верхнего уровня — это объект

CacheStorage, который доступен в глобальной области действия служебного потока через свойство caches. Каждое значение в этом словаре верхнего уровня — это объект Cache, который является словарем объектов Request, связанных с объектами Response.

Как и в случае LocalStorage, объекты Cache внутри CacheStorage сохраняются в течение неопределенного времени и сохранятся после окончания сеанса браузера. Кроме того, записи кеша доступны только для каждого конкретного источника.

> **ПРИМЕЧАНИЕ** Хотя объекты CacheStorage и Cache определены в спецификации Service Worker, они могут использоваться главной страницей или другими типами рабочих потоков.

Объект CacheStorage — это хранилище пар ключ–значение строковых ключей, сопоставленных с объектами Cache. Объект CacheStorage имеет API, напоминающий асинхронный Map. Интерфейс CacheStorage доступен для глобального объекта через его свойство caches.

```
console.log (caches); // CacheStorage {}
```

Отдельные кеши внутри CacheStorage извлекаются путем передачи их строкового ключа в caches.open(). Нестроковые ключи преобразуются в строку. Если кеш еще не существует, он будет создан.

Объект Cache возвращает промис:

```
caches.open('v1').then(console.log);

// Cache {}
```

Подобно Map, CacheStorage имеет методы has(), delete() и keys(). Все эти методы ведут себя как основанные на промисах их аналоги в Map:

CACHESTORAGEEXAMPLE01.JS

```
  // открытие нового кеша v1,
  // проверка кеша v1,
  // проверка несуществующего кеша v2

  caches.open('v1')
      .then(() => caches.has('v1'))
      .then(console.log)    // true
      .then(() => caches.has('v2'))
      .then(console.log);    // false
```

CACHESTORAGEEXAMPLE02.JS

```
  // открытие нового кеша v1,
  // проверка кеша v1,
  // удаление кеша v1,
  // еще одна проверка удаленного кеша v1

  caches.open('v1')
```

```
      .then(() => caches.has('v1'))
      .then(console.log)    // true
      .then(() => caches.delete('v1'))
      .then(() => caches.has('v1'))
      .then(console.log);    // false
```

CACHESTORAGEEXAMPLE03.JS

```
  // открытие кешей v1, v3 и v2
  // проверка ключей текущих кешей
  // ПРИМЕЧАНИЕ: ключи кешей выводятся в порядке создания

  caches.open('v1')
      .then(() => caches.open('v3'))
      .then(() => caches.open('v2'))
      .then(() => caches.keys())
      .then(console.log);    // ["v1", "v3", "v2"]
```

Интерфейс `CacheStorage` также имеет метод `match()`, который можно использовать для проверки объекта `Request` по всем объектам `Cache` в `CacheStorage`. Объекты `Cache` проверяются в `CacheStorage.keys()`, и первое совпадение возвращается в ответе:

```
CACHESTORAGEEXAMPLE04.JS
// Создание одного ключа запроса и двух значений ответа
const request = new Request('');
const response1 = new Response('v1');
const response2 = new Response('v2');

// Использование одинаковых ключей в обоих кешах. v1 будет найден первым, поскольку
// caches.keys() имеет приоритет
caches.open('v1')
    .then((v1cache) => v1cache.put(request, response1))
    .then(() => caches.open('v2'))
    .then((v2cache) => v2cache.put(request, response2))
    .then(() => caches.match(request))
    .then((response) => response.text())
    .then(console.log); // v1
```

`CacheStorage.match()` можно настроить с помощью объекта параметров. Этот объект подробно описан в следующем разделе «Объект Cache».

Объект Cache

`CacheStorage` соотносит строки с объектами `Cache`. Объекты `Cache` ведут себя подобно `CacheStorage` в том, что они также напоминают асинхронный `Map`. Ключи `Cache` могут быть либо строкой URL, либо объектом `Request`; эти ключи будут сопоставлены со значениями объекта `Response`.

Кеш служебного рабочего потока предназначен только для кеширования HTTP-запросов GET. Это должно иметь смысл: этот HTTP-метод подразумевает, что ответ не изменится со временем. С другой стороны, методы запроса, такие как POST, PUT и DELETE, по умолчанию запрещены `Cache`. Они подразумевают динамический обмен с сервером и, следовательно, не подходят для кеширования клиентом.

Для заполнения Cache существуют три метода:

- **put (request, response)** — используется, когда у вас уже есть оба ключа (объект Request или строка URL) и значение (объект Response) и вам нужно добавить запись в кеш. Этот метод возвращает промис, который разрешается, когда запись успешно добавляется в кеш.
- **add (request)** — используется, когда у вас есть только объект запроса или когда URL.add() отправит fetch() в сеть и кеширует ответ. Этот метод возвращает промис, который разрешается, когда запись успешно добавляется в кеш.
- **addAll(requests)** — используется, когда нужно выполнить массовое добавление в кеш по принципу «все или ничего», например начальное заполнение кеша при инициализации служебного потока. Метод принимает массив URL-адресов или объектов запроса. addAll() выполняет операцию add() для каждой записи в массиве запросов. Этот метод возвращает промис, которое разрешается только после успешного добавления каждой записи в кеш.

Как и в Map, в Cache есть методы delete() и keys(). Все эти методы ведут себя как основанные на промисах аналоги методов в Map:

```
const request1 = new Request('https://www.foo.com');
const response1 = new Response('fooResponse');

caches.open('v1')
    .then((cache) => {
        cache.put(request1, response1)
            .then(() => cache.keys())
            .then(console.log) // [Request]
            .then(() => cache.delete(request1))
            .then(() => cache.keys())
            .then(console.log); // []
});
```

Для проверки Cache существуют два метода:

- **matchAll(request, option)** — возвращает промис, который преобразуется в массив соответствующих объектов Response кеша.
  - Этот метод полезен в сценариях, где нужно выполнить массовое действие с аналогично организованными записями кеша, например удалить все кешированные значения в каталоге images.
  - Схема сопоставления запросов может быть настроена с помощью объекта параметров, описанного далее в этом разделе.
- **match(request, options)** — возвращает промис, который разрешается в соответствующий объект Response кеша или undefined, если нет попаданий в кеш.
  - Этот метод по сути эквивалентен matchAll(request, options)[0].
  - Схема сопоставления запросов может быть настроена с помощью объекта параметров, описанного далее в этом разделе.

Попадания в кеш определяются по соответствующим строкам URL и/или URL запросам. Строки URL и объекты `Request` являются взаимозаменяемыми, поскольку соответствие определяется путем извлечения URL-адреса объекта `Request`. Эта взаимозаменяемость демонстрируется здесь:

```
const request1 = 'https://www.foo.com';
const request2 = new Request('https://www.bar.com');

const response1 = new Response('fooResponse');
const response2 = new Response('barResponse');

caches.open('v1').then((cache) => {
    cache.put(request1, response1)
        .then(() => cache.put(request2, response2))
        .then(() => cache.match(new Request('https://www.foo.com')))
        .then((response) => response.text())
        .then(console.log)    // fooResponse
        .then(() => cache.match('https://www.bar.com'))
        .then((response) => response.text())
        .then(console.log);    // barResponse
});
```

Объект `Cache` использует метод `clone()` объектов `Request` и `Response` для создания дубликатов и сохранения их в виде пары ключ–значение. Это демонстрируется здесь, где извлеченные экземпляры не соответствуют исходной паре ключ–значение:

```
const request1 = new Request('https://www.foo.com');
const response1 = new Response('fooResponse');

caches.open('v1')
    .then((cache) => {
        cache.put(request1, response1)
            .then(() => cache.keys())
            .then((keys) => console.log(keys[0] === request1))    // false
            .then(() => cache.match(request1))
            .then((response) => console.log(response === response1));    // false
});
```

`Cache.match()`, `Cache.matchAll()` и `CacheStorage.matchAll()` поддерживают необязательный объект `options`, который позволяет настроить поведение соответствия URL, задав следующие свойства:

- **cacheName** — поддерживается только `CacheStorage.matchAll()`. Если задана строка, она будет соответствовать только значениям кеша внутри `Cache`, заданным указанной строкой.
- **ignoreSearch** — при значении `true` указывает сопоставителю URL игнорировать строки с вопросами как в запросах, так и в ключе кеша. Например, https://example.com?foo=bar и https://example.com будут совпадать.
- **ignoreMethod** — при значении `true` указывает сопоставителю URL игнорировать http-метод запроса. Рассмотрим следующий пример, где запрос POST может быть сопоставлен с GET:

```
const request1 = new Request('https://www.foo.com');
const response1 = new Response('fooResponse');

const postRequest1 = new Request('https://www.foo.com',
    { method: 'POST' });

caches.open('v1')
    .then((cache) => {
        cache.put(request1, response1)
            .then(() => cache.match(postRequest1))
            .then(console.log)    // undefined
            .then(() => cache.match(postRequest1, { ignoreMethod: true }))
            .then(console.log);    // Response {}
});
```

- **ignoreVary** — сопоставитель `Cache` учитывает HTTP-заголовок `Vary`, который указывает, какие заголовки запроса могут вызвать различия в ответе сервера. Когда `ignoreVary` имеет значение `true`, это позволяет URL-сопоставителю игнорировать заголовок `Vary` при сопоставлении.

```
const request1 = new Request('https://www.foo.com');
const response1 = new Response('fooResponse',
    { headers: {'Vary': 'Accept' }});

const acceptRequest1 = new Request('https://www.foo.com',
    { headers: { 'Accept': 'text/json' } });

caches.open('v1')
.then((cache) => {
    cache.put(request1, response1)
        .then(() => cache.match(acceptRequest1))
        .then(console.log)    // undefined
        .then(() => cache.match(acceptRequest1, { ignoreVary: true }))
        .then(console.log);    // Response {}
});
```

Максимальный объем кеш-памяти

Браузеры должны ограничивать объем хранилища, который может использовать любой данный кеш; в противном случае неограниченное хранилище, несомненно, будет предметом злоупотребления. Этот предел хранения не соответствует какой-либо формальной спецификации и полностью зависит от предпочтений конкретного поставщика браузера.

Используя API StorageEstimate, можно приблизительно определить, сколько места доступно (в байтах) и сколько в настоящее время используется. Этот метод доступен только в безопасном контексте браузера:

```
navigator.storage.estimate()
    .then(console.log);

// Вывод в вашем браузере может отличаться:
// { quota: 2147483648, usage: 590845 }
```

Согласно спецификации служебного рабочего потока:

> *Это не точные цифры; между сжатием, дедупликацией и запутыванием по соображениям безопасности они не будут совпадать.*

Клиенты служебных потоков

Служебный поток отслеживает связь с объектом `Client` окна, рабочего потока или служебного потока. Служебные потоки могут получить доступ к этим объектам `Client` через интерфейс `Clients`, доступный для глобального объекта через свойство `self.clients`.

Объект `Client` имеет следующие свойства и методы:

- **id** — возвращает универсально уникальный идентификатор для этого клиента, например `7e4248ec-b25e-4b33b15f-4af8bb0a3ac4`. Он может быть использован для получения ссылки на клиента через `Clients.get()`.
- **type** — возвращает тип клиента в виде строки. Его значением будет одно из: `window`, `worker` или `sharedworker`.
- **url** — возвращает URL клиента.
- **postMessage()** — позволяет отправлять целевые сообщения одному клиенту.

Интерфейс `Clients` позволяет получать доступ к объектам `Client` через `get()` и `matchAll()`, которые используют промис для возврата результатов. В `matchAll()` также может быть передан объект параметров, который поддерживает следующие свойства:

- **includeUncontrolled** — при значении `true` возвращает клиентов, которые еще не контролируются этим служебным потоком. По умолчанию имеет значение `false`.
- **type** — при значениях `window`, `worker` или `sharedworker` фильтры возвращают клиентов только этого типа. По умолчанию имеет значение `all`, который возвращает все типы клиентов.

Интерфейс `Clients` также предоставляет два метода:

- **openWindow(url)** — позволяет открывать новое окно по указанному URL-адресу, эффективно добавляя нового `Clients` к данному служебному потоку. Новый объект `Client` возвращается в разрешенном промисе. Этот метод полезен при нажатии на уведомление; служебный поток может обнаружить событие щелчка и открыть окно в ответ на этот щелчок.
- **claim()** — принудительно установит этот служебный поток для контроля всех клиентов в своей области действия. Это полезно, когда нет необходимости ждать перезагрузки страницы, чтобы служебный поток начал управлять страницей.

Служебные рабочие потоки и согласованность

Служебные рабочие потоки следует понимать сквозь призму их общего предназначения: позволить веб-страницам эмулировать поведение собственных приложений.

Поведение родного приложения требует, чтобы служебные потоки поддерживали *управление версиями*.

На высоком уровне управление версиями служебного потока обеспечивает *согласованность* между работой двух веб-страниц с одним источником в любой момент времени. Эта гарантия согласованности принимает две основные формы:

- **Согласованность кода** — веб-страницы создаются не из одного двоичного файла, такого как собственное приложение, а вместо этого из множества HTML, CSS, JavaScript, изображений, JSON и действительно любого типа файловых ресурсов, которые страницы могут загрузить. Веб-страницы обычно подвергаются постепенным обновлениям версий — для добавления или изменения поведения. Если веб-страница загружает в общей сложности 100 файлов, а загруженные ресурсы представляют собой сочетание версий 1 и 2, результирующее поведение совершенно непредсказуемо и, вероятно, неверно. Служебные потоки предоставляют механизм принудительного применения, гарантирующий, что все одновременно запущенные страницы одного и того же источника всегда создаются из ресурсов одной и той же версии.
- **Согласованность данных** — веб-страницы не являются герметичными приложениями. Они могут читать и записывать данные на локальном устройстве через различные API браузера, такие как `LocalStorage` или `IndexedDB`. Они также могут отправлять и получать данные на удаленные API. Формат чтения или записи данных может изменяться в зависимости от версии. Если одна страница записывает данные в формате версии 1, но вторая страница пытается прочитать данные в формате версии 2, результирующее поведение совершенно непредсказуемо и, вероятно, неверно. Механизм согласованности ресурсов служебного потока также гарантирует, что операции ввода-вывода веб-страниц ведут себя одинаково для всех одновременно работающих страниц одного источника.

Чтобы сохранить согласованность, жизненный цикл служебного потока идет очень долго, чтобы избежать достижения состояния, которое может нарушить эту согласованность. Например:

- **Служебные потоки рано выходят из строя.** При попытке установить рабочий служебный поток любая непредвиденная проблема помешает установке потока. Это включает в себя невозможность загрузки сценария, синтаксическую или временную ошибку в сценарии, невозможность загрузки зависимости потока с помощью `importScripts()` или загрузку даже одного ресурса кеша.
- **Служебные потоки агрессивно обновляются.** Когда браузер снова загружает сценарий служебного потока (вручную через `register()` или при перезагрузке страницы), браузеры начнут установку новой версии служебного потока, если есть хотя бы один байт разницы между сценарием рабочего потока *или* какими-либо зависимостями, загружаемыми через `importScripts()`.
- **Неактивные служебные потоки пассивно активируются.** Когда `register()` вызывается в первый раз на странице, служебный поток устанавливается, но не

активируется и начинает управлять страницей до окончания события навигации. Это должно иметь смысл: текущая страница предположительно уже имеет загруженные ресурсы, поэтому служебный поток не должен активироваться и начинать загрузку несовместимых активов.

- **Активные служебные потоки — липкие.** Пока существует хотя бы один клиент, связанный с активным служебным потоком, браузер будет продолжать использовать его для всех страниц этого источника. Браузер может начать установку нового экземпляра служебного потока, предназначенного для замены активного, но он не переключится на новый поток, пока активный клиент не будет контролировать 0 потоков (или пока служебный поток принудительно не обновится). Эта стратегия удаления служебных потоков не позволяет двум клиентам одновременно запускать две разные версии потоков.

Жизненный цикл служебных рабочих потоков

Спецификация служебного потока определяет шесть отдельных состояний, в которых он может существовать: *анализируемый*, *устанавливаемый*, *установленный*, *активируемый*, *активированный* и *избыточный*. Полный жизненный цикл служебного потока будет всегда посещать эти состояния в этом порядке, хотя не все состояния могут быть пройдены. Поток, который обнаружит ошибку во время установки или активации, перейдет в избыточное состояние.

Каждое изменение состояния инициирует событие `statechange` в объекте `ServiceWorker`. Обработчик может быть настроен на прослушивание этого события следующим образом:

```
navigator.serviceWorker.register('./serviceWorker.js')
    .then((registration) => {
        registration.installing.onstatechange = ({ target: { state } }) => {
            console.log('state changed to', state);
        };
    });
```

Состояние «анализируемый»

Вызов `navigator.serviceWorker.register()` инициирует процесс создания экземпляра служебного потока. Состояние «*анализируемый*» присваивается этому вновь созданному потоку. Данное состояние не имеет событий или значения `ServiceWorker.state`, связанных с ним.

> **ПРИМЕЧАНИЕ** Обратите внимание, что хотя «анализируемый» является формально определенным состоянием в спецификации Service Worker, `ServiceWorker.prototype.state` никогда не вернет значение `parsed` (проанализированный). Самое раннее состояние, которое может вернуть свойство, – это `installing` (устанавливаемый).

Браузер извлекает файл сценария и выполняет некоторые начальные задачи, чтобы начать жизненный цикл:

1. Убедиться, что сценарий служебного потока обрабатывается в том же домене.
2. Убедиться, что регистрация служебного потока происходит в безопасном контексте.
3. Убедиться, что сценарий служебного потока может быть успешно проанализирован интерпретатором JavaScript браузера без каких-либо ошибок.
4. Сделать захват сценария служебного потока. В следующий раз, когда браузер загрузит сценарий служебного потока, он будет сравнивать его с этим захватом и использовать его, чтобы решить, следует обновить поток или нет.

Если все это выполнится успешно, промис, возвращаемый функцией `register()`, разрешится с помощью объекта `ServiceWorkerRegistration`, а вновь созданный экземпляр служебного потока перейдет в состояние «*устанавливаемый*».

Состояние «устанавливаемый»

Состояние «*устанавливаемый*» — это то место, где должны выполняться все задачи по настройке служебного потока. Это включает в себя работу, которая должна выполняться до того, как служебный поток начнет контролировать страницу.

На клиенте этот этап можно определить, проверив, установлено ли свойство `ServiceWorkerRegistration.installing` в значение экземпляра `ServiceWorker`:

```
navigator.serviceWorker.register('./serviceWorker.js')
    .then((registration) => {
        if (registration.installing) {
            console.log('Service worker is in the installing state');
        }
    });
```

Связанный объект `ServiceWorkerRegistration` также будет запускать событие `updatefound` каждый раз, когда служебный поток достигнет этого состояния:

```
navigator.serviceWorker.register('./serviceWorker.js')
    .then((registration) => {
        registration.onupdatefound = () =>
            console.log('Service worker is in the installing state');
        };
    });
```

В служебном потоке этот этап можно определить, установив обработчик для события `install`:

```
self.oninstall = (installEvent) => {
    console.log('Service worker is in the installing state');
};
```

Состояние «*устанавливаемый*» часто используется для заполнения кеша служебного потока. Потоку может быть предложено оставаться в этом состоянии до тех пор, пока коллекция ресурсов не будет успешно кеширована. Если какой-либо из ресурсов не сможет кешироваться, служебный поток не сможет установить его и будет отправлен в состояние «*избыточный*».

Служебный поток может удерживаться в состоянии «*устанавливаемый*» с помощью `ExtendableEvent`. `InstallEvent` наследуется от `ExtendableEvent` и поэтому предоставляет API, который позволяет отложить переход состояния до разрешения промиса. Это достигается с помощью метода `ExtendableEvent.waitUntil()`. Предполагается, что этот метод передаст промис, который задержит переход к следующему состоянию, пока этот промис не разрешится. Например, следующий пример задержит переход в состояние «*установленный*» на пять секунд:

```
self.oninstall = (installEvent) => {
    installEvent.waitUntil(
        new Promise((resolve, reject) => setTimeout(resolve, 5000))
    );
};
```

Более прагматичным использованием этого метода будет кеширование группы активов с помощью `Cache.addAll()`:

```
const CACHE_KEY = 'v1';

self.oninstall = (installEvent) => {
    installEvent.waitUntil(
        caches.open(CACHE_KEY)
        .then((cache) => cache.addAll([
            'foo.js',
            'bar.html',
            'baz.css',
        ]))
    );
};
```

Если нет выданной ошибки или промис отклонен, служебный поток переходит в состояние «*установленный*».

Состояние «установленный»

Состояние «*установленный*», также называемое состоянием *ожидания*, указывает на то, что у служебного потока нет дополнительных задач по настройке и он готов взять на себя управление клиентами, как только ему это позволят. Если активного служебного потока нет, только что установленный служебный поток пропустит это состояние и перейдет непосредственно в состояние «*активируемый*», поскольку не будет причин ждать.

На клиенте этот этап можно определить, проверив, установлено ли свойство `ServiceWorkerRegistration.wait` в значение экземпляра `ServiceWorker`:

```
navigator.serviceWorker.register('./serviceWorker.js')
    .then((registration) => {
        if (registration.waiting) {
            console.log('Service worker is in the installing/waiting state');
        }
    });
```

Если активный служебный поток уже существует, состояние «установленный» может быть подходящим временем для запуска логики, которая переведет этот новый служебный поток в состояние «активируемый». Это можно сделать путем принудительного продвижения потока через `self.skipWaiting()` или путем перезагрузки приложения, тем самым позволяя браузеру органично продвигать служебный поток.

Состояние «активируемый»

Состояние «*активируемый*» указывает, что служебный поток был выбран браузером, чтобы стать потоком, который должен управлять страницей. Если в браузере нет действующего поддерживаемого служебного потока, этот новый поток автоматически перейдет в состояние «*активируемый*». Однако, если действующий активный поток существует, этот новый замещающий поток может достичь состояния «*активируемый*» следующими способами:

- Количество клиентов, контролируемых действующим служебным потоком, становится равным 0. Это часто происходит в момент закрытия всех контролируемых вкладок браузера. При следующем событии навигации новый служебный поток перейдет в состояние «*активируемый*».
- Установленный служебный поток вызывает `self.skipWaiting()`. Это вступает в силу немедленно и не требует ожидания события навигации.

Пока служебный поток находится в состоянии «*активируемый*», никакие функциональные события, такие как извлечение или отправка, не отправляются до тех пор, пока этот поток не достигнет состояния «*активированный*».

На клиенте этот этап можно частично идентифицировать, проверив, установлено ли свойство `ServiceWorkerRegistration.active` в значение экземпляра `ServiceWorker`:

```
navigator.serviceWorker.register('./serviceWorker.js')
    .then((registration) => {
        if (registration.active) {
            console.log('Service worker is in the activating/activated state');
        }
    });
```

Обратите внимание, что свойство `ServiceWorkerRegistration.active` указывает, что служебный поток находится либо в состоянии «*активируемый*», либо в состоянии «*активированный*».

У служебного потока этот этап можно определить, установив обработчик для события `activate`:

```
self.oninstall = (activateEvent) => {
    console.log('Service worker is in the activating state');
};
```

Событие `activate` указывает на то, что после старого служебного потока можно безопасно выполнять очистку, и это событие часто используется для очистки старых

данных кеша и переноса баз данных. Например, следующий пример удаляет все старые версии кеша:

```
const CACHE_KEY = 'v3';

self.oninstall = (activateEvent) => {
    caches.keys()
        .then((keys) => keys.filter((key) => key != CACHE_KEY))
        .then((oldKeys) => oldKeys.forEach((oldKey) => caches.delete(oldKey));
};
```

Событие `activate` также наследуется от `ExtendableEvent` и, следовательно, также поддерживает соглашение `waitUntil()` для задержки перехода в состояние «*активированный*» или перехода в состояние «*избыточный*» после отклонения промиса.

ПРИМЕЧАНИЕ Событие `active` в служебном потоке не означает, что этот поток контролирует клиентов.

Состояние «активированный»

Состояние «*активированный*» указывает, что служебный поток контролирует одного или нескольких клиентов. В этом состоянии поток будет захватывать события `fetch()` внутри своей области действия, а также события уведомлений.

На клиенте этот этап можно частично определить, проверив, установлено ли свойство `ServiceWorkerRegistration.active` в значение экземпляра `ServiceWorker`:

```
navigator.serviceWorker.register('./serviceWorker.js')
    .then((registration) => {
        if (registration.active) {
            console.log('Service worker is in the activating/activated state');
        }
    });
```

Обратите внимание, что свойство `ServiceWorkerRegistration.active` указывает, что служебный поток находится либо в состоянии «активируемый», либо в состоянии «*активированный*».

Превосходящим признаком того, что поток находится в состоянии «*активированный*», является проверка свойства контроллера `ServiceWorkerRegistration`. Она вернет активированный экземпляр `ServiceWorker`, который контролирует страницу:

```
navigator.serviceWorker.register('./serviceWorker.js')
    .then((registration) => {
        if (registration.controller) {
            console.log('Service worker is in the activated state');
        }
    });
```

Когда новый служебный поток получает контроль над клиентом, `ServiceWorkerContainer` в этом клиенте вызовет событие `controllerchange`:

```
navigator.serviceWorker.oncontrollerchange = () => {
    console.log('A new service worker is controlling this client');
};
```

Также можно использовать промис `ServiceWorkerContainer.ready` для обнаружения активного служебного потока. Этот промис с состоянием `ready` разрешается, когда на текущей странице есть активный поток:

```
navigator.serviceWorker.ready.then(() => {
    console.log('A new service worker is controlling this client');
});
```

Состояние «избыточный»

Состояние «*избыточный*» является кладбищем служебных потоков. Никакие события не будут переданы ему, и браузер может уничтожить его и освободить ресурсы.

Обновление служебного потока

Поскольку концепция управления версиями предназначена для служебных потоков, ожидается, что они периодически меняются. Таким образом, служебные потоки имеют надежный и сложный процесс обновления для безопасной замены устаревшего активного потока.

Этот процесс начинается с проверки обновлений, когда браузер повторно запрашивает сервис рабочего потока. Проверка обновления может быть инициирована следующими событиями:

- `navigator.serviceWorker.register()` вызывается со строкой URL, отличной от текущего активного служебного потока.
- Браузер переходит на страницу внутри области служебного потока.
- Произошло такое функциональное событие, как извлечение или отправка, и проверка обновления не выполнялась в течение как минимум 24 часов.

Сценарий свежеприготовленного служебного потока сравнивается со сценарием действующего потока. Если они не идентичны, браузер инициализирует новый служебный поток с новым сценарием. Обновленный поток будет продолжать свой жизненный цикл, пока не достигнет состояния «*установленный*». После этого обновленный служебный поток будет ждать, пока браузер решит, что он может безопасно получить контроль над страницей (или пока пользователь не заставит его взять управление страницей на себя).

Важно отметить, что обновление страницы не позволит обновленному служебному потоку активировать и заменить действующей поток. Рассмотрим сценарий, в котором открыта одна страница с управляющим работающим служебным потоком и обновленным потоком, ожидающим в состоянии «*установленный*». Клиенты

перекрываются во время обновления страницы, то есть новая страница загружается до того, как старая страница уничтожается, и, следовательно, работающий служебный поток никогда не отказывается от контроля, поскольку он по-прежнему контролирует ненулевое число клиентов. Из-за этого закрытие всех контролируемых страниц является единственным способом, позволяющим заменить действующий служебный поток.

Инверсия контроля и постоянство служебных потоков

Принимая во внимание, что выделенные и общие рабочие потоки разработаны для сохранения состояния, служебные рабочие потоки разработаны так, чтобы не иметь состояния. Более конкретно: служебные потоки следуют паттерну инверсии контроля (IOC) и созданы для управления событиями.

Основным следствием этого является то, что служебные потоки не должны полагаться на глобальное состояние рабочего потока. Почти весь код внутри служебного потока должен быть определен внутри обработчиков событий — заметным исключением являются глобальные константы, такие как версия потока. Число выполнений сценария служебного потока сильно варьируется и сильно зависит от состояния браузера, поэтому поведение сценария потока должно быть идемпотентным.

Важно понимать, что срок службы потока не связан со временем жизни клиентов, к которым он подключен. Большинство браузеров реализуют служебные потоки как отдельный процесс, и этот процесс независимо управляется браузером. Если браузер обнаруживает, что поток бездействует, он может завершить работу и снова запустить его при необходимости. Это означает, что, хотя вы можете полагаться на служебный поток для обработки событий после активации, нельзя полагаться на постоянство глобального состояния потока.

Управление кешированием файлов служебных потоков с помощью updateViaCache

Обычно все ресурсы JavaScript, загружаемые браузером, подчиняются HTTP-кешу браузера, как определено их заголовком `Cache-Control`. Поскольку сценарии служебных потоков не имеют преимущественной обработки, браузер не будет получать обновления сценариев потока, пока не истечет срок хранения кешированного файла.

Чтобы распространять обновления служебного потока как можно быстрее, обычным решением является обслуживание сценариев потока с заголовком `Cache-Control: max-age=0`. При этом браузер всегда будет загружать самый последний файл сценария.

Это решение с мгновенным истечением работает хорошо, но выбор полагаться только на HTTP-заголовки для определения поведения служебного потока означает, что только сервер решает, как клиент должен обновляться. Чтобы разрешить средствам клиента контролировать его поведение при обновлении, существует свойство `updateViaCache`, позволяющее контролировать, как клиент должен обрабатывать

сценарии служебных потоков. Это свойство может быть определено при регистрации потока, и оно принимает три строковых значения:

- `import`: значение по умолчанию. Файл сценария служебного потока верхнего уровня никогда не будет кешироваться, но файлы, импортированные внутри потока с помощью `importScripts()`, по-прежнему будут подчиняться HTTP-кешу и заголовку `Cache-Control`.
- `all`: никакие сценарии служебных потоков не получают особого отношения. Все файлы подчиняются HTTP-кешу и заголовку `Cache-Control`.
- `none`: и сценарий служебного потока верхнего уровня, и файлы, импортированные внутри потока через `importScripts()`, никогда не будут кешироваться.

Свойство `updateViaCache` используется следующим образом:

```
navigator.serviceWorker.register('/serviceWorker.js', {
    updateViaCache: 'none'
});
```

Браузеры все еще находятся в процессе перехода к поддержке этой опции, поэтому настоятельно рекомендуется использовать как `updateViaCache`, так и заголовок `Cache-Control` для определения поведения кеширования на клиенте.

Принудительный запуск операций в служебном потоке

В некоторых случаях имеет смысл как можно быстрее перевести служебный рабочий поток в состояние «*активированный*» — даже за счет потенциальных конфликтов версий активов. Обычно это происходит в форме кеширования ресурсов при событии установки, заставляющего служебный поток активироваться, а затем заставляющего активированный служебный поток контролировать связанных клиентов.

Обычно это выглядит следующим образом:

```
const CACHE_KEY = 'v1';

self.oninstall = (installEvent) => {
    // Заполнение кеша, затем принудительный перевод служебного потока
    // в состояние "активированный". Это вызовет событие активации.
    installEvent.waitUntil(
        caches.open(CACHE_KEY)
            .then((cache) => cache.addAll([
                'foo.css',
                'bar.js',
            ]))
            .then(() => self.skipWaiting())
        );
    };

// Принуждение служебного потока к контролированию клиентов. Это вызовет
// событие controllerchange на каждом клиенте.
self.onactivate = (activateEvent) => clients.claim();
```

Браузеры будут проверять наличие нового сценария служебного потока при каждом событии навигации, но иногда это происходит слишком редко. Объект `ServiceWorkerRegistration` имеет метод `update()`, который можно использовать для указания браузеру повторно запросить сценарий служебного потока, сравнить его с существующим и, если необходимо, начать установку обновленного потока. Это может быть достигнуто следующим образом:

```
navigator.serviceWorker.register('./serviceWorker.js')
    .then((registration) => {
        // Проверка обновленной версии каждые ~17 минут
        setInterval(() => registration.update(), 1E6);
    });
```

Обмен сообщениями в служебном потоке

Как и в случае с выделенными и общими рабочими потоками, служебные потоки могут обмениваться асинхронными сообщениями с клиентами с помощью `postMessage()`. Один из самых простых способов сделать это — отправить сообщение активному потоку и использовать объект события для отправки ответа. Сообщения, отправленные служебному потоку, могут быть обработаны в глобальной области действия, тогда как сообщения, отправленные обратно клиенту, могут быть обработаны в объекте `ServiceWorkerContext`:

SERVICEWORKER.JS

```
  self.onmessage = ({data, source}) => {
      console.log('service worker heard:', data);

      source.postMessage('bar');
  };
```

MAIN.JS

```
  navigator.serviceWorker.onmessage = ({data}) => {
      console.log('client heard:', data);
  };

  navigator.serviceWorker.register('./serviceWorker.js')
      .then((registration) => {
          if (registration.active) {
              registration.active.postMessage('foo');
          }
  });

  // service worker heard: foo
  // client heard: bar
```

Так же легко можно использовать свойство `serviceWorker.controller`:

SERVICEWORKER.JS

```
  self.onmessage = ({data, source}) => {
      console.log('service worker heard:', data);

      source.postMessage('bar');
```

```
};
```

MAIN.JS

```
navigator.serviceWorker.onmessage = ({data}) => {
    console.log('client heard:', data);
};

navigator.serviceWorker.register('./serviceWorker.js')
.then(() => {
    if (navigator.serviceWorker.controller) {
        navigator.serviceWorker.controller.postMessage('foo');
    }
});

// service worker heard: foo
// client heard: bar
```

Предыдущие примеры будут работать каждый раз при перезагрузке страницы, так как служебный поток будет отвечать на новое сообщение, отправленное клиентским сценарием после каждой перезагрузки, что также сработает каждый раз при открытии этой страницы в новой вкладке.

Если вместо этого служебный поток должен инициировать обмен сообщениями, ссылку на клиента можно получить следующим образом:

SERVICEWORKER.JS

```
self.onmessage = ({data}) => {
    console.log('service worker heard:', data);
};

self.onactivate = () => {
    self.clients.matchAll({includeUncontrolled: true})
    .then((clientMatches) => clientMatches[0].postMessage('foo'));
};
```

MAIN.JS

```
navigator.serviceWorker.onmessage = ({data, source}) => {
    console.log('client heard:', data);

    source.postMessage('bar');
};

navigator.serviceWorker.register('./serviceWorker.js')

// client heard: foo
// service worker heard: bar
```

Предыдущий пример сработает только один раз, поскольку активное событие вызывается только один раз для каждого служебного потока.

Поскольку клиенты и служебные потоки могут отправлять сообщения туда и обратно, также можно настроить `MessageChannel` или `BroadcastChannel` для обмена сообщениями.

Перехват события извлечения

Одной из наиболее важных особенностей служебных потоков является их способность перехватывать сетевые запросы. Сетевой запрос внутри области действия служебного потока будет зарегистрирован как событие извлечения. Эта способность перехвата не ограничивается методом `fetch()`; он также может перехватывать запросы на JavaScript, CSS, изображения и HTML, включая сам основной документ HTML. Эти запросы могут исходить от JavaScript, или они могут быть запросами, созданными тегами, такими как теги `<script>`, `<link>` или `<img>`. Интуитивно понятно, что это имеет смысл: чтобы поток эмулировал автономное веб-приложение, он должен иметь возможность учитывать все запрошенные ресурсы, необходимые для правильной работы страницы.

`FetchEvent` наследуется от `ExtendableEvent`. Ценный метод, который позволяет служебным потокам решать, как обрабатывать событие извлечения — `event.respondWith()`. Этот метод ожидает промис, который должен разрешаться с помощью объекта `Response`. Конечно, служебный поток должен решить, откуда на самом деле этот объект `Response` был получен. Он может быть получен из сети, из кеша или создан на лету. В следующих разделах рассматривается несколько стратегий работы с сетью/кешем внутри служебного потока.

Возврат из сети

Эта стратегия — простое прохождение для события извлечения. Хорошим вариантом использования для этого могут быть любые запросы, которые обязательно должны достигнуть сервера, например запрос POST. Данная стратегия может быть реализована следующим образом:

```
self.onfetch = (fetchEvent) => {
    fetchEvent.respondWith(fetch(fetchEvent.requeest));
};
```

ПРИМЕЧАНИЕ Предыдущий код предназначен только для демонстрации того, как использовать `event.respondWith()`. Если `event.respondWith()` не вызывается, браузер отправит запрос в сеть.

Возврат из кеша

Эта стратегия — простая проверка кеша. Хорошим вариантом использования для нее могут быть любые запросы, которые гарантированно находятся в кеше, например ресурсы, кешированные на этапе установки.

```
self.onfetch = (fetchEvent) => {
    fetchEvent.respondWith(caches.match(fetchEvent.request));
};
```

Возврат из сети с резервированием кеша

Эта стратегия отдает предпочтение современным ответам из сети, но все равно будет возвращать значения в кеше, если они существуют. Хороший пример — это когда приложению нужно показывать самую актуальную информацию как можно чаще, но все же хотелось бы показывать что-то, если приложение находится в автономном режиме.

```
self.onfetch = (fetchEvent) => {
    fetchEvent.respondWith(
        fetch(fetchEvent.request)
        .catch(() => caches.match(fetchEvent.request))
    );
};
```

Возврат из кеша с резервированием сети

Эта стратегия отдает предпочтение ответам, которые она может показывать быстрее, но, если значение не кешировано, ответ все равно будет извлекаться из сети. Это превосходная стратегия обработки выборки для большинства прогрессивных веб-приложений.

```
self.onfetch = (fetchEvent) => {
    fetchEvent.respondWith(
        caches.match(fetchEvent.request)
        .then((response) => response || fetch(fetchEvent.request))
    );
};
```

Генерация на лету

Приложения должны учитывать сценарии, когда и кеш, и сеть не могут создать ресурс. Служебные потоки могут справиться с этим, кешируя резервные ресурсы при установке и возвращая их при сбое как кеша, так и сети.

```
self.onfetch = (fetchEvent) => {
    fetchEvent.respondWith(
        // Начнем со стратегии "Возврат из кеша с резервированием сети"
        caches.match(fetchEvent.request)
        .then((response) => response || fetch(fetchEvent.request))
        .catch(() => caches.match('/fallback.html'))
    );
};
```

Предложение `catch()` может быть расширено для поддержки множества различных типов резервирования, таких как изображения-заполнители, фиктивные данные и т. д.

ПРИМЕЧАНИЕ На сайте разработчиков Google есть потрясающая статья о стратегии резервирования сети/кеша: https://web.dev/offline-cookbook/.

Всплывающие уведомления

Чтобы веб-приложение правильно имитировало родное приложение, оно должно поддерживать push-уведомления. Это означает, что веб-страница должна иметь возможность получать push-событие от сервера и отображать уведомление на устройстве, даже если приложение не запущено. На обычных веб-страницах это, конечно, невозможно, но добавление служебных потоков означает, что это поведение теперь поддерживается.

Чтобы push-уведомления работали в прогрессивном веб-приложении, должны поддерживаться четыре поведенческих аспекта:

- Служебный поток должен иметь возможность отображать уведомления.
- Служебный поток должен иметь возможность обрабатывать взаимодействия с этими уведомлениями.
- Служебный поток должен иметь возможность подписываться на отправленные сервером push-уведомления.
- Служебный поток должен уметь обрабатывать push-сообщения, даже если приложение не находится на переднем плане или не открыто.

Отображение уведомлений

Служебные потоки имеют доступ к Notification API через свой объект регистрации. Для этого есть веская причина: уведомления, связанные со служебным потоком, также будут вызывать события взаимодействия внутри этого потока.

Отображение уведомлений требует явного разрешения от пользователя. Как только оно будет предоставлено, уведомления могут быть показаны через `ServiceWorkerRegistration.showNotification()`. Это можно сделать следующим образом:

```
navigator.serviceWorker.register('./serviceWorker.js')
.then((registration) => {
    Notification.requestPermission()
    .then((status) => {
        if (status === 'granted') {
            registration.showNotification('foo');
        }
    });
});
```

Точно так же уведомление может быть запущено изнутри служебного потока, используя глобальное свойство `registration`:

```
self.onactivate = () => self.registration.showNotification('bar');
```

В этих примерах после предоставления разрешений на уведомление в браузере отображается уведомление `foo`. Это уведомление будет визуально неотличимо от уведомления, генерируемого с помощью `new Notification()`. Кроме того, не требуется,

чтобы служебный поток выполнял какую-либо работу для его появления. Он вступает в игру только когда требуются уведомления.

Обработка событий уведомлений

Уведомление, созданное с помощью объекта `ServiceWorkerRegistration`, будет отправлять события уведомления о клике и закрытии служебному потоку. Предположим, что сценарий служебного потока в предыдущем примере был определен следующим образом:

```
self.onnotificationclick = ({notification}) => {
    console.log('notification click', notification);
};

self.onnotificationclose = ({notification}) => {
    console.log('notification close', notification);
};
```

В этом примере оба типа взаимодействий с уведомлением будут регистрироваться внутри служебного потока. `notificationevent` предоставляет свойство `notification`, содержащее объект `Notification`, который сгенерировал событие. Эти обработчики событий могут решить, что делать после взаимодействия.

Часто нажатие на уведомление означает, что пользователь желает перейти в определенный режим просмотра. Внутри обработчика служебного потока это может быть выполнено с помощью метода `clients.openWindow()`, показанного здесь:

```
self.onnotificationclick = ({notification}) => {
    clients.openWindow('https://foo.com');
};
```

Подписка на push-события

Чтобы push-сообщения отправлялись служебному потоку, подписка должна происходить через `PushManager` потока. Это позволит потоку обрабатывать push-сообщения в обработчике push-событий.

Подписка может быть произведена с использованием `ServiceWorkerRegistration.pushManager`, как показано здесь:

```
navigator.serviceWorker.register('./serviceWorker.js')
.then((registration) => {
    registration.pushManager.subscribe({
        applicationServerKey: key, // получен из открытого ключа сервера
        userVisibleOnly: true
    });
});
```

С другой стороны, служебный поток может подписаться, используя глобальное свойство `registration`:

```
self.onactivate = () => {
    self.registration.pushManager.subscribe({
```

```
        applicationServerKey: key, // derived from server's public key
        userVisibleOnly: true
    });
};
```

Обработка push-событий

После подписки служебный поток будет получать push-события каждый раз, когда сервер отправляет сообщение. Они могут быть обработаны следующим образом:

```
self.onpush = (pushEvent) => {
    console.log('Service worker was pushed data:', pushEvent.data.text());
};
```

Чтобы реализовать настоящее push-уведомление, этот обработчик должен только создать уведомление через объект регистрации. Тем не менее правильно работающее push-уведомление должно поддерживать служебный поток, который создал его, достаточно долго для обработки события взаимодействия.

Для этого событие `push` наследуется от `ExtendableEvent`. Промис, возвращаемый из `showNotification()`, может быть передан в функцию `waitUntil()`, которая будет поддерживать служебный поток до разрешения промиса уведомления.

Простая реализация push-уведомлений может выглядеть следующим образом:

MAIN.JS

```
navigator.serviceWorker.register('./serviceWorker.js')
.then((registration) => {
    // Запрос разрешения на показ уведомлений
    Notification.requestPermission()
        .then((status) => {
            if (status === 'granted') {
                // Подписка на push-уведомления
                // только при наличии разрешения на показ уведомлений
                registration.pushManager.subscribe({
                applicationServerKey: key, // получен из открытого ключа сервера
                userVisibleOnly: true
            });
        }
    });
});
```

SERVICEWORKER.JS

```
// При получении push-уведомления данные показываются в текстовом формате
// внутри уведомления.
self.onpush = (pushEvent) => {
    // Поддержка служебного потока после разрешения промиса уведомления
    pushEvent.waitUntil(
        self.registration.showNotification(pushEvent.data.text())
    );
};

// При нажатии на уведомление открывается соответствующая страница приложения
self.onnotificationclick = ({notification}) => {
    clients.openWindow('https://example.com/clicked-notification');
};
```

ИТОГИ

Рабочие потоки позволяют запускать асинхронный JavaScript, который не будет блокировать пользовательский интерфейс. Это очень полезно для сложных вычислений и обработки данных, которые в противном случае занимали бы много времени и мешали бы пользователю использовать страницу. Рабочим потокам всегда предоставляется собственная среда выполнения, и с ними можно общаться только через асинхронный обмен сообщениями.

Рабочие потоки могут быть выделенными, что означает, что они связаны с одной страницей, или общими, что означает, что любая страница с одним источником может установить соединение с одним рабочим потоком.

Служебные рабочие потоки предназначены для того, чтобы веб-страницы могли вести себя как родные приложения. Служебные потоки — это тоже тип рабочих потоков, но они ведут себя скорее как сетевой прокси, а не как отдельный поток в браузере. Они могут вести себя как настраиваемый сетевой кеш, а также могут включать push-уведомления для прогрессивных веб-приложений.

28

Лучшие практики

- Удобство сопровождения кода
- Повышение быстродействия кода
- Развертывание кода

Загрузки с сайта wrox.com для этой главы

Обратите внимание, что все примеры кода для этой главы доступны на веб-сайте книги по ссылке www.wrox.com/go/projavascript4e на вкладке Downloads.

Начиная с 2000 г., разработка веб-приложений развивалась стремительными темпами. Область, которая была виртуальным Диким Западом, где можно было практически все, стала полноценной дисциплиной с общепризнанными оптимальными практиками программирования, основанными на надежных исследованиях. По мере трансформации простых веб-сайтов в сложные веб-приложения и превращения энтузиастов в высокооплачиваемых профессионалов копилось множество данных о том, какие приемы и подходы работают, а какие нет. Исследования и гипотезы очень позитивно сказались на развитии JavaScript, и теперь нам известно множество наилучших практик программирования веб-приложений.

УДОБСТВО СОПРОВОЖДЕНИЯ КОДА

На ранних веб-сайтах JS-код использовался преимущественно для создания несложных эффектов и проверки форм. Современные веб-приложения содержат тысячи строк JS-кода, управляющего разнообразными сложными процессами. Это требует, чтобы разработчики при написании кода учитывали удобство его сопровождения в будущем. Они приносят прибыль своим компаниям и делают это, не

просто создавая программные продукты вовремя, но и наращивая объемы интеллектуальной собственности, которая продолжает приносить прибыль длительное время после выпуска продукта.

Написание поддерживаемого кода важно, поскольку большинство разработчиков тратят много времени на поддержку чужого кода. Новые приложения редко разрабатывают «с нуля» — гораздо чаще за основу берутся уже имеющиеся компоненты. Удобство сопровождения кода помогает другим разработчикам делать свою работу как можно лучше.

ПРИМЕЧАНИЕ Концепция удобства сопровождения кода не уникальна для JavaScript. Многие ее аспекты относятся ко всем языкам программирования, хотя есть и специфичные для JavaScript.

Какой код удобно сопровождать?

Удобство сопровождения кода включает несколько аспектов. Говорят, что код удобно сопровождать, если он:

- **понятный** — цель написания кода и использованный при этом подход можно понять без пояснений автора кода;
- **интуитивный** — смысл кода ясен независимо от его сложности;
- **адаптируемый** — при изменении данных не требуется переписывать значительный объем кода;
- **расширяемый** — архитектура кода допускает расширение его функционала в будущем;
- **удобен для отладки** — при возникновении проблемы код позволяет достаточно точно определить ее источник.

Умение писать удобный для сопровождения JS-код — важный навык, который среди прочих отличает профессионального веб-разработчика, по-настоящему владеющего своим мастерством, от энтузиаста, способного «состряпать» за выходные худо-бедно работающий сайт.

Соглашения по формату кода

Один из простейших способов начать писать удобный для сопровождения JS-код — это принять те или иные соглашения. Соглашения по оформлению кода доступны для большинства языков программирования, и в интернете можно найти тысячи документов по этой теме. Профессиональные организации уже давно предлагают разработчикам использовать соглашения, чтобы упростить сопровождение кода. Лучшие проекты с открытым исходным кодом включают строгие требования к его написанию, помогающие всем участникам сообщества понимать организацию кода.

Соглашения по оформлению кода особо важны для JavaScript из-за того, что он настолько гибок. В отличие от большинства объектно-ориентированных языков, JavaScript не вынуждает разработчиков определять все как объекты и поддерживает много стилей программирования: от традиционной объектно-ориентированной парадигмы до декларативных и функциональных подходов. Быстро просмотрев несколько JavaScript-библиотек с открытым исходным кодом, можно легко обнаружить разнообразие подходов к созданию объектов, определению методов и управлению средой.

В последующих разделах описаны общие принципы разработки соглашений по оформлению кода. Они очень важны сами по себе, хотя конкретные способы их реализации могут зависеть от специфических требований.

Удобочитаемость кода

Чтобы код было удобно сопровождать, он должен быть удобочитаемым, а это напрямую зависит от форматирования кода в текстовом файле, в том числе от отступов. Если все разработчики используют одну схему отступов во всем проекте, читать код гораздо проще. Отступы обычно создают с помощью пробелов, а не знака табуляции, который отображается в текстовых редакторах по-разному. Популярный размер отступа — четыре пробела, хотя при желании его можно уменьшить или увеличить.

Кроме того, удобочитаемость кода зависит от комментариев. В большинстве языков программирования принято комментировать каждый метод, но из-за того, что JavaScript позволяет создавать функции в любом месте кода, этим правилом часто пренебрегают. Но это лишь подчеркивает важность документирования каждой JavaScript-функции. Перечислим элементы кода, которые следует комментировать:

- **Функции и методы** — все функции и методы должны включать комментарии, описывающие их назначение и, возможно, алгоритмы решения задач. Также важно указывать сделанные предположения, смысл аргументов и описывать возвращаемое функцией значение, если оно имеется (поскольку это нельзя выяснить по определению функции).
- **Большие фрагменты кода** — перед многострочными фрагментами кода, которые используются для решения конкретных задач, следует добавлять комментарии с описанием этих задач.
- **Сложные алгоритмы** — если вы используете уникальный подход к решению проблемы, объясните в комментарии, в чем он заключается. Это не только поможет другим людям, которые будут читать ваш код, но и упростит жизнь вам самим.
- **Трюки и хитрости** — из-за различий браузеров код JavaScript обычно использует некоторые хитрости. Не думайте, что кто-то другой, кто будет читать код, поймет, для чего они используются. Если вы решаете какую-то задачу нетрадиционным образом, потому что один из браузеров не поддерживает обычный подход, укажите это в комментарии. Благодаря этому другие разработчики не будут пытаться «исправить» ваш код, внося в него ошибки, которые вы уже исправили.

Отступы и комментарии делают код более удобочитаемым, упрощая его дальнейшее сопровождение.

Имена переменных и функций

Простота понимания и сопровождения кода во многом зависит от правильного выбора имен переменных и функций. Многие JavaScript-разработчики поначалу воспринимали программирование как хобби, поэтому по привычке иногда используют бессмысленные имена переменных вроде `"foo"` или `"bar"` и имена функций вроде `"doSomething"`. Если вы хотите заниматься разработкой профессионально, то должны отказаться от таких привычек, иначе сопровождать ваш код будет гораздо труднее. Рассмотрим общие правила выбора имен:

- В качестве имен переменных следует использовать существительные, такие как `"car"` или `"person"`.
- Имена функций следует начинать с глагола, например `getName()`. Имена функций, которые возвращают логические значения, обычно начинают словом `is`, например `isEnabled()`.
- Следует использовать логичные имена переменных и функций, не беспокоясь об их длине. Длину имен можно сократить путем постобработки и сжатия (см. далее).
- Переменные, функции и методы должны начинаться со строчной буквы и должен использоваться верблюжий регистр, например, `getName()` и `isPerson`. Классы, вроде `Person` и `RequestFactory`, должны быть написаны заглавными буквами. Постоянные значения должны быть в верхнем регистре с нижними подчеркиваниями, например `REQUEST_TIMEOUT`.
- Делайте имена описательными и понятными, но не слишком многословными. `getName()` интуитивно вернет значение имени. `PersonFactory` будет производить какой-либо объект или сущность `Person`.

Не используйте имена, которые ничего не говорят о данных, содержащихся в переменных. Если имена подобраны удачно, код читается как рассказ и понятен даже при беглом просмотре.

Прозрачность типов переменных

Поскольку переменные в JavaScript слабо типизированы, легко забыть, какие данные должна содержать переменная. Правильный подбор имен отчасти решает эту проблему, но иногда этого недостаточно. Указать тип данных переменной можно тремя способами.

Первый способ основан на инициализации. Его суть в том, что при определении переменной вы инициализируете ее значением, указывающим, как она будет использоваться в будущем. Например, переменную, которая будет содержать логическое значение, следует инициализировать значением `true` или `false`, а числовую переменную — числом:

```
// указание типов переменных путем инициализации
let found = false;        // логическое значение
let count = -1;           // число
let name = "";            // строка
let person = null;        // объект
```

Инициализация переменной конкретным значением ясно указывает на ее тип. Недостаток этого подхода в том, что его нельзя использовать с аргументами в объявлениях функций.

Второй способ указать тип переменной — это использовать венгерскую нотацию. В этом случае для указания типа переменной добавляют к ее имени один или несколько символов. Такая нотация часто используется в языках сценариев и какое-то время была популярна в JavaScript. Наиболее традиционный вариант венгерской нотации для JavaScript предполагает добавление одного символа для базовых типов данных, например: `"o"` — для объектов, `"s"` — для строк, `"i"` — для целых чисел, `"f"` — для чисел с плавающей точкой, `"b"` — для логических значений:

```
// указание типов переменных с помощью венгерской нотации
let bFound;               // логическое значение
let iCount;               // целое число
let sName;                // строка
let oPerson;              // объект
```

Венгерская нотация в JavaScript привлекательна тем, что ее можно использовать с аргументами функций. Ее недостаток в том, что она затрудняет чтение кода, нарушая его интуитивное восприятие цельными блоками. По этой причине некоторые разработчики отказались от венгерской нотации.

Последний способ указания типов переменных — использование комментариев типов. Такие комментарии добавляются сразу после имен переменных, но перед их инициализацией, например:

```
// указание типов переменных с помощью комментариев
let found    /*:логическое значение*/   = false;
let count    /*:целое число*/           = 10;
let name     /*:строка*/                = "Nicholas";
let person   /*:объект*/                = null;
```

Комментарии типов добавляют в код сведения о типах переменных, но при этом сохраняют общую удобочитаемость кода. Недостаток таких комментариев в том, что они мешают комментировать крупные блоки кода с помощью многострочных комментариев, потому что комментарии типов тоже являются многострочными:

```
// следующий подход не сработает
/*
let found    /*:логическое значение*/   = false;
let count    /*:целое число*/           = 10;
let name     /*:строка*/                = "Nicholas";
let person   /*:объект*/                = null;
*/
```

Здесь мы хотим закомментировать все переменные с помощью многострочного комментария, но комментарии типов препятствуют этому, потому что первое сочетание символов `/*` во второй строке сопоставляется с первым сочетанием `*/` в третьей строке, что приводит к синтаксической ошибке. Если нужно закомментировать строки кода с комментариями типов, лучше использовать однострочные комментарии в каждой строке (многие редакторы поддерживают такую возможность).

Таковы наиболее популярные способы, позволяющие указать типы данных переменных. Каждый из них имеет преимущества и недостатки. Выберите способ, наиболее подходящий для проекта, и используйте его согласованным образом.

Слабая связанность

Если части приложения чересчур зависят одна от другой, код становится сильно связанным и сложным для сопровождения. Типичный пример этого имеет место, если объекты напрямую ссылаются друг на друга так, что изменение одного всегда требует изменения другого. Сильно связанные программы трудно сопровождать, их фрагменты часто приходится переписывать.

Сильная связанность в веб-приложениях формируется в результате некомпетентного использования соответствующих технологий. Важно знать о том, когда и как это происходит, и стараться поддерживать слабую связанность кода.

Ослабление связанности HTML и JavaScript

Один из наиболее частых вариантов связанности имеет место между кодом HTML и JavaScript. В веб-приложениях эти технологии работают на разных уровнях: HTML служит для представления данных, а JavaScript — для реализации поведения. Есть несколько способов объединения этих технологий, но некоторые из них связывают HTML и JavaScript слишком сильно.

Сильная связанность наблюдается, если JS-код встраивается в HTML с помощью элемента `<script>` или если обработчики событий назначаются HTML-атрибутам, например:

```
<!-- использование тега <script> - сильная связанность HTML и JavaScript -->
<script>
  document.write("Hello world!");
</script>

<!-- назначение обработчика события атрибуту –
     сильная связанность HTML и JavaScript -->
<input type="button" value="Click Me" onclick="doSomething()" />
```

Хотя оба эти примера технически корректны, на практике они сильно связывают HTML-код, представляющий данные, с JS-кодом, который определяет поведение. В идеале HTML и JavaScript должны быть полностью разделены; JS-код следует включать из внешних файлов и назначать поведение с помощью DOM.

Если HTML и JavaScript связаны слишком сильно, при возникновении ошибки приходится сначала определять, где она произошла — в HTML-коде или в JS-файле. Кроме того, сильная связанность может привносить в приложения ошибки новых типов, касающиеся доступности кода. В приведенном примере пользователь может щелкнуть на кнопке, когда функция `doSomething()` еще не стала доступной, что приведет к ошибке. Это также затрудняет сопровождение кода, потому что для любой модификации поведения кнопки требуется изменять не только JS-сценарий, но и HTML-код.

HTML и JavaScript могут быть слишком сильно связаны и в противоположной ситуации: если HTML-код содержится в JS-сценарии. Обычно это имеет место при использовании свойства `innerHTML` для вставки блока HTML-кода в страницу:

```
// сильная связанность HTML и JavaScript
function insertMessage(msg) {
    let container = document.getElementById("container");
    container.innerHTML = `<div class="msg">
        <p> class="post">${msg}</p>
        <p><em>Latest message above.</em></p>
        </div>`;
}
```

Вообще говоря, в JS-сценариях не следует использовать много HTML-кода. Это помогает поддерживать разделение уровней приложения и упрощает поиск причин ошибок. В приведенном фрагменте при неправильном форматировании динамически создаваемого HTML-кода может возникнуть проблема с макетом страницы. Однако найти причину ошибки будет непросто, потому что при просмотре кода страницы динамически генерируемого HTML-кода вы не увидите. На сильную связанность указывает еще и то, что для модификации данных или макета потребуется внести изменения в JS-код.

Генерирование HTML-кода также по возможности следует отделять от JS-сценариев. Если JS-код используется для вставки данных, желательно, чтобы он не добавлял еще и разметку. Как правило, разметку можно добавить и скрыть при генерировании всей страницы, а позднее просто показать ее средствами JavaScript. Другой подход — получить нужный дополнительный HTML-код с помощью Ajax-запроса; это позволяет использовать тот же уровень генерирования HTML-кода (PHP, JSP, Ruby и т. д.) для вывода разметки, а не встраивать ее в JS-сценарий.

Разделение HTML и JavaScript позволяет сэкономить время на отладке, облегчая поиск причин ошибок, а также упрощает сопровождение кода, потому что изменение поведения ограничено файлами JavaScript, а изменение разметки — файлами генерирования HTML-кода.

Ослабление связанности CSS и JavaScript

Каскадные таблицы стилей (Cascading Style Sheets, CSS) определяют вид страницы. JavaScript и CSS служат для модификации HTML-страниц и часто используются

вместе, а потому также иногда могут быть связаны чересчур сильно. Чаще всего это имеет место, когда JS-код предназначен для изменения отдельных стилей, например:

```
// сильная связанность CSS и JavaScript
element.style.color = "red";
element.style.backgroundColor = "blue";
```

Поскольку вид страницы определяется таблицами CSS, в идеале для решения любой проблемы с видом должно быть достаточно просмотров CSS-файлов. Однако если для изменения отдельных стилей, например цвета, используется JavaScript, это требует проверки и, возможно, изменения дополнительного фрагмента. В результате код JavaScript, который частично отвечает за отображение страницы, оказывается связан с CSS сильнее, чем хотелось бы. Если в будущем придется изменить стили такой страницы, может потребоваться изменить не только CSS-файл, но и JS-код, что затрудняет сопровождение приложения. Стили и поведение следует разделять более четко.

В современных приложениях JS-код часто используется для изменения стилей, так что полностью разделить его и CSS невозможно, но все же их связанность можно ослабить. Для этого следует динамически изменять классы, а не отдельные стили, например:

```
// слабая связанность CSS и JavaScript
element.className = "edit";
```

Изменяя только CSS-класс элемента, вы оставляете большинство данных о стилях строго в CSS. Даже если для изменения класса используется JS-код, это не влияет на стиль элемента напрямую. Если к элементам применены правильные классы, причины любых проблем с отображением будут крыться только в CSS, а не в коде JavaScript.

Сказанное еще раз подчеркивает важность правильного разделения уровней приложения. Единственным источником проблем с видом страницы должен быть CSS-код, а единственным источником ошибок поведения — JS-код. Поддержание слабой связанности между разными технологиями упрощает сопровождение всего приложения.

Ослабление связанности логики приложения и обработчиков событий

Веб-приложения обычно содержат множество обработчиков различных событий, но логика приложения редко отделяется от обработчиков. Рассмотрим пример:

```
function handleKeyPress(event) {
    if (event.keyCode == 13) {
        let target = event.target;
        let value = 5 * parseInt(target.value);
        if (value > 10) {
            document.getElementById("error-msg").style.display = "block";
        }
    }
}
```

Эта функция не только обрабатывает событие, но и содержит логику приложения. Проблем с этим подходом две. Во-первых, логика запускается только с помощью события, что затрудняет отладку. Что, если предполагаемый результат не получен? Означает ли это, что обработчик не был вызван или логика приложения содержит дефект? Во-вторых, если последующее событие должно запускать такой же код, вам придется дублировать функционал или выносить его в отдельную функцию. В обоих случаях потребуется внести больше изменений, чем необходимо.

Вместо этого лучше отделить логику приложения от обработчиков событий, чтобы каждый блок решал конкретную задачу. Обработчик события должен получать нужные данные из объекта `event` и передавать их в некоторый метод, содержащий логику приложения. Например, предыдущий код можно переписать следующим образом:

```
function validateValue(value) {
    value = 5 * parseInt(value);
    if (value > 10) {
        document.getElementById("error-msg").style.display = "block";
    }
}

function handleKeyPress(event) {
    if (event.keyCode == 13) {
        var target = event.target;
        validateValue(target.value);
    }
}
```

В этом коде логика и обработчик события разделены. Функция `handleKeyPress()` проверяет, нажата ли клавиша Enter (значение `event.keyCode` равно `13`), а затем получает целевой элемент события и передает свойство `value` в функцию `validateValue()`, которая содержит логику приложения. В ней нет ничего, что как-либо зависит от логики обработчика события; она просто получает значение и в зависимости от него определяет, что делать дальше.

Отделение логики приложения от обработчиков событий обеспечивает несколько преимуществ. Во-первых, это позволяет легко менять события, запускающие те или иные процессы. Например, если первоначально обработчик запускался щелчком кнопки мыши, его можно с легкостью заменить нажатием клавиши. Во-вторых, вы можете тестировать код, не подключая обработчики событий, что упрощает создание модульных тестов и автоматизацию процессов в приложении.

Вот несколько принципов ослабления связанности логики приложения и бизнес-логики:

- в методы следует передавать не сам объект `event`, а только нужные данные из него;
- приложение должно поддерживать выполнение каждого действия без вызова обработчика события;
- обработчики событий должны обрабатывать событие, а затем передавать управление логике приложения.

Соблюдение этих принципов значительно упрощает сопровождение любого кода, помогая при этом повысить эффективность тестирования и дальнейшей разработки.

Принципы программирования

Конечно, простота сопровождения JS-кода зависит не только от его оформления, но и от того, что он, собственно, делает. Над корпоративными веб-приложениями обычно работают многие люди, и в таких условиях очень важно унифицировать используемую всеми среду браузера с помощью каких-то неизменных правил. Для этого разработчики должны придерживаться определенных принципов программирования.

Соблюдайте права владения объектами

Динамическая природа JavaScript позволяет в любой момент изменить практически любое значение. Говорят, что в JavaScript «нет ничего святого», потому что никакое значение нельзя сделать окончательным или постоянным. Хотя с добавлением в ECMAScript5 объектов, защищенных от изменений, это уже не так, все равно по умолчанию вы можете изменять любые объекты. В других языках объекты и классы неизменны, если они не содержат исходного кода, но в JavaScript любой объект можно изменить когда угодно, что иногда дает непредвиденные результаты. Поскольку язык предоставляет такие широкие возможности, важно, чтобы разработчики сами соблюдали некоторые ограничения.

Возможно, наиболее важным принципом программирования в корпоративной среде является соблюдение прав владения объектами. Это означает, что вы не должны изменять объекты, которые вам не принадлежат. Проще говоря, если вы не отвечаете за создание или обслуживание объекта, его конструктора или методов, то изменять его вы тоже не должны. Или, если точнее:

- не добавляйте свойства в экземпляры или прототипы;
- не добавляйте методы в экземпляры или прототипы;
- не переопределяйте существующие методы.

Проблема в том, что разработчики имеют определенные ожидания в отношении того, как должна работать текущая среда браузера, из-за чего изменение объектов, используемых несколькими людьми, часто приводит к ошибкам. Если кто-то исходит из того, что функция `stopEvent()` отменяет для события поведение, предлагаемое по умолчанию, а вы изменяете ее, подключая в ней другие обработчики, это наверняка повлечет за собой проблемы. Другие разработчики, не подозревающие о новых возможностях функции, будут использовать ее согласно старой спецификации, что может нарушить работу приложения.

Эти правила относятся не только к пользовательским, но и к встроенным типам и объектам, таким как `Object`, `String`, `document`, `window` и т. д. В этом случае потенциальные проблемы еще опаснее, потому что производители браузеров могут изменять эти объекты по своему усмотрению без предварительного уведомления.

В качестве примера можно привести историю с популярной библиотекой Prototype. У объекта `document` в ней был реализован метод `getElementsByClassName()`, который возвращал экземпляр `Array`, расширенный методом `each()`. Проблема возникла, когда в браузерах был реализован встроенный метод `getElementsByClassName()`, возвращающий не `Array`, а объект `NodeList`, у которого нет метода `each()`, — о событиях, приведших к проблеме, можно прочитать в блоге Джона Резига (John Resig) по адресу http://ejohn.org/blog/getelementsbyclassname-pre-prototype-16/. Разработчики, использующие библиотеку Prototype, привыкли писать подобный код:

```
document.getElementsByClassName("selected").each(Element.hide);
```

Хотя этот код нормально работал в браузерах, не имеющих встроенного метода `getElementsByClassName()`, в обновленных браузерах с этим методом он стал вызывать ошибку из-за разных возвращаемых значений. Вы не можете предугадать, как производители браузеров изменят встроенные объекты, так что любое их изменение может привести к конфликтам с вашим кодом, если он написан неаккуратно.

Таким образом, лучше никогда не изменять объекты, которые вам не принадлежат. А какие объекты принадлежат вам? Только те, которые вы создали сами, это может быть, например, пользовательский тип или литерал объекта. В то же время объекты вроде `Array`, `document` и т. д., существовавшие еще до написания вашего кода, изменять не следует. Если нужно расширить функционал имеющихся объектов, можно сделать следующее:

- создайте новый объект с нужным функционалом и реализуйте его взаимодействие с исходным объектом;
- унаследуйте пользовательский тип от типа, который нужно изменить, и добавьте нужный функционал в новый тип.

Этот подход характерен для многих современных JavaScript-библиотек, что позволяет расширять и адаптировать их, несмотря на частые изменения браузеров.

Не используйте глобальные сущности

С соблюдением прав владения объектами тесно связан принцип отказа от глобальных переменных и функций. Он также имеет целью формирование согласованной и удобной для сопровождения среды выполнения сценариев. Суть принципа в том, что у вас должна быть только одна глобальная переменная, содержащая другие объекты и функции. Рассмотрим пример:

```
// две глобальные сущности – НЕ ДЕЛАЙТЕ ТАК!!!
var name = "Nicholas";
function sayName() {
    console.log(name);
}
```

Этот код содержит две глобальные сущности: переменную `name` и функцию `sayName()`. Однако вы можете легко создать их в единственном глобальном объекте:

```
// одна глобальная сущность – предпочтительный подход
var MyApplication = {
    name: "Nicholas",
    sayName: function() {
        console.log(this.name);
    }
};
```

Эта версия кода содержит единственный глобальный объект `MyApplication` с переменной `name` и методом `sayName()`, что устраняет проблемы прежнего кода. Во-первых, переменная `name` больше не перезаписывает свойство `window.name`, что в предыдущем примере могло мешать работе другого кода. Во-вторых, в новом коде проще понять, к чему относятся сущности. Например, вызов `MyApplication.sayName()` ясно показывает, что причины проблем с методом `sayName()` следует искать в объекте `MyApplication`.

Логичным развитием принципа единственного глобального объекта является концепция *пространств имен*. Она включает создание объекта исключительно в качестве контейнера для того или иного функционала. Библиотека Google Closure использует пространства имен для организации своего содержимого. Вот некоторые примеры:

- `goog.string` — методы манипулирования строками.
- `goog.html.utils` — методы для работы с HTML.
- `goog.i18n` — методы помощи с интернационализацией (i18n).

В качестве контейнера используется единственный глобальный объект `goog`, в котором определены другие объекты. Когда объекты нужны для подобного группирования функционала, их называют пространствами имен. Вся библиотека Google Closure построена на основе этой концепции, что позволяет применять ее с любой другой JavaScript-библиотекой на одной странице.

При создании пространства имен важно выбрать подходящее имя глобального объекта, которое должно быть интуитивно понятным для всех, кто будет его использовать, но при этом достаточно уникальным, чтобы его не выбрали другие разработчики. Им вполне может быть имя компании, для которой вы разрабатываете приложение, например `goog` или `Wrox`. После этого сгруппировать функционал в пространствах имен можно следующим образом:

```
// создание глобального объекта
var Wrox = {};

// создание пространства имен для данной книги
Wrox.ProJS = {};

// добавление других объектов, используемых в книге
Wrox.ProJS.EventUtil = { ... };
Wrox.ProJS.CookieUtil = { ... };
```

В этом примере пространства имен создаются в глобальном объекте `Wrox`. Если весь код для этой книги находится в пространстве имен `Wrox.ProJS`, ничто не мешает

другим авторам добавить код в объект Wrox. Если все соблюдают этот принцип, можно не беспокоиться о том, что кто-то другой также создаст объект EventUtil или CookieUtil, потому что он будет относиться к другому пространству имен, например:

```
// создание пространства имен для книги Professional Ajax
Wrox.ProAjax = {};

// добавление других объектов
Wrox.ProAjax.EventUtil = { ... };
Wrox.ProAjax.CookieUtil = { ... };

// объекты из пространства имен ProJS также доступны
Wrox.ProJS.EventUtil.addHandler( ... );

// доступ к объекту из пространства имен ProAjax
Wrox.ProAjax.EventUtil.addHandler( ... );
```

Хотя для использования пространств имен требуется немного больше кода, это компенсируется удобством его сопровождения. Пространства имен гарантируют, что ваш код будет работать на одной странице с другим кодом без каких бы то ни было конфликтов.

Не сравнивайте значения с null

В JavaScript типы не проверяются автоматически, за это отвечают разработчики, которые в реальности часто этим пренебрегают. Чаще всего при проверке типа значение сравнивают с null, но этот прием нередко используется неправильно, что приводит к ошибкам. Рассмотрим пример:

```
function sortArray(values) {
    if (values != null) {        // НЕ ДЕЛАЙТЕ ТАК!!
        values.sort(comparator);
    }
}
```

Эта функция предназначена для сортировки массива с помощью компаратора. Чтобы она работала правильно, аргумент values должен быть массивом, но инструкция if лишь сравнивает его с null. Проблема в том, что эту проверку пройдут также и другие значения, в том числе строка и любое число, что приведет к ошибке.

На практике сравнение значения с null нужно не так уж часто. Значения следует сравнивать с тем, чем они должны быть, а не наоборот. Например, в предыдущем фрагменте аргумент values должен быть массивом, поэтому следует проверить, действительно ли это массив, а не сравнивать его с null:

```
function sortArray(values) {
    if (values instanceof Array) {    // предпочтительный подход
        values.sort(comparator);
    }
}
```

Эта версия функции отсекает все недопустимые значения, а не только null.

Обнаружив в коде сравнение с `null`, попытайтесь заменить его, выбрав один из следующих приемов:

- если значение должно быть ссылочным, проверьте его конструктор с помощью оператора `instanceof`;
- если значение должно быть примитивным, проверьте его тип с помощью оператора `typeof`;
- если вы ожидаете объект с конкретным методом, убедитесь в наличии этого метода с помощью оператора `typeof`.

Чем меньше в коде сравнений с `null`, тем проще определить его назначение и устранить возможные ошибки.

Используйте константы

Идея применения констант заключается в том, что данные следует изолировать от логики приложения, чтобы их можно было изменять, не рискуя внести ошибки.

Строки, которые отображаются в пользовательском интерфейсе, всегда следует выделять в специальные блоки, чтобы упростить локализацию приложения. С URL-адресами следует поступать так же, потому что они имеют свойство изменяться по мере развития приложения. Оба этих элемента данных могут измениться по той или иной причине, что потребует изменения кода функции. Каждый раз, когда вы меняете код логики приложения, вы можете внести в него ошибку. Чтобы предотвратить это, вы можете изолировать логику приложения от данных, заменив их константами, определенными отдельно.

Главное — отделить данные от логики, в которой они используются. Перечислим типы данных, с которыми имеет смысл так поступать:

- **Повторяющиеся значения** — любые значения, используемые более чем в одном месте (в том числе имена CSS-классов), следует определять как константы. Так вы не сможете допустить ошибку, изменив лишь одно значение, когда нужно изменить несколько.
- **Строки пользовательского интерфейса** — для упрощения интернационализации любые строки, отображаемые пользователю, следует извлекать из кода.
- **URL-адреса** — ссылки на ресурсы часто изменяются в веб-приложениях, так что рекомендуется хранить все URL-адреса в одном месте.
- **Любые значения, которые могут измениться** — каждый раз, когда вы используете в коде литерал, спросите себя, может ли он в будущем измениться. Если да, это значение следует выделить в константу.

Использование констант — важный аспект разработки корпоративных JavaScript-приложений, потому что это упрощает сопровождение кода и защищает его от изменений данных.

БЫСТРОДЕЙСТВИЕ

Объем JS-кода на типичной веб-странице за время существования языка значительно увеличился, в связи с чем его быстродействие стало вызывать нарекания. JavaScript изначально был интерпретируемым языком, так что скорость выполнения написанного на нем кода традиционно намного ниже, чем в компилируемых языках. Chrome стал первым браузером, в котором был представлен оптимизирующий модуль, компилирующий JavaScript во встроенный машинный код. Вслед за этим компиляторы JavaScript были реализованы во всех основных браузерах.

Конечно, компиляция мало чем поможет, если код написан плохо, но использование некоторых базовых шаблонов позволяет сделать код максимально быстрым.

Область видимости

В главе 4 «Переменные, область видимости и память» мы обсудили области видимости и цепочку областей видимости в JavaScript. При увеличении количества областей видимости в цепочке возрастает и время доступа к переменным вне текущей области видимости. Из-за просмотра цепочки доступ к глобальной переменной всегда осуществляется медленнее, чем к локальной. Сократив время просмотра цепочки областей видимости, можно повысить общее быстродействие сценария.

Минимизируйте доступ к глобальным сущностям

Пожалуй, самое важное, что можно сделать для повышения быстродействия сценариев, — это минимизировать доступ к глобальным сущностям. Для доступа к глобальным переменным и функциям всегда требуется больше ресурсов в сравнении с локальными операциями, потому что он подразумевает просмотр цепочки областей видимости. Рассмотрим следующую функцию:

```
function updateUI() {
    let imgs = document.getElementsByTagName("img");
    for (let i=0, len=imgs.length; i < len; i++) {
        imgs[i].title = '${document.title} image ${i}';
    }

    let msg = document.getElementById("msg");
    msg.innerHTML = "Update complete.";
}
```

Эта функция работает правильно, но содержит три ссылки на глобальный объект `document`. Если на странице много изображений, код доступа к нему в цикле `for` может быть выполнен десятки и даже сотни раз, при этом каждый раз требуется просмотреть цепочку областей видимости. Создав локальную переменную, указывающую на объект `document`, вы можете сократить количество просмотров до одного, намного повысив быстродействие функции:

```
function updateUI() {
    let doc = document;
    let imgs = doc.getElementsByTagName("img");
    for (var i=0, len=imgs.length; i < len; i++) {
        imgs[i].title = '${doc.title} image ${i}';
    }

    let msg = doc.getElementById("msg");
    msg.innerHTML = "Update complete.";
}
```

Здесь объект `document` сохраняется в локальной переменной `doc`, которая затем используется вместо него во всем остальном коде. Эта функция обращается к глобальному объекту лишь один раз, благодаря чему выполняется гораздо быстрее.

На практике имеет смысл сохранять в локальной переменной любой глобальный объект, используемый в функции более одного раза.

Не используйте инструкцию with

Если быстродействие действительно важно, инструкцию `with` лучше не использовать. Подобно функциям, она создает собственную область видимости, увеличивая длину цепочки для кода внутри `with`. Такой код всегда выполняется медленнее, чем код снаружи `with`, из-за накладных расходов на просмотр цепочки областей видимости.

Инструкция `with` требуется редко, потому что она служит в основном для сокращения объема вводимого кода. Как правило, того же результата можно добиться с помощью локальной переменной без создания новой области видимости:

```
function updateBody() {
    with(document.body) {
        console.log(tagName);
        innerHTML = "Hello world!";
    }
}
```

В этом коде инструкция `with` упрощает работу со свойством `document.body`, но локальная переменная позволяет делать то же самое:

```
function updateBody() {
    let body = document.body;
    console.log(body.tagName);
    body.innerHTML = "Hello world!";
}
```

Хотя этот код немного длиннее, он более понятен, чем предыдущий фрагмент, потому что сразу ясно, к какому объекту относятся свойства `tagName` и `innerHTML`. Кроме того, сохранение значения `document.body` в локальной переменной экономит ресурсы на доступ к глобальному объекту.

Выбор оптимального подхода

Как и в других языках, в JavaScript быстродействие во многом зависит от алгоритма решения проблемы. Профессиональные разработчики по опыту знают, какие подходы более эффективны, а какие — менее. Многие подходы и приемы, часто используемые в других языках программирования, можно применять и в JavaScript.

Не просматривайте свойства без необходимости

В программировании сложность алгоритмов описывается с помощью O-нотации. Простейшие и быстрейшие алгоритмы имеют постоянную сложность, или O(1), после чего алгоритмы становятся более сложными и медленными. Основные типы JavaScript-алгоритмов описаны в таблице.

НОТАЦИЯ	СЛОЖНОСТЬ	ОПИСАНИЕ
O(1)	Постоянная	
O(log *n*)	Логарифмическая	Время выполнения алгоритма зависит от количества значений, но получать все значения не требуется. Пример: двоичный поиск
O(*n*)	Линейная	Время выполнения алгоритма пропорционально количеству значений. Пример: перебор всех элементов массива
O(n^2)	Квадратичная	Для выполнения алгоритма необходимо обратиться к каждому значению не менее *n* раз. Пример: сортировка вставками

Примером операции O(1), может служить доступ к литералу или значению переменной. Нотация O(1) указывает, что время, необходимое для получения значения, остается постоянным независимо от количества значений. Получение значения — эффективная и быстрая операция, например:

```
let value = 5;
let sum = 10 + value;
console.log(sum);
```

В этом коде мы обращаемся к четырем значениям: числу 5, переменной `value`, числу 10 и переменной `sum`. Общая сложность этого кода — O(1).

Доступ к элементам массива также является в JavaScript операцией со сложностью O(1) и почти столь же эффективен, как и простой просмотр переменной, например:

```
let values = [5, 10];
let sum = values[0] + values[1];
console.log(sum);
```

Использование переменных и массивов более эффективно, чем доступ к свойствам объектов, который имеет сложность O(n). Просмотр свойства объекта занимает больше времени, чем доступ к переменной или массиву, потому что свойство с указанным именем нужно найти в цепочке прототипов. Проще говоря, чем больше в коде операций доступа к свойствам, тем медленнее работает такой код. Рассмотрим пример:

```
let values = { first: 5, second: 10};
let sum = values.first + values.second;
console.log(sum);
```

Здесь для вычисления значения `sum` требуется прочитать два свойства. Просмотр одного или двух свойств — это, конечно, не проблема, но если таких операций сотни или тысячи, они определенно скажутся на быстродействии.

Избегайте чтения нескольких свойств для получения одного значения, например:

```
let query =
    window.location.href.substring(window.location.href.indexOf("?"));
```

Этот код просматривает шесть свойств: три при вызове метода `window.location.href.substring()` и еще три при вызове `window.location.href.indexOf()`. Вы можете легко подсчитать операции доступа к свойствам по точкам. Этот код примечателен еще и тем, что доступ к свойству `window.location.href` выполняется два раза без какой-либо оптимизации.

Если свойство объекта используется более одного раза, следует сохранить его в локальной переменной. В этом случае для первого доступа к свойству потребуется время O(n), но зато каждый последующий доступ будет выполняться за время O(1), что более чем компенсирует начальные издержки. Например, предыдущий код можно переписать так:

```
let url = window.location.href;
let query = url.substring(url.indexOf("?"));
```

Такая версия включает только четыре просмотра свойства, что на треть меньше в сравнении с оригиналом. В большом сценарии это может сэкономить много ресурсов.

Всякий раз, когда можно снизить сложность алгоритма, не следует пренебрегать этим. Таким образом, обращения к свойствам следует во всех возможных случаях заменять локальными переменными, а если доступ к какому-либо значению возможен с помощью числового индекса или именованного свойства (например, при работе с объектом `NodeList`), лучше использовать индекс.

Оптимизируйте циклы

Циклы — одна из чаще всего используемых конструкций в программировании в целом и в JavaScript в частности. Оптимизация циклов вносит особо важный вклад в повышение быстродействия, потому что код в циклах выполняется многократно.

Оптимизация циклов в других языках хорошо изучена, и многие приемы из них также применимы к JavaScript. Рассмотрим эти основные приемы:

1. **Упрощение окончательного условия** — поскольку окончательное условие оценивается на каждой итерации цикла, желательно сделать его как можно более простым. Это означает, что в нем в идеале не должно быть просмотров свойств и других операций со сложностью O(n).
2. **Упрощение тела цикла** — основная доля ресурсов расходуется на выполнение тела цикла, поэтому его оптимизация заслуживает повышенного внимания. Убедитесь, что тело цикла не содержит ресурсоемких вычислений, которые можно вынести за пределы цикла.
3. **Использование циклов с постусловием** — чаще всего используются циклы `for` и `while`, в которых проверяется предусловие. В циклах с постусловием, таких как `do-while`, нет первоначальной оценки окончательного условия, поэтому они обычно выполняются быстрее.

> **ПРИМЕЧАНИЕ** В старых браузерах было более эффективным запускать итераторы цикла в их максимальном значении и уменьшать их до 0. Это было быстрее из-за уменьшенного числа инструкций, которые механизм JavaScript использовал бы для проверки состояния ветвления цикла. В современных браузерах ощутимой разницы в производительности не будет, поэтому следует использовать формат итератора, наиболее подходящий для целей кода.

Эти приемы лучше всего продемонстрировать на примере. Вот простой цикл `for`:

```
for (let i=0; i < values.length; i++) {
    process(values[i]);
}
```

Этот код увеличивает переменную `i` с нуля до общего количества элементов в массиве `values`. Если порядок обработки элементов неважен, увеличение итератора можно заменить уменьшением:

```
for (let i=values.length-1; i >= 0; i--) {
    process(values[i]);
}
```

В новой версии переменная `i` на каждой итерации уменьшается, благодаря чему в окончательном условии доступ к значению `values.length` — операция сложности O(n) — сменился сравнением итератора с нулем, которое имеет сложность O(1). Поскольку тело цикла включает только одну инструкцию, оптимизировать его невозможно, однако сам цикл можно преобразовать в цикл с постусловием:

```
let i=values.length-1;
if (i > -1) {
    do {
        process(values[i]);
    }while(--i >= 0);
}
```

Главный фактор оптимизации здесь — это объединение окончательного условия и оператора декремента в одной инструкции. Теперь цикл оптимизирован полностью, и дальнейшее повышение быстродействия без изменения функции `process()` невозможно.

Цикл с постусловием дает прирост быстродействия лишь в том случае, если вы точно знаете, что нужно будет обработать хотя бы одно значение. Например, при пустом массиве в таком цикле все равно будет выполнена одна итерация, которая не потребовалась бы в цикле с предусловием.

Развертывайте циклы

Если количество итераций цикла конечно, часто эффективнее заменить его несколькими вызовами функции. Давайте вернемся к циклу из предыдущего раздела. Если длина массива всегда будет одинаковой, возможно, лучше просто вызвать функцию `process()` для каждого элемента:

```
// развернутый цикл
process(values[0]);
process(values[1]);
process(values[2]);
```

В этом примере предполагается, что массив `value` содержит только три элемента, поэтому функция `process()` вызывается три раза. Такое развертывание цикла позволяет не тратить ресурсы на его настройку и обработку конечного условия, что повышает быстродействие кода.

Если количество итераций цикла неизвестно, можно попробовать так называемый *метод Даффа* (Duff's device). Этот прием придумал Том Дафф (Tom Duff), впервые предложивший использовать его в языке программирования C. Реализацию метода Даффа на JavaScript приписывают Джеффу Гринбергу (Jeff Greenberg). Суть этого метода — развертывание цикла в последовательность инструкций, выполняемых в новом цикле с количеством итераций, уменьшенным в 8 раз. Рассмотрим пример:

```
// реализация метода Даффа на JS — Джефф Гринберг
// предполагается, что values.length > 0
let iterations = Math.ceil(values.length / 8);
let startAt = values.length % 8;
let i = 0;

do {
    switch(startAt) {
        case 0: process(values[i++]);
        case 7: process(values[i++]);
        case 6: process(values[i++]);
        case 5: process(values[i++]);
        case 4: process(values[i++]);
        case 3: process(values[i++]);
        case 2: process(values[i++]);
        case 1: process(values[i++]);
    }
    startAt = 0;
} while (--iterations > 0);
```

Эта реализация метода Даффа начинается с вычисления количества итераций цикла, для чего общее количество элементов в массиве `values` делится на 8 и результат округляется вверх до целого числа методом `ceil()`. Переменной `startAt` присваивается количество элементов, оставшихся после деления на 8, и на первой итерации цикла она используется для обработки этих элементов. Например, если массив содержит 10 значений, переменная `startAt` будет равна 2, поэтому на первой итерации функция `process()`вызывается только два раза. В конце первой итерации переменная `startAt` обнуляется, так что на каждой последующей итерации функция `process()` вызывается 8 раз. Такое развертывание цикла ускоряет обработку больших наборов данных.

В книге *Speed Up Your Site* Эндрю Б. Кинг (Andrew B. King, New Riders, 2003) предложил еще более быстрый метод Даффа, подразумевающий разделение цикла `do-while` на два цикла:

```
// код взят из книги "Speed Up Your Site" (New Riders, 2003)
let iterations = Math.floor(values.length / 8);
let leftover = values.length % 8;
let i = 0;

if (leftover > 0) {
    do {
        process(values[i++]);
    } while (--leftover > 0);
}

do {
    process(values[i++]);
    process(values[i++]);
    process(values[i++]);
    process(values[i++]);
    process(values[i++]);
    process(values[i++]);
    process(values[i++]);
    process(values[i++]);
} while (--iterations > 0);
```

В этой реализации первый цикл обрабатывает элементы, оставшиеся после деления на 8 (значение `leftover`). Когда эти дополнительные элементы обработаны, выполняется основной цикл, в котором 8 раз вызывается функция `process()`. Этот подход почти на 40 % быстрее, чем оригинальная реализация метода Даффа.

Развертывание циклов может сэкономить много ресурсов при обработке больших наборов данных, но менее эффективно, если данных немного. Небольшое повышение быстродействия за счет увеличения объема кода в этом случае обычно не стоит затрачиваемых усилий.

Избегайте двойной интерпретации

Двойная интерпретация имеет место, когда код JavaScript интерпретирует другой код JavaScript. Такая ситуация возникает при использовании функции `eval()`,

конструктора `Function` и вызове функции `setTimeout()` со строковым аргументом, например:

```
// интерпретация некоторого кода – НЕ ДЕЛАЙТЕ ТАК!!
eval("console.log('Hello world!')");

// создание новой функции – НЕ ДЕЛАЙТЕ ТАК!!
let sayHi = new Function("console.log('Hello world!')");

// установка тайм-аута – НЕ ДЕЛАЙТЕ ТАК!!
setTimeout("console.log('Hello world!')", 500);
```

В каждом из этих примеров необходимо интерпретировать строку, содержащую JS-код. При первоначальном синтаксическом анализе сделать это невозможно, потому что код содержится в строке, а это означает, что для ее анализа нужно запустить новый синтаксический анализатор, что требует много ресурсов. В результате такой код работает медленнее в сравнении со встроенным.

Все эти примеры можно заменить более эффективными аналогами. Метод `eval()` по-настоящему требуется редко, и по возможности лучше его не использовать. В приведенном примере код можно просто встроить. Вызов конструктора `Function` можно легко переписать как обычную функцию, а что касается метода `setTimeout()`, то в него можно передать функцию в качестве первого аргумента:

```
// исправлено
console.log('Hello world!');

// создание новой функции - исправлено
let sayHi = function() {
    console.log('Hello world!');
};

// установка тайм-аута - исправлено
setTimeout(function() {
    console.log('Hello world!');
}, 500);
```

Чтобы код работал быстрее, старайтесь не использовать строки JS-кода, требующие интерпретации.

Другие соображения по поводу быстродействия

При анализе быстродействия сценария следует учитывать еще несколько факторов. Их влияние менее заметно, но при частом использовании они могут повлиять на быстродействие кода.

- **Встроенные методы быстрее**. Старайтесь использовать встроенные методы, а не написанные на JavaScript. Встроенные методы реализованы на компилируемых языках, таких как C и C++, а потому работают гораздо быстрее. В частности, разработчики часто забывают о сложных математических методах, доступных

объекту `Math`; эти методы работают быстрее, чем любые эквиваленты синуса, косинуса и т. д., написанные на JavaScript.

- **Инструкция switch быстрее**. Если код содержит сложную последовательность инструкций `if-else`, то преобразовав ее в одну инструкцию `switch`, можно повысить его быстродействие. Чтобы сделать инструкцию `switch` еще более быстрой, организуйте варианты от наиболее вероятных к наименее вероятным.
- **Поразрядные операторы быстрее**. Поразрядные математические операции всегда выполняются быстрее, чем логические и числовые. Выборочная замена арифметических операций поразрядными может значительно ускорить сложные вычисления. Как нельзя лучше для такой замены подходят операции вроде деления по модулю, логического ИЛИ и логического И.

Сокращение количества инструкций

Скорость выполнения операций также зависит от количества инструкций в JS-коде. Единственная инструкция, охватывающая много операций, выполняется быстрее, чем много инструкций, каждая из которых выполняет одну операцию. Грамотное объединение инструкций позволяет сократить время выполнения всего сценария, при этом особого внимания заслуживают несколько шаблонов, о которых пойдет речь в этом разделе.

Несколько объявлений переменных

При объявлении переменных разработчики часто создают избыточные инструкции. Например, нередко встречается код, в котором переменные объявляются с помощью нескольких инструкций `let`:

```
// четыре инструкции — лишние расходы
let count = 5;
let color = "blue";
let values = [1,2,3];
let now = new Date();
```

В строго типизированных языках переменные разных типов нужно объявлять с помощью отдельных инструкций, но в JavaScript все переменные можно объявить в одной инструкции `let`:

```
// одна инструкция
let count = 5,
    color = "blue",
    values = [1,2,3],
    now = new Date();
```

Здесь все объявления переменных относятся к одной инструкции `var` и разделены запятыми. Этот код работает гораздо быстрее, чем объявление переменных по отдельности, а внести такое изменение совсем несложно.

Вставка итеративных значений

Каждый раз, когда вы используете итеративное значение (то есть значение, которое в разных местах кода увеличивается или уменьшается на единицу), объединяйте инструкции, если это возможно. Рассмотрим пример:

```
let name = values[i];
i++;
```

Каждая из двух предыдущих инструкций выполняет единственную задачу: первая получает значение из массива `values` и сохраняет его в переменной `name`, а вторая увеличивает переменную `i`. Их можно объединить в одной инструкции следующим образом:

```
let name = values[i++];
```

Эта инструкция эквивалентна двум предыдущим. Поскольку оператор инкремента является постфиксным, значение `i` увеличивается только после выполнения остальной части инструкции. Встретив похожий фрагмент, попытайтесь вставить итеративное значение в последнюю инструкцию, где оно используется.

Использование литералов массивов и объектов

В книге мы обсудили два способа создания массивов и объектов: с помощью конструктора и с помощью литерала. В первом случае для определения свойств или вставки элементов всегда требуется несколько инструкций, тогда как во втором все делается в одной инструкции. Рассмотрим пример:

```
// четыре инструкции для создания и инициализации массива — лишние расходы
let values = new Array();
values[0] = 123;
values[1] = 456;
values[2] = 789;

// четыре инструкции для создания и инициализации объекта — лишние расходы
let person = new Object();
person.name = "Nicholas";
person.age = 29;
person.sayName = function() {
    console.log(this.name);
};
```

Этот код создает и инициализирует массив и объект. В обоих случаях требуется выполнить четыре инструкции: одну для вызова конструктора и три для назначения данных. С помощью литералов можно заменить этот код следующим:

```
// одна инструкция для создания и инициализации массива
let values = [123, 456, 789];

// одна инструкция для создания и инициализации объекта
let person = {
    name : "Nicholas",
```

```
    age : 29,
    sayName : function() {
        console.log(this.name);
    }
};
```

Измененный код содержит только две инструкции: одна создает и инициализирует массив, а вторая — объект. Таким образом, мы заменили восемь инструкций двумя, уменьшив их количество на 75 %. Польза от такой оптимизации особенно заметна, если база кода содержит тысячи строк.

В общем, старайтесь заменять объявления массивов и объектов их литералами для сокращения количества инструкций.

> **ПРИМЕЧАНИЕ** Сокращение количества операторов в вашей кодовой базе – хорошая цель, но не абсолютный закон. Можно объединить слишком много логики в одно утверждение, чтобы оно перестало быть понятным.

Оптимизация взаимодействия с DOM

Из всех JavaScript-компонентов самым медленным, несомненно, является DOM. Взаимодействие с DOM занимает много времени потому, что часто требует частичной или полной перерисовки страницы. Даже тривиальные операции могут выполняться неожиданно долго, потому что DOM управляет огромными объемами данных. Оптимизация взаимодействия с DOM может существенно ускорить работу сценариев.

Минимизируйте динамические обновления

При доступе к той части DOM, которая отображается на странице, выполняется *динамическое обновление* (live update). Оно называется так потому, что страница обновляется немедленно. Каждое изменение, будь то вставка одного знака или удаление целого раздела, имеет свою цену, потому что для обновления браузер пересчитывает тысячи разных параметров. Чем больше операций динамического обновления вы инициируете, тем дольше выполняется код, и наоборот. Рассмотрим пример:

```
let list = document.getElementById("myList"),
           item;

for (let i=0; i < 10; i++) {
    item = document.createElement("li");
    list.appendChild(item);
    item.appendChild(document.createTextNode('Item ${i}');
}
```

Этот код добавляет в список 10 записей. Для каждой из них выполняются два динамических обновления: одно при добавлении элемента `<li>` и одно при добавлении

к нему текстового узла. Иначе говоря, для добавления 10 записей требуется 20 динамических обновлений.

Для экономии ресурсов желательно сократить количество таких обновлений. Это можно сделать двумя способами. Первый — удалить список со страницы, выполнить его обновление, а затем вставить список обратно в том же месте. К сожалению, этот подход может вызывать мерцание при обновлениях страницы. Второй способ — создать DOM-структуру с помощью фрагмента документа, а затем добавить ее к элементу `list`. Этот подход предотвращает динамические обновления и мерцание страницы. Рассмотрим пример:

```
let list = document.getElementById("myList"),
    fragment = document.createDocumentFragment(),
    item;

for (let i=0; i < 10; i++) {
    item = document.createElement("li");
    fragment.appendChild(item);
    item.appendChild(document.createTextNode("Item " + i));
}

list.appendChild(fragment);
```

В этом примере выполняется только одно динамическое обновление после создания всех записей. Созданные записи временно сохраняются во фрагменте документа, а затем все записи добавляются в список с помощью метода `appendChild()`. Помните, что когда фрагмент документа передается в метод `appendChild()`, к родительскому элементу добавляются все дочерние элементы фрагмента, но не сам фрагмент.

Если необходимо обновить DOM, иногда имеет смысл создать DOM-структуру с помощью фрагмента документа, чтобы можно было внести в документ сразу все изменения.

Используйте свойство innerHTML

Создавать DOM-узлы на странице можно двумя способами: с помощью DOM-методов, таких как `createElement()` и `appendChild()`, и с помощью свойства `innerHTML`. При небольших изменениях в DOM быстродействие в обоих случаях примерно одинаково, но при значительных изменениях второй способ гораздо быстрее.

Когда вы присваиваете значение свойству `innerHTML`, за кулисами создается синтаксический анализатор HTML, а структура DOM составляется с помощью встроенных методов DOM, а не методов, реализованных на JavaScript. Встроенные методы выполняются гораздо быстрее, потому что они уже скомпилированы, то есть их не нужно интерпретировать. Предыдущий пример можно переписать со свойством `innerHTML` следующим образом:

```
let list = document.getElementById("myList"),
    html = "";

for (let i=0; i < 10; i++) {
```

```
    html += '<li>Item ${i}</li>';
}

list.innerHTML = html;
```

Здесь мы создаем строку HTML-кода, а затем назначаем ее свойству `list.innerHTML`, формируя нужную DOM-структуру. Хотя конкатенация строк тоже требует некоторых ресурсов, эта методика все же работает быстрее, чем многократное взаимодействие с DOM.

Как и другие операции с DOM, обращения к свойству `innerHTML` нужно стараться свести к минимуму. Например, в следующем коде их слишком много:

```
let list = document.getElementById("myList");

for (let i=0; i < 10; i++) {
    list.innerHTML += '<li>Item ${i}</li>';    // НЕ ДЕЛАЙТЕ ТАК!!!
}
```

Проблема с этим кодом в том, что свойство `innerHTML` обновляется на каждой итерации цикла, что очень неэффективно. На самом деле обновление свойства `innerHTML` является динамическим, поэтому код будет работать гораздо быстрее, если сначала полностью составить строку, а затем назначить ее свойству за один раз.

> **ПРИМЕЧАНИЕ** `innerHTML` может предложить превосходную производительность, но он предоставляет огромную поверхность для XSS-атак. Каждый раз, когда он используется для интерполяции данных, которые явно не контролируются, злоумышленник может внедрить исполняемый код. Пользуйтесь им с осторожностью.

Делегируйте события

В большинстве веб-приложений для взаимодействия с пользователями используется множество обработчиков событий. Время отклика страницы на действия пользователя напрямую зависит от количества обработчиков на ней. Чтобы сократить эти накладные расходы, следует по мере возможности делегировать события.

Делегирование событий возможно благодаря тому, что некоторые из них всплывают. Любое всплывающее событие можно обработать не только у целевого элемента события, но и у любых его предков. Это позволяет обработчику, подключенному к элементу более высокого уровня, обрабатывать события с несколькими целевыми элементами. По возможности старайтесь подключать обработчики событий на уровне документа, чтобы обрабатывать события всей страницы.

Оптимизируйте доступ к объектам HTMLCollection

Из-за ряда недостатков, которые обсуждались во многих местах книги, объекты `HTMLCollection` существенно снижают быстродействие веб-приложений. Помните, что при каждом доступе к свойству или методу `HTMLCollection` выполняется запрос

документа, на что требуется довольно много ресурсов. Сведя к минимуму использование объектов HTMLCollection, можно заметно ускорить сценарий.

Пожалуй, важнее всего оптимизировать доступ к объектам HTMLCollection в циклах. Вычисление количества элементов при инициализации цикла for мы уже обсуждали, а теперь рассмотрим такой пример:

```
let images = document.getElementsByTagName("img");

for (let i=0, len=images.length; i < len; i++) {
    // обработка данных
}
```

Суть примера в том, что мы сохраняем значение length в переменной len, что позволяет не обращаться каждый раз к свойству length объекта HTMLCollection. Чтобы далее оптимизировать использование объекта HTMLCollection в цикле, следует получать ссылки на его элементы и работать с ними, а не обращаться к HTMLCollection непосредственно:

```
let images = document.getElementsByTagName("img"),
    image;

for (let i=0, len=images.length; i < len; i++) {
    image = images[i];
    // обработка данных
}
```

В этом цикле текущее изображение сохраняется в переменной image. После этого обращаться к объекту images типа HTMLCollection в цикле больше не требуется.

Чтобы свести к минимуму количество операций доступа к объектам HTMLCollection, важно понимать, когда они возвращаются. Это имеет место в следующих случаях:

- вызов метода getElementsByTagName();
- получение свойства childNodes элемента;
- получение свойства attributes элемента;
- доступ к специальному набору, такому как document.forms, document.images и т. д.

Грамотное использование объектов HTMLCollection может заметно повысить быстродействие кода.

РАЗВЕРТЫВАНИЕ

Наверное, наиболее важным этапом создания веб-сайта или веб-приложения на JavaScript является его развертывание. К этому моменту вы уже спроектировали решение, написали его код, оптимизировали и, наконец, готовы вывести его в интернет, чтобы пользователи смогли оценить его по достоинству. Но прежде чем сделать это, нужно разобраться с несколькими вопросами.

Процесс сборки

Для подготовки JS-кода к развертыванию очень важно продумать процесс его сборки. Типичный цикл разработки ПО включает этапы написания кода, его компиляции и тестирования, но поскольку JavaScript не компилируется, второй этап этого цикла выпадает, так что в браузере вы тестируете тот же код, который написали. К сожалению, этот подход не оптимален. По приведенным далее причинам написанный код не должен нетронутым попадать в браузер.

- **Права на интеллектуальную собственность.** Если вы поместите в интернет полностью прокомментированный исходный код, другим людям будет проще понять, что вы делаете, повторно использовать фрагменты кода и, возможно, даже атаковать ваше приложение.
- **Размеры файлов.** Разработчики пишут код так, чтобы его было легко читать. Это хорошо с точки зрения его сопровождения, но плохо в плане быстродействия. Дополнительные пробелы, отступы и подробные имена функций и переменных никак не помогают браузеру, а только замедляют его работу.
- **Организация кода.** Код, удобный для сопровождения, не всегда наиболее эффективно обрабатывается браузером.

По этим причинам следует хорошо продумать процесс сборки JS-файлов.

Структура файлов

Процесс сборки начинается с определения логической структуры для хранения файлов в системе управления исходным кодом. Писать весь JS-код в одном файле не рекомендуется — лучше придерживаться подхода, который обычен для объектно-ориентированных языков, и определять каждый объект и пользовательский тип в отдельном файле. Благодаря этому каждый файл будет содержать минимум кода, что поможет изменять его без внесения ошибок. Кроме того, в распределенных системах управления исходным кодом, таких как Git, CVS или Subversion, это снижает вероятность конфликтов во время операций слияния.

Помните, что разделение кода на несколько файлов выполняется для удобства его сопровождения, а не для развертывания. Для развертывания, наоборот, желательно объединить исходные файлы в один или несколько пакетов. В веб-приложениях рекомендуется использовать как можно меньше JS-файлов, потому что обработка HTTP-запросов — одно из основных «узких мест» в Сети. Не забывайте о том, что включение JS-файла в код страницы с помощью тега `<script>` — блокирующая операция, на время которой прекращается загрузка всех других файлов, и старайтесь для развертывания логически группировать JS-код в пакеты.

Автоматизаторы задач

Если вы собираете приложение, которое состоит из нескольких файлов, то, скорее всего, обнаружите, что вам нужен автоматизатор для запуска задач. Средство выполнения задач может запускать такие задачи, как связывание, пакетирование,

перенос, запуск локального сервера, развертывание или любую другую программу с использованием сценариев.

В большинстве случаев задания, выполняемые вашим автоматизатором задач, доступны через интерфейсы командной строки, и поэтому он будет просто инструментом, помогающим группировать и упорядочивать сложные вызовы командной строки. В этом смысле автоматизатор задач во многих отношениях очень похож на файл `.bashrc`. В других случаях инструменты, которые нужно использовать в автоматизированных задачах, будут иметь совместимые подключаемые модули.

Если вы используете NodeJS и npm для упаковки своих ресурсов JavaScript, пригодятся два популярных автоматизатора задач Grunt (www.gruntjs.com) и Gulp (www.gulpjs.com). Оба эти инструмента являются надежными автоматизаторами задач, чьи задания и инструкции определены в файлах конфигурации, написанных на простом JavaScript. Преимущество их использования заключается в том, что каждый из них обладает экосистемой плагинов, которые позволяют инструментам напрямую взаимодействовать с пакетами npm. Детали этих плагинов можно найти в приложениях.

Встряхивание дерева

Все более распространенной и чрезвычайно эффективной стратегией уменьшения размера полезной нагрузки является встряхивание дерева. Как упоминалось в главе 26 «Модули», использование стиля объявления статического модуля означает, что инструменты сборки могут определять, какие части кодовой базы зависят от других частей. Что еще более важно, встряхивание дерева также способно определить, какие части кодовой базы вообще не нужны.

Инструменты сборки, которые реализуют встряхивание дерева, подтверждают, что импорт модулей часто избирателен и что целые сегменты файлов модулей могут быть проигнорированы в конечном объединенном файле. Предположим, это пример из вашего приложения:

```
import { foo } from './utils.js';

console.log(foo);
export const foo = 'foo';
export const bar = 'bar';    // unused
```

Здесь экспорт `bar` никогда не используется, и статический анализ с помощью инструмента сборки может легко определить это. При выполнении встряхивания дерева инструмент сборки полностью удалит экспорт `bar` из связанного файла. Статический анализ также означает, что инструмент сборки может определять неиспользуемые зависимости и не включать их. После выполнения встряхивания дерева экономия размера файла возможного пакета может быть огромной.

Сборщик модулей

То, что ваша кодовая база написана в модулях, не означает, что она обязательно должна быть использована в качестве модулей. Часто кодовые базы JavaScript,

состоящие из большой коллекции модулей, объединяются во время сборки, и они служат одним или несколькими различными файлами JavaScript.

Работа сборщика модулей состоит в том, чтобы идентифицировать ландшафт зависимостей JavaScript, вовлеченных в приложение, объединить их в монолитное приложение, принять обоснованные решения о том, как модули должны быть последовательно организованы и объединены, и генерировать выходные файлы, которые будут предоставлены браузеру.

Существует множество инструментов для сборки, которые позволяют совершить такой подвиг. Webpack, Rollup и Browserify — это лишь некоторые из множества опций для преобразования кодовой базы на основе модулей в универсально совместимый сценарий страницы.

Проверка кода

Хотя уже начали появляться IDE с поддержкой JavaScript, большинство разработчиков все еще проверяет правильность кода, запуская его в браузере. Однако этот подход имеет несколько недостатков. Во-первых, такой способ проверки трудно автоматизировать или перенести в другую систему. Во-вторых, если не брать в расчет синтаксические ошибки, этот способ выявляет проблемы только при выполнении кода. Для обнаружения потенциальных проблем с JS-кодом есть несколько средств, наиболее популярным из которых является утилита *JSLint* (www.jslint.com), предложенная Дугласом Крокфордом (Douglas Crockford), и ESLint (www.eslint.org).

Утилиты ищут в коде синтаксические и некоторые другие часто допускаемые ошибки, в том числе следующие:

- использование функции `eval()`;
- использование необъявленных переменных;
- отсутствие точек с запятой;
- неправильные разрывы строк;
- неправильное использование запятых;
- отсутствие фигурных скобок;
- отсутствие ключевого слова `break` в инструкции `switch`;
- двойное объявление переменных;
- использование инструкции `with`;
- использование одинарных знаков равенства вместо двойных или тройных;
- недоступный код.

Добавление проверки кода в цикл разработки помогает избегать ошибок. Рекомендуется делать это также во время сборки для выявления потенциальных проблем, прежде чем они проявят себя как ошибки.

ПРИМЕЧАНИЕ Список средств проверки JS-кода приведен в приложении Г.

Сжатие

В контексте сжатия JS-файлов нас интересуют две величины: *размер кода* и *объем трафика*. Размер кода — это количество байтов, которые должен обработать браузер, а объем трафика — это количество байтов, передаваемых сервером браузеру. На заре веб-разработки эти два значения почти всегда были равны, потому что сервер передавал клиенту исходные файлы без каких-либо изменений. В наше время эти значения не совпадают почти никогда.

Минимизация кода

Поскольку JS-код не компилируется в байт-код и передается по сети в исходном виде, файлы с ним часто содержат дополнительные данные и элементы форматирования, которые не влияют на интерпретатор JavaScript в браузере. Минимизатор JavaScript выполнит преобразования исходного кода, чтобы сделать размер файла как можно меньшим при сохранении идентичного программного потока.

Комментарии, дополнительные пустые места и длинные имена переменных или функций упрощают чтение кода, но при отправке лишь увеличивают потребление ресурсов. Минимизатор может уменьшить размер файла.

При сжатии обычно выполняются следующие действия (все или некоторые из них):

- удаление ненужных пустот (включая разрывы строк);
- удаление всех комментариев;
- сокращение имен переменных.

ПРИМЕЧАНИЕ В контексте веб-разработки термин «минимизация» часто используется взаимозаменяемо с «сжатием». Это может считаться небольшой ошибкой, так как семантика каждого из них имеет мало общего.

Минимизация – это процесс, в котором минимизированный размер файла меньше исходного, но минимизированный файл все еще имеет синтаксически правильный код. Как правило, минимизация полезна только для интерпретируемых языков, таких как JavaScript, потому что языки, которые формально скомпилированы в двоичный файл, будут минимизированы компилятором как само собой разумеющееся.

Сжатие отличается от минимизации тем, что сжатый файл также меньше исходного, но не является синтаксически правильным кодом. Сжатый файл должен быть распакован для восстановления формы читаемого кода. Сжатие приведет к меньшему размеру файла, чем при минимизации, поскольку алгоритмы сжатия не должны сохранять синтаксическую структуру файла и, следовательно, могут почувствовать большую свободу.

Все файлы JavaScript должны быть минимизированы с помощью инструмента минимизации перед развертыванием в производственной среде. Добавление шага минимизации в процесс сборки для сжатия файлов JavaScript — простой способ убедиться, что ничего не будет пропущено.

Компиляция JavaScript

Подобно минимизации, компиляция кода обычно относится к процессу получения исходного кода и его преобразования в форму, которая идентична по поведению, но использует меньше байтов JavaScript. Она отличается от минимизации тем, что структура кода после компиляции может отличаться, но он все равно будет демонстрировать то же поведение, что и исходный код. Компиляторы могут выполнить это, приняв весь код JavaScript и выполнив тщательный анализ потока программы.

Компиляция может выполнять некоторые из следующих операций:

- удаление неиспользуемого кода;
- преобразование части кода для использования более краткого синтаксиса;
- глобальное встраивание вызовов функций, констант и переменных.

Транспиляция JavaScript

Код в репозитории проекта почти никогда не будет точной копией кода, который будет выполняться в вашем браузере. ES6, ES7 и ES8 предоставляют замечательные возможности в спецификации ECMAScript, но разные браузеры будут полностью реализовывать каждую из своих функций в разном темпе.

Использование транспиляции позволит использовать все новейшие функции синтаксической спецификации, не беспокоясь об обратной совместимости браузера. Вы можете перенести свой современный код в более старую версию ECMAScript — обычно ES3 или ES5, в зависимости от ваших потребностей, — чтобы код мог работать везде. Инструменты для транспиляции описаны в приложениях.

> **ПРИМЕЧАНИЕ** Термины «транспиляция» и «компиляция» часто используются взаимозаменяемо. Компиляция – это процесс преобразования исходного кода, написанного на одном языке, на другой язык. По сути, транспиляция – это тот же процесс, что и компиляция, но конечный язык будет иметь такой же уровень абстракции, что и исходный. Следовательно, технически преобразование кода ES6/7/8 в ES3/5 является как компиляцией, так и транспиляцией, хотя транспиляция является более точным термином для описания процесса.

Сжатие HTTP-трафика

Объем трафика — это фактическое количество байтов, отправляемых сервером браузеру. Оно может не совпадать с размером кода из-за его сжатия на сервере. Все пять основных веб-браузеров — Internet Explorer/Edge, Firefox, Safari, Chrome

и Opera — поддерживают декомпрессию полученных ресурсов, благодаря чему на сервере можно смело сжимать JS-файлы. В отправляемый ответ сервер добавляет заголовок, указывающий, что файл был сжат в конкретном формате. Получив ответ, браузер изучает этот заголовок, выясняет, что файл был сжат, и восстанавливает его в исходном виде. Благодаря этому количество байтов, передаваемых по сети, значительно меньше, чем размер оригинального кода.

Для примера, использование двух модулей, доступных для веб-сервера Apache (`mod_gzip` и `mod_deflate`), приведет к сжатию оригинальных JS-файлов примерно на 70 %. Такая высокая эффективность сжатия достигается благодаря тому, что JS-файлы содержат обычный текст. Сокращение объема трафика ускоряет доставку файлов браузеру, но при этом имеет место небольшой компромисс, потому что сервер тратит некоторое время на сжатие файлов, а браузер — на их декомпрессию. Как правило, выгода от сжатия перевешивает эти накладные расходы.

> **ПРИМЕЧАНИЕ** Большинство веб-серверов – как с открытым исходным кодом, так и коммерческих, – поддерживают сжатие HTTP-трафика. Сведения о том, как настроить сжатие на конкретном сервере, можно найти в его документации.

ИТОГИ

С развитием JavaScript формировались лучшие практики программирования на этом языке. То, что когда-то считалось хобби, стало полноценным направлением разработки, потребовавшим изучения вопросов удобства сопровождения кода, его быстродействия, развертывания приложений и т. д.

Удобство сопровождения JS-кода частично связано с конвенциями кодирования. Упомянем некоторые нюансы их применения:

- При использовании комментариев и отступов в JavaScript можно следовать конвенциям из других языков, однако слабо типизированная природа JavaScript требует также соблюдения некоторых специальных правил.
- Поскольку JavaScript используется вместе с HTML и CSS, важно четко разграничивать области применения этих технологий: код JavaScript должен определять поведение страницы, HTML — ее контент, а CSS — вид. Смешение этих областей может приводить к неуловимым ошибкам и проблемам сопровождения кода.

С увеличением объема JS-кода в веб-приложениях одной из важнейших его характеристик стало быстродействие. В связи с этим желательно помнить о некоторых аспектах:

- Время выполнения JS-кода напрямую влияет на общее быстродействие веб-страницы.
- Многие рекомендации по оптимизации кода для C-подобных языков относятся и к JavaScript. Примерами могут служить приемы оптимизации циклов и использование инструкций `switch` вместо `if`.

- На взаимодействие с DOM требуется много ресурсов, поэтому количество операций с DOM желательно свести к минимуму.

Логичным итогом разработки приложения является его развертывание. В связи с этим важно отметить следующее:

- Чтобы упростить развертывание, продумайте процесс сборки, объединяющий JS-файлы в меньшее количество файлов (в идеале — один).
- Наличие процесса сборки позволяет автоматически запускать для исходного кода дополнительные процессы и фильтры. Например, вы можете запустить средство проверки JS-кода, чтобы убедиться, что в нем нет синтаксических и других ошибок.
- Средство сжатия JS-кода может уменьшить его размер настолько, насколько это возможно перед развертыванием.
- Дополнив сжатие кода сжатием HTTP-трафика, можно добиться максимальной экономии ресурсов сети при минимальном влиянии на быстродействие страницы.

М. Фрисби

JavaScript для профессиональных веб-разработчиков

4-е международное издание

Перевел с английского *А. Павлов*

Заведующая редакцией	*Ю. Сергиенко*
Ведущий редактор	*А. Юринова*
Литературные редакторы	*М. Петруненко, А. Попова*
Художественный редактор	*В. Мостипан*
Корректоры	*С. Беляева, Н. Сидорова*
Верстка	*Л. Егорова*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 08.2021. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12.000 —
Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 30.07.21. Формат 70×100/16. Бумага офсетная. Усл. п. л. 94,170. Тираж 700. Заказ 0000.

Приложения

A

ES2018 и ES2019

Начиная с ECMAScript 2015, комитет TC-39 начал выпускать новую спецификацию ECMA каждый год. Это позволяет собрать все отдельные предложения, которые находятся на достаточно продвинутой стадии, и упаковать их в единый пакет. Однако эта упаковка имеет ограниченное значение, поскольку производители браузеров, как правило, принимают предложения по частям. Когда предложение достигнет стадии 4, его поведение не изменится, только, скорее всего, оно будет включено в следующую версию ECMAScript и браузеры начнут применять функции предложения по своему усмотрению.

Предложение ECMAScript 2018 было завершено в январе 2018 года и содержит улучшения для асинхронной итерации, операторов остатка и распространения, регулярных выражений и промисов. TC-39 поддерживает GitHub-репозиторий (https://github.com/tc39/ecma262), который можно использовать для отслеживания состояния различных предложений:

TODO FIX ASYNC ITERATION

TODO ES2019 http://exploringjs.com/es2018-es2019/toc.html

> **ПРИМЕЧАНИЕ** Поскольку функции, описанные в этой главе, новые, они будут поддерживаться браузером в ограниченном виде (если вообще будут). Обратитесь к https://caniuse.com/, чтобы определить, поддерживает ли версия браузера определенную функцию.

АСИНХРОННАЯ ИТЕРАЦИЯ

Асинхронное выполнение и протокол итератора — две чрезвычайно распространенные темы в новых функциях ECMAScript последних выпусков. Асинхронное выполнение включает в себя высвобождение контроля над потоком выполнения, чтобы позволить

медленным операциям завершаться до восстановления управления, а протокол итератора включает определение канонического порядка для произвольных объектов. Асинхронная итерация — это просто логическое усвоение этих двух понятий.

Синхронный итератор предоставляет пару `{value, done}` каждый раз при вызове `next()`. Конечно, это требует, чтобы вычисления и извлечения ресурсов, необходимые для определения содержимого этой пары, были завершены к моменту выхода из вызова `next()`, иначе эти значения не будут определены. При использовании *синхронного* итератора для перебора значений, определенных *асинхронно*, основной поток выполнения будет заблокирован в ожидании завершения асинхронной операции.

С асинхронными итераторами эта проблема полностью решена. Асинхронный итератор предоставляет промис, который разрешается в пару `{value, done}` каждый раз при вызове `next()`. Таким образом, поток выполнения может быть освобожден и выполнит работу в другом месте, пока разрешается текущая итерация цикла.

Создание и использование асинхронного итератора

Асинхронные итераторы проще всего понять при сравнении с традиционными синхронными итераторами. Ниже приведен простой класс `Emitter`, который содержит функцию синхронного генератора, создающего синхронный итератор, который будет считать от 0 до 4:

```
class Emitter {
    constructor(max) {
        this.max = max;
        this.syncIdx = 0;
    }

    *[Symbol.iterator]() {
        while(this.syncIdx < this.max) {
            yield this.syncIdx++;
        }
    }
}

const emitter = new Emitter(5);

function syncCount() {
    const syncCounter = emitter[Symbol.iterator]();

    for (const x of syncCounter) {
        console.log(x);
    }
}

syncCount();
// 0
// 1
// 2
// 3
// 4
```

Предыдущий пример работает только потому, что на каждой итерации следующее значение может быть сразу же получено. Если вместо этого вы не хотите блокировать основной поток выполнения при определении следующего значения для выдачи, вы также можете определить функцию асинхронного генератора, которая будет выдавать значения, обернутые в промис.

Это может быть выполнено с использованием асинхронных версий итераторов и генераторов. ECMAScript 2018 определяет `Symbol.asyncIterator`, который позволяет определять и вызывать функции генерации Promise. В спецификации также представлен асинхронный итератор цикла `for`, цикл `for-await-of`, предназначенный для использования этого асинхронного итератора.

С их использованием предыдущий пример может быть расширен для поддержки синхронной и асинхронной итерации:

```
class Emitter {
    constructor(max) {
        this.max = max;
        this.syncIdx = 0;
        this.asyncIdx = 0;
    }

    *[Symbol.iterator]() {
        while(this.syncIdx < this.max) {
            yield this.syncIdx++;
        }
    }
    async *[Symbol.asyncIterator]() {
        *[Symbol.asyncIterator]() {
            while(this.asyncIdx < this.max) {
                yield new Promise((resolve) => resolve(this.asyncIdx++;
            }
        }
    }

    const emitter = new Emitter(5);

    function syncCount() {
        const syncCounter = emitter[Symbol.iterator]();

        for (const x of syncCounter) {
            console.log(x);
        }
    }

    async function asyncCount() {
        const asyncCounter = emitter[Symbol.asyncIterator]();

        for await(const x of asyncCounter) {
            console.log(x);
        }
    }

syncCount();
```

```
// 0
// 1
// 2
// 3
// 4

asyncCount();
// 0
// 1
// 2
// 3
// 4
```

Для дальнейшего понимания поменяйте приведенный выше пример так, чтобы синхронный генератор был передан в цикл `for-await-of`:

```
const emitter = new Emitter(5);

async function asyncIteratorSyncCount() {
    const syncCounter = emitter[Symbol.iterator]();

    for await(const x of syncCounter) {
        console.log(x);
    }
}

asyncIteratorSyncCount();
// 0
// 1
// AsyncIteratorExample01.js
2
// 3
// 4
```

Даже несмотря на то что синхронный счетчик перебирает примитивные значения, цикл `for-await-of` будет обрабатывать значения, как если бы они были возвращены, завернутые в промисы. Это демонстрирует мощь цикла ожидания, который позволяет ему свободно обрабатывать как синхронные, так и асинхронные итерации. Это неверно для обычного цикла `for`, который не может обрабатывать асинхронный итератор:

```
function syncIteratorAsyncCount() {
    const asyncCounter = emitter[Symbol.asyncIterator]();

    for (const x of asyncCounter) {
        console.log(x);
    }
}

syncIteratorAsyncCount();
// TypeError: asyncCounter is not iterable
```

Одна из наиболее важных концепций асинхронных итераторов состоит в том, что обозначение `Symbol.asyncIterator` не изменяет поведение функции генератора или

способ его использования. Обратите внимание, что функция генератора определяется как асинхронная функция и обозначается как генератор с использованием звездочки. `Symbol.asyncIterator` просто обещает внешней конструкции, такой как цикл ожидания, что связанный итератор возвратит объекты-промисы последовательности.

Понимание очереди асинхронного итератора

Конечно, предыдущий пример является довольно надуманным, так как промисы, возвращаемые итератором, мгновенно разрешаются, и поэтому он представляет собой не что иное, как тонко завернутый синхронный итератор. Предположим вместо этого, что полученные промисы разрешаются через неопределенный период времени; более того, предположим, что они возвращаются вне очереди. Асинхронный итератор должен эмулировать синхронный итератор всеми возможными способами, включая выполнение по порядку кода, связанного с каждой итерацией. Для решения этой проблемы асинхронные итераторы поддерживают очередь обратных вызовов, чтобы гарантировать, что обработчик итератора для более раннего значения всегда завершит выполнение, прежде чем перейти к более позднему значению, даже если более позднее значение разрешается до более раннего значения.

Чтобы доказать это, асинхронный итератор в следующем примере возвращает промисы, которые разрешаются через произвольный период времени. Асинхронная очередь итераций гарантирует, что порядок разрешения промисов не влияет на порядок итераций. В результате целые числа будут напечатаны в нужном порядке (через случайные интервалы):

```
class Emitter {
    constructor(max) {
        this.max = max;
        this.syncIdx = 0;
        this.asyncIdx = 0;
    }

    *[Symbol.iterator]() {
        while(this.syncIdx < this.max) {
            yield this.syncIdx++;
        }
    }
    async *[Symbol.asyncIterator]() {
        while(this.asyncIdx < this.max) {
            yield new Promise((resolve) => {
                setTimeout(() => {
                    resolve(this.asyncIdx++);
                }, Math.floor(Math.random() * 1000));
            });
        }
    }
}

const emitter = new Emitter(5);
```

```
function syncCount() {
    const syncCounter = emitter[Symbol.iterator]();

    for (const x of syncCounter) {
        console.log(x);
    }
}

async function asyncCount() {
    const asyncCounter = emitter[Symbol.asyncIterator]();

    for await(const x of asyncCounter) {
        console.log(x);
    }
}

syncCount();
// 0
// 1
// 2
// 3
// 4

asyncCount();
// 0
// 1
// 2
// 3
// 4
AsyncIteratorExample02.js
```

Обработка reject() в асинхронном итераторе

Поскольку композиция асинхронных итераторов состоит из промисов, необходимо учитывать возможность отклонения одного из промисов, созданных итератором. Поскольку проект асинхронной итерации требует завершения по порядку, нет смысла проходить через отклоненный промис в цикле; следовательно, отклоненный промис заставит итератор завершиться:

```
class Emitter {
    constructor(max) {
        this.max = max;
        this.asyncIdx = 0;
    }

    async *[Symbol.asyncIterator]() {
        while (this.asyncIdx < this.max) {
            if (this.asyncIdx < 3) {
                yield this.asyncIdx++;
            } else {
                throw 'Exited loop';
            }
        }
```

```
    }
}

const emitter = new Emitter(5);

async function asyncCount() {
    const asyncCounter = emitter[Symbol.asyncIterator]();

    for await (const x of asyncCounter) {
        console.log(x);
    }
}

asyncCount();
// 0
// 1
// 2
// Uncaught (in promise) Exited loop
```

Ручное управление асинхронным итератором с использованием next()

Цикл `for-await-of` предлагает две полезные функции: он использует очередь асинхронных итераторов для обеспечения порядка выполнения и скрывает структуру промисов асинхронных итераторов. Однако использование такого цикла скрывает большую часть базового поведения.

Поскольку асинхронный итератор по-прежнему следует протоколу итератора, можно также легко пройти асинхронную итерацию, используя `next()`. Как описано ранее, `next()` содержит промис, который преобразуется в `{value, done}`. Это означает, что нужно использовать Promise API для извлечения методов, но это также означает, что не обязательно использовать очередь асинхронных итераторов.

```
const emitter = new Emitter(5);

const asyncCounter = emitter[Symbol.asyncIterator]();

console.log(asyncCounter.next());
// { value: Promise, done: false }
```

Асинхронные циклы верхнего уровня

Как правило, асинхронное поведение, включая циклы `for-await-of`, не может существовать вне асинхронной функции. Тем не менее может возникнуть необходимость использовать асинхронное поведение в таком контексте. Это может быть достигнуто путем создания асинхронного IIFE:

```
class Emitter {
    constructor(max) {
        this.max = max;
        this.asyncIdx = 0;
```

```
    }

    async *[Symbol.asyncIterator]() {
        while(this.asyncIdx < this.max) {
            yield new Promise((resolve) => resolve(this.asyncIdx++));
        }
    }
}

const emitter = new Emitter(5);

(async function() {
    const asyncCounter = emitter[Symbol.asyncIterator]();

    for await(const x of asyncCounter) {
        console.log(x);
    }
})();

// 0
// 1
// 2
// 3
// 4
```

Реализация наблюдаемых объектов

Поскольку асинхронные итераторы будут терпеливо ждать следующей итерации, не неся при этом вычислительных затрат, открывается совершенно новый путь для реализации наблюдаемого интерфейса. На высоком уровне она примет форму захвата событий, оборачивания их в промисы, а затем передачи этих событий через итератор, чтобы позволить наблюдателю подключиться к асинхронному итератору. Когда событие запускается, следующий промис в асинхронном итераторе разрешается с этим событием.

> **ПРИМЕЧАНИЕ** Тема наблюдаемых объектов выходит за рамки этой книги, потому что они в основном реализованы в сторонних библиотеках. Для дальнейшего изучения загляните в чрезвычайно популярную библиотеку RxJS (http://reactivex.io/rxjs/).

Упрощенным примером этого будет захват видимого потока событий браузера. Для этого требуется очередь промисов, каждый из которых соответствует отдельному событию. Очередь также сохранит порядок, в котором генерируются события, что является желательным для такого рода проблем.

```
class Observable {
    constructor() {
        this.promiseQueue = [];

        // Содержит разрешатель для следующего промиса в очереди
        this.resolve = null;
```

```
        // Выдвигает начальный промис в очереди, который будет
        // разрешен с первым наблюдаемым событием
        this.enqueue();
    }

    // Создание нового промиса, сохранение его метода resolve и
    // хранение его в очереди
    enqueue() {
        this.promiseQueue.push(
            new Promise((resolve) => this.resolve = resolve));
    }

    // Удаление промиса из начала очереди и
    // его возврат
    dequeue() {
        return this.promiseQueue.shift();
    }
}
```

Чтобы использовать этот вывод из очереди промисов, определите метод асинхронного генератора для класса. Этот генератор должен работать для любого типа события:

```
class Observable {
    constructor() {
        this.promiseQueue = [];

        // Содержит разрешатель для следующего промиса в очереди
        this.resolve = null;

        // Выдвигает начальный промис в очереди, который будет
        // разрешен с первым наблюдаемым событием
        this.enqueue();
    }

    // Создание нового промиса, сохранение его метода resolve и
    // хранение его в очереди
    enqueue() {
        this.promiseQueue.push(
            new Promise((resolve) => this.resolve = resolve));
    }

    // Удаление промиса из начала очереди и
    // его возврат
    dequeue() {
        return this.promiseQueue.shift();
    }

    async *fromEvent (element, eventType) {
        // Всякий раз, когда генерируется событие, промис в начале очереди
        // разрешается с помощью объекта события и
        // в очередь помещается другой промис.
        element.addEventListener(eventType, (event) => {
            this.resolve(event);
            this.enqueue();
        });
```

```
        // Каждый разрешенный промис в начале очереди будет
        // передавать объект события асинхронному итератору.
        while (1) {
            yield await this.dequeue();
        }
    }
}
```

С этим полностью определенным классом теперь можно легко определить наблюдаемый объект на элементах DOM. Предположим, что на странице есть кнопка `<button>`; можно записать поток событий `click` на этой кнопке и вывести каждое из них в консоль следующим образом:

```
class Observable {
    constructor() {
        this.promiseQueue = [];

        // Содержит разрешатель для следующего промиса в очереди
        this.resolve = null;

        // Выдвигает начальный промис в очереди, который будет
        // разрешен с первым наблюдаемым событием
        this.enqueue();
    }

    // Создание нового промиса, сохранение его метода resolve и
    // хранение его в очереди
    enqueue() {
        this.promiseQueue.push(
            new Promise((resolve) => this.resolve = resolve));
    }

    // Удаление промиса из начала очереди и
    // его возврат
    dequeue() {
        return this.promiseQueue.shift();
    }

    async *fromEvent (element, eventType) {
        // Всякий раз, когда генерируется событие, промис в начале очереди
        // разрешается с помощью объекта события и
        // в очередь помещается другой промис.
        element.addEventListener(eventType, (event) => {
            this.resolve(event);
            this.enqueue();
        });

        // Каждый разрешенный промис в начале очереди будет
        // передавать объект события асинхронному итератору.
        while (1) {
            yield await this.dequeue();
        }
    }
}
```

```
(async function() {
    const observable = new Observable();

    const button = document.querySelector('button');
    const mouseClickIterator = observable.fromEvent(button, 'click');

    for await (const clickEvent of mouseClickIterator) {
        console.log(clickEvent);
    }
})();
```

ОПЕРАТОРЫ ОСТАТКА И РАСПРОСТРАНЕНИЯ ДЛЯ ЛИТЕРАЛОВ ОБЪЕКТОВ

В спецификации ECMAScript 2018 вся элегантность операторов остатка и распространения в массивах теперь также доступна внутри литералов объектов. Это позволяет объединять объекты или собирать свойства в новые объекты.

Оператор остатка

Оператор остатка позволяет использовать один оператор при деструктурировании объекта, чтобы собрать все оставшиеся неуказанные перечислимые свойства в один объект. Это можно сделать следующим образом:

```
const person = { name: 'Matt', age: 27, job: 'Engineer' };
const { name, ...remainingData } = person;

console.log(name);              // Matt
console.log(remainingData);     // { age: 27, job: 'Engineer' }
```

Оператор остатка может использоваться не более одного раза для каждого литерала объекта и должен быть указан последним. Поскольку для каждого литерала объекта может существовать только один оператор остатка, можно использовать вложенные операторы остатка. При вложении из-за отсутствия двусмысленности относительно того, какие элементы поддерева свойств выделяются любому заданному оператору остатка, результирующие объекты никогда не будут перекрываться по своему содержанию:

```
const person = { name: 'Matt', age: 27, job: { title: 'Engineer', level: 10 } };

const { name, job: { title, ...remainingJobData }, ...remainingPersonData }
    = person;

console.log(name);                  // Matt
console.log(title);                 // Engineer
console.log(remainingPersonData);   // { age: 27 }
console.log(remainingJobData);      // { level: 10 }

const { ...a, job } = person;
// SyntaxError: Rest element must be last element
```

Оператор остатка выполняет поверхностное копирование между объектами, поэтому ссылки на объекты будут копироваться вместо создания полных копий объектов:

```
const person = { name: 'Matt', age: 27, job: { title: 'Engineer', level: 10 } };

const { ...remainingData } = person;

console.log(person === remainingData);            // false
console.log(person.job === remainingData.job);    // true
```

Оператор остатка скопирует все перечисляемые собственные свойства, включая символы:

```
const s = Symbol();
const foo = { a: 1, [s]: 2, b: 3 }

const {a, ...remainingData} = foo;

console.log(remainingData);
// { b: 3, Symbol(): 2 }
```

Оператор распространения

Оператор распространения позволяет объединять два объекта таким же образом, как и конкатенация массивов. Оператор распространения, примененный к внутреннему объекту, выполнит поверхностное копирование всех перечисляемых собственных свойств, включая символы, во внешний объект:

```
const s = Symbol();
const foo = { a: 1 };
const bar = { [s]: 2 };

const foobar = {...foo, c: 3, ...bar};

console.log(foobar);
// { a: 1, c: 3 Symbol(): 2 }
```

Порядок, в котором распределяются объекты, имеет значение по двум причинам:

1. Объекты отслеживают порядок вставки. Свойства, скопированные из распространяемых объектов, будут выполняться в том порядке, в котором они перечислены внутри литерала объекта.
2. Объекты будут перезаписывать свойства при обнаружении дубликатов. Последнее найденное свойство будет тем, значение которого будет записано.

Эти соглашения о порядке показаны здесь:

```
const foo = { a: 1 };
const bar = { b: 2 };
const foobar = {c: 3, ...bar, ...foo};

console.log(foobar);
// { c: 3, b: 2, a: 1}
```

```
const baz = { c: 4 };

const foobarbaz = {...foo, ...bar, c: 3, ...baz };

console.log(foobarbaz);
// { a: 1, b: 2, c: 4 }
```

Как и в случае с оператором остатка, все выполняемые копии являются поверхностными:

```
const foo = { a: 1 };
const bar = { b: 2, c: { d: 3 } };

const foobar = {...foo, ...bar};

console.log(foobar.c === bar.c); // true
```

ОПИСАНИЕ FINALLY() В ПРОМИСАХ

Раньше существовали только неэлегантные способы определения поведения, которое имело бы место после того, как промис выходит из состояния «ожидания» независимо от результата. Обычно оно принимает форму утилизации обработчика:

```
let resolveA, rejectB;

function finalHandler() {
    console.log('finished');
}

function resolveHandler(val) {
    console.log('resolved');
    finalHandler();
}

function rejectHandler(err) {
    console.log('rejected');
    finalHandler();
}

new Promise((resolve, reject) => {
    resolveA = resolve;
})
.then(resolveHandler, rejectHandler);

new Promise((resolve, reject) => {
    rejectB = reject;
})
.then(resolveHandler, rejectHandler);

resolveA();
rejectB();
// resolved
```

```
// finished
// rejected
// finished
```

С `Promise.prototype.finally()` можно объединить все в один общий обработчик. Обработчику `finally()` не передаются никакие аргументы и он не знает, обрабатывает он разрешенный или отклоненный промис. Предыдущий пример может быть изменен следующим образом:

```
let resolveA, rejectB;

function finalHandler() {
    console.log('finished');
}

function resolveHandler(val) {
    console.log('resolved');
}

function rejectHandler(err) {
    console.log('rejected');
}

new Promise((resolve, reject) => {
    resolveA = resolve;
})
.then(resolveHandler, rejectHandler);
.finally(finalHandler);

new Promise((resolve, reject) => {
    rejectB = reject;
})
.then(resolveHandler, rejectHandler);
.finally(finalHandler);

resolveA();
rejectB();
// resolved
// rejected
// finished
// finished
```

Здесь вы заметите, что порядок ведения журнала изменился. Каждый `finally()` создает новый экземпляр промиса, и эти новые промисы добавляются в очередь микрозадач браузера и разрешаются только после разрешения предыдущих промисов обработчика.

РАСШИРЕНИЕ РЕГУЛЯРНЫХ ВЫРАЖЕНИЙ

ECMAScript 2018 содержит несколько новых наворотов для регулярных выражений.

Флаг dotAll

Одна досадная особенность регулярных выражений заключалась в том, что токен совпадения с одним символом (точка «.») не соответствовал символам конца строки, таким как \n и \r, или не-BMP-символам, таким как эмодзи.

```
const text = `
foo
bar
`;

const re = /foo.bar/;

console.log(re.test(text));    // false
```

В этом предложении вводится флаг «s» (обозначает одиночную линию), который исправляет это поведение:

```
const text = `
foo
bar
`;

const re = /foo.bar/s;

console.log(re.test(text));    // true
```

Ретроспективные проверки

Регулярные выражения поддерживают как положительные, так и отрицательные опережающие проверки, которые позволяют декларировать ожидания после сопоставления сегментов:

```
const text = 'foobar';

// Положительная опережающая проверка
// Утверждение, что значение следует за указанным, но не захватывается
const rePositiveMatch = /foo(?=bar)/;
const rePositiveNoMatch = /foo(?=baz)/;

console.log(rePositiveMatch.exec(text));
// ["foo"]

console.log(rePositiveNoMatch.exec(text));
// null

// Отрицательная опережающая проверка
// Утверждение, что значение не следует за указанным, но не захватывается
const reNegativeNoMatch = /foo(?!bar)/;
const reNegativeMatch = /foo(?!baz)/;

console.log(reNegativeNoMatch.exec(text));
// null

console.log(reNegativeMatch.exec(text));
// ["foo"]
```

Новое предложение вводит зеркальное отражение этих проверок, положительные и отрицательные ретроспективные проверки. Они работают идентично опережающим проверкам, за исключением того, что они работают для проверки содержимого, предшествующего сопоставленным сегментам:

```
const text = 'foobar';

// Положительная ретроспективная проверка
// Утверждение, что значение предшествует указанному, но не захватывается
const rePositiveMatch = /(?<=foo)bar/;
const rePositiveNoMatch = /(?<=baz)bar/;

console.log(rePositiveMatch.exec(text));
// ["bar"]

console.log(rePositiveNoMatch.exec(text));
// null

// Отриательная ретроспективная проверка
// Утверждение, что значение не предшествует указанному, но не захватывается
const reNegativeNoMatch = /(?<!foo)bar/;
const reNegativeMatch = /(?<!baz)bar/;

console.log(reNegativeNoMatch.exec(text));
// null

console.log(reNegativeMatch.exec(text));
// ["bar"]
```

Именованные группы захвата

Как правило, в нескольких группах захвата обращение выполнялось по индексу, что оказалось ужасным разочарованием, потому что индексы не дают контекста относительно того, что они на самом деле содержат:

```
const text = '2018-03-14';

const re = /(\d+)-(\d+)-(\d+)/;

console.log(re.exec(text));
//["2018-03-14", "2018", "03", "14"]
```

Предложение позволяет связать действительный идентификатор JavaScript с группой захвата, которую затем можно извлечь из свойства `groups` результата:

```
const text = '2018-03-14';

const re = /(?<year>\d+)-(?<month>\d+)-(?<day>\d+)/;

console.log(re.exec(text).groups);
// { year: "2018", month: "03", day: "14" }
```

Экранирование свойств Unicode

Стандарт Unicode определяет свойства для каждого символа. Свойства символов, такие как имя символа, категории, обозначение пробелов, а также сценарий или язык, внутри которого определяется символ, доступны как свойства символов. Экранирование свойств Unicode позволяют использовать эти свойства внутри регулярных выражений.

Некоторые свойства являются двоичными, что означает, что они могут применяться автономно. Примерами этого являются `Uppercase` и `White_Space`. Другие свойства ведут себя как пары ключ/значение, где свойство будет соответствовать значению свойства. Примером этого является `Script_Extensions=Greek`.

Список свойств Unicode можно найти по адресу http://unicode.org/Public/UNIDATA/PropertyAliases.txt.

Список значений свойств Unicode можно найти по адресу http://unicode.org/Public/UNIDATA/PropertyValueAliases.txt.

Экранирование свойств Unicode в регулярных выражениях может использовать `\p` для выбора соответствия или `\P` для выбора несоответствия:

```
const pi = String.fromCharCode(0x03C0);
const linereturn = `
`;

const reWhiteSpace = /\p{White_Space}/u;
const reGreek = /\p{Script_Extensions=Greek}/u;
const reNotWhiteSpace = /\P{White_Space}/u;
const reNotGreek = /\P{Script_Extensions=Greek}/u;

console.log(reWhiteSpace.test(pi));               // false
console.log(reWhiteSpace.test(linereturn));       // true
console.log(reNotWhiteSpace.test(pi));            // true
console.log(reNotWhiteSpace.test(linereturn));    // false

console.log(reGreek.test(pi));               // true
console.log(reGreek.test(linereturn));       // false
console.log(reNotGreek.test(pi));            // false
console.log(reNotGreek.test(linereturn));    // true
```

МЕТОДЫ СГЛАЖИВАНИЯ МАССИВОВ

ECMAScript 2019 добавил два метода в прототип `Array`, `flat()` и `flatMap()`, которые значительно упрощают операции сглаживания массива. Без этих методов сглаживание — это неприятное дело, которое требует итеративного или рекурсивного решения.

> **ПРИМЕЧАНИЕ** `flat()` и `flatMap()` строго ограничены сглаживанием вложенных массивов. Вложенные итерируемые объекты, такие как `Map` и `Set`, не будут сглажены.

Array.prototype.flatten()

Ниже приведен пример того, как может выглядеть простая рекурсивная реализация без использования этих новых методов:

```
function flatten(sourceArray, flattenedArray = []) {
    for (const element of sourceArray) {
        if (Array.isArray(element)) {
            flatten(element, flattenedArray);
        } else {
            flattenedArray.push(element);
        }
    }

    return flattenedArray;
}

const arr = [[0], 1, 2, [3, [4, 5]], 6];

console.log(flatten(arr))
// [0, 1, 2, 3, 4, 5, 6]
```

Во многих отношениях этот пример напоминает древовидную структуру данных; каждый элемент в массиве ведет себя как дочерний узел, а элементы, не являющиеся массивами, являются листьями. Следовательно, в этом примере входной массив представляет собой дерево высотой 2 с 7 листьями. Сглаживание этого массива по сути является упорядоченным обходом листьев.

Иногда полезно иметь возможность указать, сколько уровней вложенности массива должно быть сглажено. Рассмотрим следующий пример, который изменяет исходную реализацию и позволяет указать глубину выравнивания:

```
function flatten(sourceArray, depth, flattenedArray = []) {
    for (const element of sourceArray) {
        if (Array.isArray(element) && depth > 0) {
            flatten(element, depth - 1, flattenedArray);
        } else {
            flattenedArray.push(element);
        }
    }

    return flattenedArray;
}

const arr = [[0], 1, 2, [3, [4, 5]], 6];

console.log(flatten(arr, 1))
// [0, 1, 2, 3, [4, 5], 6]
```

Для решения этих случаев использования был добавлен метод `Array.prototype.flat()`. Он принимает аргумент `depth` (по умолчанию с глубиной 1) и возвращает поверхностную копию экземпляра `Array`, сглаженного до указанной глубины. Это показано здесь:

```
const arr = [[0], 1, 2, [3, [4, 5]], 6];

console.log(arr.flat(2));
// [0, 1, 2, 3, 4, 5, 6]

console.log(arr.flat());
// [0, 1, 2, 3, [4, 5], 6]
```

Поскольку выполняется поверхностное копирование, массивы с циклами будут копировать значения из исходного массива при сглаживании:

```
const arr = [[0], 1, 2, [3, [4, 5]], 6];

arr.push(arr);

console.log(arr.flat());
// [0, 1, 2, 3, 4, 5, 6, [0], 1, 2, [3, [4, 5]], 6]
```

Array.prototype.flatMap()

Метод `Array.prototype.flatMap()` позволяет выполнить операцию отображения перед сглаживанием массива. `arr.flatMap(f)` функционально эквивалентен `arr.map(f).flat()`, но `arr.flatMap()` более эффективен, поскольку браузер должен выполнять только один обход.

Сигнатура функции `flatMap()` идентична `map()`. Простой пример выглядит так:

```
const arr = [[1], [3], [5]];

console.log(arr.map(([x]) => [x, x + 1]));
// [[1, 2], [3, 4], [5, 6]]

console.log(arr.flatMap(([x]) => [x, x + 1]));
// [1, 2, 3, 4, 5, 6]
```

`flatMap()` особенно полезен в ситуациях, когда метод объекта, не являющегося массивом, возвращает массив — к примеру, `split()`. Рассмотрим следующий пример, где набор входных строк разбивается на слова и объединяется в массив из одного слова:

```
const arr = ['Lorem ipsum dolor sit amet,', 'consectetur adipiscing elit.'];

console.log(arr.flatMap((x) => x.split(/[\W+]/)));
// ["Lorem", "ipsum", "dolor", "sit", "amet", "", "consectetur", "adipiscing",
"elit", ""]
```

Удобный трюк (хотя он может привести к снижению производительности) — использовать пустой массив для фильтрации результатов после `map()`. В следующем примере расширяется приведенный выше пример для удаления пустых строк:

```
const arr = ['Lorem ipsum dolor sit amet,', 'consectetur adipiscing elit.'];

console.log(arr.flatMap((x) => x.split(/[\W+]/)).flatMap((x) => x || []));
// ["Lorem", "ipsum", "dolor", "sit", "amet", consectetur", "adipiscing", "elit"]
```

Здесь каждая пустая строка в результатах сначала отображается в пустой массив. При сглаживании они эффективно пропускаются в возвращаемом массиве.

> **ПРИМЕЧАНИЕ** Использование этой стратегии не рекомендуется, поскольку каждое фильтруемое значение требует создания нового экземпляра `Array`, который немедленно отбрасывается.

OBJECT. FROMENTRIES()

ECMAScript 2019 добавил статический метод `fromEntries()` в класс `Object`, который создает объект из набора пар массивов ключ—значение. Этот метод выполняет операцию, противоположную `Object.entries()`, и демонстрируется здесь:

```
const obj = {
    foo: 'bar',
    baz: 'qux'
};

const objEntries = Object.entries(obj);

console.log(objEntries);
// [["foo", "bar"], ["baz", "qux"]]

console.log(Object.fromEntries(objEntries));
// { foo: "bar", baz: "qux" }
```

Статический метод ожидает итерируемый объект, содержащий любое количество итерируемых объектов размером 2. Это особенно полезно в тех случаях, когда нужно преобразовать экземпляр `Map` в экземпляр `Object`, поскольку выходные данные итератора `Map` точно соответствуют сигнатуре, которую получает `fromEntries()`:

```
const map = new Map().set('foo', 'bar');

console.log(Object.fromEntries(map));
// { foo: "bar" }
```

МЕТОДЫ ОБРЕЗКИ СТРОК

ECMAScript 2019 добавил два метода к прототипу `String`, `trimStart()` и `trimEnd()`, которые позволяют выполнять целевое удаление пробелов. Эти методы предназначены для замены `trimLeft()` и `trimRight()`, которые имеют неоднозначное значение в контексте языков, читаемых справа налево, таких как арабский и иврит.

Эти два метода фактически противоположны `padStart()` и `padEnd()` с одним пробелом. Следующий пример добавляет пробел к строке, а затем удаляет ее с обеих сторон:

```
let s = ' foo ';

console.log(s.trimStart());    // "foo "
console.log(s.trimEnd());      // " foo"
```

SYMBOL.PROTOTYPE.DESCRIPTION

В ECMAScript 2019 добавлена возможность проверки необязательного описания `Symbol` через свойство `description`. Ранее это было доступно только тогда, когда символ был приведен к строке:

```
const s = Symbol('foo');

console.log(s.toString());
// Symbol(foo)
```

С выходом ES2019 каждый объект `Symbol` получил свойство `description` только для чтения, которое предоставляет описание. Если описания нет, по умолчанию используется значение `undefined`.

```
const s = Symbol('foo');

console.log(s.description);
// foo
```

НЕОБЯЗАТЕЛЬНАЯ ПРИВЯЗКА CATCH

До ES2019 структура блока `try/catch` была довольно жесткой. Даже если не нужно использовать перехваченный объект ошибки, парсер все равно потребует присвоения имени переменной объекту ошибки внутри условия `catch`:

```
try {
    throw 'foo';
} catch (e) {
    // Произойдет ошибка, но сам объект ошибки не имеет значения
}
```

В ES2019 можно опустить присвоение объекта ошибки и просто полностью игнорировать ошибку:

```
try {
     throw 'foo';
} catch {
    // Произойдет ошибка, но сам объект ошибки не имеет значения
}
```

ЕЩЕ НЕСКОЛЬКО УЛУЧШЕНИЙ

ES2019 также добавляет несколько настроек в существующий инструментарий:

- `Array.prototype.sort()` *стабилен*, что означает, что эквивалентные объекты не будут переупорядочены в выходных данных.
- Одиночные суррогатные символы UTF-16 не могут быть закодированы в UTF-8, что вызывает проблемы с `JSON.stringify()`. Вместо того чтобы возвращать непарные суррогатные кодовые точки в виде отдельных кодовых единиц UTF16, теперь они будут представлены экранированными последовательностями JSON.
- Ранее и U+2028 LINE SEPARATOR, и U+2029 PARAGRAPH SEPARATOR были действительны в строках JSON, но недопустимы в строках ECMAScript. ES2019 представляет совместимость между строками ECMAScript и JSON.
- Раньше у поставщиков браузеров было множество возможностей указать, что было возвращено из `Function.prototype.toString()`. ES2019 требует, чтобы этот метод возвращал исходный код функции всякий раз, когда это возможно, в противном случае значение `{ [native code] }`.

Б

Строгий режим

Строгий режим (strict mode) был представлен в ECMAScript 5. Он позволяет реализовать более строгую проверку ошибок глобально или локально (в одной функции). Преимущество строго режима в том, что он позволяет раньше получать уведомления об ошибках, и вы можете оперативно устранять их.

Важно понимать правила строгого режима, так как в будущих версиях ECMAScript он будет использоваться по умолчанию. Строгий режим поддерживается во всех основных браузерах.

ВКЛЮЧЕНИЕ СТРОГОГО РЕЖИМА

Включить строгий режим можно с помощью следующей *директивы* (pragma) — строки, которая не назначается никакой переменной:

```
"use strict";
```

Этот синтаксис допустим даже в ECMAScript 3. Если интерпретатор JavaScript не поддерживает строгий режим, эта директива просто игнорируется как строковый литерал, не назначенный никакой переменной.

Если эта директива применяется глобально, то есть вне функции, строгий режим включается для всего сценария. Это означает, что добавление директивы в сценарий, который объединяется с другими сценариями в одном файле, включает строгий режим для всего кода в файле.

Строгий режим можно также включить только внутри отдельной функции, например:

```
function doSomething() {
    "use strict";

    // другие операции
}
```

Если у вас нет полного контроля над всеми сценариями на странице, рекомендуется включать строгий режим только внутри конкретных функций, которые были протестированы в нем.

ПЕРЕМЕННЫЕ

Способ и время создания переменных в строгом и обычном режимах различаются. Так, в строгом режиме нельзя создать глобальную переменную случайно. Например, следующий код в нестрогом режиме создает глобальную переменную:

```
// Переменная не объявлена
// Нестрогий режим: создается глобальная переменная
// Строгий режим: генерируется ошибка ReferenceError

message = "Hello world!";
```

Хотя переменной `message` не предшествует ключевое слово `let` и она не определена как свойство глобального объекта, она все же автоматически создается как глобальная. В строгом режиме присваивание значения необъявленной переменной приводит при запуске кода к ошибке `ReferenceError`.

Кроме того, в строгом режиме нельзя вызвать для переменной оператор `delete`. В нестрогом режиме это возможно, при этом интерпретатор просто возвращает `false`. В строгом режиме попытка удалить переменную приводит к ошибке:

```
// Удаление переменной
// Нестрогий режим: ошибка игнорируется без каких-либо действий
// Строгий режим: генерируется ошибка ReferenceError

let color = "red";
delete color;
```

Строгий режим также налагает ограничения на имена переменных. В нем запрещено использовать переменные с именами `implements`, `interface`, `let`, `package`, `private`, `protected`, `public`, `static` и `yield`. Они теперь являются зарезервированными словами, которые, возможно, будут задействованы в будущих редакциях ECMAScript. Попытка использовать их в качестве имен переменных в строгом режиме приведет к синтаксической ошибке.

ОБЪЕКТЫ

В строгом режиме операции с объектами чаще возвращают ошибки, потому что в нестрогом режиме многие ошибки просто игнорируются. Благодаря этому строгий режим способствует раннему устранению ошибок.

Вот некоторые ситуации, когда при доступе к свойству объекта возникает ошибка:

- присваивание значения свойству, доступному только для чтения, приводит к ошибке `TypeError`;

- применение оператора `delete` к неконфигурируемому свойству приводит к ошибке `TypeError`;
- попытка добавить свойство к нерасширяемому объекту приводит к ошибке `TypeError`.

Другое ограничение объектов имеет место, когда вы объявляете их с помощью литералов. Если используется литерал объекта, имена свойств должны быть уникальными, например:

```
// Два свойства с одним именем
// Нестрогий режим: ошибки нет, приоритет отдается второму свойству
// Строгий режим: генерируется синтаксическая ошибка

let person = {
                name: "Nicholas",
                name: "Greg"
            };
```

Этот литерал объекта `person` содержит два свойства с именем `name`. В нестрогом режиме к объекту будет добавлено второе свойство, а в строгом возникнет синтаксическая ошибка.

> **ПРИМЕЧАНИЕ** Ограничение на дубликаты имен свойств было снято в ECMAScript 6. Ключи литеральных свойств дублирующихся объектов не выдают ошибку в строгом режиме.

ФУНКЦИИ

Прежде всего, строгий режим требует, чтобы именованные аргументы функций были уникальными, например:

```
// Повторяющиеся именованные аргументы
// Нестрогий режим: ошибки нет, действителен только второй аргумент
// Строгий режим: генерируется ошибка SyntaxError

function sum (num, num) {
    // какие-то действия
}
```

В нестрогом режиме это объявление функции не приводит к ошибке. Вы можете получить доступ ко второму аргументу `num` по имени, тогда как первый доступен только через объект `arguments`.

В строгом режиме также слегка меняется поведение объекта `arguments`. В нестрогом режиме изменения именованного аргумента отражаются в этом объекте, а в строгом — нет, например:

```
// Изменение значения именованного аргумента
// Нестрогий режим: изменение отражается на объекте arguments
```

```
// Строгий режим: изменение не отражается на объекте arguments

function showValue(value) {
    value = "Foo";
    alert(value);           // "Foo"
    alert(arguments[0]);    // Нестрогий режим: "Foo"
                            // Строгий режим: "Hi"
}

showValue("Hi");
```

Эта функция `showValue()` принимает единственный именованный аргумент `value`. Мы вызываем ее с аргументом `"Hi"`, который назначается переменной `value`, но внутри функции значение `value` изменяется на `"Foo"`. В нестрогом режиме при этом также изменяется значение `arguments[0]`, но в строгом режиме это разные сущности.

В строгом режиме также недоступны свойства `arguments.callee` и `arguments.caller`. В нестрогом режиме они представляют текущую функцию и вызвавшую ее функцию соответственно, а в строгом попытка доступа к любому из этих свойств приводит к ошибке `TypeError`, например:

```
// Попытка доступа к свойству arguments.callee
// Нестрогий режим: код выполняется обычным образом
// Строгий режим: генерируется ошибка TypeError

function factorial(num) {
    if (num <= 1) {
        return 1;
    } else {
        return num * arguments.callee(num-1)
    }
}

let result = factorial(5);
```

При попытке прочитать или записать свойство `caller` или `arguments` функции также генерируется ошибка `TypeError`. В приведенном примере эта ошибка возникла бы при доступе к свойствам `factorial.caller` и `factorial.callee`.

Как и в случае переменных, в строгом режиме нельзя назначать функциям имена `implements`, `interface`, `let`, `package`, `private`, `protected`, `public`, `static` и `yield`.

Наконец, в строгом режиме можно объявлять функции только на верхнем уровне сценария или функции. Это означает, например, что объявление функции в инструкции `if` является синтаксической ошибкой:

```
// Объявление функции в инструкции if
// Нестрогий режим: функция поднимается за пределы инструкции if
// Строгий режим: генерируется синтаксическая ошибка

if (true) {
    function doSomething() {
        // ...
    }
}
```

Этот синтаксис допустим во всех браузерах в нестрогом режиме, но приводит к синтаксической ошибке в строгом.

Параметры функций

ES6 представил оператор остатка, деструктурированные параметры и параметры по умолчанию — обширный набор новых возможностей для функций для организации, структурирования и определения их параметров. Небольшое изменение, введенное в ECMAScript 7, диктует, что функции, использующие любой из этих расширенных параметров, не могут использовать строгий режим в своем теле без выдачи ошибки. Использование глобального строгого режима все еще разрешено.

```
// допустимо
function foo(a, b, c) {
    "use strict";
}

// недопустимо
function bar(a, b, c='d') {
    "use strict";
}

// недопустимо
function baz({a, b, c}) {
    "use strict";
}

// недопустимо
function qux(a, b, ...c) {
    "use strict";
}
```

Новые функции, представленные в ES6, предполагают, что параметры будут анализироваться в том же режиме, что и тело функции. Поскольку прагма `"use strict"` встречается внутри тела функции, синтаксический анализатор JavaScript должен был бы проверить прагму внутри тела функции, прежде чем принимать параметры, что вносит много путаницы. Поэтому спецификация ES7 ввела это соглашение для того, чтобы синтаксический анализатор мог точно знать, в каком режиме он работает перед анализом функции.

Функция eval()

Печально известная функция `eval()` в строгом режиме стала работать иначе. Основное изменение состоит в том, что она больше не создает переменные или функции в контексте-контейнере, например:

```
// Использование функции eval() для создания переменной
// Нестрогий режим: в оповещении выводится число 10
// Строгий режим: при вызове alert(x) генерируется ошибка ReferenceError
```

```
function doSomething() {
    eval("let x=10");
    alert(x);
}
```

В нестрогом режиме этот код создает локальную переменную `x` в функции `doSomething()`, а затем выводит ее значение в оповещении. В строгом режиме вызов `eval()` не создает переменную `x` внутри функции `doSomething()`, и при вызове `alert()` возникает ошибка `ReferenceError`, потому что переменная `x` не объявлена.

Переменные и функции можно объявлять в функции `eval()`, но они остаются ограниченными специальной областью видимости, которая доступна в течение выполнения кода, а по его завершении уничтожается. Например, следующий код работает без ошибок:

```
"use strict";
let result = eval("let x=10, y=11; x+y");
alert(result);    // 21
```

Здесь мы объявляем переменные `x` и `y` внутри функции `eval()`, а затем вычисляем их сумму. Переменная `result` содержит результат сложения, хотя сами переменные `x` и `y` в момент вызова `alert()` больше не существуют.

ИДЕНТИФИКАТОРЫ EVAL И ARGUMENTS

В строгом режиме теперь явно запрещено использовать имена `eval` и `arguments` в качестве идентификаторов и изменять значения этих переменных:

```
// Переопределение eval и arguments
// Нестрогий режим: все в порядке
// Строгий режим: синтаксическая ошибка

let eval = 10;
let arguments = "Hello world!";
```

В нестрогом режиме вы можете перезаписать значения `eval` и `arguments`, но в строгом это приводит к синтаксической ошибке. Ошибка возникнет во всех перечисленных далее случаях использования этих имен:

- объявление переменной с помощью ключевого слова `let`;
- присваивание переменной другого значения;
- попытка изменить значение переменной, например с помощью оператора `++`;
- использование в качестве имени функции;
- использование в качестве именованного аргумента функции;
- использование в качестве имени исключения в инструкции `try-catch`.

ПРЕОБРАЗОВАНИЕ ЗНАЧЕНИЯ THIS

Одной из серьезнейших проблем с безопасностью и одним из самых непонятных аспектов применения JS-кода является преобразование значения `this` в ряде ситуаций. При вызове метода `apply()` или `call()` для функции значение `null` или `undefined` приводится в нестрогом режиме к глобальному объекту. В строгом режиме значение `this` функции всегда используется так, как указано, например:

```
// Доступ к свойству
// Нестрогий режим: доступ к глобальному свойству
// Строгий режим: ошибка, потому что this имеет значение null

let color = "red";

function displayColor() {
    alert(this.color);
}

displayColor.call(null);
```

Этот код передает в метод `displayColor.call()` значение `null`, так что в нестрогом режиме значением `this` функции является глобальный объект, в результате выводится оповещение со строкой `"red"`. В строгом режиме значение `this` функции равно `null`, и при доступе к свойству объекта `null` генерируется ошибка.

Как правило, функции преобразуют свое значение `this` в тип объекта, что в разговорной речи называется «упаковкой». Примитивы будут принудительно использоваться в своих эквивалентах оболочек объектов.

```
function foo() {
    console.log(this);
}

foo.call();     // Window {}
foo.call(2);    // Number {2}
```

При выполнении в строгом режиме значение `this` больше не будет упаковываться:

```
function foo() {
    "use strict";
    console.log(this);
}

foo.call();     // undefined
foo.call(2);    // 2
```

КЛАССЫ И МОДУЛИ

Классы и модули являются двумя носителями контейнеров кода, введенных в ECMAScript 6. Для любого из них не существует концепции предшественника ECMAScript, и поэтому нет необходимости поддерживать синтаксическую совместимость с унаследованными версиями ECMAScript. Поэтому TC-39 решил, что

весь код, определенный внутри классов и модулей ES6, по умолчанию находится в строгом режиме.

Для классов это включает как объявления классов, так и выражения; конструктор, методы экземпляра, статические методы, методы получения и установки находятся в строгом режиме. Для модулей весь код, определенный внутри них, будет находиться в строгом режиме.

ДРУГИЕ ИЗМЕНЕНИЯ

Следует также упомянуть несколько других отличий строгого режима. Так, в нем недоступна инструкция `with`. Она изменяет способ разрешения идентификаторов и была удалена из строгого режима ради упрощения языка. Попытка использовать `with` в строгом режиме приведет к синтаксической ошибке.

```
// Использование инструкции with
// Нестрогий режим: все в порядке
// Строгий режим: синтаксическая ошибка

with(location) {
    alert(href);
}
```

Также в строгом режиме недоступны *восьмеричные литералы*. Они начинаются с нуля и традиционно были причиной многих ошибок. Теперь использование восьмеричного литерала в строгом режиме считается синтаксической ошибкой.

```
// Использование восьмеричного литерала
// Нестрогий режим: value имеет значение 8
// Строгий режим: синтаксическая ошибка

let value = 010;
```

Как уже говорилось, в ECMAScript 5 метод `parseInt()` был изменен; в нестрогом режиме он обрабатывает восьмеричные литералы так же, как десятичные литералы с начальным нулем, например:

```
// Использование восьмеричного литерала в функции parseInt()
// Нестрогий режим: value имеет значение 8
// Строгий режим: value имеет значение 10

let value = parseInt("010");
```

B

JavaScript-библиотеки и фреймворки

Библиотеки (libraries) в JavaScript помогают компенсировать различия браузеров и упрощают доступ к их сложным функциям. Различают *библиотеки общего назначения* (general libraries) и *специальные библиотеки* (specialty libraries). JavaScript-библиотеки общего назначения предоставляют доступ к наиболее востребованному функционалу браузеров и могут использоваться для реализации базовых возможностей веб-сайтов и веб-приложений. Специальные библиотеки предназначены для решения специфичных задач и используются только в отдельных частях веб-сайтов и веб-приложений. В этом приложении представлен обзор нескольких библиотек и даны ссылки на дополнительные ресурсы.

ФРЕЙМВОРКИ

Обозначение «фреймворк» охватывает спектр различных шаблонов, но в некоторой форме все они обеспечивают продуманную организационную структуру, в рамках которой могут формироваться сложные приложения. Использование фреймворка позволяет приложениям поддерживать согласованные условные обозначения кода при элегантном масштабировании по размеру и сложности. Они предлагают надежные механизмы для общих задач, таких как определение и повторное использование компонентов, управление потоком данных, маршрутизация и многие другие.

Все чаще JavaScript-фреймворки принимают форму *одностраничного приложения* (single page application, SPA). Одностраничные приложения используют API истории браузера HTML5 для предоставления всего пользовательского интерфейса приложения с маршрутизацией URL-адресов и только одной начальной

загрузкой страницы. Фреймворк управляет состоянием приложения, а также всеми компонентами пользовательского интерфейса во время выполнения приложения. У большинства популярных платформ SPA есть сильные сообщества разработчиков, а также множество сторонних расширений.

React

Созданный в Facebook фреймворк *React* охватывает компонент «представление» в модели Model-View-Controller. Его ограниченная сфера применения означает, что его можно использовать вместе с другими платформами или расширениями React для достижения полного охвата MVC. React использует однонаправленный поток данных, является декларативным и основанным на компонентах, использует виртуальную DOM для эффективной перерисовки страницы и предлагает синтаксис JSX, который позволяет писать разметку внутри JavaScript. Facebook также поддерживает дополнительную структуру React, которая называется «Flux».

Лицензия: MIT License.

Веб-сайт: https://reactjs.org/

Angular

Проект *Angular*, впервые выпущенный компанией Google в 2010 году, представляет собой полнофункциональный фреймворк для веб-приложений с архитектурой Model-View-Viewmodel. В 2016 году проект разделился на две ветви: Angular 1.x, который является продолжением первоначального проекта AngularJS, и Angular 2, который является полностью переработанным фреймворком, построенным на конструкциях ES6 и TypeScript. Последние версии обеих выпусков — директивные и основанные на компонентах реализации, и оба проекта используются надежными сообществами разработчиков и сторонними надстройками.

Лицензия: MIT License.

Веб-сайт: https://angularjs.org/ и https://angular.io/

VUE

Vue — это полнофункциональный фреймворк для веб-приложений, менее своеобразный, чем такие фреймворки, как Angular. Со времени выпуска в 2014 году сообщество его разработчиков сильно выросло, и многие разработчики перешли на Vue из-за его производительности и организационных выгод и в то же время менее — из-за строгого соблюдения принципов.

Лицензия: MIT License.

Веб-сайт: https://vuejs.org/

Ember

Ember похож на Angular тем, что также является архитектурой Model-View-Viewmodel и использует предпочтительные соглашения для заполнения веб-приложения. В версии 2.0 в 2015 году было представлено много поведенческих характеристик, используемых в фреймворке React.

Лицензия: MIT License.

Веб-сайт: https://emberjs.com/

Meteor

Meteor совершенно не похож на другие фреймворки в этом списке, потому что это изоморфный фреймворк JavaScript, то есть клиент и сервер имеют общую кодовую базу. Он также использует протокол обновления данных в режиме реального времени, который постоянно передает свежие изменения данных из БД клиенту. Meteor — чрезвычайно самоуверенный фреймворк; однако преимущество этого заключается в том, что функции приложения можно быстро разрабатывать с помощью надежного готового набора инструментов.

Лицензия: MIT License.

Веб-сайт: http://meteor.com/

Backbone.js

Минимальная библиотека с открытым исходным кодом Model-View-Controller (MVC), созданная поверх Underscore.js, *Backbone.js* оптимизирована для одностраничных приложений, что позволяет легко обновлять части страницы по мере изменения состояния приложения.

Лицензия: MIT License.

Веб-сайт: http://backbonejs.org/

БИБЛИОТЕКИ ОБЩЕГО НАЗНАЧЕНИЯ

JavaScript-библиотеки общего назначения предоставляют функционал, охватывающий несколько областей. Все библиотеки общего назначения уравнивают различия в интерфейсе и реализациях браузеров, заключая общий функционал в новые API. Некоторые из этих API напоминают встроенный функционал, тогда как другие выглядят совершенно иначе. Библиотеки общего назначения обычно поддерживают взаимодействие с DOM, технологию Ajax и вспомогательные методы, помогающие решать типичные задачи.

jQuery

jQuery — это библиотека с открытым исходным кодом, которая предоставляет функциональный интерфейс программирования для JavaScript. В основе jQuery лежит использование CSS-селекторов для работы с DOM-элементами. Благодаря цепочке вызовов jQuery-код больше похож не на JS-сценарий, а на описание того, что должно произойти. Такой стиль написания кода особо популярен среди дизайнеров и разработчиков прототипов.

Лицензия: MIT License или General Public License (GPL).

Веб-сайт: http://jquery.com/

Библиотека Google Closure

Библиотека Google Closure — это универсальный инструментарий JavaScript, во многом похожий на jQuery. Он состоит из чрезвычайно широкого набора модулей, охватывающих как низкоуровневые операции, так и высокоуровневые компоненты и виджеты. Библиотека Closure разработана так, что модули могут быть включены по мере необходимости, а библиотека создана для работы вместе с компилятором Google Closure (рассматривается в Приложении Г «JavaScript-инструменты»).

Лицензия: Apache 2.0.

Веб-сайт: https://developers.google.com/closure/library/

Underscore.js

Underscore.js, хотя и не является общей библиотекой в строгом смысле, предоставляет некоторые дополнительные функциональные возможности для функционального программирования на JavaScript. В документации говорится о Underscore.js как о дополнении к jQuery, предоставляющем дополнительные низкоуровневые функциональные возможности для работы с объектами, массивами, функциями и другими типами данных JavaScript.

Лицензия: MIT License.

Веб-сайт: http://documentcloud.github.com/underscore/

Lodash

В той же категории, что и Underscore.js, *Lodash* также является утилитарной библиотекой, которая предоставляет дополнительный инструментарий JavaScript. В нем представлены улучшенные методы для собственных типов, таких как массивы, объекты, функции и примитивы.

Лицензия: MIT License.

Веб-сайт: https://lodash.com/

Prototype

Prototype — это библиотека с открытым исходным кодом, предоставляющая простые API для решения типичных задач веб-программирования. Первоначально разработанная для Ruby on Rails, она содержит определения полезных классов и обеспечивает возможность наследования в JavaScript. Для этого в Prototype реализованы классы, инкапсулирующие часто используемый и сложный функционал в простые API-вызовы. Библиотека Prototype содержится в одном файле, который можно легко подключить к любой странице.

Лицензия: MIT License и Creative Commons Attribution-Share Alike 3.0 Unported.

Веб-сайт: www.prototypejs.org/

Dojo Toolkit

В *Dojo Toolkit*, библиотеке с открытым исходным кодом, группы функциональных возможностей организованы в пакеты, которые можно загружать по требованию. Dojo поддерживает множество параметров и конфигураций, охватывая почти все задачи, которые приходится решать средствами JavaScript.

Лицензия: «новая» BSD License или Academic Free License 2.1.

Веб-сайт: www.dojotoolkit.org/

MooTools

MooTools — это компактная оптимизированная библиотека с открытым исходным кодом, которая добавляет методы к встроенным JavaScript-объектам для расширения их функционала и предоставляет ряд новых объектов. К ее достоинствам можно отнести небольшой размер и простой API.

Лицензия: MIT License.

Веб-сайт: www.mootools.net/

qooxdoo

qooxdoo — это библиотека с открытым исходным кодом, цель которой — помочь во всем цикле разработки веб-приложений. qooxdoo реализует свои собственные версии классов и интерфейсов для создания модели программирования, аналогичной традиционным объектно-ориентированным (OO) языкам. Библиотека включает полный набор инструментов GUI и компиляторы для упрощения процесса сборки интерфейса. qooxdoo начинался как внутренняя библиотека для веб-хостинговой компании 1&1 (www.1and1.com), а затем был выпущен под лицензией с открытым исходным кодом.

Лицензия: GNU Lesser General Public License (LGPL) or Eclipse Public License (EPL).

Веб-сайт: www.qooxdoo.org/

БИБЛИОТЕКИ ДЛЯ АНИМАЦИИ И ЭФФЕКТОВ

Анимация и другие визуальные эффекты стали важными составляющими веб-контента. Реализовать плавные анимации на веб-страницах непросто, но, к счастью, некоторые разработчики уже позаботились об этом, создав специальные библиотеки. Многие из упомянутых ранее JavaScript-библиотек общего назначения также поддерживают анимации.

D3

Самая популярная библиотека анимации, *D3* (для «документов, управляемых данными»), сегодня является наиболее надежным и мощным инструментом визуализации данных JavaScript. Он имеет очень обширный набор функций, охватывающий холст, SVG, CSS и HTML5 визуализации. Библиотека дает возможность контролировать окончательную отрисовку с предельной точностью.

Лицензия: BSD.

Веб-сайт: http://www.d3js.org/

three.js

three.js — одна из самых популярных доступных библиотек WebGL. Она предлагает простой API, который позволяет выполнять сложные 3D-отрисовки и анимации.

Лицензия: MIT License.

Веб-сайт: https://threejs.org/

moo.fx

Библиотека *moo.fx* с открытым исходным кодом работает поверх Prototype или MooTools. Она очень компактна (последняя версия занимает всего 3 Кбайт) и предназначена для создания анимаций при минимуме кодирования. По умолчанию moo.fx входит в MooTools, но ее можно также загрузить отдельно для использования с Prototype.

Лицензия: MIT License.

Веб-сайт: http://moofx.mad4milk.net/

Lightbox

Библиотека *Lightbox* предназначена для создания простых графических наложений на страницах. Для работы ей требуются библиотеки Prototype и script.aculo.us. Идея в том, чтобы пользователи могли просматривать изображения или последовательности изображений, не покидая текущую страницу. Для Lightbox-наложений можно настраивать вид и переходы.

Лицензия: Creative Commons Attribution 2.5 License.

Веб-сайт: www.huddletogether.com/projects/lightbox2/

Г

JavaScript-инструменты

Написание JS-кода во многом напоминает программирование на любом другом языке, при этом, как и в других языках, вы можете использовать вспомогательные инструменты. Количество таких инструментов для JavaScript постоянно растет, благодаря чему искать проблемы, оптимизировать и развертывать приложения становится проще. Некоторые из этих инструментов запускаются из JS-кода, другие можно запускать извне браузера. В этом приложении представлен обзор нескольких полезных инструментов и ссылки на ресурсы с дополнительными сведениями.

ПРИМЕЧАНИЕ Вы заметите, что некоторые инструменты включены в несколько разделов этого приложения. Многие инструменты JavaScript, доступные сегодня, должны быть инструментами управления проектами по принципу «все в одном» и, таким образом, они относятся к нескольким сферам.

СРЕДСТВА УПРАВЛЕНИЯ ПАКЕТАМИ

Проекты JavaScript обычно должны использовать сторонние библиотеки и ресурсы, чтобы избежать дублирования кода и ускорения разработки. Сторонние библиотеки, называемые «пакетами», размещаются в общедоступных репозиториях. Пакеты могут принимать форму ресурсов, которые будут доставляться в браузер, библиотек JavaScript, которые будут скомпилированы в рамках проекта, или даже инструментов для конвейера разработки проекта. Эти пакеты почти всегда активно разрабатываются и пересматриваются, в дополнение к различным версиям выпусков. *Средства управления пакетами* в JavaScript позволяют управлять тем, от каких пакетов зависят проекты, как обращаться к ним и устанавливать их, а также какие версии устанавливать.

Средства управления пакетами предлагают интерфейс командной строки для установки или удаления зависимостей проекта. Конфигурация проекта обычно хранится в локальном файле манифеста проекта.

npm

npm, который расшифровывается как «диспетчер пакетов Node (Node Package Manager)», является менеджером пакетов по умолчанию для среды выполнения NodeJS. Сторонние пакеты, предоставляемые в реестре npm, могут быть указаны как зависимости проекта и установлены локально через командную строку. Репозиторий npm содержит как серверные, так и клиентские пакеты JavaScript.

npm предназначен для использования на сервере, где размер графа зависимостей менее критичен. При установке пакетов npm использует вложенное дерево зависимостей для разрешения всех зависимостей проекта; каждая зависимость проекта будет устанавливать свою собственную версию любых других пакетов, от которых она зависит. Это означает, что если ваш проект имеет три пакетных зависимости — A, B и C — и каждая из них зависит от отдельной версии пакета D, npm установит три отдельные версии пакета D, по одной для каждой зависимости.

Веб-сайт: https://www.npmjs.com/

Bower

Bower поддерживает многие из тех же ритмов, что и npm, включая CLI для установки и управления пакетами, но основное внимание уделяется управлению пакетами, которые будут использоваться для клиента. Одно из основных различий между Bower и npm состоит в том, что Bower использует плоскую структуру зависимостей. Это означает, что зависимости проекта будут совместно использовать пакеты, от которых они зависят, и задача пользователя состоит в том, чтобы разрешить эти зависимости. Например, если ваш проект имеет три зависимости пакета — A, B и C — и каждая из них зависит от другой версии пакета D, нужно будет найти одну версию пакета D, которая удовлетворяет требованиям всех этих пакетов, потому что плоская структура зависимостей диктует, что вы устанавливаете одну любую версию данного пакета.

Веб-сайт: https://bower.io/

JSPM

JSPM — это средство управления пакетами, построенное на использовании SystemJS для динамической загрузки модулей. Сам он похож на npm, но не зависит от реестра; пакеты из реестра npm, Github или даже пользовательских реестров могут быть установлены вместе с помощью его CLI. Вместо того чтобы связывать и предварительно компилировать ресурсы на сервере, JSPM позволяет динамически предоставлять пакеты клиенту по мере необходимости через SystemJS. Как и Bower, он использует плоскую структуру зависимостей.

Веб-сайт: https://jspm.io/

Yarn

Разработанное в Facebook, *Yarn* — это специальное средство управления пакетами, которое во многом является усовершенствованной версией npm. Он может получить доступ ко всем тем же пакетам npm через реестр Yarn, и он устанавливает их так же, как и npm. Основное различие между Yarn и npm заключается в том, что он предлагает функции, позволяющие ускорить установку, кэширование пакетов, файлы блокировки и улучшенные функции безопасности пакетов.

Веб-сайт: https://yarnpkg.com/

СРЕДСТВА ЗАГРУЗКИ МОДУЛЕЙ

Средства загрузки модулей позволяют запрашивать модули с сервера по требованию, а не загружать все модули JS или один связанный файл JS одновременно. Спецификация модуля ECMAScript 6 описывает возможную цель для браузеров, чтобы изначально поддерживать динамическую загрузку модулей. В настоящее время многие браузеры по-прежнему не поддерживают загрузку модулей ES6, и поэтому средства загрузки модулей служат своего рода заменой, которая позволяет пакетам динамически загружать модули из клиента.

SystemJS

Средство загрузки модулей *SystemJS* предназначено для использования на сервере или клиенте. SystemJS поддерживает все форматы модулей, включая AMD, CommonJS, UMD и ES6. Он также поддерживает транспиляцию в браузере (не рекомендуется для крупных проектов из-за проблем с производительностью).

Веб-сайт: https://github.com/systemjs/

RequireJS

RequireJS построен на основе спецификации модулей AMD и предлагает исключительную поддержку устаревших браузеров. Хотя RequireJS проверен временем, сообщество JavaScript в целом в значительной степени отбрасывает формат модулей AMD, поэтому начинать крупный проект с RequireJS не рекомендуется.

Веб-сайт: http://requirejs.org/

СБОРЩИКИ МОДУЛЕЙ

Сборщики модулей позволяют объединять произвольное количество модулей в различных форматах в пакеты, которые можно загрузить на клиент. Сборщик будет отслеживать граф зависимостей приложения и упорядочивать модули по мере необходимости. Часто приложение может обслуживаться как один пакет, но также возможны настройки для нескольких пакетов. Сборщик модулей также

часто поддерживает объединение необработанных или скомпилированных ресурсов CSS. Сборщики могут принимать форму самореализующейся связки или могут оставаться в виде объединенных активов модуля, которые не будут выполняться до тех пор, пока это не потребуется.

Webpack

Обладая широким набором функций и превосходной расширяемостью, *Webpack* в подавляющем большинстве является самым популярным на сегодняшний день сборщиком приложений. Он может объединять различные типы модулей, поддерживает широкий спектр плагинов и полностью совместим с большинством библиотек шаблонов и транспиляции.

Веб-сайт: https://webpack.js.org/

JSPM

JSPM — это средство управления пакетами, созданное на основе SystemJS и загрузчика модулей ES6. Один из рекомендованных рабочих процессов JSPM состоит в объединении всех модулей в один файл, который затем будет загружен с помощью SystemJS. Эта возможность доступна через JSPM CLI.

Веб-сайт: https://jspm.io/

Browserify

Browserify — это несколько более старый, но широко используемый и проверенный в бою пакет модулей, который поддерживает синтаксис зависимостей `require()` CommonJS в стиле NodeJS.

Веб-сайт: http://browserify.org/

Rollup

Rollup во многих отношениях похож на Browserify по своим возможностям связывания модулей, но также предоставляет встроеную возможность встряхивания дерева. Rollup может анализировать граф зависимостей приложения и удалять любые модули, которые фактически не используются.

Веб-сайт: https://rollupjs.org/

СРЕДСТВА КОМПИЛЯЦИИ/ТРАНСПИЛЯЦИИ И СИСТЕМЫ СТАТИЧЕСКОГО ТИПА

Код веб-приложения, написанный в редакторе, почти никогда не является точной копией кода, который будет передан клиенту. Разработчики часто хотят использовать новые функции спецификации ECMAScript, которые еще не получили

универсального одобрения браузером. Кроме того, разработчики также часто хотят расширить или усовершенствовать свою кодовую базу с помощью статической системы типов или функций, выходящих за пределы спецификации ECMA. Существует целый ряд инструментов для решения различных аспектов этих потребностей.

Babel

Babel — один из самых популярных инструментов, используемых для компиляции новейших функций спецификации ECMAScript до версии ECMA, удобной для браузера. Он также поддерживает JSX от React и широкий спектр плагинов и совместим со всеми основными инструментами сборки.

Веб-сайт: https://babeljs.io/

Google Closure Compiler

Google Closure Compiler — это мощный JS-компилятор, способный варьировать уровни оптимизации компиляции, а также надежная система проверки статического типа. Аннотации типа выполняются в комментариях в стиле JSDoc.

Веб-сайт: https://developers.google.com/closure/

CoffeeScript

CoffeeScript — это расширенный синтаксис ECMAScript, который компилируется в обычный JavaScript; почти все в CoffeeScript является выражением. Его улучшения синтаксиса вдохновлены Ruby, Python и Haskell.

Сайт: http://coffeescript.org/

TypeScript

TypeScript от Microsoft — это типизированный расширенный набор JavaScript, который включает в себя надежную проверку статических типов и основные улучшения синтаксиса. Поскольку это строгий расширенный набор JavaScript, обычные JS-программы имеют допустимый синтаксис TypeScript. TypeScript также может использовать файлы определения типа для указания информации о типе для существующих библиотек JavaScript.

Веб-сайт: http://typescriptlang.org/

Flow

Flow от Facebook — это система аннотаций простого типа для JavaScript. Его синтаксис типа очень похож на TypeScript, но он не добавляет дополнительных возможностей языка, кроме объявлений типов.

Веб-сайт: https://flow.org/

ВЫСОКОЭФФЕКТИВНЫЕ ИНСТРУМЕНТЫ СЦЕНАРИЕВ

Распространенная критика JavaScript заключается в том, что он невероятно медленный и неуместен для вычислений, требующих гибкости. Независимо от того, сколько опасности таится в обозначении «медленный», это не меняет того факта, что язык никогда не создавался для поддержки гибких вычислений. Для решения этой проблемы существует ряд проектов, которые пытаются расширить парадигмы выполнения кода в браузере, чтобы позволить программе работать с почти родной скоростью и использовать аппаратную оптимизацию.

WebAssembly

Проект *WebAssembly* (или *wasm*) работает над реализацией языка, который может выполняться во многих местах (переносимый) и который существует как двоичный язык, компилируемый из нескольких языков низкого уровня, таких как C++ и Rust. Код WebAssembly работает на совершенно отдельной виртуальной машине и на JavaScript в браузере, и его способность взаимодействовать с различными API браузера крайне ограничена. С JavaScript и DOM можно взаимодействовать косвенным и ограниченным образом, но главной целью WebAssembly является создание чрезвычайно быстрого языка, который может работать в веб-браузерах (и в других местах) и предлагать практически собственную производительность и аппаратное ускорение. Спецификация WebAssembly все еще дорабатывается и уточняется, но это одна из самых многообещающих областей в технологии браузеров.

Веб-сайт: http://webassembly.org/

asm.js

asm.js основан на идее, что скомпилированный JavaScript может выполняться намного быстрее, чем рукописный JavaScript. asm.js — это подмножество JavaScript, с которым может быть скомпилирован и выполнен код языка низкого уровня в обычном браузере или движке Node. Современные механизмы JavaScript выводят типы во время выполнения, а код asm.js делает эти выводы типов (и, следовательно, связанные с ними операции) гораздо менее затратными в вычислительном отношении, выполняя принудительное приведение типов с помощью лексических подсказок. Он также широко использует TypedArrays, которые предлагают огромный скачок производительности по сравнению с традиционным ассоциативным массивом JS. asm.js не так быстр, как WebAssembly, но все же обеспечивает значительный прирост производительности за счет компиляции.

Веб-сайт: http://asmjs.org/

Emscripten и LLVM

Хотя он никогда не выполняется в браузере, *Emscripten* является важным набором инструментов для компиляции низкоуровневого кода в WebAssembly и asm.js.

Emscripten использует компилятор *LLVM* для компиляции таких языков, как C, C ++ и Rust в код, который может выполняться либо непосредственно в браузере (asm.js), либо на виртуальной машине (WebAssembly).

Веб-сайт Emscripten: http://kripken.github.io/emscripten-site/

Веб-сайт LLVM: http://llvm.org/

РЕДАКТОРЫ

VIM, Emacs и им подобные являются отличными текстовыми редакторами, но по мере того, как ваша среда сборки и приложения масштабируются по сложности, становится все более полезным автоматизация типичных задач в редакторах, таких как автозаполнение, форматирование файлов, кодирование и организация каталогов проекта. Доступен широкий выбор *редакторов* и IDE, предлагающих эти функции, как бесплатных, так и платных.

Sublime Text

Sublime Text — популярный текстовый редактор с закрытым исходным кодом. Его можно использовать для разработки на всех языках, но он предлагает чрезвычайно обширную библиотеку дополнений, поддерживаемую сообществом. Производительность редактора является одной из лучших в своем классе.

Веб-сайт: https://sublimetext.com/

Стоимость: бесплатная пробная версия; одноразовая лицензия за 80 долларов.

Atom

Atom от Github — это текстовый редактор с открытым исходным кодом, обладающий многими теми же функциями, что и Sublime Text, включая процветающее сообщество и сторонние дополнения к пакетам. Редактор иногда имеет проблемы с производительностью, но он постоянно делает успехи в этой области.

Сайт: https://atom.io/

Стоимость: бесплатно.

Brackets

Brackets от Adobe похож на Atom в том, что они оба распространяются с открытым исходным кодом, но редактор разработан специально для веб-разработчиков и предлагает ряд очень впечатляющих и уникальных функций, предназначенных для внешнего интерфейса. Сопровождает все это обширная библиотека дополнений.

Сайт: http://brackets.io/

Стоимость: бесплатно.

Visual Studio Code

Visual Studio Code от Microsoft — это редактор с открытым исходным кодом, основанный на платформе Electron. Подобно другим основным редакторам, он легко расширяется благодаря своей библиотеке пакетов.

Веб-сайт: https://jetbrains.com/webstorm/

Стоимость: бесплатно.

WebStorm

WebStorm от JetBrains — это неприступно высокооктановая среда разработки, предназначенная для использования в качестве окончательного инструментария для разработки проектов, с надежной интеграцией с ведущими средами внешнего интерфейса. Он также интегрируется с большинством инструментов сборки и систем контроля версий.

Веб-сайт: https://jetbrains.com/webstorm/

Стоимость: бесплатная пробная версия; ежемесячные/ежегодные лицензионные сборы.

СИСТЕМЫ СБОРКИ И АВТОМАТИЗАЦИИ, СРЕДСТВА ЗАПУСКА ЗАДАЧ

Процесс преобразования локального каталога проекта в приложение, обслуживаемое в конечной версии продукта, обычно объединяется в ряд задач. Каждая из этих задач часто состоит из конвейера из множества подзадач — например, создание и развертывание приложения будет включать в себя объединение модулей, компиляцию, минимизацию и передачу статических ресурсов, среди многих других задач. Выполнение модульного или интеграционного тестирования может включать в себя инициализацию тестовых наборов и ускорение работы браузеров. Чтобы упростить управление этими задачами и их использование, существует ряд инструментов, позволяющих более эффективно составлять и организовывать задачи приложения.

Grunt

Grunt — это средство запуска задач NodeJS, которое использует объекты конфигурации для декларативного определения того, как задачи должны выполняться. Он имеет обширное сообщество и большой выбор дополнений, с помощью которых можно организовать работу над задачами проекта.

Веб-сайт: https://gruntjs.com/

Gulp

Подобно Grunt, *Gulp* также выполняет задачи NodeJS, но вместо этого использует конвейерный подход в стиле Unix для определения задач, где отдельные задачи определяются как функции JavaScript. Gulp также имеет динамичное сообщество и библиотеку пакетов.

Веб-сайт: https://gulpjs.com/

Brunch

Brunch — это еще один инструмент сборки NodeJS, который разработан для упрощения и удобства использования, а не для максимальной конфигурируемости. Он новее, чем Gulp или Grunt, но уже имеет огромный выбор дополнений.

Сайт: http://brunch.io/

npm

Несмотря на то что *npm* не является инструментом сборки, он предлагает функцию сценариев, которая, по мнению многих разработчиков, прекрасно работает как средство запуска задач без дополнительных излишеств. Сценарии определяются непосредственно внутри пакета NodeJS.

Веб-сайт: https://docs.npmjs.com/misc/scripts/

СРЕДСТВА АНАЛИЗА И ФОРМАТИРОВАНИЯ КОДА

Отчасти проблема с отладкой JavaScript заключается в том, что многие IDE не указывают автоматически на синтаксические ошибки при вводе. Большинство разработчиков пишут какой-либо код, а затем загружают его в браузер для поиска ошибок. Можно значительно уменьшить количество таких ошибок, проверяя код JavaScript перед развертыванием. *Средства анализа* проверяют основной синтаксис и выдают предупреждения о стиле.

Средства форматирования — это инструменты, которые неявно понимают синтаксические правила языка и используют набор отступов, пробелов, переносов строк и других стратегий, чтобы предоставить возможность автоматически аккуратно организовать содержимое файла. Средства форматирования никогда не нарушат или не изменят код или его семантическое значение, поскольку они осведомлены о видах изменений, которые могут повлиять на выполнение.

ESLint

ESLint — это JavaScript-утилита с открытым исходным кодом, изначально разработанная не кем иным, как автором предыдущих изданий этой книги Николасом Закасом. Он спроектирован так, чтобы быть полностью «подключаемым»: он поставляется из коробки с общим набором правил по умолчанию, но правила полностью

настраиваемы, и имеется большая библиотека изменяемых и переключаемых правил, которые можно использовать для настройки поведения анализатора.

Веб-сайт: www.eslint.org/

Google Closure Compiler

В *Closure Compiler* встроена служебная программа, которая может быть активирована с помощью флагов командной строки. Этот анализатор работает с абстрактным синтаксическим деревом кода, поэтому он не выполняет проверки пробелов, отступов или других вопросов организации кода, которые не влияют на выполнение.

Веб-сайт: https://developers.google.com/closure/

JSLint

JSLint — это средство проверки JavaScript, написанное Дугласом Крокфордом. Он проверяет синтаксические ошибки на уровне ядра, используя наименьший общий знаменатель для кросс-браузерных проблем. (Следуйте самым строгим правилам, чтобы ваш код работал везде.) Вы можете включить предупреждения Крокфорда о стиле написания кода, включая формат, использование необъявленных глобальных переменных и многое другое. Хотя JSLint написан на JavaScript, его можно запустить из командной строки через интерпретатор Rhino на основе Java, а также через WScript и другие интерпретаторы JavaScript. Веб-сайт предоставляет собственные версии для каждого интерпретатора командной строки.

Веб-сайт: www.jslint.com/

JSHint

JSHint — это ответвление JSLint, который предоставляет больше возможностей для настройки применяемых правил. Подобно JSLint, он сначала проверяет синтаксические ошибки, а затем ищет проблемные паттерны кодирования. Каждая проверка JSLint также присутствует в JSHint, но разработчики лучше контролируют, какие правила нужно применять. Также как и JSLint, JSHint можно запустить из командной строки с помощью Rhino.

Веб-сайт: www.jshint.com/

Clang Format

ClangFormat — это набор инструментов форматирования, созданный на основе библиотеки LibFormat на основе проекта Clang. Он использует правила форматирования Clang для автоматической пространственной реорганизации кода (без фактического изменения его семантической структуры). Он работает как самостоятельный инструмент, который можно использовать из командной строки, но также имеет несколько интеграций с редакторами.

Веб-сайт: https://clang.llvm.org/docs/ClangFormat.html

СРЕДСТВА УМЕНЬШЕНИЯ КОЛИЧЕСТВА КОДА

Важной частью процесса сборки JavaScript является обработка вывода для удаления лишних символов. Это гарантирует, что только минимальное количество байтов передается в браузер для анализа и, в конечном итоге, ускоряет работу пользователя. Существует несколько таких *средств уменьшения количества кода* (*минимизаторов*) с различными степенями сжатия.

Uglify

Uglify, в настоящее время его третий выпуск, представляет собой набор инструментов, который позволяет минимизировать, украсить и сжать JS-код. Его можно запустить из командной строки, и он предлагает чрезвычайно широкий диапазон параметров сжатия, которые можно настроить для настройки минимизации.

Веб-сайт: https://github.com/mishoo/UglifyJS2/

Google Closure Compiler

Несмотря на то что *Closure* не является минимизатором в строгом смысле, он предлагает несколько уровней оптимизации, которые в процессе проведения процедур оптимизации в том числе минимизируют код.

Веб-сайт: https://developers.google.com/closure/

JSMin

JSMin — это основанный на C инструмент, написанный Дугласом Крокфордом, который выполняет базовое сжатие JavaScript. В первую очередь он удаляет пробелы и комментарии, чтобы гарантировать, что полученный код все еще может быть выполнен без проблем. JSMin доступен как исполняемый файл Windows с исходным кодом на C и многих других языках.

Веб-сайт: www.crockford.com/javascript/jsmin.html

Dojo ShrinkSafe

Те же люди, ответственные за Dojo Toolkit, имеют инструмент под названием *ShrinkSafe*, который использует интерпретатор Rhino в JavaScript, чтобы сначала анализировать код JavaScript в потоке токенов, а затем использовать его для безопасного кодирования. Как и в случае с JSMin, ShrinkSafe удаляет лишние пробелы (но не разрывы строк) и комментарии, но также делает еще один шаг вперед и заменяет имена локальных переменных двухсимвольными именами. Результатом является улучшенная производительность по сравнению с JSMin, без риска появления синтаксических ошибок.

Веб-сайт: http://shrinksafe.dojotoolkit.org/

СРЕДСТВА МОДУЛЬНОГО ТЕСТИРОВАНИЯ

Большинство библиотек JavaScript используют некоторую форму модульного тестирования для своего собственного кода, а некоторые публикуют среду модульного тестирования для использования другими. Разработка через тестирование (Test-driven development , TDD) — это процесс разработки программного обеспечения, основанный на использовании модульного тестирования.

Mocha

Mocha предлагает отличную конфигурируемость и расширяемость при разработке модульных тестов. Тесты достаточно гибкие, а их последовательное выполнение означает точную отчетность и более простую отладку.

Веб-сайт: https://mochajs.org/

Jasmine

Несмотря на то что он находится на более старой стороне спектра модульных тестов, *Jasmine* по-прежнему чрезвычайно популярен. Он поставляется со всем необходимым, без каких-либо внешних зависимостей, а его синтаксис прост и удобен для чтения.

Веб-сайт: https://jasmine.github.io/

qUnit

qUnit — это среда модульного тестирования, разработанная для использования с jQuery. Действительно, сам jQuery использует qUnit для всего своего тестирования. Несмотря на это, qUnit не зависит от jQuery и может использоваться для тестирования любого JS-кода. qUnit известен как очень простой фреймворк для модульного тестирования, позволяющий легко приступить к работе.

Веб-сайт: https://github.com/jquery/qunit

JsUnit

Исходная библиотека модульного тестирования JavaScript не привязана к какой-либо конкретной библиотеке. *JsUnit* — это порт популярной среды тестирования JUnit для Java. Тесты выполняются на странице и могут быть настроены для автоматического тестирования и отправки результатов на сервер. Сайт содержит примеры и основную документацию.

Веб-сайт: www.jsunit.net/

Dojo Object Harness

Dojo Object Harness (DOH) начинался как инструмент внутреннего юнит-тестирования для Dojo, а затем был выпущен для всеобщего использования. Как и в других средах, модульные тесты запускаются внутри браузера.

Веб-сайт: www.dojotoolkit.org/

СРЕДСТВА ГЕНЕРАЦИИ ДОКУМЕНТАЦИИ

Большинство IDE добавляют *генераторы документации* для основного языка. Поскольку в JavaScript нет официальной IDE, документация традиционно выполняется вручную или с помощью перепрофилирования генераторов документации для других языков. Однако существует ряд генераторов документации, специально предназначенных для JavaScript.

ESDoc

ESDoc способен генерировать очень продвинутые страницы документации для кода, включая возможность размещать ссылки на исходный код со страницы самой документации. Он также имеет библиотеку дополнений для расширения своих возможностей. Однако ESDoc требует, чтобы кодовая база состояла исключительно из модулей ES6.

Веб-сайт: https://esdoc.org/

documentation.js

documentation.js обрабатывает комментарии JSDoc внутри кода для автоматической генерации документации в HTML, Markdown или JSON. Он совместим с последними версиями ECMAScript и всеми основными инструментами сборки, а также может работать с аннотациями Flow.

Веб-сайт: https://documentation.js.org/

Docco

По данным веб-сайта, *Docco* является «быстрым и грязным» генератором документации. Мотивация этого инструмента заключается в том, что это очень простой способ создания HTML-страниц, которые описывают кодовую базу. Docco хрупок в некоторых отношениях, но он предлагает наименьшее количество барьеров для создания документации напрямую из кода.

Веб-сайт: http://ashkenas.com/docco/

JsDoc Toolkit

JsDoc Toolkit был одним из первых генераторов документации JavaScript. Требуется ввести Javadoc-подобные комментарии в исходный код, которые затем будут обработаны и выведены в виде HTML-файлов. Можно настроить формат HTML с помощью одного из готовых шаблонов JsDoc или создать свой собственный. Набор инструментов JsDoc доступен в виде пакета Java.

Веб-сайт: https://github.com/jsdoc3/jsdoc/

YUI Doc

YUI Doc — генератор документации YUI. Генератор написан на Python, поэтому ему нужна среда исполнения Python для установки. YUI Doc выводит файлы HTML со встроенными поисками свойств и методов, реализованными с использованием виджета автозаполнения YUI. Как в случае с JsDoc, YUI Doc требует, чтобы Javadoc-подобные комментарии были вставлены в исходный код. HTML по умолчанию можно изменить, изменив файл шаблона HTML по умолчанию и связанную таблицу стилей.

Веб-сайт: www.yuilibrary.com/projects/yuidoc/

AjaxDoc

Цель *AjaxDoc* немного отличается от целей предыдущих генераторов. Вместо создания HTML-файлов для документации JavaScript, он создает XML-файлы в том же формате, что и файлы, созданные для языков .NET, таких как C # и Visual Basic .NET. Это позволяет стандартным генераторам документации .NET создавать документацию в виде файлов HTML. AjaxDoc требует формат комментариев к документации, аналогичный комментариям к документации для всех языков .NET. AjaxDoc был создан для использования с решениями ASP.NET Ajax, но его можно использовать и в отдельных проектах.

Веб-сайт: www.codeplex.com/ajaxdoc/