3-е издание

Беэр Бибо, Иегуда Кац, Аурелио де Роза

jQuery в действии

С предисловиями
Дейва Метвина,
Джона Резига

MANNING

jQuery in Action

THIRD EDITION

BEAR BIBEAULT
YEHUDA KATZ
AURELIO DE ROSA

MANNING
Shelter Island

Беэр Бибо, Иегуда Кац, Аурелио де Роза

jQuery в действии

3-е издание

ПИТЕР®
Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону • Самара • Минск

2017

ББК 32.988.02-018
УДК 004.738.5
Б59

Бибо Беэр, Кац Иегуда, де Роза Аурелио
Б59 jQuery в действии. 3-е издание. — СПб.: Питер, 2017. — 528 с.: ил. — (Серия «Для профессионалов»).

ISBN 978-5-496-02973-5

Третье издание книги «jQuery в действии» — это динамичное и исчерпывающее руководство по библиотеке jQuery. В книге рассматриваются задачи, с которыми приходится сталкиваться при реализации практически любого веб-проекта. Книга ориентирована на читателей, обладающих минимальным опытом JavaScript, содержит новые примеры и упражнения, а также глубоко и практично раскрывает темы, связанные с этой библиотекой. Вы узнаете, как делать обход HTML-документов, обрабатывать события, создавать анимацию, писать плагины, и даже освоите модульное тестирование кода. Уникальные лабораторные работы помогают закрепить каждую концепцию на реальных примерах кода. В книгу добавлено несколько новых глав, из которых вы узнаете, как работать с новейшими фреймворками и одностраничными приложениями.

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

ББК 32.988.02-018
УДК 004.738.5

ISBN 978-1617292071 англ.
ISBN 978-5-496-02973-5

Краткое содержание

Оглавление

Часть II. Основы jQuery

Часть III. Дополнительные возможности

Аннарите, которая сбалансировала
мою жизнь.

Аурелио

Предисловие к третьему изданию

Десять лет назад Джон Резиг выпустил библиотеку JavaScript, с помощью которой люди смогли бы упростить создание сайтов. По данным BuiltWeb.com, сегодня эта библиотека под названием jQuery применяется более чем на 80 % всех сайтов, задействующих JavaScript. Было бы сложно назвать себя веб-разработчиком без знания jQuery.

С технической стороны jQuery упрощает громоздкие нативные вызовы методов, используемые браузерами, и сокращает количество строк кода, требуемого для их выполнения. Вот почему лейтмотив jQuery — «Пишите меньше, делайте больше». jQuery также нивелирует различия в поведении — даже если это откровенные ошибки, — существующие в браузерах. Это упрощает и разработку, и тестирование.

С самого начала библиотека разрабатывалась с учетом возможности ее расширения другими людьми. Модель плагина jQuery позволяет каждому делать все что угодно — от создания лайтбокса до подтверждения соответствия форм. В результате многие люди даже без большого опыта программирования могут создавать красивые и функциональные сайты, пользуясь результатами работы, выполненной другими.

Впрочем, не код сам по себе сделал jQuery популярной. С самого начала инициативные разработчики, количество которых увеличивалось с каждым днем, заполнили онлайн-форумы и рассылки, отвечая на вопросы новичков. Знания, полученные в тех дискуссиях, привели к улучшению документации, учебным курсам и появлению книг вроде этой.

Книга «jQuery в действии» — отличный инструмент для изучения jQuery. Уже на первоначальном этапе она объясняет основной принцип jQuery API, который заключается в выборе ряда элементов на веб-странице и выполнении с ними неких действий. Тот же принцип работает, если вы скрываете, показываете, анимируете, убираете или изменяете элемент. В процессе выбора используется стандартный синтаксис селектора CSS с некоторыми усовершенствованиями jQuery, делающими процедуру выбора еще более эффективной.

Должен признать, что глава о событиях — моя любимая, поскольку мой наибольший вклад в код jQuery — переписанная модель событий в jQuery 1.7. В этой главе объясняются цели и полезность событий, которые позволяют известить вас о взаимодействии пользователя с веб-страницей. Практически любая выполняемая вами операция jQuery начинается с определенного рода события.

Я также рад тому, что в книге описаны многие темы, которыми часто пренебрегают, например модульное тестирование и организация больших процессов. Ведь нередко из малых проектов вырастают большие, и приведенные в издании советы помогут вам управлять этим ростом так, чтобы уменьшить хлопоты, связанные с поддержкой данных проектов.

В главах о создании демоприложений вы увидите, как все части jQuery сходятся воедино, и познакомитесь с важными понятиями, например шаблонизацией — ключевым понятием во всех современных фреймворках и приложениях JavaScript. Даже сейчас я немного поражаюсь таким демоприложениям, показывающим, как можно создать что-то дельное, используя минимум кода.

Аурелио де Роза уже несколько лет состоит в сообществе jQuery и является членом команды jQuery, обеспечивающей поддержку актуальности онлайн-документации библиотеки. Благодаря его работе в третьей редакции «jQuery в действии» вы получите актуальную информацию, соответствующую последней версии jQuery. В процессе написания этой книги Аурелио улучшил онлайн-документацию, находя несоответствия и пропущенные сведения. Вы, как ее читатель и вскоре jQuery-разработчик, — счастливчик. Идите вперед — и «пишите меньше, делайте больше»!

Дэйв Метвин,
президент Фонда jQuery

Предисловие к первому изданию

Все дело в простоте. Почему веб-разработчики вынуждены писать длинные, сложные, размером с книгу куски кода, когда они хотят создать простые компоненты для взаимодействия? Нет никаких оснований полагать, что сложность должна быть обязательным требованием в разработке приложений.

Когда я только вознамерился создать jQuery, я решил, что хочу уделить особое внимание тому, чтобы код оставался небольшим, простым, способным прослужить всем практическим приложениям, с которыми веб-разработчики ежедневно имеют дело. Я был очень доволен, прочитав книгу «jQuery в действии», так как увидел в ней отличное воплощение принципов библиотеки.

Благодаря подробному описанию кода, применимого на практике в реальной жизни и представленного в кратком и сжатом формате, «jQuery в действии» послужит идеальным ресурсом для всех желающих познакомиться с библиотекой.

Что мне больше всего понравилось в данной книге — значительный интерес к тем деталям, на которые Беэр и Иегуда обращали внимание в своих работах над библиотекой. Они очень основательно подошли к изучению и распространению jQuery API. Наверное, не было и дня, когда по электронной почте или СМС мне не приходили бы от них вопросы, сообщения о свежеобнаруженных ошибках или рекомендации об улучшениях. Будьте уверены: перед вами одно из наиболее тщательно продуманных и проанализированных изданий по библиотеке jQuery в современной литературе.

Одна из особенностей книги, удививших меня еще в содержании, — явное включение плагинов jQuery и тактика и теория их разработки. Библиотека jQuery остается такой простой потому, что имеет плагинную архитектуру. Она обеспечивает ряд задокументированных точек расширения, на которых плагины могут добавлять функциональность. Зачастую эта функциональность, несмотря на свою полезность, была бы слишком специфичной для добавления в саму библиотеку — и это делает необходимой плагинную архитектуру. Отдельные плагины, такие как Forms, Dimension и Live-Query, обсуждаются в книге. Они получили широкое распространение, и причина очевидна: эти плагины написаны, задокументированы и поддерживаются на высшем уровне. Обязательно обратите внимание на то, как они были созданы и используются, поскольку их применение имеет фундаментальное значение в эксплуатации библиотеки.

С такими ресурсами, как эта книга, проект jQuery определенно будет продолжать успешно развиваться. Я надеюсь, что издание сослужит хорошую службу в исследовании и использовании вами jQuery.

Джон Резиг,
создатель jQuery

Введение

Удивительно, сколько работы и усилий я вложил в эту книгу. Когда в издательстве Manning мне предложили написать «jQuery в действии», я знал, что работа не будет похожа на увеселительную прогулку в парке, но определенно недооценил задачу. Я подумал: «Ну, это будет несложно. Несколько месяцев — и все сделаю». Спустя два года и многие ночи за работой я не жалею о своем выборе. Написание этой книги — невероятный путь, позволивший мне улучшить многие навыки в разных сферах: я повысил свой уровень как разработчика, научился писать книги и усовершенствовал свои навыки в jQuery.

Два года назад я был веб-разработчиком, страстно увлеченным jQuery, и радовался тому, что библиотека позволяет мне совершенно бесплатно решать множество проблем. До того как я приступил к этому проекту, мне казалось, что я хорошо знаю jQuery. Но, вне всяких сомнений, написание и пересмотр глав, которые вы прочтете, вынуждали меня копать значительно глубже, и в результате я смог получить новые навыки и перейти на следующий уровень. Помимо того, у меня была возможность открыть для себя новые темы, связанные с jQuery и ее документацией. Пересмотр книги позволил мне вносить свой вклад в библиотеку на регулярной основе — вклад столь большой, что меня пригласили присоединиться к команде разработчиков jQuery и стать частью их замечательного проекта. Разумеется, это было неожиданное и очень желанное достижение, и я горжусь им.

Теперь, когда вы знаете, с чего все началось, подумаем над ключевым вопросом: так ли необходимо было третье издание? Я думаю, да, и это связано с двумя основными причинами. Предыдущее издание книги охватывает jQuery до версии 1.4, в то время как последняя версия — 1.11, и скоро выйдет jQuery 3[1] (она также рассматривается в книге). Вторая причина заключается в том, что jQuery определенно самая используемая библиотека JavaScript. Она применяется в 63 % из миллиона наиболее посещаемых сайтов в мире. Два этих факта должны привести вас к пониманию того, что многое изменилось с тех пор, как было опубликовано второе издание «jQuery в действии», и библиотека не только по-прежнему актуальна, но и не собирается исчезать в обозримом будущем.

В третьем издании книги вы увидите довольно много изменений. Прежде всего, я удалил главы о jQuery UI, так как и jQuery, и jQuery UI выросли настолько, что заслуживают отдельных книг. Кроме того, я решил добавить несколько дополнительных тем, которые не были описаны ранее. И наконец, я включил много новых примеров, лабораторных работ, фрагментов кода, демонстрационных версий и др., чтобы сделать это издание еще лучше.

Переверните страницу, погрузитесь в книгу и начните изучение наиболее часто используемой в мире библиотеки JavaScript. Удачи!

Аурелио де Роза

---

[1] Версия 3.1.1 вышла 22 сентября 2016 г. — *Примеч. ред.*

Благодарности

Написание (хорошей) книги занимает много времени и, кроме того, требует вклада множества людей с различными знаниями, умениями и навыками, чтобы ее можно было подготовить и издать. Так происходило с любой изданной успешной книгой, так было и с книгой, которую вы сейчас держите в руках, и с ее предыдущими изданиями. Коллектив издательства Manning напряженно работал, чтобы привести эту книгу к должному качеству, и я благодарю всех сотрудников за приложенные ими усилия. В «заключительных титрах» книги хотелось бы указать не только издателя, Марьяна Бейса, но и следующих людей: Эла Шерера, Ану Ромак, Кендейс Гиллхулли, Синтию Кейн, Дотти Марсико, Джеффа Блаела, Кевина Салливана, Линду Ректенвальд, Мэри Пиргис, Мелоди Добал, Озрена Харловица, Робира де Йонга, Скотта Мейерса и Шона Денниса. Я благодарю каждого из них, а также многих других, кто работал «за кулисами».

Большое спасибо рецензентам, которые помогли найти ошибки, от простых опечаток до ошибок в терминологии и коде. Количество людей, рецензировавших эту книгу, вас наверняка удивит, но они очень помогли. За вклад и проницательность я хотел бы поблагодарить Криса Маки, Кристофера Хаупта, Чака Дерфри, Франческо Бианки, Гэри А. Стеффорда, Грегора Зуровски, Яна Гойвертса, Жана-Франсуа Морена, Джона Д. Льюиса, Джона Стемпера, Карена Кристенсона, Кейт Вебстер, Мэтта Форсинта, Рикардо Микса, Сурая Кумара, Вильяма Э. Вилера и Вилли Робертса.

Отдельная благодарность Ричарду Скотту-Робинсону, работавшему научным редактором. Он вложил множество сил и времени (и я уверен, что это не было просто или быстро) для проверки каждого примера кода из книги в разных средах. Он также внес бесценный вклад в техническую точность текста и комментарии по существу. Большая часть их включена в книгу, которую вы держите в руках (или читаете ее цифровую копию).

Искренняя благодарность Дэйву Метвину за написание предисловия к данному изданию с одобрением моей работы и Беэру Бибо и Иегуде Кацу за написание двух популярных изданий, предшествовавших этому.

В личном отношении наиболее значимый человек, которого я бы хотел поблагодарить, — моя будущая жена Аннарита. Твои любовь, терпение, ласка очень важны для меня. За эти два года ты не раз жаловалась, что я работаю над проектом, вместо того чтобы проводить время с тобой. Твои поддержка и понимание были неоценимыми, и поэтому я посвящаю эту книгу тебе. Ты, моя дорогая Аннарита, сбалансировала мою жизнь. Спасибо за те прекрасные моменты, которые мы провели вместе, и те, что еще впереди. Я люблю тебя.

Большое спасибо также моей семье: Рафаэль, Евфемии, Джузи, Виоле, моим бабушкам Джузеппине и Анне, моему дедушке Аурелио. Спасибо вам за всю вашу любовь. Вы поддерживали меня так, как могли, и я обязан вам многим.

Я хочу также поблагодарить Франческо Палладино. Ты лучший друг, который может быть у человека. Ты всегда был со мной, когда мне это было нужно. Я желаю тебе лучшего, что может предложить жизнь, и пусть все твои мечты исполнятся.

И раз уж я заговорил о мечтах — хочу посвятить эту книгу всем людям, которые страстно верят в свои мечты. Не прекращайте верить из-за того, что вам говорят другие люди, даже если трудно продолжать двигаться вперед. Наступит день, и вы достигнете их. Всем мечтателям я желаю удачи.

Я хочу поблагодарить всех людей, которые внесли вклад в формирование меня как того человека, каким я стал, в той или иной степени: Альберта Эйнштейна, Людвига ван Бетховена, Луция Аннея Сенеку, Роберто де Росу, Леонардо Гристолио и неизвестного продавца зонтиков.

Наконец, я хочу поблагодарить всех людей в команде jQuery. Если я написал хорошую книгу, то лишь благодаря замечательной работе, проделанной вами за эти годы. Вы классные!

Аурелио де Роза

Об этой книге

Эта книга — для веб-разработчиков, желающих углубиться в jQuery, наиболее популярную и используемую библиотеку JavaScript. Наша цель, дорогой читатель, — сделать вас профессионалом jQuery, будь ваш уровень начальным или продвинутым. Книга охватывает всю библиотеку целиком, включая некоторые дополнительные инструменты и фреймворки, такие как Bower и QUnit, рекомендуя лучшие практики. Каждый метод API представлен в удобном для применения синтаксическом блоке, описывающем параметры и возвращаемые значения.

Третье издание «jQuery в действии» охватывает темы от простых — что такое jQuery и как ее добавить на веб-страницу — до продвинутых, таких как способ, с помощью которого библиотека реализует промисы, и как создать jQuery-плагин. Книга содержит множество полезных примеров, три плагина и три типовых проекта. Она также включает то, что мы называем Lab Pages. Эти содержательные и забавные работы помогут вам увидеть нюансы методов «jQuery в действии» без необходимости самостоятельно писать множество строк кода.

Материал книги предполагает наличие у читателей фундаментальных знаний HTML, CSS и JavaScript. Предварительных знаний jQuery не требуется, но они могут помочь вам быстрее уловить общую идею.

Дорожная карта

Мы разделили книгу на три части: введение в jQuery, что собой представляет библиотека и для чего применяется, основы jQuery с описанием всех ее особенностей и темы продвинутого уровня.

Глава 1 дает представление о философии jQuery и о том, как эта библиотека следует так называемым принципам ненавязчивого JavaScript. Здесь обсуждается, что такое jQuery, какие проблемы она пытается решить и почему вы можете захотеть задействовать ее в ваших веб-проектах.

Глава 2 охватывает выбор DOM-элементов на основе использования селекторов и повествует о том, как создать ваш собственный селектор. Мы также познакомим вас с таким термином, как *коллекция jQuery* (или *объект jQuery*), применяемым для ссылки на объект JavaScript, возвращаемый методами jQuery. Он содержит набор элементов, выбирая которые вы можете работать с библиотекой.

Глава 3 расширяет главу 2, показывая, как улучшить или создать новую выборку элементов, начиная с предыдущей выборки. Вы также научитесь создавать новые DOM-элементы с помощью jQuery.

Глава 4 фокусируется на многих методах jQuery, предназначенных для работы с атрибутами и свойствами, и различиях между ними. Кроме того, она объясняет, как хранить пользовательские данные в одном или нескольких DOM-элементах.

Глава 5 целиком посвящена работе с именами класса элемента, клонированию и установке содержимого DOM-элементов, изменению дерева DOM посредством добавления, перемещения или замены элементов.

Глава 6 знакомит вас с различными моделями событий и тем, как браузеры позволяют устанавливать обработчики для управления происходящим при возникновении события. Затем мы рассмотрим, как jQuery дает возможность разработчикам делать те же вещи, избегая проблем с несовместимостью браузера. Кроме того, глава описывает такие важные понятия, как *делегирование события* и *всплытие события*.

Глава 7 отличается от предыдущих, ее цель — подробно рассказать вам о разработке проекта, локатора DVD, где вы можете применить полученные до этого момента знания.

Глава 8 рассматривает методы, применяемые для отображения и скрытия элементов, и способы создания анимации. Кроме того, уделяется внимание функции очереди для последовательно запущенных эффектов, равно как и общих функций.

Глава 9 посвящена вспомогательным функциям — функциям с пространством имен в jQuery, обычно не работающих с DOM-элементами.

Глава 10 рассматривает одно из важнейших понятий последних лет — Ajax. Мы увидим, как jQuery облегчает использование Ajax на веб-страницах, защищая нас от всех популярных подводных камней, при этом значительно упрощая наиболее распространенные типы взаимодействий Ajax (такие как возврат объектов JSON).

Глава 11 представляет решение проблемы, с которой сталкиваются многие разработчики в реальном мире: создание работающей контактной формы, не требующей полного обновления страницы для уведомления пользователя об успешной или неудачной отправке сообщения.

Глава 12 — первая из части III, где мы переходим к более сложным вопросам, большая часть которых не связана строго с ядром библиотеки. Здесь мы обсудим, как расширить функциональность jQuery с помощью создания плагинов для нее. Эти плагины могут быть в двух вариантах: методы и вспомогательные функции. Мы рассмотрим оба варианта.

Глава 13 объясняет, как избежать того, что называется *callback hell* (дословно — «ад обратного вызова»), описывая реализацию jQuery-промисов. Как вы узнаете, это тонкая и спорная тема, служившая предметом дискуссий на протяжении многих лет.

Глава 14 познакомит вас с тестированием: что это и почему важно. Мы сосредоточим внимание на одном конкретном виде тестирования — модульном. Затем рассмотрим QUnit, JavaScript-фреймворк, применяемый некоторыми проектами jQuery (jQuery, jQuery UI и jQuery Mobile) для модульного тестирования кода.

Глава 15, последняя, начинается с советов и рекомендаций по улучшению производительности кода, который использует jQuery, благодаря правильному способу выбора элементов. Затем мы рассмотрим несколько инструментов, фреймворков и шаблонов, не относящихся напрямую к библиотеке, но тех, которые можно применять для написания быстрого, надежного и красивого кода. В частности, эта глава

объясняет, как организовать ваш код в модулях, как загрузить модули, используя RequireJS, и как управлять зависимостями клиентской стороны с помощью Bower. Наконец, мы покажем вам, как jQuery вписывается в приложения одной страницы, бегло просмотрев Backbone.js.

В довершение ко всему в книге есть приложение, освещающее основные понятия JavaScript, такие как контексты функций и замыкания, — немаловажные для наиболее эффективного применения jQuery на наших страницах — для читателей, незнакомых с этими понятиями или желающих освежить их в памяти.

Соглашения и загрузка исходного кода

Исходный код в книге, во фрагментах или в листингах кода, написан моноширинным шрифтом, подобным этому, что выделяет его среди остального текста. В некоторых листингах код сопровождается комментариями для указания ключевых понятий, а пронумерованные маркеры иногда предоставляют дополнительную информацию о коде. Он форматирован так, чтобы поместиться на странице, с помощью переносов строк и отступов.

Весь исходный код для примеров в книге доступен по ссылке на GitHub: https://github.com/AurelioDeRosa/jquery-in-action. Кроме того, исходный код доступен для загрузки с сайта издательства на www.manning.com/derosa/ или www.manning.com/jquery-in-action-third-edition.

Требования к ПО

Примеры кода для этой книги сгруппированы по папкам, по одной для каждой главы. Они готовы к использованию на локальном веб-сервере, таком как Apache HTTP Server (http://httpd.apache.org/). За исключением проектов, создаваемых в главах 7 и 10, и еще нескольких случаев, примеры не требуют наличия веб-сервера и могут быть загружены для выполнения непосредственно в браузер. Проекту в главе 10 необходимо большее взаимодействие с серверной частью, чем может предоставить Apache, так что для локального запуска понадобится PHP для Apache.

Все примеры тестировались в различных браузерах, включая Internet Explorer, Firefox, Safari, Opera и Chrome.

Об обложке

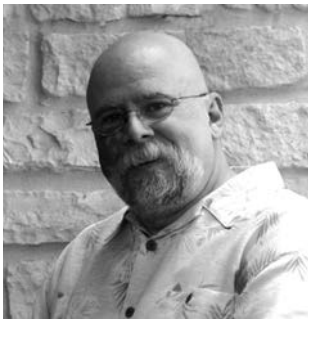
Изображение на обложке «jQuery в действии. 3-е издание» — это «Дозорный». Иллюстрация взята из французской книги о путешествиях, *Encyclopédie des Voyages* (автор Жак Грассе де Сен-Совер), опубликованной почти 200 лет назад. Путешествия ради удовольствия были относительно новым явлением в то время, и справочники, подобные этому, были очень популярны, знакомя и настоящих,

и «диванных» туристов как с другими регионами мира, так и с региональными костюмами и униформой французских солдат, гражданских служащих, торговцев, купцов и крестьян.

Разнообразие рисунков в *Encyclopédie de Voyages* отражает уникальность и индивидуальность городов и регионов мира двухсотлетней давности. Изолированные друг от друга, люди говорили на разных диалектах и языках. На улицах или в сельской местности было несложно определить, где человек жил и чем занимался, каков был его социальный статус, просто по тому, как он говорил или какую одежду носил. С тех времен дресс-код изменился, и нет такого регионального разнообразия, столь богатого в тот период. Сейчас зачастую непросто даже определить, с какого континента человек. Наверное, если посмотреть на это с оптимизмом, то мы променяли культурное и визуальное многообразие на более разнообразную личную жизнь — или, возможно, более разнообразную и интересную интеллектуальную и техническую жизнь.

Мы в Manning прославляем изобретательность, инициативу и удовольствие от компьютерного бизнеса, создавая обложки книг, отражающие богатое разнообразие региональной жизни два столетия назад.

Об авторах



Беэр Бибо создает программное обеспечение уже более трех десятилетий, начиная с игры в крестики-нолики, написанной на суперкомпьютере Control Data Cyber через 100-бодный телетайп. С двумя дипломами по электротехнике Беэр должен был разрабатывать антенны или что-то в этом роде, но, поскольку его первым местом работы стала корпорация Digital Equipment, он всегда в куда большей степени увлекался программированием.

Кроме того, Беэр некоторое время сотрудничал с такими компаниями, как Lightbridge Inc., BMC Software, Dragon Systems, Works.com и рядом других. Он даже работал в армии США, обучая пехотинцев взрывать танки; эти навыки порой могут очень пригодиться во время ежедневных скрам-митингов. В настоящее время Беэр является ведущим веб-разработчиком в крупной компании — разработчике ПО для обеспечения хранения объектов.

Помимо своей основной работы, Беэр также пишет книги (надо же!) и запустил небольшой бизнес, который создает веб-приложения и предлагает их другим медиасервисам, а также помогает модерировать JavaRanch.com как «шериф» (старший модератор). В свободное время Беэр любит готовить (что сказывается на размере его джинсов), балуется с фотографиями и видео, ездит на Yamaha V-Star и носит футболки с тропическими принтами. Он работает и живет в Техасе, в Остине, городе, который он нежно любит.

Иегуда Кац в течение последних нескольких лет участвует в ряде проектов с открытым исходным кодом. Помимо членства в основной команде проекта jQuery, он принимает участие и в Merb, альтернативе Ruby on Rails (написанной также на Ruby).

Иегуда родился в штате Миннесота, вырос в Нью-Йорке, а сейчас живет в Калифорнии, в солнечном городе Санта-Барбара. Работал над сайтами New York Times, Allure Magazine, Architectural Digest, Yoga Journal и других столь же выдающихся клиентов. Он профессионально программирует на ряде языков, включая Java, Ruby, PHP и JavaScript.

В свое свободное время он поддерживает VisualjQuery.com и помогает ответами на вопросы новым пользователям Query на IRC-канале и в официальном списке рассылки jQuery.

Аурелио де Роза — старший веб-разработчик (полный стек) более чем с пятилетним профессиональным опытом веб-программирования с использованием стека WAMP и HTML5, CSS3, Sass, JavaScript и PHP. Он член команд jQuery и JoindIn и эксперт в API JavaScript и HTML5. Ему также интересны веб-безопасность, доступность, производительность и SEO.

Когда Аурелио не занят написанием кода, он обычный писатель, лектор, автор книг и соавтор некоторых научных работ.

Часть I
Начинаем работу с jQuery

Если вы читаете эту страницу, то наверняка слышали о jQuery от знакомого разработчика или прочитали о ней на сайте или форуме, либо хотите понять, что это за библиотека и для чего она нужна. Возможно, вы пользуетесь ею на работе и хотите улучшить свои навыки, чтобы произвести впечатление на босса. Или, возможно, вы никогда и не слышали о jQuery и ваше внимание привлекла красивая картинка на обложке книги. Какая бы причина ни сподвигла вас открыть ее и прочитать эту страницу, следующая глава, надеемся, даст вам все необходимые объяснения.

В этой единственной главе, относящейся к части I, вы узнаете о том, что такое jQuery, какие проблемы она пытается решить и почему вы можете захотеть применять ее в ваших веб-проектах. В главе 1 мы научим вас, как не запутаться в различных доступных версиях jQuery и решить, какая из них наилучшим образом подходит для ваших потребностей. Если вы уже занимаетесь веб-разработкой и хотите научиться профессионально пользоваться наиболее популярной библиотекой в мире, то перейдите к главе 1 и начните удивительное путешествие по страницам книги.

1 Знакомство с jQuery

В этой главе:

- что такое jQuery и почему вы должны ее использовать;
- ненавязчивая стратегия JavaScript;
- выбор правильной версии jQuery;
- основные элементы и понятия jQuery.

«Есть только два типа языков: на одни люди жалуются, а другими никто не пользуется». Как хорошо эта фраза, сказанная Бьярном Страуструпом, разработавшим и реализовавшим C++, описывает чувства по поводу JavaScript! На этот язык, как и некоторые другие (яркий пример — PHP), сетовали как на «плохой» в течение нескольких лет. Затем случилось что-то сверхъестественное. Благодаря подъему Ajax, выпуску отдельных библиотек, таких как Prototype, Moo Tools и jQuery, и новым высокоинтерактивным веб-приложениям (которые, как вы наверняка слышали, иногда называют *single-page applications* (одностраничными приложениями)), разработчики начали понимать потенциал JavaScript. На сегодняшний день он также является одним из наиболее распространенных языков благодаря Node.js, платформе, позволяющей задействовать его как язык серверной части, и PhoneGap, фреймворку для создания гибридных мобильных приложений.

> jQuery — бесплатная (под лицензией MIT) популярная библиотека JavaScript, созданная Джоном Резигом в 2006 году, разработанная для упрощения написания HTML-кода для клиентской стороны. Как заявлено на сайте jQuery, это быстрая небольшая библиотека JavaScript с богатыми возможностями. Она значительно быстрее делает такие вещи, как обход и обработка HTML-документа, обработка событий, анимации и Ajax, благодаря простому в использовании API, работающему с множеством браузеров. Сочетая универсальность и расширяемость, jQuery изменила способ, которым миллионы разработчиков и дизайнеров пишут JavaScript.

Конечно, вы можете счесть это утверждение саморекламой, тем не менее все сказанное — чистая правда. jQuery *действительно* изменила способ написания кода. К ней обращаются настолько часто, что, согласно статистическим данным

от BuiltWith (за апрель 2015), jQuery используется в 63 % из миллиона наиболее посещаемых сайтов в мире (http://trends.builtwith.com/javascript/jQuery). Цитируемая прежде библиотека Moo Tools, ее ближайший конкурент, применяется лишь в 3 % (http://trends.builtwith.com/javascript/MooTools), а Prototype — и вовсе в 2,5 % (http://trends.builtwith.com/javascript/Prototype).

jQuery пользуются важнейшие компании и сайты в мире, такие как Microsoft, Amazon, Dell, Etsy, Netflix, Best Buy, Instagram, Fox News, GoDaddy и многие другие. Если у вас были какие-либо сомнения по поводу jQuery, то представленные сведения должны убедить вас, что это стабильная и надежная библиотека, арсеналом которой вы можете оперировать в своих проектах.

Наша книга охватывает многие аспекты библиотеки, начиная от базовых понятий, таких как селекторы и методы обхода объектной модели документа (Document Object Model — DOM), до более продвинутых, таких как расширение функциональности (создание плагинов), улучшение производительности вашего кода и тестирование. Подразумевается, что у вас уже есть минимальные знания JavaScript. При необходимости освежить эти знания загляните в приложение. Если вы незнакомы с языком, то настоящий текст может показаться вам слишком сложным, поэтому предлагаем подучиться и вернуться к книге позже. Мы подождем.

Вернулись? Рады видеть вас снова! Начнем с самого начала, то есть с обсуждения того, что jQuery готова предложить и как может помочь вам в процессе веб-разработки.

1.1. Пишите меньше, делайте больше

Слоган jQuery — «Пишите меньше, делайте больше». Если вы хоть раз пробовали добавить динамическую функциональность к вашим страницам, то знаете, что выполнение простых задач с использованием сырого JavaScript может привести к написанию десятков строк кода. Джон Резиг специально создал библиотеку jQuery, чтобы сделать распространенные задачи тривиальными и простыми в освоении, совмещая их с решением проблем, вызванных несовместимостью браузеров.

Например, каждый, кто имел дело с группами переключателей в JavaScript, знает, что это предельно скучная затея — выяснять, какое положение переключателя в группе сейчас выбрано, и получать их атрибуты `value`. Группа должна быть обнаружена, результирующий набор элементов переключателей должен быть проверен один за другим для выяснения, у какого элемента установлен атрибут `checked`. Этот атрибут `value` элемента может быть получен позже.

Для совместимости с Internet Explorer 6 и выше (если вы игнорируете некоторые старые браузеры, существует более эффективный подход) такой код можно реализовать следующим образом:

```
var checkedValue;
var elements = document.getElementsByTagName('input');
for (var i = 0; i < elements.length; i++) {
  if (elements[i].type === 'radio' &&
      elements[i].name === 'some-radio-group' &&
```

```
      elements[i].checked) {
    checkedValue = elements[i].value;
    break;
  }
}
```

А теперь сравните с тем, как это делается с помощью jQuery:

```
var checkedValue =
  jQuery('input:radio[name="some-radio-group"]:checked').val();
```

Не беспокойтесь, если код сейчас выглядит для вас как шифровка. Вскоре вы поймете, как он работает, и сможете создать свои собственные лаконичные — но эффективные — операторы jQuery, чтобы оживить свои страницы. Суть вот в чем: мы хотим показать, как библиотека может включить множество строк кода лишь в одну.

Что делает предыдущий оператор jQuery столь коротким, так это действенность *селектора*, выражения, используемого для определения целевых элементов на странице. Он позволяет просто найти и захватить нужные вам элементы; в данном случае выбранные положения переключателей в группе.

Если вы еще не загрузили пример кода, то сейчас самое время. Он может быть получен по ссылке на веб-странице книги: https://www.manning.com/books/jquery-in-action-third-edition. Распакуйте код и загрузите в ваш браузер HTML-страницу из файла `chapter-1/radio.group.html`. Эта страница, показанная на рис. 1.1, задействует оператор jQuery, который мы сейчас рассмотрели, чтобы выяснить, в какое положение установлен переключатель.

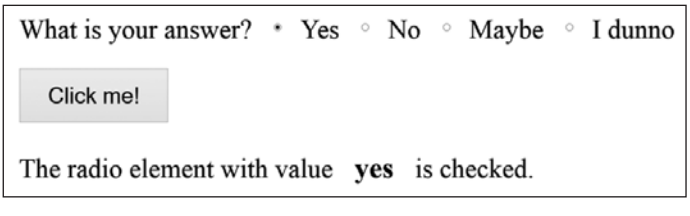


Рис. 1.1. Определить, в какое положение установлен переключатель, очень просто с помощью одной команды jQuery!

Пример показывает, как прост и лаконичен может быть написанный с помощью jQuery код. Это не единственное доказательство реальной мощи библиотеки, в противном случае она не была бы столь востребованной. Тем не менее одна из сильнейших сторон jQuery заключается в возможности извлекать элементы, прибегая к сложным селекторам, не заботясь о кросс-браузерной совместимости, особенно в старых браузерах.

Делая выбор, вы полагаетесь на два элемента: метод и селектор. На сегодняшний день все больше браузеров поддерживают нативные методы для выбора элементов, как `document.querySelector()` и `document.querySelectorAll()`. Они позволят применять более сложные селекторы вместо обычного выбора по ID или классу.

Кроме того, новые CSS3-селекторы широко поддерживаются современными браузерами. Если вы нацелены на поддержку только современных браузеров, то вам

не обязательно волноваться о подключении библиотеки на вашем сайте. Тот факт, что многие люди до сих пор пользуются старыми браузерами, которые вам может понадобиться поддерживать, может стать реальной проблемой, поскольку вам придется иметь дело со всеми несовместимостями. Это одна из главных причин применения jQuery. Библиотека позволит вам надежно использовать свои селекторы, не беспокоясь о том, что код не работает в браузерах, не поддерживающих его изначально.

ПРИМЕЧАНИЕ

Если вам интересно, какие браузеры считаются современными сегодня, то это Internet Explorer 11 и последние версии Chrome, Opera, Firefox и Safari.

Все еще не уверены? Вот список проблем, которые вам придется решать по своему усмотрению, если вы не используете jQuery: http://goo.gl/eULyPT. К тому же, как мы уже отмечали, возможности библиотеки этим не ограничиваются, в чем вы убедитесь, прочитав остальную часть книги.

Теперь рассмотрим, как следует применять JavaScript на своих страницах.

1.2. Ненавязчивый JavaScript

Вы можете помнить те давние тяжелые времена перед появлением CSS, когда должны были смешивать стилистическую разметку с разметкой структуры документа в ваших HTML-страницах. Безусловно, это помнят все, кто так или иначе был связан с созданием страниц, но вряд ли поминают добрым словом.

Добавление CSS к вашим наборам инструментальных средств разработки веб-приложений позволяет отделить стилистическую информацию от структуры документа и отказаться от использования, например, тега `<font>`. Отделение стиля от структуры не только упрощает управление вашими документами, но и позволяет гибко менять оформление страницы благодаря изменениям различных таблиц стилей. Мало кто из вас добровольно вернулся бы в те дни, когда стили применялись с элементами HTML, но по-прежнему довольно часто встречается разметка вроде следующей:

```
<button onclick="document.getElementById('xyz').style.color='red';">
  Click Me
</button>
```

Вы можете легко заметить, что стиль элемента `button` задан не с помощью тега `<font>` и остальной устаревшей ориентированной на стиль разметки. Это определяется любыми, если таковые есть, правилами CSS (не показанными здесь), действующими на странице. Хотя указанное объявление не смешивает разметку *стиля* со структурой, оно смешивает *поведение* со структурой. Оно включает JavaScript, который должен выполняться при нажатии кнопки как часть разметки элемента `button`, задействуя атрибут `onclick` (что в данном случае меняет цвет DOM-элемента со значением ID `xyz` на красный). Теперь рассмотрим, как можно улучшить эту ситуацию.

1.2.1. Отделение поведения от структуры

По тем же причинам, по которым желательно отделить стиль от структуры в HTML-документе, очень полезно (если не сказать больше) отделить от структуры и *поведение*. В идеале HTML-страница должна быть организована, как показано на рис. 1.2, с разделенными структурой, стилем и поведением — все на своем месте.

Такая стратегия, известная как *ненавязчивый JavaScript*, теперь применяется в каждой крупной библиотеке JavaScript, помогая авторам страниц достичь полезного разделения на своих страницах. Как популяризировавшая это движение библиотека, ядро jQuery хорошо оптимизировано для легкого создания ненавязчивого JavaScript. Он рассматривает *любые* выражения или операторы JavaScript, помещенные внутри HTML-тегов или между ними `<body>` HTML-страниц, либо как атрибуты HTML-элементов (как, например, `onclick`), либо как некорректные фрагменты в блоках сценариев, размещенных где бы то ни было, кроме самого конца тела страницы.

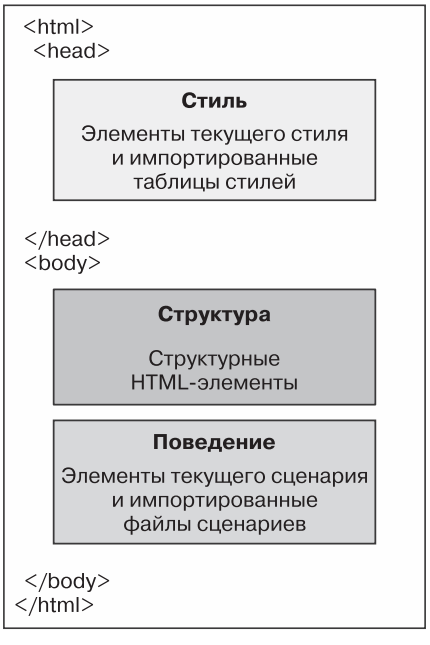


Рис. 1.2. Со структурой, стилем и поведением, аккуратно разделенными на странице, максимально улучшаются читаемость и поддерживаемость

«Но как же я смогу сделать кнопку без атрибута `onclick`?» — спросите вы. Рассмотрим следующее изменение кнопки элемента.

```
<button id="test-button">Click Me</button>
```

Намного проще! Но теперь, как вы заметили, кнопка не выполняет никаких действий. Можно нажимать ее хоть целый день — и безрезультатно. Попробуем это исправить.

1.2.2. Отделение сценария

Вместо того чтобы встраивать поведение кнопки в разметку, отделите сам сценарий, переместив его в блок `script`. Следуя современным лучшим практикам, нужно поместить его внизу страницы перед закрытием тега тела (`<body>`).

```
<script>
  document.getElementById('test-button').addEventListener(
    'click',
    function() {
      document.getElementById('xyz').style.color = 'red';
    },
    false
  );
</script>
```

Поскольку вы размещаете сценарий внизу страницы, вам не нужно добавлять обработчик, прикрепленный к событию `onload` объекта `window`, как разработчики (ошибочно) делали раньше, или ждать события `DOMContentLoaded`, доступного лишь в современных браузерах. Это событие срабатывает, когда HTML-документ полностью загружен и проанализирован, не дожидаясь окончания загрузки таблицы стилей, изображений и всего прочего. Событие `load` срабатывает после загрузки HTML-страницы и всех зависимых ресурсов. При размещении сценария в нижней части страницы, когда браузер разбирает (parses) оператор, элемент `button` остается, потому что его разметка разобрана, так что вы вполне можете дополнить его.

ПРИМЕЧАНИЕ

> Из соображений производительности элементы сценариев всегда должны размещаться в нижней части документа. Первая причина — разрешить прогрессивный рендеринг, вторая — иметь большее распараллеливание. Под первой причиной подразумевается то, что рендеринг блокируется для всего контента ниже элемента script. Под второй — то, что браузер не будет начинать другие загрузки, даже по другому имени хоста, во время загрузки элемента script.

Предыдущий фрагмент кода является еще одним примером, когда код не на 100 % совместим с браузерами, для работы в которых может разрабатываться ваш проект. Во фрагменте задействован метод JavaScript, `addEventListener()`, Internet Explorer 6–8 его не поддерживает. Как вы узнаете позже, jQuery помогает вам решить и эту проблему.

Хотя применение ненавязчивого JavaScript — эффективный способ разделения обязанностей в пределах веб-приложения, он имеет свою цену. Вы наверняка заметили, что потребовалось больше строк сценария для достижения цели, чем когда вы разместили его в разметке кнопки. Ненавязчивый JavaScript может увеличить количество строк сценария, который нужно писать, и это требует определенной дисциплины и применения хороших шаблонов кодирования сценариев клиентской стороны.

Но не спешите огорчаться: все, что убеждает вас писать код клиентской стороны так же уважительно, как и код серверной стороны, — хорошо! Но это *действительно* дополнительная работа — без библиотеки во всяком случае.

jQuery специально ориентирована на то, чтобы вы могли восхитительно просто создавать код для ваших страниц, используя техники ненавязчивого JavaScript, не заплатив за это множеством усилий или строк кода. Вы обнаружите, что эффективное применение библиотеки позволит вам сделать гораздо больше на ваших страницах при написании *меньшего* количества кода. Слоган по-прежнему «Пишите меньше, делайте больше», верно? Начнем же смотреть, как jQuery упрощает работу, чтобы вы смогли добавить без ожидаемых проблем богатые функциональные возможности на ваши страницы.

1.3. Установка jQuery

Теперь, когда вы знаете о jQuery и ее возможностях, нужно скачать библиотеку, чтобы начать непосредственную работу. Для загрузки зайдите на страницу http://jquery.com/download/. Сделав это, вы наверняка будете поражены изобилием доступных вариантов. Ветка 1.x, 2.x или 3.x? В сжатом или несжатом виде? Загрузить или использовать сеть доставки контента (Content Delivery Network, CDN)? Выбор зависит от нескольких факторов. Чтобы вы смогли сделать его осознанно, рассмотрим различия.

1.3.1. Выбор правильной версии

В апреле 2013 года команда jQuery выпустила версию 2.0 с намерением смотреть на будущее Интернета вместо его прошлого, особенно с точки зрения браузеров. До того момента jQuery поддерживала все последние версии Chrome, Firefox, Safari, Opera и Internet Explorer, начиная с 6-й версии. С введением версии 2.0 команда решила оставить позади старые версии Internet Explorer 6, 7 и 8, чтобы сосредоточиться не на том, какой Сеть была, а на том, какой она будет.

Такое решение привело к удалению множества строк кода, созданного несовместимостью старых браузеров и отсутствием в них функционала. Выполнение данной задачи привело к сокращению (на 12 %) и ускорению работы основы кода. Хотя 1.x и 2.x — две разные ветви, они тесно взаимосвязаны. Это паритет функционала между версиями 1.10 и 2.0, версиями 1.11 и 2.1 и т. д.

В октябре 2014 года Дэйв Метвин, президент Фонда jQuery (фонд, который занимается jQuery и другими проектами, — https://jquery.org/), опубликовал в блоге сообщение (http://blog.jquery.com/2014/10/29/jquery-3-0-the-next-generations/), где анонсировал план выпуска новой крупной версии библиотеки: jQuery 3. Так же, как версия 1.x поддерживает старые браузеры, а версия 2.x — современные, jQuery 3 разделяется на две версии: jQuery Compat 3 — наследница 1.x, а jQuery 3 — наследница 2.x. Далее он пояснил:

> «Мы также реструктуризируем нашу политику поддержки браузеров, начиная с этих релизов. Основной пакет jQuery остается небольшим и сжатым, поддерживая вечнозеленые браузеры (текущие и предыдущие версии определенных браузеров), которые являются распространенными на время релиза. Мы можем поддерживать дополнительные браузеры в этом пакете, основываясь на их доле на рынке. Пакет jQuery Compat предлагает поддержку значительно большего количества браузеров, но за счет большего размера файла и потенциально более низкой производительности».

При подготовке новой версии команда также воспользовалась возможностью убрать поддержку отдельных браузеров, исправить много ошибок и улучшить ряд функций.

Первый фактор, который необходимо учесть в принятии решения о требуемой версии, — это какие браузеры ваш проект должен поддерживать. В табл. 1.1 описываются браузеры, поддерживаемые каждой крупной версией jQuery.

Таблица 1.1. Обзор браузеров, поддерживаемых крупными версиями jQuery

Браузер	jQuery 1	jQuery 2	jQuery Compat 3	jQuery 3
Internet Explorer	6+	9+	8+	9+
Chrome	Текущая и предыдущая	Текущая и предыдущая	Текущая и предыдущая	Текущая и предыдущая
Firefox	Текущая и предыдущая	Текущая и предыдущая	Текущая и предыдущая	Текущая и предыдущая
Safari	5.1+	5.1+	7.0+	7.0+
Opera	12.1x. Текущая и предыдущая	12.1x. Текущая и предыдущая	Текущая и предыдущая	Текущая и предыдущая
iOS	6.1+	6.1+	7.0+	7.0+
Android	2.3, 4.0+	2.3, 4.0+	2.3, 4.0+	2.3, 4.0+

Как видите, есть определенная степень дублирования в отношении поддерживаемых версий браузеров. Но имейте в виду, что характеристика «Текущая и предыдущая» (имеется в виду текущая и предыдущая версия браузера на момент выпуска новой версии jQuery) меняется на основании даты выпуска новой версии jQuery.

Другой важный фактор для обоснования вашего решения — это где вы будете использовать jQuery. Вот несколько вариантов, которые могут помочь вам выбрать.

- Сайты, которым не нужна поддержка старых версий Internet Explorer, Opera и других браузеров, могут применять ветку 3.x. Это подходящий случай для сайтов, использующихся в контролируемой среде, например локальной сети компании.
- Сайты, целевая аудитория которых должна быть максимально широкой, например правительственные сайты, должны оперировать версией 1.x.
- Если вы разрабатываете сайт, который должен быть совместим с широкой аудиторией, но вам не нужно поддерживать Internet Explorer 6–7 и старые версии Opera и Safari, то вам следует задействовать jQuery Compat 3.x.
- Если вам нет необходимости поддерживать Internet Explorer 8 и ниже, но нужно поддерживать старые версии Opera и Safari, используйте jQuery 2.x.
- Для мобильных приложений, разрабатываемых с помощью PhoneGap или аналогичных фреймворков, можно применять jQuery 3.x.
- Приложения Firefox OS и Chrome OS могут работать с jQuery 3.x.
- Сайты, работа которых обусловлена очень старыми плагинами, в зависимости от фактического кода плагинов, могут быть вынуждены обратиться к jQuery 1.x.

Таким образом, два фактора: где вы будете использовать библиотеку, и какие браузеры собираетесь поддерживать.

Другим источником путаницы может быть выбор между версиями: сжатой (также упоминающейся как *minified* — «минимизированный»), предназначенной для производственной стадии, и несжатой, применяемой в стадии разработки (см. сравнение на рис. 1.3). Преимущество минимизированной библиотеки — уменьшение размера, что приводит к экономии трафика для конечных пользователей. Это сжатие достигнуто благодаря уменьшению ненужных пробелов (*intendation* — «отступ»), удалению комментариев в коде, которые могут быть полезны разработчикам, но игнорируются движками JavaScript, и сокращением имен переменных (*obfuscation* — «обфускация»). Эти изменения образуют код, который тяжело читать и отлаживать (вот почему вам не следует использовать такую версию для стадии разработки), но который имеет меньший размер.

Несжатая версия

```
// Handle when the DOM is ready
ready: function( wait ) {

        // Abort if there are pending holds or we're already ready
        if ( wait === true ? --jQuery.readyWait : jQuery.isReady ) {
                return;
        }

        // Make sure body exists, at least, in case IE gets a little overzealous (ticket #5443).
        if ( !document.body ) {
                return setTimeout( jQuery.ready );
        }

        // Remember that the DOM is ready
        jQuery.isReady = true;

        // If a normal DOM Ready event fired, decrement, and wait if need be
        if ( wait !== true && --jQuery.readyWait > 0 ) {
                return;
        }

        // If there are functions bound, to execute
        readyList.resolveWith( document, [ jQuery ] );

        // Trigger any bound ready events
        if ( jQuery.fn.triggerHandler ) {
                jQuery( document ).triggerHandler( "ready" );
                jQuery( document ).off( "ready" );
        }
}
```

Сжатая версия

```
ready:function(a){if(a===!0?!--m.readyWait:!m.isReady){if(!y.body)return
setTimeout(m.ready);m.isReady=!0,a!==!0&&--m.readyWait>0||(H.resolveWith(y,
[m]),m.fn.triggerHandler&&(m(y).triggerHandler("ready"),m(y).off("ready")))}}
```

Рис. 1.3. Вверху фрагмент, взятый из исходного кода jQuery, показывающий формат несжатой версии. Внизу — тот же фрагмент, минимизированный для использования в производственной стадии

В книге мы будем использовать jQuery 1.x в качестве основы, чтобы вы могли протестировать свой код в самом широком диапазоне браузеров, но сначала выделим все различия, вносимые jQuery 3, так, чтобы ваши знания были современными и актуальными, насколько это возможно.

Выбор правильной версии jQuery важен, но мы еще упоминали разницу между размещением jQuery локально или с помощью CDN.

1.3.2. Улучшение производительности с помощью CDN

На сегодняшний день существует общая практика обслуживать файлы, такие как изображения и библиотеки, через *Content Delivery Network* (сеть доставки контента). CDN — это распределенная система серверов, созданная для предоставления контента с высокой доступностью и производительностью. Вероятно, вы знаете, что браузеры могут загружать фиксированный набор контента от хоста, обычно от четырех до восьми файлов одновременно. Поскольку файлы, обслуживающиеся с использованием CDN, предоставляются с другого хоста, можно ускорить процесс загрузки, увеличив количество загружаемых файлов в период времени. Кроме того, много современных браузеров задействуют CDN, поэтому очень высока вероятность, что нужная библиотека уже есть в кэше браузера пользователя. Применение CDN для загрузки jQuery не гарантирует лучшую производительность в каждой ситуации, так как здесь играет роль множество факторов. Наш совет — проверить, какая конфигурация наилучшим образом подходит для вашего конкретного случая.

В настоящее время есть несколько CDN, на которые можно полагаться для включения jQuery, но наиболее надежные — jQuery CDN (http://code.jquery.com), Google CDN (https://developers.google.com/speed/libraries/devguide) и Microsoft CDN (http://www.asp.net/ajaxlibrary/cdn.ashx).

Например, вы хотите обратиться к сжатой версии jQuery 1.11.3, применяя jQuery CDN. Это можно сделать, написав следующий код:

```
<script src="//code.jquery.com/jquery-1.11.3.min.js"></script>
```

Как вы могли заметить, он не указывает протокол для использования (либо HTTP, либо HTTPS). Вместо этого вы задаете тот же протокол, который задействуете на вашем сайте. Но учтите, что применение данного способа на странице, открытой не через веб-сервер, приведет к ошибке.

Однако применение CDN может быть сопряжено с затруднениями. Ни один сервер и ни одна сеть не функционируют в Интернете 100 % времени бесперебойно, и CDN не исключение. Если применять CDN для загрузки jQuery, то в редких случаях, когда сервер «упал» или недоступен, и в кэше браузера посетителя нет копии, код вашего сайта прекращает работу. Для критически важных приложений это может быть реальной проблемой. Чтобы предотвратить ее, можно принять простое и разумное решение, популярное у многих других разработчиков. Опять же вы хотите включить минимизированную версию jQuery 1.11.3, но сейчас используете это решение:

```
<script src="//code.jquery.com/jquery-1.11.3.min.js"></script>
<script>window.jQuery || document.write('<script src="javascript/jquery-
    1.11.3.min.js"><\/script>');</script>
```

Идея этого кода состоит в том, чтобы запросить копию библиотеки и проконтролировать, была ли она загружена, проверяя, определено ли свойство `jQuery` объекта `window`. Если проверка не пройдена, то вставляете код, загружающий локальную копию, которая в данном конкретном примере хранится в папке с именем `javascript`.

Если свойство `jQuery` присутствует, то можно смело применять методы jQuery без необходимости загружать копию, размещенную локально.

Вы проверяете наличие свойства `jQuery`, потому что, когда библиотека загружена, она добавляет это свойство. В нем можно найти все методы библиотеки. В процессе разработки мы рекомендуем использовать локальную копию jQuery во избежание каких-либо проблем с подключением.

Помимо свойства `jQuery`, вы также можете найти ярлык с именем `$`, который будете часто видеть в процессе работы и в нашей книге. Вам может показаться странным, но в JavaScript переменная или свойство с именем `$` разрешены. Мы назвали `$` ярлыком, так как по сути это тот же объект `jQuery`, что подтверждается таким оператором, взятым из исходного кода:

```
window.jQuery = window.$ = jQuery;
```

Теперь вы знаете, как включить библиотеку в ваши веб-страницы, но ничего не знаете о том, как она структурирована. Мы рассмотрим эту тему ниже.

1.4. Структура jQuery

Репозиторий jQuery (https://github.com/jquery/jquery), размещенный на GitHub, — прекрасный пример того, как разработка клиентской части изменилась за последние годы. Несмотря на то что это не связано напрямую с библиотекой самой по себе, всегда важно знать, как опытные разработчики организуют свой рабочий процесс и какие инструменты задействуют.

Если вы опытный разработчик клиентской части, то наверняка знаете о многих (если не обо всех) таких инструментах, но имеет смысл освежить в памяти эти знания. При разработке jQuery для клиентской части команда разработчиков использовала новейшие и наилучшие технологии, актуальные в данном секторе, а именно:

- *Node.js* (http://nodejs.org) — платформу, построенную на движке JavaScript для Chrome, позволяющем запускать JavaScript как язык серверной стороны;
- *npm* (https://npmjs.org) — официальный менеджер пакетов для Node.js, применяемый для установки пакетов, таких как Grunt, и его задач;
- *Grunt* (http://gruntjs.com) — средство запуска задач для автоматизации распространенных и повторяющихся операций, таких как сборка, тестирование и минимизация;
- *Git* (http://git-scm.com) — бесплатную распределенную систему управления версиями для отслеживания изменений в коде; позволяет простое взаимодействие между разработчиками.

С другой стороны, исходный код jQuery следует формату Asynchronous module definition, AMD (асинхронное определение модуля). Формат AMD представляет собой предложение для определения модулей, где и модуль, и его зависимости могут быть загружены асинхронно. На практике это означает, что, хотя вы используете jQuery как уникальный и единственный блок, ее исходный код разбит на несколько

файлов (модулей), как показано на рис. 1.4. Зависимости относительно таких файлов управляются с помощью менеджера зависимостей — в данном случае RequireJS.

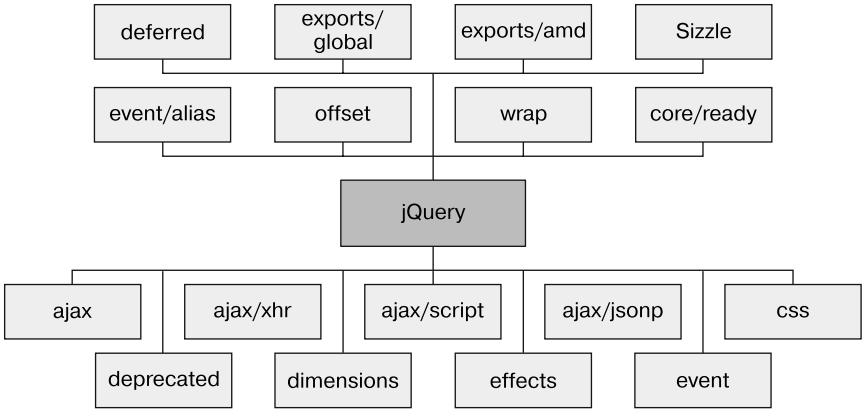


Рис. 1.4. Схема, представляющая модули jQuery: ajax, ajax/xhr, ajax/script, ajax/jsonp, css, deprecated, dimensions, effects, event, event/alias, offset, wrap, core/ready, deferred, exports/global, exports/amd и Sizzle

Чтобы дать вам представление о внутреннем наполнении модулей, приведем несколько примеров.

- `ajax` содержит все функции Ajax, такие как `ajax()`, `get()` и `post()`.
- `deprecated` содержит все устаревшие на данные момент методы, которые еще не удалены. Что внутри этого модуля — зависит от версии jQuery.
- `effects` содержит методы, позволяющие анимацию, такие как `animate()` и `slideUp()`.
- `event` содержит методы для добавления обработчиков событий к событиям браузера, такие как `on()` и `off()`.

Организация исходников в модули приводит к другому преимуществу: возможности построения пользовательской версии jQuery, содержащей только те модули, которые вам необходимы.

Сохраняем место, создавая собственную сборку. jQuery дает вам возможность создать собственную версию библиотеки, содержащую исключительно нужный вам функционал. Это позволит уменьшить вес библиотеки, что приведет к улучшению производительности, поскольку конечный пользователь должен будет загружать меньше килобайт.

Возможность избавляться от ненужных модулей очень важна. Вы можете думать, что вам понадобится весь функционал, предлагаемый jQuery, но вероятность применения всех его функций на одном и том же сайте очень низкая. Почему бы не убрать ненужные строки кода для улучшения производительности сайта?

Можно задействовать Grunt для создания пользовательской версии. Представьте, что вам нужна минимизированная версия jQuery 1.11.3 со всем ее функционалом (кроме устаревших методов и свойств) и эффектами. Для выполнения этой задачи нужно установить Node.js, Git и Grunt на своей локальной машине. Затем

клонируйте репозиторий jQuery, запустив следующую команду через интерфейс командной строки:

```
git clone git://github.com/jquery/jquery.git
```

После завершения клонирования введите две команды:

```
npm install
grunt custom:-deprecated,-effects
```

Вы закончили: внутри папки с именем `dist` вы найдете вашу пользовательскую сборку jQuery и в минимизированной, и в неминимизированной версиях.

Но у этого подхода есть и обратная сторона. Первая проблема возникает, когда выпускается новая версия jQuery. Вторая — когда новому функционалу вашего сайта нужна функция, содержащаяся в модуле, который не был включен прежде. В таких случаях нужно снова совершить описанные выше действия (обычно только команды) для создания новой пользовательской версии, включающей новые методы, исправления ошибок или пропущенные модули.

Теперь, когда вы знаете, как поместить библиотеку и создать пользовательскую сборку, пришло время погрузиться в основы jQuery.

1.5. Основы jQuery

По сути, jQuery фокусируется на извлечении элементов из HTML-страниц и выполнении операций с ними. Если вы знакомы с CSS, то уже хорошо осведомлены о возможностях селекторов, описывающих группы элементов по их типу, атрибутам, положению в пределах документа и многому другому. С jQuery вы можете применять эти знания, чтобы значительно упростить ваш JavaScript-код.

Библиотека уделяет огромное внимание последовательной работе кода во всех крупных браузерах; были решены многие из сложнейших проблем JavaScript которые ранее при столкновении с ними вынуждали пользователей этого языка испытывать беспокойство. Если вам кажется, что jQuery нужно дополнение, то у нее есть простой, но эффективный путь расширения своей функциональности с помощью плагинов (мы обсудим это подробно в главе 12).

Начнем с того, что посмотрим на объект `jQuery` сам по себе и на то, как можно использовать знания CSS для создания эффективного, но компактного кода.

1.5.1. Свойства, утилиты и методы

Как мы уже говорили ранее, библиотека раскрывается через свойство `jQuery` и ярлык `$`. С их помощью можно получить доступ к свойствам, методам и функциям, которые предоставляет jQuery.

Одним из свойств `jQuery` является `fx.off` (также именуемое флагом). Оно позволяет включать или отключать эффекты, выполняемые с применением методов jQuery. Мы подробно обсудим это и другие свойства (флаги) в главе 9.

Гораздо больший интерес представляют *утилиты*, также называемые *вспомогательными функциями*. Это наиболее часто используемые функции общего назначения,

которые включены в библиотеку. Можно сказать, что jQuery выступает для них в качестве *пространства имен*.

Чтобы дать вам о них общее представление, рассмотрим пример. Одной из доступных утилит является функция для обрезки строк. Она предназначена для удаления пробелов в начале и в конце строки. Ее вызов может выглядеть следующим образом:

```
var trimmed = $.trim(someString);
```

Если значение `someString` будет `" привет "`, то результатом вызова функции `$.trim()` будет `"привет"`. Как вы понимаете, в примере мы задействовали ярлык jQuery (`$`). Помните, что это такой же идентификатор, как и любой другой в JavaScript. Вызвать ту же функцию с использованием идентификатора `jQuery` вместо его псевдонима можно так:

```
var trimmed = jQuery.trim(someString);
```

Другим примером вспомогательной функции является `$.isArray()`. Как вы догадались, она проверяет, является ли данный аргумент массивом.

Помимо свойств и функций, библиотека также предоставляет методы, доступные после вызова функции `jQuery()`. Рассмотрим эту тему подробнее.

1.5.2. Объект jQuery

Первая функция, которую вы будете использовать при изучении jQuery, — `jQuery()`. Она принимает до двух аргументов и, в зависимости от их количества и типа, выполняет различные задачи.

Как и многие другие (почти все) методы в библиотеке, она позволяет построить *цепочку*. Построение цепочки является техникой программирования, применяемой для вызова нескольких методов в одном операторе. Вместо:

```
var obj = new Obj();
obj.method();
obj.anotherMethod();
obj.yetAnotherMethod();
```

можно написать:

```
var obj = new Obj();
obj.method().anotherMethod().yetAnotherMethod();
```

Наиболее распространенное использование функции `jQuery()` — выбор элементов из DOM для их дальнейшего изменения. В этом случае она принимает два параметра: селектор и (не обязательно) контекст. Функция возвращает объект, содержащий коллекцию элементов DOM, которые соответствуют заданным критериям. Но что такое селектор?

Когда для отделения дизайна от содержания в веб-технологии ввели CSS, было необходимо найти способ обращения к группам элементов страницы из внешних страниц стилей. Разработанный метод должен был использовать селекторы, которые сжато представляли собой элементы в зависимости от их типа,

атрибутов или положения в пределах HTML-документа. Те, кто знаком с XML, могут также знать о XPath (подробнее здесь: http://www.w3.org/TR/xpath20/) как о средстве для выбора элементов в пределах документа XML. Селекторы CSS представляют собой столь же эффективную концепцию, но настроены для работы на HTML-страницах, являются более краткими и, как правило, считаются более легкими для понимания.

jQuery использует те же селекторы CSS. Она поддерживает не только широко реализованные селекторы, принадлежащие CSS2.1, но и более эффективные, появившиеся в CSS3. Это важно, поскольку некоторые из них не могут быть в полной мере реализованы во всех браузерах или могут никогда не проявляться (например, в более старых версиях Internet Explorer). Ко всему прочему, у jQuery есть и свои собственные селекторы, и она позволяет создавать новые.

В книге вы сможете применить имеющиеся знания по CSS для быстрой наладки селекторов, а также узнаете о более продвинутых селекторах jQuery. Не беспокойтесь, если знаете о них немного. Мы рассмотрим их очень подробно в главе 2, и вы сможете найти полный список этих селекторов на сайте jQuery http://api.jquery.com/category/selectors/.

Предположим, вам необходимо выбрать все `<p>` на странице с помощью `jQuery()`. Для этого можно написать:

```
var paragraphs = jQuery('p');
```

Библиотека ищет совпадающие элементы в DOM, начиная с корня документа, так что процесс может быть медленным в связи с огромным количеством элементов.

В большинстве случаев можно ускорить поиск с помощью параметра `context`. Он используется для ограничения поиска одним поддеревом или несколькими, в зависимости от применяемого селектора. Чтобы лучше это понять, модифицируем предыдущий пример.

Предположим, вы хотите найти все `<p>`, которые содержатся в `<div>`. То, что они там *содержатся*, не означает, что `<div>` должен быть родителем `<p>`; он также может быть общим предком. Можно решить эту задачу так, как показано ниже:

```
var paragraphsInDiv = jQuery('p', 'div');
```

При обращении к псевдониму jQuery та же команда будет выглядеть следующим образом:

```
var paragraphsInDiv = $('p', 'div');
```

При использовании второго аргумента jQuery сначала собирает элементы, основанные на этом селекторе, который называется `context`, а затем извлекает потомков, соответствующих первому параметру, `selector`. Мы обсудим эту тему подробнее в главе 2.

Как мы уже говорили, функция `jQuery()` (и ее псевдоним `$()`) возвращает объект JavaScript, содержащий набор элементов DOM, соответствующих селектору, в порядке их определения в документе. Этот объект поддерживает большое количество полезных предопределенных методов, которые могут действовать на собранной группе элементов. Мы будем применять термины *«коллекция jQuery»*, *«объект jQuery»* или *«набор jQuery»* (или другие подобные выражения) для

обозначения этого возвращаемого объекта JavaScript, который содержит набор соответствующих элементов, способных работать с методами, определенными jQuery. Исходя из указанного определения, предыдущая переменная `paragraphsInDiv` представляет собой объект jQuery, содержащий все абзацы, являющиеся потомками элемента `div`. Вы будете часто прибегать к объектам jQuery, когда нужно будет совершать такие операции, как запуск определенной анимации или применение стиля, на нескольких элементах на странице.

Как уже упоминалось ранее, важной особенностью большого количества этих методов, часто нами называемых *методами jQuery*, является то, что они позволяют построить цепочку. После того как метод завершил свою работу, он возвращает группу элементов, с которыми происходило какое-то действие, готовых к уже другим действиям. Поскольку дальше все усложняется, способность jQuery построить цепочку поможет сократить строки кода, необходимого для обеспечения желаемого вами результата.

В подразделе 1.2.2 мы подчеркнули преимущества размещения кода JavaScript в нижней части страницы. На протяжении многих лет разработчики размещали сценарии на странице в заголовке `<head>`, опираясь на метод jQuery, который называется `ready()`. Такой подход в настоящее время не рекомендуется, но многие разработчики используют его до сих пор. Из следующего подраздела вы узнаете об этом больше, а также выясните, какой подход предлагается на сегодняшний день.

1.5.3. Обработчик готовности документа

Принцип ненавязчивого JavaScript позволяет отделить поведение элементов от структуры документов. Применяя его, вы совершаете операции с элементами страницы за пределами разметки документа, в котором создаются эти элементы. Здесь нам нужен механизм, позволяющий дождаться окончания загрузки элементов DOM страницы, прежде чем эти операции будут выполняться.

В примере о группе переключателей все тело должно быть загружено до применения поведения. Традиционно с этой целью используется обработчик `onload` для экземпляра `window`, выполняющий операторы после полной загрузки всей страницы. Синтаксис, как правило, выглядит так:

```
window.onload = function() {
  // здесь что-то делается
};
```

Это приводит к выполнению определенного кода *после* того, как документ полностью загрузится. К сожалению, браузер не только задерживает выполнение кода `onload` до момента создания полного дерева DOM, но также ждет, пока полностью загрузятся все внешние ресурсы и страница отобразится в окне браузера. Сюда входят и изображения, и встроенные в веб-страницы QuickTime и Flash-видео. В результате посетители могут заметить серьезную задержку между моментом, когда они увидят страницу впервые, и временем, когда будет выполнен сценарий `onload`.

Хуже того, если изображение или другой ресурс будет загружаться достаточно долго, то посетителям придется ждать окончания загрузки, прежде чем станет доступным поведение элементов. Во многих реальных случаях это могло бы свести на нет саму идею ненавязчивого JavaScript.

Куда лучшим подходом был бы такой: ждать только того, чтобы структура документа была полностью проверена и браузер преобразовал HTML в результирующее дерево DOM перед выполнением сценария, отвечающего за поведение. Добиться этого независимым от типа браузера способом достаточно сложно, но библиотека jQuery предоставляет простой механизм запуска программного кода сразу после загрузки дерева DOM (не дожидаясь внешних ресурсов).

Формальный синтаксис определения такого кода выглядит следующим образом:

```
jQuery(document).ready(function() {
  // Ваш код здесь...
});
```

Сначала вы оборачиваете (значение термина см. в подразделе 5.2.3) объект `document`, используя функцию `jQuery()`, а затем вызываете метод `ready()`, которому функция передастся для выполнения после того, как документ станет доступным для дальнейших манипуляций. Это означает, что внутри функции, переданной методу `ready()`, можно безопасно получить доступ ко всем элементам вашей страницы. Схема описанного механизма показана на рис. 1.5.

Мы назвали это *формальным синтаксисом* по причине того, что гораздо чаще используется сокращенная форма его записи:

```
jQuery(function() {
  // Ваш код здесь...
});
```

Передав функцию `jQuery()` или ее псевдоним `$()`, мы тем самым предписываем браузеру дождаться, пока дерево DOM (но только оно) полностью загрузится, прежде чем выполнить ваш код. Или, еще лучше, можно использовать эту технику несколько раз в пределах одного HTML-документа, а браузер будет выполнять все указанные функции в порядке их объявления в пределах страницы.

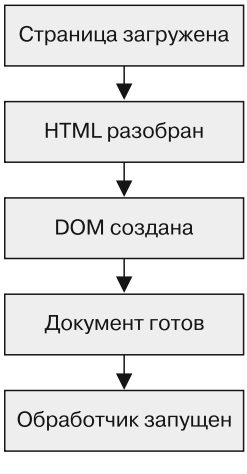


Рис. 1.5. Представление шагов, выполняемых браузерами перед запуском обработчика готовности документа

В противоположность этому методика на базе обработчика `onload` позволяет указать только одну функцию. Такое ограничение чревато трудноуловимыми ошибками, если какой-либо сторонний сценарий задействует механизм `onload` для своих собственных целей (что никак нельзя признать хорошей практикой).

Применение обработчика готовности документа — хороший способ овладеть техникой ненавязчивого JavaScript, но не обязательный, его можно избежать.

Поскольку метод `ready()` берет на себя выполнение кода после загрузки DOM, разработчики прежде размещали элементы `script` в `<head>` страницы. Как мы уже говорили в подразделе 1.2.2, можно разместить их непосредственно перед закрывающим тело тегом (`</body>`). Поступая таким образом, вы можете полностью избежать использования функции `$(document).ready()`, так как в этой точке все остальные элементы уже находятся в DOM. Таким образом, можно их безопасно получить и применить. Если хотите увидеть пример, как избежать данной функции, то посмотрите на исходный код файла `chapter-1/radio.group.html`.

В оставшейся части книги мы будем придерживаться этого способа, так что вам не придется обращаться к методу `ready()`.

1.6. Резюме

Мы рассмотрели много материала в нашем головокружительном введении в jQuery. Подведем итог: такое отделение операций в целом полезно для любой страницы, над которой нужно выполнить нечто более сложное, чем тривиальные команды JavaScript. Кроме того, мы фокусируемся на возможности использования авторами на своих страницах концепции ненавязчивого JavaScript. С таким подходом поведение отделяется от структуры так же, как от структуры отделяются стили CSS, и таким образом достигается лучшая организация страницы и повышается универсальность кода.

Несмотря на то что jQuery привносит только два имени в пространство имен JavaScript — функцию с тем же именем `jQuery` и ее псевдоним `$` — библиотека обеспечивает большую функциональность, делая эту функцию в высшей степени универсальной, регулируя выполняемую ею работу, основываясь на переданных ей параметрах.

Мы уже упоминали, как хорошо организованы репозиторий библиотеки и код в целом. Мы также уделили большое внимание доступным версиям библиотеки и различиям, чтобы вы смогли сделать сознательный выбор. Производительность — важный фактор, который необходимо рассмотреть, так что мы описали возможности сведения к минимуму расхода ресурсов при добавлении библиотеки к вашим страницам. Использование CDN и настройка модулей, которые вам понадобятся, — это отличный способ ускорить загрузку jQuery.

В последующих главах мы рассмотрим все функции, которые jQuery может предложить вам как веб-разработчику. Мы продолжим наше путешествие в следующей главе, и вы узнаете, как применять селекторы библиотеки для быстрой и простой идентификации элементов, с которыми вы захотите работать дальше.

Часть II
Основы jQuery

Прошло много лет с тех пор, как Джон Резиг представил миру jQuery. Чуть меньше, но все же немало лет прошло с тех пор, как jQuery представляла собой не более чем библиотеку для манипуляций с DOM. В течение этого времени она образовала вокруг себя целую «экосистему», состоящую из сопутствующих библиотек и других проектов. Представляем их.

- *jQuery UI* — библиотека, состоящая из набора взаимодействий пользовательского интерфейса, эффектов, виджетов и тем, помогающего создавать удивительные пользовательские интерфейсы.
- *jQuery Mobile* — система пользовательского интерфейса на основе HTML5 для всех популярных платформ мобильных устройств. Помогает создать красивый дизайн для мобильных устройств.
- *QUnit* — фреймворк для модульного тестирования JavaScript, используемый всеми другими проектами JQuery.
- *Plugins* — плагины, опубликованные на npm (https://www.npmjs.com/), и несчетное количество других плагинов, которые распространились по всему Интернету, созданы, чтобы покрыть случаи использования, не охваченные jQuery или чтобы улучшить ее функциональные возможности.

В части II мы рассмотрим основную библиотеку целиком. После ознакомления с главами части вы будете досконально разбираться в jQuery и сможете взяться за любой веб-проект, вооружившись одним из самых эффективных инструментов, доступных для клиентской стороны. Смело переворачивайте страницу, вникайте и будьте готовы узнать, как вдохнуть жизнь в ваши веб-приложения так, чтобы это было не только легко, но и интересно!

2 Выбор элементов

В этой главе:

- выбор элементов с использованием jQuery и селекторов CSS;
- открытие уникальных фильтров jQuery;
- разработка пользовательских фильтров;
- изучение параметра `context` функции `jQuery()`.

В этой главе мы подробно рассмотрим, как определяются элементы DOM, над которыми будут проводиться действия, благодаря одной из самых действенных и часто используемых возможностей функции jQuery `$()`: выбора элементов DOM с помощью *селекторов*. Изучив эту главу, вы ознакомитесь с множеством доступных селекторов. Библиотека не только обеспечивает полную поддержку всех селекторов CSS, но и вводит свои. Мы также познакомим вас с *фильтрами* (многие из них являются специальными селекторами, присущими только jQuery, и обычно работают с другими типами селекторов), чтобы в дальнейшем сократить набор соответствующих элементов. Ко всему прочему, вы узнаете, как создавать *пользовательские фильтры* (также известные как *пользовательские селекторы* или *пользовательские псевдоселекторы*) в случае, когда вашим страницам понадобится фильтр, не поддерживаемый библиотекой. Мы также обсудим `context`, второй параметр функции `$()`, и опишем последствия его использования.

Большое количество возможностей, необходимых для интерактивных веб-приложений, появляется за счет манипулирования элементами DOM, составляющими страницы. Но прежде, чем ими оперировать, необходимо их идентифицировать и отобрать. Эта и следующая главы познакомят вас с концепциями выбора элементов. В предыдущем издании «jQuery в действии» им посвящалась одна глава, поскольку их содержание весьма связано между собой. Мы решили разделить их, чтобы помочь вам усвоить огромное количество информации. Но, несмотря на такое разделение, глава 2 все еще довольно большая, хоть и сжатая. Возможно, для полного овладения всеми описанными концепциями потребуется прочитать ее несколько раз. Итак, начнем наше путешествие по детальному изучению множества способов, с помощью которых библиотека позволяет определять элементы, подлежащие выбору для выполнения манипуляций.

2.1. Выбор элементов для манипуляции

Первое, что необходимо сделать перед использованием практически всех методов jQuery, — выбрать некоторые элементы документа для совершения соответствующих действий. Как вы уже знаете из главы 1, чтобы выбрать элементы на странице с помощью jQuery, нужно передать селектор функции `jQuery()` (или ее псевдониму `$()`). Они возвращают объект jQuery, содержащий набор из элементов DOM, которые соответствуют заданным критериям, а также выставляют многие методы и свойства jQuery.

Иногда набор необходимых для выбора элементов описать легко, например: «все элементы абзацев на странице». Но часто они требуют более сложного описания, такого как «все элементы списков, которые имеют класс `list-element`, содержат ссылку и являются первыми в списке». К счастью, jQuery предоставляет надежный синтаксис селекторов, который позволяет кратко и элегантно определить любой набор элементов. Наверняка вы многое знаете о синтаксисе. Библиотека использует уже знакомый и любимый вами синтаксис CSS и расширяет его пользовательскими средствами для выполнения как обычных, так и сложных выборов.

Чтобы помочь вам изучить принципы выбора элементов, мы включили в состав загружаемого пакета с примерами для нашей книги страницу jQuery Selectors Lab (доступен в файле `chapter-2/lab.selectors.html`). Она позволяет вводить строки селекторов jQuery и наблюдать (в режиме реального времени!), какие элементы DOM они отбирают. В окне браузера страница должна выглядеть так, как показано на рис. 2.1.

СОВЕТ

> Если вы еще не загрузили пример кода, то сейчас самое время. Вам будет гораздо проще усвоить информацию этой главы, выполняя лабораторные работы. Посетите веб-страницу книги по адресу: https://www.manning.com/books/jquery-in-action-third-edition, чтобы найти ссылку для загрузки, или перейдите на https://github.com/AurelioDeRosa/jquery-in-action.

Область Selector Panel (Панель селектора) в верхнем левом углу содержит текстовое поле и кнопку. Чтобы активизировать лабораторную работу Experiment, введите селектор в текстовое поле и нажмите кнопку Apply (Применить). Далее введите строку `li` в поле и снова нажмите ту же кнопку.

Введенный селектор (в данном случае `li`) применяется к HTML-странице, загруженной на панели DOM Sample (Пример DOM), расположенной в правом верхнем углу. Лабораторный код, который выполняется, когда вы нажимаете кнопку Apply (Применить), добавляет класс с именем `found-element` для всех совпадающих элементов. Объявление CSS, определенное для страницы, выделяет все элементы с этим классом с помощью черной рамки и серого фона. После нажатия кнопки Apply (Применить) вы должны увидеть в окне браузера изображение (рис. 2.2), где

выделены все элементы `li` в примере DOM. Ко всему прочему, под текстовым полем селектора будет показана выполненная команда jQuery вместе с названиями тегов выбранных элементов. HTML-разметка, используемая для визуализации фрагмента образца DOM, отображается на нижней панели DOM Sample Code (Пример кода DOM). Это должно помочь вам экспериментировать с написанием селекторов, ориентированных на элементы в нашем образце.

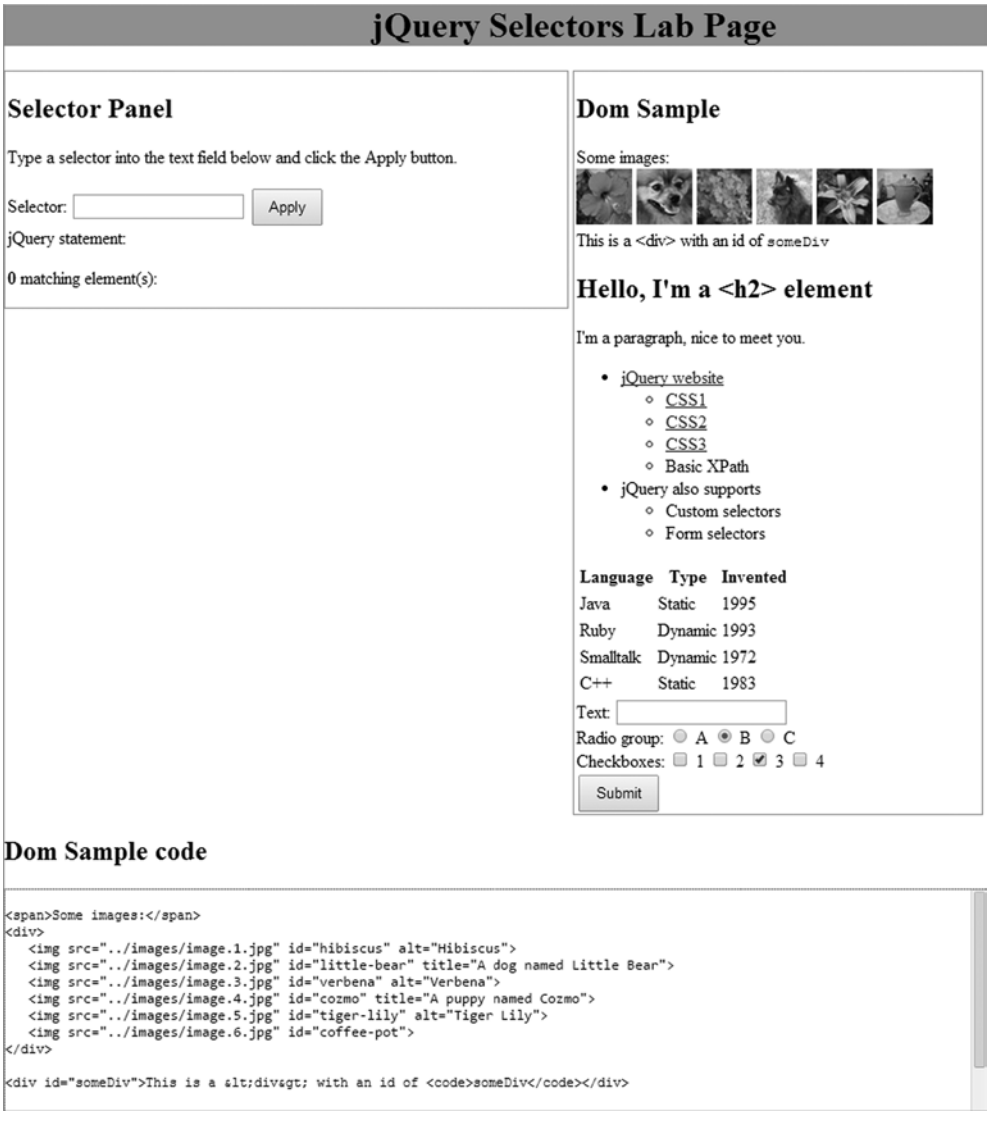


Рис. 2.1. Страница jQuery Selectors Lab позволяет вам видеть поведение любого выбранного вами селектора в режиме реального времени

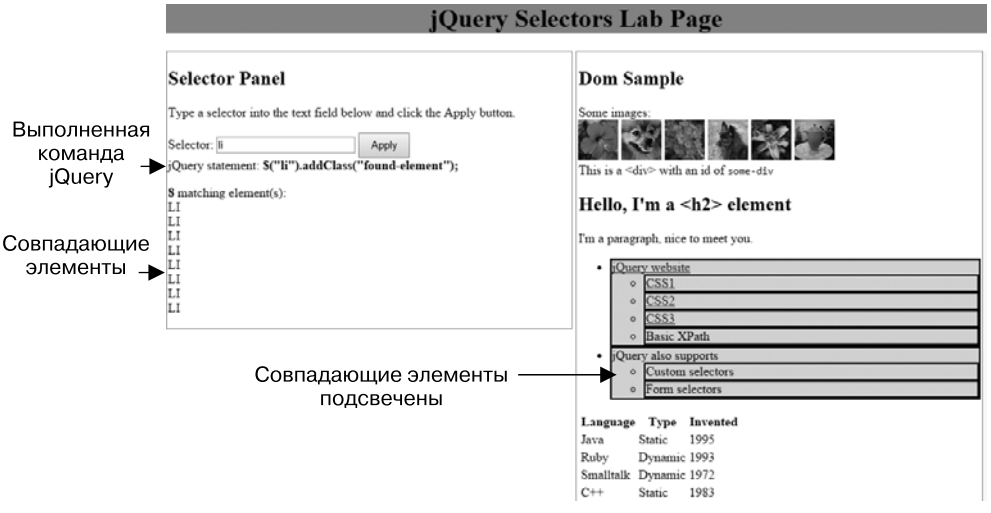


Рис. 2.2. Значение селектора li соответствует всем элементам li, как показано в отображенных результатах

Мы еще вернемся к этой странице. А сейчас посмотрим, каким образом jQuery взаимодействует с базовыми селекторами CSS.

2.2. Базовые селекторы

Для применения стилей к элементам страницы веб-разработчики используют несколько эффективных и удобных выражений для выбора, работающих во всех типах браузеров. Эти выражения могут выбирать по идентификатору элемента (по атрибуту ID), именам класса CSS и по именам тега. Особым методом выбора элементов по имени тега является универсальный селектор, позволяющий выбрать все элементы страницы в пределах DOM. Выражения выбора дают возможность выполнять основные запросы в DOM (детальнее разберем ниже). При их комбинации эти селекторы позволяют вам легче решить более сложные задачи. В табл. 2.1 приведены примеры таких селекторов и способы их объединения.

Таблица 2.1. Некоторые примеры простых селекторов CSS

Пример	Описание	В CSS
*	Соответствует всем элементам на странице	+
#special-id	Соответствует элементу со значением ID special-id	+
.special-class	Соответствует элементу с классом special-class	+
a	Соответствует всем элементам ссылок (a)	+
a.special-class	Соответствует всем элементам ссылок (a), у которых есть класс special-class	+
.class.special-class	Соответствует всем элементам с классом class и классом special-class	+

В JavaScript предусмотрен набор функций, таких как `getElementById()` и `getElementsByClassName()`, предназначенных для работы с определенным типом селектора для получения элементов DOM, над которыми будут проводиться действия. К сожалению, могут возникнуть отдельные проблемы с использованием даже таких простых функций. Например, `getElementsByClassName()` не поддерживается Internet Explorer до версии 9. Если вы хотите применить только нативные методы, то следует обратить внимание на кросс-браузерные сочетания.

jQuery спешит на помощь! Если браузер изначально поддерживает селектор или функцию, то библиотека для большей эффективности будет полагаться на него; в противном случае она использует свои методы, чтобы достичь ожидаемого результата. Хорошая новость: не придется беспокоиться об этой разнице. jQuery сделает свою работу для вас «за кулисами», так что можно сосредоточиться на других аспектах кода.

Библиотека полностью совместима со спецификацией CSS3, поэтому операция выбора элементов не содержит в себе сюрпризов — движок селекторов jQuery отберет те же элементы, которые могли быть отобраны реализацией таблиц стилей в совместимых браузерах. Библиотека *не* зависит от реализации CSS в браузере, в котором она работает. Даже если в браузере нет корректной реализации селекторов CSS, то jQuery все равно будет корректно отбирать элементы в соответствии с правилами, установленными стандартами World Wide Web Consortium (W3C).

Поэкспериментируйте с различными базовыми селекторами CSS на странице Selectors Lab, чтобы напрактиковаться.

Еще одна хорошая новость состоит в том, что jQuery решит для нас все вопросы кросс-браузерности (для поддерживаемых браузеров), поэтому теперь можно углубиться в изучение множества доступных селекторов.

2.2.1. Универсальный селектор

Первый из доступных селекторов — универсальный, который отображается в виде звездочки (`*`). Как следует из названия, он позволяет получить все элементы DOM веб-страницы, даже элемент `head` и его потомков. Для подкрепления этой концепции предположим, что у вас есть следующая HTML-страница:

```
<!DOCTYPE html>
<html>
  <head>
    <title>jQuery в действии, 3-е издание</title>
  </head>
  <body>
    <p>Я абзац</p>
  </body>
</html>
```

Чтобы получить все элементы страницы, вам следует использовать универсальный селектор и передать его функции `jQuery()` (или ее псевдониму `$()`) в объявлении таким образом:

```
var allElements = $('*');
```

Прежде чем двигаться дальше, хочется упомянуть общепринятые соглашения. При сохранении результата выбора, совершенного с помощью jQuery в переменной, полагается добавить перед именем переменной или (реже) в конце его знак доллара. Это действие не несет особого значения, а всего лишь служит в качестве напоминания о том, какая переменная сохраняется. Оно также применяется для того, чтобы не ссылаться на знак `$()` в множестве элементов DOM, на которых мы уже назвали этот метод. Например, вы можете ошибочно записать следующее:

```
var allElements = $('*');
// Остальной код здесь...

$(allElements);
```

Используя вышеупомянутые соглашения, можно переписать предыдущую команду, добавив знак доллара перед именем переменной, как показано здесь:

```
var $allElements = $('*');
```

Либо вы также можете написать это следующим образом:

```
var allElements$ = $('*');
```

Мы рекомендуем принять одну из этих конвенций и придерживаться ее. В книге мы будем приписывать знак доллара в начале.

Теперь посмотрим на первый полный пример применения jQuery на вебстранице (листинг 2.1). Здесь мы используем CDN, чтобы включить jQuery, задействуя запасную технику, описанную в главе 1, и универсальный селектор для выбора всех элементов страницы. Код этого листинга доступен в файле `chapter-2/listing-2.1.html`, предоставленном с книгой. В оставшейся части книги примеры будут включать лишь ссылку на локальную версию библиотеки jQuery, не используя CDN. Такой выбор сделан по двум основным причинам: для краткости (то есть доведется писать меньше кода) и во избежание дополнительного HTTP-запроса (который не будет работать, если вы запускаете примеры в автономном режиме).

Листинг 2.1. Использование универсального селектора в jQuery

```
<!DOCTYPE html>
<html>
   <head>
      <title>jQuery в действии, 3-е издание</title>
   </head>
<body>
      <p>Я абзац</p>
      <script src="//code.jquery.com/jquery-1.11.3.min.js"></script>
      <script>
         window.jQuery || document.write('<script src="../js/jquery-
                          1.11.3.min.js"><\/script>');
         var $allElements = $('*');
      </script>
   </body>
</html>
```

Запрос jQuery из jQuery CDN

Возврат к локальной копии, если CDN недоступен

Выбор всех элементов на странице

Как мы уже упоминали, предыдущий листинг был создан с целью выбора всех элементов страницы, но какие это элементы? Если вы проинспектируете переменную с помощью отладчика или консоли (при условии их наличия), то увидите, что это `html`, `head`, `title`, `body`, `p`, `script` (первый на странице) и `script` (второй на странице).

ВНИМАНИЕ

Мы хотим подчеркнуть, что метод console.log() не поддерживается старыми версиями Internet Explorer (IE 6–7). В приведенных примерах мы не будем обращать на это внимания, а станем использовать этот метод в значительной степени вместо весьма раздражающего метода window.alert(). Но вам следует помнить об отсутствии такой поддержки в случае, если ваш код должен работать в этих браузерах.

Помните, что элементы извлекаются и хранятся в том же порядке, в котором они появляются на странице.

Инструменты разработчика

Попытка разработать DOM-сценарий приложения без помощи инструментов отладки подобна попытке сыграть на концертном пианино в перчатках для сварки. Зачем же поступать так?

В зависимости от используемого браузера существуют различные варианты, которые вы можете выбрать, чтобы проверить код. Во всех основных современных браузерах есть набор встроенных инструментов для этой цели, хотя и с другим именем, нежели вы можете принять. Например, в Chrome такие инструменты называются *Chrome Developer Tools* (https://developer.chrome.com/devtools), в то время как в Internet Explorer они носят имя *F12 developer tools* (http://msdn.microsoft.com/en-us/library/bg182326(v=vs.85).aspx). У Firefox также есть свои собственные встроенные инструменты, но разработчики обычно прибегают к плагину *Firebug* (http://getfirebug.com). Эти инструменты позволяют не только инспектировать консоль JavaScript, но и просматривать актуальный DOM, CSS, сценарии и многое другое.

Как вы уже успели заметить, применение универсального селектора вынуждает jQuery пройти через все узлы DOM. Если элементов в DOM много, то данный процесс может быть очень медленным; поэтому его использование не рекомендуется. Кроме того, вряд ли вам необходимо получить все элементы страницы, хотя может понадобиться собрать принадлежащие к определенному поддереву DOM, как вы увидите позже.

Если вы когда-либо экспериментировали с JavaScript и браузером, то знаете, что один из наиболее популярных выборов осуществляется с помощью ID данного элемента. Поговорим об этом подробнее.

2.2.2. Селектор ID

Селектор ID является одним из наиболее часто используемых селекторов не только в jQuery, но и в обычном JavaScript. В последнем случае, чтобы выбрать элемент по его ID, вы передаете идентификатор нативной функции `document.getElementById()`. Если у вас есть некоторые знания CSS, то вы вспомните, что селектор ID обозначается знаком решетки (#) (в ряде стран известен под другим именем: *знак номера* или *знак фунта*), который ставится перед ID элемента. Если возникает такой пункт на вашей странице:

```
<p id="description">jQuery в действии – это книга о jQuery</p>
```

то можно получить его с помощью селектора ID и jQuery, написав следующее:

```
$('#description');
```

При использовании вместе с селектором ID jQuery возвращает коллекцию из нуля или одного элемента DOM. Если у вас есть более одного элемента на странице с одинаковым ID, то библиотека получает только первый встретившийся совпавший элемент. Несмотря на то что у вас может быть более одного элемента с одинаковым ID, это будет считаться ошибкой и вам так делать не следует.

ПРИМЕЧАНИЕ

Спецификации W3C по HTML5 утверждают, что значение идентификатора «не должно содержать пробелов. Нет никаких других ограничений на формирование идентификатора; в частности, идентификаторы могут состоять только из цифр, начинаться с цифры, подчеркивания, знака препинания и т. д.». Помимо этого, можно использовать символы, такие как точка (.), которые имеют особое значение в CSS и jQuery (потому что она следует соглашениям CSS). Из-за этого их нужно скрывать, предваряя два обратных слеша особым символом. Таким образом, если вы хотите выбрать элемент с идентификатором .description, то следует написать $('#\\.description').

Не случайно в начале главы мы сравнивали выбор элементов по их ID в jQuery и JavaScript с помощью функции `getElementById()`. В библиотеке выбор по ID является самым быстрым, независимо от используемого браузера, так как она задействует данную функцию «за кулисами» и это происходит очень быстро.

Применяя селектор идентификатора, вы сможете быстро получить один элемент в DOM. Часто бывает необходимо извлечь элементы на основе имен используемых классов. Как можно выбрать элементы с одним и тем же стилем?

2.2.3. Селектор класса

К селектору класса прибегают для извлечения элементов по именам используемых классов CSS. Как разработчик JavaScript, вы должны быть знакомы с этим видом выбора в рамках применения нативной функции `getElementsByClassName()`. jQuery следует конвенциям CSS, так что вам следует ставить точку перед вы-

бранным именем класса. Например, если у вас есть следующий HTML-код внутри страницы <body>:

```
<div>
  <h1 class="green">Заголовок</h1>
  <p class="description">Я абзац</p>
</div>
<div>
  <h1 class="green">Другой заголовок</h1>
  <p class="description blue">Я — еще один абзац</p>
</div>
```

и вы хотите выбрать элементы с классом description, то вам необходимо передать функцию .description в функцию $(), написав следующую команду:

```
var $descriptions = $('.description');
```

Результатом является объект, который часто в документации называют *объектом jQuery* или *коллекцией jQuery* (вы можете встретить и другие имена, такие как *множество соответствующих элементов*, просто *набор* или *коллекция*), содержащий два абзаца из HTML. Библиотека также выберет узлы, имеющие несколько классов, где один из них соответствует указанному имени (например, второй абзац).

В jQuery, как и в CSS, также можно объединить несколько селекторов имен класса. Если вы хотите выбрать все элементы, имеющие класс description и blue, то можете соединить их, в результате получив выражение $('.description.blue').

Селектор класса, безусловно, является одним из наиболее часто используемых в JavaScript и CSS, но существует еще один базовый селектор, который нам все еще нужно рассмотреть.

2.2.4. Селектор элементов

Этот селектор позволяет отобрать элементы на основе имени их тега. Благодаря его поддержке практически в любом браузере (включая IE6) jQuery применяет функцию getElementsByTagName() для выбора элементов по имени тега «за кулисами». Чтобы понять, какой вид выбора можно совершить с помощью селектора, предположим, что вам нужны все элементы на странице. Для выполнения этой задачи вы должны написать следующее:

```
var $divs = $('div');
```

Часто данный селектор используют в сочетании с другими, ведь, как правило, на страницах встречается много элементов одного и того же типа. В таких случаях его следует писать *перед* другими селекторами. Поэтому, если вам нужны все , которые имеют класс clearfix, необходимо написать так:

```
var $clearfixDivs = $('div.clearfix');
```

Селектор элементов также можно комбинировать с селектором ID, но мы настоятельно рекомендуем вам этого не делать по двум причинам: производительность и полезность. Задействуя сложный селектор, jQuery будет выполнять поиск, опираясь на свои собственные методы, как правило, избегая применения встроенных

функций, что приводит к более медленному совершению операции. Кроме того, как указывалось в подразделе 2.2.2, jQuery будет получать первый (если таковые имеются) элемент, имеющий искомый ID. Так что, если вы ищете только один элемент, нет необходимости усложнять ваш селектор, смешивая два типа.

jQuery также позволяет использовать различные типы в одном выборе, обеспечивая высокую производительность, так как DOM проходится только один раз. Для включения в код селектора элементов следует добавлять запятую после предпоследнего селектора (пробелы после запятой игнорируются, поэтому их применение является вопросом стиля кода). Чтобы выбрать все элементы `<div>` и `<span>` на странице, можно написать:

```
$('div, span');
```

В случае если данный элемент соответствует более чем одному из разделенных запятыми селекторов (что невозможно при использовании только селектора элементов, так как, к примеру, элементом является `div` или `span`), то библиотека получит его только один раз, удаляя все повторения.

Теперь благодаря описанным селекторам можно выполнить базовый поиск в DOM. Но часто бывает необходимо выбирать элементы, применяя более сложные критерии. Вам может понадобиться извлечь узлы DOM на основе их связи с другими узлами, такими как «все ссылки внутри непронумерованного списка». Здесь вам необходимо указать выбор в соответствии с иерархией элементов. Как произвести такой поиск — обсудим в следующем разделе.

2.3. Получение элементов по их иерархии

Получение набора элементов по имени их класса — хорошее свойство, но часто нет желания осуществлять поиск по всей странице. Иногда бывает нужно выбрать только прямых потомков определенного элемента. Рассмотрим следующий HTML-фрагмент из образца DOM на странице Selectors Lab:

```
<ul class="my-list">
  <li>
    <a href="http://jquery.com">jQuery поддерживает</a>
    <ul>
      <li><a href="css1">CSS1</a></li>
      <li><a href="css2">CSS2</a></li>
      <li><a href="css3">CSS3</a></li>
      <li>Базовый XPath</li>
    </ul>
  </li>
  <li>jQuery также поддерживает
    <ul>
      <li>Пользовательские селекторы</li>
      <li>Селекторы форм</li>
    </ul>
  </li>
</ul>
```

Предположим, вы хотите отобрать элемент, указывающий на сайт jQuery, но не на различные местные страницы, которые описывают различные спецификации CSS. Это можно сделать с помощью *селектора потомка*, в котором родитель и его прямой потомок разделены символом угловой скобки (>). Можно написать следующее:

```
ul.my-list > li > a
```

Селектор будет отбирать только ссылки, являющиеся прямыми потомками элементов списка, которые, в свою очередь, также являются прямыми потомками <ul>, входящими в класс my-list. Содержащиеся в подсписках ссылки исключены, так как у элемента ul, являющегося их родителем, нет класса my-list. Запуск данного селектора на странице лабораторной работы дает результат, показанный на рис. 2.3.

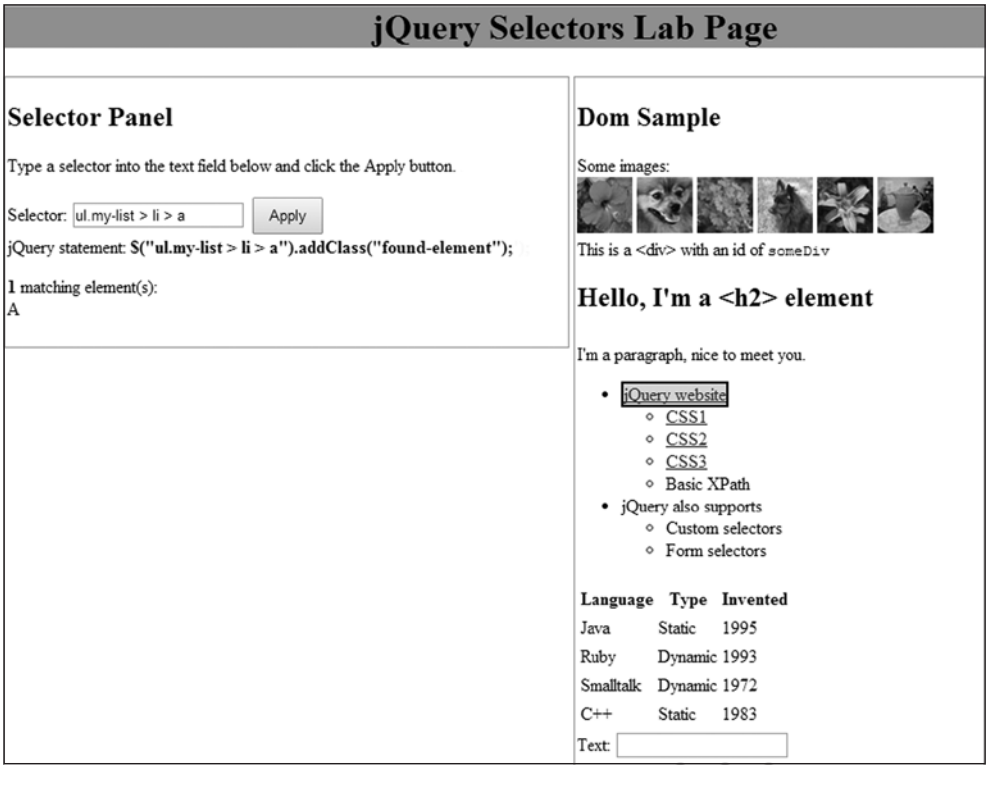


Рис. 2.3. С селектором ul.my-list > li > a выбираются только прямые потомки родительского узла

Выражать отношения между двумя или более элементами, основанными на иерархии дерева DOM, можно с помощью не только селектора потомка. В табл. 2.2 представлен обзор селекторов этого типа.

Таблица 2.2. Селекторы иерархии CSS, поддерживаемые jQuery

Селектор	Описание	В SCC
E F	Соответствует всем элементам с именем тега F, которые являются потомками E	+
E>F	Соответствует всем элементам с именем тега F, которые являются прямыми потомками E	+
E+F	Соответствует всем элементам с именем тега F, непосредственно перед которыми на этом же уровне идут E	+
E~F	Соответствует всем элементам с именем тега F, которым на этом же уровне предшествуют E	+

Все селекторы, описанные в таблице, кроме первого, являются частью спецификации CSS2.1, поэтому не поддерживаются Internet Explorer 6. Но вы можете безопасно использовать их все в jQuery, так как библиотека решает для вас такие виды задач.

Эти селекторы улучшили вашу способность точно ориентировать DOM-узлы, с которыми следует взаимодействовать. Со временем было создано много других CSS-селекторов, чтобы дать разработчикам больше свободы. Одной из новых особенностей стала возможность выбора элементов на основе их атрибутов. Об этих селекторах мы расскажем в следующем разделе.

2.4. Выбор элементов с помощью атрибутов

Селекторы атрибутов являются чрезвычайно продуктивными и позволяют выбирать элементы на основе их атрибутов. Такие селекторы легко распознать по их квадратным скобкам (например, `[селектор]`).

Чтобы увидеть их в действии, еще раз взглянем на фрагмент лабораторной страницы:

```
<ul>
  <li>
    <a href="http://jquery.com">jQuery поддерживает</a>
    <ul>
      <li><a href="css1">CSS1</a></li>
      <li><a href="css2">CSS2</a></li>
      <li><a href="css3">CSS3</a></li>
      <li>Базовый XPath</li>
    </ul>
  </li>
</ul>
```

Ссылка на внешний сайт отличается от других написанием `http://` в начале значения строки для атрибута ссылки `href`. На самом деле у внешних ссылок также могут быть приставки `https://`, `ftp://` и множество других протоколов. Кроме того, ссылка, указывающая на страницы того же сайта, все еще может начинаться

с `http://`. Но для упрощения мы будем принимать во внимание только `http://` и делать вид, что внутренние ссылки используют только относительные пути.

В CSS можно выбрать ссылки, имеющие значение `href`, которое начинается с `http://` с таким селектором:

```
a[href^='http://']
```

Применяя jQuery, последние могут быть написаны следующим образом:

```
var $externalLinks = $("a[href^='http://']");
```

Запись соответствует всем ссылкам со значением `href`, начиная с точной строки `http://`. Знак вставки (`^`) служит для указания соответствия в начале значения. Поскольку этот символ используется большинством процессоров регулярных выражений для обозначения соответствия в начале строки кандидата, его очень легко запомнить.

Снова зайдите на лабораторную страницу (из которой был взят предыдущий фрагмент HTML), введите `a[href^='http://']` в текстовое поле и нажмите кнопку Apply (Применить). Обратите внимание, что подсвечивается только ссылка jQuery.

Одинарные и двойные кавычки

Обращайте внимание на одинарные и двойные кавычки, когда прибегаете к селекторам атрибутов. Неправильная комбинация последних приведет к неработоспособной команде. Если ваш стиль кода предполагает использование двойных кавычек для строк и вы хотите добавить те же кавычки для оборачивания значения атрибутов, то вы должны их экранировать. Если же вам проще читать выборку без экранированных кавычек, то можете выбирать кавычки разных типов. Запуск селектора `a[href^="http://"]` вызывается следующими равнозначными командами:

```
$("a[href^=\"http://\"]");
$('a[href^=\'http://\']');
$("a[href^='http://']");
$('a[href^="http://"]');
```

А теперь представьте, что вам нужны все ссылки, кроме указывающих на главную страницу сайта jQuery. Используя нашу прекрасную библиотеку, можно написать:

```
$("a[href!='http://jquery.com']")
```

Это объявление с применением селектора «не равно атрибуту» дает ожидаемый результат. Поскольку данный селектор не является частью спецификации CSS, jQuery не может воспользоваться «за кулисами» нативным методом `querySelectorAll()`, что приводит к более медленному выполнению задачи.

Эти символы нельзя объединить с другими, чтобы создать еще более эффективный селектор. Например, если вы хотите выбрать все ссылки, кроме внешних (принимая во внимание только те, которые начинаются с `http://`), то не сможете написать:

```
$("a[href!^='http://']");
```

Сейчас вы можете подумать, что выбор элементов по их атрибуту возможен только с помощью селектора элементов. Но это не так. Вы можете использовать любой селектор, который вам нравится, или вообще их не задействовать, в результате чего получится селектор `[href^='http://']`. В таком случае неявно предполагается применение универсального селектора (`*`).

Есть и другие способы использования селекторов атрибутов. Чтобы соответствовать элементу — например, `form`, — обладающему особым атрибутом, независимо от его значения, можно обратиться к:

```
form[method]
```

Это соответствует любой `<form>`, у которой есть явный атрибут `method`.

Чтобы соответствовать определенному значению атрибута, попробуйте нечто похожее на:

```
input[type='text']
```

Данный селектор соответствует всем элементам `input` с атрибутом `type`, равным `text`.

Вы уже успели увидеть селектор «соответствие атрибуту в начале» в действии. Вот еще один пример:

```
div[title^='my']
```

Выбираются все элементы `div` с атрибутом `title`, значение которого начинается с `my`.

А как насчет селектора «атрибут заканчивается на»? Подходит следующее:

```
a[href$='.pdf']
```

Это полезный селектор для нахождения всех ссылок с указанием на PDF-файлы.

А вот селектор, который называется «атрибут содержит» для размещения элементов, чьи атрибуты содержат произвольные строки в любом месте значения атрибута:

```
a[href*='jquery.com']
```

Как вы и предположили, этот селектор соответствует всем элементам, которые ссылаются на сайт jQuery.

Следующий селектор — «содержит префикс». Он выбирает элементы с заданным значением атрибута, равным указанной строке или заданной строке с последующим дефисом. Если вы напишете:

```
div[class|='main']
```

то он найдет все `<div>`, имеющие `class="main"` или имя класса которых начинается с `main-`, как, например, `class="main-footer"`.

Последний селектор, который мы обсудим, похож на предыдущий, за исключением того, что служит для поиска слова в пределах значения атрибута. Допустим,

вы используете атрибут *data HTML5 — например, data-technologies — для указания списка значений в некоторых вашей страницы. Вам необходимо выполнить поиск, чтобы узнать, содержит ли один из них значение javascript. Можно произвести выбор, применяя следующий селектор:

```
span[data-technologies~="javascript"]
```

Этот селектор выбирает теги , которые содержат атрибут, подобный data-technologies="javascript", но также и data-technologies="jquery javascript qunit". Можете воспринимать это как соответствие селектору класса, но для обобщенного атрибута.

Представленные селекторы также могут быть соединены в цепочку, если нужно получить узлы, соответствующие нескольким критериям. Вы можете объединить столько селекторов, сколько посчитаете нужным; не существует фиксированного предела. Например, можно написать:

```
input[type="text"][required]
```

Этот селектор извлекает все необходимые теги <input> (атрибут required был введен в HTML5), которые имеют тип text.

Библиотека поддерживает ряд селекторов CSS, имеющих дело с атрибутами (табл. 2.3).

Таблица 2.3. Селекторы атрибутов, поддерживаемые jQuery

Селектор	Описание	В CSS
E[A]	Соответствует всем элементам с именем тега E, у которых есть атрибут A с любым значением	+
E[A='V']	Соответствует всем элементам с именем тега E, у которых есть атрибут A со значением в точности V	+
E[A^='V']	Соответствует всем элементам с именем тега E, у которых есть атрибут A, чье значение начинается на V	+
E[A$='V']	Соответствует всем элементам с именем тега E, у которых есть атрибут A, чье значение заканчивается на V	+
E[A!='V']	Соответствует всем элементам с именем тега E, у которых есть атрибут A, не соответствующий V (не равный V), или вообще отсутствует атрибут A	–
E[A*='V']	Соответствует всем элементам с именем тега E, у которых есть атрибут A, чье значение содержит V	+
E[A|='V']	Соответствует всем элементам с именем тега E, у которых есть атрибут A, чье значение равно V или V- (V с последующим дефисом)	+
E[A~='V']	Соответствует всем элементам с именем тега E, у которых есть атрибут A, чье значение равно V или содержит V, разделенные пробелами	+
E[C1][C2]	Соответствует всем элементам с именем тега E, у которых есть атрибуты, соответствующие критериям C1 и C2	+

Вооружившись этими знаниями, отправляйтесь к странице jQuery Selectors Lab, чтобы провести там еще некоторое время, тренируясь в использовании селекторов различных типов из табл. 2.3. Попробуйте сделать несколько целевых выборов с применением элемента `input` с типом `checkbox` и значением `1` (подсказка: для получения результата вам понадобится комбинация селекторов).

К селекторам прибегают не только для получения элементов с помощью функции `$()`. Как вы узнаете далее, они являются одним из наиболее часто используемых параметров для передачи методам jQuery. Например, совершив выбор, можно задействовать метод jQuery и новый селектор для добавления новых элементов к предыдущему выбору или отфильтровать некоторые элементы. Другой пример — поиск всех потомков элементов в ранее сохраненном наборе, который соответствует заданному селектору.

Поскольку мощности селекторов, которые мы обсуждали, недостаточно, есть несколько других вариантов, позволяющих еще лучше разобраться в DOM. В следующем разделе мы познакомим вас с другими типами селекторов, известными как *фильтры*. В спецификации CSS эти типы селекторов относятся к *псевдоклассам*.

2.5. Введение в фильтры

Фильтры — это селекторы, как правило работающие с другими типами селекторов, чтобы уменьшить набор совпавших элементов. Их легко узнать по двоеточию (`:`), с которого они обычно начинаются. Так же как и в случае с атрибутами, неявно предполагается использование универсального селектора, если не указан другой. Одна из особенностей этих селекторов состоит в том, что некоторые из них принимают дополнительный аргумент в скобках, например `p:nth-child(2)`. Ниже мы рассмотрим все доступные фильтры jQuery, разбив их на различные категории.

2.5.1. Фильтры позиции

Иногда вам необходимо выбрать элементы по их положению на странице, например первую или последнюю ссылку на странице или из третьего абзаца. jQuery поддерживает механизмы для осуществления таких конкретных выборов.

Например, рассмотрим:

```
a:first
```

Этот формат селектора совпадает с первым `<a>` на странице. Теперь предположим, что вы хотите получить ссылки, начиная с третьей на странице. Можно записать следующее:

```
a:gt(1)
```

Данный селектор действительно представляет интерес, поскольку позволяет обсудить несколько моментов. Во-первых, мы используем селектор `Greater`

than (gt) (больше чем), так как нет селектора с названием Greater than or equal (больше чем или равен). Кроме того, в отличие от уже известных вам селекторов, он принимает аргумент (1 в данном случае), указывающий на индекс, с которого следует начинать. Почему вы пропускаете 1, если хотите начинать с третьего элемента? Не должен ли он равняться 2? Ответ кроется в наших знаниях по программированию, где индексы, как правило, начинаются с 0. Индекс первого элемента 0, второго — 1 и т. д.

Эти особые для jQuery селекторы обеспечивают удивительно изящные решения иногда даже очень сложных задач. В табл. 2.4 приведен список таких фильтров позиций (которые расположены внутри основной категории фильтров в документации jQuery).

Таблица 2.4. Фильтры позиций, поддерживаемые jQuery

Селектор	Описание	В CSS
:first	Выбирает первое совпадение в пределах контекста. li a:first возвращает первый якорь, являющийся наследником элемента списка	–
:last	Выбирает первое совпадение в пределах контекста. li a:last возвращает последний якорь, являющийся наследником элемента списка	–
:even	Выбирает четные элементы в пределах контекста. li :even возвращает каждый элемент списка с четным индексом	–
:odd	Выбирает нечетные элементы в пределах контекста. li:odd возвращает каждый элемент списка с нечетным индексом	–
:eq(n)	Выбирает n-й совпадающий элемент	–
:gt(n)	Выбирает элементы после n-го совпадающего элемента (n-й элемент исключен)	–
:lt(n)	Выбирает элементы до n-го совпадающего элемента (n-й элемент исключен)	–

Как мы уже отмечали, первый индекс в наборе элементов всегда равен 0. По этой причине селектор :even будет — как ни парадоксально — извлекать элементы, расположенные нечетно из-за их четных индексов. Например,:even соберет первый, третий и т. д. элементы комплекта, потому что у них четные индексы (0, 2 и т. д.). Таким образом, :even и :odd (четный и нечетный) связаны с индексом элементов в наборе, а не с их позицией.

Отметим также, что можно передать отрицательный индекс в :eq(), :gt() и :lt(). В этом случае элементы фильтруются, считая назад от последнего элемента. Если вы пишете p:gt(-2), то собираете только последний абзац на странице. Учитывая, что у последнего абзаца индекс –1, у предпоследнего индекс –2 и т. д., в основном вы запрашиваете все абзацы, которые идут после предпоследнего.

В отдельных ситуациях требуется выбирать не первый или последний элемент во всей странице, а каждый первый или последний элемент относительно данного родителя на странице. Узнаем, как это сделать.

2.5.2. Фильтры-потомки

Мы упомянули, что jQuery охватывает селекторы и спецификации CSS. Таким образом, неудивительно, что вы можете использовать дочерние псевдоклассы, введенные в CSS3. Они разрешают выбирать элементы на основе их позиции внутри родительского элемента. Если последний опущен, то предполагается универсальный селектор. Допустим, вы хотите получить элементы в зависимости от их позиции внутри данного элемента. Например:

```
ul li:last-child
```

выбирает последний дочерний из родительских элементов. В этом примере последний потомок `<li>` соответствует каждому элементу `<ul>`.

Может возникнуть необходимость выбрать элементы типа при условии, если они являются четвертым потомком данного родителя. Например:

```
div p:nth-child(4)
```

извлекает все `<p>` внутри `<div>`, которые являются четвертым потомком своего родительского элемента.

Псевдокласс `:nth-child()` отличается от `:eq()`, хотя их часто путают. Если используется первый из них, то подсчитываются все потомки содержащего элемента, независимо от их типа. Если второй, то подсчитываются только элементы, соответствующие прикрепленному к псевдоклассу селектору, независимо от того, сколько сестринских элементов (то есть элементов одного уровня с общим родителем) имеется перед ними. Другое важное отличие: `:nth-child()` выводится из спецификации CSS; поэтому предполагается, что индекс начинается с 1 вместо 0.

Следующим случаем применения, о котором следует вспомнить, является «извлечение всех вторых элементов, содержащих класс `description` внутри `<div>`». Такой запрос осуществляется с помощью селектора:

```
div .description:nth-of-type(2)
```

В ходе изучения этого подраздела вы поймете, что имеющиеся селекторы довольно многочисленны и действенны. Таблица 2.5 показывает все дочерние фильтры, описанные до сих пор, и многие другие. Обратите внимание: когда селектор разрешает больше синтаксисов, таких как `:nthchild()`, плюс в столбце «В CSS» означает, что поддерживаются все синтаксисы.

Таблица 2.5. Фильтры-потомки jQuery

Селектор	Описание	В CSS
:first-child	Соответствует первому элементу-потомку в пределах контекста	+
:last-child	Соответствует последнему элементу-потомку в пределах контекста	+
:first-of-type	Соответствует первому элементу-потомку указанного типа	+

Селектор	Описание	В CSS
:last-of-type	Соответствует последнему элементу-потомку указанного типа	+
:nth-child(n) :nth-child(even|odd) :nth-child(Xn+Y)	Соответствует n-му элементу-потомку, четному или нечетному элементу-потомку или n-му элементу-потомку, определенному по предоставленной формуле, в пределах контекста, основанного на данном параметре	+
:nth-last-child(n) :nth-last-child(even|odd) :nth-last-child(Xn+Y)	Соответствует n-му элементу-потомку, четному или нечетному элементу-потомку, или n-му элементу-потомку, определенному по предоставленной формуле, в пределах контекста, считая от последнего до первого элемента, основанного на данном параметре	–
:nth-of-type(n) :nth-of-type(even|odd) :nth-of-type(Xn+Y)	Соответствует n-му элементу-потомку, четному или нечетному элементу-потомку, или n-му элементу-потомку своего родителя в отношении к сестринским элементам с тем же именем элемента	+
:nth-last-of-type(n) :nth-last-of-type(even|odd) :nth-last-of-type(Xn+Y)	Соответствует n-му элементу-потомку, четному или нечетному элементу-потомку, или n-му элементу-потомку своего родителя в отношении к сестринским элементам с тем же именем элемента, считая от последнего до первого элемента	+
:only-child	Соответствует элементам, у которых нет сестринских элементов	+
:only-of-type	Соответствует элементам, у которых нет сестринских элементов того же типа	+

Как видите, `:nth-child()`, `:nth-last-child()`, `:nth-last-of-type()` и `:nth-of-type()` принимают различные типы параметров. Параметр может быть индексом, словами «четный», «нечетный» или уравнением. Последнее представляет собой формулу, где неизвестную переменную можно обозначить как `n`. Если вы хотите наметить элемент в любой позиции, кратной 3 (например, 3, 6, 9 и т. д.), то должны написать `3n`. Если необходимо выбрать все элементы в позиции, кратной 3 плюс 1 (например, 1, 4, 7 и т. д.), то вы должны написать `3n+1`.

Чем дальше, тем сложнее, поэтому лучше увидеть несколько примеров. Рассмотрим следующую таблицу с лабораторной страницы DOM. Она содержит список языков программирования и некоторые основные сведения о них:

```
<table id="languages">
  <thead>
    <tr>
      <th>Язык</th>
      <th>Тип</th>
      <th>Разработан</th>
    </tr>
  </thead>
  <tbody>
```

```
    <tr>
      <td>Java</td>
      <td>Статический</td>
      <td>1995</td>
    </tr>
    <tr>
      <td>Ruby</td>
      <td>Динамический</td>
      <td>1993</td>
    </tr>
    <tr>
      <td>Smalltalk</td>
      <td>Динамический</td>
      <td>1972</td>
    </tr>
    <tr>
      <td>C++</td>
      <td>Статический</td>
      <td>1983</td>
    </tr>
  </tbody>
</table>
```

Предположим, вы хотели бы получить все ячейки таблицы, которые содержат названия языков программирования. Поскольку все они являются первыми ячейками в своих строках, можно обратиться к:

```
#languages td:first-child
```

Можно также написать:

```
#languages td:nth-child(1)
```

но первый синтаксис считается более содержательным и элегантным.

Чтобы захватить ячейки типа языка, следует изменить селектор: `:nth-child(2)`, а для получения года их разработки можно применить `:nth-child(3)` или `:last-child`. Если нужна самая последняя ячейка таблицы (содержащая текст `1983`), то следует использовать псевдокласс `:last`, рассматриваемый в предыдущем подразделе, в результате которого возникает `td:last`.

Для проверки своих способностей можете представить другую ситуацию. Предположим, вы хотите получить названия языков и год их разработки с помощью `:nth-child()`. В принципе, здесь необходимо взять для каждой строки таблицы (`<tr>`) первый и третий столбцы (`<td>`). Первым и простейшим решением представляется передача `odd` в качестве аргумента фильтру, в результате чего будет:

```
#languages td:nth-child(odd)
```

Просто чтобы немного повеселиться, усложним предыдущий пример: предположим, вы хотите выполнить тот же выбор, применяя формулу к фильтру `:nth-child()`. Помня, что индекс для данного фильтра начинается с 1, можно превратить предыдущий селектор в такой:

```
#languages td:nth-child(2n+1)
```

Этот последний пример должен укрепить вас в мысли, что jQuery предоставляет много возможностей.

Прежде чем двигаться дальше, вернитесь к странице Selectors Lab и попробуйте выбрать вторую и четвертую записи из списка. Затем попытайтесь найти три различных способа, чтобы выбрать ячейку, содержащую текст `1972` в таблице. Кроме того, постарайтесь понять разницу между типом фильтра `:nth-child()` и селекторами абсолютной позиции.

Даже несмотря на то, что селекторы CSS, которые мы разбирали до сих пор, являются невероятно эффективными, мы обсудим, как сделать еще более действенными селекторы jQuery, специально разработанные для целевых элементов формы или их статуса.

2.5.3. Фильтры формы

Селекторы CSS, которые встречались вам до сих пор, дают большую продуктивность и гибкость для поиска и выбора элементов DOM на странице, но есть еще большее количество селекторов, которые дают вам более широкие возможности для фильтрации выбора.

Например, нужно выбрать все флажки в состоянии «установлен» (checked). Может возникнуть соблазн попробовать такой вариант:

```
$('input[type="checkbox"][checked]');
```

Но команда «выбрать по атрибуту» проверит только начальное состояние контрольного элемента, как указано в HTML-разметке. Что действительно нужно проверить, так это состояние контрольных элементов в режиме реального времени. CSS предлагает псевдокласс `:checked`, который выбирает элементы, находящиеся в состоянии «установлен». Например, в то время, как селектор `input[type="checkbox"]` выбирает все элементы `input`, являющиеся флажками, селектор `input[type="checkbox"]:checked` сужает поиск только элементов `input`, которые также являются флажками и в настоящее время установлены. При переписывании предыдущей команды для выбора всех флажков, которые в настоящее время проверяются с помощью фильтра, можно написать:

```
$('input[type="checkbox"]:checked');
```

jQuery также предоставляет несколько эффективных селекторов пользовательских фильтров, не определенных CSS, которые упрощают выбор целевых элементов. Например, пользовательский селектор `:checkbox` определяет все элементы флажка. Сочетание этих пользовательских селекторов может быть очень действенным и позволит сократить количество ваших селекторов. Рассмотрим наш еще раз переписанный пример с использованием одних только фильтров:

```
$('input:checkbox:checked');
```

Как мы уже говорили ранее, библиотека поддерживает селекторы фильтров CSS, а также определяет количество пользовательских селекторов. Они описаны в табл. 2.6.

Таблица 2.6. Селекторы CSS и пользовательские фильтры jQuery

Селектор	Описание	В CSS
:button	Выбирает только элементы кнопок (input[type=submit], input[type=reset], input[type=button] или button)	–
:checkbox	Выбирает только элементы флажков (input[type=checkbox])	–
:checked	Выбирает элементы флажков или переключателей в состоянии checked или option элементов select, находящиеся в выбранном состоянии	+
:disabled	Выбирает только элементы, находящиеся в неактивном состоянии	+
:enabled	Выбирает только элементы, находящиеся в активном состоянии	+
:file	Выбирает только элементы ввода файла (input[type=file])	–
:focus	Выбирает только элементы, у которых есть фокус на момент выполнения селектора	+
:image	Выбирает только элементы изображения (input[type=image])	–
:input	Выбирает только элементы форм (input, select, textarea, button)	–
:password	Выбирает только элементы пароля (input[type=password])	–
:radio	Выбирает только элементы переключателей (input[type=radio])	–
:reset	Выбирает только кнопки reset (input[type=reset] или button[type=reset])	–
:selected	Выбирает только элементы option, находящиеся в выбранном состоянии	–
:submit	Выбирает только кнопки подтверждения формы (button[type=submit] или input[type=submit])	–
:text	Выбирает только элементы текста (input[type=text]) или ввод без указанного типа (поскольку значение по умолчанию type=text)	–

Эти CSS и пользовательские селекторы фильтров jQuery могут быть также объединены. Например, если нужно выбрать только включенные и проверенные флажки, то используйте следующее:

```
$('input:checkbox:checked:enabled');
```

Потренируйтесь на как можно большем количестве таких фильтров на странице Selectors Lab, пока не почувствуете, что хорошо понимаете принципы их деятельности.

Эти фильтры являются чрезвычайно полезным дополнением к набору селекторов из вашего арсенала. Возможно, вы подумали, пусть даже на мгновение, что наши селекторы закончились? Вот уж нет!

2.5.4. Фильтры содержимого

Еще одна из категорий, которую можно найти в документации jQuery, содержит *фильтры содержимого*. Как следует из названия, эти фильтры предназначены для выбора элементов в зависимости от их содержания. Например, вы можете выбрать элементы, если они содержат определенное слово или если содержимое абсолютно

пусто. Обратите внимание, что под *содержимым* мы имели в виду не только сырой текст, но также и дочерние элементы.

Как вы уже успели заметить, CSS определяет полезный селектор для выбора элементов, являющихся потомками конкретных родителей. Например, селектор

```
div span
```

выберет все элементы `span`, являющиеся наследниками `div`.

А если наоборот? Если вы хотите выбрать все `<div>`, содержащие элементы `span`? Эту работу выполняет фильтр `:has()`. Селектор:

```
div:has(span)
```

выбирает элементы-предков `div` в противоположность дочерним элементам `span`.

Эти фильтры окажутся мощным подспорьем в момент, когда необходимо выбрать элементы, которые представляют собой сложные конструкции. Допустим, вы хотите найти, какая строка таблицы содержит определенный элемент изображения, который может быть однозначно идентифицирован с помощью его атрибута `src`. Можно использовать селектор наподобие этого:

```
$('tr:has(img[src="puppy.png"])');
```

Он может вернуть любой элемент строки таблицы, содержащий идентифицированное изображение, в любое место в его дочерней иерархии.

Полный перечень содержимого фильтров приведен в табл. 2.7.

Таблица 2.7. Фильтры содержимого, поддерживаемые jQuery

Селектор	Описание	В CSS
:contains(text)	Выбирает только элементы, содержащие указанный текст (текст в потомках и их потомках также рассматривается)	–
:empty	Выбирает только элементы, у которых нет потомков (включая текстовые узлы)	+
:has(selector)	Выбирает только элементы, содержащие по меньшей мере один элемент, соответствующий указанному селектору	–
:parent	Выбирает только элементы, у которых есть хотя бы один узел-потомок (элемент либо текст)	–

Если вы устали от обилия всех этих селекторов и фильтров, то мы советуем вам немного передохнуть, поскольку это еще не все!

2.5.5. Другие фильтры

Вы ознакомились с невероятным количеством селекторов и фильтров (специальных селекторов), о существовании которых, возможно, даже не догадывались. Ваше путешествие в мир селекторов еще не закончилось, и в этом подразделе мы обсудим оставшиеся. Часть из них, `:visible` и `:hidden`, классифицируются в документации библиотеки под названием *«фильтры видимости»*, но для краткости мы решили включить их сюда.

Если вы хотите инвертировать селектор — скажем, чтобы он соответствовал любому элементу ввода, который *не* является флажком, — то можете использовать фильтр `:not()`. Например, для выбора элементов `input`, не являющихся флажками, можно применить:

```
input:not(:checkbox)
```

Но будьте осторожны! Легко сбиться с пути и получить неожиданные результаты!

Предположим, вы хотите выбрать все изображения, за исключением тех, у которых атрибут `src` содержит текст `dog`. Можно быстро придумать следующий селектор:

```
$(':not(img[src*="dog"])');
```

Но если вы его используете, то обнаружите, что получите не только все элементы изображения, которые не содержат `dog` в их `src`, но и вообще каждый элемент в DOM, не являющийся изображением элемента со значением такого атрибута `src`!

Ой! Следует помнить: когда основной селектор опущен, по умолчанию работает универсальный. Ваш ошибочный селектор фактически читает задачу, как «извлечь все элементы, не являющиеся изображением, которые относятся к `'dog'` в их атрибутах `src`». То, что вы имели в виду, выглядит так: «извлечь все элементы изображения, которое не относится к `'dog'` в их атрибутах `src`» — и может быть выражено следующим образом:

```
$('img:not([src*="dog"])');
```

Потренируйтесь на лабораторной странице, пока не разберетесь с тем, как задействовать фильтр `:not()` для инвертирования выбора.

При работе с jQuery часто принято использовать ее методы, чтобы скрыть один или несколько элементов на странице. Для этого можно применить фильтр `:hidden`. Элемент считается скрытым не только тогда, когда к нему применяется:

```
display: none;
```

но и если он не занимает места. Например, скрытым называется также элемент, ширина и высота которого устанавливаются равными нулю. С помощью селектора:

```
input:hidden
```

вы делаете целевыми все скрытые элементы `input` страницы.

jQuery 3: изменения

jQuery 3 немного изменила значение фильтра `:visible` (а следовательно, и `:hidden`). Начиная с jQuery 3, элементы рассматриваются `:visible`, если у них есть контейнеры шаблона, даже если имеют нулевую ширину или высоту. Например, элементы `br` и встроенные (inline) элементы без содержимого сейчас будут выбираться фильтром `:visible`.

При создании веб-страниц часто используются иностранные слова. Если вы пишете правильный, семантический HTML, то сможете добавить возле этих слов теги `<em>` с атрибутами `lang` для указания языка. Предположим, у вас есть страница о пицце; можно создать разметку наподобие следующей:

```
<p>Первая пицца названа <em lang="it">Margherita</em>, и она была
       создана в городе <em lang="it">Napoli</em> (Италия).</p>
```

Вы можете выбрать все иностранные слова этого примера с помощью фильтра `:lang()` таким образом:

```
var $foreignWords = $('em:lang(it)');
```

Полный перечень оставшихся фильтров приведен в табл. 2.8.

Таблица 2.8. Оставшиеся фильтры, поддерживаемые jQuery

Селектор	Описание	В CSS
:animated	Выбирает только элементы, которые в настоящее время находятся под управлением анимирования	–
:header	Выбирает только элементы, которые являются заголовками: от <h1> до <h6>	–
:hidden	Выбирает только скрытые элементы	–
:lang(язык)	Выбирает элементы на указанном языке	+
:not(селектор)	Отрицает указанный селектор	+
:root	Выбирает элемент, являющийся корнем документа	+
:target	Выбирает целевой элемент, указанный по идентификатору фрагмента в URI документа	+
:visible	Выбирает только видимые элементы	–

Хотя библиотека jQuery предлагает невероятное количество селекторов, она не охватывает все возможные случаи использования. Команда разработчиков библиотеки знает об этом, вследствие чего дает вам возможность создавать свои собственные фильтры. Посмотрим, как это можно сделать.

2.5.6. Как создать пользовательские фильтры

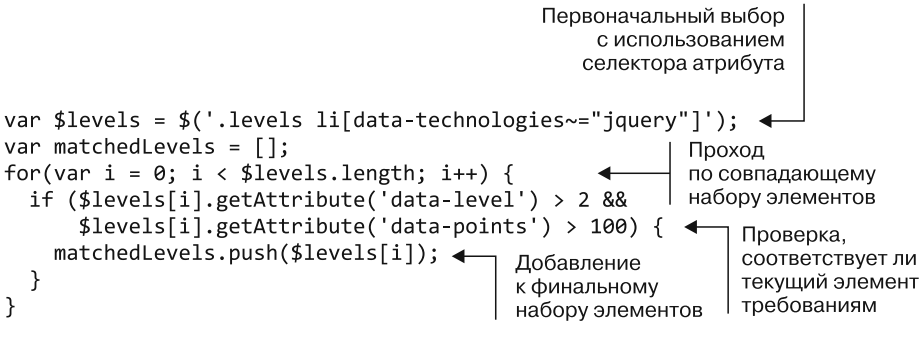
В предыдущих подразделах вы изучили все селекторы и фильтры, поддерживаемые jQuery. Вне зависимости от их количества вы можете иметь дело с не описанными здесь случаями. Вы также можете осуществлять один и тот же выбор, а затем одну и ту же фильтрацию извлеченного набора много раз, задействуя конструкции циклов и выборов. В подобных ситуациях можно создать ярлык для сбора узлов DOM или, выражаясь точнее, создать *пользовательский фильтр* (также называемый *пользовательским селектором* или *пользовательским псевдоселектором*).

В jQuery есть два способа создания пользовательского фильтра. Первый написать проще, но применять его не рекомендуется, так как, начиная с jQuery 1.8, он был заменен на второй. В этой книге мы опишем только новый метод, но если вы хотите взглянуть на старый, то мы подготовили для вас JS Bin (http://jsbin.com/ImIboXAz/edit?html,js,console,output). Пример также доступен в файле `chapter-2/custom.filter.old.html`, предоставленном с книгой. Имейте в виду: при использовании нового подхода вы разрабатываете пользовательский фильтр, который не будет работать в версиях jQuery до 1.8. Тем не менее во многих случаях проблем быть не должно, поскольку эта версия устарела.

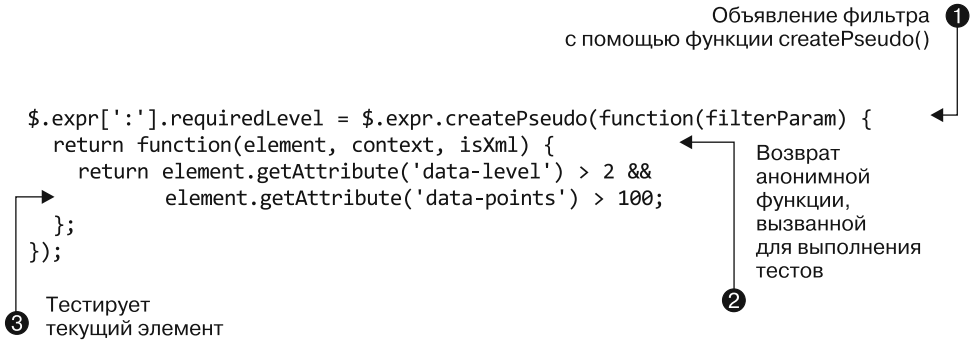
Чтобы объяснить новый способ создания пользовательского фильтра, начнем с примера. Представьте: вы разрабатываете техническую игру. У вас есть список уровней, которые следует пройти с определенной степенью сложности, количество очков, которые может заработать пользователь, а также перечень технологий для ее успешного прохождения. Ваш гипотетический список может выглядеть примерно так:

```
<ul class="levels">
  <li data-level="1" data-points="1" data-technologies="javascript node
    grunt">Уровень 1</li>
  <li data-level="2" data-points="10"
      data-technologies="php composer">Уровень 2</li>
  <li data-level="3" data-points="100" data-technologies="jquery
    requirejs">Уровень 3</li>
  <li data-level="4" data-points="1000" data-technologies="javascript jquery
    backbone">Уровень 4</li>
</ul>
```

Теперь представьте, что нужно часто выбирать уровни (`data-level`) выше, чем 2, но только тогда, когда они позволяют вам заработать больше чем 100 очков (`data-points`) и имеют jQuery в списке технологий для использования (`data-technologies`). Вы уже знаете, как искать элементы `li`, имеющие ключевое слово `jquery` внутри атрибута `data-technologies` (`li[data-technologies~="jquery"]`). Но как вы будете сравнивать числа с помощью селекторов? Правда в том, что вы не можете этого сделать. Чтобы достичь цели, необходимо перебрать изначальный выбор, а затем сохранить только нужные вам элементы, как показано здесь:



Вместо того чтобы повторять эти строки каждый раз, можно создать пользовательский фильтр:



Как видите, фильтр — не что иное, как функция, добавленная к свойству `:`, которое принадлежит атрибуту jQuery `expr`. Здесь нет ошибки, дорогой читатель. Это свойство называется «двоеточие». Оно содержит собственные фильтры jQuery, и его можно использовать для добавления своих собственных.

Вы можете назвать ваш пользовательский фильтр `requiredLevel` и вместо передачи функции напрямую применить утилиту jQuery (на самом деле она принадлежит основному инструменту селекторов Sizzle), которая называется `createPseudo()` 1.

Функции `createPseudo()` вы передаете анонимную функцию, где объявляете параметр `filterParam`. Название последнего, означающее «параметр фильтра», является произвольным и при желании заменяется на любое другое. Данный параметр необязательный, и его можно передать в фильтр, как и фильтры, такие как `:eq()` и `:nth-child()`, с которыми вы не будете работать в данный момент. Внутри этой анонимной функции вы создаете другую, и она будет возвращена. Эта последовательность отвечает за выполнение фильтрации. Внутренней функции jQuery передает элементы, которые будут обрабатываться по одному за раз (параметр `element`), DOMElement или DOMDocument. Из них будет производиться выбор (параметр `context`), и булево значение, определяющее, работаете ли вы в XML-документе или нет (параметр `isXML`) 2. В середине самой внутренней функции вы пишете код, чтобы проверить, должен ли сохраняться элемент 3. В вашем случае вы проверяете, соблюдаются ли требуемые показатели: уровень должен быть выше чем 2, и количество возможных заработанных пользователем очков должно превышать 100.

В предыдущем примере мы ввели аргумент `filterParam`, который можно использовать для передачи параметра в ваш пользовательский фильтр. Из-за фиксированного характера ваших требований мы его не применяли. Ради интереса посмотрим, как он может вам помочь.

Представьте, что вы хотите выбрать уровни на основе предложенного количества баллов — например, «выбрать все уровни с количеством очков выше чем X». Этот большой X является хорошей возможностью использовать параметр для

передачи вашему псевдоселектору. Основываясь на данном требовании, можно создать новый фильтр:

По сравнению с предыдущим примером есть несколько отличий. Как и прежде, вы используете функцию `createPseudo()`, но называете фильтр `pointsHigherThan`. Прежде чем объявить вторую функцию, нужно сохранить аргумент в переменную `points` (1), поэтому она будет доступна в его замыкании (если вы не знаете, что это такое, прочитайте соответствующий раздел в приложении). Теперь можно задействовать данный аргумент через применение сохраненной переменной (2).

Запустим этот новый фильтр. Если вы хотите выбрать все уровни, которые позволяют вам заработать более 50 очков, то можете написать получение двух последних элементов списка:

```
var $elements = $('.levels li:pointsHigherThan(50)');
```

Оба описанных здесь пользовательских фильтра доступны в файле `chapter-2/custom.filter.html`, предоставленном с книгой, а также в JS Bin (http://jsbin.com/mucigo/edit?html,js,console,output).

До сих пор вы использовали половину мощности функции `jQuery()`, служащей для выбора элементов, так как применяли только один из двух параметров, которые могут вам встретиться. Пришло время это исправить.

2.6. Улучшение производительности с использованием контекста

До сих пор мы действовали так, как будто существовал только один аргумент, который можно передать в функцию jQuery `$()`, но это было сделано для того, чтобы рассказать вначале о более простых вещах. В главе 1 мы кратко представили второй параметр `context` (контекст). Он используется для ограничения выбора одного или нескольких поддеревьев DOM, в зависимости от применяемого селектора. Этот аргумент очень полезен, когда ваша страница содержит большое количество элементов, потому что он может сузить поддерево (поддеревья), где jQuery будет выполнять второй этап поиска.

Как вы увидите, со многими методами jQuery, когда необязательный аргумент опущен, предполагается приемлемое значение по умолчанию. Точно так же с `context`. Когда селектор передается в качестве первого параметра, `context` по умолчанию передается объекту `document`, применяя этот селектор к каждому элементу в дереве DOM.

Часто это именно то, чего вы хотите, то есть это хороший принцип по умолчанию. Но бывают случаи, когда нужно ограничить поиск подмножества всего DOM. В таких случаях можете идентифицировать подмножество DOM, служащее в качестве корня поддерева, к которому применяется селектор.

Страница Selectors Lab предлагает хороший пример такого сценария. Когда страница применяется к селектору, который вы ввели в текстовое поле, селектор обрабатывает только подгруппу DOM, загруженную на панель DOM Sample (Пример DOM).

Вы можете использовать ссылку на DOM-элемент в качестве параметра `context`, но также и строку, содержащую селектор jQuery или коллекцию jQuery. (Да, это означает, что можно передать результат одного вызова `$()` к другому — не торопитесь хвататься за голову — все не так страшно, как кажется на первый взгляд.)

Когда селектор или коллекция jQuery предоставляется в качестве `context`, выявленные элементы служат как контекст для применения селектора. Поскольку может быть множество таких элементов, это хороший способ обеспечить разнородные поддеревья в DOM, которые бы послужили в качестве контекста для процесса выбора.

Возьмем для примера лабораторную страницу. Мы предполагаем, что строка селектора хранится в переменной, для удобства названной `selector`. Когда вы применяете этот представленный селектор, можете захотеть применить его только к тому образцу DOM, который содержится внутри элемента `div` с ID `sample-dom`.

Если бы вам пришлось кодировать вызов функции jQuery таким образом:

```
$(selector);
```

то селектор применялся бы ко всему дереву DOM, в том числе и к элементу `form`, в котором он был задан. Такой результат вас не устраивает. Нужно ограничить выбор поддеревом DOM, начинающимся с элемента `div` с ID `sample-dom`, так что вместо предыдущего выражения вы пишете:

```
$(selector, '#sample-dom');
```

что ограничивает применение селектора к нужной части DOM.

При использовании параметра `context` jQuery сначала получает элементы, основанные на нем, а затем выбирает потомков, соответствующих селектору в качестве первого аргумента. Иными словами, вы ищете элементы, которые соответствуют `selector` и которые должны иметь `context` в качестве их предшественника. Таким образом, селектор потомка может быть заменен с помощью `context`. Рассмотрим следующий пример, где можно выбрать `<p>` внутри `<div>`:

```
$('div p');
```

Он может быть превращен в:

```
$('p', 'div');
```

давая тот же результат.

На этом завершим обсуждение селекторов jQuery. Мы знаем, как трудно было пройти через все эти селекторы, но вы не должны чувствовать себя обескураженными. Постарайтесь усвоить описанные концепции не торопясь, и когда почувствуете, что готовы, двигайтесь дальше.

Прежде чем рассматривать методы главы 3, мы проверим ваши навыки с помощью некоторых упражнений, ориентированных на ранее описанные концепции.

2.7. Проверьте свои навыки с помощью упражнений

Здесь вы будете практиковаться, выполняя ряд упражнений, связанных с представленными в этой главе селекторами и фильтрами. Если вы хотите проверить свои варианты ответов, то воспользуйтесь страницей jQuery Selectors Lab. Кроме того, мы предоставим наши решения, и вы сможете сравнить их с вашими.

2.7.1. Упражнения

Вот список упражнений.

1. Выберите все ссылки на странице.
2. Выберите все прямые дочерние ссылки элементов `<div>`, имеющих класс `wrapper`.
3. Выберите все ссылки и абзацы, чей предок — `<div>`.
4. Выберите все элементы `<span>`, у которых атрибут `data-level` равен `hard`, но нет атрибута `data-completed`, равного `true`.
5. Выберите все элементы на странице с именем класса `wrapper` без использования селектора класса.
6. Выберите третий элемент списка внутри списка, имеющего ID `list`, на любом уровне.
7. Выберите все элементы списка (`li`) внутри списка с ID `list`, после второго.
8. Выберите абзацы, которые являются кратным 3 плюс 1 (1, 4, 7 и т. д.) потомком их предка, чей класс `description`.
9. Выберите элементы `<input>` типа `password`, только если они необходимы (атрибут для HTML5 `required`) и являются первыми дочерними элементами `<form>`.
10. Выберите все `<div>` на странице, у которых нет потомков и класса `wrapper` и которые имеют нечетную позицию (подсказка: не индекс!).
11. Создайте пользовательский фильтр для выбора элементов, имеющих только цифры, буквы или подчеркивание (_) в качестве текста.

2.7.2. Решения

Вот список решений.

1. `$('a')`.
2. `$('div.wrapper > a')`.
3. `$('div a, div p')` или даже лучше, используя параметр `context`, `$('a, p', 'div')`.

4. `$('span[data-level="hard"][data-completed!="true"]')`.
5. `$('[class~="wrapper"]')`.
6. `$('#list li:eq(2)')` или даже лучше `$('li:eq(2)', '#list')`.
7. `$('li:gt(1)', '#list')`.
8. `$('p.description:nth-child(3n+1)')`.
9. `$('input[required]:password:first-child', 'form')`.
10. `$('div:empty:even:not(.wrapper)')`.
11.
```
$.expr[":"].onlyText = $.expr.createPseudo(function(filterParam) {
  return function(element, context, isXml) {
    return element.innerHTML.match(/^\w+$/);
  }
});
```

Как вы справились с заданиями? Чувствуете, что поняли весь материал? Хорошо! Мы завершили обзор доступных селекторов и способов создания своих собственных.

2.8. Резюме

В главе 2 мы сосредоточились на создании и настройке наборов элементов (называемых в этой главе и за ее пределами *коллекцией jQuery* или *набором соответствующих элементов*) с помощью многих средств jQuery, благодаря которым обеспечивается идентификация элементов на странице HTML.

Библиотека предоставляет универсальный и действенный набор селекторов, чьими образцами были селекторы CSS, для идентификации элементов в пределах страницы документа с кратким, но эффективным синтаксисом. Эти селекторы включают в себя синтаксис CSS3, который в настоящее время поддерживается большинством современных браузеров. jQuery не только поддерживает все селекторы CSS, но и расширяет их за счет собственного набора селекторов, предлагая вам еще более выразительные средства для выбора элементов на веб-странице. Помимо всего прочего, библиотека является настолько гибкой, что позволяет вам создавать собственные фильтры.

В этой главе мы рассмотрели все селекторы, доступные в jQuery. В следующей мы увидим, как использовать функцию `$()` для создания *новых* HTML-элементов. Вы также познакомитесь с методами, которые принимают селектор в качестве параметра для выполнения отдельных операций на наборе соответствующих элементов.

3 Операции с коллекцией jQuery

В этой главе:

- создание и введение новых HTML-элементов в DOM;
- манипулирование коллекцией jQuery;
- итерирование по элементам коллекции jQuery.

Из этой главы вы узнаете, как создавать новые элементы DOM, используя чрезвычайно гибкую функцию `jQuery()`. При работе с библиотекой довольно часто возникает необходимость создавать новые элементы, и jQuery это позволяет. Вам понадобится такая возможность особенно тогда, когда мы начнем обсуждать, как вводить внешние данные в веб-страницу с помощью форматов JSON и XML и методов jQuery в работе с Ajax.

Кроме того, вы изучите другие методы, отличающиеся от `jQuery()`. Они будут разделены на две части. Сперва мы опишем методы, которые, начиная с коллекции jQuery, принимают селектор в качестве параметра для создания нового набора элементов. Например, вы увидите, как, исходя из набора, создать новый, содержащий все дочерние элементы исходного набора, дополнительно фильтрованные с помощью селектора, переданного в качестве аргумента. Затем мы рассмотрим методы, строго не связанные с селекторами, но позволяющие вам итерировать по элементам в наборе или проверить их. Начнем!

3.1. Добавление нового HTML-кода

Во многих случаях вам понадобится создать новые фрагменты HTML и вставить их в страницу. Такие динамические элементы могут быть как простыми, например дополнительный текст, который вы хотите отобразить, так и сложными, такими как создание таблицы в базе данных с результатами, полученными от сервера. Типичная ситуация, в которой вам может пригодиться данная функция, — когда нужно получить внешние данные, как правило, в виде JSON или XML, с помощью Ajax.

Используя jQuery, создать динамические элементы просто. Вы легко можете добавить динамически созданный объект jQuery, содержащий элементы DOM,

передавая функции `$()` строку с HTML-разметкой для этих элементов. Рассмотрим следующую строку:

```
$('<div>Привет</div>');
```

Этот код создает новый объект jQuery, содержащий элемент `div`, готовый к добавлению на страницу (на данный момент он не введен в DOM). Любой метод jQuery, который вы можете запустить для набора существующих элементов, может быть запущен и для свежесозданных HTML-фрагментов. На первый взгляд не впечатляет, но, когда будете работать одновременно с обработчиками событий, Ajax и эффектами (а это вам предстоит в последующих главах), поймете, насколько это действенный инструмент на самом деле.

Обратите внимание: если хотите создать пустой элемент `div`, то можете использовать краткую запись:

```
$('<div>');
```

Она идентична записям `$('<div></div>')` и `$('<div />')`, хотя настоятельно рекомендуем применять хорошо сформированную разметку и включать открытые и закрытые теги для любых типов элементов, которые могут содержать другие элементы. С точки зрения производительности эти три варианта эквивалентны, что становится понятным, если посмотреть на тест, показанный на рис. 3.1 (актуальный тест — на http://jsperf.com/jquery-create-markup/4).

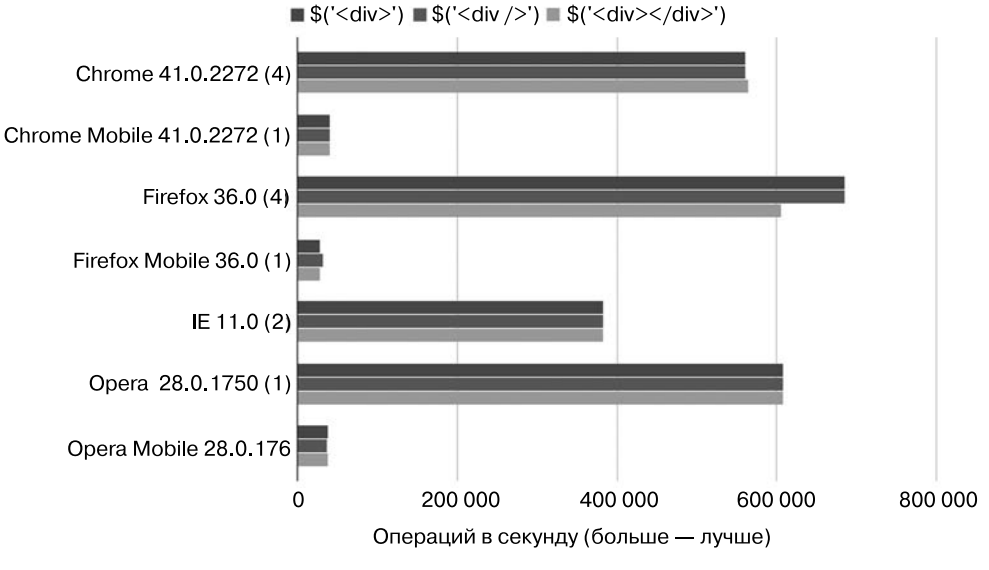


Рис. 3.1. Эталонный тест, сравнивающий три способа создания нового элемента с помощью jQuery(). Он доказывает, что эти способы эквивалентны по производительности практически в каждом браузере

Такие простые HTML-элементы создавать очень легко, а благодаря возможностям цепочки методов jQuery вводить более сложные элементы не намного

сложнее. Вы можете применить любой метод библиотеки к коллекции jQuery, содержащей вновь созданный элемент. Вы можете также присваивать атрибуты элементу с помощью метода `attr()` (мы разберем это в следующей главе), но у jQuery еще более эффективное средство.

В предыдущей главе мы познакомили вас с параметром `context` функции `$()`. При создании нового элемента с функцией `$()` вы используете данный параметр, чтобы указать атрибуты и их значения для элемента, создаваемого вами в виде объекта JavaScript. Свойства такого объекта служат в качестве имен атрибутов, которые будут применены к элементу, а значения соответственно в качестве значений атрибутов.

Предположим, вы хотите создать элемент `img` в комплекте с несколькими атрибутами и сделать его доступным для загрузки щелчком кнопкой мыши. Взгляните на код в листинге 3.1.

Листинг 3.1. Динамически создаваемый полнофункциональный элемент img

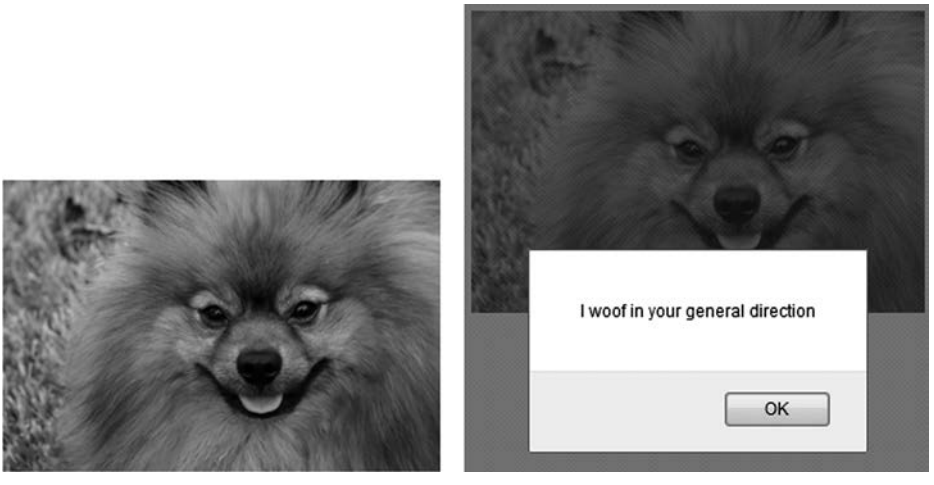
```
$('<img>',                          ← 1 Создает базовый элемент img
  {
    src: 'images/little.bear.png',     2 Присваивает различные атрибуты
    alt: 'Маленький Медведь',
    title:'Гав на вас!',
    click: function() {                3 Устанавливает обработчик для щелчка кнопкой мыши
      alert($(this).attr('title'));
    }
  })
  .appendTo('body');                ← 4 Добавляет элемент в DOM, в конце элемента body
```

В листинге один и тот же оператор jQuery создает базовый элемент `img` ❶; наделяет его важными атрибутами, используя второй параметр, такой как его источник (`source`), альтернативный текст (`alt`) и всплывающее название (`title`) ❷; присоединяет его к дереву DOM (в качестве дочернего элемента `body`) ❹. В показанном примере вы добавляете элемент в DOM с помощью метода `appendTo()`. Мы его еще не рассматривали, но он добавляет элементы в коллекцию jQuery — в этом случае только свежесозданное изображение — к элементу, указанному в аргументе (в нашем примере — элементу `body`).

Мы здесь немного схитрили. В примере вы также применяете второй параметр, чтобы установить обработчик событий, выдающий сообщение (текст которого получен из атрибута `title` изображения), когда мы щелкнем кнопкой мыши на изображении ❸.

Независимо от того, как вы составите код, это довольно громоздкий оператор. Он занимает несколько строк, имеет логические отступы для удобного чтения — но и делает немало. Такие операторы не редкость на страницах с поддержкой jQuery, и если вам это кажется чересчур сложным, то не волнуйтесь. Мы рассмотрим каждый метод, используемый в данном операторе, в нескольких последующих главах. И вскоре вы легко сможете писать составные операторы, подобные этому.

На рис. 3.2 показан результат этого кода, как при первой загрузке страницы *(а)*, так и после щелчка на изображении *(б)*. Полный код для примера можно найти в файле `chapter-3/listing-3.1.html`, предоставленном с книгой.



а *б*

Рис. 3.2. Пример динамически создаваемого изображения: *а* — создание сложных элементов (включая изображение, генерирующее сообщение по щелчку кнопкой мыши) — просто как дважды два; *б* — динамически создаваемое изображение обладает всеми необходимыми стилями и атрибутами, включая обработчик щелчка кнопкой мыши, который создает сообщение

До сих пор вы применяли методы для всего набора соответствующих элементов, но в дальнейшем вам понадобится совершить с ними некие операции, прежде чем пускать в действие.

3.2. Управление коллекцией jQuery

После получения набора jQuery, сделанного из существующих элементов DOM с селекторами либо созданного в виде новых элементов с использованием HTML-фрагментов (а возможно, и того и другого), вы готовы манипулировать этими элементами, задействуя эффективный набор методов библиотеки. Мы начнем рассматривать их в следующей главе, но что, если вы хотите еще большего от набора jQuery, с которым хотите работать? В этом разделе мы рассмотрим множество способов его улучшения, расширения или фильтрации.

Чтобы помочь вам, мы добавили еще одну страницу в загружаемом коде проекта для этой главы: jQuery Operations Lab (`chapter-3/lab.operations.html`). Она показана на рис. 3.3 и во многом напоминает страницу Selectors Lab, к которой мы обращались в главе 2.

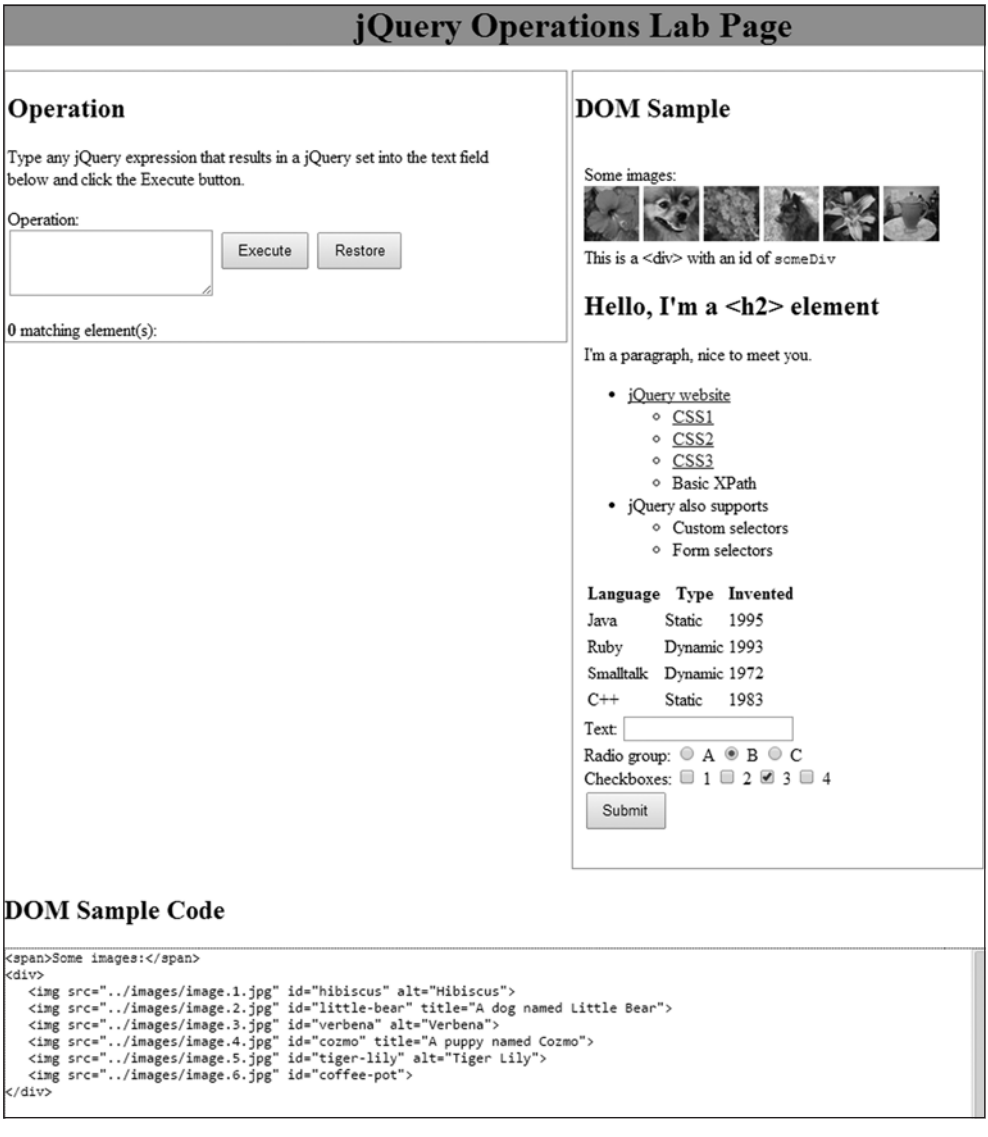


Рис. 3.3. Страница jQuery Operations Lab позволит вам создавать коллекции jQuery в режиме реального времени, чтобы помочь увидеть, как можно создавать и управлять коллекциями

Эта новая страница не только выглядит, как Selectors Lab, но и работает аналогичным образом. Но здесь вместо того, чтобы вводить селектор, можно ввести любую полную операцию jQuery, которая приведет к коллекции jQuery. Операция выполняется в контексте DOM Sample, и, как в случае с Selectors Lab, ее результаты отображаются на экране.

ПРИМЕЧАНИЕ

Эта лабораторная страница загружает элементы, на которые она действует, внутри iframe. Из-за ограничений безопасности некоторых браузеров данная операция может завершиться неудачей. Чтобы этого избежать, вы можете либо запустить страницу в работу с помощью таких веб-серверов, как Apache, Tomcat или IIS, либо поискать конкретное решение для вашего браузера. Например, для браузеров на основе WebKit можно запустить их через интерфейс командной строки (CLI), установив флажок --allow-file-access-fromfiles. Важно, что команда создает новый процесс, поэтому должна открываться не новая вкладка, а новое окно.

Страница jQuery Operations Lab позволяет ввести любое выражение, которое приводит к набору jQuery. Из-за особенностей поэтапной работы библиотеки данное выражение может также включать в себя методы jQuery. Это позволяет странице принести реальную пользу при изучении операций jQuery.

Имейте в виду: вам необходимо ввести правильный синтаксис, а также выражения, которые создают набор jQuery. В противном случае вы столкнетесь с ошибками JavaScript.

Загрузите лабораторную страницу в своем браузере и введите этот текст в поле Operation (Операция):

```
$('img').hide();
```

Затем нажмите кнопку Execute (Выполнить). Операция выполняется в контексте примера DOM, и вы увидите, как изображения исчезают из примера.

После любой операции можно вернуть пример DOM в исходное состояние, нажав кнопку Restore (Восстановить). Хотя мы этого еще не рассматривали, метод `hide()` принадлежит jQuery, и мы еще уделим ему внимание. На данный момент нужно знать: данная функция позволяет скрыть все элементы в наборе. Мы использовали ее с целью продемонстрировать вам конкретный пример того, что вы можете сделать на новой странице Operations Lab. Вы увидите ее в действии, когда проработаете материал в последующих разделах, и, возможно, она пригодится вам в дальнейшем, чтобы проверить различные операции jQuery.

3.2.1. Определение размера набора

Мы уже упоминали ранее, что набор элементов jQuery во многом работает как массив. Это сходство включает в себя также и свойство `length`, совсем как в массивах JavaScript, — оно определяет количество элементов в коллекции jQuery.

Допустим, вы хотите узнать количество всех абзацев на странице и вывести получившееся значение на экран. Можно написать следующую инструкцию:

```
alert($('p').length);
```

Итак, теперь известно, сколько у вас элементов. Что если вы хотите обратиться к ним напрямую?

3.2.2. Получение элементов из набора

После того как у вас появляется набор jQuery, вы часто используете методы jQuery для выполнения с ним какой-либо операции в целом. Возможны случаи, когда вы хотите получить прямую ссылку на элемент или элементы для совершения с ними операций JavaScript. Посмотрим на некоторые из способов, благодаря которым это возможно.

Получение элементов по индексу

Поскольку библиотека позволяет работать с коллекцией jQuery как с массивом JavaScript, можно использовать простую индексацию массива для получения любого элемента в списке по его позиции. Например, чтобы получить первый элемент в множестве всех `<img>` с атрибутом `alt` на странице, можно написать:

```
var imgElement = $('img[alt]')[0];
```

Самые наблюдательные из вас могли заметить, что мы не ставим знак доллара (`$`) перед именем переменной (`imgElement`). Мы о нем не забыли. Данный набор jQuery содержит массив элементов DOM, так что если вы извлекаете один элемент, то это будет не набор jQuery из одного элемента, а простой элемент DOM.

Если вы предпочитаете использовать метод, а не индексацию массивов, то jQuery определяет для этой цели метод `get()`.

Синтаксис метода: get

`get([index])`

Получает одно или все соответствующие элементы в наборе. Если параметр не указан, то все элементы в объекте jQuery будут возвращены как массив JavaScript. Если предоставляется параметр `index`, то возвращается индексированный элемент. `index` может быть отрицательным, в этом случае отсчет будет осуществляться от конца соответствующего набора.

Параметры

`index` (Число) Индекс отдельного элемента, который будет возвращаться. Если он не указан, то весь набор будет возвращен как массив. Если задано отрицательное число, то отсчет идет с конца набора. Если индекс вне границ массива, то есть больше или равен количеству элементов, то метод вернет значение `undefined`.

Возвращает

Элемент DOM, или массив элементов DOM, или `undefined`.

Фрагмент:

```
var imgElement = $('img[alt]').get(0);
```

эквивалентен предыдущему примеру, где использовалась индексация массивов.

Метод `get()` также принимает отрицательный индекс. `get(-1)` возвращает последний элемент в наборе, `get (-2)` — предпоследний и т. д. Вдобавок к получению одного элемента `get()` также может вернуть массив всех элементов в наборе, если он применяется без параметра.

Иногда может понадобиться объект jQuery, содержащий конкретный элемент, а не этот элемент сам по себе. Довольно странно (хотя синтаксически корректно) выглядела бы запись, подобная следующей:

```
$($('p').get(2))
```

Для этой цели у jQuery есть метод `eq()`. Он имитирует действие фильтра селектора `:eq()`, который мы рассматривали в предыдущей главе. Чтобы увидеть их различия в условиях кода, предположим, что вы хотите выбрать второй элемент в наборе, содержащий все `<div>` на странице. Вот как можно выполнить эту задачу:

```
var $secondDiv = $('div').eq(1); var $secondDiv = $('div:eq(1)');
```

Выбор с использованием метода eq()

Выбор с использованием фильтра :eq()

Разница между операторами минимальна, но для большей производительности (подробнее будем рассматривать в главе 15) лучше придерживаться первой формы (метод `eq()`). Как правило, мы предлагаем пропускать методы через фильтры, так как это обычно приводит к улучшению продуктивности.

Теперь, когда мы отметили разницу между методом и фильтром, детально рассмотрим первый.

Синтаксис метода: eq

`eq(index)`

Получает индексированный элемент в наборе и возвращает новый набор, содержащий только этот элемент.

Параметры

`index`	(Число) Индекс возвращаемого единичного элемента. Отрицательный индекс может быть указан для выбора элемента, начиная с конца набора.

Возвращает

Коллекцию jQuery, содержащую один элемент или ноль.

Получение первого элемента набора — настолько распространенная операция, что существует удобный метод, который делает ее еще проще, — метод `first()`.

Синтаксис метода: first

`first()`

Получает первый элемент в наборе и возвращает новый набор, содержащий только этот элемент. Если набор пустой — возвращается также пустой набор.

Параметры

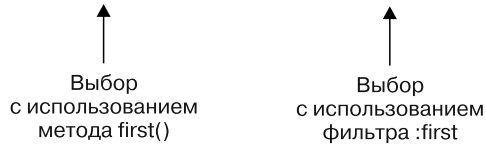
Отсутствуют.

Возвращает

Коллекцию jQuery, содержащую один элемент или ноль.

У метода `first()` есть аналог в фильтре `:first`. Мы хотим еще раз показать пример из двух вариантов. Если вы хотите получить первый абзац страницы, то можно написать одну из следующих инструкций:

```
var $firstPar = $('p').first(); var $firstPar = $('p:first');
```



Неудивительно, что разница с точки зрения кода минимальна, но метод `first()` здесь предпочтительнее, чем фильтр `:first`.

Как вы уже могли ожидать, существует соответствующий метод для получения и последнего элемента в наборе, который является аналогом фильтра `:last`.

Синтаксис метода: last

`last()`

Получает последний элемент в наборе и возвращает новый набор, содержащий только этот элемент. Если набор пустой — возвращается также пустой набор.

Параметры

Отсутствуют.

Возвращает

Коллекцию jQuery, содержащую один элемент или ноль.

Если вы хотите попрактиковаться с этими методами, то можете использовать страницу jQuery Operations Lab. Например, для получения первого элемента из списка, показанного на странице, можно написать:

```
$('li', '.my-list').first();
```

Теперь рассмотрим другой метод для получения массива элементов в наборе.

Получение всех элементов массива

Для получения всех элементов в объекте jQuery в виде JavaScript-массива из элементов DOM библиотека предоставляет метод `toArray()`.

Синтаксис метода: toArray

`toArray()`

Возвращает элементы набора как массив элементов DOM.

Параметры

Отсутствуют.

Возвращает

JavaScript-массив элементов DOM в этом наборе.

Рассмотрим такой пример:

```
var allLabeledButtons = $('label + button').toArray();
```

Команда собирает все `<button>` на странице, которым непосредственно предшествует `<label>`, в объект jQuery, а затем создает JavaScript-массив этих элементов, после чего присваивает его переменной `allLabeledButtons`.

Поиск индекса элементов

Притом что метод `get()` находит элемент по индексу, можно использовать обратную операцию `index()` для поиска индекса определенного элемента в наборе. Синтаксис метода `index()` выглядит следующим образом.

Синтаксис метода: index

`index([element])`

Находит указанный элемент в наборе и возвращает его порядковый индекс в наборе или находит порядковый индекс первого элемента в наборе среди других элементов того же уровня. Если элемент не найден, то возвращается значение `-1`.

Параметры

`element` (Селектор|Элемент|jQuery) Строка, содержащая селектор, ссылку на элемент или объект jQuery, порядковый номер которого необходимо определить. Если дается объект jQuery, то осуществляется поиск первого элемента в наборе. Непредоставление аргумента влечет возвращение индекса первого элемента в наборе в пределах списка родственных элементов одного уровня.

Возвращает

Порядковый номер указываемого элемента в наборе или его родственных элементов одного уровня или `-1`, если он не найден.

Чтобы помочь понять этот метод, предположим, что у вас есть следующий HTML-код:

```
<ul id="main-menu">
  <li id="home-link"><a href="/">Главная страница</a></li>
  <li id="projects-link"><a href="/projects">Проекты</a></li>
  <li id="blog-link"><a href="/blog">Блог</a></li>
  <li id="about-link"><a href="/about">О нас</a></li>
</ul>
```

Допустим, вы хотите узнать порядковый номер элемента списка (`<li>`), содержащий ссылку на блог, которым является элемент, имеющий идентификатор `blog-link`, в неупорядоченном списке с идентификатором `main-menu`.

ПРИМЕЧАНИЕ

Заполнять страницы таким количеством идентификаторов — не лучшая практика: в больших приложениях ими тяжело управлять и трудно дать гарантию, что там не будет дубликатов. Мы использовали их ради примера.

Можно получить это значение таким образом:

```
var index = $('#main-menu > li').index($('#blog-link'));
```

Основываясь на уже полученных знаниях о параметрах, принятых по методу `index()`, эту команду можно написать еще и так:

```
var index = $('#main-menu > li').index(document.getElementById('blog-link'));
```

Помните: индекс начинается с нуля. Индекс первого элемента — 0, второго — 1 и т. д. Таким образом, значение, которое вы получите, равняется `2`, потому что элемент является третьим в списке. Этот код доступен в файле `chapter-3/jquery.index.html`, предоставленном с книгой, а также в JS Bin (http://jsbin.com/notice/edit?html,js,console).

Метод `index()` вполне можно использовать для поиска индекса элемента в пределах его родительского элемента (то есть среди сестринских элементов). Посмотрим, что это значит, чтобы лучше понять. Родителем элемента списка с идентификатором `blog-link` является неупорядоченный список `main-menu`. Сестринскими будут элементы одного с `blog-link` уровня в дереве DOM, у которых один и тот же общий предок (неупорядоченный список). Учитывая нашу разметку, этими элементами являются все остальные элементы списка. Ссылки исключены, так как находятся внутри идентификатора `main-menu`, но не на одном и том же уровне, что и `blog-link`. Написание:

```
var index = $('#blog-link').index();
```

установит `index` опять к `2`.

Чтобы понять, чем интересен вызов `index()` без параметра, рассмотрим следующую разметку:

```
<div id="container">
  <p>Это текст</p>
  <img src="image.jpg" />
  <a href="/">Главная страница</a>
  <img src="anotherImage.jpg" />
  <p>Другой текст</p>
</div>
```

На этот раз разметка содержит несколько различных элементов. Допустим, вы хотите узнать порядковый номер первого элемента `img` в пределах его родительского элемента (`<div>` с идентификатором `container`). Вы можете написать:

```
var index = $('#container > img').index();
```

Значение `index` устанавливается как `1`, поскольку среди потомков `container` первый найденный `<img>` оказывается вторым элементом (он следует за `<p>`).

Помимо получения индекса элемента, библиотека дает возможность получить поднаборы набора, основанные на отношениях элементов коллекции jQuery к другим элементам в DOM. Посмотрим, как это сделать.

3.2.3. Получение наборов с использованием отношений

jQuery позволяет получить новые наборы из уже существующих на основе иерархических взаимоотношений элементов внутри DOM.

Допустим, у вас есть абзац, имеющий идентификатор `description`, и вы хотите узнать количество его предков, которые являются `<div>`. Основываясь на текущих знаниях о селекторах и методах, вы можете сказать, что это невозможно. На помощь приходит функция `parents()`. Рассмотрим следующий код:

```
var count = $('#description').parents('div').length;
```

Используя `parents()`, можно получить нужную информацию. Этот метод возвращает предков каждого элемента в текущий набор соответствующих элементов (который состоит из единственного абзаца, имеющего `description` в качестве своего идентификатора). Вы можете задать фильтры для предшественников с помощью селектора, как показано в примере. Поскольку в вашей коллекции jQuery находится только один элемент (мы предполагаем, что он существует на вашей странице), последует ожидаемый результат.

Что делать, если нужно узнать количество потомков вашего гипотетического абзаца? Это можно легко выяснить с помощью селекторов:

```
var count = $('#description > *').length;
```

Но погодите! Вы применяете все тот же универсальный селектор, который мы настоятельно не рекомендовали в предыдущем разделе? К сожалению, да. С точки зрения производительности лучше всего задать ту же самую команду, используя метод `children()`, как в таком примере:

```
var count = $('#description').children().length;
```

Однако этот метод не возвращает текстовые узлы. Что здесь можно сделать?

Для ситуаций, когда вы должны работать с текстовыми узлами, можно использовать метод `contents()`. Этот метод и `children()` отличаются тем, что первый из них не принимает никаких параметров. Возвращаясь к нашему примеру подсчета, вы можете написать:

```
var count = $('#description').contents().length;
```

Вы знаете, что подсчет элементов сам по себе не приносит большой пользы, и мы понимаем, что вам не терпится погрузиться в создание удивительных эффектов с помощью jQuery. Просим вас подождать еще чуть-чуть, пока мы не предоставим все необходимые сведения.

Метод `find()`, вероятно, один из наиболее часто используемых. Он позволяет осуществлять поиск через потомков элементов (с помощью поиска в глубину) в наборе и возвращает новый объект jQuery. Этот объект содержит все элементы,

которые соответствуют выражению селектора. Например, на основе набора соответствующих элементов в переменной `$set` можно получить еще один набор jQuery всех цитат (`<cite>`) в пределах абзацев (`<p>`), являющихся потомками элементов в исходном наборе:

```
$set.find('p cite');
```

Как и в случае со многими другими методами, действенность метода `find()` проявляется в случае применения в цепочке операций jQuery. Этот метод может пригодиться, если нужно ограничить поиск элементов-потомков в середине цепи методов jQuery, где нельзя использовать какой-либо другой контекст или ограничивающий механизм.

Прежде чем перечислить все методы, относящиеся к этой категории, мы хотим показать другой пример. Представьте, что у вас есть следующий фрагмент кода HTML:

```
<ul>
  <li class="awesome">Первый</li>
  <li>Второй</li>
  <li class="useless">Третий</li>
  <li class="good">Четвертый</li>
  <li class="brilliant amazing">Пятый</li>
</ul>
```

Вы хотите выбрать все элементы одного уровня из списка, в котором есть класс `awesome`, вплоть до того, который содержит классы `brilliant` и `amazing` (это пятый по счету элемент), но не включая его. Для решения задачи можно использовать метод `nextUntil()`. Он принимает селектор в качестве первого аргумента и извлекает все последующие сестринские элементы в наборе, пока не достигнет элемента, соответствующего данному селектору. Таким образом, можно написать:

```
var $listItems = $('.awesome').nextUntil('.brilliant.amazing');
```

Что делать, если вы хотите выполнить ту же операцию, но получить только элементы, содержащие класс `good`? Функция принимает необязательный второй аргумент `filter`, который позволяет достичь желаемого. Можно обновить предыдущую команду, и в результате получится:

```
var $listItems = $('.awesome').nextUntil('.brilliant.amazing', '.good');
```

Вы можете решить данный пример в браузере, загрузив файл `chapter-3/jquery.nextuntil.html` или получив соответствующий JS Bin (http://jsbin.com/fuhen/edit?html,js,console).

В табл. 3.1 представлены этот и другие методы, принадлежащие данной категории и позволяющие получить новый объект jQuery из существующего. Большинство методов принимают необязательный аргумент, который будет в соответствии с общепринятыми обозначениями заключен в квадратные скобки.

Таблица 3.1. Методы получения нового набора, базируясь на отношения к другому HTML DOM

Метод	Описание
children([selector])	Возвращает набор, состоящий из всех дочерних элементов в наборе, дополнительно отфильтрованных по селектору
closest(selector[, context])	Возвращает набор, содержащий ближайшего предка каждого элемента в наборе, который соответствует указанному селектору, начиная от элемента самого по себе. В качестве первого аргумента элемент или объект jQuery также может быть передан. В этом случае он будет проверяться от предков. Возвращается содержащий его набор, если он найден; в противном случае будет возвращен пустой массив. Элемент DOM может быть дополнительно указан как context. В таком случае для совпадения предок должен быть также потомком этого элемента
contents()	Возвращает набор содержимого элементов в этом наборе, который может также включать текстовые узлы
find(selector)	Возвращает набор потомков каждого элемента в наборе, отфильтрованный по данному селектору, объекту jQuery или элементу
next([selector])	Возвращает набор, содержащий следующий элемент того же уровня для каждого элемента в наборе соответствующих элементов. Если элемент является элементом одного уровня для более чем одного элемента, то берется только один раз. Если предоставляется селектор, то он берет следующий элемент того же уровня только в случае их соответствия этому селектору
nextAll([selector])	Возвращает набор, содержащий все последующие элементы того же уровня для элементов в наборе. Если предоставляется селектор, то он берет элементы только в случае их соответствия этому селектору
nextUntil([selector[, filter]])	Возвращает набор всех последующих элементов того же уровня в наборе вплоть до элемента, соответствующего селектору, но не включая его. Если нет соответствий селектору или он пропущен, то все последующие элементы одного уровня выбираются. В этом случае selector может быть строкой, содержащей выражение, узел DOM или объект jQuery. Метод дополнительно принимает другое выражение селектора, filter, как второй аргумент. Если он предоставлен, то элементы будут отфильтрованы по соответствию ему
offsetParent()	Возвращает набор, содержащий ближайшего относительно, абсолютно или фиксированно позиционированного (в понимании CSS) предка элементов в наборе
parent([selector])	Возвращает набор, состоящий из прямого предка всех элементов в наборе. Если элемент является родителем для более чем одного элемента, то берется только однажды. Если предоставляется селектор, то родители собираются лишь в случае соответствия ему
parents([selector])	Возвращает набор, состоящий из уникальных предков (элемент выбирается только один раз, даже если соответствует несколько раз) всех элементов в коллекции. Это включает прямых родителей, равно как и оставшихся предков, все время вплоть до корня документа. Если предоставляется селектор, то предки собираются только в случае соответствия ему

Продолжение ⇩

Таблица 3.1. (продолжение)

Метод	Описание
parentsUntil([selector[, filter]])	Возвращает набор из всех предков элементов в коллекции, вплоть до элемента, соответствующего селектору, но не включая его. Если селектору не найдены соответствия или он не предоставлен, то выбираются все предки. В этом случае selector может быть строкой, содержащей выражение селектора, узлом DOM или объектом jQuery. Метод дополнительно принимает другое выражение селектора, filter, как второй аргумент. Если он предоставлен, то элементы будут отфильтрованы по соответствию ему
prev([selector])	Возвращает набор, состоящий из ближайших предыдущих элементов одного уровня для каждого элемента в наборе соответствующих элементов. Если элемент является элементом одного уровня для более чем одного элемента, то берется только один раз. Если предоставляется селектор, то он берет предыдущий элемент одного уровня только в случае его соответствия селектору
prevAll([selector])	Возвращает набор, содержащий все предыдущие элементы одного уровня для элементов в наборе. Элементы могут быть дополнительно отфильтрованы по селектору
prevUntil([selector[, filter]])	Возвращает набор всех предшествующих элементов того же уровня в коллекции, вплоть до элемента, соответствующего селектору, но не включая его. Если селектору нет соответствия или он не указан, то выбираются все предыдущие элементы того же уровня. В этом случае selector может быть строкой, содержащей выражение селектора, узел DOM или объект jQuery. Метод дополнительно принимает другое выражение селектора, filter, как второй аргумент. Если он предоставлен, то элементы будут отфильтрованы по соответствию ему
siblings([selector])	Возвращает набор, состоящий из всех элементов того же уровня для элементов в наборе, взятых только один раз. Элементы могут быть дополнительно отфильтрованы по селектору

Теперь, когда мы описали все методы, рассмотрим на конкретном примере некоторые из них.

Представим ситуацию, когда обработчик события кнопки (об этом подробно расскажем в главе 6) запускается элементом кнопки, на который ссылается ключевое слово `this` в обработчике. Такая ситуация возникает, когда вы хотите выполнить код JavaScript (например, вычисление или вызов Ajax) при нажатой кнопке. Кроме того, допустим, вы хотите найти блок `<div>`, в котором определена данная кнопка. Метод `closest()` делает это невероятно легко:

```
$(this).closest('div');
```

Но команда найдет только самого ближайшего предка `<div>`; а если нужный вам `<div>` находится гораздо выше в дереве предков? Не проблема. Вы можете усовершенствовать селектор, который передаете `closest()`, чтобы различить, какой элемент выбран:

```
$(this).closest('div.my-container');
```

Теперь будет выбран первый предок `<div>` с классом `my-container`.

Все остальные методы работают таким же образом. Возьмем, например, ситуацию, когда нужно найти кнопку одного уровня с определенным атрибутом `title`:

```
$(this).siblings('button[title="Закрыть"]');
```

То есть здесь происходит извлечение всех элементов одного уровня, которые являются `<button>` и имеют `title` со значением `"Закрыть"`. Если вы хотите убедиться, что извлекается только первый элемент одного уровня, то можете использовать метод `first()`, описанный в данной главе:

```
$(this).siblings('button[title="Закрыть"]').first();
```

Эти методы позволяют вам достаточно свободно выбрать элементы из DOM на основании их отношения к другим элементам DOM. Как вы будете настраивать набор элементов, которые находятся в коллекции jQuery?

3.2.4. Вырезки и перетасовки набора

Когда у вас есть набор, вы можете его расширить, добавляя к нему элементы, или уменьшить до поднабора изначально соответствующих элементов. jQuery предлагает большую коллекцию методов для управления набором. Прежде всего, посмотрим, как добавлять элементы.

Добавление элементов в набор

Предположим, нужно добавить больше элементов к существующей коллекции jQuery. Такая возможность есть, и она будет более полезна, если вы захотите добавить дополнительные элементы после применения какого-либо метода к исходному набору параметров. Помните, что цепочка jQuery позволяет выполнить огромный объем работы в одной инструкции.

Ниже мы рассмотрим некоторые конкретные примеры таких ситуаций, но сначала начнем с более простого сценария. Предположим, вы хотите найти соответствие всем `<img>`, содержащим атрибут либо `alt`, либо `title`. Многофункциональные селекторы jQuery позволяют выразить это как один селектор, например:

```
$('img[alt], img[title]');
```

Но для иллюстрации работы метода `add()` можно сопоставить элементы с:

```
$('img[alt]').add('img[title]');
```

Использование данного метода таким образом позволит вам построить цепь из ряда селекторов, объединяя элементы, которые подойдут любому из селекторов.

Методы, подобные `add()`, весьма значимы (и более гибки, чем сложные селекторы) в рамках цепочек методов jQuery, так как не увеличивают первоначальный набор, а создают новый с результатом. Вскоре вы увидите, что это может быть чрезвычайно полезным в сочетании с такими методами, как `end()` (мы рассмотрим его

в подразделе 3.2.5), который может быть использован, чтобы избежать операций, увеличивающих оригинальные наборы.

Так выглядит синтаксис метода `add()`.

Синтаксис метода: add

`add(selector[, context])`

Создает новый объект jQuery и добавляет к нему элементы, определенные по параметру `selector`, которым может быть строка, содержащая селектор, HTML-фрагмент, элемент DOM, массив элементов DOM или объект jQuery.

Параметры

`selector` (Селектор|Элемент|Массив|jQuery) Определяет, что должно быть добавлено в набор. Это может быть селектор; в таком случае любые соответствующие элементы добавляются в набор. Если параметром является HTML-фрагмент, то соответствующие элементы будут созданы и добавлены в набор. Если это элемент DOM, то он добавляется в набор. Если это массив элементов DOM или объект jQuery, то все содержащиеся в них элементы добавляются в набор.

`context` (Селектор|Элемент|jQuery) Определяет контекст для ограничения поиска элементов, соответствующих первому параметру. Является тем же параметром, который может быть передан функции `jQuery()`.

Возвращает

Копию оригинального набора с дополнительными элементами.

Откройте страницу jQuery Operations Lab в вашем браузере и введите следующее выражение:

```
$('td').add('th');
```

Затем нажмите кнопку Execute (Выполнить). Команда выберет все ячейки таблицы. На рис. 3.4 показан снимок экрана с результатом.

Как видите, все ячейки таблицы, в том числе заголовки (`<th>`), были добавлены к набору. Выбранные элементы обозначены черным контуром и серым фоном. Этот стиль присваивается каждому элементу, к которому автоматически добавляется класс `found-element`.

Теперь взглянем на более реалистичное использование метода `add()`. Допустим, мы хотим применить красную границу ко всем `<img>`, у которых есть атрибут `alt`, добавив к ним класс `red-border`. Затем мы применим уровень прозрачности для всех элементов `img`, имеющих атрибут `alt` или `title`, добавив класс `opaque`. Оператор запятой (`,`) из селекторов CSS не сможет помочь, поскольку нужно применить операцию к набору, а затем добавить к нему дополнительные элементы перед применением другой операции. С этим можно легко справиться с помощью нескольких инструкций. Но эффективнее и элегантнее будет использовать мощность цепочки jQuery, чтобы выполнить эту задачу в одном выражении. Для добавления упомянутых классов следует задействовать функцию jQuery `addClass()`. В своей простейшей форме она принимает имя класса в качестве аргумента и добавляет его в элементы в наборе.

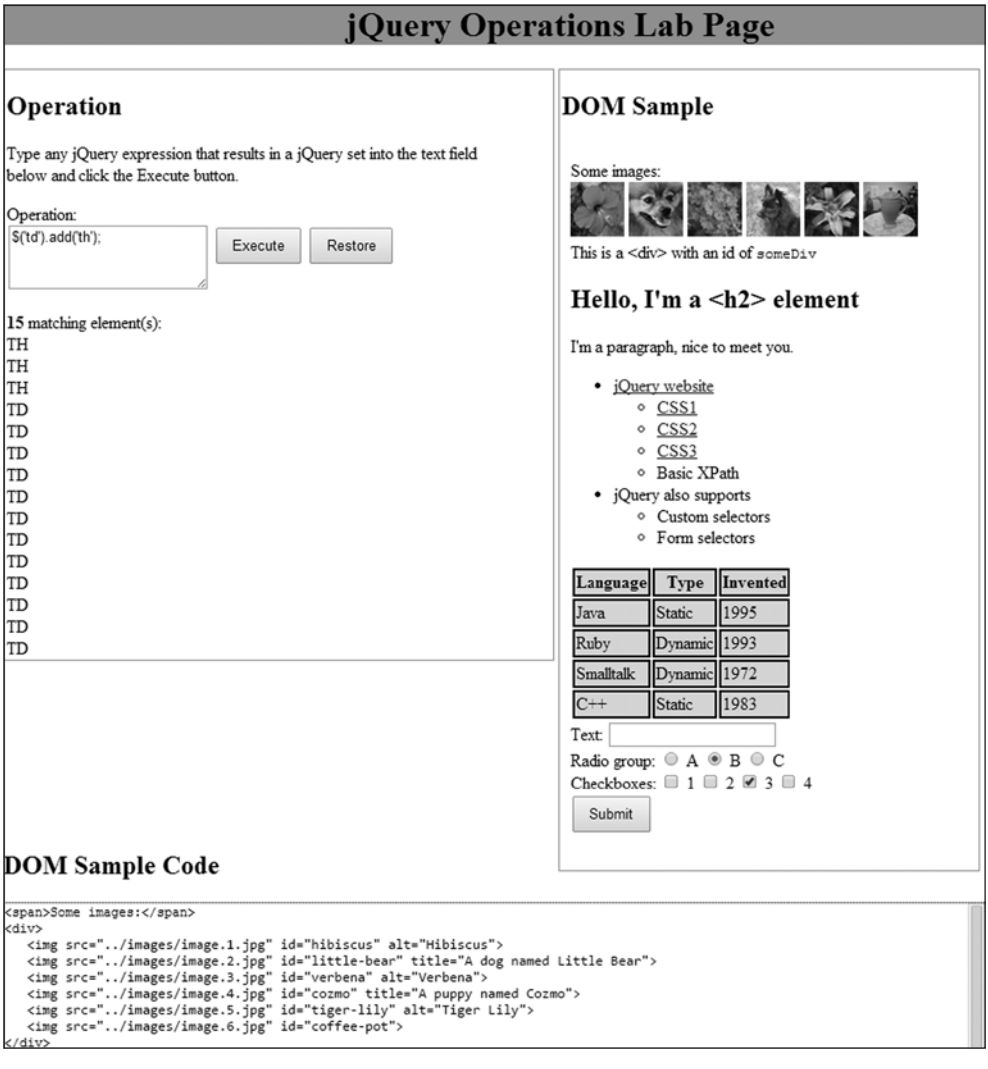


Рис. 3.4. Ячейки таблицы были сопоставлены с помощью выражения jQuery

Результат этого в одном объявлении выглядит так:

```
$('img[alt]')
  .addClass('red-border')
  .add('img[title]')
  .addClass('opaque');
```

Здесь вы создаете набор всех `<img>` с атрибутом `alt` и применяете предопределенный класс, который добавляет красную границу. Затем вы добавляете элементы `img`, содержащие атрибут `title`, и наконец, применяете класс, который устанавливает уровень прозрачности по отношению к вновь дополненному набору.

Введите эту инструкцию на странице jQuery Operations Lab (где предопределены упомянутые классы), нажмите кнопку Execute (Выполнить) и просмотрите результаты, показанные на рис. 3.5.

Рис. 3.5. Цепочка действий jQuery позволит вам осуществить сложные операции в одной инструкции

Вы можете увидеть, что у изображений цветов (тех, которые с `alt`) есть красная граница. Кроме того, все изображения, кроме кофейного чайника, единственного изображения, не содержащего ни `alt`, ни `title`, — тусклые в результате применения правила непрозрачности. Как вы, вероятно, заметили, изображения, не являющиеся цветами (и исключая кофейник), также имеют черную границу. Причина в том, что ранее упомянутый класс `found-element` был автоматически добавлен к классу `opaque`. На самом деле `found-element` был добавлен во *все* выбранные изображения, но черная граница для изображений цветов была отменена объявлениями стиля класса `red-border`.

Метод `add()` можно также использовать для добавления элементов к существующему набору, учитывая прямые ссылки на них. Передача ссылки элемента или массива ссылок элементов методу `add()` добавляет элементы к набору. Если у вас была ссылка элемента в переменной с именем `someElement`, то можете добавить ее в набор всех изображений, содержащий свойство `alt`, с помощью такой инструкции:

```
$('img[alt]').add(someElement);
```

Если этого недостаточно, метод `add()` позволяет добавлять не только существующие элементы в набор, но и новые элементы, передавая ему строку, содержащую разметку HTML. Рассмотрим такой пример:

```
$('p').add('<div>Всем привет!</div>');
```

Фрагмент создает сначала набор всех элементов `p` в документе, а затем новый набор, который включает в себя динамически создаваемый `<div>`.

Метод `add()` — простой и эффективный инструмент для дополнения набора, но сейчас посмотрим на методы jQuery, которые позволяют удалять из него элементы.

Оттачивание содержимого набора

Вы видели, что дополнить объект jQuery за счет нескольких селекторов, соединенных друг с другом с помощью метода `add()`, довольно просто. Помимо этого, можно объединить в цепочку селекторы, чтобы сформировать *исключающие* отношения, используя метод `not()`. Он похож на селектор фильтра `:not`, который мы обсуждали в предыдущей главе, но может быть применен аналогично методу `add()` для удаления элементов из набора в любом месте в пределах цепочки методов jQuery.

Предположим, вы хотите выбрать все элементы `img` страницы, у которых есть атрибут `title`, кроме содержащих текст `"щенок"` в качестве их значения. Это можно сделать одним селектором, который выражает данное условие (а именно, `img[title]:not([title*="щенок"])`), но для наглядности представим, что вы забыли о фильтре `:not`. Используя метод `not()`, удаляющий все элементы из набора, которые соответствуют указанному селектору, можно выразить тип *исключающих* отношений. Для этого нужно написать:

```
$('img[title]').not('[title*="щенок"]');
```

Введите это выражение на странице jQuery Operations Lab и выполните его. Вы увидите, что было выделено только первое изображение собаки. Черный щенок, включенный в исходный набор, обладает атрибутом `title` и удаляется вызовом `not()`, так как его `title` содержит текст `"щенок"`.

Синтаксис метода: not

`not(selector)`

Создает копию набора без элементов, которые соответствуют критериям, указанным в значении параметра `selector`.

Параметры

`selector` (Селектор|Элемент|Массив|jQuery|Функция) Определяет, какие элементы должны быть удалены. Если параметром является селектор jQuery, то соответствующие элементы удаляются. Если переданы ссылка на элемент, массив элементов или набор jQuery, то эти элементы удаляются из набора. Если передана функция, то она вызывается для каждого элемента в наборе (с `this`, установленным к элементу), и если в результате вызова возвращается `true`, то элемент будет удален из набора. Вдобавок jQuery передает индекс элемента внутри набора как первый аргумент функции и текущий элемент — как второй.

Возвращает

Копию первоначального набора с удаленными элементами.

Метод `not()` можно применять для удаления отдельных элементов из набора, пропуская ссылки на элемент или массив ссылок на элементы. Последнее особенно интересно и способно быть мощным подспорьем в работе: как вы помните, любой набор jQuery может использоваться как массив ссылок на элементы.

Когда необходима максимальная гибкость, можно передать функцию `not()` и принять решение о том, следует ли сохранить или удалить элемент. jQuery передает функции аргумент, определяющий индекс элемента в наборе. Рассмотрим следующий пример:

```
$('div').not(function(index) {
  return $(this).children().length > 2 && index % 2 === 0;
});
```

Эта инструкция выберет все `<div>` на странице, а затем удалит те, которые имеют более чем двух потомков и нечетный индекс (не позицию) внутри набора. Если хотите увидеть фактический результат, то можете попробовать выполнить те же действия на лабораторной странице и получите четыре соответствия.

Данный метод позволяет фильтровать набор способами, которые трудно или невозможно выразить выражением селектора, прибегая к программной фильтрации набора элементов.

Для тех случаев, когда тест, примененный в рамках функции, передающейся в `not()`, дает результат, абсолютно противоположный нужному вам, у `not()` есть обратный метод `filter()`. Он действует аналогично `not()`, за исключением того, что удаляет элементы, когда функция возвращает значение `false`.

Допустим, вы хотите создать набор всех `<td>`, которые содержат положительное целое значение. Для таких ситуаций можно использовать метод `filter()` следующим образом:

```
$('td').filter(function() {
  return this.innerHTML.match(/^\d+$/);
});
```

Эта инструкция создает набор из всех элементов `td`, а затем вызывает функцию, передаваемую методу `filter()` для каждого из них, с текущим соответствующим элементом в виде значения `this` для вызова. Функция передает регулярное выражение для определения соответствия содержания элемента описанному шаблону (последовательности одной или более цифр), возвращая `null`, если соответствия нет. Элементы, для которых вызванная функция фильтра возвращает `false` или *ложное* значение в целом (`null`, `undefined` и т. д.), не включаются в возвращаемый набор.

Синтаксис метода `filter()` следующий.

Синтаксис метода: filter

`filter(selector)`

Создает копию набора и удаляет из него элементы, не соответствующие критериям, указанным в значении параметра `selector`.

Параметры

`selector` (Селектор|Элемент|Массив|jQuery|Функция) Определяет, какие элементы должны быть удалены. Если параметром является строка, содержащая селектор, то не соответствующие ему элементы удаляются. Если переданы ссылка на элемент, массив элементов или объект jQuery, то все элементы, кроме этих, удаляются из набора. Если передана функция, то она вызывается для каждого элемента в наборе (с `this`, установленным к элементу), и если в результате вызова возвращается `false`, то элемент будет удален из набора. Вдобавок jQuery передает индекс элемента внутри набора как первый аргумент функции и текущий элемент — как второй.

Возвращает

Копию первоначального набора без удаленных элементов.

Снова вернитесь к странице jQuery Operations Lab, введите предыдущее выражение и выполните его. Вы увидите, что ячейки таблицы для колонки «Разработан» — единственные элементы `td`, оставшиеся выбранными в конечном счете.

Метод `filter()` может быть использован с выражением селектора. Если его применять таким образом, то он действует противоположно соответствующему методу `not()`, удаляя любые элементы, которые не относятся к переданному селектору. Этот метод не является супердейственным (как правило, проще в первую очередь

прибегнуть к более ограничивающему селектору), но может пригодиться в цепочке методов jQuery. Рассмотрим такой пример:

```
$('img')
  .addClass('opaque')
  .filter('[title*="dog"]')
  .addClass('red-border');
```

Эта инструкция-цепочка выбирает все изображения страницы, применяет к ним класс `opaque`, а затем уменьшает набор только до тех элементов изображения, у которых атрибут `title` содержит строку `dog` перед применением другого класса — `red-border`. В результате все изображения собак становятся полупрозрачными, но только изображение кобеля обведено красной границей.

Методы `not()` и `filter()` дают вам замечательную возможность легко корректировать набор элементов в коллекции на основе практически любых критериев, касающихся элементов набора. Но можно также создать поднабор на основе положения элементов в пределах набора. Посмотрим на методы, позволяющие это сделать.

Получение поднабора из набора

Иногда вы можете получить фрагмент набора на основе положения элементов в пределах набора. jQuery предоставляет для этого метод `slice()`. Он создает и возвращает новый набор из любой смежной части, среза или оригинального набора.

Синтаксис метода: slice

`slice(start[, end])`

Создает и возвращает новый набор, содержащий сопредельную часть соответствующего набора.

Параметры

`start`	(Число) Начинающаяся с нуля позиция первого элемента, которую следует включить в возвращаемую вырезку.
`end`	(Число) Необязательная начинающаяся с нуля позиция первого элемента, который не следует включать в возвращаемую вырезку, или следующая позиция после последнего элемента, который должен быть включен. Если значение отрицательное, то показывает сдвиг от конца набора. Если пропущено, то вырезка будет выполнена до конца набора.

Возвращает

Созданный набор.

Если вы хотите получить набор, содержащий один элемент из другого набора, на основе его позиции в исходном наборе, то можете использовать метод `slice()`. Например, для получения третьего элемента предыдущего набора можно написать:

```
$('img, div.wrapper', 'div').slice(2, 3);
```

Это объявление выбирает все `<img>` и `<div>`, у которых есть класс `wrapper`, в пределах `<div>`, а затем создает новый набор, содержащий только третий элемент в соответствующем наборе. Как видите, знания, обретенные вами по ходу чтения книги, становятся нужны снова и снова по мере продвижения вперед.

Обратите внимание: метод `slice()` отличается от `get(2)`, который возвращает третий элемент DOM в наборе, но действует так же, как `eq(2)`.

Объявление:

```
$('*').slice(0, 4);
```

выбирает все элементы на странице, а затем создает набор, содержащий первые четыре элемента.

Чтобы захватить элементы вплоть до конца набора, инструкция:

```
$('*').slice(4);
```

соответствует всем элементам на странице, а затем возвращает набор, содержащий все элементы, кроме первых четырех.

Еще один метод, подходящий для получения поднабора, называется `has()`. Как и фильтр `:has`, метод проверяет потомков элементов в объекте jQuery, используя эту проверку, чтобы выбрать элементы, которые станут частью поднабора.

Синтаксис метода: has

`has(selector)`

Создает и возвращает новый набор, содержащий только те элементы, которые содержат потомков, соответствующих переданному выражению `selector`.

Параметры

`selector` (Селектор|Элемент) Строка, содержащая селектор, который следует применить ко всем потомкам элементов в наборе, или к проверяемому элементу DOM. В возвращаемый набор включаются только элементы, у которых есть потомок, соответствующий селектору или переданному элементу.

Возвращает

Объект jQuery.

Например, рассмотрим следующую строку:

```
$('div').has('img[alt]');
```

Это выражение собирает все `<div>`, а затем создает и возвращает второй набор, который содержит только `<div>`, включающие по меньшей мере одного потомка `<img>`, обладающего атрибутом `alt`.

Преобразование элементов набора

Часто бывает необходимо выполнять преобразования элементов набора. Например, можно собрать все идентификаторы элементов в наборе или, возможно, собрать значения из набора элементов `form` для того, чтобы создать из них строку запроса. Для таких случаев будет удобен метод `map()`.

Например, следующий код соберет все идентификаторы всех `<div>` на странице:

```
var $allIDs = $('div').map(function() {
  return this.id;
});
```

Синтаксис метода: map

`map(callback)`

Вызывает функцию `callback` (обратный вызов) для каждого элемента в наборе и собирает возвращаемые значения в объект jQuery.

Параметры

`callback` (Функция) Функция обратного вызова, которая вызывается для каждого элемента в наборе. Ей передаются два параметра: начинающийся с нуля индекс элемента в наборе и сам элемент. Элемент также установлен как контекст функции (ключевое слово `this`). Для добавления элемента в новый набор должно вернуться значение, отличающееся от `null` или `undefined`.

Возвращает

Набор преобразованных значений.

С помощью этой инструкции вы извлекаете объект jQuery, содержащий идентификаторы. Как правило, они не являются теми, которые вы хотите. Если вы хотите работать с простым массивом JavaScript, то можете добавить метод `toArray()` к цепочке, например, так:

```
var allIDs = $('div').map(function() {
  return this.id;
})
.toArray();
```

Есть и другие методы, о которых мы хотим рассказать в этой главе. Узнаем о них больше.

Перемещение элементов набора

Метод `map()` полезен для итерации по элементам набора, когда нужно собрать значения или преобразовать элементы каким-либо другим способом. Но, скорее всего, в ряде случаев перебор элементов потребуется для более общих целей. Здесь будет просто незаменим метод jQuery `each()`.

Синтаксис метода: each

`each(iterator)`

Перемещает все элементы в соответствующем наборе, вызывая переданную функцию `iterator` для каждого из них.

Параметры

`iterator` (Функция) Функция, вызываемая для каждого элемента в соответствующем наборе. Ей передается два параметра: начинающийся с нуля индекс элемента в наборе и сам элемент. Элемент также устанавливается как функциональный контекст (ссылка `this`).

Возвращает

Коллекцию jQuery.

Примером использования метода `each()` может служить установка значений свойств всех элементов в соответствующем наборе. Рассмотрим следующий пример:

```
$('img').each(function(i){
  this.alt = 'Это изображение[' + i + '] с id ' + this.id;
});
```

Эта инструкция будет вызывать переданную функцию для каждого элемента `img` на странице, изменяя его свойство `alt`, используя индекс элемента в наборе и его ID.

Вы уже увидели немало методов, которые можно применять при манипуляциях с объектом jQuery, но мы не закончили!

3.2.5. Еще больше способов использования набора

Библиотека предоставляет еще несколько замечательных способов, которые позволят вам усовершенствовать свою коллекцию объектов.

Следующий метод, который мы рассмотрим, позволяет тестировать набор, чтобы проверить, содержит ли он хотя бы один элемент, соответствующий данному выражению селектора. Метод `is()` возвращает значение `true`, если хотя бы один элемент соответствует селектору, и `false` — в противном случае. Взгляните на пример:

```
var hasImage = $('*').is('img');
```

Это утверждение устанавливает значение переменной `hasImage` в `true`, если текущая страница имеет по меньшей мере одно изображение.

Синтаксис метода: is

`is(selector)`

Определяет, есть ли в наборе элемент, соответствующий выражению селектора.

Параметры

`selector` (Селектор|Элемент|Массив|jQuery|Функция) Выражение селектора, элемент, массив элементов или объект jQuery, используемый для проверки элементов в наборе. Если передана функция, то она вызывается для каждого элемента в коллекции jQuery (с `this`, установленным к элементу). Если в результате вызова возвращается `true`, то сама функция возвращает `true`. Вдобавок jQuery передает индекс элемента внутри набора как первый аргумент функции и текущий элемент — как второй.

Возвращает

`true`, если хотя бы один элемент соответствует переданному селектору; в противном случае — `false`.

Это высокооптимизированная и быстрая команда в jQuery, которую без колебаний можно использовать там, где большое значение имеет производительность.

Мы проделали большую работу, изучая возможности методов цепочки jQuery, позволяющих совершить много операций с помощью одной инструкции, и будем продолжать изучение, потому что это *действительно* нечто особенное. Цепочка не только разрешает писать продуктивные операции в сжатой форме, но и повы-

шает эффективность работы, поскольку наборы не нужно пересчитывать, чтобы применить к ним несколько методов.

А сейчас рассмотрим следующую инструкцию:

```
$('img').filter('[title]').hide();
```

В ней создаются два набора: первоначальный набор всех `<img>` в DOM и второй набор, состоящий только из элементов с атрибутом `title`. (Да, вы могли бы сделать это с помощью одного селектора, но оставайтесь с нами, пока мы описываем нашу концепцию. Представьте, что вы делаете нечто важное в цепочке перед вызовом метода `filter()`.) Затем вы скрываете все элементы в наборе (имеющие атрибут `title`).

Но задумайтесь вот над чем: а если впоследствии вы захотите применить такой метод, как добавление имени класса, к исходному набору после того, как он отфильтруется? Вы не можете переставить его в конец существующей цепи; это затронет изображения с `title`, но не исходный набор изображений.

Для таких случаев jQuery предоставляет метод `end()`. При использовании в цепочке jQuery он восстановит предыдущую коллекцию и вернет ее в качестве своего значения, так что последующие операции будут применяться к этому предыдущему набору.

Рассмотрим, например, такой код:

```
$('img')
  .filter('[title]')
  .hide()
  .end()
  .addClass('my-class');
```

Метод `filter()` возвращает набор изображений с атрибутом `title`. Вызвав `end()`, вы вернетесь к предыдущему набору соответствующих элементов (первоначальному набору всех изображений), к которым будет применен метод `addClass()`. Без `end()` метод `addClass()` поработал бы только на наборе изображений с атрибутом `title`. Чтобы этого избежать, вы должны сохранить первоначальный набор в переменной и затем написать два объявления. Метод `end()` позволяет избавиться от такой переменной и выполнять все операции в одной инструкции. Синтаксис данного метода выглядит следующим образом.

Синтаксис метода: end

`end()`

Используется в цепочке методов jQuery; заканчивает последнюю операцию фильтрации в текущей цепочке и возвращает набор соответствующих элементов в его предыдущем состоянии.

Параметры

Отсутствуют.

Возвращает

Предыдущую коллекцию jQuery.

Объекты jQuery поддерживают внутренний стек, который отслеживает изменения в соответствующем наборе элементов. Когда вызывается метод, как показано только что, новый набор записывается в стек. После вызова метода `end()` вызывается верхний (самый недавний) набор из стека, делая таким образом предыдущий набор доступным для работы в последующем методе.

Еще одним удобным методом jQuery, изменяющим упомянутый стек, является `addBack()`, который добавляет предыдущий набор элементов в стеке к текущему набору, дополнительно отфильтрованному селектором.

Синтаксис метода: addBack

`addBack([selector])`

Добавляет в предыдущий набор элементов в стеке текущий набор, дополнительно отфильтрованный селектором.

Параметры

`selector` (Селектор) Строка, содержащая выражение селектора, по которому будет сопоставлен текущий набор элементов.

Возвращает

Объединенную коллекцию jQuery.

Рассмотрим следующее:

```
$('div')
  .addClass('my-class')
  .find('img')
  .addClass('red-border')
  .addBack()
  .addClass('opaque');
```

Оператор выбирает все элементы `div` на странице, добавляет к ним класс `my-class`, после чего создает новый набор, состоящий из всех элементов `img`, являющихся потомками этих элементов `div`. Затем он применяет к ним класс `red-border` и создает третий набор, который будет объединением элементов `div` (так как это был самый верхний набор в стеке) и их потомков, элементов `img`. И наконец, он применяет для них класс `opaque`.

Уф! В конце этого всего `<div>` получает классы `my-class` и `opaque`, тогда как изображения, являющиеся потомками этих элементов, — классы `red-border` и `opaque`.

Глава 3 должна была доказать вам, что освоение селекторов и методов работы с ними — чрезвычайно важный момент. Их возможности наряду с использованием инструментов для перемещения DOM позволяют точно выбирать элементы, независимо от сложности ваших требований. Теперь, когда у вас есть твердое понимание темы, можно углубиться в еще более интересные материалы.

3.3. Резюме

В данной главе описывается, как создать и увеличить набор соответствующих элементов с использованием HTML-фрагментов. Эти отдельные элементы могут быть задействованы наряду с любыми другими элементами в наборе и в конце концов прикреплены к части страницы документа.

Мы рассмотрели набор методов, с помощью которых можно привести в порядок набор, отшлифовать его содержимое — либо сразу после его создания, либо на пути к этому, используя цепочку методов. Применение фильтров к текущему набору облегчит и ускорит создание новой коллекции jQuery.

В общем и целом, jQuery предлагает множество инструментов для легкого и точного определения элементов страницы, которыми вы хотите манипулировать.

В этой главе мы охватили много вопросов, но на самом деле они не имели никакого отношения к элементам DOM страницы. Теперь, зная, как выбрать элементы, с которыми хотите работать, вы будете готовы начать оживлять свои страницы с помощью методов jQuery для манипулирования DOM.

4 Работа со свойствами, атрибутами и данными

В этой главе:

- чтение и запись атрибутов элементов;
- работа со свойствами элементов;
- хранение пользовательских данных элементов.

Каждый, кто впервые столкнулся с разработкой программного обеспечения, усвоил очень важный урок: даже огромное и сложное ПО состоит из сочетания элементарных инструкций.

Суммирование чисел, подсчет элементов, итерирование по элементам — вот лишь несколько примеров этих основных операций. Точно так же можно создать красивый визуальный ряд с помощью jQuery, оперируя атрибутами, свойствами, классами, стилями и т. д.

Вы можете манипулировать атрибутами и свойствами элементов, используя нативные функции JavaScript, но некоторые задачи не такие уж и простые, как хотелось бы. К тому же, применяя встроенные функции, можно столкнуться с несовместимостью браузеров. jQuery предоставляет полный набор методов для упрощения работы с атрибутами и свойствами, решая для вас все проблемы, связанные с несовместимостью.

Еще одним ключевым понятием при работе с элементами DOM и создании эффектов является возможность хранения пользовательских данных в элементах, с которыми вы работаете. jQuery позволяет сохранить состояние элемента в заданный момент времени. Такая возможность очень пригодится при создании плагинов, в чем вы убедитесь в части III.

В этой главе основное внимание уделяется множеству методов, предлагаемых jQuery для работы с атрибутами, свойствами и данными.

4.1. Определение свойств элементов и атрибутов

Самые основные компоненты элементов DOM, которыми можно манипулировать, — установленные для этих элементов свойства и атрибуты. Они изначально заданы экземплярам объектов JavaScript, представляющим элементы DOM в результате разбора разметки HTML; они могут также динамически изменяться под управлением сценария. Удостоверимся, что вы правильно понимаете терминологию и понятия.

Свойства — это объекты, присущие JavaScript, каждый из которых имеет имя и значение. Динамичный характер JavaScript позволяет создавать свойства на своих объектах под управлением сценария. (В приложении данная концепция представлена детально, если вы еще новичок в JavaScript.)

Говоря об *атрибутах*, мы имеем в виду значения, определенные в разметке элементов DOM, а не свойства экземпляра объекта. Рассмотрим следующую HTML-разметку для элемента изображения:

```
<img id="my-image" src="image.gif" alt="An image" class="some-class"
     title="Это — изображение" />
```

В разметке этого элемента именем тега является `img`, а разметка для `id`, `src`, `alt`, `class` и `title` представляет атрибуты элемента, каждый из которых состоит из имени и значения. Браузер читает и интерпретирует эту разметку элемента для создания экземпляра объекта JavaScript типа `HTMLElement`, представляющего данный элемент в DOM.

Первое различие между этими двумя понятиями состоит в том, что значения свойств могут отличаться от значений соответствующих атрибутов, в то время как последние всегда являются строками, булевыми значениями, числами или даже объектами. Например, попытка получить `tabindex` в качестве атрибута HTML даст вам строку (состоящую только из цифр, но все-таки строку). Но получение ее соответствующего свойства даст вам число. Еще одним примером является `style`, который при получении в качестве атрибута будет строкой, а в качестве свойства — объектом (типа `CSSStyleDeclaration`). Посмотрим на эту разницу на практике. Допустим, у вас есть следующий элемент HTML на веб-странице:

```
<input id="surname" tabindex="1" style="color:red; margin:2px;" />
```

Теперь создайте сценарий на той же странице из следующих операторов или введите их непосредственно в консоли браузера:

```
var element = document.getElementById('surname');
console.log(typeof element.getAttribute('tabindex'));
console.log(typeof element.tabIndex);
console.log(element.getAttribute('style'));
console.log(element.style);
```

Печатает "string"

Печатает "number"

Печатает строку "color:red; margin:2px;"

Печатает объект CSSStyleDeclaration, содержащий все стили, примененные к элементу

Все атрибуты элемента собраны в объекте, хранящемся как свойство с именем `attributes` (вполне обоснованным) на экземпляре элемента DOM. К тому же объект, представляющий элемент, наделен рядом свойств, в том числе тех, которые представляют атрибуты разметки элемента.

Таким образом, значения атрибутов сохраняются не только в свойстве `attributes`, но и в нескольких других. На рис. 4.1 показан упрощенный обзор этого процесса.

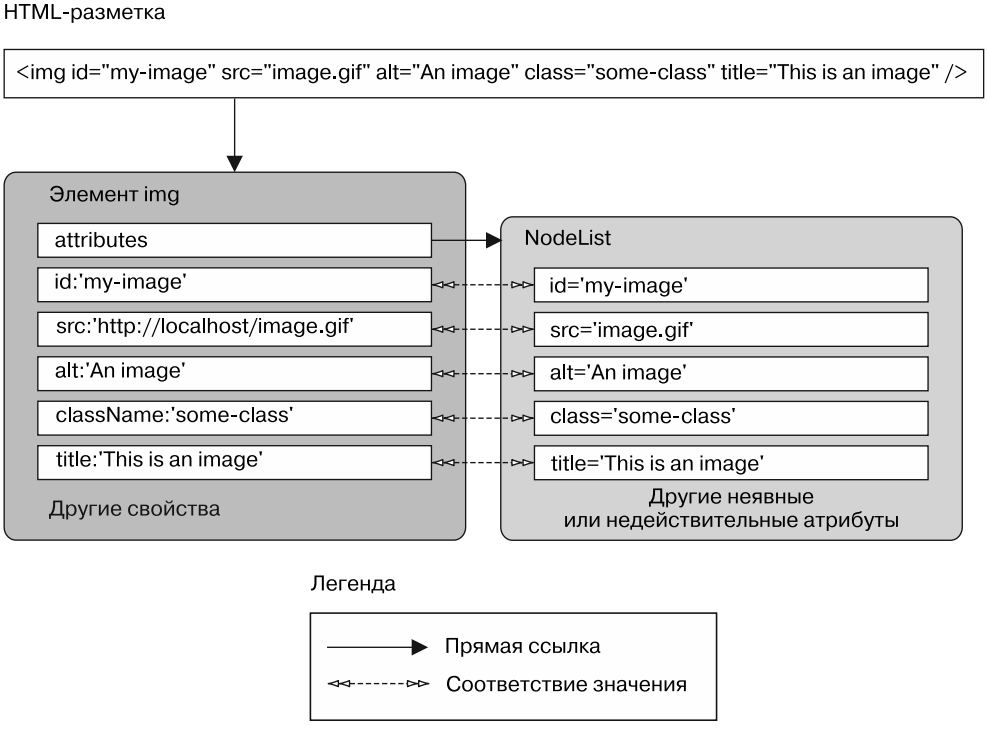


Рис. 4.1. HTML-разметка преобразована в элементы DOM, включая атрибуты тегов и созданных из них свойств. Браузер создает соответствие между атрибутами и свойствами элементов

Активное соединение остается между значениями атрибутов, хранящихся в объекте `attributes`, и соответствующих свойствах. Изменение значения атрибута обычно приводит к изменению соответствующего значения свойства, и наоборот. Для большей конкретики рассмотрим элемент `input` (`checkbox`, если быть точными) с дополнительным, нестандартным атрибутом (выделен жирным шрифтом):

```
<input type="checkbox" id="book" name="book" title="Проверьте!"
       book="jQuery в действии" />
```

Сохраняются следующие условия.

- Если атрибут существует в виде встроенного (нативного) свойства соответствующего объекта DOM, то значение синхронизируется. Например, `title` — это стандартный атрибут, который существует в элементе DOM, представляющем собой изображение. Таким образом, любое изменение его значения приведет к обновлению в соответствующем свойстве и наоборот.
- Если атрибут существует в виде встроенного свойства, но имеет булево значение, то значение не синхронизируется. Например, если `checked` возвращен в качестве атрибута, то дает начальное состояние элемента флажка (`null`, если не определен, как в нашем элементе). Если он возвращен в качестве свойства, независимо от того, был определен или нет, то мы получим булево значение (`true`, если включен, `false` — если нет) текущего состояния элемента.
- Если атрибут не существует в качестве встроенного свойства, то не будет создан и значение не синхронизируется. Например, атрибут `book` не будет по умолчанию создан в виде свойства элемента DOM.

Чтобы проверить эту идею синхронизации, рассмотрим предыдущий элемент `checkbox` и следующий оператор:

```
var checkbox = document.getElementById('book');
console.log(checkbox.getAttribute('Заголовок') === checkbox.title);

checkbox.title = 'Новый заголовок!';
console.log(checkbox.getAttribute('Заголовок') === checkbox.title);

checkbox.setAttribute('Заголовок', 'Другой заголовок!');
console.log(checkbox.getAttribute('Заголовок') === checkbox.title);

console.log(checkbox.getAttribute('checked') === checkbox.checked);
```

Вызывает весь `console.log()`, но последний выводит на консоль `true`, что и подтверждает наш тезис.

ВНИМАНИЕ

Мы еще раз хотим подчеркнуть, что метод console.log() не поддерживается старыми версиями Internet Explorer (IE 6–7). В примерах книги мы будем игнорировать эту проблему и в значительной степени будем его использовать, чтобы не применять раздражающий многих метод window.alert(). Но вы должны помнить о данном отсутствии поддержки в случае, если вам понадобится поддерживать код для этих браузеров.

Предыдущий пример можно найти в файле `chapter-4/attributes.and.properties.html`, предоставленном с книгой, а также в JS Bin (http://jsbin.com/soqexa/edit?html,js,console).

Как и свойства, значения не всегда идентичны. Например, если для атрибута `src` элемента `image` установить `image.gif`, то в результате свойству `src` будет присвоен полный абсолютный URL изображения (то есть `http://www.yourdomain.com/image.gif`).

В большинстве случаев имя свойства JavaScript совпадает с соответствующим атрибутом, но есть некоторые случаи, когда они отличаются. Так, например, атрибут класса представлен свойством `className`, а атрибут `tabindex` представляется свойством `tabIndex`.

Определение поддержки для атрибута

HTML5 представляет несколько новых атрибутов, которые вы можете добавить в элементы вашей страницы. Различие между атрибутами, определенными в разметке, но не присутствующими в элементе DOM, сгенерированном браузером, может быть очень удобным, когда вам нужно определить поддержку этих атрибутов. Возьмем в качестве примера новый атрибут HTML5 `required`. Когда пользователь добавляет его в поле формы, браузер требует ввести некие данные в этом поле перед отправлением формы. Вы можете определить, поддерживает ли браузер данный атрибут, написав:

```
if ("required" in document.createElement("input")) {
  // Атрибут поддерживается
}
```

Этот тест вернет `true`, если браузер поддерживает функцию; в противном случае — `false`.

Если вам нужно определить поддержку многих функций, то мы советуем вам использовать библиотеку Modernizr (о ней поговорим в главе 9).

jQuery предоставляет средства для простой манипуляции атрибутами элемента и обеспечивает доступ к экземпляру элемента, в связи с чем вы также можете изменить его свойства. Выбор параметра для манипуляций зависит от того, что вы хотите сделать. Начнем с получения и установки атрибутов элементов.

4.2. Работа с атрибутами

В этом разделе мы углубимся в мир атрибутов и методов, которые наша библиотека предлагает для работы с ними.

4.2.1. Извлечение значений атрибутов

В jQuery для получения и установки значения атрибута можно задействовать метод `attr()`. Это типичная особенность jQuery. Как вы можете заметить, один и тот же метод применяется либо для чтения, либо для записи значения. Дру-

гими словами, метод может работать в качестве геттера (метода чтения) или сеттера (метода записи). Действия, которые будет предпринимать jQuery, зависят от количества и типов прошлых параметров. Метод `attr()` может быть использован либо для получения значения атрибута из первого элемента согласованного набора, либо для установки значения атрибутов на всех соответствующих элементах.

Синтаксис для варианта выборки метода `attr()` (в качестве геттера) выглядит следующим образом.

Синтаксис метода: attr

`attr(name)`

Получает значение, присвоенное указанному атрибуту для первого элемента в соответствующем наборе.

Параметры

`name` (Строка) Имя атрибута, значение которого должно быть взято.

Возвращает

Значение атрибута для первого соответствующего элемента. Значение `undefined` возвращается, если соответствующий набор пуст или в первом элементе нет данного атрибута.

Даже если вы привыкли думать, что атрибуты элементов предопределены HTML, то можете использовать метод `attr()` с пользовательскими атрибутами, установленными через JavaScript, или HTML-разметку. Как мы уже видели в главе 2, для добавления пользовательского атрибута можно применить новый атрибут HTML5 `data-*`. В качестве примера рассмотрим следующий элемент `img` с пользовательским атрибутом (выделен жирным шрифтом):

```
<img id="my-image" src="image.gif" alt="An image" alt="Красивая картинка"
  data-custom="some value" />
```

Обратите внимание: мы добавили к элементу пользовательский атрибут с незамысловатым именем `data-custom`. Можно получить значение данного атрибута, как если бы это был какой-либо из стандартных атрибутов, с:

```
$('#my-image').attr('data-custom');
```

Имена атрибутов регистронезависимы в HTML. Независимо от того, как атрибут, например, `title` объявлен в разметке, можно получить (или установить, как увидите позже) атрибуты с помощью любого варианта из заданных: `Title`, `TITLE`, `TiTlE` и любых других комбинаций, они все эквивалентны. Хотя в XHTML для имен атрибутов в разметке должен использоваться нижний регистр, вы можете получить их, применяя любой вариант.

Вариант набора `attr()` содержит некоторые собственные полезные функции. Рассмотрим их подробнее.

4.2.2. Установка значений атрибутов

У jQuery есть два способа установить атрибуты на выбранные элементы. Начнем с самого простого, позволяющего установить один атрибут за раз на все извлеченные элементы. Его синтаксис выглядит следующим образом.

Синтаксис метода: attr(name, value)

`attr(name, value)`

Устанавливает указанному атрибуту переданное значение для всех элементов в объекте jQuery.

Параметры

`name`	(Строка) Имя атрибута, который должен быть установлен.
`value`	(Строка|Число|Логический тип|Функция) Указывает значение атрибута. Это может быть любое выражение JavaScript, которое следует в значении указанного типа. Любое значение, кроме функции, преобразуется в строку. Функция вызывается для каждого элемента в наборе, передает индекс элемента и текущее значение атрибута с данным именем в элементе. Возвращаемое функцией значение становится значением атрибута.

Возвращает

Коллекцию jQuery.

Этот вариант `attr()` на первый взгляд может показаться простым, но на самом деле довольно сложен в работе.

В своей самой основной форме параметр `value` может быть любым выражением JavaScript, результатом которого будет значение, впоследствии преобразованное в строку. Станет куда интереснее, когда параметр `value` будет встроенной функцией или ссылкой на функцию. В таких случаях функция вызывается для каждого извлеченного элемента с возвращаемым значением функции, используемым в качестве значения атрибута. Когда вызывается функция, ей передается два параметра. Один из них содержит отсчитываемый от нуля индекс элемента в наборе, а другой — текущее значение атрибута с именем элемента. Кроме того, элемент устанавливается в качестве контекста функции (`this`) для ее вызова. Это позволяет функции настроить свою обработку для каждого конкретного элемента — главное преимущество использования ее таким образом.

Рассмотрим следующую инструкцию:

```
$('[title]').attr('title', function(index, previousValue) {
  return previousValue + ' Я элемент ' + index +
         ' и меня зовут ' + (this.id || 'unset');
});
```

Этот метод проверит все элементы на странице, которые содержат атрибут `title`. Он изменит данный атрибут каждого элемента путем добавления к предыдущему значению строки, составленной с использованием индекса элемента в DOM и идентификатора атрибута каждого конкретного элемента, если таковой

имеется, или в противном случае — строки `'unset'` (позволяет сбросить все настройки).

Желательно применять это средство установки значения атрибута всякий раз в случае зависимости данного значения от других аспектов элемента, когда требуется *исходное* значение для вычисления нового значения или по другим причинам, вынуждающим вас установить значения по отдельности.

Второй вариант набора `attr()` позволяет удобно установить несколько атрибутов за раз.

Синтаксис метода: attr

`attr(attributes)`

Использует свойства и значения, указанные передаваемым объектом, для установки соответствующих атрибутов на всех элементах соответствующего набора.

Параметры

`name`	(Строка) Имя атрибута, который должен быть установлен.
`attributes`	(Объект) Объект, свойства которого копируются как атрибуты ко всем элементам в наборе.

Возвращает

Коллекцию jQuery.

Этот формат позволяет быстро и просто установить несколько атрибутов по всем элементам набора. Переданный параметр может быть ссылкой на объект, как правило на литерал объекта, чьи свойства определяют имена и значения атрибутов, которые будут установлены. Рассмотрим следующее:

```
$('input').attr({
  value: '',
  title: 'Пожалуйста, введите значение'
});
```

Данный оператор присваивает параметру `value` всех элементов `input` пустую строку, а `title` — строку `'Пожалуйста, введите значение'`.

Обратите внимание: если значение свойства в объекте, переданное в качестве параметра `value`, является ссылкой функции, то она работает аналогично предыдущему формату `attr()`; функция вызывается для каждого отдельного элемента в объекте jQuery.

ПРЕДУПРЕЖДЕНИЕ

Попытка изменить атрибут type на элемент input или button, созданный через document.createElement(), создаст исключение в Internet Explorer 6–8.

Теперь, когда вы знаете, как получить и установить атрибуты, рассмотрим возможности их удаления.

4.2.3. Удаление атрибутов

Для того чтобы удалить атрибуты из элементов DOM, jQuery предоставляет метод `removeAttr()`. Его синтаксис выглядит следующим образом.

Синтаксис метода: removeAttr

`removeAttr(name)`

Удаляет указанный атрибут или атрибуты из всех соответствующих элементов.

Параметры

`name`	(Строка) Имя атрибута или разделенный пробелами список имен атрибутов, которые должны быть удалены.

Возвращает

Коллекцию jQuery.

Посмотрим на пример использования этого метода. Цель состоит в том, чтобы удалить атрибуты `title` и `alt` из всех изображений на странице. Для выполнения задачи можно написать следующую инструкцию:

```
$('img').removeAttr('title alt');
```

Метод `removeAttr()` использует функцию JavaScript `removeAttribute()`. Но у него есть преимущество: он вызывается напрямую на каждом элементе в объекте jQuery и позволяет применить цепочку.

Удаление атрибута не затрагивает соответствующее свойство из DOM-элемента JavaScript, хотя это может привести к изменению его значения. Например, удаление атрибута `readonly` из элемента поменяет свойство элемента `readonly` с `true` на `false`, но само свойство останется в элементе.

Теперь посмотрим на примеры того, как можно применить эти знания к вашим страницам.

4.2.4. Игры с атрибутами

Пример 1. Форсирование открытия нового окна для ссылки

Представьте, что хотите на вашем сайте открыть в новых окнах все ссылки, которые указывают на внешние домены. Это довольно тривиальная задача, если вы полностью контролируете разметку. Можно добавить атрибут `target`, как показано здесь:

```
<a href="http://external.com" target="_blank">Какой-то внешний сайт</a>
```

Все это прекрасно, но как быть, если разметка вам неподконтрольна? Возможен запуск системы управления контентом или вики, где конечные пользователи смогут добавлять контент, но вы не можете полагаться на то, что они будут добавлять `target="_blank"` ко всем внешним ссылкам. Прежде всего необходимо определить, чего вы хотите: чтобы все ссылки, чьи атрибуты `href` начинаются с `http://`, откры-

вались в новом окне. Это можно сделать путем установки атрибута `target` в `_blank`. Вдобавок вы не принимаете во внимание ссылки, у которых уже есть установленный атрибут `target` с правильным значением. Для простоты мы в нашем примере намеренно игнорируем другие протоколы, такие как FTP, HTTPS и т. п.

Вы можете использовать способы, о которых узнали в этом разделе, чтобы сформулировать задачу лаконично:

```
$('a[href^="http://"]')
  .not('[target="_blank"]')
  .attr('target', '_blank');
```

Сначала вы выбираете все ссылки с атрибутом `href`, начинающиеся с `http://`. Протокол указывает на внешнее расположение ссылки (предполагается, что страница не использует абсолютные URL-адреса для внутренних ресурсов). Затем исключаете все ссылки, у которых есть уже атрибут `target` с установленным значением `_blank`. Наконец, устанавливаете атрибут для остальных элементов. Миссия выполнена с помощью одной строки кода jQuery! Вы можете увидеть его в действии, открыв файл `chapter-4/new.window.links.html`, предоставленный с книгой.

Пример 2. Имитация атрибута заполнителя

Еще один отличный пример использования функциональности атрибутов jQuery — имитация нового атрибута HTML5 `placeholder`. Здесь мы покажем базовую реализацию, чтобы имитировать то, что служит хорошей иллюстрацией применения метода `attr()`, но не подходит для использования в готовой версии продукта. Вы не должны использовать этот код в проектах по двум причинам. Первая — вы не будете проверять поддержку браузера, поэтому запущенный код выполнит операцию в браузерах, которые также поддерживают атрибут `placeholder`. Вторая — текст не будет скрыт после того, как поле получит фокус, так что пользователь должен удалить значение вручную, что весьма досадно.

Атрибут placeholder

Атрибут `placeholder` показывает текст, который объясняет, для чего нужно поле, или дает пример возможного значения, внутри поля. Данный текст показывается, пока поле в фокусе или пока пользователь не введет значение, в зависимости от браузера, и в этом случае текст будет спрятан. Атрибут применяется к `<input>` и `<textarea>`. Старые браузеры, которые его не поддерживают (Internet Explorer 6–9), проигнорируют этот атрибут, и ничего не произойдет, как если бы вы его не устанавливали.

В данном примере скопируйте значение атрибута `placeholder` и установите его в качестве значения атрибута `value`. Таким образом, каждый браузер будет показывать текст заполнителя, даже если `placeholder` не поддерживается. В качестве примера мы рассмотрим только вариант с элементами `<input>`, и вот почему. Чтобы поработать также и с `<textarea>`, вам понадобится метод jQuery `text()`, который мы еще не обсудили (но в ближайшее время сделаем это).

Предположим, у вас есть следующая форма:

```
<form>
  <label for="username">Имя пользователя:</label>
  <input id="username" name="username" placeholder="JohnDoe" />
  <label for="email">Электронная почта:</label>
  <input type="email" id="email" name="email" placeholder="email@fake.com" />
  <input type="submit" value="Вход" />
</form>
```

Далее можно написать этот простой код:

```
$('input').each(function(index, element) {
  var $element = $(element);
  $element.attr('value', $element.attr('placeholder'));
});
```

Выбирает все теги <input> и итерирует по ним

Копирует значение заполнителя (placeholder) как значение для атрибута

Создает новый объект jQuery, содержащий текущий элемент, и сохраняет его в переменной

Результат выполнения команд показан на рис. 4.2. Кроме того, код доступен в файле `chapter-4/placeholder.html`, предоставленном с книгой, и в JS Bin (http://jsbin.com/onuMiDU/edit?html,js,output).

Рис. 4.2. Поля формы заполнены значениями, указанными в атрибуте placeholder, с использованием метода attr()

Теперь, когда вы познакомились со всеми тонкостями работы с атрибутами, взглянем на то, как управлять свойствами элемента.

4.3. Манипулирование свойствами элементов

Чтобы получить и установить значение свойства в jQuery, можно использовать метод `attr()`. Но есть и похожий подход — `prop()`. Эти методы аналогичны по своим возможностям и параметрам, которые они принимают. Кроме того, метод `prop()` был извлечен из `attr()` как способ реорганизации последнего и чтобы позволить ему сосредоточиться исключительно на атрибутах. До версии 1.6 `attr()` работал и с атрибутами, и со свойствами. Но это было очень давно.

Синтаксис метода `prop()` для получения значения свойства выглядит следующим образом.

Синтаксис метода: prop

prop(name)

Получает значение данного свойства для первого элемента в соответствующем наборе.

Параметры

name (Строка) Имя свойства, значение которого нужно получить.

Возвращает

Значение свойства для первого соответствующего элемента. Значение undefined возвращается, если значение свойства не было задано или соответствующий набор не содержит элементов.

Рассмотрим еще раз элемент checkbox:

```
<input type="checkbox" id="legal-age" name="legal-age" title="Нажмите!" />
```

Вы можете проверить, включен ли он, используя метод prop(). В этом примере сделаем вид, что совсем забыли о фильтре :checked и методе is():

```
if ($('#legal-age').prop('checked') === false) {
  console.log('Флажок сейчас не установлен');
}
```

Метод prop() обеспечивает доступ к наиболее часто используемым свойствам, традиционно очень раздражавшим авторов страниц из-за зависимости от браузеров. Этот набор имен с нормализированным доступом, который также применяется attr(), приведен в табл. 4.1.

Таблица 4.1. Имена с нормализированным доступом в jQuery prop()

Нормализированное имя jQuery	Имя DOM
cellspacing	cellSpacing
cellpadding	cellPadding
class	className
colspan	colSpan
contenteditable	contentEditable
for	htmlFor
frameborder	frameBorder
maxlength	maxLength
readonly	readOnly
rowspan	rowSpan
tabindex	tabIndex
usemap	useMap

Хорошо знать, включен ли определенный флажок, но что делать, если вы хотите программно отметить его как включенный? Для этого вы можете использовать метод `prop()` как метод записи (сеттер). Синтаксис его первого варианта в качестве сеттера показан здесь.

Синтаксис метода: prop

`prop(name, value)`

Устанавливает названному свойству заданное значение для всех элементов в коллекции jQuery.

Параметры

`name` (Строка) Имя свойства, которое нужно записать.

`value` (Любой|Функция) Указывает значение свойства. Это может быть любое выражение JavaScript, которое записывается в значение, или функция. Она вызывается для каждого элемента, ей передается индекс элемента в коллекции jQuery и текущее значение названного атрибута для того конкретного элемента. Возвращаемое функцией значение становится значением свойства.

Возвращает

Коллекцию jQuery.

Вы можете программно отметить данный флажок как установленный следующим образом:

```
$('#legal-age').prop('checked', true);
```

Как мы уже отмечали, у методов `attr()` и `prop()` есть много общего и возможность указать несколько свойств сразу не исключение. Синтаксис этого варианта представлен здесь.

Синтаксис метода: prop

`prop(properties)`

Использует свойства и значения, указанные данным объектом, для установки соответствующих свойств во всех элементах соответствующего набора.

Параметры

`properties` (Объект) Объект, свойства которого копируются как свойства для всех элементов в наборе.

Возвращает

Коллекцию jQuery.

Этот формат позволяет быстро установить несколько свойств, избегая длинной цепочки отдельных вызовов только к `prop()`. Например, можно написать следующее:

```
$('input:checkbox').prop({
  disabled: true,
  checked: true
});
```

Последний метод, связанный с управлением свойствами, который мы обсудим, называется `removeProp()`. Он удаляет набор свойств, используя метод `prop()` для всех выбранных элементов. Его синтаксис показан ниже.

Синтаксис метода: removeProp

`removeProp(name)`

Удаляет указанное свойство из каждого элемента в коллекции jQuery.

Параметры

`name` (Строка) Имя свойства, которое нужно удалить.

Возвращает

Коллекцию jQuery.

В отличие от `removeAttr()`, этот метод не позволяет получить список разделенных пробелами имен. Его не следует использовать для удаления нативных свойств, таких как `checked` или `required`, поскольку метод может полностью удалить свойство, после чего его нельзя снова добавить к элементу. Если вы хотите изменить текущий статус одного из свойств, то установите значение `false`, прибегнув к `prop()`.

Атрибуты и свойства элемента — полезные понятия для данных, поскольку они определены в HTML и W3C, но в ходе создания страниц вам часто необходимо хранить ваши собственные данные. Посмотрим, что может сделать для вас jQuery в этом случае.

4.4. Хранение пользовательских данных в элементах

Скажем прямо: глобальные переменные — крайне неудачная идея. За исключением нечастых, поистине глобальных значений, трудно представить себе худшее место для хранения информации, которая вам потребуется при определении и реализации сложного поведения ваших страниц. Вы столкнетесь с проблемами, связанными не только с областью видимости, но и с масштабированием, когда у вас есть несколько операций, происходящих одновременно (открытие и закрытие меню, запрос на запуск Ajax, выполнение анимации и т. д.).

Функциональная природа JavaScript может помочь минимизировать трудности за счет использования замыканий (если нужно освежить знания по данной теме, загляните в приложение), но они подходят не для всех ситуаций.

Поскольку поведение ваших страниц ориентировано на элементы, имеет смысл использовать их сами по себе в качестве области хранения. Напомним, что JavaScript с его способностью динамически создавать для объектов пользовательские свойства может вам в этом помочь. Но будьте осторожны. Поскольку

элементы DOM представлены экземплярами объектов JavaScript, они, как и все другие экземпляры объектов, могут быть расширены — пользователь может добавить свойства по собственному выбору. Но не гоните лошадей!

Эти пользовательские свойства, так называемые *expandos*, не совсем безопасны. В частности, легко создать циклические ссылки, которые могут привести к серьезным утечкам памяти, например, при хранении ссылки на элемент, который вам больше не нужен. В традиционных веб-приложениях, где DOM сбрасывается по мере загрузки новых страниц, утечки памяти не будут столь серьезной проблемой. Но для авторов высокоинтерактивных веб-приложений, которые используют много сценариев, способных повлечь зависание страницы в течение достаточно долгого времени, утечки могут быть огромной проблемой.

jQuery приходит к вам на помощь, предоставляя средства для присоединения данных в отношении любого элемента DOM, который вы выбираете, с автоматическим управлением, не полагаясь на потенциально проблематичные динамические объекты. Можно разместить любое произвольное значение JavaScript, даже массивы и объекты, в элементах DOM, используя метод, разумно названный `data()`. Вот его синтаксис.

Синтаксис метода: data

`data(name, value)`

Добавляет переданное значение в управляемое jQuery хранилище данных для всех элементов в наборе.

Параметры

`name`	(Строка) Имя данных, которые нужно записать.
`value`	(Любой) Значение, которое нужно записать, кроме `undefined`.

Возвращает

Коллекцию jQuery.

Метод `data()` рассматривает имена, написанные в верблюжьем регистре, точно так же, как имена с дефисом. Инструкция:

```
$('.class').data('lastValue');
```

эквивалентна:

```
$('.class').data('last-value');
```

К тому же, в отличие от метода `attr()`, который хранит значения всегда в виде строк, `data()` может хранить тип значения. `data()` также пытается преобразовать значение атрибута в его встроенный тип при использовании в качестве геттера. Рассмотрим на примере.

Допустим, у вас есть такой код на странице:

```
$('.class').attr('last-value', 10);
console.log(typeof $('.class').attr('last-value')); ← Выводит "string"
```

Вы увидите "string" в консоли, потому что attr() преобразовал число 10 в его строковый эквивалент ("10"). С другой стороны, если вы пишете:

```
$('.class').data('last-value', 10);
console.log(typeof $('.class').data('last-value')); // Выводит "number"
```

вы увидите "number" в консоли, так как data() сохраняет тип значения в том же виде.

Теперь представьте, что у вас есть такой HTML-код:

```
<input id="name" name="name" data-mandatory="true" />
```

Вы получаете различные результаты, используя attr() и data() следующим образом:

```
console.log(typeof $('#name').attr('data-mandatory')); // Выводит "string"
console.log(typeof $('#name').data('mandatory')); // Выводит "boolean"
```

Метод attr() возвращает значение в виде строки, поэтому, введя его тип, вы получаете "string". А data() может преобразовать значение в булево значение (то же самое относится к number, null и т. д.), так что вы увидите "boolean".

Стоит отметить: undefined не признается значением, но по-прежнему возвращает объект jQuery. Таким образом, оператор:

```
$('#name').data('mandatory', undefined);
```

не изменяет значения mandatory, но возвращает вызванный jQuery объект, что позволяет построить цепочку методов.

Прекрасно, когда есть возможность добавлять новые данные к элементу, но вы до сих пор были вынуждены добавлять один элемент данных за раз, и это не очень практично. К счастью, у jQuery есть вариант data() в качестве сеттера (метода записи), который принимает объект пары ключ-значение.

Синтаксис метода: data

data(object)

Добавляет пары «ключ — значение» этого объекта в управляемое jQuery хранилище данных для всех элементов в наборе.

Параметры

object (Объект) Объект пар «ключ — значение», которые нужно записать.

Возвращает

Коллекцию jQuery.

Вскоре мы продолжим наше исследование метода data(), но сначала хотим упомянуть, что библиотека также предоставляет функцию-утилиту объекта jQuery, которая вызывается и действует таким же образом, как и вышеупомянутый метод data().

jQuery.data() (эквивалент $.data()) представляет собой метод низкого уровня, так как действует на элементе DOM вместо объекта jQuery. Принимает те же параметры, что и data(), но при этом вводит новый параметр (первый в списке), где вы можете передать элемент DOM, на котором хотите хранить данные.

Чтобы дать представление об их различиях, предположим: у вас есть элемент, содержащий book в качестве ID, и вы хотите сохранить заданное значение с помощью $.data() (метода, который не работает с объектом jQuery). Можно сделать это следующим образом:

```
$.data(document.getElementById('book'), 'price', 10);
```

Если хотите воспользоваться data() (методом, работающим с объектом jQuery), можете вызвать его так:

```
$('#book').data('price', 10);
```

Вполне ожидаемо, что этот метод может также и читать данные, а не просто записывать их. Вот синтаксис для извлечения данных с его помощью.

Синтаксис метода: data

data([name])

Извлекает сохраненные прежде данные или атрибут HTML5 data-* с указанным именем в первом элементе в наборе. Если имя не указано, то метод возвращает объект, содержащий все сохраненные прежде данные.

Параметры

name (Строка) Имя извлекаемых данных.

Возвращает

Извлеченные данные или undefined, если они не найдены.

Поведение метода data() в качестве геттера очень интересно и может немного запутать, так что заслуживает дальнейшего обсуждения.

Версия геттера у данного метода будет полезна при извлечении данных, записанных в память с помощью версии сеттера. Если прежде сохраненные данные не найдены, то метод выполняет поиск атрибута data-* HTML-элемента с тем же заданным именем. После того как data() извлекает значение из атрибута data-*, метод сохраняет данное значение в управляемом jQuery хранилище данных (рассмотрим его как внутреннюю память jQuery, используемую для отслеживания разного рода изменений). Любой следующий вызов метода больше не будет получать значение из атрибута, даже если вы изменили последний с помощью метода attr(), потому что теперь он хранится в памяти jQuery. В случае если атрибут не найден, то возвращается значение undefined. На рис. 4.3 показан описанный механизм, что поможет вам его визуализировать.

Данный процесс может быть сложным, поэтому мы рассмотрим несколько примеров. Предположим, у вас есть следующий элемент:

```
<input id="level1" type="text" value="Я текст!" data-custom="foo" />
```

Вам нужно получить значение data-custom. Это можно сделать с помощью метода либо data(), либо attr(), но с разными параметрами. Вот так:

```
console.log($('#level1').data('custom'));
console.log($('#level1').attr('data-custom'));
```

В обоих случаях выводится "foo", но методам нужны разные параметры

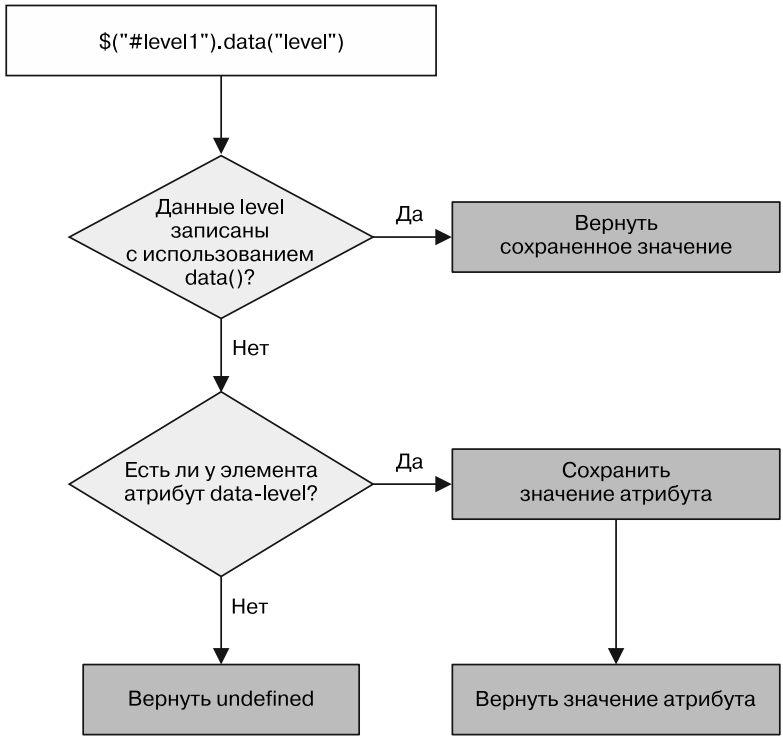


Рис. 4.3. Как data() ищет записанную информацию. Версия геттера этого метода поможет вам извлечь данные, записанные с помощью его версии сеттера. Если требуемые записанные данные не найдены, то метод ищет атрибут data-* HTML-элемента с тем же именем. Если атрибут не найден, то возвращается значение undefined

Оба предыдущих оператора выводят в консоли строку `"foo"`. Но что произойдет, если вы используете метод `data()`, как показано ниже?

```
$('#level1').data('custom', 'new value');

console.log($('#level1').data('custom'));
console.log($('#level1').attr('data-custom'));
```

Обновляет значение атрибута данных

Выводит значение, используя attr(). Выведенная строка — "foo"

Выводит значение, используя data(). Выведенная строка — "new value"

Вы изменили значение `custom`, используя метод `data()`, и попытались получить его, одновременно задействуя версии геттеров `data()` и `attr()`. На этот раз результат отличается! Звучит безумно? Мы еще не закончили.

В качестве последнего примера представьте: у вас есть предыдущий элемент без каких-либо ранее добавленных данных, так как вы только что загрузили страницу. На сей раз вы хотите доказать, что после того, как значение атрибута HTML-элемента извлекается в первый раз с использованием `data()`, значение, управляемое `data()`, совершенно не зависит от атрибута и, таким образом, от любого будущего вызова `attr()`. Чтобы увидеть это поведение в действии, вы получите значение атрибута, примененяя `data()`, затем измените атрибут с помощью `attr()` и, наконец, вызовете `attr()` и `data()`, чтобы отметить разницу. Это описание реализуется следующим кодом:

Если хотите потренироваться и лучше понять концепцию, то можете открыть файл `chapter-4/getting.and.setting.data.html`, предоставленный с книгой, или использовать созданный нами JS Bin (http://jsbin.com/uHAzIyoD/edit?html,js,console).

Метод `jQuery.data()`, описанный ранее, также может быть применен для получения сохраненных данных. Чтобы сделать это, нужно передать элемент DOM, с которым хотите работать, в первый параметр, и имя данных, которые хотите получить, — во второй. И здесь имя данных не является обязательным. Если вы его опускаете, то jQuery извлекает все данные, сохраненные в памяти для этого элемента DOM. К вспомогательной функции `jQuery.data()` можно обратиться для извлечения данных, сохраненных с использованием метода `data()`, и наоборот.

jQuery 3: исправленная ошибка

В jQuery 3 исправлена ошибка в методе `data()`, возникавшая при работе с атрибутами с цифрами в имени — например, если у вас есть следующий элемент:

```
<div id="name" data-foo-42="test"></div>
```

Если вы используете более раннюю версию jQuery, чем 3-я, и напишете:

```
console.log($('#name').data());
```

то вместо объекта, содержащего свойство `foo-42` со значением `test`, получите пустой объект.

jQuery 3: функциональные изменения

В jQuery 3 изменено поведение метода `data()`, чтобы привести его в соответствие со спецификацией Dataset API (http://www.w3.org/TR/html5/dom.html#dom-dataset). В частности, главное изменение в том, что jQuery преобразует все ключи свойств в верблюжий регистр.

Чтобы понять это изменение, рассмотрим следующий элемент:

```
<div id="name"></div>
```

Если вы используете более раннюю версию jQuery, чем 3-я, и напишете:

```
$('#name').data({'my-property': 'hello'});
console.log($('#name').data());
```

то получите такой результат в консоли:

```
{my-property: "hello"}
```

В jQuery 3 вы вместо этого получите такой результат:

```
{myProperty: "hello"}
```

Обратите внимание: в jQuery 3 имя свойства дано верблюжьим регистром, тогда как в jQuery 1.11 и 2.1 оно содержит дефис. Если хотите узнать больше об этом изменении, можете обратиться к описанию соответствующей проблемы на GitHub: https://github.com/jquery/jquery/issues/2257.

jQuery содержит методы не только для установки и получения данных. В интересах правильного управления памятью она также предоставляет метод `removeData()` как способ сбросить любые данные, которые больше не понадобятся. Метод позволяет удалить значения, ранее установленные с помощью `data()`, но не затрагивает атрибут элемента HTML5 `data-*` (для этого вы должны использовать `removeAttr()`). Синтаксис данного метода следующий.

Синтаксис метода: removeData

`removeData([name])`

Удаляет сохраненные прежде данные с указанным именем на всех элементах объекта jQuery. Параметром может также быть массив или строка с разделенными пробелами именами, которые нужно удалить. Если аргументы не указаны, то все данные удаляются.

Параметры

`name` (Строка|Массив) Строка, содержащая имя или разделенные пробелами имена данных, которые нужно удалить. Если предоставлен массив, то его элементы используются для поиска имен данных, подлежащих удалению.

Возвращает

Коллекцию jQuery.

Основываясь на свежеполученных знаниях, чтобы удалить все данные, сохраненные для конкретного элемента, можете написать следующее:

```
$('#legal-age').removeData();
```

Если вы хотите удалить данные `foo` и `bar`, то можете написать:

```
$('#legal-age').removeData(['foo', 'bar']);
```

или:

```
$('#legal-age').removeData('foo bar');
```

Обратите внимание: нет необходимости избавляться от данных вручную, когда элемент удаляется из DOM с помощью методов jQuery; библиотека сама отлично справится с задачей.

Что касается метода `data()`, jQuery предоставляет вспомогательную функцию, эквивалентную `removeData()`. Эта утилита называется `jQuery.removeData()` (или `$.removeData()`, используя сокращенную запись). Она принимает первым параметром элемент DOM, в котором вы хотите удалить данные, и дополнительно вторым параметром — имена данных, подлежащих удалению. Если вы хотите применить `$.removeData()`, чтобы удалить все данные в элементе, имеющем `legal-age` в качестве ID, должны написать:

```
$.removeData(document.getElementById('legal-age'));
```

Помимо метода `jQuery.removeData()`, у jQuery есть другая вспомогательная функция, которая имеет дело с данными в элементе. Этот метод, называемый `jQuery.hasData()`, позволяет проверить, содержит ли элемент DOM какие-либо связанные с ним данные jQuery. Его синтаксис следующий.

Синтаксис метода: jQuery.hasData

`jQuery.hasData(element)`

Определяет, есть ли у элемента какие-либо связанные данные.

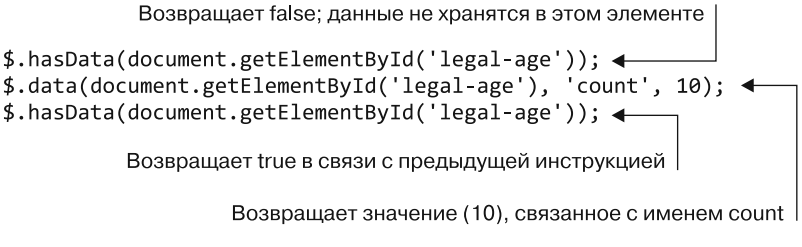
Параметры

`element` (Элемент) Элемент DOM, который нужно проверить на наличие данных.

Возвращает

`true`, если есть какие-либо связанные с элементом данные; в противном случае — `false`.

Чтобы увидеть, как это можно использовать, проверьте наличие данных в элементе, имеющем `legal-age` в качестве ID, сохраните некоторые данные, а затем снова запустите проверку. Таким образом, вы ожидаете, что первый тест вернет `false`, а второй — `true`. Посмотрим на `jQuery.hasData()` в действии:



Мы будем пользоваться возможностями добавления данных в элементы DOM, чтобы успешно решать многие примеры в последующих главах. Но если вы уже сталкивались с проблемами, которые могут возникнуть из-за работы с глобальными переменными, то заметили, как хранение данных в контексте внутри иерархии элемента открывает новые перспективы. По существу, дерево DOM превратилось в полноценную иерархию «пространства имен», которую вы можете использовать; больше не приходится ограничиваться одним глобальным пространством.

Ранее в этом разделе мы упоминали свойство `className` как пример случая, когда имена атрибутов разметки отличаются от имен свойств. По правде говоря, имена классов немного отличаются в других отношениях, а также обрабатываются как таковые библиотекой. В следующей главе описывается лучший способ работы с именами классов, нежели прямой доступ к свойству `className` или применение метода `attr()`.

4.5. Резюме

В этой главе мы вышли за пределы искусства выбора элементов и начали манипулировать их атрибутами и свойствами. Вы узнали о различиях между этими категориями, о том, как можно работать с ними, используя jQuery. Кроме того, вы узнали, как выполнять основные задачи, например форсирование открытия внешних ссылок в новых окнах, просто с помощью добавления или изменения атрибута.

Еще одной важной ролью методов, описанных в этой главе, является управление пользовательскими данными. Как вы увидели, отдельные методы демонстрируют весьма интересное поведение, которое может привести к путанице. Надеемся, мы помогли ее прояснить с помощью приведенных примеров.

Обновление или удаление свойств, данных и атрибутов может быть полезным, но необходимо продолжать изучение материалов книги, чтобы увидеть реальную эффективность jQuery. Кроме того, важно понимать такие концепции, как перемещение элементов, создание новых элементов для обертки других и обработка событий. В следующей главе мы разберем некоторые из этих тем более подробно.

5 Оживляем страницы с помощью jQuery

В этой главе:

- манипулирование именами классов элементов;
- установка содержимого элементов;
- получение и установка значений элементов форм;
- клонирование элементов DOM;
- изменение дерева DOM путем добавления, изменения или перемещения элементов.

Нынешние веб-разработчики и дизайнеры знают больше, чем те, кто работал десять лет назад (или даже они сами это же время назад), и используют эффективность разработки сценариев DOM для *улучшения* веб-опыта пользователей, а не демонстрирования назойливых фокусов вроде мерцающего текста и анимированных GIF. Что бы это ни было — постепенно появляющееся содержимое, создание элементов управления на базе основного набора, заданного HTML, или предоставление пользователю возможности настраивать страницы по собственному усмотрению, — манипулирование DOM позволило многим веб-разработчикам удивлять (а не раздражать) своих пользователей.

Почти каждый день многие из нас сталкиваются с веб-страницами, которые делают нечто, позволяющее нам сказать: «Ух ты! Я даже не знал, что так можно!» И, как всякий профессионал, коим является любой из нас (не говоря уже о том, что нам просто очень интересно узнать о таких вещах), мы немедленно смотрим исходный код, чтобы понять, как авторы страниц это сделали. Прелесть Интернета в том, что можно посмотреть на код других разработчиков в любое время, верно?

Но вместо того, чтобы писать все эти сценарии самостоятельно, мы узнаем, что jQuery предоставляет эффективный набор инструментов для манипуляции DOM, делая разработку всех типов «вау»-страниц возможной с удивительно небольшим количеством кода. В предыдущей главе вы узнали, как работать со свойствами, атрибутами и данными с jQuery, а в этой описывается, как совершать операции с элементами DOM.

5.1. Изменение стилей элемента

Повторим то, о чем упоминали в конце предыдущей главы. Свойство `className` — это пример случая, когда имена атрибутов разметки отличаются от имен свойств. Но честно говоря, имена классов немного отличаются в других отношениях, а также обрабатываются как таковые библиотекой. В данном разделе описывается лучший способ работы с именами классов, нежели прямой доступ к свойству `className` или использование метода jQuery `attr()`.

Когда вы хотите изменить стиль элемента, есть два варианта, к которым прибегают чаще других. Первый — можно добавить или удалить класс, вызывая обновление стиля для элемента на основе его нового или удаленного класса. Второй — можно работать с элементом DOM, применяя стили напрямую.

Начнем с того, что посмотрим, как с помощью jQuery можно просто изменить стили элементов через классы.

5.1.1. Добавление и удаление имен классов

Атрибут `class` элементов HTML критически важен для создания интерактивных интерфейсов. В HTML он служит для подачи этих имен в качестве строки, разделенной пробелами. Можно использовать столько пробелов, сколько хотите, но обычно применяют один. Например, у вас может быть:

```
<div class="some-class my-class    another-class"></div>
```

К сожалению, в элементах DOM имена классов в соответствующем свойстве `className` представлены не массивом, а строкой, в которой имена разделены пробелами. Как это досадно и неудобно! Получается так: всякий раз, когда вы хотите добавить или удалить имена классов из элементов, у которых уже есть эти имена, нужно анализировать строку, чтобы определить отдельные имена при чтении и обеспечивать восстановление в корректный, разделенный пробелами формат при записи.

Взяв пример с jQuery и других подобных библиотек, HTML5 представила лучший способ выполнять эту задачу через API, называемый `classList`. Он прибегает к примерно тем же методам, что и jQuery. Но, к сожалению, нативные методы могут работать только с одним элементом за раз, чем и отличаются от методов jQuery. Если вы захотите добавить класс в набор элементов, то придется итерировать по ним. Вдобавок, будучи нововведением, `classList` не поддерживается старыми браузерами, в первую очередь Internet Explorer 6–9. Чтобы лучше понять эту разницу, рассмотрим код, написанный на чистом JavaScript, который выбирает все элементы с классом `some-class` и добавляет класс `hidden`:

```
var elements = document.getElementsByClassName('some-class');
for(var i = 0; i < elements.length; i++) {
    elements[i].classList.add('hidden');
}
```

Предыдущий фрагмент совместим только с современными браузерами, включая Internet Explorer 10 или выше. Теперь сравним код с эквивалентом в jQuery:

```
$('.some-class').addClass('hidden');
```

Версия jQuery не только короче, но и совместима с Internet Explorer, начиная с 6 (конечно же, в зависимости от того, какую версию jQuery вы используете!).

ПРИМЕЧАНИЕ

Список имен классов рассматривается как неупорядоченный; таким образом, порядок следования имен в разделенном пробелами списке не имеет семантического значения.

Хотя написание кода, обрабатывающего добавление и удаление имен классов из набора элементов, не настолько монументальный труд, все же это хорошая идея абстрагировать подобные детали с помощью API, который скрывает механические детали подобных операций. К счастью, вам не нужно писать свой код, потому что jQuery уже сделала это за вас.

Добавление имен классов к элементам набора — простая операция, осуществляемая благодаря методу `addClass()`, который был использован в предыдущем фрагменте кода.

Синтаксис метода: addClass

`addClass(names)`

Добавляет указанное имя (имена) класса ко всем элементам в наборе. Если предоставлена функция, то ей передается каждый элемент набора, по одному за раз, и возвращаемое значение используется в качестве имени (имен) класса.

Параметры

`names` (Строка|Функция) Указывает имя класса или разделенную пробелами строку имен, которые нужно добавить. Если это функция — она будет вызвана для каждого элемента, с тем элементом, установленным как контекст функции (`this`). Функции передается два значения: индекс элемента и текущее значение класса элемента. Возвращаемое значение функции используется как новое имя или имена класса, которые будут добавлены к текущему значению.

Возвращает

Коллекцию jQuery.

Удалять имена класса просто с помощью метода `removeClass()`.

Чтобы увидеть, когда метод `removeClass()` может быть полезен, допустим, что у вас есть такой элемент на странице:

```
<p id="text" class="hidden">Краткое описание</p>
```

Вы можете удалить скрытый класс, используя следующую команду:

```
$('#text').removeClass('hidden');
```

Синтаксис метода: removeClass

`removeClass(names)`

Удаляет указанное имя (имена) класса из каждого элемента в коллекции jQuery. Если предоставлена функция, то ей передается каждый элемент набора, по одному за раз, и возвращаемое значение используется для удаления из имени (имен) класса.

Параметры

`names` (Строка|Функция) Указывает имя класса или разделенную пробелами строку имен, которые нужно удалить. Если это функция, она будет вызвана для каждого элемента, устанавливая тот элемент как контекст функции (`this`). Функции передается два значения: индекс элемента и значение класса перед началом удаления. Возвращаемое значение функции используется как новое имя или имена класса, которые должны быть удалены.

Возвращает

Коллекцию jQuery.

Часто может понадобиться переключить набор стилей, принадлежащий имени класса, туда и обратно, видимо, чтобы показать изменения между двумя состояниями или по какой-то другой причине, нужной вашему интерфейсу. jQuery упрощает эту работу с помощью метода `toggleClass()`.

Синтаксис метода: toggleClass

`toggleClass([names][, switch])`

Добавляет указанное имя (имена) класса элементам, не содержащим его, или удаляет имя (имена) из элементов, уже содержащий данное имя (имена) класса. Обратите внимание: каждый элемент проверяется индивидуально, так что к некоторым элементам это имя класса может быть добавлено, а в других — удалено.

Если предоставлен параметр `switch`, то имя (имена) класса всегда добавляется к элементам без них, если данный параметр имеет значение `true`, и удаляются, если `false`.

Если метод вызван без параметров, то все имена класса каждого элемента в наборе будут удалены и впоследствии восстановлены при новом вызове этого метода.

Если предоставляется только параметр `switch`, то все имена класса каждого элемента в наборе будут сохранены или удалены из этого элемента, в зависимости от значения `switch`.

Если предоставлена функция, то возвращаемое значение используется как имя или имена класса и действие выполняется в зависимости от значения `switch`.

Параметры

`names` (Строка|Функция) Указывает имя класса или разделенную пробелами строку имен, подлежащих переключению. Если это функция, она будет вызвана для каждого элемента, устанавливая тот элемент как контекст функции (`this`). Функции передается два значения: индекс элемента и значение класса того элемента. Возвращаемое значение функции используется как новое имя или имена класса, которые будут переключены.

`switch` (Логический тип) Управляющее выражение, значение которого определяет, будет ли класс только добавлен к элементам (`true`) или только удален (`false`).

Возвращает

Коллекцию jQuery.

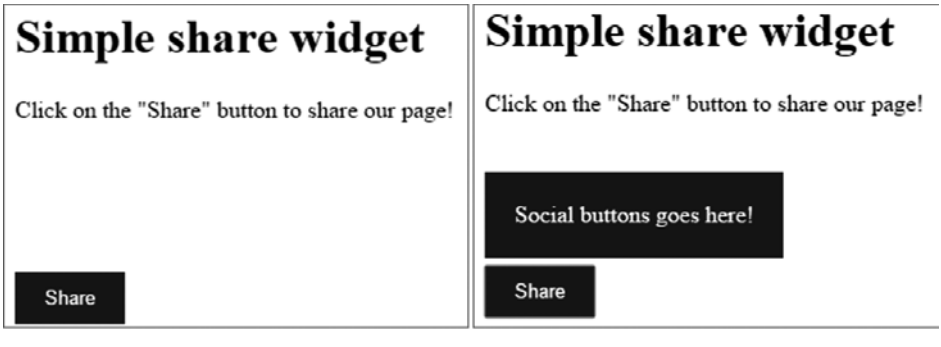
Как видите, метод `toggleClass()` дает много возможностей. Прежде чем перейти к другим методам, рассмотрим несколько примеров.

Ситуация, когда метод `toggleClass()` может понадобиться, — если вы хотите легко и быстро переключить визуальное представление элементов, обычно основываясь на некоторых других элементах. Представьте, что хотите разработать простенький виджет Share (Поделиться), при щелчке показывающий окно с кнопками социальных сетей, чтобы поделиться ссылкой на эту страницу. Если кнопку нажать повторно, то окно должно быть скрыто.

Используя jQuery и метод jQuery `click()`, вы легко сможете сделать этот виджет:

```
$('.share-widget').click(function() {
    $('.socials', this).toggleClass('hidden');
});
```

Полный код для этого демо доступен в файле `chapter-5/share.widget.html`, предоставленном с книгой. Прежде чем вы разочаруетесь, мы хотим подчеркнуть, что данное демо не предоставляет никаких фактических функций обмена, только замещающий текст. Полученная страница в двух состояниях (окно скрыто или отображается) показана на рис. 5.1, *а* и *б* соответственно.



а *б*

Рис. 5.1. Страница в двух состояниях: *а* — состояние класса hidden переключается каждый раз, когда пользователь нажимает кнопку, в результате чего окно показывается или скрывается; в исходном состоянии окно скрыто; *б* — при нажатии кнопки Share (Поделиться) переключается класс hidden; на этом рисунке показано отображающееся окно

Переключения класса на основании его наличия или отсутствия у элемента — достаточно распространенная операция, но возможны и переключения класса, основанные на других, произвольных условиях. Рассмотрим следующий код:

```
if (aValue === 10) {
    $('p').addClass('hidden');
} else {
    $('p').removeClass('hidden');
}
```

Для этой распространенной ситуации jQuery предоставляет параметр `switch`, который мы обсудили в описании метода. Можно сократить предыдущий фрагмент кода следующим образом:

```
$('p').toggleClass('hidden', aValue === 10);
```

В таком случае класс `hidden` будет добавлен ко всем абзацам, выбранным, если переменная `aValue` строго равна `10`; в противном случае он будет удален.

В качестве последнего примера представьте, что может понадобиться добавить класс для каждого элемента, основываясь на заданном условии. Вы можете захотеть добавить ко всем нечетным элементам в наборе класс, называемый `hidden`, не меняя при этом классов для четных элементов. Достигнуть цели можно, передав функцию как первый аргумент `toggleClass()`:

```
$('p').toggleClass(function(index) {
    return (index % 2 === 0) ? 'hidden' : '' ;
});
```

Иногда вам потребуется определить, есть ли у элемента конкретный класс, чтобы в соответствии с этим выполнить некие операции. С jQuery можно это сделать, используя метод `hasClass()`:

```
$('p:first').hasClass('surprise-me');
```

Метод будет возвращать значение `true`, если любой элемент в наборе имеет указанный класс; в противном случае — `false`. Синтаксис данного метода следующий.

Синтаксис метода: hasClass

`hasClass(name)`

Определяет, есть ли в наборе элемент, содержащий переданное имя класса.

Параметры

`names` (Строка) Имя класса, поиск которого будет выполняться.

Возвращает

Возвращает `true`, если какой-либо из элементов в наборе содержит переданное имя класса, в противном случае — `false`.

Вспомнив метод `is()` из главы 3, можно достичь цели так:

```
$('p:first').is('.surprise-me');
```

Но, возможно, метод `hasClass()` делает код более читаемым и внутренне является гораздо более эффективным.

Манипулирование стилистическим исполнением элементов с помощью имен классов CSS — действенный инструмент, но в некоторых случаях вы захотите сосредоточиться на мельчайших стилях как объявленных непосредственно на элементах. Посмотрим, что jQuery предложит здесь.

5.1.2. Получение и установка стилей

Изменение класса элемента позволяет вам выбрать, какой предопределенный набор заданных правил таблиц стилей должен применяться. Но иногда вы просто хотите установить значение лишь одного или нескольких свойств, неизвестных заранее; таким образом, имени класса не существует. Применение стилей напрямую к элементам (через свойство `style`, доступное для всех элементов DOM) автоматически переопределит стиль, заданный в таблице стилей (за некоторыми исключениями, например `!important`, но мы не собираемся здесь подробно обсуждать всю специфику CSS), что дает более точный контроль над отдельными элементами и их стилями.

Метод jQuery `css()` позволит вам манипулировать этими стилями, работая по принципу, аналогичному `attr()`. Можно установить индивидуальный стиль CSS, указав его имя и значение, или серию стилей, передавая их в объекте.

Синтаксис метода: css

`css(name, value)`
`css(properties)`

Устанавливает названному свойству или свойствам стиля CSS указанное значение для каждого элемента набора.

Параметры

`name` (Строка) Имя свойства CSS, которому нужно установить значение. И CSS, и DOM поддерживают форматирование свойств, название которых состоит из нескольких слов (например, `background-color` и `backgroundColor`). В большинстве случаев вы будете использовать версию первого формата.

`value` (Строка|Число|Функция) Строка, число или функция, содержащие значение свойства. Если передается число, то jQuery преобразует его в строку и добавит `"px"` в конце этой строки. Если вам нужна другая единица, то преобразуйте значение в строку и добавьте соответствующую единицу перед вызовом метода. Если в качестве параметра передается функция, она будет вызвана для каждого элемента в коллекции, устанавливая элемент в качестве контекста функции (`this`). Функции передаются два значения: индекс элемента и текущее значение. Возвращаемое значение будет новым значением для свойства CSS.

`properties` (Объект) Определяет объект, свойства которого будут скопированы как свойства CSS для всех элементов в наборе.

Возвращает

Коллекцию jQuery.

Аргумент `value` может также быть функцией аналогично методу `attr()`. Это значит, что вы можете, например, увеличить ширину всех элементов в наборе на 20 пикселов, умноженных на индекс элемента, как в следующем примере:

```
$('.expandable').css('width', function(index, currentWidth) {
  return parseInt(currentWidth, 10) + 20 * index;
});
```

В этом фрагменте нужно передать текущее значение в функцию `parseInt()`, потому что ширина элемента возвращается в пикселах как строка (например, `"50px"`). Без преобразования сумма будет совершаться как конкатенация строк наподобие `"50px20"` (если значение `index` равно `1`).

Если вы хотите увеличить ширину всех элементов на 20 пикселов, то jQuery может предложить вам хорошее сокращение. Вместо написания функции можно указать:

```
$('.expandable').css('width', '+=20');
```

И аналогичное сокращение возможно, если вы хотите отнять заданное количество пикселов:

```
$('.expandable').css('width', '-=20');
```

Приведем еще один пример того, как jQuery упрощает работу: обычно проблемное свойство `opacity` будет отлично работать со всеми браузерами (даже старыми), если передать значение между `0.0` и `1.0`; больше никакой путаницы с фильтрами alpha в старых IE!

Теперь рассмотрим пример использования второй сигнатуры метода `css()`:

```
$('p').css({
  margin: '1em',
  color: '#FFFFFF',
  opacity: 0.8
});
```

Этот код установит указанные значения ко всем элементам в наборе. Но что, если вы хотите создать эффект уменьшения прозрачности для ваших элементов?

Так же как и в сокращенной версии метода `attr()`, можно применить функции как значения к любому свойству CSS в объекте параметров свойства, и они будут вызваны для каждого элемента в наборе, чтобы определить, какое значение следует применить. Для выполнения этой задачи задействуйте функцию как значение `opacity` вместо фиксированного числа:

```
$('p').css({
  margin: '1em',
  color: '#1933FF',
  opacity: function (index, currentValue) {
    return 1 - ((index % 10) / 10);
  }
});
```

Пример страницы, использующей этот код, доступен в файле `chapter-5/descending.opacity.html`, предоставленном с книгой, и в JS Bin (http://jsbin.com/cuhexe/edit?html,js,output).

Наконец, обсудим, как можно применить `css()` с переданным ему именем или массивом имен для получения вычисленного стиля свойства или свойств, связанных с этим именем (именами) первого элемента в объекте jQuery. Когда мы

говорим «*вычисленный* стиль», имеем в виду стиль после применения всех связанных, включенных и встроенных CSS.

Синтаксис метода: css

`css(name)`

Получает вычисленное значение или значения свойства или свойств CSS, указанных по `name` для первого элемента в наборе.

Параметры

`name` (Строка|Массив) Указывает имя свойства CSS или массив свойств CSS, вычисленное значение которых будет возвращено.

Возвращает

Вычисленное значение как строку или объект пар «свойство — значение».

Этот вариант метода `css()` всегда возвращает значения как строку, так что если нужно число или какой-то другой тип, то следует проанализировать возвращаемое значение, используя, в зависимости от ситуации, функции `parseInt()` или `parseFloat()`.

Чтобы понять, как работает метод чтения `css()`, когда ему передается массив имен, рассмотрим такой пример. Необходимо вывести в консоль свойство и соответствующее ему значение элемента, имеющего `special` как его класс для следующих свойств: `font-size`, `color` и `text-decoration`. Для выполнения задачи вы должны написать:

```
var styles = $('.special').css([
   'font-size', 'color', 'text-decoration'
]);
for(var property in styles) {
   console.log(property + ': ' + styles[property]);
}
```

Получает объект с парами «свойство — значение»

Итерирует по объекту

Этот код доступен в файле `chapter-5/css.and.array.html`, предоставленном с книгой, и в JS Bin (http://jsbin.com/mimixu/edit?html,css,js,console,output). Загрузив страницу (или JS Bin) в ваш браузер, вы увидите, как выведенные значения являются результатом комбинации всех стилей, определенных на страницу. Например, значение, выведенное для `font-size`, будет не `"20px"`, а `"24px"`. Так получается потому, что значение, определенное для специального класса (`24px`), более специфично, нежели значение, определенное для элементов `div` (`20px`).

Метод `css()` — еще один пример того, как jQuery решает множество проблем с кросс-браузерной несовместимостью. Чтобы решить эту задачу с использованием только нативных методов, вам пришлось бы применить метод `getComputedStyle()` для всех версий Chrome, Firefox, Opera, Safari и Internet Explorer, начиная с версии 9, и свойства `currentStyle` и `runtimeStyle` для Internet Explorer 8 и ниже.

Прежде чем продолжить, мы хотим обратить внимание на два важных факта. Первый: разные браузеры могут возвращать разные значения для цветов в CSS, эквивалентных логически, но не текстуально. Например, если у вас есть объявления вроде `color: black;`, то некоторые браузеры могут вернуть `#000`, `#000000` или `rgb(0, 0, 0)`. Второй факт: получение сокращенных форм в свойствах CSS, таких как `margin` или `border`, библиотекой не гарантируется (хотя и работает в отдельных браузерах).

Для маленького набора общедоступных значений CSS jQuery предоставляет удобные методы, которые получают доступ к этим значениям и преобразуют их в наиболее часто используемые типы.

Получение и установка размеров

Когда дело доходит до стилей CSS, которые вы хотите получить или установить на ваших страницах, есть ли более часто используемый набор свойств, нежели ширина и высота элемента? Наверняка нет, так что библиотека упрощает работу с размерами элементов, чтобы вы смогли оперировать числовыми значениями, а не строками.

В частности, можно получить (или установить) ширину и высоту элемента как число, используя удобные методы `width()` и `height()`. Это можно сделать следующим образом.

Синтаксис метода: width и height

`width(value)`
`height(value)`

Устанавливает ширину и высоту всех элементов в соответствующем наборе.

Параметры

`value` (Число|Строка|Функция) Значение, которое будет установлено. Это может быть количество пикселов или строка, определяющая значение в единицах (таких как `px`, `em` или `%`). Если единица не указана, то по умолчанию будет использовано `px`.

Если предоставляется функция, то она вызывается для каждого элемента в наборе, передавая этот элемент как контекст функции (`this`). Функции передается два значения: индекс элемента и его текущее значение. Возвращаемое функцией значение используется как новое.

Возвращает

Коллекцию jQuery.

Учтите: это сокращения для метода `css()`, так что:

```
$('div').width(500);
```

идентично:

```
$('div').css('width', 500);
```

Вы также можете получить значение ширины или высоты следующим образом.

Синтаксис метода: width и height

`width()`
`height()`

Получает ширину и высоту первого элемента объекта jQuery.

Параметры

Отсутствуют.

Возвращает

Вычисленное значение ширины или высоты в пикселах; `null`, если объект jQuery пустой.

Версия для чтения у этих двух методов немного отличается от их аналога `css()`. Последний возвращает строку, содержащую значение и единицу измерения (например, `"40px"`), тогда как `width()` и `height()` возвращают число, являющееся значением без единицы измерения и преобразованное в тип данных `Number`. Если ваш стиль определяет ширину или высоту с помощью единиц, отличных от пикселов (`em`, `%` и т. д.), то jQuery по-прежнему вернет значение для ширины или высоты элемента в пикселах.

jQuery 3: исправленная ошибка

В jQuery 3 исправлена ошибка для методов `width()`, `height()` и всех связанных с ними. Эти методы больше не округляют значение до ближайшего пиксела, что затрудняло позиционирование элементов в некоторых случаях. Чтобы понять проблему, предположим: у вас есть три элемента шириной 33 % в пределах контейнера шириной 100 px:

```
<div class="wrapper">
    <div>Здравствуй</div>
    <div>Привет</div>
    <div>Пока</div>
</div>
```

До jQuery 3, если вы пытались получить ширину одного из трех дочерних элементов таким образом:

```
$('.wrapper div:first').width();
```

вы получали в результате значение `33`, поскольку библиотека округляла значение `33.33333`.

В jQuery 3 эта ошибка исправлена, так что теперь вы получаете более точный результат.

Тот факт, что значения ширины и высоты возвращаются из перечисленных функций как числа, — не единственное удобство, которое могут предложить эти методы. Если вы когда-либо пытались найти ширину или высоту элемента, посмотрев на их свойства `style.width` или `style.height`, то сталкивались с такой проблемой: эти свойства просто установлены соответствующим атрибутом `style` искомого

элемента. И чтобы узнать его размеры через эти свойства, нужно установить их в первую очередь. Не лучший образец полезности!

А вот методы `width()` и `height()` вычисляют и возвращают размер элемента. Знать его точные размеры на простых страницах, где бы они ни располагались, обычно не требуется. Но знать размеры на высокоинтерактивных сценарных страницах крайне важно для того, чтобы иметь возможность корректно разместить активные элементы, такие как контекстные меню, пользовательские всплывающие подсказки, расширенные средства управления и другие динамические компоненты.

Проверим в работе эти методы. На рис. 5.2 показан пример страницы, созданной с использованием двух основных элементов: первый `div` в качестве тестового заголовка, содержащего абзац текста (с границей и цветом фона для наглядности), и второй `div`, в котором должны отображаться размеры. Чтобы написать размеры во втором `div`, применим метод `html()` (его рассмотрим позже).

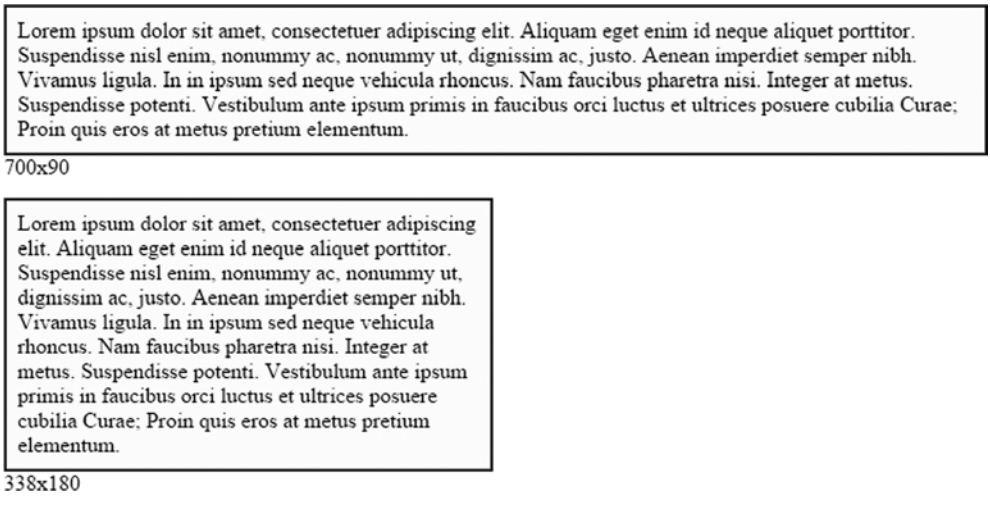


Рис. 5.2. Ширина и высота тестового элемента не фиксирована и зависит от высоты окна браузера

Размеры тестового заголовка не известны заранее, потому что к стилям не применены никакие правила, определяющие размеры. Ширина элемента зависит от ширины окна браузера, а высота — от того, как много места нужно для отображения содержимого текста. Изменение размеров окна браузера вызовет изменение обоих размеров.

На нашей странице мы определяем функцию, которая будет использовать методы `width()` и `height()` для получения размеров `div` тестового заголовка (определенного как `test-subject`) и отображает значения во второй `div` (определенном как `display`).

```
function displayDimensions() {
  $('#display').html(
    $('#test-subject').width() + 'x' + $('#test-subject').height()
  );
}
```

Мы вызываем эту функцию как последнюю инструкцию в нашем сценарии, что приводит к отображению первоначальных значений (см. рис. 5.2, верхняя часть).

Мы также добавляем вызов к той же функции в обработчике изменения размера в окне, которое обновляет экран, когда размер окна браузера изменен (вы можете узнать, как это делать, в главе 6) (см. рис. 5.2, нижняя часть). Полный код данной страницы показан в листинге 5.1, также его можно найти в файле `chapter-5/dimensions.html`, предоставленном с книгой.

Листинг 5.1. Динамическое отслеживание и отображение размеров элемента

```
<!DOCTYPE html>
<html>
   <head>
      <title>Пример динамического размера — jQuery
             в действии, 3-е издание</title>
      <link rel="stylesheet" href="../css/main.css"/>
      <style>
         #test-subject
         {
            background-color: #FFFFCC;
            border: 2px ridge maroon;
            padding: 0.5em;
         }
      </style>
   </head>
   <body>
      <div id="test-subject">
         Lorem ipsum dolor sit amet, consectetuer adipiscing elit.
         Aliquam eget enim id neque aliquet porttitor. Suspendisse
         nisl enim, nonummy ac, nonummy ut, dignissim ac, justo.
         Aenean imperdiet semper nibh. Vivamus ligula. In in ipsum
         sed neque vehicula rhoncus. Nam faucibus pharetra nisi.
         Integer at metus. Suspendisse potenti. Vestibulum ante
         ipsum primis in faucibus orci luctus et ultrices posuere
         cubilia Curae; Proin quis eros at metus pretium elementum.
      </div>
      <div id="display"></div>

      <script src="../js/jquery-1.11.3.min.js"></script>
      <script>
         function displayDimensions() {
            $('#display').html(
               $('#test-subject').width() + 'x' +
               $('#test-subject').height()
            );
         }

         $(window).resize(displayDimensions);
         displayDimensions();
      </script>
   </body>
</html>
```

Объявляет тестовый заголовок с текстом-заполнителем

Отображает размеры в этой области

Определяет функцию, которая отображает ширину и высоту тестового заголовка

Устанавливает обработчик изменения размеров, который вызывает функцию отображения

Вызывает функцию для отображения первоначальных значений

Вдобавок к удобным методам `width()` и `height()` jQuery также предоставляет аналогичные методы для получения более определенных значений измерений, как описано в табл. 5.1.

Таблица 5.1. Дополнительные методы jQuery, связанные с размерами

Метод	Описание
innerHeight()	Возвращает внутреннюю высоту первого элемента набора, что исключает border, но включает padding. Возвращается значение типа Число, а если объект jQuery не пустой — тогда возвращается null. Если возвращается числовое значение, то оно относится к значению внутренней высоты в пикселах
innerHeight(value)	Устанавливает внутреннюю высоту всех элементов набора значением, указанным в value. Типом значения может быть строка, число или функция. Единица измерения по умолчанию — px. Если предоставлена функция, то она вызывается для каждого элемента в объекте jQuery. Функции передаются два значения: индекс позиции элемента в объекте jQuery и текущее значение высоты. В этой функции this указывает на текущий элемент в объекте jQuery. Возвращаемое значение функции устанавливается как новое значение внутренней высоты текущего элемента
innerWidth()	То же, что и innerHeight(), кроме того, что возвращается значение ширины первого элемента набора, что исключает border, но включает padding
innerWidth(value)	То же, что и innerHeight(value), кроме того, что значение используется для установки внутренней ширины всех элементов набора
outerHeight ([includeMargin])	То же, что и innerHeight(), кроме того, что возвращается внешняя высота первого элемента набора, включая border и padding. С параметром includeMargin, если он true, включается и margin
outerHeight(value)	То же, что и innerHeight(value), кроме того, что значение используется для установки внешней высоты всех элементов набора
outerWidth ([includeMargin])	То же, что и innerHeight(), кроме того, что возвращается внешняя ширина первого элемента набора, что включает border и padding. С параметром includeMargin, если он true, включается и margin
outerWidth(value)	То же, что и innerHeight(value), кроме того, что значение используется для установки внешней ширины всех элементов набора

Это еще не все; библиотека также предоставляет поддержку для значений позиций и скроллинга.

Позиционирование и скроллинг

jQuery предоставляет два метода для получения позиции элемента. Оба этих метода возвращают объект JavaScript, содержащий два свойства: `top` и `left`, что показывает верхнее и левое значения позиции элемента.

Два метода используют разные источники, на основе которых они вычисляют относительные позиции. Один из этих методов, `offset()`, возвращает позицию по отношению к документу.

Синтаксис метода: offset

`offset()`

Возвращает текущие координаты (в пикселах) первого элемента в наборе по отношению к документу.

Параметры

Отсутствуют.

Возвращает

Объект со свойствами `left` и `top` как числами, описывающими позицию в пикселах по отношению к документу.

Этот метод также может использоваться для установки текущих координат одного или нескольких элементов.

Синтаксис метода: offset

`offset(coordinates)`

Устанавливает текущие координаты (в пикселах) всех элементов в наборе по отношению к документу.

Параметры

`coordinates` (Объект|Функция) Объект, содержащий свойства `top` и `left`, представляющие собой числа, которые определяют новые верхнюю и левую координаты для элемента в наборе. Если предоставляется функция, то она вызывается для каждого элемента в наборе, передавая тот элемент в виде функционального контекста (`this`) и передавая два значения: индекс элемента и объект, содержащий текущие значения `top` и `left`. Возвращаемый функцией объект используется для установки новых значений.

Возвращает

Коллекцию jQuery.

Второй метод, `position()`, возвращает значения смещения по отношению к ближайшему родителю. Для элемента это ближайший предок, у которого определено явное правило позиционирования — `relative`, `absolute` или `fixed`. Синтаксис для `position()` следующий.

Синтаксис метода: position

`position()`

Возвращает позицию (в пикселах) первого элемента в наборе по отношению к ближайшему родителю.

Параметры

Отсутствуют.

Возвращает

Объект со свойствами `left` и `top` как числами, описывающими позицию в пикселах по отношению к ближайшему родителю.

И `offset()`, и `position()` могут быть использованы только для видимых элементов.

Вдобавок к позиционированию элементов jQuery предоставляет вам возможность получить и установить позицию полосы прокрутки для элемента. В табл. 5.2 описываются эти методы, которые работают как с видимыми, так и со скрытыми элементами.

Таблица 5.2. Методы jQuery для управления элементами полосы прокрутки

Метод	Описание
scrollLeft()	Возвращает горизонтальную позицию полосы прокрутки для первого элемента набора. Возвращается значение типа Число, а если объект jQuery пустой — возвращается null. Если возвращается число, то оно обозначает значение позиции в пикселах
scrollLeft(value)	Устанавливает горизонтальную позицию полосы прокрутки для всех элементов набора в value пикселов. Этот метод возвращает набор jQuery, который был вызван
scrollTop()	То же, что и scrollLeft(), кроме того, что он возвращает вертикальную позицию полосы прокрутки для первого элемента набор
scrollTop(value)	То же, что scrollLeft(value), кроме того, что он устанавливает вертикальную позицию полосы прокрутки для всех элементов набора

Теперь, когда вы узнали, как получить и установить горизонтальную и вертикальную позицию полосы прокрутки элементов с помощью jQuery, посмотрим пример.

Представьте: есть элемент с ID `elem`, который выводится в середине вашей страницы и который вы через одну секунду хотите переместить в левый верхний угол документа. Координаты описанной точки — [0; 0], что значит следующее: для перемещения элемента нужно и `left`, и `top` установить в `0`. Для достижения этой цели нужно написать всего несколько строчек кода:

```
setTimeout(function() {
    $('#elem').offset({
      left: 0,
      top: 0
    });
}, 1000);
```

Теперь обсудим различные способы изменения контекста элемента.

5.2. Доступ к содержимому элемента

Есть множество разнообразных методов для изменения содержимого элементов в зависимости от типа текста, подлежащего вставке. Если речь идет о тексте без элементов, которые будут затем распознаны как разметка HTML, то можно использовать такие свойства, как `textContent` или `innerText`, в зависимости от браузера.

Но библиотека снова избавит вас от необходимости учитывать несовместимость браузеров, предоставив ряд методов, которыми можно пользоваться.

5.2.1. Замена HTML и текстового содержимого

Начнем с простого метода `html()`. Если применять его без параметров, то вы получите содержимое элемента в формате HTML. Если же, подобно многим другим методам jQuery, использовать его с параметром, то содержимое всех указанных в наборе элементов будет изменено.

Вот как можно получить HTML-содержимое элемента.

Синтаксис метода: html

`html()`

Получает HTML-содержимое первого элемента из набора.

Параметры

Отсутствуют.

Возвращает

HTML-содержимое первого элемента набора.

А вот как можно изменить HTML-содержимое всех элементов набора.

Синтаксис метода: html

`html(content)`

Присваивает переданный HTML-фрагмент всем элементам набора в качестве содержимого.

Параметры

`content` (Строка|Функция) HTML-фрагмент, который будет присвоен всем элементам набора в качестве содержимого. Если это функция, то она будет вызвана для каждого элемента набора, причем сам элемент будет передан в нее в виде контекста функции (`this`). Функция принимает два значения: индекс элемента и его содержимое, а возвращаемое ею значение используется как новое содержимое элемента.

Возвращает

Коллекцию jQuery.

Предположим, на вашей странице есть такой элемент:

```
<div id="message"></div>
```

Вы запустили созданную функцию, а когда она завершила работу, нужно вывести сообщение с некоей информацией для пользователя. Это можно сделать с помощью такого выражения:

```
$('#message').html('<p>Ваш баланс <b>1000$</b></p>');
```

Выполнение этого кода приведет к тому, что предыдущий элемент будет обновлен и станет выглядеть так:

```
<div id="message"><p>Ваш баланс <b>1000$</b></p></div>
```

В данном случае теги, переданные в метод, будут обработаны как разметка HTML. В частности, итоговая сумма будет выделена жирным шрифтом.

Получать и изменять можно не только HTML-код, но и обычное текстовое содержимое элементов. Метод `text()`, вызванный без параметров, возвращает строку, представляющую собой объединение всех текстовых фрагментов, содержащихся во всех элементах набора. Например, рассмотрим такой HTML-код:

```
<ul id="the-list">
  <li>Один</li><li>Два</li><li>Три</li><li>Четыре</li>
</ul>
```

Выполнение выражения:

```
var text = $('#the-list').text();
```

приведет к тому, что переменной `text` будет присвоено значение `ОдинДваТриЧетыре`. Обратите внимание: если между элементами есть пробелы или пустые строки (например, между закрывающим тегом `</li>` и следующим открывающим тегом `<li>`), то они также будут включены в получившуюся строку.

Метод `text()` имеет следующий синтаксис.

Синтаксис метода: text

`text()`

Получает и объединяет в одну строку текстовое содержимое всех элементов набора, включая их потомков.

Параметры

Отсутствуют.

Возвращает

Строку, представляющую собой объединение всего полученного текстового содержимого.

Метод `text()` также можно использовать для изменения текстового содержимого элементов из набора объекта jQuery. В этом случае синтаксис метода следующий.

Синтаксис метода: text

`text(content)`

Изменяет текстовое содержимое всех элементов набора в соответствии с переданным значением. Если переданный текст содержит угловые скобки (< и >) или амперсанд (&), то эти символы заменяются эквивалентными им сущностями HTML.

Параметры

`content` (Строка|Число|Логический тип|Функция) Текстовое содержимое, которое будет назначено элементам в наборе. Если параметр является числом или логическим значением, то он будет преобразован в строку. Любые символы квадратных скобок экранируются как HTML-сущности. Если параметр является функцией, то она вызывается для каждого элемента из набора, который передается в функцию в качестве ее контекста (`this`). Функция принимает два значения: индекс элемента и готовый текст. Возвращаемое значение становится новым содержимым элемента.

Возвращает

Коллекцию jQuery.

Изменение внутреннего текстового или HTML-содержимого элементов с помощью представленных методов приводит к тому, что предыдущее содержимое этих элементов пропадает, так что используйте методы осторожно. В jQuery есть и другие возможности, рассмотрим их.

5.2.2. Перемещение элементов

Средства манипулирования деревом DOM без обновления страницы открывают целый новый мир возможностей, позволяющих создавать динамические, интерактивные страницы. Вы уже имеете представление о том, как с помощью jQuery можно динамически создавать элементы DOM. Есть масса способов встроить эти новые элементы в дерево DOM, а также перемещать уже существующие элементы (в том числе оставляя на старом месте их копии).

Чтобы добавить содержимое в конец уже существующего элемента, используется метод `append()`.

Синтаксис метода: append

`append(content[, content, ..., content])`

Добавляет переданные аргументы (один или несколько) в конец всех элементов набора. Данный метод принимает произвольное количество аргументов, но не менее одного.

Параметры

`content` (Строка|Элемент|jQuery|Массив|Функция) Строка, элемент DOM, массив элементов DOM или объект jQuery, который будет добавлен к заданным элементам. Если параметром является функция, то она вызывается для каждого элемента набора, передающегося в функцию в качестве контекста (`this`). Функция принимает два значения: индекс элемента и его предыдущее содержимое. Возвращаемое значение функции используется как содержимое, которое будет добавлено к существующему.

Возвращает

Коллекцию jQuery.

Рассмотрим пример работы с этим методом. Предположим, имеется такой простой код:

```
$('p').append('<b>какой-то текст<b>');
```

Здесь из строки, передаваемой функции `append()`, создается HTML-фрагмент и добавляется в конец всех элементов `p` на странице.

Есть и более сложный вариант использования этого метода, когда добавляемым контентом являются уже существующие элементы DOM. Рассмотрим такой пример:

```
$('p.append-to').append($('a.append'));
```

Здесь все элементы `a` с классом `append` переносятся в конец всех элементов `p` с классом `append-to`. Если адресных элементов (элементов jQuery, для которых вызывается метод `append()`) несколько, то исходный элемент клонируется нужное количество раз и добавляется к каждому адресному элементу. В любом случае исходный элемент удаляется со старого места.

Если адресный элемент только один, то семантически данная операция является *переносом*; исходный элемент удаляется с его старого места и появляется в конце списка потомков адресного элемента.

Рассмотрим такой HTML-код:

```
<a href="http://www.manning.com" class="append">Текст</a>
<p class="append-to"></p>
```

Если выполнить предыдущее выражение, то получим следующую разметку:

```
<p class="append-to">
  <a href="http://www.manning.com" class="append">Текст</a>
</p>
```

Если же адресных элементов несколько, то эта операция представляет собой перенос с копированием, поскольку для каждого адресного элемента создается копия исходного элемента, которая затем добавляется в список его потомков.

Вместо полного набора можно применить ссылку на отдельный элемент DOM, например:

```
$('p.appendToMe').append(someElement);
```

Вот еще один пример использования данного метода:

```
$('#message').append(
    '<p>Это</p>',
    [
      '<p> - </p>',
      $('<p>').text('мой')
    ],
    $('<p>текст</p>')
);
```

Как видите, методу `append()` можно передать несколько элементов разных типов (в данном случае строку, массив и объект jQuery). Результатом выполнения кода будет элемент с ID `message` с четырьмя абзацами внутри, содержимое которых складывается в предложение «Это — мой текст».

Добавление одного или нескольких элементов в конец другого элемента — пунктов списка в конец списка, строк в таблицу, нового элемента в конец тела страницы — очень распространенная операция. Но иногда возникает необходимость вставить новый или существующий элемент не в конец адресного элемента, а в начало.

В таких случаях на помощь приходит метод `prepend()`.

Синтаксис метода: prepend

`prepend(content[, content, ..., content])`

Вставляет переданные аргументы (один или несколько) в начало всех элементов набора. Метод принимает произвольное количество аргументов, но не менее одного.

Параметры

`content` То же, что и параметр `content` метода `append()`, только эти аргументы добавляются не в конец, а в начало каждого элемента из набора.

Возвращает

Коллекцию jQuery.

Иногда возникает необходимость вставить новое содержимое не в начало или конец элемента, а где-нибудь еще. jQuery позволяет разместить новые или существующие элементы в любом месте DOM, задав тот элемент, после или перед которым нужно выполнить вставку.

Соответствующие методы называются `before()` и `after()`, что неудивительно. Их синтаксис вам покажется знакомым.

Синтаксис метода: before

`before(content[, content, ..., content])`

Вставляет переданные аргументы (один или несколько) в DOM перед элементами набора, на одном уровне с ними. Элементы набора уже должны быть частью DOM. Метод принимает произвольное количество аргументов, но не менее одного.

Параметры

`content` То же, что и параметр `content` для метода `append()`, только аргументы вставляются перед каждым элементом набора.

Возвращает

Коллекцию jQuery.

Синтаксис метода: after

`after(content[, content, ..., content])`

Вставляет переданные аргументы (один или несколько) в DOM после элементов набора, на одном уровне с ними. Элементы набора уже должны быть частью DOM. Метод принимает произвольное количество аргументов, но не менее одного.

Параметры

`content` То же, что и параметр `content` для метода `append()`, но аргументы вставляются после каждого элемента набора.

Возвращает

Коллекцию jQuery.

Указанные операции имеют важнейшее значение для эффективного манипулирования DOM на ваших страницах. Поэтому мы приводим страницу Move and Copy Lab, на которой вы сможете экспериментировать с этими операциями, пока досконально не усвоите принцип их работы. Страница доступна в файле `chapter-5/lab.move.and.copy.html`, предоставленном с книгой, а ее начальный вид показан на рис. 5.3.

На левой панели размещены три картинки, которые могут служить источником содержимого в ваших экспериментах по переносу и копированию. Чтобы выбрать одну или несколько картинок, установите соответствующие флажки.

На правой панели показаны возможные адресные элементы для операций переноса и копирования. Их тоже можно выбирать с помощью переключателей. Управляющие элементы внизу страницы позволяют применить один из четырех методов: `append()`, `prepend()`, `before()` или `after()`. (Метод `clone` пока не трогайте, мы займемся им позже.)

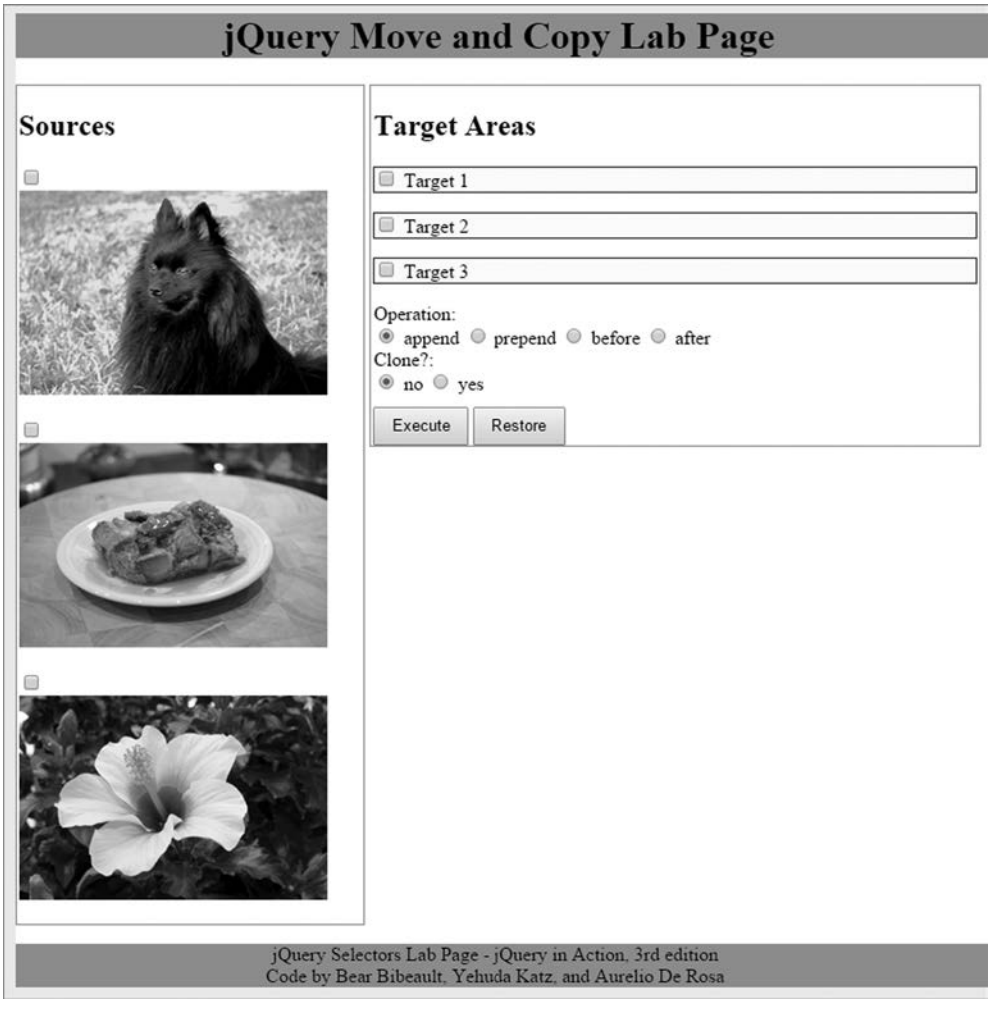


Рис. 5.3. Страница Move and Copy Lab поможет вам исследовать действие методов манипулирования DOM

Если нажать кнопку Execute (Выполнить), то все выбранные исходные изображения будут добавлены к выбранным адресным элементам с использованием указанной операции. Чтобы вернуть первоначальный вид и продолжить эксперименты, воспользуйтесь кнопкой Restore (Восстановить).

Проверим работу метода append. Выберите картинку с собакой и нажмите элемент Target 2. Оставьте выбранным метод append и нажмите Execute (Выполнить). Результат операции показан на рис. 5.4.

Используйте Move and Copy Lab, чтобы попробовать разные комбинации исходных и адресных элементов для всех четырех операций, пока не поймете, как они действуют.

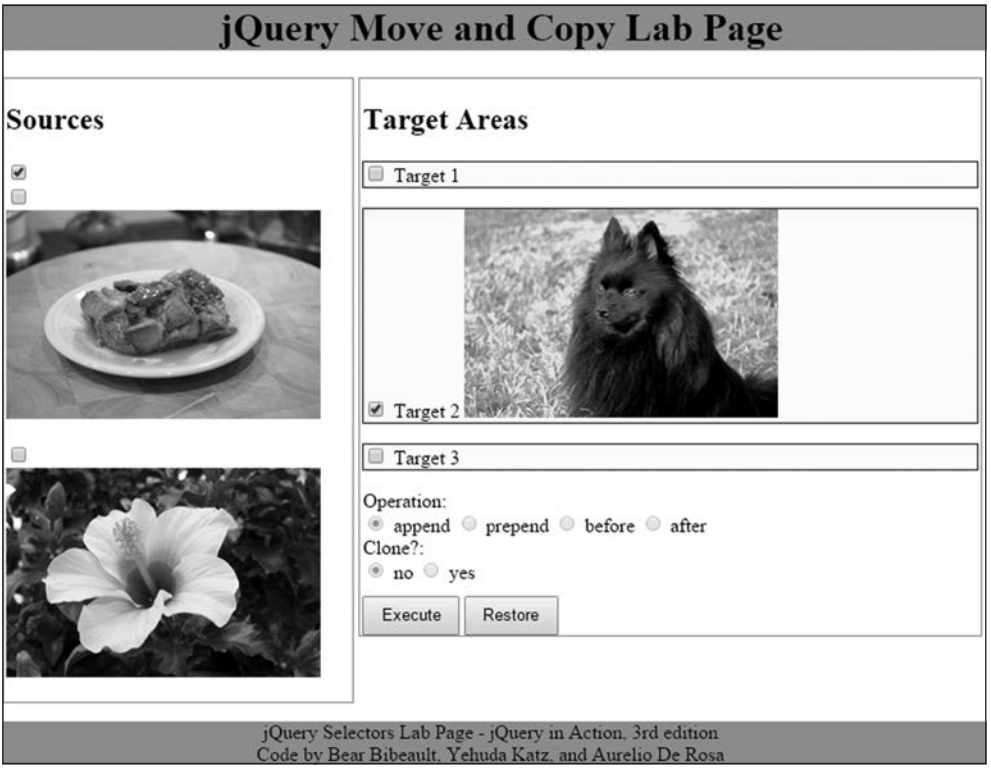


Рис. 5.4. Результат выполнения операции append: картинка с собакой добавилась в конец элемента Target 2

Иногда код проще читается, если изменить последовательность передаваемых элементов. Один из возможных подходов при переносе или копировании элемента с одного места на другое — указать в наборе исходные элементы, а в качестве параметров метода — адресные. Именно так — библиотека позволяет так сделать благодаря тому, что в ней реализованы четыре операции, аналогичные только что рассмотренным, но только в них исходные и адресные элементы поменялись местами. Это методы `appendTo()`, `prependTo()`, `insertBefore()` и `insertAfter()`, они описаны в табл. 5.3.

Таблица 5.3. Дополнительные методы переноса элементов в DOM

Метод	Описание
appendTo(target)	Вставляет каждый элемент из набора в конец указанного адресного элемента или нескольких элементов. Аргумент метода (target) может быть строкой-селектором, строкой HTML, элементом DOM, массивом элементов DOM или объектом jQuery. Метод возвращает объект jQuery, для которого он был вызван
PrependTo(target)	То же, что и appendTo(target), но элементы из набора вставляются в начало указанного адресного элемента (или нескольких элементов)

Метод	Описание
InsertBefore(target)	То же, что и appendTo(target), но элементы из набора вставляются перед адресным элементом (или элементами)
insertAfter(target)	То же, что и appendTo(target), но элементы из набора вставляются после адресного элемента (или элементов)

Вы можете подумать: «Слишком много новых методов, все сразу не выучишь!» Чтобы помочь вам разобраться, рассмотрим несколько примеров.

Пример 1. Перенос элементов

Предположим, на странице есть такой HTML-код:

```
<div id="box">
  <p id="description">jQuery — это круто!</p>
  <button id="first-btn">Я кнопка</button>
  <p id="adv">jQuery в действии рулит!</p>
  <button id="second-btn">Нажми меня</button>
</div>
```

В Chrome данный код отображается так, как показано на рис. 5.5, *а*. Нашей первой задачей будет перенести все кнопки и разместить их перед первым абзацем, с ID `description`. Можно воспользоваться методом `insertBefore()`:

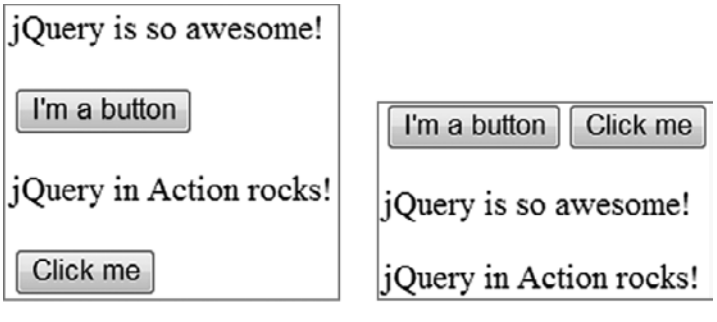
```
$('button').insertBefore('#description');
```

Внимательные читатели могут заметить: да это же метод `before()`, в котором просто поменяли местами селекторы и адресный элемент поместили внутри функции `$()`!

Поздравляем, вы угадали. Предыдущее выражение эквивалентно такому:

```
$('#description').before($('button'));
```

Какое бы из этих выражений вы ни выбрали, после его выполнения вид страницы изменится (см. рис. 5.5, *б*).



а *б*

Рис. 5.5. Вид HTML-кода в Chrome: *а* — до выполнения выражения; *б* — после выполнения выражения

Страница с рабочим кодом этого примера доступна в файле `chapter-5/moving.buttons.html`, предоставленном с книгой (после загрузки страницы кнопки

будут передвинуты), и в JS Bin (http://jsbin.com/ARedIWU/edit?html,js,output). А теперь рассмотрим более сложный пример.

Пример 2. Копирование и объединение элементов

Предположим, у нас такая же разметка, как и в предыдущем примере, но теперь мы хотим создать новый абзац и поместить его сразу после `div`. Содержимое этого нового абзаца представляет собой объединение содержимого двух абзацев внутри тега `div`. Новый абзац будет содержать такую фразу: «jQuery — это круто! "jQuery в действии" рулит!» Чтобы получить нужный результат, используем сочетание двух методов, рассмотренных в этой главе: `text()` и `after()`. Данная задача решается с помощью следующего кода:

```
var $newParagraph = $('<p></p>').text(
  $('#description').text() + ' ' + $('#adv').text()
);
$('#box').after($newParagraph);
```

Надеемся, эти два примера убедили вас, что чем больше методов вы знаете, тем больше у вас возможностей. Но мы не закончили! Есть еще один момент, который надо разъяснить, прежде чем двигаться дальше. Иногда вместо вставки одних элементов в другие нужно сделать нечто противоположное. Посмотрим, какие инструменты для этого предлагает библиотека.

5.2.3. Обертывание элементов и удаление обертки

Еще один распространенный тип манипулирования деревом DOM — помещение элементов (или набора элементов) внутрь какой-либо разметки (обертывание). Например, можно обернуть все ссылки определенного класса в элемент `<div>`. Для выполнения таких манипуляций с DOM используется метод jQuery `wrap()`.

Синтаксис метода: wrap

`wrap(wrapper)`

Обертывает элементы объекта jQuery в код, представленный в виде аргумента.

Параметры

`wrapper` (Строка|Элемент|jQuery|Функция) Строка, содержащая открывающий (и дополнительно закрывающий) тег элемента, в который будет обернут каждый элемент из набора. Аргументом также может быть элемент, объект jQuery или селектор, определяющий элементы, которые будут клонированы и использованы в качестве обертки. Если условиям селектора удовлетворяет несколько элементов или переданных объектов jQuery, то в качестве обертки будет применен только первый из них. Если аргументом является функция, то она вызывается для каждого элемента набора, а этот элемент соответственно передается ей в качестве контекста функции (`this`). Функция принимает один параметр — индекс элемента. Ее возвращаемое значение, которым может быть HTML-код или объект jQuery, используется как обертка для данного элемента набора.

Возвращает

Коллекцию jQuery.

Чтобы лучше понять действие этого метода, рассмотрим такую разметку:

```
<a class="surprise">Текст</a>
<a>Привет!</a>
<a class="surprise">Другой текст</a>
```

Обернуть каждую ссылку, имеющую класс `surprise`, в элемент `<div>` с классом `hello` можно с помощью такого кода:

```
$('a.surprise').wrap('<div class="hello"></div>');
```

Результат его выполнения будет следующим:

```
<div class="hello"><a class="surprise">Текст</a></div>
<a>Привет!</a>
<div class="hello"><a class="surprise">Другой текст</a></div>
```

Если вы захотите обернуть ссылку в клон гипотетического первого элемента `div` на странице, можете воспользоваться выражением:

```
$('a.surprise').wrap($('div:first'));
```

или:

```
$('a.surprise').wrap('div:first');
```

Помните: во втором случае содержимое `div` также будет клонировано и использовано как окружение для выбранных элементов.

Если несколько элементов объединены в объект jQuery, то метод `wrap()` применяется к каждому из них в отдельности. Если вместо этого нужно обернуть все элементы объекта jQuery как единое целое, то можно применить метод `wrapAll()`.

Синтаксис метода: wrapAll

`wrapAll(wrapper)`

Обертывает все элементы набора в заданный код как единое целое.

Параметры

`wrapper` То же, что и параметр `wrapper` метода `wrap()`.

Возвращает

Коллекцию jQuery.

jQuery 3: исправленная ошибка

В jQuery 3 исправлена ошибка метода `wrapAll()`. Она возникала при передаче методу функции в качестве аргумента. До выпуска jQuery 3 при передаче функции в `wrapAll()` каждый элемент набора оборачивался в отдельности, вместо того чтобы обернуть их все вместе. Другими словами, происходило то же самое, что при передаче функции в метод `wrap()`.

Кроме того, поскольку в jQuery 3 функция вызывается только один раз, ей не передается индекс элемента в объекте jQuery. И последнее: контекст функции (`this`) теперь указывает на первый элемент набора.

Иногда требуется обернуть не сами элементы набора, а их содержимое. В таких случаях применяется метод `wrapInner()`.

Синтаксис метода: wrapInner

`wrapInner(wrapper)`

Обертывает содержимое элементов набора, включая текстовые узлы, в заданный тег.

Параметры

`wrapper` То же, что и параметр `wrapper` метода `wrap()`.

Возвращает

Коллекцию jQuery.

Операция, противоположная `wrap()`, то есть удаление родительского тега для заданных элементов, выполняется с помощью метода `unwrap()`.

Синтаксис метода: unwrap

`unwrap()`

Удаляет родительский элемент для всех элементов набора. Дочерние элементы, а также элементы одного с ними уровня заменяют родительский элемент в дереве DOM.

Параметры

Отсутствуют.

Возвращает

Коллекцию jQuery.

jQuery 3: новый параметр

В jQuery 3 у метода `unwrap()` появился необязательный параметр `selector` — строка с селектором, определяющим родительский элемент. Если родительский элемент соответствует заданному селектору, то он будет удален. Иначе операция не будет выполнена.

Прежде чем двигаться дальше, рассмотрим пример использования этих методов.

Как создать обертку для пар label-input и label textarea внутри формы. Рассмотрим такую форму:

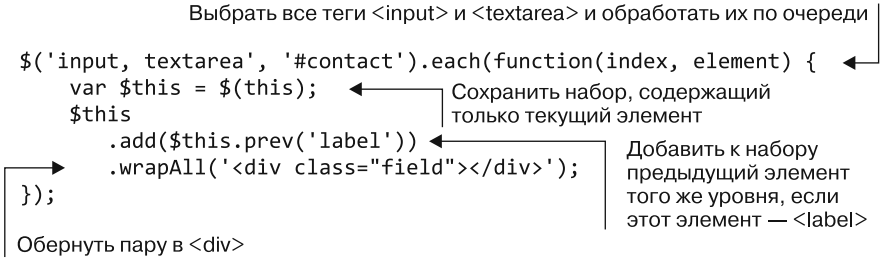
```
<form id="contact" method="post">
  <label for="name">Имя:</label>
  <input name="name" id="name" />
  <label for="email">Email:</label>
  <input name="email" id="email" />
  <label for="subject">Тема:</label>
```

```
  <input name="subject" id="subject" />
  <label for="message">Сообщение:</label>
  <textarea name="message" id="message"></textarea>
  <input type="submit" value="Отправить" />
</form>
```

Предположим, мы хотим обернуть каждую пару элементов `label-input` и `label-textarea` в тег `<div>` с классом `field`. Этот класс определен так:

```
.field
{
  border: 1px solid black;
  margin: 5px 0;
}
```

«Обернуть пары» означает, что мы не хотим создавать обертку для каждого элемента формы в отдельности. Для решения задачи нам понадобятся знания, изложенные в начальных главах книги. Но не беспокойтесь! Это будет хорошей проверкой того, насколько вы усвоили описанные ранее концепции и не требуется ли вам освежить память. Вот одно из возможных решений:



Чтобы выполнить это упражнение, нужно выбрать все теги `<input>` и `<textarea>` внутри `<form>` (для краткости мы проигнорировали другие теги, такие как `<select>`), найти способ привязать к ним соответствующие теги `<label>` и потом обработать каждую пару в отдельности. Для этого после выбора нужных элементов воспользуемся методом `each()`, описанным в главе 3. В переданной ему анонимной функции поместим текущий элемент в функцию `$()` и затем сохраним его в переменной, поскольку мы намерены использовать его дважды. Потом добавим к данному элементу предшествующий ему элемент того же уровня, если это элемент `label`. Теперь у нас есть набор из двух нужных элементов, и можно обернуть их в тег `<div>`. Ура, задача решена!

Обратите внимание на рис. 5.6. Каждая пара `label-input` и `label-textarea` заключена в черную рамку — это доказывает, что пары обернуты в тег `<div>`, как и требовалось по условиям задачи.

Если вы захотите продолжить эксперименты с этим примером, то его код доступен в файле `chapter-5/wrapping.form.elements.html`, предоставленном с книгой, и в JS Bin (http://jsbin.com/IrUhEfAg/edit?html,css,js,output).

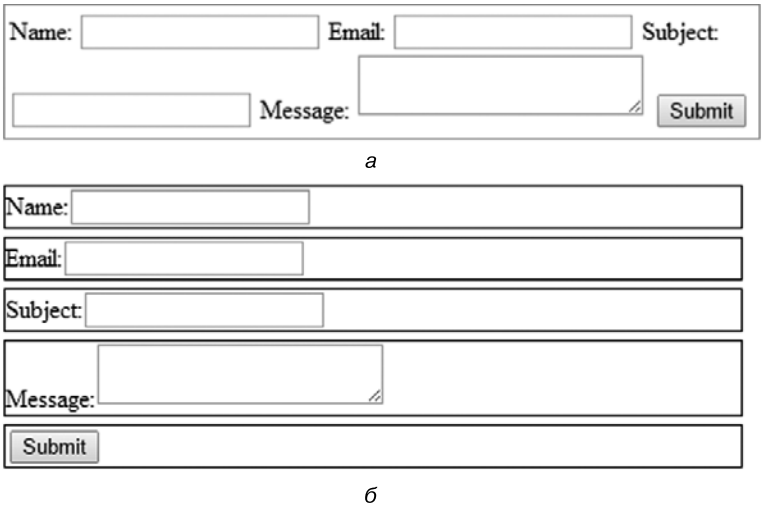


Рис. 5.6. Вид формы: *а* — перед выполнением кода; *б* — после выполнения

Теперь мы умеем выполнять много различных операций. Пора узнать, как удалить элемент из DOM.

5.2.4. Удаление элементов

Иногда возникает необходимость избавиться от ненужных элементов.

Чтобы удалить набор элементов со всем их содержимым, используется метод `remove()`, синтаксис которого следующий.

Синтаксис метода: remove

`remove([selector])`

Удаляет со страницы все элементы набора и их содержимое, включая назначенные им слушатели событий (объяснение термина см. в главе 6) и все сохраненные данные.

Параметры

`selector` (Строка) Необязательный селектор, позволяющий выбрать, какие именно элементы из набора следует удалить.

Возвращает

Коллекцию jQuery.

Следует отметить, что, как и со многими другими методами jQuery, набор возвращается в результате работы этого метода. Элементы, которые были удалены из DOM, по-прежнему ссылаются на этот набор (и, следовательно, не подпадают под сборку мусора) и в дальнейшем могут быть использованы другими методами jQuery, такими как `appendTo()`, `prependTo()`, `insertBefore()`, `insertAfter()`. Однако все сохраненные в этих элементах данные и назначенные им слушатели событий будут потеряны.

Чтобы удалить элементы из DOM, но сохранить все связанные с ними события и данные (добавленные с помощью метода `data()`), можно воспользоваться методом `detach()`.

Синтаксис метода: detach

`detach([selector])`

Удаляет все элементы набора и их содержимое из дерева DOM страницы с сохранением всех связанных событий и данных jQuery.

Параметры

`selector` (Селектор) Необязательный параметр — строка-селектор, позволяющий выбрать, какие именно элементы из набора следует исключить из DOM.

Возвращает

Коллекцию jQuery.

Метод `detach()` предпочтительно использовать для удаления тех элементов, которые вы планируете позже вернуть в DOM вместе с их событиями и данными. Типична ситуация, когда нужно извлечь элемент из DOM, внести в него некие изменения и затем снова поместить в DOM. При таком подходе повышается производительность кода, поскольку изменение элемента, исключенного из DOM, требует меньше времени, чем внесение тех же изменений в один или несколько элементов, принадлежащих DOM.

Полностью удалить содержимое элементов DOM, но сохранить сами элементы можно с помощью метода `empty()`. Он имеет следующий синтаксис.

Синтаксис метода: empty

`unwrap()`

Удаляет содержимое всех элементов DOM из набора.

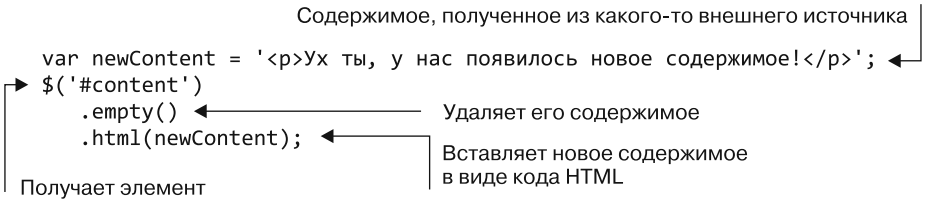
Параметры

Отсутствуют.

Возвращает

Коллекцию jQuery.

Этот метод полезен, когда вы имеете дело со вставкой содержимого извне, задействуя Ajax. Предположим, мы получили некое новое содержимое и теперь надо разместить его на странице внутри элемента `<div>` с ID `content`. Эту задачу можно решить, выполнив следующий код:



Напоминаем: будьте внимательны при размещении на странице содержимого, полученного из внешних источников, с помощью метода `html()`. Эта операция чревата атаками типа XSS (Cross-Site Scripting, межсайтовые сценарии) и CSRF (Cross-Site Request Forgery, поддельный межсайтовый запрос).

Удалять элементы, конечно, хорошо, но иногда их надо клонировать.

5.2.5. Клонирование элементов

Еще один вариант манипулирования DOM — создание копий элементов для размещения в другом месте дерева. Для этого в jQuery имеется удобный метод-обертка `clone()`.

Синтаксис метода: clone

`clone([copyHandlersAndData[, copyChildrenHandlersAndData]])`

Создает полную копию элементов из коллекции jQuery и возвращает новую коллекцию jQuery, которая содержит новую копию. Элементы копируются вместе со всеми потомками. Обработчики событий и данные также могут быть скопированы в зависимости от значения параметра `copyHandlersAndData`.

Параметры

`copyHandlersAndData`	(Логический тип) Если этот параметр равен `true`, то обработчики событий и данные также копируются. Если же равен `false` или не задан, то они не копируются.
`copyChildrenHandlersAndData`	(Логический тип) Если этот параметр равен `true`, то копируются обработчики событий и данные для всех потомков копируемых элементов. Если первый параметр задан, а этот — нет, то его значение приравнивается к значению первого. Иначе считается, что он равен `false`. В таком случае обработчики событий и данные не копируются.

Возвращает

Вновь созданную коллекцию jQuery.

Копирование существующих элементов с помощью метода `clone()` принесет мало пользы, если вы что-то делаете с копиями. Как правило, к набору, содержащему клонированные элементы, затем применяется другой метод jQuery для вставки в другом месте дерева DOM. Например, в выражении:

```
$('img').clone().appendTo('fieldset.photo');
```

создаются копии всех элементов `img`, которые затем вставляются во все элементы `fieldset` с классом `photo`.

А вот более интересный пример:

```
$('ul').clone(true).insertBefore('#here');
```

Эта цепочка методов выполняет похожую операцию, но объекты клонирования — все элементы `ul` — копируются *вместе* с их данными и обработчиками событий. Кроме того, поскольку метод `clone()` копирует также дочерние элементы, а у любого элемента `ul`, скорее всего, есть дочерние элементы `li`, можно быть уве-

ренными, что при клонировании никакая информация не потеряется. Поскольку мы пропустили второй элемент, но задали первый, данные и обработчики событий всех дочерних элементов также будут скопированы.

Прежде чем перейти к следующей теме, рассмотрим еще один, последний, пример. Предположим, у нас есть набор ссылок с картинками внутри. И у ссылок, и у картинок есть свои данные и назначенные им обработчики событий. Мы хотим скопировать все эти элементы и разместить их копии после всех элементов страницы, в конце первого тега `div`. Кроме того, мы хотим сохранить данные и обработчики событий только для ссылок, но не картинок. Данная задача решается с помощью следующего выражения:

```
$('a').clone(true, false).appendTo('div:first');
```

Здесь продемонстрировано использование необязательных параметров, обсуждавшихся в описании метода.

Чтобы увидеть результат применения операции клонирования, вернемся к странице Move and Copy Lab. Как раз над кнопкой Execute (Выполнить) находится пара переключателей, позволяющих задать операцию клонирования как часть основной операции по манипулированию DOM. Когда переключатель Clone? (Клонировать?) установлен в положение yes (да), сначала создаются клоны исходных элементов, а затем к ним применяется метод `append()`, `prepend()`, `before()` или `after()`.

Проделайте снова часть экспериментов, которые вы проводили раньше, добавив клонирование. Обратите внимание: при этом после выполнения операций исходные элементы остаются неизменными.

Теперь вы умеете вставлять, удалять и копировать элементы. Комбинируя эти операции, можно совершать такое сложное действие, как *замена*. Но знаете что? Вам не придется это делать!

5.2.6. Замена элементов

Для тех случаев, когда надо заменить существующие элементы новыми или переместить один элемент на место другого, в jQuery есть метод `replaceWith()`.

Синтаксис метода: replaceWith

`replaceWith(content)`

Заменяет каждый элемент набора заданным контентом.

Параметры

`content` (Строка|Элемент|Массив|jQuery|Функция) Строка, содержащая HTML-фрагмент для замены существующего контента, или элемент DOM, массив элементов DOM, либо объект jQuery, содержащий элементы, которые будут перемещены на место элементов набора. Если это функция, то она вызывается для каждого элемента набора, принимает данный элемент в качестве контекста (`this`) и не имеет параметров. Значение, возвращаемое функцией, используется в качестве нового содержимого элемента.

Возвращает

Коллекцию jQuery, содержащую замененные элементы.

Чтобы понять, как работает этот метод, рассмотрим пример. Предположим, имеется код:

```
<img src="image1.jpg" alt="Мяч" />
<img src="image2.jpg" alt="Синяя птица" />
<img src="image3.jpg" />
```

Мы хотим заменить каждую картинку с атрибутом `alt` на элемент `span`, внутри которого разместится текст — значение атрибута `alt` замененной картинки. Используя методы `each()` и `replaceWith()`, получим такой код:

```
$('img[alt]').each(function(){
  $(this).replaceWith('<span>' + $(this).attr('alt') + '</span>');
});
```

Метод `each()` позволяет перебрать все элементы набора и применить к каждому из них метод `replaceWith()`, чтобы заменить картинки сгенерированными элементами `span`. Получится следующая разметка:

```
<span>Мяч</span>
<span>Синяя птица</span>
<img src="image3.jpg" />
```

На этом примере мы снова убеждаемся, как легко манипулировать DOM с помощью библиотеки.

Метод `replaceWith()` возвращает набор jQuery, содержащий удаленные из DOM элементы, на тот случай, если вы захотите сделать с ними что-то еще, а не просто сбросить их. В порядке упражнения подумайте, как бы вы дополнили код примера, чтобы вставить удаленные элементы в другом месте дерева DOM.

Если аргументом метода `replaceWith()` является уже существующий элемент, то он снимается со своего первоначального места в DOM и прикрепляется на месте удаляемых элементов. Если в наборе указано несколько элементов, то исходный элемент клонируется нужное количество раз.

Иногда бывает удобно изменить порядок элементов, указанных в `replaceWith()`, чтобы в селекторе стоял замещающий элемент. Нам уже встречались взаимно дополняющие методы, такие как `append()` и `appendTo()`, позволяющие задавать элементы в той последовательности, при которой получится более логичный код.

Аналогичным образом у метода `replaceAll()` есть его зеркальная копия `replaceWith()`, позволяющая выполнить такую же операцию. Но только в этом случае заменяемые элементы не являются набором, для которого вызывается метод, а передаются методу в виде аргумента-селектора.

Подобно методу `replaceWith()`, `replaceAll()` возвращает коллекцию jQuery. Но эта коллекция содержит *замещающие* элементы, а не замененные. Последние теряются, и дальнейшие операции с ними невозможны. Помните об этом, когда будете решать, какой метод замены использовать.

Синтаксис метода: replaceAll

`replaceAll(target)`

Заменяет каждый элемент, соответствующий переданному в метод селектору, на набор элементов, для которых вызван метод.

Параметры

`target` (Строка|Элемент|Массив|jQuery) Строка-селектор, элемент DOM, массив элементов DOM или коллекция jQuery, определяющая заменяемые элементы.

Возвращает

Коллекцию jQuery, содержащую вставленные элементы.

Судя по описанию метода `replaceAll()`, можно получить тот же результат, что и в предыдущем примере, написав такой код:

```
$('img[alt]').each(function(){
  $('<span>' + $(this).attr('alt') + '</span>').replaceAll(this);
});
```

Обратите внимание на то, как мы поменяли местами аргументы, переданные в `$()` и `replaceAll()`.

Теперь, когда мы обсудили поддержку основных элементов DOM, рассмотрим в общих чертах обработку особого типа элементов — элементов форм.

5.3. Обработка значений элементов форм

У элементов форм есть специальные свойства, поэтому в состав ядра jQuery вошло несколько удобных функций для таких действий:

- получение и присваивание значений элементам формы;
- их сериализация;
- выбор элементов формы по их специфическим свойствам.

Что такое элемент формы?

Говоря об *элементах формы*, мы имеем в виду элементы, которые появляются внутри форм, имеют атрибуты `name` и `value` и чьи значения после отправки формы передаются на сервер в виде параметров запроса HTTP.

Рассмотрим одну из самых распространенных операций, обычно выполняемых с элементом формы, — получение ее значения. В самых распространенных случаях эту задачу решает метод `val()`. Он возвращает значение атрибута `value` для первого элемента объекта jQuery. Этот метод имеет следующий синтаксис.

Синтаксис метода: val

`val()`

Возвращает текущее значение первого элемента из коллекции jQuery. Если первым элементом является `<select>`, в котором не выбран ни один вариант, то метод возвращает `null`. Если же это селектор с множественным выбором (`<select>` с атрибутом `multiple`) и выбран хотя бы один из вариантов, то возвращается массив из всех выбранных значений.

Параметры

Отсутствуют.

Возвращает

Полученное значение или несколько значений.

Метод `val()` очень полезен, но у него есть ряд ограничений, о которых следует помнить. Если первый элемент набора не является элементом формы, то метод возвращает пустую строку. Некоторых это вводит в заблуждение. Данный метод не делает различий между состояниями флажков и переключателей. Он возвращает значения атрибута `value` этих элементов, независимо от того, включены они или выключены.

Когда речь заходит о переключателях, возможности селекторов jQuery в сочетании с методом `val()` способны сэкономить дни работы. Представим форму с группой переключателей (набор переключателей с общим именем) `radio-group`, для которой выполняется такое выражение:

```
$('input[type="radio"][name="radio-group"]:checked').val();
```

Это выражение возвращает значение единственного — выбранного — положения переключателя или же `undefined`, если ни одно положение переключателя еще не выбрано. Так гораздо удобнее, чем перебирать в цикле все кнопки в поисках включенного элемента, не правда ли?

Поскольку метод `val()` срабатывает только для первого элемента из набора, то в случае группы флагов, где может быть включено сразу несколько элементов, он менее полезен. Но jQuery не бросает разработчиков на произвол судьбы. Посмотрите на этот код:

```
var checkboxValues =
  $('input[type="checkbox"][name="checkboxgroup"]:checked').map(function() {
    return $(this).val();
  })
.toArray();
```

Метод `val()` прекрасно подходит для получения значения отдельных элементов формы, но если нужно получить весь набор значений, который будет передан на сервер при отправке формы, гораздо удобнее воспользоваться методом `serialize()` или `serializeArray()` (с ними вы познакомитесь в главе 10).

Еще одна распространенная операция — присвоение значения элементу формы. Метод `val()` можно использовать и здесь, передав ему значение. В этом случае он имеет следующий синтаксис.

Синтаксис метода: val

`val(value)`

Присваивает переданное значение всем элементам набора. Если передан массив значений, то все указанные в наборе флажки, переключатели и варианты элементов `select` становятся выбранными, если их значения соответствуют одному из элементов массива.

Параметры

`value` (Строка|Число|Массив|Функция) Значение, которое будет присвоено атрибуту `value` для каждого элемента из набора. Массив значений используется для того, чтобы определить, какие переключатели, флажки или варианты списка будут выбраны. Если значением является функция, то она вызывается для каждого элемента набора и этот элемент передается ей в качестве контекста (`this`). Функция также принимает два значения: индекс и текущее значение элемента. Значение, возвращаемое функцией, присваивается параметру `value` элемента.

Возвращает

Коллекцию jQuery.

Как следует из описания, метод `val()` можно использовать для установки флажков и положений переключателей, а также для выбора вариантов в элементах `<select>`. Рассмотрим такой код:

```
$('input[type="checkbox"], select').val(['один', 'два', 'три']);
```

Здесь будут найдены все элементы типа `checkbox` и все элементы `select`, значения которых совпадают с одним из вариантов: `"один"`, `"два"` или `"три"`. Все элементы, удовлетворяющие этим условиям, будут выбраны или активизированы. Если для элемента `select` не задан атрибут `multiple`, то будет выбран только один вариант, с первым подходящим значением. Так, в предыдущем коде будет выбран только вариант со значением `один`, поскольку в массиве, переданном методу `val()`, строка `"one"` идет раньше, чем `"два"` и `"три"`. Таким образом, метод `val()` полезен не только для текстовых элементов форм.

5.4. Резюме

Благодаря методам, описанным в этой главе, вы можете копировать, перемещать, заменять и даже удалять элементы. Вы также можете вставлять новое содержимое в начало и конец элементов и создавать обертки для любого элемента или группы элементов. Кроме того, мы обсудили способы работы со значениями элементов форм, позволяющими писать эффективный, но лаконичный код.

Обладая этими знаниями, вы готовы к изучению более сложных понятий, начиная с часто грязной работы по обработке событий на ваших страницах.

6 События и где они происходят

В этой главе:

- реализация модели событий в браузерах;
- модель событий jQuery;
- подключение обработчиков событий к элементам DOM;
- делегирование событий;
- события и пространства имен;
- экземпляр объекта `Event`;
- вызов обработчиков событий из сценариев;
- регистрирование проактивных обработчиков событий.

Подобно многим другим системам управления GUI, интерфейсы, создаваемые HTML-страницами, являются *асинхронными*. Несмотря на то что протокол HTTP, благодаря которому браузер получает эти страницы, по своей природе синхронный, упомянутыми интерфейсами управляют *события*. Независимо от реализации GUI — в настольной программе, используя Java Swing, X11 или .NET Framework, либо же на веб-странице, применяя HTML и JavaScript, — программа выполняет одни и те же основные действия.

1. Запускает пользовательский интерфейс.
2. Ждет, пока произойдет нечто заслуживающее внимания.
3. Реагирует соответственно.
4. Возвращается к пункту 2.

На первом шаге формируется *вид* пользовательского интерфейса, остальные определяют *поведение* программы. В случае веб-страниц браузер формирует их вид в зависимости от переданных ему разметки (HTML) и стилей (CSS). Включенные в состав страницы сценарии не только определяют поведение пользовательского интерфейса (UI), но также могут менять его вид.

Эти сценарии реализуются в виде *слушателей событий*, также называемых *обработчиками событий* (впрочем, между данными понятиями есть некоторое техническое различие). Обработчики реагируют на различные события, происходящие во время отображения страницы. События могут создаваться системой (такие как таймеры или выполнение асинхронных запросов), но чаще являются следствием каких-то действий пользователя (движений мыши, щелчков кнопкой мыши, ввода текста с клавиатуры и даже сенсорных жестов). Если бы не возможность реагировать на события, то вся мощь Всемирной паутины свелась бы к показу картинок.

В HTML очень мало встроенных семантических действий, не требующих от разработчика написания сценариев (таких как обновление страницы после перехода по ссылке или отправка формы по нажатию кнопки отправки). Любое другое поведение страницы потребует от вас обработки различных событий, происходящих во время взаимодействия пользователя с ней.

В данной главе мы изучим различные способы, с помощью которых браузеры распознают события и позволяют вам устанавливать обработчики, определяющие, что произойдет в результате этих событий. Мы также узнаем, какие сложности возникают из-за разницы в обработке событий каждым из браузеров. Затем мы увидим, как библиотека решает эти проблемы и освобождает вас от них.

Итак, начнем с того, как события обрабатываются в браузерах.

JavaScript надо знать

Одним из преимуществ jQuery в отношении веб-страниц является способность реализовать большое количество разрешенных сценарием вариантов поведения, не вынуждая пользователя писать большой сценарий самостоятельно.

Библиотека позволяет не углубляться в детали реализации конкретных действий и сконцентрироваться на логике работы приложения.

До настоящего момента все шло сравнительно гладко. Чтобы понимать примеры jQuery, представленные в предыдущих главах, вам требовались только самые базовые знания JavaScript. В этой и последующих главах для эффективного использования jQuery вам потребуется понимание некоторых фундаментальных понятий JavaScript.

Возможно, ваша подготовка такова, что эти понятия уже знакомы. Но некоторым авторам страниц удается делать довольно много, не углубляясь в теорию, — такие ситуации становятся возможными благодаря гибкости JavaScript. Прежде чем двигаться дальше, мы хотим убедиться, что вы хорошо усвоили эти базовые понятия.

Если вы свободно работаете с объектами и функциями JavaScript, хорошо понимаете, что такое контексты и замыкания, то можете продолжать чтение настоящей и следующих глав. Если же перечисленные понятия вам незнакомы или вы имеете о них лишь смутное представление, то мы настоятельно рекомендуем сначала обратиться к приложению, чтобы изучить эти необходимые понятия, а затем вернуться к данной главе.

6.1. Модели обработки событий в браузерах

Задолго до того, как кто-либо задумался о стандартизации обработки событий браузерами, компания Netscape Communications Corporation представила свою модель обработки в браузере Netscape Navigator. Сегодня эта модель известна под несколькими названиями. Вы могли слышать о ней как о модели событий Netscape, базовой модели событий и даже под весьма размытым названием модели событий браузеров, но большинство людей привыкло называть ее *моделью событий DOM уровня 0*.

ПРИМЕЧАНИЕ

Уровень DOM соответствует уровню реализации требований спецификации W3C DOM. Нулевого уровня (DOM уровня 0) не существует, этот термин используется для неформального описания того, что было сделано до DOM уровня 1.

Концерн W3C не создал стандартизированную модель обработки событий ниже DOM уровня 2, представленного в ноябре 2000 года. Эта модель поддерживается всеми современными браузерами, такими как Internet Explorer версии 9 и выше, Firefox, Chrome, Safari и Opera. Internet Explorer версии 8 и ниже имеет собственную реализацию. Он поддерживает функционал модели событий DOM уровня 2 лишь частично и при этом использует свой интерфейс. Ранее мы уже показали, что для jQuery это досадное обстоятельство не проблема, но вам все же нужно потратить некоторое время, чтобы понять, как действуют эти различные модели.

6.1.1. Модель событий DOM уровня 0

Данную модель событий используют большинство любителей и начинающих разработчиков при создании веб-страниц, поскольку она не зависит от браузера и сравнительно проста.

В этой модели, для того чтобы объявить обработчик событий, нужно назначить свойству DOM-элемента ссылку на экземпляр функции. Эти свойства настраиваются на обработку определенных типов событий: например, чтобы обрабатывать событие `click`, нужно назначить функцию свойству элемента `onclick`, а для обработки события `mouseover` надо назначить функцию свойству `onmouseover` — если, конечно, элемент поддерживает такой тип событий.

Браузеры также позволяют сделать тело функции-обработчика события значением атрибута и встроить его прямо в HTML-разметку DOM-элементов, создав таким образом простейший обработчик событий. В листинге 6.1 показаны примеры с обоими вариантами обработчиков. Его код доступен в файле `chapter-6/dom.0.events.html`, предоставленном с книгой.

Листинг 6.1. Объявление обработчиков событий в модели DOM уровня 0

```
<!DOCTYPE html>
<html>
  <head>
    <title>Модель событий DOM уровня 0 - jQuery
           в действии, 3-е издание</title>
    <link rel="stylesheet" href="../css/main.css"/>
    <style>
      img
      {
        display: block;
        margin: auto;
      }
    </style>
  </head>
  <body>
    <img id="example" src="images/example.jpg" alt="Вспышка молнии"
      onclick="console.log('В ' + formatDate(new Date()) +
                ' БАБАХ!');" />
    <script>
      function formatDate(date) {
        return (date.getHours() < 10 ? '0' : '') +
           date.getHours() +
           ':' + (date.getMinutes() < 10 ? '0' : '') +
           date.getMinutes() +
           ':' + (date.getSeconds() < 10 ? '0' : '')
           + date.getSeconds() +
           '.' + (date.getMilliseconds() < 10 ?
                 '00' : (date.getMilliseconds() < 100 ? '0' : '')) +
           date.getMilliseconds();
      }
      document.getElementById('example').onmouseover =
      function(event) {
      console.log('В ' + formatDate(new Date()) + ' Тарарах!');
      };
      </script>
  </body>
</html>
```

Приводит элемент img ❶

❷ Определен обработчик события mouseover

Выводит текст в консоль ❸

В этом примере показаны оба стиля объявления обработчиков событий: в атрибуте разметки ❶ и в разделе сценариев ❷. В теле страницы определен элемент `img` с ID `example`, для которого определен обработчик события `click` с использованием атрибута `onclick` ❶.

Внутри тега `<script>` определена функция `formatDate()`, форматирующая и возвращающая строку с датой для переданного ей объекта `Date`. Затем с помощью метода JavaScript `getElementById()` мы получаем ссылку на изображение и назначаем его свойству `onmouseover` встроенную функцию ❷. Она становится обработчиком события `mouseover` и выполняется, когда это событие происходит. Обратите внимание: данная функция ожидает один параметр, чтобы возвратить его, о чем мы поговорим немного позже. Функция выводит в консоль дату, отформатированную с помощью определенной ранее функции `formatDate()` ❸.

ПРИМЕЧАНИЕ

Здесь мы опускаем концепцию ненавязчивого JavaScript. Задолго до того, как вы достигнете конца этой главы, вы поймете, почему никогда не следует встраивать обработку событий в разметку DOM!

Если вы загрузите эту страницу (доступна в файле `chapter-6/dom.0.events.html`, предоставленном с книгой) в Chrome и несколько раз щелкнете на изображении, то получите картину, показанную на рис. 6.1.

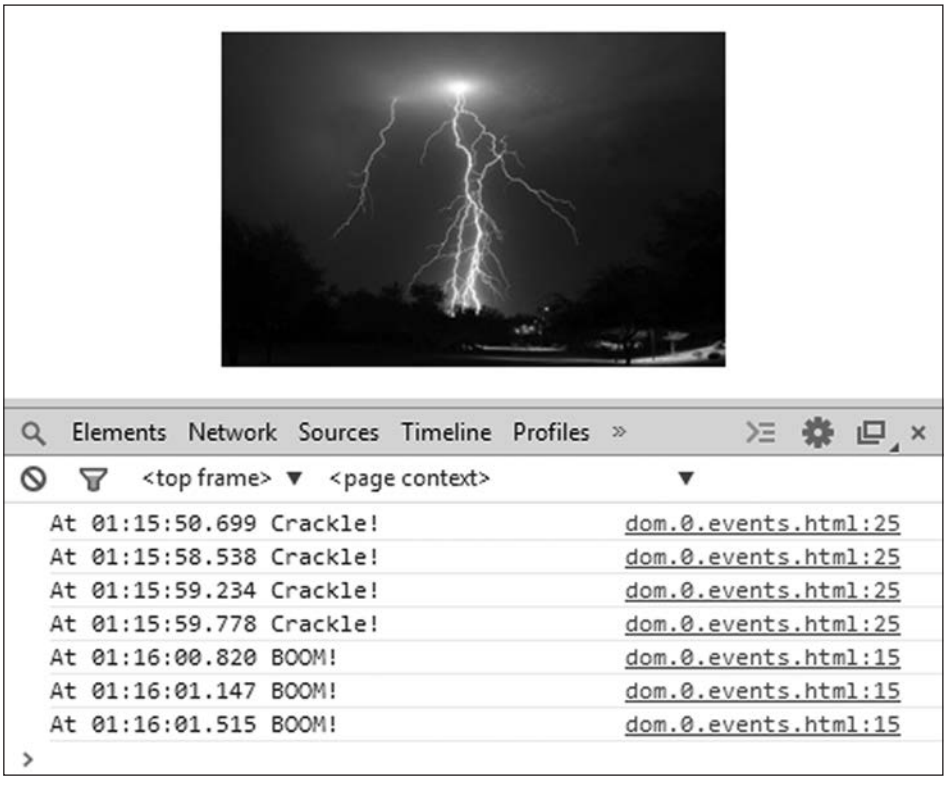


Рис. 6.1. Если навести указатель мыши на картинку и щелкнуть кнопкой, то обработчик события создаст и выведет в консоль эти сообщения

Мы объявили обработчик события `click` в элементе разметки `img`, использовав следующий атрибут:

```
onclick="console.log('В ' + formatDate(new Date()) + ' БАБАХ!');"
```

Может показаться, что именно этот оператор является обработчиком события `click`, однако все иначе. Когда обработчик объявлен через атрибут разметки HTML, для него автоматически создается анонимная функция, *телом* которой является значение атрибута. Если предположить, что `imageElement` — ссылка на

элемент изображения, то объявление обработчика события в атрибуте эквивалентно созданию такой конструкции:

```
imageElement.onclick = function(event) {
  console.log('В ' + formatDate(new Date()) + ' БАБАХ!');
};
```

Заметьте, как используется значение атрибута в теле сгенерированной функции. Эта функция создана так, что параметр `event` доступен внутри нее. Кроме того, в функции можно обратиться и к самому элементу с помощью ключевого слова `this`. Мы не применили его в данном простом примере, но обращаем ваше внимание на эту особенность.

Напомним снова: использование механизма атрибутов при объявлении обработчиков событий в модели DOM уровня 0 является нарушением принципа ненавязчивости JavaScript. Вскоре вы поймете, что этот язык и библиотека jQuery, которая также компенсирует несовместимости браузеров, предоставляют гораздо лучший способ назначения обработчиков событий, чем описанные выше. Но сначала мы изучим основные параметры событий.

Экземпляр Event

Во всех браузерах, поддерживающих стандарты W3C, обработчику события при его вызове передается экземпляр объекта `Event`. Напомним, такими браузерами являются Internet Explorer версии 9 и выше, Firefox, Chrome, Safari и Opera. В Internet Explorer 8 и более ранних версиях все сделано по-своему, там экземпляр `Event` передается глобальному свойству (другими словами, свойству окна `window`) с именем `event`. Следует отметить, что для сохранения обратной совместимости с более старыми сценариями новые версии Internet Explorer по-прежнему обращаются к событиям в объекте `window`. Более того, Microsoft продолжает использовать, кроме стандартных, собственные свойства объектов. Примечательно, что браузер Chrome, хоть и был выпущен намного позже публикации DOM уровня 2, также задействует ссылки на события в объекте `window` и поддерживает свойства Internet Explorer.

Чтобы справиться с этим несоответствием, вы часто будете встречать в обработчиках событий, не использующих jQuery, такую конструкцию:

```
if (!event) {
    event = window.event;
}
```

Читатели, хорошо знакомые с JavaScript, знают, что этот код можно сократить до одной строки:

```
event = event || window.event;
```

На данном уровне с помощью технологии *обнаружения свойств* (feature detection — понятие, которое мы рассмотрим в главе 9) проверяется, передан ли функции параметр `event`. Если нет, то ему присваивается значение свойства `event`

экземпляра `window`. После этого можно обращаться к параметру `event` независимо от того, как он стал доступен обработчику события.

В свойствах экземпляра `Event` хранится самая разная информация о событии, которое произошло и сейчас обрабатывается, — в частности, какой элемент вызвал событие, координаты указателя мыши, а для событий клавиатуры — какая клавиша была нажата.

Но не будем спешить. В старых версиях Internet Explorer используются не только собственные средства для передачи экземпляра `Event` обработчику, но и собственное определение самого объекта `Event` вместо описываемого стандартом W3C, так что вам предстоит распознавать объект. Теперь, зная обо всех этих различиях, вы поймете радость разработчиков, когда новые версии Internet Explorer наконец-то стали соответствовать стандартам W3C. Microsoft не такое уж зло, как многим кажется.

Рассмотрим несовместимости разных браузеров на таком примере. Чтобы получить ссылку на исходный элемент (элемент, вызвавший событие), в поддерживающих стандарт браузерах нужно добавить свойство `target`, но в старых версиях Internet Explorer это свойство называется `srcElement`. Чтобы учесть данное несоответствие, используют распознавание браузера с помощью такого кода:

```
var target = event.target || event.srcElement;
```

Этот оператор проверяет, определено ли свойство `event.target`, и если да, то присваивает его значение локальной переменной `target`. Если же нет, то данной переменной присваивается значение `event.srcElement`. Вам придется выполнять аналогичные действия и для других интересующих вас свойств объекта `Event`.

До сих пор мы исходили из предположения, что обработчики событий всегда связаны с элементом, запускающим данное событие, таким как элемент `img` в листинге 6.1 (см. выше), однако события перемещаются по дереву DOM. Посмотрим, как это происходит.

Всплытие событий

При возникновении события в элементе дерева DOM механизм обработки события браузера проверяет, существует ли обработчик данного события для этого элемента, и если да, то вызывает его. Но это вовсе не конец истории.

После того как исходный элемент использовал (или упустил) шанс обработать событие, объектная модель проверяет родителя этого элемента. Если у него есть собственный обработчик событий данного типа, то данный обработчик также вызывается. Затем проверяется его родитель и родитель родителя и т. д. до самой верхушки дерева DOM. Поскольку такая обработка событий напоминает всплывающие цепочки пузырьков в бокале шампанского (если представить себе дерево DOM с корнем наверху), то весь процесс называется *всплытием событий* (иногда *цепочкой событий*) (event bubbling).

Преобразуем пример, приведенный в листинге 6.1, так, чтобы проследить этот процесс. Изменения будут отражены в листинге 6.2. Его код доступен в файле `chapter-6/dom.0.propagation.html`, предоставленном с книгой.

Листинг 6.2. Продвижение события из точки возникновения до вершины дерева DOM

```
<!DOCTYPE html>
<html>
  <head>
    <title>Цепочка событий в DOM уровня 0 – jQuery
           в действии, 3-е издание</title>
    <link rel="stylesheet" href="../css/main.css"/>
    <style>
      img
      {
        display: block;
        margin: auto;
      }
    </style>
  </head>
<body>
  <div id="greatgrandpa">
    <div id="grandpa">
      <div id="pops">
        <img id="example" src="images/example.jpg"
             alt="Вспышка молнии" />
      </div>
    </div>
  </div>
  <script>
    function formatDate(date) {
      return (date.getHours() < 10 ? '0' : '') + date.getHours() +
         ':' + (date.getMinutes() < 10 ? '0' : '') +
         date.getMinutes() +
         ':' + (date.getSeconds() < 10 ? '0' : '')
         + date.getSeconds() +
         '.' + (date.getMilliseconds() < 10 ?
                '00' : (date.getMilliseconds() < 100 ? '0' : '')) +
                date.getMilliseconds();
    }

    var elements = document.getElementsByTagName('*');
    for(var i = 0; i < elements.length; i++) {
      (function(current) {
        current.onclick = function(event) {
          event = event || window.event;
          var target = event.target || event.srcElement;
          console.log(
               'B ' + formatDate(new Date()) +
               ' для ' + current.tagName + '#'+ current.id +
               ' исходный элемент ' + target.tagName +
               '#' + target.id
          );
        };
```

Выбирает все элементы на странице ❶

Перебирает в цикле все выбранные элементы ❷

Назначает каждому элементу обработчик onclick ❸

```
        })(elements[i]);
      }
    </script>
  </body>
</html>
```

В этом примере много интересных изменений. Прежде всего, исчез обработчик события `mouseover` и все внимание сконцентрировано на событии `click`. Затем, у элемента `img` появились три вложенных родительских элемента `div` — главным образом, чтобы искусственно поместить элемент `img` на более низкое место в иерархии DOM. И почти у каждого элемента на странице появился ID.

Внутри тега `<script>` для выбора всех элементов страницы использован метод JavaScript `getElementsByTagName()` и универсальный селектор ❶. Затем они перебираются с помощью цикла `for` ❷, где каждому из них назначается обработчик события `click` ❸. Для каждого элемента создается *замыкание* (если данное слово вызывает у вас недоумение, пожалуйста, прочитайте в приложении раздел, посвященный этой теме), записывающее его экземпляр в локальную переменную `current`. Внутри обработчика вы не можете обратиться к `elements[i]`, поскольку в момент выполнения обработчика значение индекса `i` находится за пределами этого массивоподобного объекта.

ПРИМЕЧАНИЕ

> Метод getElementsByTagName() возвращает не массив, а объект HTMLCollection. Такие объекты называют массивоподобными, поскольку они позволяют обращаться к их элементам по индексу, а также имеют свойство length. Однако в них не реализованы методы массивов, такие как push() и join().

В обработчике учитываются особенности разных браузеров, рассмотренные в предыдущем разделе, чтобы определить экземпляр `Event` и идентифицировать исходный объект. Затем в консоль выводится сообщение. Это еще одна интересная часть примера. Благодаря замыканиям в сообщении есть имя тега и идентификатор текущего элемента (если он есть), а также ID исходного элемента. Таким образом, каждое сообщение в консоли содержит информацию о текущем элементе в цепочке, а также об исходном элементе, с которого она началась.

Если вы загрузите страницу (доступна в файле `chapter-6/dom.0.propagation.html`, предоставленном с книгой), наведете указатель мыши на картинку и щелкнете кнопкой, то получите результат, показанный на рис. 6.2.

Здесь ясно видно, что после возникновения события оно вначале передается исходному элементу, а затем — всем его предкам по цепочке, до корневого элемента `html`.

Это эффективная возможность: она позволяет назначать обработчики элементам любого уровня для обработки событий, происходящих с их потомками. Представьте, к примеру, обработчик элемента `form`, реагирующий на любое изменение его дочерних элементов и динамически изменяющий вид формы в зависимости от их новых значений.

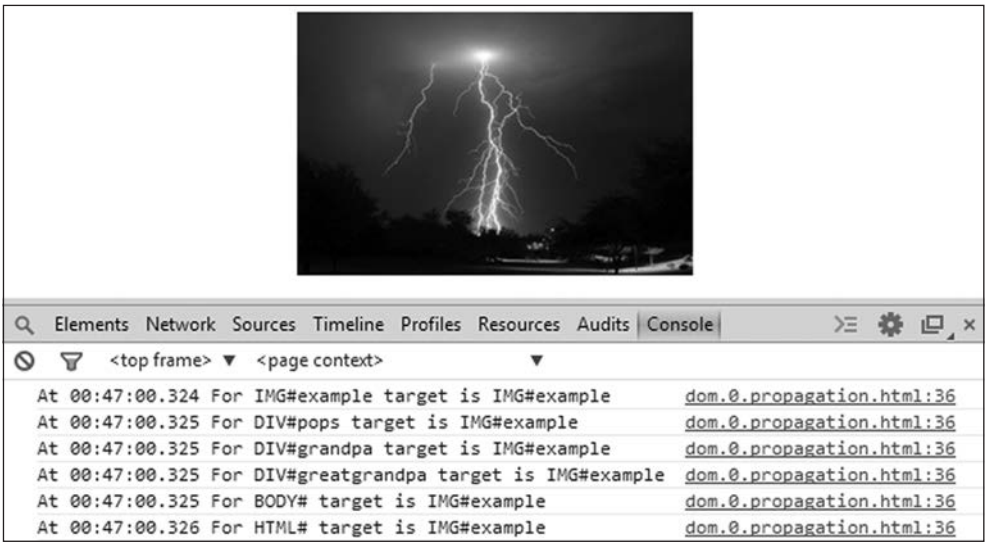


Рис. 6.2. По сообщениям в консоли наглядно прослеживается продвижение события по цепочке вверх по дереву DOM — от исходного элемента до самого корня

Но что, если вы не хотите перемещать события? Можно ли это остановить?

Воздействие на продвижение событий: семантические действия

Может возникнуть ситуация, когда вы захотите воспрепятствовать тому, чтобы событие всплывало по дереву DOM. Это может быть потому, что вы скрупулезны и знаете, что все необходимые действия по обработке события уже выполнены, или же хотите предотвратить нежелательную обработку этого события выше по цепочке.

Независимо от причины можно препятствовать всплытию события с помощью механизмов, заданных экземпляром `Event`. В современных браузерах для прерывания продвижения события по иерархии элементов вызывается метод `stopPropagation()` экземпляра `Event`. В старых версиях Internet Explorer необходимо присвоить свойству `cancelBubble` экземпляра `Event` значение `true`. Любопытно, что многие современные браузеры с поддержкой стандарта W3C также поддерживают и механизм `cancelBubble`, хотя он и не входит ни в один в стандарт W3C.

С некоторыми элементами связана семантика умолчания. Например, в результате события `click` на элементе `a` браузер перейдет по ссылке, указанной в атрибуте `href` данного элемента, а событие `submit` для элемента `form` вызовет обработку формы. Если вы захотите отменить эти семантические действия — обычно их называют *действиями по умолчанию* — для какого-то элемента, то в современных браузерах можно вызвать метод `preventDefault()` экземпляра `Event`. В старых версиях Internet Explorer этого метода нет, поэтому нужно присвоить значение `false` свойству `returnValue`. Или же можно сделать так, чтобы ваш обработчик события возвращал `false` — результат будет тот же. Иногда возникает необходимость остановить продвижение события и отменить действия по умолчанию.

Такие операции часто используются при валидации форм. В обработчике события `submit` можно реализовать валидационные проверки различных элементов формы и в случае любых проблем с данными возвращать `false`.

Возможно, вам также встречались такие элементы `form`:

```
<form name="myForm" onsubmit="return false;" ...
```

Этот код надежно блокирует обработку формы чем угодно, кроме управляющего сценария (методом `form.submit()`, который не генерирует события `submit`).

В модели событий DOM уровня 0 практически любой шаг обработчика события требует распознавания браузера, чтобы затем выполнить соответствующее действие. Какой ужас! Но запаситесь терпением и мужеством — нам предстоит рассмотреть модель следующего уровня.

6.1.2. Модель событий DOM уровня 2

Один из немногих недостатков модели событий DOM уровня 0 — у одного элемента может быть только один обработчик события для каждого события. Причина вот в чем: для хранения ссылки на функцию, обрабатывающую событие, используется свойство. Если вы хотите, чтобы по щелчку на элементе выполнялись два действия, то такой код не поможет:

```
someElement.onclick = function doFirstThing() {};
someElement.onclick = function doSecondThing() {};
```

Поскольку второй оператор присваивания заменяет предыдущее значение свойства `onclick`, при возникновении события будет вызвана только функция `doSecondThing()`. Конечно, можно вставить обе функции в третью, которая будет вызывать обе, но по мере усложнения страниц становится все труднее отслеживать подобные вещи.

Более того, установленные на странице компоненты и библиотеки многоразового использования «не знают» об обработке событий для других компонентов. Код, написанный посторонними людьми, может также пытаться назначить элементу какое-нибудь свойство `someElement.onclick`. Один из этих обработчиков точно будет переопределен, а какой именно — чужой или ваш — неизвестно. Можно обратиться к другим решениям, но все они усложняют и без того достаточно непростой код страниц.

Новый *стандарт* модели событий, DOM уровня 2, был создан специально для решения таких проблем. Рассмотрим, как обработчики событий, в том числе и несколько сразу, назначаются элементам DOM в этой усовершенствованной модели.

Назначение обработчиков событий

Вместо того чтобы присваивать свойству элемента ссылку на функцию, обработчики событий в DOM уровня 2 назначаются *методом* элемента. Для каждого элемента DOM определен метод `addEventListener()`, который используется для назначения слушателей событий этому элементу. Формат данного метода таков:

```
addEventListener(eventType, listener, useCapture)
```

Параметр `eventType` — строка, определяющая тип обрабатываемого события. Эти строковые значения, в сущности, те же имена событий, которые использовались в модели DOM уровня 0, но без префикса: `click`, `mouseover`, `keydown` и т. д.

Параметр `listener` — это ссылка на функцию (часто встроенную, обычно анонимную), которая будет обработчиком данного типа событий для этого элемента. Как и в базовой модели событий, экземпляр `Event` передается данной функции в качестве первого параметра.

Последний параметр, `useCapture`, — логическое значение, его использование мы обсудим немного позже, когда поговорим о продвижении событий в модели уровня 2. Пока что оставим его равным `false`.

Теперь, когда мы изучили параметры метода `addEventListener()`, угадайте, о чем пойдет речь дальше? Правильно, о старых версиях Internet Explorer и их особом способе назначения обработчиков! Мы займемся этим вплотную в подразделе 6.1.3.

Чтобы увидеть, как работает этот метод, снова внесем изменения в пример из листинга 6.1 (см. выше). Теперь мы используем усовершенствованную модель событий. Мы займемся только событиями типа `click`. На этот раз создадим три обработчика события `click` для элемента `img`. Код нового примера, показанного в листинге 6.3, доступен в файле `chapter-6/dom.2.events.html`, предоставленном с книгой.

Листинг 6.3. Назначение обработчиков событий в модели событий DOM уровня 2

```
<!DOCTYPE html>
<html>
  <head>
    <title>События в DOM уровня 2 – jQuery в действии,
          3-е издание</title>
    <link rel="stylesheet" href="../css/main.css"/>
    <style>
      img
      {
        display: block;
        margin: auto;
      }
    </style>
  </head>
  <body>
    <img id="example" src="images/example.jpg"
         alt="Вспышка молнии" />
    <script>
      function formatDate(date) {
        return (date.getHours() < 10 ? '0' : '') + date.getHours() +
          ':' + (date.getMinutes() < 10 ? '0' : '')
          + date.getMinutes() +
          ':' + (date.getSeconds() < 10 ? '0' : '')
          + date.getSeconds() +
          '.' + (date.getMilliseconds() < 10 ?
                '00' : (date.getMilliseconds() < 100 ? '0' : '')) +
          date.getMilliseconds();
      }

      var element = document.getElementById('example');
```

```
      element.addEventListener('click', function(event) {
        console.log('В ' + formatDate(new Date()) + '
                    БАБАХ один раз!');
      }, false);
      element.addEventListener('click', function(event) {
        console.log('В' + formatDate(new Date()) +
                    ' БАБАХ два раза!');
      }, false);
      element.addEventListener('click', function(event) {
        console.log('В ' + formatDate(new Date()) +
                    ' БАБАХ три раза!');
      }, false);
    </script>
  </body>
</html>
```

Создает три обработчика событий для элемента img

Несмотря на свою простоту, данный код наглядно демонстрирует, как создать несколько обработчиков одного и того же типа событий для одного и того же элемента — в базовой модели событий сделать это было совсем не так легко. Внутри тега `<script>` мы берем ссылку на элемент `img` и создаем *три* обработчика для события `click` (1).

Загрузив эту страницу в любой браузер, поддерживающий стандарты W3C (но не в Internet Explorer версии 8 или ниже), и щелкнув на картинке, вы получите результат, показанный на рис. 6.3.

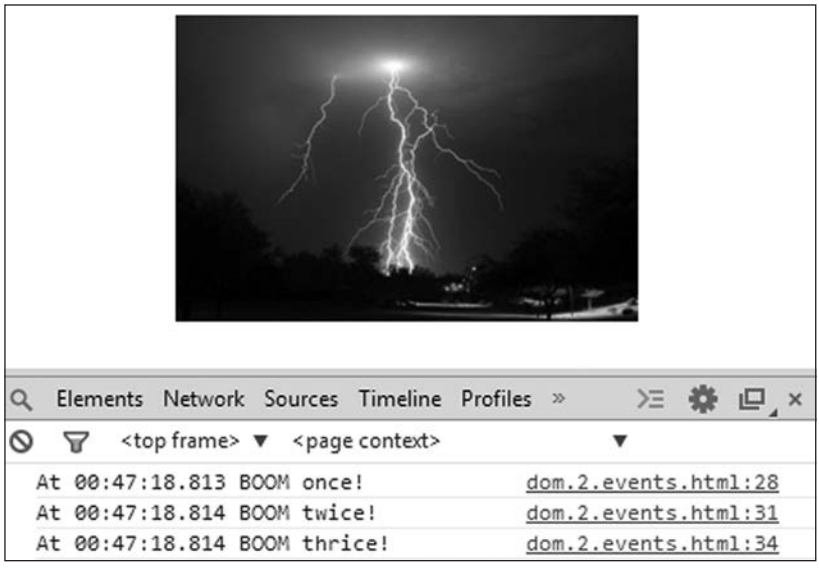


Рис. 6.3. Если щелкнуть на картинке один раз, сработают все три созданных обработчика событий

Теперь, когда мы продемонстрировали это очень удобное свойство, рассмотрим параметр `useCapture`.

Продвижение событий

Ранее мы показали, что в модели событий DOM уровня 0 после возникновения события в элементе оно передается от исходного элемента вверх по дереву DOM всем его предкам. В усовершенствованной модели событий уровня 2 эта цепочка сохраняется, но к ней добавляется этап перехвата.

В модели событий DOM уровня 2 событие после возникновения сначала передается от корня дерева DOM вниз до исходного элемента, а затем обратно. Начальный этап (от корня до исходного элемента) называют *стадией перехвата (capture phase)*, а последующий (от исходного элемента до корня) — *стадией всплытия (bubble phase)*.

Когда функция назначается обработчиком события, она может быть помечена как обработчик захвата и в этом случае вызывается на стадии захвата. Если же она помечена как обработчик всплытия, то будет вызываться на стадии всплытия. Нетрудно догадаться, что параметр `useCapture` функции `addEventListener()` как раз и определяет, какой тип обработчика будет назначен: значение `false` соответствует обработчику всплытия, а `true` — обработчику захвата. В новых версиях всех основных браузеров (например, в Firefox, начиная с версии 6) этот параметр сделан обязательным и по умолчанию равен `false`.

Вернемся ненадолго к примеру в листинге 6.2, где мы рассматривали продвижение событий по иерархической структуре DOM в базовой модели DOM. В нем мы поместили элемент `img` внутрь трех вложенных один в другой элементов `div`. В такой иерархии при использовании модели DOM уровня 2 событие `click`, вызванное исходным элементом `img`, будет продвигаться по дереву DOM, как показано на рис. 6.4.

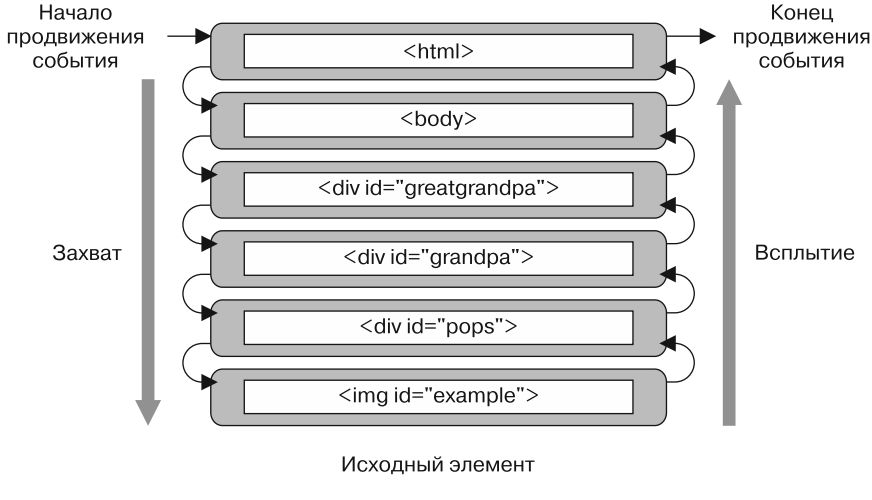


Рис. 6.4. В модели DOM уровня 2 событие продвигается по иерархическому дереву DOM дважды: от вершины до исходного элемента на стадии захвата и от исходного элемента до вершины на стадии всплытия

Не пора ли проверить, как это работает? В листинге 6.4 (доступен в файле `chapter-6/dom.2.propagation.html`, предоставленном с книгой) показан код страницы с иерархией элементов, аналогичной изображенной на рис. 6.4).

Листинг 6.4. Отслеживание продвижения события на стадиях всплытия и захвата

```
<!DOCTYPE html>
  <html>
    <head>
      <title>Продвижение событий в DOM уровня 2 – jQuery
             в действии, 3-е издание</title>
      <link rel="stylesheet" href="../css/main.css"/>
      <style>
        img
        {
          display: block;
          margin: auto;
        }
      </style>
    </head>
    <body>
      <div id="greatgrandpa">
        <div id="grandpa">
          <div id="pops">
            <img id="example" src="images/example.jpg"
                 alt="Вспышка молнии" />
          </div>
        </div>
      </div>
      <script>
        function formatDate(date) {
          return (date.getHours() < 10 ? '0' : '') + date.getHours() +
             ':' + (date.getMinutes() < 10 ? '0' : '') +
             date.getMinutes() +
             ':' + (date.getSeconds() < 10 ? '0' : '') +
             date.getSeconds() +
             '.' + (date.getMilliseconds() < 10 ?
                   '00' : (date.getMilliseconds() < 100 ? '0' : '')) +
             date.getMilliseconds();
          }
          var elements = document.getElementsByTagName('*');
          for(var i = 0; i < elements.length; i++) {
            (function(current) {
              current.addEventListener('click', function(event) {
              console.log(
                    'При ' + formatDate(new Date()) +
                    ' захвате для ' + current.tagName + '#'+
                    current.id +' исходный элемент ' +
                    event.target.tagName + '#' +
                    event.target.id
              );
          }, true);
```

Создается слушатель событий для текущего ❶ элемента на стадии захвата

```
            current.addEventListener('click', function(event) {
                console.log(
                      'При ' + formatDate(new Date()) +
                      current.id +' всплытии для ' +
                      current.tagName + '#'+ ' исходный элемент ' +
                      event.target.tagName + '#' +
                      event.target.id
                );
            }, false);
          })(elements[i]);
        }
      </script>
    </body>
  </html>
```

Создается слушатель событий для ❷ текущего элемента на стадии всплытия

В этом коде по сравнению с примером из листинга 6.2 для создания обработчиков событий используется модель событий DOM уровня 2. В теге `<script>` с помощью метода `getElementsByTagName()` и универсального селектора выбираются все элементы страницы. Для каждого из них создается два обработчика: на стадии захвата ❶ и на стадии всплытия ❷. Каждый обработчик выводит в консоль сообщение, в котором указаны тип обработчика, текущий элемент и ID исходного элемента.

Если вы загрузите страницу в Chrome и щелкнете на картинке, то получите сообщения, показанные на рис. 6.5, в которых отражается последовательность обработки события на разных стадиях прохождения по дереву DOM. Обратите внимание: поскольку у исходного элемента тоже есть обработчики для обеих стадий, то для исходного элемента оба они будут выполнены так же, как и для его родительских узлов.

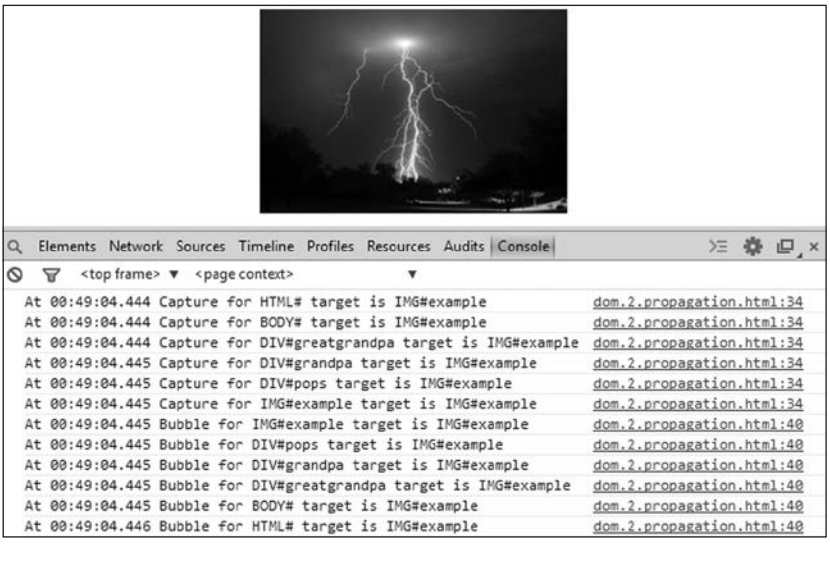


Рис. 6.5. Если щелкнуть на картинке, то каждый обработчик выведет в консоль сообщение, в котором указан путь прохождения данного события на стадиях захвата и всплытия

Теперь, когда мы справились со всеми сложностями и вы понимаете различия между этими двумя типами обработчиков, настало время сообщить вам, что такие обработчики используются на веб-страницах крайне редко. Одна из исторических причин тому — старые версии Internet Explorer не поддерживают данный тип прохождения событий.

Прежде чем мы покажем, как jQuery помогает решить эту проблему, рассмотрим вкратце модель обработки событий в Internet Explorer.

6.1.3. Модель событий в Internet Explorer

До версии 9 Internet Explorer не поддерживал модель событий DOM уровня 2. В этих версиях браузера Microsoft был реализован собственный интерфейс, очень похожий на стадию всплытия в стандартной модели. Вместо `addEventListener()` в модели Internet Explorer для каждого элемента DOM существует метод `attachEvent()`. Как и в стандартной модели, данный метод принимает два параметра:

```
attachEvent(eventName, handler)
```

Первый параметр — это строка, определяющая тип обрабатываемого события. Но типы событий именуются согласно названиям свойств элемента, как в модели DOM уровня 0: `onclick`, `onmouseover`, `onkeydown` и т. п.

Второй параметр — это функция, которая назначается обработчиком, а экземпляр `Event` извлекается из свойства `window.event`. Кроме того, данный метод не поддерживает ничего похожего на стадию захвата.

Даже в случае применения относительно браузерно-независимой модели DOM уровня 0 на каждом этапе обработке события приходится постоянно учитывать особенности браузеров. А при использовании более эффективной модели DOM уровня 2 или модели Internet Explorer приходится даже разветвлять код по обработке событий, чтобы страница была доступна более широкой аудитории.

Библиотека скроет от вас данную проблему несовпадения браузеров и тем самым облегчит вашу работу. Посмотрим, как это делается!

6.2. Модель событий jQuery

Создание высокоинтерактивных приложений невозможно без интенсивной обработки событий, но сама мысль о том, что придется писать много кода с учетом всех различий браузеров, способна ужаснуть даже самых отважных авторов веб-страниц.

Но все эти различия можно скрыть за интерфейсом API, который абстрагирует их от кода страниц. Зачем беспокоиться, если jQuery сделает все за вас?

В модели событий (мы будем неформально называть ее моделью событий jQuery, поскольку она реализована в библиотеке) решены следующие задачи:

- унифицирован метод создания обработчиков событий;
- предоставлена возможность назначать несколько обработчиков для одного и того же события в одном и том же элементе;

- используется стандартное именование событий: `click`, `mouseover` и т. д.;
- первым аргументом обработчиков событий является экземпляр `Event`;
- для наиболее часто используемых свойств экземпляр `Event` нормализован;
- унифицированы методы отмены обработки события и блокирования действий по умолчанию.

К сожалению, стадия захвата не поддерживается. За этим исключением, функционал модели событий jQuery практически соответствует модели событий DOM уровня 2. Он предоставляет единый API как для стандартных браузеров, так и для старых версий Internet Explorer. Для подавляющего большинства разработчиков веб-страниц отсутствие стадии захвата не проблема, так как из-за отсутствия поддержки в старых версиях IE они никогда ею не пользуются (если вообще знают о ее существовании). Но действительно ли все так просто, как кажется? Проверим.

6.2.1. Назначение обработчиков событий в jQuery

Чтобы назначить обработчик события, в модели событий jQuery применяется метод `on()`. Мы уже видели, как назначить элементу один или несколько обработчиков событий, но одним из преимуществ библиотеки является использование эффективного метода `jQuery()`, который позволяет выбрать набор элементов и назначить им всем один и тот же обработчик в одной строке кода. Рассмотрим следующий простой пример:

```
$('img').on('click', function(event) {
    alert('Привет!');
});
```

Здесь обработчиком события `click` для каждого элемента `img` назначается встраиваемая анонимная функция. Полный синтаксис метода `on()` выглядит так.

Синтаксис метода: on

`on(eventType[, selector][, data], handler)`
`on(eventsHash[, selector][, data])`

Назначает функцию в качестве обработчика одного или нескольких событий для выбранных элементов.

Параметры

`eventType` (Строка) Тип или типы событий (полный список см. в табл. 6.1), для которых создается обработчик. Если типов событий несколько, то они перечисляются через запятую. Область применения событий может быть ограничена с помощью суффикса, определяющего пространство имен. Между именем события и суффиксом ставится точка (например, `click.myapp`). Можно указывать несколько пространств имен для одного события — например, `click.myapp.mymodule`.

`selector` (Строка) Необязательный селектор, используемый для *делегирования* события потомкам данного элемента. Если селектор отсутствует, то при достижении выбранного элемента событие всегда обрабатывается.

`data`	(Любой) Данные передаются экземпляру `Event` в виде свойства `data` и доступны для функции-обработчика.
`handler`	(Функция) Функция-обработчик события. При вызове ей в качестве первого аргумента передается экземпляр `Event`, а ее контекст (`this`) — это текущий элемент на стадии всплытия. Вместо функции, состоящей из единственной строки `return false;`, можно написать просто `false`. Данная функция также может получать дополнительные параметры за счет использования методов `trigger()` и `triggerHandler()`, которые мы обсудим ниже.
`eventsHash`	(Объект) Объект JavaScript, позволяющий назначать в одном вызове обработчики нескольких событий. Свойство `names` определяет тип события (как в параметре `eventType`), а свойство `value` — обработчик.

Возвращает

Коллекцию jQuery.

jQuery 3: улучшенная сигнатура

В jQuery 3 обработчиком события может быть объект. На момент написания данной книги об этой реализации было мало известно, но не беспокойтесь — она используется не так уж часто. Подробнее о свойстве вы можете узнать в статье на GitHub: https://github.com/jquery/jquery/issues/1735.

Прежде чем углубиться в детали данного ключевого метода jQuery, рассмотрим простой пример. Возьмем код из листинга 6.3 и преобразуем его из модели событий DOM уровня 2 в модель событий jQuery. Мы получим код, показанный в листинге 6.5 (вы также найдете его в файле `chapter-6/jquery.events.html`, предоставленном с книгой).

Листинг 6.5. Использование улучшенных обработчиков событий без браузерно-зависимого кода

```
<!DOCTYPE html>
  <html>
    <head>
      <title>Пример обработки событий в jQuery – jQuery
             в действии, 3-е издание</title>
      <link rel="stylesheet" href="../css/main.css"/>
      <style>
        img
        {
          display: block;
          margin: auto;
        }
      </style>
    </head>
    <body>
    <img id="example" src="images/example.jpg"
         alt="Вспышка молнии" />
```

```
    <script src="../js/jquery-1.11.3.min.js"></script>
    <script>
      function formatDate(date) {
        return (date.getHours() < 10 ? '0' : '') +
           date.getHours() +
           ':' + (date.getMinutes() < 10 ? '0' : '') +
           date.getMinutes() +
           ':' + (date.getSeconds() < 10 ? '0' : '') +
           date.getSeconds() +
           '.' + (date.getMilliseconds() < 10 ?
                 '00' : (date.getMilliseconds() < 100 ? '0' : '')) +
           date.getMilliseconds();
      }

      $('#example')
        .on('click', function (event) {
          console.log('В ' + formatDate(new Date()) +
                      ' БАБАХ один раз!');
        })
        .on('click', function (event) {
          console.log('В ' + formatDate(new Date()) +
                      ' БАБАХ два раза!');
        })
        .on('click', function (event) {
          console.log('В ' + formatDate(new Date()) +
                      ' БАБАХ три раза!');
        });
    </script>
  </body>
</html>
```

С помощью jQuery привязываем обработчики ❶ событий к элементу img

Изменения в коде просты, но очень важны — это способ назначения обработчиков событий ❶. Мы создаем здесь набор, состоящий из исходного элемента `img`, и трижды вызываем для него метод `on()`. Обратите внимание: библиотека позволяет вызывать для одного элемента несколько методов, объединяя их в цепочки. Каждый из указанных методов является обработчиком события `click` для данного элемента.

Если загрузить эту страницу в любой браузер, поддерживающий jQuery (наконец-то можно забыть о браузерах, совместимых и несовместимых со стандартом W3C!), и щелкнуть на картинке, то получим результат, показанный на рис. 6.6. Он, что неудивительно, ничем не отличается от рис. 6.3.

Данный код работает и в старых версиях Internet Explorer (в каких именно — зависит от используемой версии jQuery). Для кода из листинга 6.3 это было невозможно без проверки версии браузера и ветвления кода, чтобы выбрать корректную модель событий для данного браузера.

Следует отметить: в отличие от других методов jQuery в методе `on()` параметр `selector` не используется в момент вызова метода, чтобы затем фильтровать объекты коллекции jQuery, которые получат этот обработчик событий. Данный параметр применяется в момент наступления события и определяет, будет ли задействован

обработчик. Изложенный принцип станет более понятным немного позже, когда мы рассмотрим его на конкретном примере.

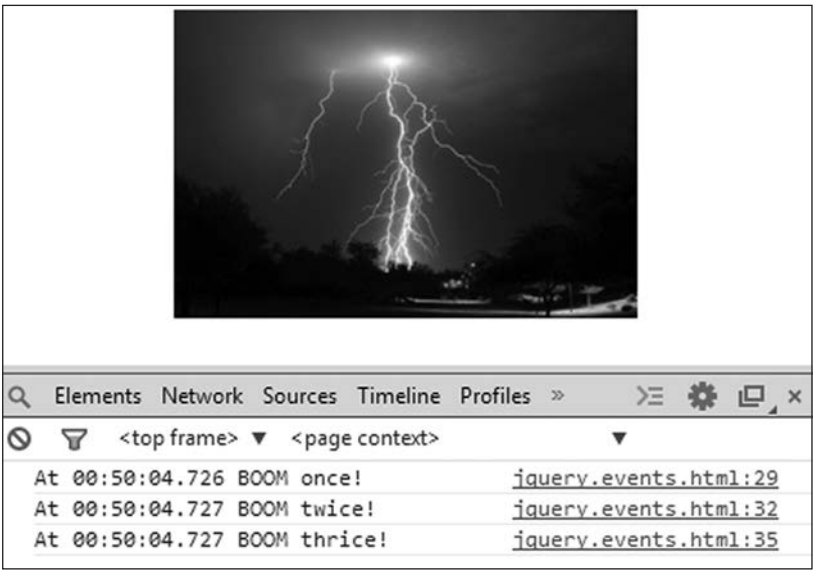


Рис. 6.6. Использование модели событий jQuery позволит вам задать несколько обработчиков событий, как в модели событий DOM уровня 2

jQuery 3: устаревшие методы

Метод `on()` обеспечивает единый интерфейс, который пришел на смену методам `bind()`, `delegate()` и `live()`. Метод `live()` устарел еще в версии 1.7, а в версии 1.9 был удален. Методы `bind()` и `delegate()` все еще существуют, но их настоятельно не рекомендуется использовать. В jQuery 3 методы `bind()` и `delegate()` рассматриваются как устаревшие, так что мы советуем придерживаться метода `on()` (и поэтому не включили их описание в эту книгу).

У метода `on()` есть важное отличие от нативных методов JavaScript — в нем иначе указываются обработчики событий. Когда обработчик события, назначенный с помощью jQuery, возвращает `false` — это то же самое, как если бы были вызваны методы `preventDefault()` и `stopPropagation()`, а возврат `false` в обработчике события, назначенном с использованием чистого JavaScript, эквивалентен вызову только `preventDefault()`.

В этот момент разработчики, уже успевшие написать множество строк браузерно-зависимого кода по обработке событий, уже наверняка празднуют хеппи-энд, вертясь на своих офисных стульях. И кто их осудит?

В листинге 6.5 показано, насколько гибким и умным может быть метод `on()`. Если нужно только передать обработчик без указания данных или селектора, то

не потребуется передавать null вместо аргументов. Метод on() позволяет передать обработчик события в качестве второго параметра. Вместо кода:

```
$('img').on('click', null, null, function() { ... });
```

можно написать:

```
$('img').on('click', function() { ... });
```

В данной книге мы не будем рассматривать, как такое возможно, но настоятельно рекомендуем вам ознакомиться с исходным кодом метода, чтобы понять это.

Как мы уже знаем из описания метода on(), его первым параметром может быть список событий, разделенных запятой, например:

```
$('button')
    .on('click', function(event) {
      console.log('Button clicked!');
    })
    .on('mouseenter mouseleave', myFunctionHandler);
```

В этом простом фрагменте выбираются все элементы <button> на странице и им назначается три обработчика событий. Первый из них — анонимная функция, срабатывающая при наступлении события click. Второй — функция myFunctionHandler (очевидно, где-то определенная), выполняющаяся при наступлении события mouseenter или mouseleave.

В описании сигнатуры метода on() мы также обратили внимание на то, что ее первым параметром может быть объект JavaScript eventsHash, в котором свойство names определяет тип события, а свойство value — обработчик. Для данного случая предыдущий пример можно переписать так:

```
$('button').on({
  click: function(event) {
    console.log('Button clicked!');
  },
  mouseenter: myFunctionHandler,
  mouseleave: myFunctionHandler
});
```

Каким из вариантов пользоваться — решать вам, но мы советуем выбрать один стиль и придерживаться его.

Иногда возникает необходимость передать обработчику события некоторые данные. Конечно, можно сохранить их в переменной и воспользоваться замыканием, но иногда проще обойтись без этого и использовать параметр data. Предположим, у нас есть обработчик, назначенный событию click для кнопки, и он выводит в консоль чье-то полное имя. Данное имя можно передать в параметре data:

```
$('#my-button').on('click', {
  name: 'Джон Резиг'
}, function (event) {
    console.log('Имя: ' + event.data.name);
});
```

Как видите, доступ к свойству `name` происходит через свойство `data` экземпляра `Event` (`event`). Рабочий пример кода доступен в файле `chapter-6/on.data.parameter.html`, предоставленном с книгой, а также в JS Bin (http://jsbin.com/IVONuWol/edit?html,js,console,output).

До сих пор мы назначали события элементам DOM, уже существующим в HTML-разметке страницы. Однако как быть, если элемента еще нет, но позже он появится?

Как мы уже говорили, jQuery позволяет динамически манипулировать деревом DOM — добавлять, изменять и удалять его элементы. Когда мы добавим еще и Ajax, элементы DOM будут появляться и исчезать постоянно, все время существования страницы, особенно в одностраничных приложениях.

Решение данной задачи называется *делегированием событий*. Это важная технология, позволяющая назначить обработчик не самому элементу (или элементам), а его (их) предку. Делегирование можно использовать и в чистом JavaScript, но это тот случай, когда возможности jQuery проявляются во всей своей полноте. Представим, что у нас есть пустой маркированный список. Позже он заполнится с помощью вызова Ajax и будет состоять из пяти элементов. Но на момент выполнения сценария он еще пуст:

```
<ul id="my-list"></ul>
```

После вызова Ajax список станет таким:

```
<ul id="my-list">
  <li>Item 1</li>
  <li>Item 2</li>
  <li>Item 3</li>
  <li>Item 4</li>
  <li>Item 5</li>
</ul>
```

Мы хотим печатать в консоли индекс элемента списка, на который наведен указатель мыши. Сравним, как можно решить эту задачу путем делегирования события в чистом JavaScript и с использованием jQuery.

В чистом JavaScript код выглядит так:

```
document.getElementById('my-list').addEventListener('mouseover',
  function(event) {
    if (event.target.nodeName === 'LI') {
      console.log('Элемент списка: ' +
        (Array.prototype.indexOf.call(
          document.getElementById('my-list').children,
          event.target
        ) + 1)
      );
    }
  },
  false
);
```

Благодаря jQuery код можно сократить всего до трех строк (если сохранять тот же стиль):

```
$('#my-list').on('mouseover', 'li', function(event) {
  console.log('Элемент списка: ' + ($(this).index() + 1));
});
```

Какой чистый и понятный код! Вы думаете, все ограничивается только уменьшением количества строк? Ничуть! Вы заметили, что в jQuery, в отличие от чистого JavaScript, учитываются различия браузеров? Разумеется, «за кулисами» библиотека выполняет приблизительно те же самые действия, но зачем беспокоиться о проблемах совместимости, если jQuery сделает это за вас? Вот теперь вы, вероятно, начинаете понимать, почему библиотека получила такое широкое распространение в разработке сайтов. Если вы захотите поэкспериментировать с примерами, то код чистого JavaScript и код с примером делегирования событий на jQuery доступны в файлах `chapter-6/javascript.event.delegation.html` и `chapter-6/jquery.event.delegation.html`, предоставленных с книгой, и в JS Bin (http://jsbin.com/fihixa/edit?html,js,console,output и http://jsbin.com/bobaza/edit?html,js,console,output).

Преимущества делегирования событий не ограничиваются запуском обработчиков для не существующих пока элементов. Оно также экономит время и память. Предположим, после вызова Ajax в списке появилось не пять элементов, как в нашем примере, а несколько сотен. Чтобы назначить обработчик непосредственно элементам списка, нужно перебрать их все (это можно сделать с помощью метода `jQuery()`, но «за кулисами» будет совершен все тот же поэлементный перебор). Такая задача потребовала бы немало времени, возможно, даже несколько сотен миллисекунд. Пока выполняется цикл, пользовательский интерфейс браузера блокирован и пользователь жалуется на низкую эффективность. Кроме того, поскольку JavaScript — однопоточный язык, во время выполнения цикла все остальные операции тоже приостановлены. Очевидно, если назначить обработчик только одному (родительскому) элементу, то времени уйдет меньше. Мы также упомянули, что делегирование событий требует меньше памяти. В данном примере вместо хранения нескольких сотен обработчиков приходится хранить только один. Ощутимая экономия!

При делегировании событий иногда обработчик назначается элементу `document`. Тогда измененный вариант нашего примера будет выглядеть так:

```
$(document).on('mouseover', '#my-list li', function(event) {
    console.log('Элемент списка: ' + ($(this).index() + 1));
});
```

Обратите внимание на то, как изменился параметр `selector`, чтобы обработчик назначался тому же набору элементов. После этого примера вам может понравиться идея назначать все обработчики элементу `document`, однако назначение большого количества делегированных обработчиков `document` или другому элементу, находящемуся вблизи корня DOM, может привести к потере производительности. В таких случаях каждый раз при возникновении события jQuery сравнивает параметр `selector` с элементом, вызвавшим событие, — и так для всех элементов от исходного до верхнего (по схеме всплытия). Чтобы этого не произошло, считается

хорошим стилем назначать делегированные события элементу, расположенному как можно ближе к тому, который будет данное событие генерировать.

Все это время мы уделяли много внимания событиям и даже использовали несколько из них. Но сколько их всего и какие они? В табл. 6.1 представлен полный список событий, для которых можно назначать обработчики.

Таблица 6.1. События, доступные для обработки

События			
blur	focusin	mousedown	mouseup
change	focusout	mouseenter	ready
click	keydown	mouseleave	resize
dblclick	keypress	mousemove	scroll
error	keyup	mouseout	select
focus	load	mouseover	submit
			unload

Теперь, когда вы все знаете о том, как назначаются обработчики событий, обсудим вариант метода `on()`, позволяющий назначить обработчик, который вызывается только один раз, а потом удаляется.

Однократная обработка события. В jQuery есть специальный вариант метода `on()`, называемый `one()`, позволяющий создавать обработчик события, выполняющийся только один раз. Как только операция завешится, он будет удален. Его синтаксис идентичен методу `on()`, так что мы пропустим описание его параметров (если вам требуется освежить память, обратитесь к описанию синтаксиса данного метода).

Синтаксис метода: one

```
one(eventType[, selector][, data], handler)
one(eventsHash[, selector][, data])
```

Определяет функцию-обработчик заданного типа событий для указанного набора элементов. После первого выполнения обработчик автоматически удаляется.

Возвращает

Коллекцию jQuery.

Теперь вы знаете, как привязать обработчик события к набору элементов. Однако иногда обработчики нужно не только назначать, но и удалять. Посмотрим, как это делается.

6.2.2. Удаление обработчиков событий

Обычно обработчик события, назначенный с помощью метода `on()`, существует до тех пор, пока существует страница. Однако бывают особые обстоятельства, требующие удалить обработчик при наступлении тех или иных условий. Например,

предположим, что на странице выполняется последовательность действий и после очередного действия все элементы формы на данной странице переводятся в режим «только для чтения».

В этом случае было бы удобно удалить из сценария страницы все обработчики событий и таким образом сэкономить память. Как мы уже знаем, метод `one()` позволяет автоматически удалить обработчик после того, как он выполнится в первый (и последний) раз. Но в более общих случаях, когда нужно удалить обработчики событий при наступлении произвольного условия, в jQuery используется антипод метода `on()` под названием `off()`. Синтаксис данного метода представлен ниже. Его параметры имеют то же назначение, что и в методах `on()` и `one()`, так что не будем повторяться.

Синтаксис метода: off

```
off(eventType[, selector][, handler])
off(eventsHash, [, selector])
off()
```

Удаляет обработчики событий для всех элементов объекта jQuery с учетом последующих (необязательных) параметров. Если параметры не заданы, то удаляются все слушатели для указанного набора элементов.

Возвращает

Коллекцию jQuery.

Этот метод можно использовать для удаления обработчиков событий, назначенных элементам из объекта jQuery, с разной степенью детализации. Если у метода `off()` нет параметров, то будут удалены все слушатели. Чтобы удалить только слушатели определенного типа, нужно указать в методе данный тип событий. Если в методе указан один или несколько типов событий, то будут удалены все их обработчики, независимо от делегирования. Наконец, можно удалить слушатель, указав имя его функции. Для этого ранее, при создании слушателя события, надо сохранить ссылку на функцию-обработчик. По этой причине, если функцию-слушатель планируется впоследствии удалить, то либо она определяется как функция верхнего уровня (так что на нее можно сослаться по ее имени переменной), либо каким-то другим способом создается доступная ссылка на нее. Если функция описана как анонимная и встроенная, то впоследствии сослаться на нее в методе `off()` невозможно.

jQuery 3: устаревшие методы

Метод `off()` предоставляет унифицированный интерфейс, заменяющий методы `unbind()`, `undelegated()` и `die()` (какое страшное имя!). Как соответствующий противоположный метод, `die()` (обратный метод `live()`) был объявлен устаревшим в версии 1.7, а в версии 1.9 был удален. Что же касается методов `unbind()` (противоположность `bind()`) и `undelegate()` (противоположность `delegate()`), то они все еще присутствуют в ядре, но использовать их не рекомендуется.

В jQuery 3 методы `unbind()` и `undelegate()` объявлены устаревшими, так что мы советуем применять вместо них метод `off()`.

В случае анонимных функций очень кстати оказывается ограничение действия событий заданным пространством имен — в этом случае можно отменить обработку событий, принадлежащих определенному пространству имен, сохранив ссылки на слушатели. Например, такой код:

```
$('*').off('.fred');
```

удалит слушателей всех событий в пространстве имен `fred` (напомним, в данном случае точка в начале имени не означает селектор класса). Такое применение пространства имен особенно полезно, когда обработчики событий назначаются из плагина jQuery — как это делается, будет показано в главе 12.

Прежде чем двинуться дальше, рассмотрим пример использования методов `on()` и `off()`. Предположим, у нас есть такая разметка с тремя кнопками:

```
<button id="btn">Ничего не делать</button>
<button id="btn-attach">Назначить обработчик</button>
<button id="btn-remove">Удалить обработчик</button>
```

Далее у нас есть следующий код:

```
var $btn = $('#btn');
var counter = 1;

function logHandler() {
    console.log('click ' + counter);
    counter++;
};
$('#btn-attach').on('click', function() {
    $btn
        .on('click', logHandler)
        .text('Log');
});
$('#btn-remove').on('click', function() {
    $btn
        .off('click', logHandler)
        .text('Does nothing');
});
```

❶ Определяет функцию, выводящую в консоль сообщение о том, сколько раз она была выполнена

❷ Назначает обработчик события click для кнопки с идентификатором btn-attach

❸ Назначает обработчик события click для кнопки с идентификатором btn-remove

Итак, у нас есть первая кнопка, не делающая ничего, и две других, к которым привязаны обработчики событий. Идея состоит в том, чтобы назначить обработчик первой кнопке, а с другой кнопки его соответственно убрать. Сам обработчик выводит в консоль количество нажатий кнопки, к которой он привязан ❶.

Чтобы выполнить эту операцию, в начале кода объявлена переменная `$btn`. Она содержит набор из одного элемента — первой кнопки (с ID `btn`). Затем объявлены счетчик (`counter`) и функция, которую мы затем используем в качестве обработчика событий для кнопки. В следующей части кода второй кнопке (с ID `btn-attach`)

с помощью метода `on()` назначается встроенный обработчик событий ❷. Наша цель — назначить функцию `logHandler` обработчиком событий для первой кнопки после того, как будет обработано событие `click`. Третьей кнопке (с ID `btn-remove`) тоже назначен встроенный обработчик событий, который должен удалить обработчик события первой кнопки ❸. Рабочий пример с этим кодом доступен в файле `chapter-6/adding.removing.handlers.html`, предоставленном с книгой, и в JS Bin (http://jsbin.com/iquRujug/edit?html,js,console,output). Обратите внимание: если несколько раз нажать кнопку Attach handler (Назначить обработчик), то ей будут добавлены соответствующее количество обработчиков событий, так что счетчик будет реагировать на каждое нажатие кнопки Log (Записать).

Итак, вы уже поняли, что модель событий jQuery упрощает назначение (и удаление) обработчиков событий и позволяет забыть о различиях между браузерами. А что насчет кода самих обработчиков?

6.2.3. Внутри экземпляра Event

После того как обработчик события назначен — с помощью метода `on()` или другим уже известным нам способом — в качестве первого параметра функции-обработчику, независимо от используемого браузера, передается экземпляр объекта `Event`, поэтому можно не беспокоиться о свойстве `window.event` в старых версиях Internet Explorer. Но тогда получается, что нам все равно приходится иметь дело с разными свойствами объекта `Event`?

К счастью, это не так. В действительности jQuery не передает обработчикам экземпляр `Event`. Вот так оборот (в голове слышен скрип мозгов)!

Да, мы изящно умолчали об этой маленькой детали, поскольку до сих пор она не имела значения. Но теперь, когда мы знаем достаточно, чтобы изучить поведение экземпляра `Event` внутри обработчиков событий, настало время сказать правду.

В действительности jQuery определяет объект типа `jQuery.Event`, который и передается обработчикам событий. Но вы можете простить нам это упрощение, поскольку jQuery копирует в данный объект большинство свойств исходного экземпляра `Event`. Таким образом, если вам нужны только те свойства, которые вы ожидали найти в `Event`, этот объект практически ничем не отличается от исходного экземпляра объекта `Event`.

Но не это главное в объекте `jQuery.Event`. Вот что действительно важно и оправдывает его существование — в нем хранится набор нормализованных значений и методов, которые можно использовать независимо от браузера, игнорируя различия в реализации экземпляра `Event`.

В табл. 6.2 представлен список свойств объекта `jQuery.Event`, которые можно уверенно использовать независимо от платформы. Обратите внимание: отдельные свойства для некоторых событий имеют значение `undefined`.

Таблица 6.2. Браузеро-независимые свойства объекта jQuery.Event

Свойства			
altKey	currentTarget	offsetY	screenX
bubbles	data	originalTarget	screenY
button	detail	originalEvent	shiftKey
cancelable	delegateTarget	pageX*	target*
charCode	eventPhase	pageY*	timeStamp
clientX	metaKey*	prevValue	type
clientY	namespace	relatedTarget*	view
ctrlKey	offsetX	result	which*

Как видите, некоторые свойства помечены звездочкой. Это означает, что jQuery нормализует их для кросс-браузерной совместимости. Другими словами, в некоторых браузерах (да, именно в старых версиях Internet Explorer) у данных свойств другие имена.

Чтобы избавить вас от хлопот, связанных с этими различиями, в jQuery приняты единые имена свойств, а остальное библиотека берет на себя. Например, свойство `target` в старых версиях Internet Explorer называется `srcElement`. Кроме того, есть свойства, доступные для некоторых событий через свойство `originalEvent`.

У объекта `jQuery.Event` также есть несколько методов, описанных в табл. 6.3.

Таблица 6.3. Браузерно-независимые методы объекта jQuery.Event

Методы	
preventDefault()	Предотвращает любые семантические действия по умолчанию (такие как передача формы, переход по ссылке, изменение состояния переключателя и т. п.)
stopPropagation()	Останавливает дальнейшее прохождение события по дереву DOM. На другие события для данного элемента это не влияет. Применяется как для стандартных событий браузера, так и для событий, созданных разработчиком
stopImmediatePropagation()	Останавливает дальнейшую обработку событий, в том числе и других событий для данного элемента
isDefaultPrevented()	Возвращает true, если для данного экземпляра объекта jQuery.Event был вызван метод preventDefault()
IsPropagationStopped()	Возвращает true, если для данного экземпляра объекта jQuery.Event был вызван метод stopPropagation()
isImmediatePropagationStopped()	Возвращает true, если для данного экземпляра объекта jQuery.Event был вызван метод stopImmediatePropagation()

jQuery не только позволяет управлять обработкой событий и объектом `Event`, не заботясь о различиях между браузерами, но и предоставляет набор методов, дающих возможность вызывать события или запускать обработчики под управлением сценария. Рассмотрим это подробнее.

6.2.4. Запуск обработчиков событий

Обработчики событий предназначены для того, чтобы запускаться в момент, когда действие браузера или пользователя активизирует прохождение соответствующего события по иерархии DOM. Но иногда возникает потребность запустить обработчик события под управлением сценария. Такие обработчики событий определяют в виде функций верхнего уровня, чтобы затем вызывать их по имени. Но, как видите, гораздо чаще обработчики событий определяют в виде встроенных анонимных функций — ведь это очень удобно! Более того, вызов обработчика события по имени функции не позволяет активизировать семантические действия и обработку методом всплытия. Поэтому в jQuery реализованы методы, позволяющие запустить обработчик события под управлением сценария. Самым общим из них является `trigger()`. Данный метод имеет следующий синтаксис.

Синтаксис метода: trigger

`trigger(eventType[, data])`

Запускает все обработчики и другие реакции, связанные с заданным типом событий для всех выбранных элементов.

Параметры

`event`	(Строка|jQuery.Event) Тип событий, для которых запускаются обработчики, включая события в пространстве имен, или объект `jQuery.Event`.
`data`	(Любой) Данные для обработчиков. Если это массив, то его элементы передаются обработчику как отдельные параметры.

Возвращает

Коллекцию jQuery.

Метод `trigger()` и другие более удобные методы, которые мы представим позже, полностью имитируют запуск события, включая его прохождение по иерархии DOM и выполнение семантических действий.

Каждому вызываемому обработчику передается заполненный значениями экземпляр `jQuery.Event`. Поскольку это не настоящее событие, свойства, значения которых зависят от конкретных событий, такие как координаты указателя мыши или код нажатой клавиши, не передаются (при этом свойства вообще не существуют, а не просто имеют значение `undefined`). Свойство `target` указывает на тот элемент из набора, к которому было привязано событие.

Как и для настоящих событий, вызванное таким способом прохождение обработчика по дереву DOM может быть прервано, если вызвать метод `stopPropagation()` для экземпляра `jQuery.Event` или вернуть `false` из любого вызванного обработчика.

Параметр `data`, передаваемый методу `trigger()`, отличается от того, что передается при запуске обработчика события. В последнем случае данные помещаются внутрь экземпляра `jQuery.Event` в качестве свойства `data`, а методу `trigger()` (и, как мы скоро убедимся, методу `triggerHandler()`) передается как параметр функции. Это позволяет использовать оба значения `data`, не вызывая конфликта между ними.

Прежде чем перейти к методу `triggerHandler()`, обратим внимание на параметр `data` метода `trigger()` и то, чем передача массива отличается от передачи данных другого типа. Рассмотрим для примера следующий код, где обработчик события для гипотетического элемента с ID `foo` выполняется при возникновении события `click`:

```
$('#foo').on('clic¢1, par2, par3);
});
```

Как видите, параметр `event` задан как обычно, но к нему добавлены еще три параметра: `par1`, `par2`, и `par3`, значения которых обработчик выводит в консоль. Теперь представим, будто мы воспользовались методом jQuery `trigger()`, чтобы запустить этот обработчик, как показано далее, и передали ему, кроме ссылки на активизируемое событие, три числа — `1`, `2` и `3`:

```
$('#foo').trigger('click', 1, 2, 3);
```

В результате в консоли вы увидите следующее:

```
1 undefined undefined
```

Так произошло потому, что метод `trigger()` принимает только один аргумент, который передает данные обработчику, а остальные (в данном случае `2` и `3`) игнорируются. Если вы хотите передать несколько аргументов, то их надо объединить в массив:

```
$('#foo').trigger('click', [1, 2, 3]);
```

Теперь при выполнении обработчика, назначенного данному элементу, в консоль будет выведена следующая строка (доказывающая, что все элементы массива переданы в виде отдельных параметров и `par2` и `par3` больше не равны `undefined`):

```
1 2 3
```

На случай, если вы захотите еще поэкспериментировать с параметром `data`, мы подготовили пример, который доступен в файле `chapter-6/trigger.data.parameter.html`, предоставленном с книгой, и в JS Bin (http://jsbin.com/mefohu/edit?html,js,console,output).

Для тех ситуаций, когда нужно запустить обработчик, но не передавать событие по дереву DOM и не выполнять семантических действий, в jQuery есть метод `triggerHandler()`. Он выглядит и действует так же, как `trigger()`, но только без передачи управления родительским элементам и без совершения семантических действий.

Синтаксис метода: triggerHandler

`triggerHandler(eventType[, data])`

Вызывает все обработчики, определенные для данного типа событий, для всех элементов из указанного множества, без прохождения по дереву DOM, семантических действий и выполнения настоящих событий. Этот метод возвращает то значение, которое было возвращено последним из запущенных обработчиков, или же `undefined`.

Параметры

`eventType` (Строка) Тип событий, для которого вызываются обработчики.

`data` (Любой) Данные, передаваемые обработчикам. Если передается массив, то его элементы будут переданы обработчику как отдельные параметры.

Возвращает

Значение произвольного типа. Если не был вызван ни один обработчик или он не возвращал значений, то возвращается `undefined`.

Как и у `trigger()`, у метода `triggerHandler()` есть параметр `data`, так что предыдущий пример будет работать и для `triggerHandler()`. Но здесь есть ряд важных различий, которые заслуживают внимания. Прежде всего, `trigger()` воздействует на все элементы коллекции jQuery, а `triggerHandler()` — только на первый из них. Затем, после `triggerHandler()` нельзя продолжить цепочку методов jQuery, поскольку он возвращает то же значение, что и последний выполненный обработчик (или `undefined`, если ни один обработчик не сработал либо не возвращал значение), а `trigger()` возвращает набор заданных элементов. Наконец, события, активизированные в `triggerHandler()`, не передаются вверх по иерархии DOM.

Впрочем, никакие описания не заменят хороший пример. Поэтому, прежде чем перейти к следующему разделу, сравним `trigger()` и `triggerHandler()` в действии (наша книга называется «jQuery в действии», верно?). Код примера показан в листинге 6.6 и доступен в файле `chapter-6/jquery.triggering.events.html` и в JS Bin (http://jsbin.com/AqaqAGO/edit?html,css,js,console,output).

Листинг 6.6. Запуск событий в jQuery

```
<!DOCTYPE html>
<html>
   <head>
      <title>Запуск событий – jQuery в действии,
```

```
            3-е издание</title>
      <link rel="stylesheet" href="../css/main.css"/>
      <style>
         #wrapper
         {
            border: 1px solid #3A5895;
            padding: 10px;
         }
         #address:focus
         {
            border: 3px solid #000000;
         }
      </style>
   </head>
   <body>
      <div id="wrapper">
         <button id="btn">Щелкните здесь!</button>
         <input type="text" id="address" />
      </div>

      <script src="../js/jquery-1.11.3.min.js"></script>
      <script>
         $('#wrapper')
            .on('focus', function() {
               console.log('Div focused');
            })
            .on('click', function() {
               console.log('Div clicked');
            });
         $('#address')
            .on('focus', function() {
               console.log('Input focused');
            })
            .triggerHandler('focus');
         $('#btn')
            .on('click', function() {
               console.log('Button clicked');
            })
            .trigger('click');
      </script>
   </body>
</html>
```

Назначаем элементу div два обработчика — для событий focus и click

Назначаем элементу input обработчик, который запускается при активизации события focus

Назначаем элементу button обработчик, который будет выполняться при наступлении события click

С помощью метода triggerHandler() активируем событие focus для элемента input

С помощью метода trigger() активизируем событие click для элемента button

Код в этом листинге демонстрирует поведение описанных ранее методов `trigger()` и `triggerHandler()`. Открыв страницу, вы должны увидеть макет, показанный на рис. 6.7.

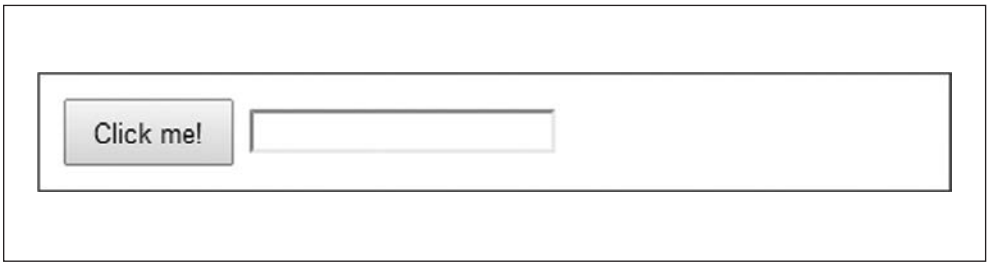


Рис. 6.7. Макет страницы jquery.triggering.events.html

В листинге 6.6 элемент `<div>` играет роль обертки для двух других элементов страницы — кнопки и поля ввода. В обычном элементе `<script>` мы сначала выбираем элемент-обертку и назначаем ему с помощью метода `on()` два слушателя: для событий `focus` и `click` ❶. Эти обработчики выводят в консоль сообщения о том, какое событие и для какого элемента было активизировано. Затем мы назначаем обработчик события `focus` для элемента `input`, выбрав его по ID ❷. Благодаря объединению методов jQuery в цепочки, прежде чем закрыть данное выражение точкой с запятой, мы также вызываем метод `triggerHandler()` и передаем ему в качестве аргумента строковое значение `focus` ❸. Вызов `triggerHandler()` запускает выполнение функции, только что назначенной обработчиком события на шаге ❷. Как уже говорилось, метод `triggerHandler()` не передает обработку дальше, поэтому обработчик, назначенный элементу-обертке по его ID, не будет выполнен.

Наконец, мы выбираем элемент `button` и назначаем ему обработчик события `click` ❹. Как и для элемента `input`, прежде чем закрыть выражение, мы активизируем событие, но на сей раз используем метод `trigger()` ❺. Поэтому после вывода в консоль сообщения данного обработчика событие передается родительскому элементу и выполняется обработчик события `click` для элемента `div`.

После обсуждения этого примера вы поймете, почему, загрузив страницу, увидите в консоли то, что показано на рис. 6.8.

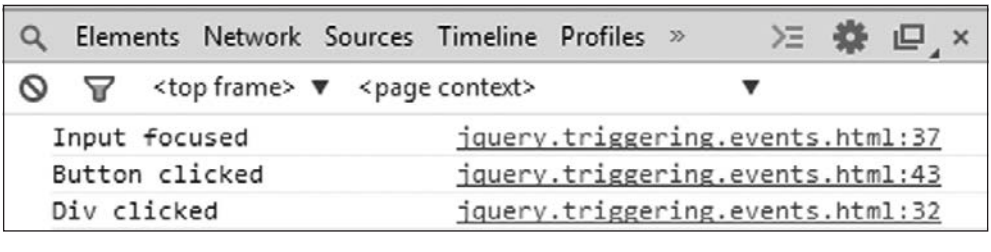


Рис. 6.8. Консольный вывод для страницы jquery.triggering.events.html

Методы `on()` и `trigger()` использовались так часто, что разработчикам вскоре надоело каждый раз писать их полный синтаксис. Поняв это, команда jQuery представила набор сокращенных методов.

6.2.5. Сокращенные методы

В jQuery есть целый ряд сокращенных методов для назначения обработчиков конкретных событий, а также для активизации этих событий. Поскольку синтаксис у них одинаковый — разница только в имени метода, — мы сэкономим немного места в книге и дадим им всем одно общее описание.

Синтаксис метода, назначающего обработчик определенного события

`eventName([data,] handler)`

Назначает заданную функцию обработчиком события, тип которого соответствует имени метода. Поддерживаются следующие методы:

```
blur        focusout    mouseleave  resize
change      keydown     mousemove   scroll
click       keypress    mouseout    select
dblclick    keyup       mouseover   submit
focus       mousedown   mouseup
focusin     mouseenter  ready
```

Параметры

`data` (Любой) Данные, передаваемые экземпляру `Event` в виде свойства `data` и доступные функции-обработчику.

`handler` (Функция) Функция, назначаемая обработчиком события.

Возвращает

Коллекцию jQuery.

Представленные методы не только позволяют назначить обработчик события, но и могут использоваться вместо метода `trigger()`. Их синтаксис в этом случае тоже одинаковый, за исключением имени метода.

Синтаксис метода активизации определенного события

`eventName()`

Запускает все обработчики события и другие поведенческие реакции, назначенные данному типу событий для всех элементов заданного набора. Поддерживаются следующие методы.

```
blur        focusout    mouseleave  resize
change      keydown     mousemove   scroll
click       keypress    mouseout    select
dblclick    keyup       mouseover   submit
focus       mousedown   mouseup
focusin     mouseenter  ready
```

Параметры

Отсутствуют.

Возвращает

Коллекцию jQuery.

jQuery 3: удаленные методы

В jQuery 3 избавились от устаревших сокращенных методов `load()`, `unload()` и `error()`. Эти методы не включены в предыдущее описание, поскольку уже давно устарели (еще в jQuery 1.8). Если вы все еще пользуетесь ими в своих проектах или используете плагины, в которых есть хотя бы один из них, то обновление до jQuery 3 сделает ваш код неработоспособным.

Чтобы лучше понять, как работают сокращенные методы, используем один из них, изменив последнее выражение в листинге 6.6. Для удобства приведем это выражение здесь еще раз:

```
$('#btn')
  .on('click', function() {
    console.log('Button clicked');
  })
  .trigger('click');
```

Если использовать сокращенный метод, данное выражение будет выглядеть так:

```
$('#btn')
  .click(function() {
    console.log('Button clicked');
  })
  .click();
```

В jQuery есть еще один сокращенный метод, немного отличающийся от описанных выше.

Наведение указателя мыши на элемент. В многособытийных сценариях интерактивных приложений часто используется событие наведения указателя мыши на элемент. События, информирующие о том, что указатель мыши вошел в некоторую область или покинул ее, являются основой построения многих широко распространенных элементов пользовательского интерфейса веб-страниц. Типичный пример таких элементов — каскадные меню и элементы навигации.

Капризное поведение событий `mouseover` и `mouseout` часто затрудняет создание таких элементов: событие `mouseout` возникает, когда указатель мыши попадает в область самого элемента и всех его потомков. Рассмотрим рис. 6.9. Здесь отражена структура файла `chapter-6/hover.html`, предоставленного с книгой, а также доступного в JS Bin (http://jsbin.com/nobuti/edit?html,js,console,output).

На странице возникают два одинаковых события с разными именами. Одно из них активизируется, когда указатель мыши попадает во внешнюю область, второе — когда во внутреннюю. Прежде чем читать дальше, откройте эту страницу в браузере.

Для верхней пары прямоугольников определены следующие обработчики событий `mouseover` и `mouseout`:

```
$('#outer1').on('mouseover mouseout', report);
$('#inner1').on('mouseover mouseout', report);
```

Рис. 6.9. Эта страница помогает продемонстрировать, как запускаются события мыши, когда указатель попадает в зону самого элемента и его потомков

В этих выражениях функция с именем `report` назначается обработчиком событий `mouseover` и `mouseout`. Описание данной функции такое:

```
function report(event) {
  event.stopPropagation();
  console.log(event.type + ' для элемента ' + event.target.id);
}
```

Этот обработчик вначале останавливает дальнейшее продвижение события по дереву DOM, а затем выводит в консоль имя события и ID элемента, для которого оно было активизировано.

Теперь наведем указатель мыши на область элемента, обозначенного как «Внешний 1» (аккуратно, не заденьте «Внутренний 1»). В консоли появится сообщение об активизации события `mouseover`. Если убрать указатель из данной области, то, как и ожидалось, мы увидим сообщение о событии `mouseout`.

Теперь снова наведем указатель на «Внешний 1», но на этот раз достаточно глубоко, так, чтобы попасть и на «Внутренний 1». Когда указатель попадет в его область, будет активизировано событие `mouseover` для этого элемента и `mouseout` — для элемента «Внешний 1». Если поводить указателем мыши по границе обоих элементов, то получим целый шквал событий `mouseout` и `mouseover`. Это обычное, хоть и не совсем логичное поведение. Хоть указатель мыши и не покинул границ родительского элемента «Внешний 1», когда он попадает в область внутреннего элемента, модель событий считает данный переход выходом за пределы внешнего элемента.

Логично оно или нет, но такое поведение далеко не всегда желательно. Часто нам нужна информация о том, покинул ли указатель мыши границы внешнего элемента, и нас не интересует, попадает указатель в область внутренних элементов или нет.

К счастью, в основных браузерах определена другая пара событий, `mouseenter` и `mouseleave`, впервые представленных компанией Microsoft в Internet Explorer. Поведение этих событий несколько логичнее, событие `mouseleave` не возникает при переходе в область дочерних элементов.

jQuery позволяет назначать обработчики для данной пары событий, используя следующий код:

```
$(element).mouseenter(function1).mouseleave(function2);
```

Однако в jQuery также есть метод `hover()`, который сильно упрощает дело. Синтаксис этого метода следующий.

Синтаксис метода: hover

`hover(enterHandler, leaveHandler)`
`hover(handler)`

Назначает обработчики событий `mouseenter` и `mouseleave` для указанных элементов. Обработчики срабатывают, только когда указатель мыши попадает в область этих элементов или покидает ее. Переход к дочерним элементам игнорируется.

Параметры

`enterHandler`	(Функция) Функция — обработчик события `mouseenter`.
`leaveHandler`	(Функция) Функция — обработчик события `mouseleave`.
`handler`	(Функция) Общий обработчик для событий `mouseenter` и `mouseleave`.

Возвращает

Коллекцию jQuery.

Воспользуемся следующим сценарием и назначим обработчики событий мыши для второго набора элементов («Внешний 2» и его потомок «Внутренний 2») в нашем примере:

```
$('#outer2').hover(report);
$('#inner2').hover(report);
```

Как и для первого набора элементов, обработчиком обоих событий, `mouseenter` и `mouseleave`, для элементов «Внешний 2» и «Внутренний 2» является функция `report()`. Но, в отличие от первого набора элементов, когда указатель мыши попадает на границу между элементами, ни один из обработчиков (для «Внешнего 2») не будет вызван. Это удобно для тех случаев, когда родительские обработчики не должны реагировать на попадание указателя мыши в область дочерних элементов.

Теперь двинемся дальше и узнаем, как с помощью jQuery можно создавать *пользовательские события*.

6.2.6. Создание пользовательских событий

Создать пользовательское событие в jQuery очень легко. Для этого используются методы, о которых мы уже писали. Пользовательские события — удобный способ выполнить одно или несколько выражений при наступлении определенного условия в той или иной части кода.

Предположим, нам надо выполнить ряд логически связанных выражений. Их можно объединить в обработчик пользовательского события и затем при необходимости вызывать это событие. Откровенно говоря, нам не нужно создавать само событие — достаточно лишь отслеживать соблюдение условия и запускать обработчик. Это значит, что можно, используя метод `on()`, назначить обработчик пользовательского события и передать данному методу имя нового события в качестве первого параметра. Затем остается активизировать событие с помощью метода `trigger()`, передав ему то же имя.

Вот простейший пример создания и использования пользовательского события:

```
$('#btn').on('customEvent', function(){
    alert('customEvent');
});
$('#anotherBtn').click(function() {
    $('#btn').trigger('customEvent');
});
```

В этом коде элементу с ID `btn` назначен обработчик пользовательского события `customEvent`. Затем элементу с ID `anotherBtn` назначен другой обработчик, который при наступлении события `click` активизирует событие `customEvent`. Поскольку событие `customEvent` назначено элементу с ID `btn`, то будет выведено соответствующее сообщение.

Обратите внимание: jQuery не создает автоматически сокращенную функцию с именем, соответствующим имени пользовательского события, так что этот код приведет к ошибке:

```
$('#btn').customEvent();
```

Кроме пользовательских событий, jQuery позволяет создавать события в пространствах имен. Мы уже упоминали о такой возможности в предыдущих разделах, но не углублялись в детали. Пора исправить данное упущение.

6.2.7. События в пространствах имен

Еще одно изящное дополнение к обработке событий в jQuery — это возможность группировать обработчики по пространствам имен. В отличие от обычного использования таких пространств (когда имя пространства имен является префиксом), в именах событий принадлежность к пространству имен определяется *суффиксом*, который добавляется к имени события и отделяется от него точкой. При желании

можно применять несколько суффиксов и таким образом поместить событие в несколько пространств имен — как было показано в описании метода `on()`. Группируя события таким образом, впоследствии можно легко обращаться к целым группам событий.

Рассмотрим в качестве примера страницу, которая может отображаться в одном из двух режимов: просмотра или редактирования. В режиме редактирования на странице есть множество слушателей событий, назначенных различным элементам. Но в режиме просмотра эти слушатели не нужны, и при переходе из режима редактирования их нужно удалить. В таком случае для режима редактирования можно создать пространство имен:

```
$('.my-class').on('click.editMode', myFunction);
```

Если объединить все обработчики в пространство имен `editMode`, то впоследствии к ним можно будет обращаться ко всем сразу. Например, можно удалить все обработчики, принадлежащие пространству имен `editMode` для всех элементов страницы:

```
$('*').off('.editMode');
```

Как уже говорилось, jQuery также позволяет использовать несколько пространств имен для одного и того же элемента. Это свойство продемонстрировано в следующем примере:

```
$('.elements').on('click.myApp.myName', myFunction);
```

Пространства имен чувствительны к регистру символов, так что если после предыдущего выражения написать:

```
$('.elements').trigger('click.myapp');
```

то обработчик, назначенный событию, не будет выполнен (потому что буква `a` набрана в нижнем регистре).

Обратим внимание на еще один важный момент. Предположим, у нас есть такой код:

```
$('.elements').on('click.myApp.myName', myFunction);
$('.other-elements').on('click.myApp', myOtherFunction);
```

Мы хотим запустить все обработчики, назначенные событию `click` и принадлежащие пространству имен `myApp`. Это значит, мы хотим, чтобы выполнились обе функции — и `myFunction()`, и `myOtherFunction()`. Для этого не требуется писать два выражения наподобие следующих:

```
$('.elements').trigger('click.myApp');
$('.other-elements').trigger('click.myApp.myName');
```

Можно выбрать все элементы обоих множеств и активизировать событие `click`, указав только пространство имен `myApp`:

```
$('.elements, .other-elements').trigger('click.myApp');
```

Возможно, чтобы лучше понять сущность пространства имен, вам поможет такая аналогия: цепочка пространств имен подобна логическому оператору OR. Если активизировано событие, относящееся к некоторым пространствам имен, то запускаются все обработчики, принадлежащие хотя бы к одному из указанных пространств.

Теперь, когда вы вооружены всеми этими знаниями о разных способах обработки событий, пора применить их на практике. В следующей главе мы рассмотрим пример страницы с использованием описанных и некоторых других технологий jQuery, уже известных вам из предыдущих глав.

6.3. Резюме

Опираясь на уже полученные вами знания о jQuery, в этой главе мы познакомили вас с обработкой событий на веб-страницах. Вы узнали, что данный процесс сопряжен с рядом сложностей, но необходим для создания интерактивных веб-приложений. Не последним в списке этих сложностей является наличие в современных браузерах трех разных моделей обработки событий.

Наследие прошлых технологий, базовая модель событий, получившая также неофициальное название модели событий DOM уровня 0, описывает некие браузерно-независимые операции по объявлению слушателей событий, но при реализации функций-слушателей приходится писать браузерно-зависимые ветви кода, чтобы учесть различия в реализации экземпляра `Event`. Эта модель хоть и проста, но ограничивает разработчика только одним слушателем для каждого типа события, которое возникает в элементе DOM.

Расширенная и более стандартизированная модель событий DOM уровня 2 решает эту проблему, так как позволяет привязывать слушатели к типам событий и к элементам. Недостаток модели заключается в том, что она поддерживается только браузерами, совместимыми со стандартом W3C, такими как Chrome, Firefox, Internet Explorer версии 9 и выше, Safari и Opera.

В Internet Explorer 8 и более ранних версиях используется API, основанный на собственной модели событий, в которой доступна лишь часть функционала модели событий DOM уровня 2.

Писать в каждом обработчике по две ветви кода — отдельно для стандартных браузеров и для ранних версий Internet Explorer — весьма и весьма хлопотно и чревато различными осложнениями. К счастью, на помощь приходит jQuery. Библиотека предоставляет единый метод `on()` для назначения обработчиков событий любого типа, для любого элемента, а также набор удобных методов для назначения обработчиков отдельным событиям, таких как `change()` или `click()`. Использование этих методов не зависит от браузера; они нормализуют передаваемый обработчику экземпляр `Event` так, что в него входят стандартные свойства и методы, наиболее часто применяемые обработчиками событий.

В jQuery также есть метод `off()`, позволяющий удалить обработчик события, и средства для запуска обработчика события под управлением сценария. И это еще не все: библиотека позволяет с помощью метода `on()` заранее назначать обработчики элементам, которые еще не существуют на странице.

Наконец, мы научились создавать пользовательские события и события в пространстве имен, изучили их преимущества и способы применения.

В этой главе мы рассмотрели ряд примеров использования событий на страницах, в том числе исчерпывающий пример, наглядно демонстрирующий многие уже изученные вами понятия. В следующей главе мы обсудим, как свести воедино все, что вы уже знаете, и создать на основе jQuery полноценное приложение.

7 Демоверсия: локатор DVD

В этой главе:

- селекторы jQuery;
- прохождение по DOM и манипуляции с ним;
- присоединение обработчиков событий к элементам DOM;
- делегирование событий;
- применение пользовательских событий.

Мы еще не дошли даже до половины книги, но, надеюсь, вы изучили много новых тем, методов и техник. В предыдущих главах были представлены селекторы jQuery, методы обхода и манипуляции с DOM, а также обработки событий. Для каждой из тем мы показали несколько примеров. Они позволили вам сосредоточиться на каком-то аспекте и помогли понять концепцию, но этого было недостаточно.

В данной главе мы рассмотрим все ранее упомянутые темы, а также попытаемся восполнить пробел и продемонстрировать, что вы можете сделать с уже приобретенными знаниями. В нескольких следующих разделах вы разработаете простенькое, но полностью функциональное приложение для управления коллекцией DVD. Посмотрим, как это сделать!

7.1. Размещение событий (и не только) для работы

Представим, что вы видеолюбитель и ваша коллекция DVD достигла тысячи экземпляров. Найти нужный за разумное время стало непросто. И где хранить все эти диски в коробках? Они занимают слишком много места.

Предположим, вы уже решили вопрос хранения путем покупки органайзеров для DVD, которые содержат 100 дисков и каждый из которых занимает существенно меньший объем, чем коробка. Но, несмотря на это, организация DVD попрежнему остается проблемой. Как вы найдете нужный диск без необходимости перебирать их вручную?

Вы не можете сделать нечто вроде сортировки DVD в алфавитном порядке, чтобы помочь быстро найти конкретный диск. Это означало бы следующее: каждый раз при покупке нового DVD вам необходимо переносить все диски в, возможно, десятки органайзеров, чтобы сохранить коллекцию отсортированной. Представьте, сколько работы предстоит, и придете в ужас!

Однако у вас есть компьютер, ноу-хау для написания веб-приложений, а также jQuery! Вы решите проблему хранения и быстрого нахождения дисков, написав программу базы данных DVD. Код для этого примера можно найти в файле `chapter-7/dvds.html`, предоставленном с книгой.

ПРИМЕЧАНИЕ

Эта демонстрационная версия существенно изменилась по сравнению с той, которая была представлена во втором издании данной книги. Если вы купили его (спасибо!), то мы рекомендуем вам не пропускать эту главу. Вы найдете много новых вещей, и в этот раз фильтры действительно работают (в старой версии у вас всегда были те же статические результаты)!

Проект, который вы собираетесь сделать, использует несколько вызовов Ajax для выполнения ряда задач. Мы еще не затрагивали эту тему, но уверяем, что это не помешает увидеть все другие концепции в действии. Из-за ограничений безопасности в некоторых браузерах демо может вызвать ошибку. Мы уже давали советы по данной проблеме в главе 3, но для вашего удобства повторим их еще раз.

ПРИМЕЧАНИЕ

Из-за ограничений безопасности некоторых браузеров вы можете потерпеть неудачу при воспроизведении этой демоверсии. Чтобы избежать такой проблемы, можно выполнить страницу в веб-сервере, таком как Apache, Tomcat или IIS, или поискать конкретное решение для вашего браузера. Например, в WebKit на основе браузеров вы можете запустить его через интерфейс командной строки (CLI), используя флажок --allow-file-access-from-files. Важно, что команда создает новый процесс, так что должна открыться не новая вкладка, а новое окно.

Приступим к работе!

7.1.1. Фильтрация наборов данных больших объемов

Наша программа базы данных для DVD сталкивается с той же проблемой, что и многие другие приложения: как обеспечить пользователям (в данном случае самому себе) быстрый поиск нужной информации?

Вы могли бы просто отобразить упорядоченный список всех названий, но его пролистывание все равно будет неудобным решением, если записей достаточно много. Кроме того, вы хотите узнать, как это сделать правильно, чтобы применить то, чему вы уже научились, к реальным, ориентированным на клиента приложениям. Поэтому никаких сокращений!

Очевидно, что разработка сложного приложения выходит далеко за рамки этой главы. Таким образом, мы сконцентрируемся на разработке панели управления, позволяющей определить фильтры, с помощью которых можно настроить список названий, возвращаемых при выполнении поиска в базе данных.

Разумеется, вам захочется добавить возможность фильтрации по названию DVD. Но также можете включить возможность поиска по году выпуска фильма, органайзеру, в который вы поместили диск, и даже тому, смотрели вы фильм или еще нет. (Это поможет ответить на часто задаваемый вопрос: «Что бы посмотреть сегодня вечером?»)

Вашей первой реакцией могут быть слова: «Ну и что здесь такого?» В конце концов, можно сделать несколько полей — и дело с концом, верно? Однако не торопитесь.

Одно поле для чего-то вроде названия — это хорошо, если, например, вы хотите найти все фильмы со словом «создание» в их названии. Но что, если вы хотите найти данное слово только в фильме, выпущенном в период с 1987 по 1999 год?

Чтобы обеспечить надежный интерфейс для определения фильтров, нужно указать несколько фильтров для различных свойств DVD. Но в этом проекте вы не позволите указать один и тот же фильтр несколько раз. К тому же вместо попыток угадать, сколько понадобится фильтров, вы будете создавать их по первому требованию.

Каждый фильтр идентифицируется с помощью раскрывающегося списка (выбор одного элемента `select`), который определяет поле для фильтрации. На основе типа этого поля (строка, дата, число, даже булево значение) соответствующие элементы управления отображаются в строке для сбора информации о фильтре. Пользователям дается возможность добавлять столько фильтров, сколько им понадобится, но опять-таки по одному одного типа или удалять ранее указанные фильтры.

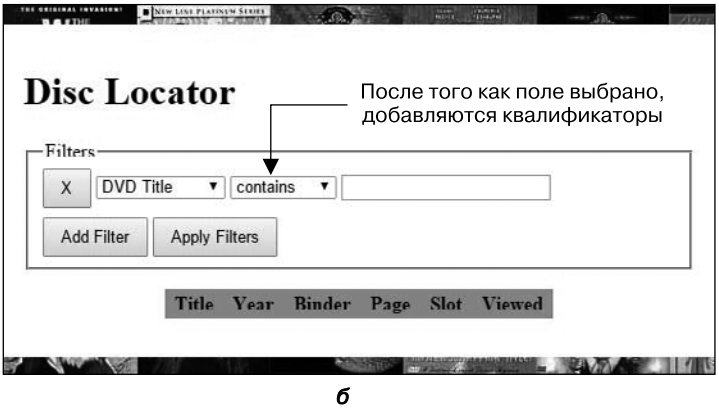
Лучше один раз увидеть, чем 100 раз услышать, поэтому изучите последовательные изображения на рис. 7.1, *а–в*. Они показывают панель фильтра, которую вы сделаете при первоначальном отображении *(а)*, после указания фильтра *(б)* и после указания нескольких фильтров *(в)*.

При просмотре взаимодействий, показанных на рисунках, видно, что многие элементы должны будут создаваться динамически. Обсудим, как это сделать.

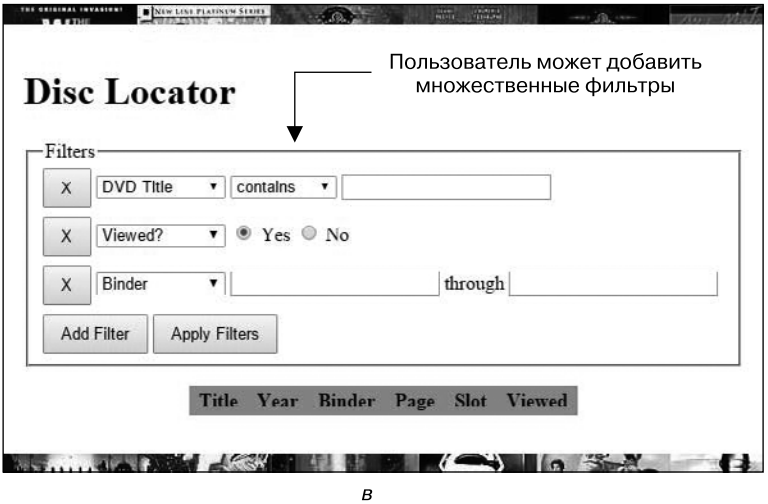
7.1.2. Создание элементов с помощью репликации шаблона

Как вы можете легко заметить, для реализации панели управления фильтрацией необходимо создать достаточное количество элементов в ответ на различные события. Например, нужно ввести новый элемент фильтра, когда пользователь нажимает кнопку Add Filter (Добавить фильтр), и новые элементы управления, предусмотренные для этого фильтра, когда выбрано определенное поле.

а



б



в

Рис. 7.1. Панель фильтра: *а* — сначала показывается отдельный ненастроенный фильтр; *б* — после выбора типа фильтра добавляются соответствующие ему элементы управления; *в* — пользователи могут добавить несколько фильтров

Нет проблем! Как вы уже узнали, jQuery позволяет динамически создавать элементы с помощью функции `$()`. Некоторые из них вы будете добавлять в этом примере и также узнаете о ряде альтернативных вариантов более высокого уровня.

При динамическом введении множества элементов весь код, необходимый для их создания и объединения, может получиться немножко громоздким, и его трудно поддерживать даже с помощью jQuery (а без нее он может стать в разы сложнее!). Было бы здорово, если бы можно было создать «план» комплексной разметки с использованием HTML, а затем повторять его всякий раз, когда нужен экземпляр «плана».

Не переживайте! Метод jQuery `clone()` дает вам такую возможность.

Порядок выполнения работы будет заключаться в создании наборов шаблонов разметки, представляющих собой HTML-фрагменты, которые вы хотите копировать, и использовании метода `clone()` всякий раз, когда нужно создать экземпляр данного шаблона. Вы не хотите, чтобы шаблоны были видны конечному пользователю, поэтому обернете их в элемент `div`, скрытый из поля зрения через CSS.

В качестве примера рассмотрим сочетание кнопки `X` и раскрывающегося списка, идентифицирующее фильтруемые поля. Вам нужно создать экземпляр этого сочетания, когда пользователь нажимает кнопку Add Filter (Добавить фильтр). Код jQuery для создания такой кнопки и элемента `select`, наряду с его дочерним элементом `option`, можно посчитать немного длинным, хотя написать или поддерживать его не так уж сложно. Но легко представить, как нечто более сложное быстро может стать громоздким.

Используя нашу технику шаблонов и размещая шаблон разметки для этой кнопки и для раскрывающегося списка в родительском `<div>`, скрывающем все шаблоны, создайте разметку:

Содержит и прячет все шаблоны ❶

Определяет фильтр, выбирающий шаблон ❷

```
<div class="templates">
  <div class="template filter-chooser">
    <input type="button" class="filter-remover" value="X" />
    <select name="filter" class="filter-type">
      <option value="" data-template-type="" selected="selected">
        Выберите фильтр
      </option>
      <option value="title" data-template-type="template-title">
        Название DVD
      </option>
      <option value="binder" data-template-type="template-binder">
        Биндер
      </option>
      <option value="year" data-template-type="template-year">
        Дата выпуска
      </option>
      <option value="viewed" data-template-type="template-viewed">
        Просмотрено?
      </option>
    </select>
  </div>
  <!-- больше шаблонов здесь -->
</div>
```

Внешний тег `<div>` с классами `template` служит в качестве контейнера для всех ваших шаблонов и будет объявлен в CSS как `display: none;` для предотвращения отображения ❶. В этом контейнере вы определяете другой `<div>`, для которого задаете классы `template` и `filter-chooser` ❷. Вы будете использовать класс `template` для идентификации (отдельных) шаблонов в целом и класс `filter-chooser` для идентификации данного конкретного типа шаблона. Вскоре вы увидите, как эти классы применяются в качестве ловушек JavaScript.

Обратите внимание: каждый элемент `<option>` в `<select>` получил пользовательский атрибут `data-template-type`. Вы будете обращаться к этому значению, чтобы определить, какой тип элемента управления фильтра должен использоваться для выбранного поля фильтра.

На основании того, какой тип фильтра идентифицирован, предстоит заполнять остальную часть строки ввода фильтра элементами управления, которые соответствуют данному типу фильтра. Например, если тип шаблона `template-title`, то вы захотите отобразить текстовое поле, в которое пользователь сможет ввести название (или его часть) для поиска, и раскрывающийся список, давая ему варианты применения этого термина («содержит», «в точности» и т. д.).

Установите шаблон для данного набора элементов управления таким образом:

```
<div class="template template-title">
   <select name="title-condition">
      <option value="contains">содержит</option>
      <option value="starts-with">начинается с</option>
      <option value="ends-with">заканчивается</option>
      <option value="equal">в точности</option>
   </select>
   <input type="text" name="title" />
</div>
```

Снова используете класс `template` для идентификации элемента как шаблона и отмечаете конкретный шаблон с помощью класса `template-title`. Мы намеренно сделали это таким образом, чтобы класс соответствовал значению `data-template-type` для поля выбора раскрывающегося списка.

Репликация этих шаблонов очень проста благодаря имеющимся знаниям о возможностях jQuery. Предположим, вы хотите добавить экземпляр шаблона в конец элемента, на который у вас есть ссылка в переменной `whatever`. Можно написать:

```
$('div.template.template-title')
   .clone()
   .appendTo(whatever);
```

В этой инструкции вы выбираете контейнер шаблона для репликации (в данном случае один, имеющий класс `template-title`), используя те удобные классы, которые размещены в разметке шаблона. Затем вы клонируете элемент с помощью метода `clone()` и наконец прикрепляете шаблон в конец содержимого элемента, идентифицируемого `whatever`. Теперь понятно, почему мы постоянно подчеркиваем эффективность цепочек методов jQuery?

Проверяя параметры раскрывающегося списка `filter-chooser`, вы видите, что есть ряд других типов шаблонов, определенных как `template-binder`, `template-year` и `template-viewed`. Определите шаблоны элементов управления для этих типов фильтров с помощью следующего кода:

```
                                Шаблон фильтра биндера
<div class="template template-binder">
   <input type="text" name="binder-min" class="numeric" />
   <span>до</span>
   <input type="text" name="binder-max" class="numeric" />
</div>
                                Шаблон фильтра года
<div class="template template-year">
   <input type="text" name="year-min" class="date" />
   <span>до</span>
   <input type="text" name="year-max" class="date" />
</div>
                    Шаблон фильтра просмотренного
<div class="template template-viewed">
   <label><input type="radio" name="viewed"
                 value="true" checked="checked" />
   Да</label>
   <label><input type="radio" name="viewed"
                 value="false" />Нет</label>
</div>
```

Теперь, когда ваша стратегия репликации определена, рассмотрим первичную разметку.

7.1.3. Настройка основной разметки

Если вернуться к рис. 7.1, *а*, то можно заметить, что начальное отображение страницы поиска DVD незамысловатое: заголовок, экземпляр первого фильтра, несколько кнопок и предустановленная таблица для отображения результатов. Посмотрите на следующую HTML-разметку:

```
<h1>Локатор дисков</h1>

<form id="form-filters" action="#">
   <fieldset>
      <legend>Фильтры</legend>
      <div id="filters"></div>          ❶ Контейнер для экземпляров фильтров
      <div class="buttons-wrapper">
         <input type="button" id="filter-add"
                value="Добавить фильтр" />
         <input type="submit" id="filter-apply"
                value="Применить фильтры"/>
      </div>
```

```
   </fieldset>
</form>

<div id="panel-results">                    ❷ Контейнер для результатов поиска
   <table id="results">
      <tr>
         <th>Название</th>
         <th>Год</th>
         <th>Биндер</th>
         <th>Страница</th>
         <th>Слот</th>
         <th>Просмотрено</th>
      </tr>
   </table>
</div>
```

Нет ничего слишком странного в этой разметке — или все же есть? Где, например, разметка для первоначального фильтра в раскрывающемся списке? Вы создали контейнер, в котором будут размещены фильтры ❶, но изначально он пустой. Почему?

Разумеется, может понадобиться возможность заполнять новые фильтры (вскоре речь зайдет и о них) динамически — так зачем же мы работаем на два фронта? Как вы скоро увидите, можно будет использовать динамический код, чтобы сначала заполнить первый фильтр, поэтому не нужно явно создавать его в статической разметке. Еще один момент, который следует отметить: вы отложили таблицу для получения результатов ❷.

У вас есть простой, основной HTML и несколько скрытых шаблонов, пригодные для быстрого создания новых элементов с помощью репликации. В итоге можно начать писать код, который будет применяться к вашей странице!

7.1.4. Добавление новых фильтров

После нажатия кнопки Add Filter (Добавить фильтр) вам необходимо добавить новый фильтр в `<div>`. Если вы помните, как легко можно установить обработчики событий, используя jQuery, то несложно будет добавить и обработчик нажатия кнопки Add Filter (Добавить фильтр). Но есть еще кое-что, стоящее внимания!

Вы уже видели, как можно реплицировать элементы управления формой, когда пользователь добавляет фильтры, и владеете хорошей стратегией для простого создания нескольких экземпляров этих элементов управления. Но в конце концов вам будет нужно отправить значения серверу, чтобы он смог найти отфильтрованные результаты в базе данных. Написание серверной части находится уже за пределами данной главы, но это не значит, что ваше приложение не будет работать. Вы будете имитировать базу данных с помощью файла `movies.json`, который содержит массив JSON. Файл можно найти в папке `chapter-7`, предоставленной с книгой. Каждый элемент массива — объект со следующими свойствами: `title`, `year`, `binder`, `page`, `slot` и `viewed`.

Чтобы не загружать объект JSON каждый раз, когда вы взаимодействуете с приложением, загрузите его один раз при открытии страницы и сохраните его в глобальной переменной `movies`. Для выполнения этой задачи вам понадобится использовать вспомогательную функцию jQuery `getJSON()`. Мы ее пока не изучали (обсудим в подразделе 10.3.2), но она открывает доступ к ресурсу (URL или файл), содержащему объект JSON (разрешается массив или любой другой тип допустимого формата JSON) и запускает обработчик после извлечения объекта. Данный метод передает полученный JSON, преобразованный в тип JavaScript, обработчику. Внутри последнего ничего не понадобится делать, кроме присвоения этого объекта JavaScript вашей глобальной переменной `movies` и запуска пользовательского события `moviesLoaded`, чтобы информировать приложение о готовности к работе.

Предвкушаю ваши восклицания: «Глобальная переменная? Думаю, вы погорячились». Эти переменные могут быть проблемой при неправильном использовании. В нашем случае это действительно глобальное значение, которое представляет полностраничное понятие, и они никогда не вызовут никаких конфликтов, поскольку все аспекты страницы будут хотеть получить доступ к данному значению последовательно. Тем не менее в идеале приложение должно иметь только одну глобальную переменную для своих задач, подобно тому как и в jQuery можно получить доступ ко всем методам, вспомогательным функциям и свойствам библиотеки через свойство `jQuery`. Вместо того чтобы присваивать данные глобальной переменной `movies`, можно создать одну глобальную переменную, например `dvdApp`, содержащую данные фильма и методы приложения.

В этой демоверсии вы проигнорируете риски, связанные с использованием глобальной переменной, чтобы сделать все как можно проще. Окончательный код, который реализует то, что мы описали ранее, показан здесь:

```
var movies;
$.getJSON('movies.json', function(data) {
  movies = data;
  $(document).trigger('moviesLoaded');
});
$(document).on('moviesLoaded', function() {
    // Бизнес-логика здесь
});
```

Внутри обработчика для вашего пользовательского события `moviesLoaded` установите другой обработчик, который будет выполняться при нажатии кнопки Add Filter (Добавить фильтр). Перед написанием необходимого кода нужно добавить две следующие инструкции в начале вашего кода:

```
var $filters = $('#filters');
var templatesAvailable = $('.template', '.templates')
  .not('.filter-chooser')
  .length;
```

Первая нужна для того, чтобы несколько раз использовать элемент, содержащий `filters` в качестве его ID. Вторая — чтобы убедиться, что не все шаблоны, определенные на странице, уже применяются. После того как вы это сделали, можно написать тело функции обратного вызова:


```
$('#filter-add')
       .click(function() {
          if ($filters.find('.template:last
                            .filter-type').val() === '') {
              return;
          }

          var filterInUse = $filters
                  .children()
                  .map(function() {
                     return $(this)
                              .children('.template')
                              .attr('class')
                              .match(/\b(template-.+?)\b/g)[0];
                  })
                  .get();

          if (filterInUse.length === templatesAvailable) {
             return;
          }

          var $filterChooser = $('div.template.filter-chooser')
                  .clone()
                  .removeClass('filter-chooser')
                  .addClass('filter');
            $filterChooser
                  .find('option[data-template-type]')
                  .filter(function() {
                     return filterInUse
                      .indexOf($(this)
                      .data('template-type')) >= 0;
                  })
                  .remove();
          $filterChooser.appendTo($filters);
       })
       .click();
```

Этот фрагмент может показаться сложным на первый взгляд, но он очень эффективен, хотя и не требует написания большого количества кода. Разобъем его на отдельные шаги.

Первым делом вы устанавливаете обработчик нажатия кнопки Add Filter (Добавить фильтр) 1 с помощью метода jQuery `click()`. Это нужно сделать в переданной

методу функции, которая будет вызываться при нажатии кнопки, — тогда и будет происходить все самое интересное.

Перед выполнением каких-либо действий нужно проверить, была ли нажата кнопка Add Filter (Добавить фильтр) прежде, чем пользователь выбрал фильтр, в этом случае вам необходимо завершить функцию преждевременно ❷. Если фильтр был выбран в связи с нажатием данной кнопки, то добавьте его, и нужно создать новый контейнер для размещения в нем фильтра. Как мы уже говорили, нельзя задействовать несколько фильтров одного и того же типа, поэтому вы должны получить все типы применяемых фильтров ❸, чтобы исключить их. Для их получения вы полагаетесь на имя класса, который использовался для каждого шаблона (`template-title`, `template-year` и т. д.), и два метода, описанные в предыдущих главах: `map()` и `get()`. По окончании проверьте, все ли доступные фильтры в работе; если да, то преждевременно остановите обработчик ❹. Если нет, то продолжайте.

Можно клонировать шаблон, используя метод jQuery `clone()`, который вы установили содержащим раскрывающийся список фильтра, с применением подхода к репликации, описанного в предыдущем разделе. Затем дайте ему класс `filter` не только для стиля CSS, но также и для того, чтобы можно было позже расположить эти элементы в коде ❺. После создания элемента следует исключить уже применяемые фильтры с помощью метода `filter()` ❻. Затем добавьте клонированный шаблон к главному контейнеру фильтра, созданному с ID-значением `filters` ❼.

Предыдущая операция была последней определенной внутри обработчика. После присоединения последнего, но перед закрытием инструкции вызовите событие щелчка на том же элементе с помощью метода `click()` ❽. Мы обсуждали это в подразделе 6.2.5. Это хорошо известная и часто используемая техника для выполнения только что присоединенного обработчика.

Зачем так делать? Помните, когда вы догадались, где находится разметка для первончального раскрывающегося списка фильтра? Да, поняли правильно! Вы запускаете обработчик, чтобы после загрузки страницы первый элемент `select` добавился к панели и позволил вам выполнить первый выбор.

Загрузите эту страницу в ваш браузер и проверьте действие кнопки Add Filter (Добавить фильтр). Обратите внимание: каждый раз, когда вы нажимаете данную кнопку, новый фильтр добавляется на страницу. Если вы проверите DOM с помощью отладчика JavaScript (Firebug в Firefox и Chrome Developer Tool отлично подходят для этого), то увидите, как шаблон скопирован в контейнер.

В одной функции (обработчике) вы использовали недавно полученные знания. Еще раз было показано, как jQuery позволяет выполнять сложные операции в нескольких строчках кода.

Но ваша работа еще не закончена. В раскрывающихся меню не указано, какое поле должно быть отфильтровано. Когда пользователь делает выбор, вам необходимо заполнить контейнер фильтра, добавив соответствующие элементы управления для этого типа фильтра.

7.1.5. Добавление шаблонов для элементов управления

При выборе из фильтра в раскрывающемся списке вам необходимо заполнять фильтр элементами управления, подходящими этому фильтру. Вы уже облегчили себе задачу, создав разметку шаблонов, чтобы они копировались при определении того, какой из них для этого подходит. Но также есть несколько других задач, которые нужно решить при изменении значения из раскрывающегося списка.

Взгляните на то, что вы будете делать при создании обработчика изменений для раскрывающегося списка. Помните: последующий код, как и в предыдущем разделе, написан внутри обработчика пользовательского события, называющегося `moviesLoaded`.

```
$('#filters').on('change', '.filter-type', function() {
   var $this = $(this);
   var $filter = $this.closest('.filter');
   var filterType = $this.find(':selected').data('template-type');

   $('.qualifier', $filter).remove();
   $('div.template.' + filterType)
           .clone()
           .addClass('qualifier')
           .appendTo($filter);
   $this.find('option[value=""]').remove();
})
```

❶ Устанавливает обработчик изменений

❷ Удаляет все старые элементы управления

❸ Копирует подходящий шаблон

❹ Удаляет настройку «Выберите фильтр»

Вы пользуетесь методом jQuery `on()` для установки обработчика, который будет автоматически устанавливаться в соответствующих местах без дальнейших действий с вашей стороны. На сей раз вы заранее установите обработчик изменения, используя делегирование событий, для любого фильтра в выпадающем меню, который будет запускаться ❶. Это необходимо потому, что во время присоединения обработчика нет фильтров внутри `div`, имеющего ID `filters`.

Когда запустится обработчик изменений, кэшируйте объект jQuery, содержащий текущий элемент (`this`), так как вы будете использовать его несколько раз. Кроме того, имейте в виду: родительский контейнер фильтра и тип фильтра записываются в пользовательском атрибуте `data-template-type`.

Если у вас есть эти значения, то нужно удалить какие-либо элементы управления фильтров, которые могут быть уже в контейнере ❷. В конце концов, пользователь может много раз изменять значения выбранного поля, а вам не хочется продолжать добавлять все больше и больше элементов управления в процессе этого! Добавьте класс `qualifier` для всех соответствующих элементов по мере их создания (в следующем операторе), чтобы их было легко выбрать и удалить.

После того как вы будете уверены, что все в порядке, скопируйте шаблон для правильной установки спецификаторов ❸, используя значение, полученное от атрибута `data-template-type`. Имя класса `qualifier` добавляется к каждому элементу, созданному для легкого выбора (как видно в предыдущей инструкции), и элемент добавляется в контейнер родительского фильтра.

И наконец, удалите пункт Choose a filter (Выберите фильтр) из раскрывающегося списка фильтра ❹: когда пользователь выбрал конкретное поле, нет никакого смысла выбирать этот вариант еще раз. Вы *можете* просто проигнорировать событие изменения, которое возникает, когда пользователь выбирает этот пункт, но лучший способ запретить пользователю делать то, что не имеет смысла, — в первую очередь не позволять ему сделать это!

Снова обратитесь к странице примера в вашем браузере. Попробуйте добавить несколько фильтров и изменить их выбор. Обратите внимание, что спецификаторы всегда совпадают с выбором поля.

Теперь удалите некоторые кнопки...

7.1.6. Удаление ненужных фильтров и другие задачи

Вы дали пользователю возможность изменить поле, к которому будет применен любой фильтр, но также предоставили ему кнопку удаления (с отметкой `X`), которую они могут использовать для полного удаления фильтра.

К этому времени вы, скорее всего, поняли, что задача будет вполне тривиальной с инструментами, которые находятся в вашем распоряжении. Когда кнопку нажали, все, что нужно сделать, — найти контейнер ближайшего родительского фильтра и убрать его! Обратите внимание: в исходном коде обработчик присоединен с помощью добавления цепочки кода к предыдущему сегменту. Для ясности мы еще раз повторим выбор в следующем коде:

```
$('#filters').on('click', '.filter-remover', function() {
  $(this).closest('.filter').remove();
});
```

Здесь опять используется делегирование событий, потому что во время загрузки страницы внутри главной панели нет элементов с классом `filter-remover`.

Теперь, когда все обработчики для фильтров установлены, осталось сделать только одно: применить фильтры и показать результат.

7.1.7. Отображение результатов

В подразделе 7.1.4 говорилось, что написание серверной части приложения выходит за рамки настоящей главы, но мы не хотим оставить вас без рабочего приложения. Как уже отмечалось, вы будете эмулировать базу данных, используя массив JSON, сохраненный в файле `movies.json`, в комплекте с приложением. В данном

подразделе вам станет понятна логика функции, которую необходимо прикрепить в качестве обработчика для события `submit` формы. Посмотрим, что представляет собой этот обработчик:


```
$('#form-filters').submit(function(event) {
  event.preventDefault();

  var titleCondition =
      $filters.find('select[name="title-condition"]').val();
  var title = $filters.find('input[name="title"]').val();
  var binderMin =
      parseInt($filters.find('input[name="binder-min"]').val(), 10);
  var binderMax =
      parseInt($filters.find('input[name="binder-max"]').val(), 10);
  var yearMin =
      parseInt($filters.find('input[name="year-min"]').val(), 10);
  var yearMax =
      parseInt($filters.find('input[name="year-max"]').val(), 10);
  var viewed = $filters.find('input[name="viewed"]:checked').val();

  $('tr:has(td)', '#results').remove();
  results = $.grep(movies, function(element, index) {
      return (
        (
          (
            titleCondition === undefined &&
            title === undefined
          ) ||
          (
            titleCondition === 'contains' &&
            element.title.indexOf(title) >= 0
          ) ||
          (
            titleCondition === 'stars-with' &&
            element.title.indexOf(title) === 0
          ) ||
          (
            titleCondition === 'ends-with' &&
            element.title.indexOf(title) ===
            element.title.length - title.length
          ) ||
          (
            titleCondition === 'equals' &&
            element.title === title
          )
        ) &&
        (isNaN(binderMin) || element.binder >= binderMin) &&
```

```
        (isNaN(binderMax) || element.binder <= binderMax) &&
        (isNaN(yearMin) || element.year >= yearMin) &&
        (isNaN(yearMax) || element.year <= yearMax) &&
        (viewed === undefined || element.viewed ===
        (viewed === 'true'))
      );
  });

  var row;
  for(var i = 0; i < results.length; i++) {
    row = '<td>' + results[i].title + '</td>';
    row += '<td>' + results[i].year + '</td>';
    row += '<td>' + results[i].binder + '</td>';
    row += '<td>' + results[i].page + '</td>';
    row += '<td>' + results[i].slot + '</td>';
    row += '<td>' + (results[i].viewed ? 'X' : '') + '</td>';
      $('#results').append(
        $('<tr>').html(row)
      );
    }
});
```

❻ Итерирует по отфильтрованным фильмам

❼ Форматирует свойства фильма как ячейки строки таблицы

❽ Добавляет ячейки в строке, созданной на лету, и затем добавляет строку в таблицу результатов

Сначала нужно прикрепить функцию в качестве обработчика для события `submit` формы ❶. Поскольку вы не хотите, чтобы ваш браузер выполнял HTTP-запрос (помните, у вас нет серверной части) внутри обработчика, вы предотвращаете поведение по умолчанию ❷. Затем выполняете набор заданий для получения значений фильтров, заполненных пользователем ❸. Даже если пользователь не будет использовать их в полном объеме, то попытайтесь получить все возможные значения. Для тех из них, которые не были добавлены, значение будет `undefined`, с этим вы будете иметь дело позже.

В следующей инструкции очистите таблицу от всех предыдущих результатов, оставив заголовки на месте ❹. После чего, используя `$.grep()`, создайте новый массив, содержащий только фильмы, которые соответствуют заполненным фильтрам ❺. Мы еще не рассматривали этот метод (обсудим его в подразделе 9.3.3), но он аналогичен методу `filter()`.

Сразу после получения результатов запроса вы должны их показать. Для этого итерируйте по отфильтрованным фильмам ❻. Внутри цикла отформатируйте свойства каждого фильма в виде ячеек строки таблицы ❼. Фактическая строка элемента (`tr`) вводится динамически, с помощью возможностей метода `jQuery()` создавать элементы на основе строки. Затем добавьте строку в таблицу ❽.

Теперь, когда вы проанализировали обработчик, нажмите кнопку Apply Filters (Применить фильтры). Получилось! Набор фильмов, которые соответствуют фильтрам, отображен на экране. Пример результатов этого проекта показан на рис. 7.2.

Сделав последний шаг, вы завершили задуманное — во всяком случае, так, как намеревались это сделать в рамках данной главы. Но, как известно...

Рис. 7.2. Пример результатов, возвращаемых локатором DVD

7.1.8. Всегда есть куда расти

В следующем списке описаны дополнительные функциональные возможности, которые также понадобятся вашей форме, чтобы она считалась максимально эффективной, или же просто было бы хорошо, если бы они были. Можете ли вы реализовать эти дополнительные функции, применив уже имеющиеся знания?

- Проверка данных в вашей форме оставляет желать лучшего. Например, внутри обработчика формы кнопки Submit (Отправить) вы преобразуете строки в числа там, где это имеет смысл, но вам могли бы понадобиться лучшие элементы управления, особенно для полей даты.

  В идеале, используя серверную часть, можно передать обработку кода серверу — в конце концов, он в любом случае должен проверять данные. Но это делает пользовательский опыт не столь приятным, и, как мы уже отмечали, лучшим способом борьбы с ошибками является предотвращение их возникновения на начальном этапе.

  Поскольку решение включает в себя инспектирование экземпляра `Event`, который не был включен в пример до сего момента, мы предоставим вам код для запрета ввода любых символов, кроме цифр в числовых полях. Работа кода должна быть понятной для вас, учитывая знания, приобретенные в этой главе, но если возникают сложности, то сейчас как раз самое время вернуться и пересмотреть ключевые моменты:

```
$('input.numeric').on('keypress', function(event) {
  // Символ с кодом 48 - "0". Символ с кодом 57 - "9".
  if (event.which < 48 || event.which > 57) return false;
});
```

- Для браузеров, которые поддерживают HTML5, есть более простое решение. Вы можете заставить пользователя применять только цифры с помощью нового типа `<input>`, который называется `number`.
- Поля даты не проверены. Как бы вы позаботились о том, чтобы вводились только допустимые диапазоны даты? Что делать, если пользователь указывает дату начала более позднюю, чем дата окончания? Вы не можете проверять на основе посимвольного ввода, как делали это с числовыми полями.
- Когда подходящие поля добавляются в фильтр, пользователь должен нажать одно из полей, чтобы перевести на него фокус. Не так уж и удобно! Добавьте в пример код, который переводит фокус на новые элементы управления после их добавления.
- Одним из требований было не позволять пользователю задействовать один и тот же фильтр более одного раза. В текущей версии есть способ применять один и тот же фильтр дважды, он также показан в дефекте демо. Можете ли вы выяснить, как это сделать, и исправить такое поведение?
- Ваша форма позволяет пользователю указывать более одного фильтра, но только один для каждого типа. Как бы вы изменили форму, чтобы позволить пользователю указать несколько фильтров того же типа?
- Типы фильтров могут быть обновлены при удалении одного из используемых фильтров. Если вы добавите все четыре типа фильтров (`DVD Title`, `Binder`, `Release Date` и `Viewed?`), а затем удалите первый, то не сможете изменить один из оставшихся фильтров, чтобы снова ввести `DVD Title`. Это происходит потому, что список создается в момент нажатия кнопки Add Filter (Добавить фильтр) и никогда не обновляется. Как вы можете обновить показ для решения этой проблемы? (Подсказка: прослушивайте событие `click` кнопок с классом `filter-remover`.)
- Какие другие улучшения вы бы предложили для того, чтобы сделать код более надежным, а интерфейс — удобным? Как в этом поможет jQuery?

7.2. Резюме

Применяя полученные знания по jQuery, в данной главе вы разработали полностью рабочее веб-приложение для управления коллекцией DVD. Один из уроков, который вы смогли извлечь, состоит в следующем: задачи, на первый взгляд сложные, на деле есть не что иное, как комбинированный набор простых инструкций. Мы надеемся, что вы весело провели время, разрабатывая это небольшое приложение, целью которого было закрепление представленных понятий.

В следующей главе мы рассмотрим, как библиотека использует описанные возможности для внедрения анимации и анимационных эффектов.

8 Наполняем страницы анимацией и эффектами

В этой главе:

- отображение и скрытие элементов без анимации;
- отображение и скрытие элементов с применением базовых анимационных эффектов;
- расширение основных функций;
- создание пользовательских анимационных эффектов;
- управление анимационными эффектами и организация очередей.

В самом начале становления Интернета возможности, доступные авторам страниц, были серьезно ограничены не только минимальными API, но и медлительностью движков сценариев и маломощными системами. Идея задействовать эти ограниченные возможности для внедрения анимации и эффектов была смехотворной, и в течение многих лет анимация осуществлялась только с помощью анимированных изображений в формате GIF (которые, как правило, применялись мало, делая страницы скорее раздражающими, чем удобными в использовании).

Сегодня движки сценариев браузеров молниеносны, работают на оборудовании, которое еще десять лет назад представить было просто невозможно, и предлагают богатый выбор возможностей для нас как авторов страниц. Еще важнее то, что в современных браузерах реализованы несколько модулей CSS3 со стандартными свойствами, позволяющими создавать удивительные анимации и эффекты. Вот ряд примеров: `transition`, `transform`, `filter`, `blur` и `mask`. К сожалению, есть отдельные нюансы, которые нужно учитывать. Во-первых, фактические модули реализованы в зависимости от браузера, так что не все браузеры поддерживают одни и те же модули. Кроме того, браузеры, поддерживающие какой-либо модуль, внедрили его в разное время, и данный промежуток времени может быть значительным. Второй момент: более старые браузеры и несколько мобильных браузеров не поддерживают эти модули, а некоторые из них (самые старые) никогда и не будут. Поэтому, если мы хотим создать анимацию, которая бы работала во всех браузерах, у нас нет другого выбора, кроме как использовать JavaScript.

Но, несмотря даже на то, что он может помочь нам решить эту задачу, не так просто создать анимационный эффект с помощью собственных функций. К счастью, jQuery приходит на помощь, предоставляя тривиально простой интерфейс для создания всевозможных изящных эффектов.

Но прежде, чем мы углубимся в добавление отличных эффектов на наши страницы, нужно рассмотреть вопрос: *стоит ли?* Подобно голливудскому фильму, в котором полно спецэффектов и нет сюжета, злоупотребляющая эффектами страница может вызвать иную реакцию, нежели мы ожидали, и зачастую отрицательную. Помните: эффекты нужно задействовать, чтобы повысить удобство использования страницы, а не просто для красоты. Кроме того, слишком много анимации может замедлить производительность сайта, особенно при доступе с мобильных устройств. Учитывая эти предостережения, посмотрим, что может предложить jQuery.

8.1. Отображение и скрытие элементов

Пожалуй, наиболее распространенным типом динамического эффекта, который вы захотите выполнить на одном или нескольких элементах, является простое действие их отображения или скрытия. Мы вернемся к более необычной анимации чуть позже, но иногда вы предпочтете оставить анимацию простой, чтобы элементы появлялись или мгновенно исчезали!

Методы для отображения и скрытия элементов в объекте jQuery называются так, как вы наверняка и ожидали: `show()` и `hide()` соответственно. Мы отложим представление их формального синтаксиса по причинам, которые станут ясны немного позже; в данный момент без всяких объяснений сосредоточимся на их использовании.

Какими бы простыми ни показались эти методы, вам следует запомнить некоторые вещи. Во-первых, jQuery скрывает элементы, изменяя их свойства `style.display` на `none`. Если элемент в наборе соответствующих элементов уже скрыт, то он останется таковым, но возможность возвратить его для построения цепочки сохранится. Например, предположим, у вас есть следующий HTML-фрагмент:

```
<div style="display: none;">Это будет скрыто</div>
<div>Это будет показано</div>
```

Если вы напишете:

```
$('div').hide().addClass('fun');
```

то получите следующий результат:

```
<div style="display: none;" class="fun"> Это будет скрыто </div>
<div style="display: none;" class="fun"> Это будет показано </div>
```

Обратите внимание: даже если первый элемент уже был скрыт, то он остается частью соответствующего набора и принимает участие в оставшейся части метода цепи. Это подтверждается тем фактом, что оба элемента обладают классом `fun` после выполнения инструкции.

Второе, о чем следует помнить, — jQuery показывает объекты путем изменения свойства `display` из `none` в `block` либо `inline`. Какое из этих значений будет выбрано, зависит от того, было ли установлено ранее указанное явное значение для элемента. Если значение было явным, то оно запоминается и возвращается. В противном случае оно зависит от состояния по умолчанию свойства `display` для типа целевого элемента. Например, у элементов `div` свойство `display` установится в `block`, тогда как то же свойство элемента `span` будет установлено в `inline`.

ПРИМЕЧАНИЕ

До версий 1.11.0 и 2.1.0 jQuery в этом механизме была ошибка. После первого вызова hide() она сохраняла старое значение свойства display во внутреннюю переменную. Затем, когда вызывался метод show(), библиотека восстановила это значение. Таким образом, если после первого вызова hide() свойство display было установлено чему-то другому с помощью выполнения метода show(), то jQuery восстановила старое значение, а не измененное. Найти больше по этому вопросу можно здесь: http://bugs.jquery.com/ticket/14750.

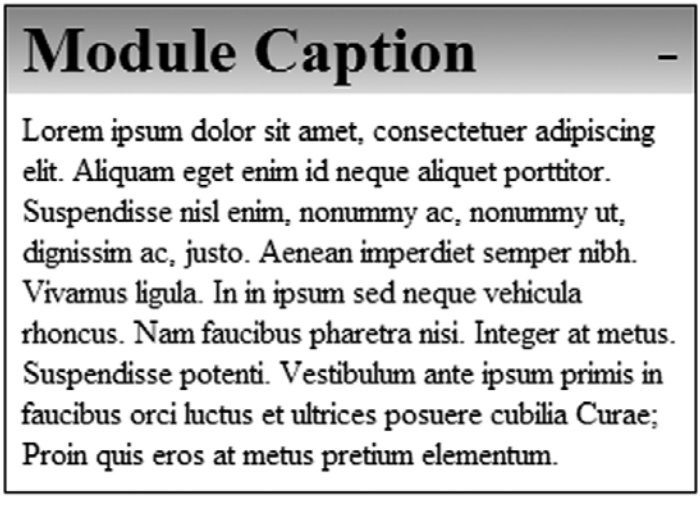
Теперь, когда вы знаете, как ведут себя эти методы, посмотрим, как можно с ними работать.

8.1.1. Реализация сворачиваемого «модуля»

Вы, несомненно, знакомы с сайтами, которые представляют различные части информации в конфигурируемых модулях (иногда называемых *tiles* — «плитки») или каких-то информационных панелях страницы. Такой вид сайта позволяет настроить многое, связанное с представлением на странице, в том числе перемещение модулей, расширение их до полностраничного размера и даже полное их удаление. Многие из них также предоставляют хорошую возможность свернуть модуль в его строку заголовка так, чтобы он занимал меньше места, а не был удален из страницы. Это, видимо, идеальный пример функциональных возможностей, который вы можете повторить, используя знания, полученные в предыдущем разделе. Таким образом, то, что вы собираетесь сделать, — это создание модулей информационной панели, позволяющих пользователям свернуть модуль в его полосе заголовка.

Прежде чем заняться написанием кода, посмотрим, как будет выглядеть модуль в его нормальном и свернутом состоянии. Эти состояния показаны на рис. 8.1, *а* и *б*.

На рис. 8.1, *а* мы показываем модуль с двумя основными разделами: полоса заголовка и тело. Тело содержит данные модуля, в данном случае случайный текст Lorem Ipsum (узнать больше, о чем этот текст, можно здесь: http://en.wikipedia.org/wiki/Lorem_ipsum). Более интересна полоса заголовка, содержащая заголовок для модуля и небольшую кнопку (знак минуса в правом верхнем углу), которая послужит вам инструментом для вызова функциональности разворачивания (и сворачивания).



а

б

Рис. 8.1. Состояния модуля: *а* — вы создадите вашу собственную информационную панель, состоящую из двух частей, полосы с заголовком, кнопки сворачивания (знак минуса в правом верхнем углу) и тела, где отображаются данные; *б* — после нажатия кнопки сворачивания тело модуля исчезает, как будто оно свернуто в полосу заголовка

После нажатия кнопки тело модуля исчезнет, как если бы оно было свернуто в строке заголовка. Последующий щелчок развернет тело, восстанавливая его первоначальный вид.

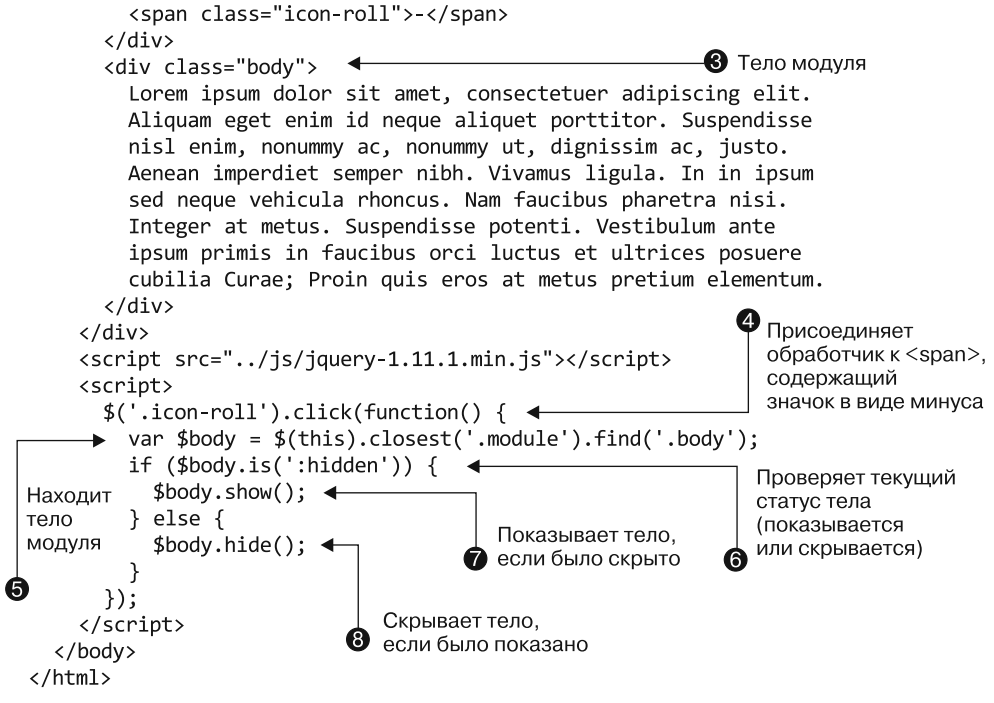
Код для реализации данной функции можно найти в файле `chapter-8/collapsible.module.take.1.html`, предоставленном с книгой, а также он показан в листинге 8.1. Если предположите, что `take.1` указывает на то, что мы будем пересматривать этот пример, то вы правы!

Листинг 8.1. Первая реализация нашего сворачиваемого модуля

```
<!DOCTYPE html>
<html>
  <head>
    <title>Сворачиваемый модуль – Попытка 1</title>
    <link rel="stylesheet" href="../css/main.css" />
    <link rel="stylesheet" href="../css/module.css" />
  </head>
  <body>
    <div class="module">
      <div class="caption clearfix">
        <h1>Заголовок модуля</h1>
```

1 Блок представления модуля

2 Заголовок модуля




```
      <span class="icon-roll">-</span>
    </div>
    <div class="body">
      Lorem ipsum dolor sit amet, consectetuer adipiscing elit.
      Aliquam eget enim id neque aliquet porttitor. Suspendisse
      nisl enim, nonummy ac, nonummy ut, dignissim ac, justo.
      Aenean imperdiet semper nibh. Vivamus ligula. In in ipsum
      sed neque vehicula rhoncus. Nam faucibus pharetra nisi.
      Integer at metus. Suspendisse potenti. Vestibulum ante
      ipsum primis in faucibus orci luctus et ultrices posuere
      cubilia Curae; Proin quis eros at metus pretium elementum.
    </div>
  </div>
  <script src="../js/jquery-1.11.1.min.js"></script>
  <script>
    $('.icon-roll').click(function() {
      var $body = $(this).closest('.module').find('.body');
      if ($body.is(':hidden')) {
        $body.show();
      } else {
        $body.hide();
      }
    });
  </script>
</body>
</html>
```

Разметка, которую вы будете использовать для создания структуры вашего модуля, довольно проста. Мы применили к нему ряд имен классов, служащих как для идентификации, так и для стилей CSS. Вся конструкция заключена в `<div>` с классом `module` ❶, содержащий заголовок ❷ и тело ❸, которые являются `<div>` с классами `caption` и `body` соответственно. Элемент заголовка также содержит класс `clearfix`, применяемый только для стилизации, поэтому мы не будем его обсуждать.

Чтобы задать этому модулю сворачивающееся поведение, разместите внутри заголовка `<span>`, имеющий класс `icon-roll`, содержащий «иконку» (на самом деле просто текст со знаком «минус»). Для данного элемента вы присоедините обработчик для события щелчка ❹.

В обработчике щелчка вы должны сначала найти тело, связанное с модулем. Нужно найти конкретный экземпляр тела модуля: у вас может быть много модулей на странице информационной панели, так что нельзя просто выбрать все элементы, которые содержат класс `body`. Вы быстро обнаружите нужный элемент тела, найдя ближайший контейнер `module`, а затем, начиная с него, ищете потомка, содержащего класс `body` ❺. Если вам неясно, как выражение находит правильный элемент, то пришло время повторить информацию, представленную в первых главах книги, о поиске и выборе элементов.

Когда тело найдено, вы проверяете, скрыто оно или отображается, используя метод jQuery `is()` ❻. Если скрыто, то показываете его с помощью `show()` ❼; в противном случае скрываете его, задействуя `hide()` ❽. Это было несложно, верно? Но, оказывается, можно сделать еще проще!

8.1.2. Отображение состояния элементов

Отображение состояния элементов между выявленными и скрытыми, как вы делали в примере сворачиваемого модуля, используется настолько часто, что jQuery определяет метод `toggle()`, который еще более упрощает данный процесс.

Применим этот метод к сворачиваемому модулю и посмотрим, как он помогает упростить код листинга 8.1. Листинг 8.2 показывает только обработчик щелчка для оптимизированной страницы (никаких других изменений не требуется) с изменениями, выделенными жирным шрифтом. Полный код страницы можно найти в файле `chapter-8/collapsible.module.take.2.html`, предоставленном с книгой.

Листинг 8.2. Код сворачиваемого модуля, упрощенный toggle()

```
$('.icon-roll').click(function() {
  $(this)
          .closest('.module')
          .find('.body')
          .toggle();
});
```

Обратите внимание: вам больше не нужен условный оператор для определения того, что нужно — скрыть или показать тело модуля; `toggle()` сам позаботится о замене отображаемого состояния. Это позволяет немного упростить код и избежать необходимости сохранить ссылку на тело в переменной.

Метод `toggle()` будет полезным не только в качестве краткой альтернативы чередующимся вызовам `show()` и `hide()`. В течение нескольких следующих страниц вы узнаете, как данный метод позволяет делать еще больше.

Моментальное появление и исчезновение элементов очень удобно, но иногда хочется, чтобы переход был менее резким. Посмотрим, как это можно сделать.

8.2. Анимация отображения состояния элементов

Человеческие когнитивные способности так устроены, что мгновенное появление и исчезновение элементов может раздражать. Если вы мигнете в неподходящий момент, то можете пропустить переход и будете озадачены: «Что произошло?»

Кратковременные постепенные переходы помогут вам увидеть, что меняется и как вы переходите от одного состояния к другому, — и здесь приходят на выручку основные эффекты jQuery.

Есть три набора типов эффектов:

- отображение и скрытие (еще немного информации о тех методах, которые мы обсуждали в разделе 8.1);
- постепенное появление и исчезновение;
- соскальзывание вниз и скольжение вверх.

Рассмотрим более подробно каждый из данных наборов.

8.2.1. Постепенное отображение и скрытие элементов

Методы `show()`, `hide()` и `toggle()` на самом деле более гибкие, чем мы показали в предыдущем разделе. При вызове без аргументов они создают эффект простого манипулирования состоянием отображения элементов DOM, заставляя их мгновенно показываться или скрываться. Но когда им передаются аргументы, эти эффекты могут анимироваться так, что изменения в состоянии отображения задействованных элементов займут некоторое время.

Теперь вы готовы взглянуть на полный синтаксис этих методов.

Синтаксис метода: hide

```
hide(duration[, easing] [,callback])
hide(options)
hide()
```

Скрывает выбранные элементы. При вызове без параметров операция выполняется мгновенно, установкой свойства стиля `display` элементов в значение `none`. Если задать параметр `duration`, то элементы будут исчезать в течение указанного интервала времени за счет плавного уменьшения их ширины, высоты и уровня непрозрачности до нуля. По истечении этого времени свойство стиля `display` устанавливается в значение `none`, чтобы полностью удалить элементы с экрана.

Можно передать необязательную функцию смягчения имени, чтобы указать, как выполняется переход между состояниями.

Можно указать необязательную функцию обратного вызова (`callback`), которая будет вызвана по окончании воспроизведения эффекта.

Во второй версии вы предоставляете объект, содержащий определенные параметры, которые будут переданы методу (о свойствах расскажем позже).

Параметры

`duration` (Число|Строка) Определяет продолжительность воспроизведения эффекта. Может быть числом в миллисекундах или одной из предопределенных строк — `"slow"`, `"normal"` или `"fast"` (то же, что и передача 600, 400 или 200 соответственно). При отсутствии этого параметра и указании функции возвращаемого вызова в качестве первого параметра предполагается значение `"normal"`.

`easing` (Строка) Указывает имя функции смягчения, которая будет использоваться во время выполнения перехода из видимого состояния в скрытое. Функции смягчения определяют темп анимации в различных точках во время исполнения. Если анимация происходит, но данный параметр не задан, то по умолчанию задается `"swing"`. Подробнее об этих функциях см. в разделе 8.3.

`callback` (Функция) Функция, вызываемая по окончании воспроизведения анимации. Вызывается без параметров, однако контекст функции (`this`) установлен элементу, над которым выполняется операция. Вызывается для каждого элемента в наборе, к которому применяется анимационный эффект.

`options` (Объект) Необязательный набор параметров, который передается методу. Доступные параметры показаны в табл. 8.1.

Возвращает

Коллекцию jQuery.

Метод `hide()` дает нам возможность обсудить несколько моментов. Первый — это параметр `easing`, позволяющий указать функцию *смягчения*, или *плавности (easing)*. Термин используется для описания способа обработки процесса и темпа рамок анимации. Если мы позволим себе некоторые математические фантазии по поводу продолжительности анимации и позиции текущего времени, то станут возможными кое-какие интересные вариации эффектов. Но какие функции доступны? Ядро jQuery поддерживает только две функции: `linear`, прогрессирующую в постоянном темпе на протяжении анимации, и `swing`, прогрессирующую немного медленнее в начале и в конце анимации по сравнению с серединой. Вы можете спросить: «Почему только две функции?» Причина не в том, что jQuery не хочет воспринимать ваше творчество. Скорее, она хочет быть максимально компактной, насколько это возможно, и иметь возможность расширения любой дополнительной функции сторонними библиотеками. Как добавить больше функций смягчения, вы узнаете из раздела 8.3.

ПРИМЕЧАНИЕ

Значение "normal" параметра duration является лишь соглашением. Вы можете использовать любую строку, которая вам понравится (за исключением "slow" и "fast", разумеется), чтобы указать переход, который должен длиться 400 миллисекунд. Например, можете использовать "jQuery", "jQuery in Action" или "wow", получив точно такой же результат. Если хотите увидеть это своими глазами, то поищите свойство jQuery.fx.speeds в исходнике jQuery. Тем не менее мы настоятельно рекомендуем вам придерживаться обычного значения "normal", чтобы не причинять неудобств вашим коллегам.

Другой параметр, который стоит обсудить, — это `options`. С его помощью можно гибко настроить действие метода `hide()`. Свойства и допустимые значения приведены в табл. 8.1.

Таблица 8.1. Свойства и значения, допустимые в параметре options (в алфавитном порядке)

Свойство	Значение
always	(Функция) Функция, вызываемая после завершения анимационного эффекта или остановки без завершения. Объект Promise, переданный ей, разрешается либо отклоняется (мы обсудим это понятие в главе 13)
complete	(Функция) Функция, вызываемая после завершения анимационного эффекта
done	(Функция) Функция, вызываемая после завершения анимационного эффекта, что значит — когда объект Promise разрешается
duration	(Строка|Число) Продолжительность эффекта в виде количества миллисекунд или в качестве одной из предопределенных строк. То же, что описано ранее
easing	(Строка) Имя функции, используемой во время выполнения перехода от видимого состояния к скрытому состоянию. То же, что описано ранее

Свойство	Значение
fail	(Функция) Функция, вызываемая, когда анимационный эффект не был завершен. Объект Promise, переданный ей, отклоняется
progress	(Функция) Функция, выполняемая после каждого шага анимации. Вызывается только один раз за каждый анимационный элемент, независимо от количества анимационных свойств
queue	(Логический тип|Строка) Логический тип, указывающий, должна ли анимация быть помещенной в очередь эффектов (подробнее об этом в следующем разделе). Если переданное значение false, то анимационный эффект запустится немедленно. Значение по умолчанию — true. Если передается строка, то анимация добавляется в очередь, представленную этой строкой. Когда используется пользовательское имя очереди, анимационный эффект не запускается автоматически
specialEasing	(Объект) Отображение одного или нескольких свойств CSS, значения которых являются функциями плавности
start	(Функция) Функция, выполняемая, когда начинается анимационный эффект
step	(Функция) Функция, выполняемая для каждого анимированного свойства каждого анимационного элемента

Все эти свойства позволяют создавать удивительные эффекты. Мы рассмотрим данную тему в ближайшее время, разрабатывая три различных пользовательских эффекта. Мы знаем, что есть много неясных понятий и, возможно, вы слегка запутались. Но не волнуйтесь; все будет подробно описано в ближайшее время, так что оставайтесь с нами.

Теперь, когда вы основательно изучили метод `hide()` и его параметры, можете больше узнать о `show()`. Поскольку смысл параметров одинаковый для обоих методов, мы опишем только синтаксис метода `show()`.

Синтаксис метода: show

```
show(duration[ , easing] [, callback])
show(options)
show()
```

Выявляет скрытые элементы из набора сопоставленных элементов, которые должны быть выявлены. При вызове без параметров операция совершается мгновенно, установкой свойства стиля `display` элементов в первоначальное значение (`block` или `inline`).

Если задан параметр `duration`, то элементы будут постепенно появляться в течение указанного промежутка времени за счет плавного увеличения их размеров и повышения уровня непрозрачности. Можно указать необязательную функцию плавности, чтобы определить, как выполняется переход между состояниями.

Можно указать необязательную функцию обратного вызова `callback`, которая будет вызвана по окончании воспроизведения анимации.

Во второй версии можно указать объект, содержащий ряд параметров, чтобы перейти к методу, описанному в табл. 8.1.

Возвращает

Коллекцию jQuery.

Как вы заметили в нашем втором примере, jQuery предоставляет ярлык `toggle()` для переключения состояний одного или нескольких элементов. Его синтаксис приведен ниже, и в данном случае мы опустим описание уже показанных параметров.

Синтаксис метода: toggle

```
toggle(duration[, easing] [, callback])
toggle(options)
toggle(condition)
toggle()
```

Для скрытых элементов выполняет метод `show()`, а для видимых — `hide()`. Соответствующая семантика этих методов приведена в описании их синтаксиса.

В своей третьей форме `toggle()` показывает или скрывает выбранные элементы на основе оценки переданного условия. Если `true`, то элементы показываются, в противном случае скрываются.

Параметры

`condition` (Логический тип) Определяет, должны ли отображаться элементы (если `true`) или скрываться (если `false`).

Возвращает

Коллекцию jQuery.

Сделаем третий дубль на сворачиваемом модуле, анимирующем открытие и закрытие секций. Учитывая предыдущую информацию о методе `toggle()`, можно подумать, что единственное изменение, которое предстоит сделать, если нужно изменить код, приведенный в листинге 8.2, — это изменение вызова с `toggle()` на `toggle('slow')`. И вы будете правы. Но не торопитесь! Раз все было так просто, воспользуемся возможностью добавить дополнительные функции к модулю.

Допустим, для предоставления пользователю безошибочной визуальной подсказки вы хотите, чтобы надпись модуля отображалась в виде другого значка, когда модуль свернут. Вы могли бы внести изменение перед запуском анимации, но было бы гораздо эффективнее не ждать, пока анимационный эффект закончится.

jQuery 3: измененный функционал

jQuery 3 изменяет поведение `hide()`, `show()`, `toggle()`, `fadeIn()` и всех остальных связанных с ней методов, которые мы рассмотрим в данной главе. Все методы больше не будут переопределять каскад CSS. Это означает, что, если элемент является скрытым — потому что в таблице стилей у вас есть объявление `display: none;`, — вызов `show()` (или аналогичных методов, таких как `fadeIn()` и `slideDown()`) на данном элементе больше не покажет его. Чтобы понять изменение, рассмотрим следующий элемент:

```
<div class="hidden">Привет!</div>
```

Теперь предположим, что в таблице стилей у вас есть следующий код:

```
.hidden { display: none; }
```

Если вы используете версию jQuery до 3-й, то, когда напишете:

```
$('div').show('slow');
```

увидите красивую анимацию, которая в конечном итоге будет отображать элемент.

В jQuery 3 выполнение того же оператора не будет иметь никакого эффекта, поскольку библиотека не переопределяет декларацию CSS. Если вы хотите получить тот же результат в jQuery 3, то понадобится использовать такой оператор:

```
$('div')
    .removeClass('hidden')
    .hide()
    .show('slow');
```

либо следующий:

```
$('div')
    .removeClass('hidden')
    .css('display', 'none')
    .show('slow');
```

Изменение носит противоречивый характер и, вероятно, может испортить код многих сайтов. На время написания окончательная версия jQuery 3 не была опубликована, так что это новое поведение может быть изменено или полностью отменено. Пожалуйста, проверьте официальную документацию, чтобы узнать больше.

Вы не можете просто сделать вызов сразу после вызова метода анимации, потому что анимации не блокируют. Операторы, которые следуют за вызовом метода анимации, вероятно, выполнятся немедленно, еще до того, как появится возможность приступить к анимации. Это отличный повод использовать параметр обратного вызова `toggle()`.

Подход, которому вы будете следовать, состоит в том, что после завершения анимации вы замените текст тега `<span>`, содержащий значок. Используйте знак минус, если тело видимо (для указания на то, что оно может быть свернуто), и знак плюс, если тело свернуто (для указания обратного). Это же можно сделать, работая с CSS и свойством `content`: добавляя имя класса к модулю, чтобы указать, что он свернут, и удаляя имя класса в обратном случае. Но, поскольку наша книга не о CSS, мы пропустим данное решение. Листинг 8.3 показывает, какое изменение нужно внести в код.

Листинг 8.3. Анимированная версия модуля с изменением значка в заголовке

```
$('.icon-roll').click(function() {
   var $icon = $(this);
   $icon
        .closest('.module')
        .find('.body')
        .toggle('slow', function() {
           $icon.text($(this).is(':hidden') ? '+' : '-');
        });
});
```

Меняет текст значка, основываясь на состоянии тела в модуле

Вы можете найти страницу с описанными изменениями в файле `chapter-8/collapsible.module.take.3.html`, предоставленном с книгой.

Зная, сколько людей любят повозиться, мы создали удобный инструмент, который будем использовать для дальнейшего изучения работы представленных и остальных методов для добавления эффектов.

8.2.2. Знакомство со страницей jQuery Effects Lab

В главе 2 мы познакомили вас с понятием лабораторных страниц, помогающих экспериментировать с применением селекторов jQuery. Для этой главы мы создали страницу jQuery Effects Lab, чтобы изучить работу эффектов. Она доступна в файле `chapter-8/lab.effects.html`, предоставленном с книгой.

Результат загрузки этой страницы в вашем браузере показан на рис. 8.2.

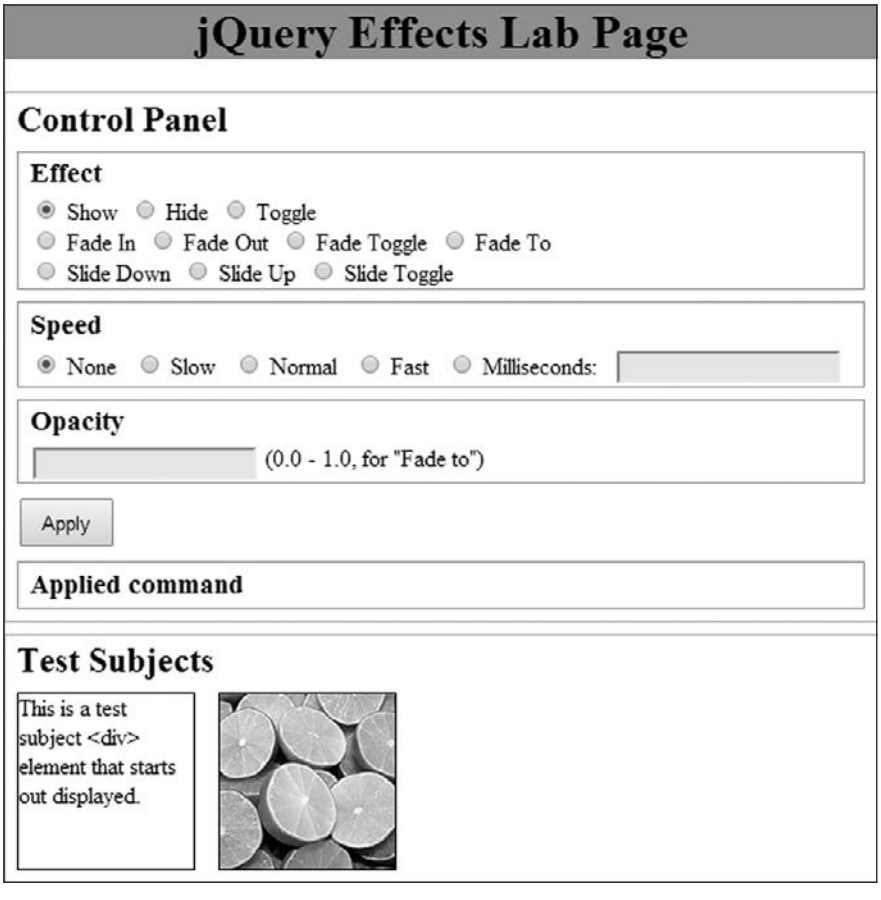


Рис. 8.2. Первоначальное состояние страницы jQuery Effects Lab, которая поможет вам проверить работу методов эффектов jQuery

Эта страница состоит из двух основных панелей: панели управления, в которой вы будете указывать, какие эффекты будут применены, и другой панели, содержащей четыре проверяемых элемента, над которыми будут действовать эффекты.

Вы можете недоуменно подумать: «Что происходит? Есть только два элемента».

Нет, все в порядке. Здесь четыре элемента, но два из них (второй `<div>` с текстом и второе изображение) изначально невидимы.

С помощью этой страницы рассмотрим действие методов, которые мы обсудили выше. Откройте страницу в вашем браузере и проделайте следующие эксперименты.

1. Оставив элементы управления в первоначальном состоянии, после загрузки страницы нажмите кнопку Apply (Применить). В результате будет выполнен метод `show()` без параметров. Примененное выражение отображается ниже указанной кнопки. Обратите внимание: изначально скрытые заголовки появляются немедленно. На вопрос, почему правое крайнее изображение немного блеклое, ответим, что значение непрозрачности (`opacity`) для него преднамеренно было установлено 50 % (фактическое значение в CSS — `0.5`).
2. Установите переключатель Effect (Эффект) в положение Hide (Скрыть) и нажмите кнопку Apply (Применить), чтобы выполнить метод `hide()` без параметров. В результате все тестовые заголовки исчезнут. Обратите внимание: панель Test subjects (Объекты исследования) свернулась. Это свидетельствует о том, что элементы не просто стали невидимыми, а были полностью *удалены из изображения страницы*.

ПРИМЕЧАНИЕ

Говоря, что элемент был удален из изображения страницы (здесь и далее при обсуждении визуальных эффектов), мы подразумеваем, что элемент больше не учитывается механизмом отображения браузера, точно так же, как если бы CSS-свойство display было установлено в значение none. Это не означает, что элемент был удален из дерева DOM, — ни один из эффектов не удаляет элементы из него.

3. Установите переключатель Effect (Эффект) в положение Toggle (Переключить) и нажмите кнопку Apply (Применить). Нажмите ее еще раз. Вы увидите, что каждое последующее выполнение метода `toggle()` переключает режим отображения тестовых заголовков.
4. Обновите страницу, чтобы вернуть ее в первоначальное состояние (в Firefox перенесите фокус в адресную строку и нажмите клавишу Enter — нажатие кнопки Reload (Обновить) не приведет элементы страницы в первоначальное состояние). Установите переключатель Effect (Эффект) в положение Toggle (Переключить) и нажмите кнопку Apply (Применить). Обратите внимание: два первоначально видимых элемента исчезли, а два других, первоначально скрытых, появились. Это доказывает, что метод `toggle()` применяется отдельно к каждому элементу, делая видимыми одни элементы и скрывая другие.
5. Перейдем в область анимационных эффектов. Обновите страницу, установите переключатель Effect (Эффект) в положение Show (Показать), а переключатель Speed (Скорость) — в положение Slow (Медленно). Нажмите кнопку Apply (Применить) и внимательно посмотрите, что происходит на панели Test subjects (Объекты исследования). Два скрытых элемента, вместо того чтобы немедленно появиться, будут постепенно расти, каждый из своего левого верхнего угла. Если вы хотите понаблюдать за эффектом в еще более медленном режиме, то обновите страницу,

установите переключатель Speed (Скорость) в положение Milliseconds (Миллисекунды) и введите в расположенном справа от него поле значение `5000`. Это увеличит продолжительность до пяти (мучительных) секунд и позволит вам подробно рассмотреть поведение эффекта.

6. Выбирая различные положения переключателя Effect (Эффект) — Show (Показать), Hide (Скрыть) и Toggle (Переключить) — и задавая разные скорости, поэкспериментируйте с этими эффектами, пока не почувствуете, что достаточно хорошо понимаете, как они действуют.

Вооружившись страницей jQuery Effects Lab и пониманием того, как работает этот первый набор эффектов, перейдем к изучению следующего.

8.2.3. Плавное растворение и проявление элементов

Если вы внимательно следили за работой методов `show()` и `hide()`, то наверняка заметили, что они масштабируют размер элементов (увеличивают либо уменьшают) и изменяют степень прозрачности по мере их увеличения или уменьшения. Следующий набор эффектов, `fadeIn()` и `fadeOut()`, воздействует только на прозрачность элементов, как только они достигают значения `0` (полностью прозрачны) или `1` (полностью видимы). В зависимости от вызванного метода для свойства `display` устанавливается либо значение `none`, либо что-то другое (мы объясняли данный механизм в разделе 8.1).

За исключением изменения размеров элементов, методы `fadeIn()` и `fadeOut()` действуют точно так же, как `show()` и `hide()` соответственно. Синтаксис этих двух пар методов похож, и значение параметров остается тем же, так что мы не будем повторяться. Единственное отличие между `fadeIn()`, `fadeOut()`, всеми другими связанными с анимацией методами и описанными прежде (`show()`, `hide()` и `toggle()`) состоит в следующем: поскольку первые при вызове без параметров выполняют переход, изменение из одного состояния в другое не происходит мгновенно.

С учетом этого их синтаксисы таковы.

Синтаксис метода: fadeIn

```
fadeIn(duration[, easing][, callback])
fadeIn(options)
fadeIn()
```

Делает видимыми элементы набора, которые до этого были скрыты, постепенно увеличивая их непрозрачность до их изначального значения. Данное значение равно либо уровню непрозрачности, первоначально примененному к элементу, либо 1 (полностью виден). Длительность изменения непрозрачности определяется параметром `duration`. Если он опущен, то значение по умолчанию составляет 400 миллисекунд (`"normal"`). Только скрытые элементы подвергаются действию метода.

Возвращает

Коллекцию jQuery.

Аналогично тому как удобный метод `toggle()` используется для элементов `hide()` и `show()`, основываясь на их текущем состоянии, у `fadeIn()` и `fadeOut()` есть `fadeToggle()`.

Синтаксис этого метода следующий.

Синтаксис метода: fadeToggle

```
fadeToggle(duration [, easing] [, callback])
fadeToggle(options)
fadeToggle()
```

Выполняет `fadeout()` на нескрытых элементах и `fadeIn()` на скрытых. Посмотрите на описание синтаксиса этих методов.

Возвращает

Коллекцию jQuery.

Еще немного поэкспериментируем со страницей jQuery Effects Lab. Загрузите ее и проведите несколько экспериментов, аналогичных представленным в предыдущем разделе, но с использованием Fade In, Fade Out и Fade Toggle (сейчас не беспокойтесь о Fade To; мы достаточно скоро до него доберемся).

Важно отметить: когда настроена непрозрачность элемента, эффекты jQuery `hide()`, `show()`, `toggle()`, `fadeIn()`, `fadeout()` и `fadeToggle()` запоминают изначальную непрозрачность элемента и соблюдают ее значение. На лабораторной странице мы намеренно установили изначальную непрозрачность изображения пояса в дальнем правом углу на 50 %, прежде чем его скрыть. На протяжении всех изменений непрозрачности, которые имеют место при применении эффектов jQuery, это изначальное значение никогда не теряется.

Следующий эффект, который предоставляет jQuery, — метод `fadeTo()`. Он настраивает непрозрачность элементов, подобно ранее рассмотренным fade-эффектам, но никогда не удаляет элементы из отображения. Прежде чем начать экспериментировать с `fadeTo()`, ознакомьтесь с его синтаксисом (мы опишем только новые параметры).

Синтаксис метода: fadeTo

```
fadeTo(duration, opacity[, easing][, callback])
```

Постепенно устанавливает непрозрачность элементов в объекте jQuery с их текущих настроек до новых настроек, указанных в `opacity`.

Параметры

`opacity` (Число) Целевая непрозрачность, которая будет установлена для элементов. Указывается как значение от `0` до `1`.

Возвращает

Коллекцию jQuery.

В отличие от других эффектов, которые регулируют непрозрачность, скрывая или отображая элементы, `fadeTo()` не запоминает первоначальную непрозрачность элемента. В этом есть смысл, так как цель данного эффекта — явно изменить непрозрачность до конкретного значения.

Загрузите лабораторную страницу и раскройте все элементы (сейчас вы уже знаете, как это сделать). Затем проделайте следующие эксперименты.

1. Выберите кнопку Fade To (Изменить непрозрачность до значения) и задайте значение скорости достаточно большое, чтобы увидеть, как действует эффект (4000 миллисекунд будет достаточно). Теперь введите в поле Opacity (Непрозрачность) число `0,1` — это поле предполагает ввод значения от `0` до `1` — и нажмите кнопку Apply (Применить). В результате непрозрачность объектов исследования плавно изменится до `0,1` за четыре секунды.
2. Введите в поле Opacity (Непрозрачность) значение `1` и нажмите кнопку Apply (Применить). Все элементы, включая изначально полупрозрачное изображение пояса, станут полностью непрозрачными.
3. Введите в поле Opacity (Непрозрачность) значение `0` и нажмите кнопку Apply (Применить). Все элементы «растворятся» до невидимого состояния, но, что самое примечательное, когда они полностью исчезнут, ограждающий модуль не свернется. В отличие от `fadeOut()`, эффект `fadeTo()` никогда не удаляет элементы из отображения страницы, даже когда они полностью невидимы.

Поэкспериментируйте с эффектом Fade To, пока не поймете, как он работает. Затем перейдите к следующей группе эффектов.

8.2.4. Закатывание и выкатывание элементов

Следующий набор эффектов, скрывающих или отображающих элементы, — `slideDown()` и `slideUp()`, — похож на эффекты `hide()` и `show()`, но элемент отображается или скрывается, «выезжая» из-под своей верхней границы или «заезжая» под нее без параметров анимации.

Как и в случае с методами `hide()` и `show()`, в эту группу эффектов входит переключение состояния элементов между видимым и невидимым: `slideToggle()`. Ниже приведен теперь уже знакомый синтаксис этих методов.

Синтаксис метода: slideDown

```
slideDown(duration[, easing][, callback])
slideDown(options)
slideDown()
```

Делает видимыми элементы набора, которые до этого были скрыты, постепенно увеличивая их вертикальный размер. Действию данного эффекта подвергаются только скрытые элементы.

Возвращает

Коллекцию jQuery.

Синтаксис метода: slideUp

```
slideUp(duration[, easing] [, callback])
slideUp (options)
slideUp()
```

Удаляет выбранные элементы, которые до этого были отображены, постепенно уменьшая их вертикальный размер.

Возвращает

Коллекцию jQuery.

Синтаксис метода: slideToggle

```
slideToggle(duration[, easing][, callback])
slideToggle(options)
slideToggle()
```

Выполняет метод `slideDown()` для скрытых элементов и метод `slideUp()` для всех видимых элементов. Подробные сведения о семантике этих методов приведены в соответствующих описаниях синтаксиса.

Возвращает

Коллекцию jQuery.

За исключением способа отображения и скрытия элементов, эти эффекты похожи по действию на эффекты `show/hide`. Можете убедиться в правоте данного утверждения с помощью экспериментов на странице jQuery Effects Lab, как и при изучении других эффектов.

8.2.5. Остановка анимации

Иногда вам по ряду причин понадобится остановить воспроизведение анимационного эффекта после его запуска. Это может быть связано с поступлением события от пользователя, по которому нужно сделать что-то еще, или возникнет необходимость запустить другой анимационный эффект. Нам поможет метод `stop()`.

Синтаксис метода: stop

```
stop( [queue] [, clearQueue[, gotoEnd])
```

Останавливает воспроизведение всех анимационных эффектов для элементов объектов jQuery.

Параметры

`Queue` (Строка) Название очереди, в которой нужно остановить анимацию (скоро мы вернемся к этому).

`clearQueue` (Логический тип) Если данный параметр указан и имеет значение `true`, то останавливается не только текущий анимационный эффект, но и все другие, находящиеся в очереди. Значение по умолчанию — `false`.

`gotoEnd` (Логический тип) Если этот параметр указан и имеет значение `true`, то текущему анимационному эффекту будет позволено полностью завершиться (в отличие от обычной остановки). Значение по умолчанию — `false`.

Возвращает

Коллекцию jQuery.

При использовании метода `stop()` имейте в виду: любые изменения, которые уже имели место для любых анимированных элементов, будут оставаться в силе. Кроме того, если вы вызываете `stop()` в наборе, когда jQuery выполняет анимацию как `slideUp()` и анимация не завершена, то часть элементов по-прежнему будет отображаться на странице. Если вы хотите восстановить первоначальное состояние элементов, то на вашей ответственности остается изменение значений CSS обратно в их исходное значение с применением метода jQuery `css()` или аналогичного метода.

Можно избежать только частичного завершения анимации элементов, присвоив значение `true` параметру `goToEnd`. Если хотите удалить все анимации в очереди и задать свойства CSS, как если бы *текущая* анимация была завершена, то должны вызвать `stop (true, true)`. Под указанием текущей анимации мы имеем в виду, что если вы выстроили цепочкой три анимации и во время работы первой вызываете `stop (true, true)`, то стиль элементов установится, как если бы первая анимация была завершена, а две другие не работали (они удаляются из очереди до выполнения).

В некоторых случаях при остановке анимации вам также нужно установить свойства CSS, как если бы все анимации были завершены. В таких ситуациях можно использовать `finish()`.

Синтаксис метода: finish

`finish [queue])`

Останавливает анимацию, которая в настоящее время осуществляется для элементов в объекте jQuery, удаляет все анимации в очереди (если таковые имеются) и сразу же устанавливает свойства CSS к их целевым значениям.

Параметры

`queue` (Строка) Название очереди, в которой следует остановить анимацию. Если не указано другое, то предполагается очередь `fx`, используемая jQuery по умолчанию.

Возвращает

Коллекцию jQuery.

Чтобы помочь вам визуализировать основную разницу между `stop()` и `finish()`, мы создали демо, которые доступны в файле `chapter-8/stop.vs.finish.html`, а также в JS Bin (http://jsbin.com/taseg/edit?html,js,output~~HEAD=pobj).

Помимо этих двух методов, для полного отключения всей анимации можно применять свойство (флаг) `jQuery.fx.off`. Присвоение ему значения `true` вызывает все

эффекты, которые вступают в силу немедленно без анимации. Другой флаг jQuery, имеющий отношение к анимации, называется `jQuery.fx.interval`. Мы рассмотрим эти флаги и причины их использования в главе 9, когда будем обсуждать и другие флаги, предоставляемые библиотекой.

Теперь, когда вы познакомились с эффектами, встроенными в ядро jQuery, узнаем, как можно написать свои собственные!

8.3. Добавление смягчающих функций jQuery

В предыдущем разделе вы узнали о параметре `easing` и функциях смягчения: `linear` и `swing`. Количество базовых эффектов, поставляемых с jQuery, намеренно небольшое, чтобы не увеличивать объем основной библиотеки. Но это вовсе не означает, что вы не можете использовать сторонние библиотеки для получения доступа к большему количеству смягчающих функций (снова напомним: смягчающие функции часто называются easings). Плагин jQuery Easing (https://github.com/gdsmith/jquery.easing) и библиотека jQuery UI (http://jqueryui.com) предоставляют дополнительные переходы и эффекты. Термин «плагин» (plugin — «расширение») должен быть вам знаком, но если нет, то это компонент, который добавляет специфическую особенность к существующему программному обеспечению, фреймворку или библиотеке. Подробнее мы рассмотрим эти компоненты и возможности создания своего собственного плагина jQuery в главе 12.

ПРИМЕЧАНИЕ

> jQuery UI представляет собой группу плагинов jQuery, облегчающих создание интерфейса веб-приложений. Это удивительная библиотека, и вам действительно стоит взглянуть на нее (после прочтения этой книги, конечно же). Помимо официального сайта, отличным ресурсом является «jQuery UI в действии» T. J. VanToll (Manning, 2014). Книга стоит прочтения!

При наличии нескольких вариантов решений люди обычно начинают спрашивать, какое из них является лучшим. Как это часто бывает, нет идеального решения для всех ситуаций, но выбор должен основываться на конкретном варианте применения. Чтобы помочь вам выбрать библиотеку, можем сказать следующее. Вы должны использовать jQuery UI для добавления эффектов, только если вы уже работаете с библиотекой в проекте по другим причинам (например, благодаря виджетам, которые она предоставляет). В противном случае вам лучше задействовать плагины. Причина заключается в том, что обе они предлагают одни и те же смягчающие функции, но плагин jQuery Easing более легкий (всего 3,7 Кбайт в своей сжатой версии), так как сосредоточен на одной функции, в то время как jQuery UI содержит много другого. Для полноты картины также отметим: загруженная страница jQuery UI предлагает возможность настройки сборки из библиотеки, в том числе только необходимых модулей, что приводит к уменьшению веса.

Вместе плагин jQuery Easing и библиотека jQuery UI добавляет 30 (да, 30, вы правильно прочитали) новых функций смягчения, перечисленных в табл. 8.2.

Таблица 8.2. Функции смягчения, добавленные и плагином jQuery, и функциями jQuery UIEasing

easeInQuad	easeOutQuint	easeInOutCirc
easeOutQuad	easeInOutQuint	easeInElastic
easeInOutQuad	easeInExpo	easeOutElastic
easeInCubic	easeOutExpo	easeInOutElastic
easeOutCubic	easeInOutExpo	easeInBack
easeInOutCubic	easeInSine	easeOutBack
easeInQuart	easeOutSine	easeInOutBack
easeOutQuart	easeInOutSine	easeInBounce
easeInOutQuart	easeInCirc	easeOutBounce
easeInQuint	easeOutCirc	easeInOutBounce

У плагина jQuery Easing есть одна особенность, стоящая обсуждения. Оно содержит ядро смягчения jQuery `swing`, которое используется по умолчанию, `jswing`. Кроме того, оно переопределяет `swing` таким образом, чтобы его поведение было эквивалентно смягчению `easeOutQuad`. Последнее из-за этого становится смягчением по умолчанию.

Использовать плагин jQuery Easing на ваших страницах можно двумя способами. Первый и более простой — включить его с помощью CDN. Большинство из вас наверняка помнят, что мы рассказали о CDN в главе 1, когда обсуждали, как их включать в jQuery.

Второй способ — это загрузить плагин из репозитория и сохранить его в папке, к которой у ваших страниц есть доступ. Например, при сохранении библиотеки в папке `js` можно включить его, написав:

```
<script src="js/jquery.easing.min.js"></script>
```

Для добавления jQuery UI на свои страницы можете задействовать те же способы: CDN или локальный хостинг. Чтобы включить библиотеку через jQuery CDN, если вы, допустим, используете версию 1.11.4, нужно написать:

```
<script src="http://code.jquery.com/ui/1.11.4/jquery-ui.min.js"></script>
```

При локальном размещении, если, допустим, вы сохранили библиотеку в папке `js`, напишите:

```
<script src="js/jquery-ui-1.11.4.min.js"></script>
```

Теперь, когда вы знаете, как включить плагин jQuery Easing и jQuery UI на ваших страницах, можете обращаться к предоставленным дополнительным функциям смягчения. Их очень просто применять: все, что нужно сделать, — это передать название перехода, которое хотите использовать в качестве параметра `easing`.

Для беглого ознакомления с эволюцией функций смягчения, перечисленных в табл. 8.2, загляните на http://api.jqueryui.com/easings/. Но этого, на наш взгляд, недостаточно; вы заслуживаете большего. Мы создали новую лабораторную страни-

цу, чтобы позволить вам увидеть эффекты, применяемые к методам, описанным в данной главе.

Эта новая страница, показанная на рис. 8.3, называется jQuery Advanced Effects Lab, и ее можно найти в файле исходного кода `chapter-8/lab.advanced.effects.html`, предоставленном с книгой.

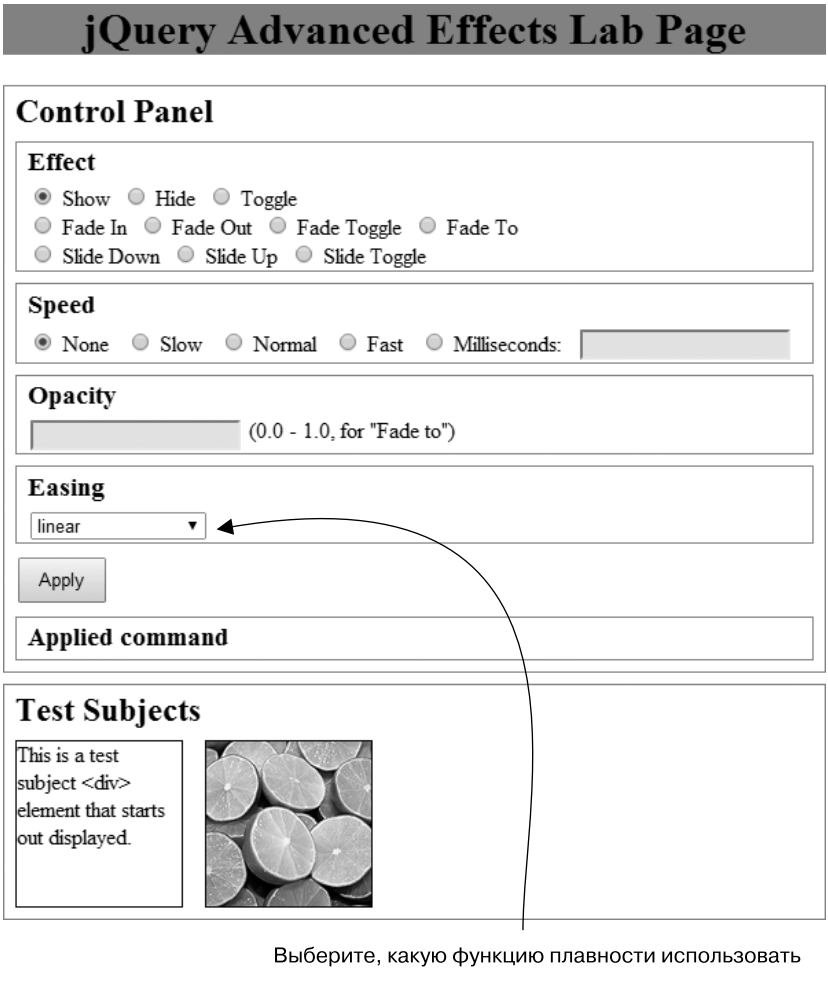


Рис. 8.3. Исходное состояние страницы jQuery Advanced Effects Lab

Поэкспериментируйте с этой страницей, пока не получите четкое представление о том, как различные функции смягчения могут изменить темп анимации элементов страниц.

До сих пор в вашем распоряжении были только предварительно созданные анимационные эффекты, но иногда возникнет необходимость создать собственные анимации. Посмотрим, как это можно сделать.

8.4. Создание собственных анимационных эффектов

В предыдущем разделе вы увидели, как легко интегрировать новые функции смягчения с помощью сторонних библиотек. Создание собственных анимаций — также удивительно простое дело.

jQuery предоставляет метод `animate()`, который позволяет применить собственные анимационные эффекты для набора элементов. Посмотрим на его синтаксис, в том числе описание параметров. Не так давно мы напомнили о значении параметров, так что, даже если они аналогичны, повторим их описание здесь. Таким образом, вам не нужно возвращаться на несколько страниц назад.

Синтаксис метода: animate

```
animate (properties[[, duration][, easing]][, callback])
animate (properties[, options])
```

Применяет анимационный эффект, заданный параметрами `properties`, ко всем элементам коллекции jQuery. Можно указать продолжительность, функцию смягчения и функцию обратного вызова. Функция обратного вызова вызывается после завершения анимации. В альтернативном варианте вдобавок к аргументу `properties` задается набор параметров `options`.

Параметры

`properties` (Объект) Объект свойств и значений CSS, согласно которым будет выполняться анимация. Анимация воспроизводится за счет изменения свойств стиля от текущих значений в элементе до значений, указанных в этом объекте. Свойства, состоящие из нескольких слов, можно указать с помощью верблюжьего регистра (например, `backgroundColor`) или взять имя в кавычки, не используя данный регистр (например, `'background-color'`).

`duration` (Число|Строка) Дополнительно определяет длительность эффекта в виде количества миллисекунд или строки с одним из предопределенных значений: `"slow"`, `"normal"` или `"fast"` (то же, что передача 600, 400 или 200 миллисекунд соответственно). Если этот параметр опущен и функция обратного вызова задается как первый параметр, то предполагается скорость `"normal"`.

`easing` (Строка) Задает необязательное имя смягчающей функции, используемой при совершении перехода. Эти функции определяют скорость анимации в различных местах во время исполнения. Если анимация выполняется, но данный параметр не указан, то по умолчанию задается `"swing"`. Подробнее см. в разделе 8.3.

`callback` (Функция) Необязательная функция, вызываемая после выполнения анимации. Вызывается без параметров, однако в контексте функции (`this`) ей передается элемент, над которым выполняется операция. Функция будет вызвана для каждого элемента в наборе, к которому применяется анимационный эффект.

`options` (Объект) Необязательный набор параметров, которые будут передаваться методу. Поддерживаемые свойства показаны в табл. 8.1 (см. выше).

Возвращает

Коллекцию jQuery.

Вы можете создавать собственные анимации с помощью свойств CSS и конечных значений этих свойств, которые должны быть достигнуты к моменту за-

вершения воспроизведения. Анимации начинаются с исходного значения стиля элемента и выполняются с изменением данного значения в сторону установленного значения. Промежуточные значения свойств стиля меняются в процессе анимации (автоматически обрабатываются при анимации движка) и определяются ее продолжительностью и функцией плавности.

Конечные значения можно задавать абсолютной величиной или смещением относительно начальных значений. Чтобы указать относительные значения, они должны предваряться оператором += или −= для указания соответственно положительного или отрицательного направления.

По умолчанию анимации добавляются в очередь на исполнение; применение нескольких анимаций к объекту заставит их работать последовательно. Если вы хотите запустить анимацию параллельно, то установите параметр `queue` в `false`.

Список свойств стилей CSS для анимации ограничен принимающими числовые значения, для которых есть логический переход от начального значения до целевого. Это ограничение вполне понятно — ведь как еще можно представить логический прогресс от исходного значения до конечного для нечислового свойства, такого как `background-image`? Для значений, которые представляют собой размеры, jQuery предоставляет по умолчанию блок пикселов, но вы также можете указать единицы em или проценты, добавив суффиксы `em` или `%`.

ПРИМЕЧАНИЕ

В CSS можно анимировать свойство color элемента, но вы не можете достичь этого эффекта, используя метод jQuery animate(), если применяете плагин jQuery.Color (https://github.com/jquery/jquery-color).

Часто анимированные свойства стиля включают в себя `top`, `left`, `width`, `height` и `opacity`. Но если это имеет смысл для эффекта, которого вы хотите достичь, то можете также анимировать числовые свойства стиля, такие как `font-size`, `margin`, `padding` и `border`.

Помимо конкретных значений для целевых свойств, можно также указать одну из строк — `"hide"`, `"show"` или `"toggle"`; jQuery вычислит конечное значение, соответствующее строке. Например, применение `"hide"` для свойства `opacity` приведет к снижению непрозрачности элемента до `0`. Использование любой из этих специальных строк имеет дополнительный эффект автоматического открытия или удаления элемента из отображения (например, методы `hide()` и `show()`). Следует также заметить: `"toggle"` запоминает начальное состояние таким образом, что оно может быть восстановлено последующим `"toggle"`.

Прежде чем идти дальше, мы подумали, что нужно дать вам лучшее представление о том, как функции смягчения могут изменить анимацию. По этой причине мы создали страницу, которую можно найти в файле `chapter-8/easings.html`, предоставленном с книгой. На ней показано, как можно использовать метод `animate()` с плагином jQuery Easing. С помощью данной страницы можно переместить изображение слева направо в соответствии с заданной функцией смягчения. Начальное состояние страницы показано на рис. 8.4.

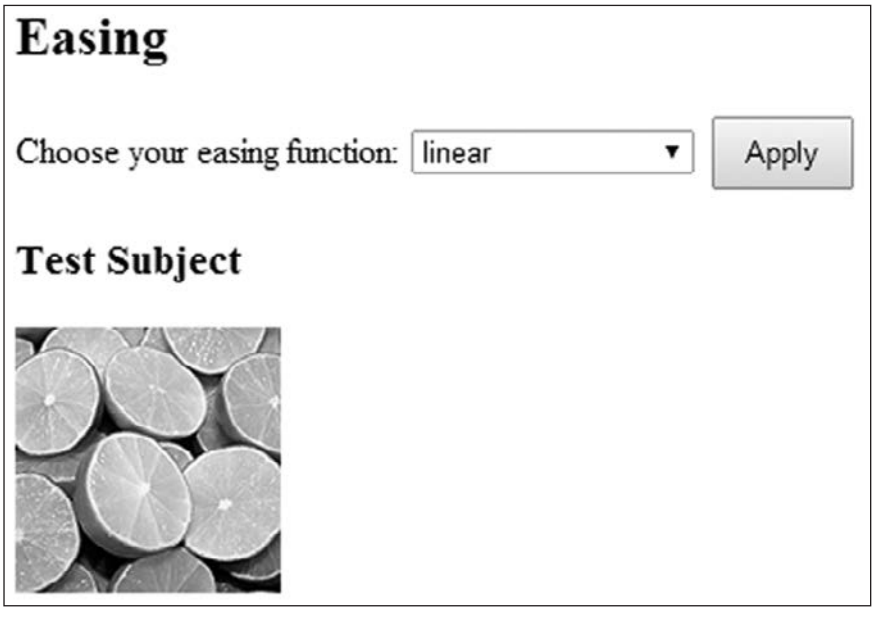


Рис. 8.4. Начальное состояние страницы easings.html

Попрактикуйтесь с этой страницей, пока не убедитесь, что хорошо понимаете, как работает функция смягчения. Если хотите протестировать больше функций смягчения, то не нужно обновлять страницу. После завершения анимации нажатие кнопки Apply (Применить) сбросит расположение изображения и запустит выбранную анимацию.

Теперь попробуйте свои силы в написании еще нескольких пользовательских анимаций.

8.4.1. Анимация пользовательского масштабирования

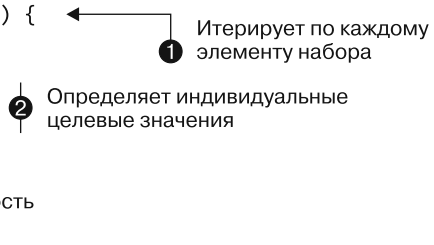
Рассмотрим простой анимационный эффект масштабирования — двукратное увеличение первоначальных размеров элементов. Пример реализации данного эффекта представлен в листинге 8.4.

ПРИМЕЧАНИЕ

Эту конкретную анимацию можно легко выполнить с помощью CSS в современных браузерах, использующих transform: scale(2). Как было отмечено во введении к этой главе, современные браузеры поддерживают много новых стандартов, но вы должны быть в курсе проблем, которые возникают в связи с ориентацией только на современные браузеры.

Листинг 8.4. Эффект масштабирования

```
$('.animated-elements').each(function() {
    var $this = $(this);
    $this.animate({
          width: $this.width() * 2,
          height: $this.height() * 2
},
2000
    );
});
```



Для реализации данного эффекта вы итерируете по всем элементам, содержащим в своем классе `animated-elements`, с помощью метода jQuery `each()` ❶. Поступая так, вы можете применять эффект отдельно к каждому элементу. Это важно, поскольку свойства, которые нужно указывать для каждого элемента, зависят от индивидуальных размеров элемента ❷. Если бы вы знали заранее, что будете анимировать единственный элемент (как в случае использования селектора ID), или применяли бы один и тот же набор значений ко всем элементам, то могли бы обойтись и без вызова метода `each()` и анимировать весь набор напрямую.

В пределах обратного вызова, переданного `each()`, метод `animate()` применяется к каждому элементу, по одному за раз. Вы получаете доступ к элементу с помощью `this` и устанавливаете значения свойств стиля `width` и `height`, чтобы удвоить первоначальные размеры элемента. В результате в течение двух секунд (как указано в параметре `duration`, который равняется `2000` ❸) каждый элемент в объекте jQuery вырастет в сравнении со своим первоначальным размером в два раза.

Мы предоставим вам демо для экспериментов с кодом, созданным в этом разделе, но сейчас попробуем создать нечто более экстравагантное.

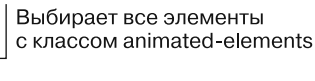
8.4.2. Пользовательский эффект падения

Предположим, вам захотелось анимировать удаление элемента с экрана так, чтобы это было заметно. Анимационный эффект, который вы будете использовать для достижения данной цели, покажет падение элемента со страницы и его исчезновение с экрана.

Есть вероятность, что, немного подумав, вы можете прийти к следующему решению. Изменяя значение свойства `top` элемента, можно заставить его перемещаться вниз по странице, имитируя падение, а изменение `opacity` усилит эффект исчезновения. И наконец, когда все это будет сделано, можно удалить элемент с дисплея. Такой эффект падения показан в листинге 8.5.

Листинг 8.5. Пользовательский эффект падения

```
$('.animated-elements').each(function() {
   var $this = $(this);
   $this
```



```
        .css('position', 'relative')
        .animate({
           opacity: 0,
           top: $(window).height() - $this.height() -
                $this.position().top
        },
        'normal',
        function() {
           $this.hide();
        }
   );
});
```

❶ Выбирает элемент из статического потока

❷ Подсчитывает высоту падения

❸ Выполняет функцию после завершения анимации

❹ Удаляет элемент с экрана

Здесь несколько больше операций, чем в предыдущем пользовательском эффекте. Вы снова итерируете по элементам, но на этот раз регулируете положение и степень непрозрачности элементов. Чтобы установить значение свойства `top` элемента относительно его исходного значения, сначала вы должны изменить свойство стиля CSS `position`, установив его в значение `relative` ❶.

Затем вы указываете конечное значение `0` для свойства `opacity` и вычисленное значение `top`. Не надо перемещать элемент за нижнюю границу окна — могут появиться нелюбимые пользователями полосы прокрутки, которых, возможно, до сего момента на странице не было. Нужно лишь привлечь внимание к анимационному эффекту — собственно, для этого он и создается! Используйте высоту и вертикальную позицию элемента и высоту окна, чтобы вычислить, как низко упадет элемент ❷. Разумеется, расчеты имеют смысл только тогда, когда имеется достаточное пространство между элементами и нижней частью страницы.

После завершения анимации нужно удалить элемент из отображения страницы, поэтому укажите функцию обратного вызова ❸, которая применит к элементу (доступному данной функции через ее контекст) неанимированную версию метода `hide()`.

ПРИМЕЧАНИЕ

В этом примере мы сделали немного больше, чем требовалось для воспроизведения эффекта, например показали, как дождаться окончания анимации и выполнить дополнительные операции с помощью функции обратного вызова. Если бы мы указали для свойства opacity значение hide вместо 0, то после завершения воспроизведения анимационного эффекта элемент был бы автоматически удален и функция обратного вызова не понадобилась бы.

8.4.3. Пользовательский эффект рассеивания

Допустим, вместо эффекта падения вы хотите воспроизвести эффект рассеивания, подобный таянию клубов дыма в воздухе. Чтобы получить такой эффект, можно скомбинировать эффекты масштабирования и изменения непрозрачности, увеличивая размеры элемента и одновременно растворяя его до полного исчезновения.

Одна из проблем, которую вам придется решить для большей реалистичности, — элемент должен увеличиваться во все стороны, без привязки к началу координат в верхнем левом углу элемента. По мере роста элемента его *центр* должен оставаться на месте. Поэтому, помимо изменения размеров элемента, вам понадобится изменять и его координаты.

Реализация эффекта рассеивания приводится в листинге 8.6.

Листинг 8.6. Пользовательский эффект рассеивания

```
$('.animated-elements').each(function() {
  var $this = $(this);
  var position = $this.position();
  $this
        .css({
           position: 'absolute',
           top: position.top,
           left: position.left
        })
        .animate({
              opacity: 'hide',
              width: $this.width() * 5,
              height: $this.height() * 5,
              top: position.top - ($this.height() * 5 / 2),
              left: position.left - ($this.width() * 5 / 2)
           },
           'fast'
        );
});
```

Выбирает все элементы с классом animated-elements и итерирует по ним ❶

Вытесняет элемент из статического потока ❷

Устанавливает размеры элемента, позицию и непрозрачность ❸

В данной анимации вы выбираете все элементы с классом `animated-elements` и итерируете по ним ❶. Затем уменьшаете непрозрачность до `0` при увеличении элемента в пять раз по отношению к его первоначальному размеру и регулируете его положение на половину этого нового размера, в результате чего центр элемента остается в том же самом положении ❸. Не нужно вытеснять соседний анимированный элемент во время роста целевого элемента — так вы полностью удалите его из потока, изменив его позицию к `absolute` и явно установив координаты его позиции ❷. Поскольку вы указали значение `hide` для свойства `opacity`, элементы будут автоматически скрыты (удалены из отображения страницы) после того, как анимация будет завершена.

Действие каждого из этих трех пользовательских эффектов можно наблюдать, загрузив страницу из файла `chapter-8/custom.effects.html`, предоставленного с книгой (рис. 8.5).

Мы бы очень хотели продемонстрировать здесь эти эффекты, но процедура создания скриншотов имеет очевидные ограничения, надеемся, вы понимаете, что мы имеем в виду. Но на рис. 8.6 показан эффект рассеивания в процессе воспроизведения. Поэкспериментируйте с различными эффектами на данной странице и понаблюдайте за их поведением.

Рис. 8.5. Пользовательские эффекты, которые вы разработали (масштабирование, падение и рассеивание), можно наблюдать в действии, используя кнопки, предоставленные на странице примера

Рис. 8.6. Эффект рассеивания увеличивает и перемещает картинку при одновременном уменьшении ее непрозрачности

До этого момента во всех примерах анимации, которые мы рассмотрели, применялся один-единственный метод. Обсудим, как выполняется воспроизведение анимационных эффектов при использовании нескольких методов.

8.5. Анимации и очереди

Вы уже увидели, как можно задействовать сразу несколько свойств элементов с помощью одного анимационного метода, но мы не изучили, как поведут себя анимационные эффекты при одновременном вызове нескольких методов. Рассмотрим данный аспект в этом разделе.

8.5.1. Одновременные анимации

Как думаете, что произойдет, если запустить такой код?

```
$('#test-subject').animate({left: '+=256'}, 'slow');
$('#test-subject').animate({top: '+=256'}, 'slow');
```

Вы знаете, что метод `animate()` не блокируется на время выполнения анимации, равно как и любой другой метод анимации. Можно доказать это с помощью следующего блока кода:

```
console.log(1);
$('#test-subject').animate({left: '+=256'}, 'slow');
console.log(2);
```

Выполнение данного кода покажет, что сообщения 1 и 2 выводятся сразу же, друг за другом, без паузы на воспроизведение анимации. Если вы хотите доказать, что это верно, то посмотрите файл `chapter-8/asynchronous.animate.html`, предоставленный с книгой, или JS Bin, который мы создали для вас (http://jsbin.com/pulik/edit?html,js,console,output).

Посмотрим на развитие событий при выполнении кода с двумя вызовами методов анимации (первый фрагмент в этом разделе). Поскольку вызов второго метода не блокируется первым, логичным представляется такое утверждение: обе анимации будут проходить одновременно (или с отставанием в несколько миллисекунд), а общий визуальный эффект будет комбинацией двух отдельных эффектов. В данном случае первый изменяет свойство `left` стиля, а второй — `top`, поэтому можно было бы предположить, что в результате их выполнения произойдет перемещение испытуемого элемента по диагонали.

Проверим. В файле `chapter-8/revolutions.html`, предоставленном с книгой, находятся два изображения (к одному из которых применим анимацию) и кнопка, запускающая эксперимент. Вдобавок на консоли будут выводиться сообщения. На рис. 8.7 приводится внешний вид страницы в начальном состоянии.

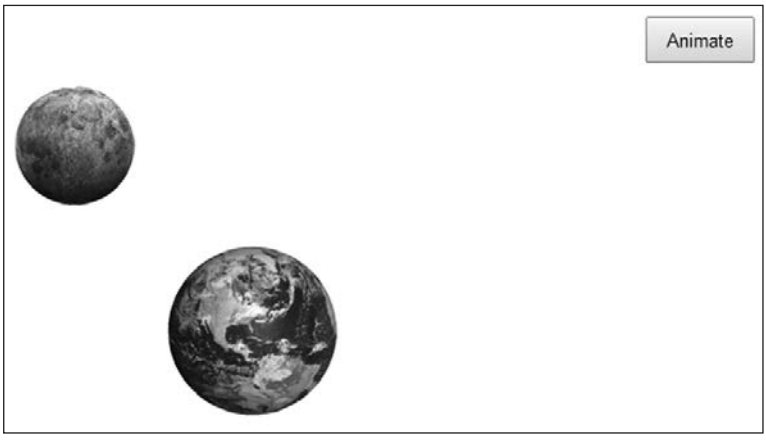


Рис. 8.7. Начальное состояние страницы, на которой вы наблюдаете поведение нескольких одновременных анимационных эффектов

Кнопка Animate (Анимировать) действует, как показано в листинге 8.7.

Листинг 8.7. Реализация воспроизведения сразу нескольких анимационных эффектов

```
function formatDate(date) {
   return (date.getHours() < 10 ? '0' : '') + date.getHours() +
   ':' + (date.getMinutes() < 10 ? '0' : '') + date.getMinutes() +
   ':' + (date.getSeconds() < 10 ? '0' : '') + date.getSeconds() +
   '.' + (date.getMilliseconds() < 10 ?
         '00' : (date.getMilliseconds() < 100 ? '0' : '')) +
   date.getMilliseconds();
}

$('#button-animate').click(function() {
   var $moonImage = $('img[alt="moon"]');
   console.log('B ' + formatDate(new Date()) + ' 1');
   $moonImage.animate({left: '+=256'}, 2500);
   console.log('B ' + formatDate(new Date()) + ' 2');
   $moonImage.animate({top: '+=256'}, 2500);
   console.log('B ' + formatDate(new Date()) + ' 3');
   $moonImage.animate({left: '-=256'}, 2500);
   console.log('B ' + formatDate(new Date()) + ' 4');
   $moonImage.animate({top: '-=256'}, 2500);
   console.log('B ' + formatDate(new Date()) + ' 5');
});
```

Определяет обработчик нажатия для кнопки

В обработчике события `click` кнопки ❶ запускаются четыре анимационных эффекта, один за другим, между которыми вставлены вызовы функции `console.log()`, которые позволяют наглядно увидеть, когда запускаются эти эффекты.

Откройте эту страницу в браузере и нажмите кнопку Animate (Анимировать). Как и следовало ожидать, сообщения с 1 по 5 немедленно появляются в консоли, как показано на рис. 8.8, отставая друг от друга на несколько миллисекунд.

Рис. 8.8. Сообщения консоли появляются одно за другим, подтверждая тем самым, что методы анимации не блокируют друг друга до завершения

Но как ведет себя анимация? Если вы внимательно исследуете код в листинге 8.7, то увидите следующее: два анимационных эффекта изменяют свойство `top` и еще два — свойство `left`. Фактически в каждой из этих пар производятся противоположные действия. Итак, какого результата следует ожидать? Можно было бы предположить: они должны уравновесить друг друга и изображение Луны (испытуемый объект) должно остаться на месте. Однако нет. Как видите, каждый анимационный эффект воспроизводится серийно, один за другим, в результате чего Луна делает полный оборот вокруг Земли (хотя и по очень неестественной квадратной орбите, что наверняка привело бы Кеплера в замешательство).

Что происходит? С помощью вывода сообщений в консоль вы доказали следующее: выполнение методов не блокируется на время применения анимационных эффектов. Сами они при этом воспроизводятся поочередно в порядке их запуска (по крайней мере относительно друг друга), так как внутри jQuery ставит эти анимации в очереди и выполняет их последовательно.

Обновите страницу для очистки консоли и нажмите кнопку Animate (Анимировать) три раза подряд. (Пауза между щелчками должна быть достаточной, чтобы два соседних щелчка не интерпретировались как двойной щелчок.) Вы заметите, как в консоли практически одновременно появятся 15 сообщений, указывая на то, что обработчик щелчка был вызван трижды. И затем подождите, пока Луна сделает три оборота вокруг Земли. Все 12 анимационных эффектов jQuery поставит в очередь и выполнит поочередно, поскольку библиотека создает очередь для каждого анимируемого элемента с именем `fx` как раз для этого случая.

Более того, jQuery позволяет вам создавать собственные очереди воспроизведения. Разберемся в этом.

8.5.2. Очереди функций для выполнения

Поочередное воспроизведение анимационных эффектов подразумевает использование очередей функций. Но есть ли здесь реальное преимущество? В конце концов, методы запуска анимационных эффектов позволяют определять функцию обратного вызова, которая выполняется по окончании воспроизведения эффекта, так почему бы просто не запускать следующий анимационный эффект из функции обратного вызова предыдущего?

Добавление функций в очередь

Еще раз просмотрим фрагмент листинга 8.7 (для большей ясности мы убрали вызовы функции `console.log()`):

```
var $moonImage = $('img[alt="moon"]');
$moonImage.animate({left: '+=256'}, 2500);
$moonImage.animate({top: '+=256'}, 2500);
$moonImage.animate({left: '-=256'}, 2500);
$moonImage.animate({top: '-=256'}, 2500);
```

Сравните этот фрагмент с эквивалентной ему реализацией, основанной на использовании функций обратного вызова:

```
var $moonImage = $('img[alt="moon"]');
$moonImage.animate({left: '+=256'}, 2500, function(){
  $moonImage.animate({top: '+=256'}, 2500, function(){
    $moonImage.animate({left: '-=256'}, 2500, function(){
      $moonImage.animate({top: '-=256'}, 2500);
    });
  });
});
```

Вариант на основе функций обратного вызова был не намного сложнее, но совершенно очевидно, что первоначальный вариант удобнее читать (да и написать его проще). А если потребуется намного более сложная реализация функций обратного вызова... В общем, легко увидеть, как возможность установки анимационных эффектов в очередь упрощает программный код.

Очереди могут быть созданы для любого элемента, и отдельные очереди создаются с помощью уникальных имен (за исключением имени `fx`, которое зарезервировано для использования с анимационными эффектами). Метод для добавления экземпляра функции в очередь носит вполне предсказуемое имя `queue()` и имеет три варианта.

Синтаксис метода: queue

```
queue([name])
queue([name], function)
queue([name], queue)
```

Первый вариант метода возвращает очередь с переданным именем, которая уже установлена на первом элементе набора, в виде массива функций.

Второй вариант добавляет переданную функцию в конец названной очереди для всех элементов из набора. Очередь с таким именем отсутствует на элементе, она создается.

Последний вариант замещает существующие очереди на всех элементах набора на переданную очередь.

Когда параметр `name` опущен, по умолчанию предполагается очередь `fx`.

Параметры

`name` (Строка) Имя очереди, которая должна быть извлечена, добавлена или замещена. Если этот параметр опущен, то подразумевается имя очереди по умолчанию `fx`.

`function` (Функция) Функция, подлежащая добавлению в конец очереди. При вызове ее контекст (`this`) будет передаваться элементу DOM, на котором установлена очередь. Данная функция передается только одному аргументу с именем `next`. `next` — другая функция, при вызове автоматически убирающая из очереди следующий элемент и сохраняющая движение очереди.

`queue` (Массив) Массив функций, которые заменят функции, уже имеющиеся в очереди.

Возвращает

Массив функций — для первого варианта. Коллекцию jQuery — для остальных вариантов.

Чаще всего метод `queue()` служит для добавления функций в конец названной очереди, но точно так же может применяться для получения существующих функций, находящихся в очереди, или для замены списка функций в очереди. Обратите внимание: третья форма, в которой методу `queue()` передается массив функций, не может использоваться для добавления нескольких функций в конец очереди, так как все функции, имеющиеся в очереди, будут удалены. Чтобы добавить в конец очереди сразу несколько функций, необходимо получить массив имеющихся в очереди функций, задействуя первый вариант метода, добавить в него новые функции и передать получившийся массив в очередь с помощью третьего варианта метода `queue()`.

Вы можете спросить: «Здесь не будет примеров?» Используя метод `queue()`, можете добавлять новые анимации в конец очереди. Если вы подумали, что мы не обсудим, каким образом их воспроизводить, то не волнуйтесь. С этой целью мы подготовили для вас демо.

Вызов функций, находящихся в очереди

Просто добавление функций в очередь не имело бы смысла без возможности запускать эти очереди функций на выполнение. Представляем метод `dequeue()`.

Синтаксис метода: dequeue

`dequeue([name])`

Удаляет самую первую функцию из названной очереди для каждого элемента в объекте jQuery и выполняет ее для каждого элемента.

Параметры

`name` (Строка) Имя очереди, из которой удаляется самая первая функция. Если этот параметр опущен, то подразумевается имя очереди по умолчанию `fx`.

Возвращает

Коллекцию jQuery.

При вызове метода `dequeue()` выполняется самая первая функция в очереди для каждого элемента в наборе с контекстом функции для вызова (`this`), передаваемым текущему элементу. Рассмотрим код в листинге 8.8, который также доступен в файле `chapter-8/manual.dequeue.html`, предоставленном с книгой.

Листинг 8.8. Создание и выполнение очередей функций для нескольких элементов

```
<!DOCTYPE html>
<html>
   <head>
      <title>Ручной вывод функций из очереди</title>
      <link rel="stylesheet" href="../css/main.css" />
      <style>
         button
         {
            display: block;
            margin: auto;
         }
      </style>
</head>
<body>
   <button>Вывести из очереди</button>

   <img src="images/earth.png" alt="Earth" />
   <img src="images/moon.png" alt="Moon" />

   <script src="../js/jquery-1.11.1.min.js"></script>
   <script>
      var $images = $('img');
      $images
            .queue('chain', function() {
               console.log('Первая: ' + $(this).attr('alt'));
            })
            .queue('chain', function() {
               console.log('Вторая: ' + $(this).attr('alt'));
            })
            .queue('chain', function() {
               console.log('Третья: ' + $(this).attr('alt'));
            })
            .queue('chain', function() {
               console.log('Четвертая: ' + $(this).attr('alt'));
            });

      $('button').click(function() {
         $images.dequeue('chain');
      });
   </script>
  </body>
</html>
```

Устанавливает четыре функции в очереди (1)

Выводит из очереди по одной функции (2)

В данном примере у вас есть два изображения, в которых вы добавляете функции в очередь с именем `chain` (1). Внутри каждой функции вы выводите в консоль атрибут `alt` из элемента DOM, работающий как контекст функции, и номер, определяющий его расположение в очереди. Таким образом, можно сказать, какая функция работает и из какой очереди элемента. То есть на данный момент вы не делаете ничего особенного с этими изображениями (они не будут двигаться).

После нажатия кнопки Dequeue (Вывести из очереди) обработчик щелчка ❷ вызывает одно выполнение метода `dequeue()`. Продолжаем. После нажатия кнопки в консоли появляются сообщения, как показано на рис. 8.9.

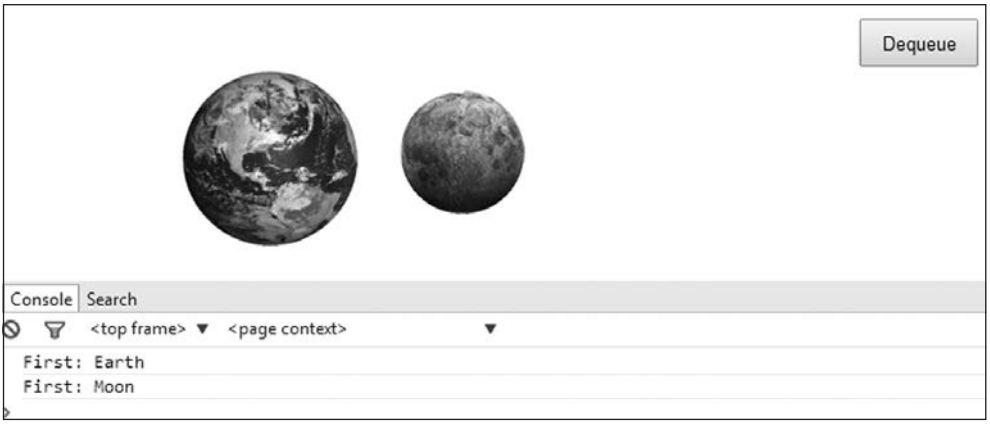


Рис. 8.9. Нажатие кнопки Dequeue (Вывести из очереди) запускает отдельный экземляр функции из очереди, по одному для каждого изображения, на которые она установлена

Как видите, первая функция, добавленная в очередь `chain`, была вызвана дважды: для изображения отдельно Земли и Луны. Дальнейшие нажатия кнопки будут удалять из очередей последующие функции, по одной за раз, и выполнять их, пока очереди не опустеют, после чего вызов метода `dequeue()` не будет делать ничего.

В данном примере удаление функций из очереди выполнялось вручную — вам потребовалось четырежды нажать кнопку для четырехкратного вызова метода `dequeue()`, чтобы вызвать все четыре функции. Однако часто вам понадобится вызвать исполнение всего набора функций, входящих в очередь. В подобных ситуациях часто используется прием, когда метод `dequeue()` вызывается внутри функции, добавляемой в очередь, или применяется параметр `next`, переданный функции в очереди. Одним из этих двух способов вы запускаете выполнение следующей функции в очереди, создавая таким образом цепочку вызовов.

Рассмотрим следующие изменения в коде из листинга 8.8, где используются обе вышеперечисленные техники:

```
var $images = $('img');
$images
        .queue('chain', function(next) {
          console.log('First: ' + $(this).attr('alt'));
          next();
        })
        .queue('chain', function(next) {
          console.log('Second: ' + $(this).attr('alt'));
          next();
        })
        .queue('chain', function() {
```

```
    console.log('Third: ' + $(this).attr('alt'));
    $(this).dequeue('chain');
  })
  .queue('chain', function() {
    console.log('Fourth: ' + $(this).attr('alt'));
  });
```

Измененную версию файла `chapter-8/manual.dequeue.html`, также содержащего предыдущий фрагмент кода, можно найти в файле `chapter-8/automatic.dequeue.html`.

Откройте эту страницу в браузере и нажмите кнопку Dequeue (Вывести из очереди). Обратите внимание: теперь один щелчок вызывает выполнение всех функций в очереди, как показано на рис. 8.10. Кроме того, отметим, что у последней функции, добавленной в очередь, нет вызова `dequeue()`, так как к тому времени очередь уже будет пустой.

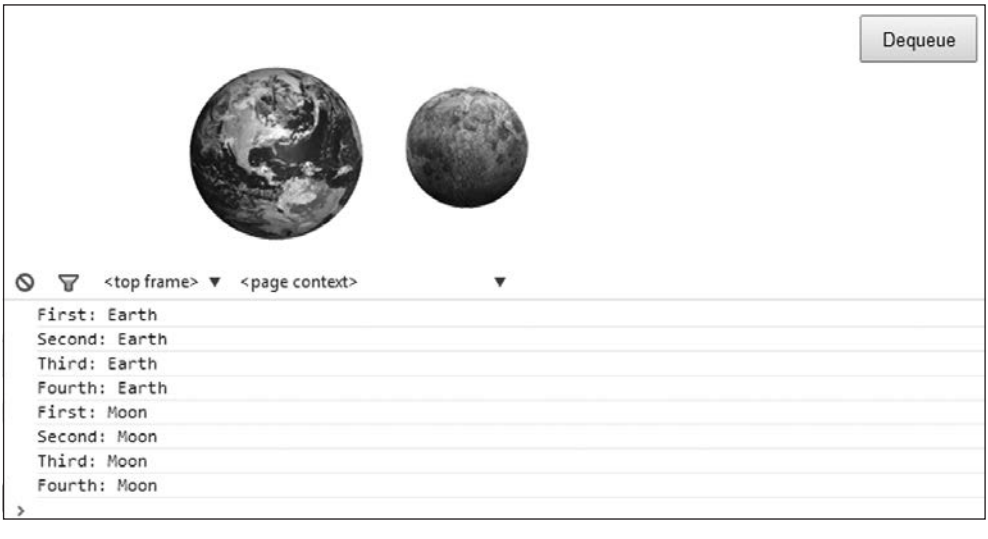


Рис. 8.10. Нажатие кнопки Dequeue (Вывести из очереди) выполняет все функции в очереди для каждого из изображений, на которых они установлены

Используя `dequeue()`, можно выполнить функцию на клиентской стороне очереди. Но иногда необходимо удалить все функции, сохраненные в очереди, без их выполнения.

Удаление функций из очереди без их выполнения

В случае возникновения такой надобности можете воспользоваться методом `clearQueue()`.

Несмотря на сходство с методом `stop()`, `clearQueue()` предназначен для использования с любыми очередями, а не только с анимационными эффектами.

Синтаксис метода: clearQueue

`clearQueue([name])`

Удаляет все невыполненные функции из названной очереди.

Параметры

`name` (Строка) Имя очереди, из которой удаляются функции без их выполнения. Если этот параметр опущен, то подразумевается имя очереди по умолчанию `fx`.

Возвращает

Коллекцию jQuery.

Отложенное выполнение функций в очереди

Еще одна операция, которая может потребоваться при работе с очередями функций, — добавление задержки между вызовами функций. Эту операцию можно выполнить с помощью метода `delay()`.

Синтаксис метода: delay

`delay(duration[, queueName])`

Добавляет задержку перед выполнением каждой функции из очереди с заданным именем.

Параметры

`duration` (Число|Строка) Определяет продолжительность задержки либо в виде числа миллисекунд, либо в виде одной из строк — `"fast"`, `"normal"` или `"slow"`, представляющих значения 200, 400 и 600 миллисекунд соответственно.

`queueName` (Строка) Имя очереди, для которой устанавливается задержка. Если этот параметр опущен, то подразумевается имя очереди по умолчанию `fx`.

Возвращает

Коллекцию jQuery.

Чтобы понять, когда может пригодиться `delay()`, представьте: у вас есть изображение, содержащее `my-image` в качестве ID, которое вы хотите скрыть, а затем показать снова через одну секунду. Можно сделать это с помощью следующего оператора:

```
$('#my-image')
    .slideUp('slow')
    .delay(1000)
    .slideDown('fast');
```

Данный метод полезен, но недостаточно гибкий, так как невозможно отменить задержку после того, как она установлена.

Прежде чем перейти к следующей главе, необходимо упомянуть еще один момент, касающийся добавления функций в очередь.

8.5.3. Добавление функций в очередь анимационных эффектов

Мы уже упоминали, что jQuery использует очередь с именем `fx`, куда помещаются функции воспроизведения анимационных эффектов. Но если вам понадобится добавить ваши собственные функции в эту очередь, чтобы обеспечить выполнение некоторых действий в процессе воспроизведения эффектов? Теперь, когда вы познакомились с методами для работы с очередями, можете это сделать без помех!

Вернемся к предыдущему примеру в листинге 8.7, где мы использовали четыре анимационных эффекта, которые заставляют Луну сделать полный оборот вокруг Земли. Представьте теперь, что нам потребовалось после второго анимационного эффекта (опускающего картинку вниз) изменить цвет фона с изображением Луны на черный. Если бы вы просто добавили вызов метода `css()` между вторым и третьим анимационным эффектом, как показано ниже:

```
var $moonImage = $('img[alt="Луна"]');
$moonImage.animate({left: '+=256'}, 2500);
$moonImage.animate({top: '+=256'}, 2500);
$moonImage.css({backgroundColor: 'black'});
$moonImage.animate({left: '-=256'}, 2500);
$moonImage.animate({top: '-=256'}, 2500);
```

то оказались бы разочарованы, так как изменение цвета фона произошло бы немедленно, возможно даже еще до начала воспроизведения первого анимационного эффекта (не забываем, что `animate()` относится к неблокирующим методам). Рассмотрим теперь следующий фрагмент (изменения выделены жирным шрифтом):

```
var $moonImage = $('img[alt="Луна"]');
$moonImage.animate({left: '+=256'}, 2500);
$moonImage.animate({top: '+=256'}, 2500);
$moonImage.queue('fx',
  function() {
    $(this)
          .css({backgroundColor: 'black'});
          .dequeue('fx');
  }
);
$moonImage.animate({left: '-=256'}, 2500);
$moonImage.animate({top: '-=256'}, 2500);
```

Здесь вы обернули вызов метода `css()` функцией, которая помещается в очередь с именем `fx` с помощью метода `queue()`. (Мы могли бы опустить имя очереди, потому что `fx` используется по умолчанию, тем не менее указали его явно для большей ясности.) В результате ваша функция изменения цвета фона будет помещена в очередь эффектов и вызвана как часть цепочки функций, выполняемых в процессе воспроизведения анимации, между воспроизведением второго и третьего эффектов.

Но обратите внимание! После вызова метода `css()` вы вызываете `dequeue()`, передавая ему очередь `fx`. Это совершенно необходимо, чтобы не прервать вы-

полнение последовательности эффектов. Если не вызвать его в данном месте, то воспроизведение последовательности анимационных эффектов будет остановлено, так как ничто не вызывает следующую функцию в очереди. Невыполненные функции будут оставаться в очереди эффектов, пока какой-то другой механизм не вызовет метод `dequeue()` или пока не произойдет обновление страницы, которое уничтожит все очереди.

Вдобавок к данному изменению задумайтесь вот над чем: каким будет цвет фона после завершения анимации? Поскольку мы не проводим других изменений, он останется черным. Этого как раз хотелось бы избежать, так что следует восстановить расположение и фон Луны. Нам сейчас нужны последний параметр `animate()` и функция обратного вызова. Таким образом можно восстановить первоначальный (белый) цвет фона изображения внутри данной функции обратного вызова:

```
$moonImage.animate({top: '-=256'}, 2500, function() {
  $(this).css({backgroundColor: 'white'});
});
```

Вы можете заменить white на #FFFFFF или любой эквивалентный вариант

Чтобы увидеть этот пример в действии, откройте страницу файла `chapter-8/revolutions.2.html`, предоставленного с книгой, в вашем браузере и нажмите кнопку Animate (Анимировать).

Очереди функций удобно применять, когда возникает необходимость выполнять их поочередно, без использования вложенных функций или асинхронных функций обратного вызова. На сегодняшний день существуют куда более продвинутые способы и методы для решения этих задач (обычно именуемых *callback hell*), которые мы обсудим в главе 13, когда будем говорить об объектах `Deferred` и `Promise`.

8.6. Резюме

В данной главе мы познакомились с анимационными эффектами, входящими в состав jQuery, а также с методом `animate()`, позволяющим создавать собственные анимационные эффекты.

Методы `show()` и `hide()` при вызове без параметров отображают и скрывают элементы сразу же, без анимационных эффектов. С помощью анимационных версий этих методов можно отображать и скрывать элементы, передавая им параметры, управляющие скоростью анимации, а также функцию обратного вызова, которая запускается после завершения анимационного эффекта. Метод `toggle()` переключает визуальное состояние элемента между видимым и невидимым.

Другая группа методов, `fadeOut()` и `fadeIn()`, также скрывает и отображает элементы, изменяя степень непрозрачности элементов во время их удаления или появления на экране. Похожим на методы `hide()` и `show()` является `fadeToggle()`, который содержит эффект тени. Следующий метод этой группы `fadeTo()` изменяет непрозрачность элементов набора без удаления их из отображения страницы.

Последняя группа встроенных анимационных эффектов удаляет или отображает элементы набора, изменяя их высоту: `slideUp()`, `slideDown()` и `slideToggle()`.

Эти методы познакомили вас с понятием плавности (смягчения). Данный термин используется для описания способа, изменяющего скорость кадров анимации. Ядро jQuery предоставляет только две смягчающие функции для поддержания библиотеки, но можно дополнить jQuery другими библиотеками или плагинами, в первую очередь jQuery Easing и jQuery UI, чтобы получить целый ряд новых смягчающих функций.

Кроме того, jQuery позволяет создавать собственные анимационные эффекты, используя метод `animate()`. С его помощью можно анимировать любые свойства стилей, которые принимают числовые значения, чаще всего это непрозрачность, расположение и размеры элементов.

Вы также изучили, как jQuery размещает анимационные эффекты в очереди для их последовательного выполнения и как можно применять методы добавления в очередь, чтобы добавить функции в очередь анимационных эффектов или в ваши собственные очереди.

Когда мы говорим о написании ваших собственных эффектов, мы имеем в виду, что вы пишете код для них как встроенный в сценарий JavaScript. Но было бы намного удобнее и полезнее оформить пакет jQuery с анимационными эффектами. Как это делается, мы покажем в главе 12, после чего предлагаем вам вернуться сюда и в качестве прекрасного дополнительного упражнения создать пакет с собственными анимационными эффектами из данной главы и другими, которые вы сможете придумать.

Но прежде, чем приступать к созданию собственных плагинов для jQuery, познакомимся с некоторыми утилитами и флагами, предоставляемыми библиотекой.

9 За пределы DOM с помощью вспомогательных функций jQuery

В этой главе:

- свойства jQuery;
- избегание конфликтов между jQuery и другими библиотеками;
- функции манипулирования массивами;
- расширение и объединение объектов;
- анализ различных форматов;
- динамическая загрузка новых сценариев.

До настоящего момента достаточно много глав было посвящено исследованию методов jQuery, воздействующих на набор элементов DOM, выбранных с помощью функции `$()`. Возможно, вы помните, что еще в главе 1 было введено такое понятие, как вспомогательные функции — принадлежащие пространству имен `jQuery/$`, но не воздействующие на объект jQuery. Их можно было бы считать глобальными, за исключением того, что они определены в экземпляре объекта `$`, а не `window` и не входят в пределы глобального пространства имен. Как правило, эти функции либо воздействуют на объекты JavaScript, которые *не* являются элементами DOM, либо выполняют ряд других операций, не относящихся к объектам (запросы Ajax или разбор строк XML).

Помимо функций, в jQuery есть несколько свойств (иногда их называют *флагами*), которые также объявлены в пространстве имен `jQuery/$`. Часть свойств предназначена только для внутреннего пользования, но поскольку они описаны на сайте jQuery API и отдельные плагины работают с ними, мы подумали, что о них стоит рассказать самым любознательным из вас.

Вы можете удивиться, почему мы до сих пор откладывали знакомство с этими функциями и свойствами. У нас были две основные причины.

- Мы хотели научить вас мыслить в терминах использования методов jQuery, а не показывать, как применять операции низкого уровня.

- Поскольку методы обеспечивают большую часть наших потребностей при манипулировании элементами DOM на страницах, эти низкоуровневые функции зачастую наиболее полезны при написании самих методов (и других плагинов), а не для создания программного кода уровня страницы. (О том, как писать собственные плагины jQuery, мы поговорим в главе 12.)

В данной главе мы не будем обсуждать вспомогательные функции, которые обеспечивают поддержку Ajax, поскольку посвятили им целую главу 10. Начнем с уже упомянутых флагов.

9.1. Применение флагов jQuery

Библиотека jQuery предоставляет авторам страниц некоторый функционал, доступный не с помощью методов или функций, а в виде переменных, определенных в пространстве имен `$`. В прошлом некоторые авторы плагинов jQuery полагались на этот функционал при разрабке плагинов. Но, как нам станет известно уже через несколько страниц, часть свойств устарела и не рекомендуется их использовать.

jQuery 3: удаленные свойства

jQuery 3 избавляется от уже устаревших свойств `context` (https://api.jquery.com/context/), `support` (https://api.jquery.com/jQuery.support/) и `selector` (https://api.jquery.com/selector/). Если вы все еще используете их в своем проекте или применяете плагин, который на них опирается, то обновление до jQuery 3 нарушит работу вашего кода.

Доступны следующие свойства jQuery:

- `$.fx.off` — разрешает или запрещает воспроизведение анимационных эффектов;
- `$.fx.interval` — изменяет скорость запуска анимации;
- `$.support` — детализирует поддерживаемые свойства (только для внутреннего пользования).

Для любознательных: свойство $.browser

До версии 1.9 jQuery предлагала набор свойств, которые разработчики применяли для разветвления кода (совершения различных операций на основе значения данного свойства). Свойства устанавливались во время загрузки библиотеки, что делало их доступными даже до выполнения каких-либо готовых обработчиков. Они были определены как свойства `$.browser`. Флаги доступны для `msie`, `mozilla`, `webkit`, `safari` и `opera`.

Изучим эти свойства, но начнем с того, что посмотрим, как jQuery позволит отключить анимацию.

9.1.1. Запрет воспроизведения анимационных эффектов

Иногда в зависимости от ряда условий нужно запретить воспроизведение различных анимационных эффектов на странице. Возможно, вы определили, что платформа или устройство вряд ли будут их поддерживать, либо по каким-либо другим причинам. Например, можно полностью отключить анимации на медленных мобильных устройствах, так как воспроизведение создаст неудобства для пользователя. Когда вы обнаружите, что находитесь в неблагоприятной для анимаций среде или попросту в них не нуждаетесь, можете просто записать в переменную `$.fx.off` значение `true`.

Данный флаг *не* отменяет эффекты, используемые на странице, — он просто запрещает анимированное воспроизведение этих эффектов. Например, эффект растворения элемента будет воспроизводиться как эффект немедленного его скрытия, без анимации. Точно так же вызов метода `animate()` будет сразу же устанавливать указанные свойства стилей CSS в конечные значения.

Флаг `$.fx.off` вместе с `$.fx.interval`, который будет обсуждаться в следующем подразделе, являются флагами для чтения и записи. Это значит, что можно как прочитать их, так и задать значения. Флаг `$.support`, напротив, предназначается только для чтения.

9.1.2. Изменение скорости воспроизведения анимаций

Как и любая библиотека, выполняющая анимации, jQuery поддерживает свойство `$.fx.interval` для установки скорости их запуска. Как вы знаете, анимация состоит из последовательности шагов, которые, если смотреть в целом, создают эффект. Приведем такое сравнение: фильм — это серия кадров, показанных в обычном темпе. Используя свойство `$.fx.interval`, можно настроить темп прохождения шагов.

Данное свойство предназначено для чтения и записи, его значение выражается в миллисекундах, по умолчанию значение установлено равным `13`. Такая цифра вовсе не случайна, ее выбор основан на компромиссе между плавной анимацией и отсутствием излишней нагрузки процессора.

В сценариях, насыщенных анимациями, стоит попробовать немного снизить данное значение для создания еще более плавной анимации. Это можно сделать, установив значение `$.fx.interval` меньше `13`. Изменение скорости может привести к улучшению качества воспроизведения эффектов в некоторых браузерах, но не будет оправданным для медленных устройств или для движков не очень быстрых браузеров (например, старых версий Internet Explorer). В этих браузерах можно не только не получить очевидных преимуществ, но и заметить ухудшение производительности из-за перегрузки процессора.

Итак, представьте, что вы хотите установить значение `$.fx.interval` равным `10`. Следует написать:

```
$.fx.interval = 10;
```

Точно так же, как уменьшили, можно и увеличить значение. Данное изменение может быть весьма кстати, если вы работаете на веб-странице, которая нагружает процессор. Это возможно, например, если одновременно работает несколько анимаций или страница совершает нагружающие процессор операции во время работы ряда анимаций. Поскольку анимации не так важны, как выполнение задач, вы можете решить замедлить скорость запуска анимаций. Например, установить значение `100`, чтобы обновление происходило десять раз в секунду:

```
$.fx.interval = 100;
```

Увидеть разницу в запуске анимаций с разным значением `$.fx.interval` можно в демонстрационном примере, доступном в файле `chapter-9/$.fx.interval.html`, предоставленном с книгой, а также в JS Bin (http://jsbin.com/tevoy/edit?html,css,js,output).

Теперь, когда мы разобрались со свойствами, имеющими дело с анимацией, кратко рассмотрим свойства, дающие информацию о среде, предоставленной агентом пользователя.

9.1.3. Свойство $.support

У jQuery есть свойство `$.support`, которое хранит результат тестирования функций, что представляет интерес для библиотеки. Оно позволяет jQuery узнать, какие функции поддерживаются в браузере, а какие — нет, и действовать соответствующим образом. Это свойство предназначено только для внутреннего пользования в jQuery и не рекомендуется в jQuery версии позже 1.9, поэтому мы настоятельно рекомендуем вам воздержаться от него. Некоторыми примерами его применения в настоящее время или в прошлом являются `boxModels`, `cssFloat`, `html5Clone`, `cors` и `opacity`.

Полагаться на объект `$.support` не стоит, так как его свойства могут быть удалены в любое время без предварительного уведомления, когда они больше не будут нужны. Свойства удаляются, чтобы повысить производительность загрузки библиотеки — таким образом отпадает необходимость выполнять ряд тестов в браузере. Мы решили упомянуть об этом, потому что можно использовать старый, но хороший плагин, который опирается на свойство `$.support`.

Данный объект также является одним из немногих случаев, когда наблюдается разница между ветвями jQuery; ветвь 1.x имеет больше свойств, чем ветвь 2.x, а Compat 3.x имеет больше свойств, чем 3.x. Например, 1.x содержит `ownLast`, `inline-BlockNeedsLayout` и `deleteExpando`, а 2.x — нет.

Выявление функции с помощью Modernizr

Если ваш проект должен работать иначе, исходя из особенностей, поддерживаемых данным браузером, то вам следует задействовать подход, известный как *выявление функции*. Вместо того чтобы определять браузер, используемый пользователем, а затем

пытаться определить, поддерживаются ли функции (подход, известный как *выявление браузера*), выявление функции позволяет обнаружить наличие функций напрямую.

Для проекта, в котором вам необходимо проверить нескольких функций, мы настоятельно рекомендуем воспользоваться внешней библиотекой, созданной для этой конкретной цели. Самая известная и наиболее часто применяемая библиотека — Modernizr (http://modernizr.com/). Она обнаруживает наличие встроенной поддержки для функций, вытекающих из технических характеристик HTML5 и CSS3.

Modernizr проверяет функции, реализуемые по крайней мере одним из основных браузеров (нет никакого смысла в проверке функции, которая ничем не поддерживается, верно?), а чаще двумя или более. Ниже приведен список того, что для вас делает данная библиотека:

- тестирует поддержку более чем 40 функций в течение нескольких миллисекунд;
- создает объект JavaScript (с именем `Modernizr`), содержащий информацию о результатах работы свойств булева значения;
- добавляет классы к элементу `html`, описывающие реализованные функции;
- предоставляет загрузчик сценариев, который позволяет использовать *полифиллы* для заполнения функциональности в старых браузерах;
- позволяет применять новое секционирование элементов HTML5 в более старых версиях Internet Explorer.

Что такое полифилл

Полифилл (polyfill) — фрагмент кода (или плагин), позволяющий добавить в старые браузеры поддержку возможностей, которые в современных браузерах являются встроенными. Термин был введен в 2010 году Рэми Шарпом, известным разработчиком JavaScript и основателем конференции Full Frontal. Вы можете найти более подробную информацию о том, как и почему возник этот термин, в оригинале публикации Рэми Шарпа «Что такое полифилл?» (http://remysharp.com/2010/10/08/what-isa-polyfill/).

В этом разделе мы рассмотрели последнее оставшееся свойство, так что теперь готовы двигаться дальше и начнем обсуждение вспомогательных функций jQuery.

9.2. Использование других библиотек совместно с jQuery

Определение глобальной переменной `$` обычно является самым большим камнем преткновения при использовании других библиотек на одной странице с jQuery. Как вам уже известно, jQuery применяет идентификатор `$` в качестве псевдонима имени `jQuery`, которое задействуется при каждом вызове функций jQuery. Но другие библиотеки, в первую очередь Prototype, также обращаются к имени `$`.

jQuery предоставляет вспомогательную функцию `$.noConflict()`, освобождающую *идентификатор* `$`, чтобы любая другая библиотека могла использовать его. Синтаксис этой функции выглядит следующим образом.

Синтаксис функции: $.noConflict

`$.noConflict (jqueryPropertyToo)`

Передает контроль над именем `$` другой библиотеке, что позволяет применять различные библиотеки на одних страницах с jQuery. После выполнения этой функции все функции библиотеки jQuery придется вызывать, используя идентификатор `jQuery` (свойство `jQuery` объекта `window`), а не `$`.

При необходимости данный идентификатор также можно освободить для использования в других библиотеках при передаче функции `true`.

Этот метод должен вызываться после подключения jQuery, но перед подключением других библиотек.

Параметры

`jqueryPropertyToo`	(Логический тип) Если в данном параметре передать значение `true`, то функция освободит не только идентификатор `$`, но и идентификатор `jQuery`, в противном случае он сохранится.

Возвращает

`jQuery`.

Поскольку `$` — это псевдоним для имени `jQuery`, после вызова функции `$.noConflict()` все функциональные возможности jQuery будут по-прежнему доступны, но уже исключительно с применением идентификатора `jQuery` объекта `window`. Но не волнуйтесь. Вы все еще можете сохранить ввод некоторых символов, определив новый короткий идентификатор, и возместить потерю короткого и любимого `$`, определив собственный, но неконфликтный псевдоним для `jQuery`, например:

```
var $j = jQuery;
```

Если нужно отказаться как от идентификатора `$`, так и от `jQuery`, то все еще можно использовать методы jQuery с помощью вспомогательной функции `$.noConflict()` и перевести возвращаемое значение (библиотеку jQuery) в другое глобальное свойство:

```
window.$new = $.noConflict(true);
```

После выполнения этого оператора можно обращаться к методам jQuery, применяя только что созданное свойство `$new` (например, `$new('p').find('a')`).

Вы часто можете наблюдать применение шаблона дизайна под названием *Immediately-Invoked Function Expression* (IIFE) (Выражение немедленно вызываемой функции). Он заключается в создании условий, при которых идентификатор `$` находится в области видимости для обозначения объекта `jQuery`. (Если вы никогда не слышали о данном шаблоне или необходимо освежить знания о нем, то, пожалуйста, просмотрите приложение.) Эта техника обычно используется в нескольких ситуациях: когда нужно расширить jQuery (в частности, авторами плагина), смоде-

лировать частные переменные или работать с замыканием. Авторам плагинов она может быть полезной потому, что они не могут делать никаких предположений относительно того, почему создатели страницы вызвали `$.noConflict()`, и уж тем более не могут проигнорировать пожелания авторов страницы, вызвав их самостоятельно.

Мы подробно рассмотрим эту модель в приложении. На данный момент нужно понять следующее: если вы напишете:

```
(function($) {
    // Тело функции здесь
})(jQuery);
```

то можете смело использовать идентификатор `$` внутри тела функции, независимо от того, определен ли он уже Prototype или какой-либо другой библиотекой вне этой функции. Довольно изящно, не так ли?

Докажем то, что мы обсуждали в данном разделе, с помощью простого теста. Для первой части теста изучите HTML-документ в листинге 9.1 (доступен в `chapter-9/overriding.$.test.1.html`, предоставленном с книгой).

Листинг 9.1. Переопределение $. Тест 1

```
<!DOCTYPE html>
<html>
   <head>
      <title>Переопределение $ - Тест 1</title>
      <link rel="stylesheet" href="../css/main.css"/>
   </head>
   <body>
      <button id="button-test">Нажмите!</button>

      <script src="../js/jquery-1.11.3.min.js"></script>
      <script>
         $ = 'Привет, мир!';
         try {
            $('#button-test').on('click', function() {
               alert('$ - псевдоним для jQuery');
            });
         } catch (ex) {
            alert('$ перемещен.Значение "' + $ + '"');
         }
      </script>
   </body>
</html>
```

1 Переписывает свойство window.$ пользовательским значением (строкой)

2 Присоединяет обработчик к кнопке, показанной на странице

3 Показывает сообщение об успехе

4 Показывает сообщение о неудаче

В этом примере вы импортируете библиотеку jQuery, которая устанавливает глобальное имя `jQuery` и его псевдоним `$`. Затем переопределяете глобальное имя `$` строковой переменной 1, переопределив таким образом установку jQuery. Заменяете `$` простым значением строки только для упрощения данного примера, но его можно переопределить путем включения другой библиотеки, такой как Prototype.

Затем пытаетесь привязать обработчик к кнопке, определенной в разметке страницы ❷. Внутри обработчика вызываете функцию JavaScript `alert()`, чтобы на экране отобразилось сообщение об успешном выполнении ❸. Если что-то пойдет не так, то вы получите сообщение об ошибке ❹.

Загрузив эту страницу в браузере, вы увидите на экране сообщение об ошибке, как показано на рис. 9.1. Причина ошибки — переопределение значения идентификатора `$`. Кроме того, нажатие кнопки не влечет никаких действий, так как `$` был переназначен перед определением щелчка обработчика.

Рис. 9.1. Страница показывает, что $ был перемещен. Значение — «Привет, мир!», поскольку переопределение было осуществлено

А сейчас немного изменим этот пример, чтобы спокойно использовать `$`. Следующий код показывает только часть кода в пределах блока `try`, который был изменен. Для удобства изменения выделены полужирным шрифтом. Полный код примера можно найти в файле `chapter-9/overriding.$.test.2.html`, предоставленном с книгой.

```
try {
  (function($) {
    $('#button-test').on('click', function() {
      alert('$ - псевдоним для jQuery');
    });
  })(jQuery);
} catch (ex) {
```

Единственное изменение состоит вот в чем: нужно обернуть оператор, к которому прикреплен обработчик с IIFE, и передать ему свойство `window.jQuery`. При загрузке этой измененной версии и нажатии кнопки вы увидите, что теперь она работает и отображается другое сообщение, как показано на рис. 9.2.

Изменение кода, как показано в предыдущем примере, позволяет использовать ярлык `$` в качестве псевдонима для `jQuery`, сохраняя оригинальное, глобальное значение `$` (в данном случае это строка «Привет, мир!»). Теперь посмотрим, как можно восстановить значение `$` в целях сохранения всего, что было раньше, включая библиотеку jQuery, с помощью метода `$.noConflict()`. Типичная ситуация, в которой вам понадобится использовать данный метод, показана в листинге 9.2 и доступна в файле `chapter-9/$.noConflict.html`, предоставленном с книгой, и в JS Bin (http://jsbin.com/bolok/edit?html,console).

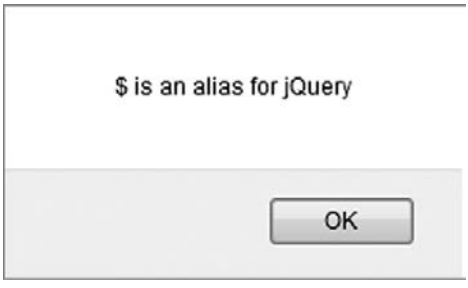


Рис. 9.2. Сообщение об успешной установке внутри обработчика

Листинг 9.2. Использование метода $.noConflict()

```
<!DOCTYPE html>
<html>
   <head>
      <title>Использование $.noConflict()</title>
      <link rel="stylesheet" href="../css/main.css"/>
   </head>
   <body>
      <script>
         window.$ = {
            customLog: function(message) {
               console.log('Функция говорит: ' + message);
            }
         };
      </script>
      <script src="../js/jquery-1.11.3.min.js"></script>
      <script>
         console.log('customLog: ' + ($.customLog === undefined));
         $.noConflict();
         console.log('customLog: ' + ($.customLog === undefined));
         $.customLog('Старое значение восстановлено!');
      </script>
    </body>
</html>
```

2 Импортирует библиотеку jQuery

1 Создает фиктивную библиотеку, назначенную window.$

Выводит в консоли сообщение посредством $.customLog() 6

Снова выводит в консоли результат, проверяющий, определена ли $.customLog 5

4 Восстанавливает старое значение $

Выводит в консоли результат, который проверяет, определена ли $.customLog 3

В этом примере вы создаете фиктивную пользовательскую библиотеку ❶, у которой есть только один метод: `customLog()`. Вы можете импортировать любую библиотеку, использующую ярлык `$`, например Prototype, вместо фиктивной, но для простоты примера мы не будем так делать. Затем импортируете библиотеку jQuery ❷, заменяющую библиотеку, которая хранится в `$`. Далее следует убедиться, что фиктивная библиотека была заменена, проверив, что метод `customLog()` не определен (у jQuery нет такого метода) ❸.

В следующей строке кода вы восстанавливаете значение `$` на существующее до импорта библиотеки jQuery, вызвав функцию jQuery `noConflict()` ❹. Наконец, проверяете еще раз, можно ли получить доступ к методу `customLog()` через ярлык `$` ❺, и выводите сообщение на консоль с помощью этого метода ❻.

Теперь, когда вы увидели, как можно использовать jQuery, чтобы избежать каких-либо конфликтов с другими библиотеками, пришло время изучить иные вспомогательные функции jQuery.

9.3. Управление объектами и коллекциями JavaScript

Большинство функций jQuery реализованы как вспомогательные для работы с объектами JavaScript, не являющимися элементами DOM. Обычно все, что предназначено для работы с DOM, реализуется как метод jQuery. Хотя ряд этих функций и позволяет работать с элементами DOM, которые сами *являются* объектами JavaScript, в целом работа с DOM — не главная область применения вспомогательных функций.

Данные функции реализуют целый спектр операций: от простых манипуляций со строками и проверки типов до сложной фильтрации коллекций, сериализации значений форм и даже реализации своего рода наследования объектов путем слияния свойств. Начнем с основ.

9.3.1. Усечение строк

Это трудно объяснить, но до ECMAScript 5 у объекта `String` не было метода удаления пробельных символов в начале и конце строки. Такие базовые функциональные возможности являются обязательной частью класса `String` в большинстве других языков, но по каким-то таинственным причинам в JavaScript не было этого полезного свойства в предыдущих версиях. Таким образом, вы не можете использовать данную функцию в версиях Internet Explorer, предшествующих 9.

Во многих приложениях JavaScript усечение строки очень распространено; хорошим примером может служить проверка данных формы. Поскольку пробелы невидимы на экране, пользователи могут легко ошибиться и ввести дополнительные символы до или после основных записей в текстовых полях или текстовых

областях. Во время проверки желательно молча удалить такие пробелы, а не предупреждать пользователя об их наличии, ведь он и так их не замечает.

Чтобы помочь справиться с этой задачей в старых браузерах, а также вообще помочь людям с любыми браузерами до введения собственного метода JavaScript (`String.prototype.trim()`), команда jQuery включила вспомогательную функцию `$.trim()`. Негласно, с целью улучшения производительности, она использует нативный метод `String.prototype.trim()` там, где он поддерживается.

Метод `$.trim()` определяется следующим образом.

Синтаксис функции: $.trim

`$.trim(value)`

Удаляет все начальные или конечные пробельные символы из переданной строки и возвращает результат.

Пробельными данная функция считает любые символы, соответствующие в JavaScript регулярному выражению `\s`, то есть не только символ пробела, но и символы перевода страницы, новой строки, возврата, табуляции и вертикальной табуляция, а также символ Юникода `\u00A0`.

Параметры

`value` (Строка) Строковое значение для усечения. Это исходное значение не изменяется.

Возвращает

Усеченную строку.

Небольшой пример усечения значения в текстовом поле с помощью данной функции:

```
var trimmedString = $.trim($('#some-field').val());
```

Имейте в виду: функция `$.trim()` преобразует параметр, который вы передаете, в тип его эквивалента `String`, так что если ошибочно передадите ей объект, то получите строку `"[object Object]"`.

А сейчас посмотрим на ряд функций, которые работают с массивами и другими объектами.

9.3.2. Итерации по свойствам и элементам коллекций

Часто при наличии нескалярных величин в составе других компонентов требуется итерировать по элементам. Вне зависимости от того, будет элемент контейнера массивом JavaScript (содержащим любое количество других значений JavaScript, включая другие массивы) или экземпляром объекта JavaScript (содержащим свойства), язык JavaScript позволяет обойти их все в цикле. Обход элементов массивов выполняется с помощью цикла `for`; обход свойств объектов — цикла `for...in` (доступны и другие конструкции, но в данный момент не будем обращать на них внимания).

Приведем примеры для всех случаев:

```
var anArray = ['one', 'two', 'three'];
for (var i = 0; i < anArray.length; i++) {
  // Здесь будет что-то делаться с anArray[i]
}
var anObject = {one: 1, two: 2, three: 3};
for (var prop in anObject) {
  // Здесь будет что-то делаться с prop
}
```

Довольно просто, но кому-то данный синтаксис может показаться слишком многословным и сложным — циклы `for` часто критикуют за это.

Несколько лет назад в JavaScript был добавлен метод объекта `Array` под названием `forEach()`. К сожалению, будучи более поздним дополнением, он не поддерживается в некоторых браузерах, в первую очередь Internet Explorer до версии 9. Вдобавок из-за принадлежности объекту `Array` данный метод не может быть использован для итерации по другим типам объектов. Но в этом случае jQuery спешит на помощь!

Как вы знаете, библиотека предоставляет метод `each()`, который позволяет легко обойти все элементы в коллекции jQuery, не используя сложный синтаксис оператора `for`. Для массивов, массивоподобных объектов и объектов у jQuery есть аналогичная вспомогательная функция `$.each()`.

Действительно, очень удобно, когда для организации итераций по элементам массива или свойствам объекта можно применять один и тот же синтаксис. Более того, его также можно задействовать в Internet Explorer 6–8. Синтаксис функции выглядит следующим образом.

Синтаксис функции: $.each

`$.each(collection, callback)`

Обобщенная функция итератора, которая может быть использована для обхода как объектов, так и массивов. Массивы и массивоподобные объекты со свойством длины (например, объекты функции `arguments`) итерируются по числовым индексам от 0 до длины – 1. Другие объекты итерируются через названные свойства.

Параметры

`collection` (Массив|Объект) Массив (или массивоподобный объект), по элементам которого должна пройти итерация, либо объект, по свойствам которого она должна пройти.

`callback` (Функция) Функция, которая будет вызываться для каждого элемента в коллекции. Если коллекция является массивом (либо массивоподобным объектом), то данная функция вызывается для каждого элемента массива, а если объектом, то для каждого свойства объекта. Первый параметр функции `callback` — индекс элемента массива или имя свойства объекта. Второй параметр — значение элемента массива или свойства объекта.

Контекст функции (`this`) вызова содержит значение, передающееся во втором параметре.

Возвращает

Ту же переданную коллекцию.

Этот унифицированный синтаксис подходит для итерации по массивам, массивоподобным объектам и объектам, с тем же форматом вызова функции. Перепишем предыдущий пример с применением данной функции:

```
var anArray = ['one', 'two', 'three'];
$.each(anArray, function(i, value) {
  // Здесь что-то делается
});
var anObject = {one:1, two:2, three:3};
$.each(anObject, function(name, value) {
  // Здесь что-то делается
});
```

Применение `$.each()` со встроенной функцией — это хорошо. Однако она дает возможность легко написать повторно используемые функции итератора или вынести ее за скобки тела цикла в другую функцию, чтобы прояснить код, как показано в следующем примере:

```
$.each(anArray,someComplexFunction);
```

Обратите внимание: при выполнении итерации по коллекции можно выйти из цикла, возвращая значение `false` из функции-итератора. И наоборот, возврат *truthy*-значения (значения, вычисляемого `true`) аналогичен использованию `continue`, что означает немедленную остановку функции и совершение следующей итерации.

jQuery 3: добавленные функции

jQuery 3 предоставляет возможность для итерации по элементам DOM коллекции jQuery с помощью цикла `for-of`, что является частью спецификации ECMAScript 6. Благодаря этой особенности теперь можно написать код наподобие следующего:

```
var $divs = $('div');
for (var element of $divs) {
  // Здесь что-то делается с элементом
}
```

Пожалуйста, обратите внимание: мы не ставим знак доллара перед переменной `element`, чтобы подчеркнуть, что ее значение будет принадлежать элементу DOM, а не коллекции jQuery.

ПРИМЕЧАНИЕ

Применение функции $.each() может быть удобным с точки зрения синтаксиса, но это, как правило, медленнее (чуть-чуть), чем использование старомодного цикла for. Выбор за вами.

Иногда обход массива нужен, чтобы выбрать элементы, которые составят новый массив. Конечно, это легко можно сделать с помощью функции `$.each()`, но посмотрим, насколько проще то же самое возможно с помощью других средств jQuery.

9.3.3. Фильтрация массивов

Приложениям, работающим с большими количествами данных, часто требуется обойти массивы, чтобы найти элементы, соответствующие определенным критериям. Вы можете, например, отфильтровать элементы, значения которых находятся выше или ниже определенного порога или, вероятно, соответствуют определенному шаблону. Для выполнения любой подобной фильтрации jQuery предоставляет вспомогательную функцию `$.grep()`.

Имя данной функции может навести на такую мысль: функция применяет те же регулярные выражения, что и ее тезка — команда UNIX `grep`. Но критерий фильтрации, который использует вспомогательная функция `$.grep()`, не является регулярным выражением. Фильтрацию проводит функция обратного вызова, предоставляемая вызывающей программой, которая определяет критерии для выявления значений, включаемых или исключаемых из полученного набора значений. Ничто не мешает данной функции задействовать регулярные выражения для выполнения своей задачи, но автоматически это не делается.

Синтаксис этой функции представлен ниже.

Синтаксис функции: $.grep

`$.grep (array, callback[, invert])`

Выполняет цикл по элементам массива, вызывая функцию обратного вызова для каждого значения. Возвращаемое значение функции обратного вызова определяет, должно ли войти это значение в новый массив, который возвращается как значение функции `$.grep()`. Если параметр `invert` опущен или имеет значение `false` и возвращаемое значение функции обратного вызова равно `true`, то данные включаются в новый массив. Если параметр `invert` имеет значение `true` и возвращаемое значение функции обратного вызова равно `false`, то данные также включаются в новый массив. Исходный массив не изменяется.

Параметры

`array` (Массив) Массив, элементы которого проверяются на возможность включения в коллекцию. При выполнении функции не изменяется.

`callback` (Функция|Строка) Функция, возвращаемое значение которой определяет, должно ли текущее значение войти в коллекцию. Она получает два параметра: текущее значение данной итерации и индекс этого значения в пределах исходного массива. Возвращаемое значение `true` приводит к включению данных при условии, что значение параметра `invert` не равно `true`, в противном случае результат меняется на противоположный.

`invert` (Логический тип) Если определен как `true`, то инвертирует нормальное действие функции.

Возвращает

Массив отобранных значений.

Предположим, мы хотим отфильтровать массив, выбрав все значения больше 100. Это можно сделать с помощью такой инструкции:

```
var bigNumbers = $.grep(originalArray, function(value) {
                    return value > 100;
                });
```

Функция обратного вызова, которая передается в `$.grep()`, может выполнять любые определенные вами действия, чтобы выяснить, какие значения должны быть включены. Решение может быть простым или сложным, в зависимости от того, что вам нужно.

Хотя функция `$.grep()` и не использует регулярные выражения напрямую (невзирая на свое название), регулярные выражения JavaScript, применяемые в функциях обратного вызова, позволяют успешно решать вопрос о включении или исключении значений из этого массива. Рассмотрим пример, когда в массиве требуется идентифицировать значения, не соответствующие шаблону почтового индекса Соединенных Штатов (также известного как ZIP codes).

Почтовый индекс в США состоит из пяти десятичных цифр, за которыми могут следовать символ дефиса и еще четыре десятичные цифры. Такая комбинация задается регулярным выражением вида `/^\d{5}(-\d{4})?$/`. Его можно применить для фильтрации исходного массива и извлечь ошибочные записи:

```
var badZips = $.grep(
                  originalArray,
                  function(value) {
                    return value.match(/^\d{5}(-\d{4})?$/) !== null;
                  },
                  true
);
```

Примечательно, что в этом примере метод `match()` класса `String` позволяет определить, соответствует ли значение шаблону; в параметре `invert` функции `$.grep()` передается значение `true` для исключения всех значений, соответствующих шаблону.

Выборка подмножеств данных — не единственное, что можно проделать с ними. Рассмотрим еще одну вспомогательную функцию jQuery.

9.3.4. Преобразование массивов

Данные не всегда представлены в нужном нам формате. Еще одна операция, которая часто выполняется в веб-приложениях, ориентированных на работу с данными, — *преобразование* одного набора значений в другой. Написать цикл `for` для создания одного массива из другого достаточно просто, но jQuery упрощает эту операцию с помощью вспомогательной функции `$.map()`.

Рассмотрим простейший пример, который демонстрирует функцию `$.map()` в действии.

```
var oneBased = $.map(
    [0, 1, 2, 3, 4],
    function(value) {
      return value + 1;
    }
);
```

Эта инструкция преобразует переданный массив следующим образом:

```
[1, 2, 3, 4, 5]
```

Синтаксис функции: $.map

`$.map(collection, callback)`

Итерирует по элементам массива или объекта, вызывая функцию обратного вызова для каждого элемента массива и собирая возвращаемые значения функции в новый массив.

Параметры

`collection` (Массив|Объект) Массив или объект, значения которых должны быть преобразованы в значения в новом массиве.

`callback` (Функция) Функция, возвращаемое значение которой является преобразованным значением элемента нового массива, возвращаемого функцией `$.map()`.

Этой функции передаются два параметра: текущее значение и индекс элемента в исходном массиве. Если передан объект, то вторым аргументом является имя свойства текущего значения.

Возвращает

Массив отобранных значений.

Еще один важный момент, на который следует обратить внимание: если функция возвращает значение `null` или `undefined`, то результат не включается в новый массив. В таких случаях в новом массиве будет меньше элементов, чем в исходном, и однозначное соответствие между ними будет утрачено.

Рассмотрим чуть более сложный пример. Допустим, у вас есть массив строк, возможно собранных из полей формы, предположительно являющихся числовыми значениями, и требуется преобразовать этот массив строк в соответствующий ему массив `Number`. Поскольку нет никакой гарантии, что строки не содержат нечисловых значений, понадобится принять меры предосторожности. Рассмотрим следующий фрагмент, который также доступен в файле `chapter-9/$.map.html`, предоставленном с книгой, и в JS Bin (http://jsbin.com/zonopor/edit?html,js,console):

```
var strings = ['1', '2', '3', '4', 'S', '6'];
var values = $.map(strings, function(value) {
  var result = new Number(value);
  return isNaN(result) ? null : result;
});
```

В начале есть массив строк, каждая из которых, как ожидается, представляет числовое значение. Но опечатка (или, возможно, ошибка пользователя) привела к тому, что вместо ожидаемого символа `5` имеется символ `S`. Наш программный код обрабатывает этот случай, проверяя экземпляр `Number`, созданный конструктором, чтобы увидеть, было ли успешным преобразование из строки в число. Если оно не удалось, то возвращаемое значение будет константой `NaN`. Но интереснее всего то, что по определению значение `NaN` не равно никакому другому значению, даже самому себе! Поэтому логическое выражение `Number.NaN == Number.NaN` даст в результате `false`!

Поскольку вы не можете использовать оператор сравнения для проверки на равенство `NaN` (что, кстати, означает *Not a Number — «не число»*), JavaScript предоставляет метод `isNaN()`, который и позволил проверить результат преобразования строки в число.

В этом примере в случае неудачи вы возвращаете `null`, гарантируя, что полученный в результате массив содержит только действительно числовые значения, а все ошибочные значения игнорируются. Если потребуется собрать все значения, то можно разрешить функции преобразования возвращать `Number.NaN` для значений, не являющихся числами.

Другая полезная особенность функции `$.map()` состоит в том, что она изящно обрабатывает случаи, когда функция преобразования возвращает массив, добавляя возвращаемое значение в массив результата. Рассмотрим следующий пример:

```
var characters = $.map(
  ['this', 'that'],
  function(value) {
    return value.split('');
  }
);
```

Эта инструкция преобразует массив строк в массив символов, из которых составлены строки. После ее выполнения значения переменной `characters` таковы:

```
['t', 'h', 'i', 's', 't', 'h', 'a', 't']
```

Данный результат получен с помощью метода JavaScript `split()`, который возвращает массив из символов строки, если в качестве разделителя была передана пустая строка. Такой массив возвращается как результат функции преобразования и затем включается в массив результата.

Но этим поддержка массивов в jQuery не ограничивается. Есть несколько второстепенных функций, которые также могут оказаться полезными.

9.3.5. Другие полезные функции для работы с массивами JavaScript

Случалось ли так, что нужно было узнать, содержит ли массив JavaScript некоторое специфическое значение и, возможно, даже определить его местоположение в массиве?

Если да, то вы по достоинству оцените функцию `$.inArray()`.

Синтаксис функции: $.inArray

`$.inArray(value, array[, fromIndex])`

Возвращает индекс первого вхождения значения `value` в массив `array`.

Параметры

`value`	(Любой) Значение, которое требуется найти в массиве.
`array`	(Массив) Массив, в котором будет производиться поиск.
`fromIndex`	(Число) Индекс массива, с которого начинается поиск. По умолчанию равняется 0.

Возвращает

Индекс первого вхождения искомого значения в массиве или `-1`, если значение не найдено.

Несложный, но показательный пример применения данной функции:

```
var index = $.inArray(2, [1, 2, 3, 4, 5]);
```

Другая полезная функция для работы с массивами создает массив JavaScript из других объектов, подобных массиву. Взгляните на следующий отрывок:

```
var images = document.getElementsByTagName('img');
```

Это заполнит переменные `image` всеми изображениями `HTMLCollection`, записанными на странице.

Довольно непросто иметь дело с подобными объектами, потому что у них недостаточно нативных методов JavaScript `Array`, таких как `sort()` и `IndexOf()`. Преобразование `HTMLCollection` (и других аналогичных объектов) в массив JavaScript существенно упрощает работу. Функция `$.makeArray` из библиотеки jQuery — как раз то, что нужно в данном случае.

Синтаксис функции: $.makeArray

`$.makeArray(object)`

Преобразует объект, подобный массиву, в массив JavaScript.

Параметры

`object` (Объект) Любой объект для преобразования в нативный `Array`.

Возвращает

Полученный в результате массив JavaScript.

Данная функция предназначена для использования в коде, который редко обращается к jQuery. Она пригодится и при обращении с объектами типа массива `arguments` в пределах функций. Представьте, что у вас есть следующие функции и вы хотите отсортировать их аргументы:

```
function foo(a, b) {
    // Аргументы сортировки здесь
}
```

Можно получить все аргументы одновременно, используя массив `arguments`. Проблема в том, что он не относится к типу `Array`, так что вы не можете написать:

```
function foo(a, b) {
    var sortedArgs = arguments.sort();
}
```

Этот код выдаст ошибку, потому что массив `arguments` не поддержвиает метод JavaScript `Array sort()`, который пригодился бы для сортировки аргументов. Здесь может помочь `$.makeArray()`. Проблему можно решить, преобразовав аргументы в реальный массив, а затем отсортировав его элементы:

```
function foo(a, b) {
    var sortedArgs = $.makeArray(arguments).sort();
}
```

После внесения этого изменения `sortedArgs` будет содержать массив с отсортированными аргументами, переданными функции `foo()`.

Допустим, сейчас у вас есть следующий оператор:

```
var arr = $.makeArray({a: 1, b: 2});
```

После его выполнения оператор `arr` будет содержать массив, состоящий из одного элемента — объекта, переданного в качестве аргумента `$.makeArray()`.

Другая редко используемая функция, которая может быть полезной в работе с массивами, созданными не средствами jQuery, называется `$.unique()`.

Синтаксис функции: $.unique

`$.unique(array)`

Получает массив элементов DOM и возвращает массив уникальных элементов из оригинального массива.

Параметры

`array` (Массив) Массив элементов DOM, который требуется исследовать.

Возвращает

Массив элементов DOM, состоящий из уникальных элементов массива, переданного функции.

Эта функция предназначена для использования с массивами элементов DOM, созданных за пределами jQuery. Хотя многие люди думают, что ее можно применять с массивами строк или чисел, мы хотим подчеркнуть: `$.unique()` работает только с массивами элементов DOM.

Прежде чем углубиться в описание следующей функции, рассмотрим пример работы `$.unique()`. Обратите внимание на разметку:

```
<div class="black">foo</div>
<div class="red">bar</div>
<div class="black">baz</div>
<div class="red">don</div>
<div class="red">wow</div>
```

Теперь представьте, что по какой-то причине вам нужно получить `<div>`, имеющие класс `black` в качестве массива элементов DOM, а затем добавить все `<div>` на странице в эту коллекцию и наконец отфильтровать ее, чтобы удалить дубликаты. Можно достичь результата, написав следующий код (который включает в себя ряд заявлений для определения разницы в числе элементов):

```
var blackDivs = $('.black').get();
console.log('Черные div-ы: ' + blackDivs.length);
var allDivs = blackDivs.concat($('div').get());
console.log('Увеличенные div-ы: ' + allDivs.length);
var uniqueDivs = $.unique(allDivs);
console.log('Уникальные div-ы: ' + uniqueDivs.length);
```

Если захотите поэкспериментировать с этим примером, то он доступен в файле `chapter-9/$.unique.html`, предоставленном с книгой, и в JS Bin (http://jsbin.com/borin/edit?html,js,console).

jQuery 3: переименованные методы

jQuery 3 переименовала вспомогательную функцию `$.unique()` в `$.uniqueSort()` для указания на то, что эта функция также может сортировать. Несмотря на данное изменение, в jQuery 3 вы можете по-прежнему вызывать `$.unique()`, которая в настоящее время является просто псевдонимом для `$.uniqueSort()`, но сейчас не рекомендуется так делать, ведь это название устарело и будет удалено в следующих версиях библиотеки.

А если необходимо объединить два массива? У jQuery есть функция для такой задачи — `$.merge`.

Синтаксис функции: $.merge

`$.merge(array1,array2)`

Добавляет элементы из второго массива в первый и возвращает результат. В ходе выполнения функции первый массив модифицируется и возвращается в качестве результата. Второй массив не изменяется.

Параметры

`array1` (Массив) Массив, в который будут добавлены элементы из второго массива.

`array2` (Массив) Массив, из которого будут добавлены элементы в первый массив.

Возвращает

Первый массив, измененный в результате операции объединения.

Обратите внимание на код:

```
var arr1 = [1, 2, 3, 4, 5];
var arr2 = [5, 6, 7, 8, 9];
var arr3 = $.merge(arr1, arr2);
```

После выполнения этого фрагмента массив `arr2` останется прежним, а массивы `arr1` и `arr3` будут содержать следующие элементы:

```
[1, 2, 3, 4, 5, 5, 6, 7, 8, 9]
```

Заметьте: есть два вхождения `5`, так как вспомогательная функция `$.merge()` не удаляет дубликаты.

Увидев, как jQuery упрощает работу с массивами, рассмотрим способ, который поможет управлять простыми объектами JavaScript.

9.3.6. Расширение объектов

Несмотря на то что часть функций JavaScript реализована как в объектно-ориентированном языке, его нельзя назвать полноценным объектно-ориентированным из-за отсутствия поддержки некоторых важных особенностей. Одной из них является

наследование (поддерживается в ECMAScript 6), когда новый класс определяется как расширение существующего класса.

Наследование в JavaScript можно имитировать, расширив объект, копируя свойства базового объекта в новый объект и затем расширяя новый объект с сохранением свойств базового объекта.

Достаточно легко написать программный код, который выполнял бы такое расширение простым копированием, но, как и во многих других случаях, у jQuery уже есть все необходимое. Библиотека предоставляет готовую вспомогательную функцию `$.extend()`. Как мы увидим в следующей главе, данную функцию можно применять не только для расширения объектов. Ее синтаксис выглядит следующим образом.

Синтаксис функции: $.extend

```
$.extend ([deep, ] target, [source1,source2,...,sourceN])
```

Расширяет объект, полученный в параметре `target`, свойствами остальных объектов, передаваемых функции. Расширенный объект также предоставляется в качестве возвращаемого значения.

Параметры

`deep`	(Логический тип) Необязательный флаг, определяющий, должно ли копирование быть полным или поверхностным. Если этот параметр опущен или имеет значение `false`, то выполняется поверхностное копирование; если `true`, то полное.
`target`	(Объект) Объект, свойства которого дополняются свойствами объектов-источников. Если этот объект является единственным предоставленным параметром, то он подразумевается как источник и объект jQuery считается целевым. Если дается более одного аргумента, то данный объект напрямую модифицируется новыми свойствами, прежде чем будет возвращен как значение функции. Все свойства, одноименные свойствам любого объекта-источника, переопределяются значениями свойств из объектов-источников.
`source1 ... sourceN`	(Объект) Один или несколько объектов, свойства которых добавляются к объекту `target`. Если объектов-источников больше одного и у них есть одноименные свойства, то объекты в конце списка имеют более высокий приоритет по сравнению с находящимися в начале списка.

Возвращает

Расширенный объект `target`.

Поведение этой функции весьма интересно благодаря ее гибкости. Почти каждый аргумент в подписи не является обязательным и позволяет изменить то, что делает данная функция. Аргументы, которые являются `null` или `undefined`, игнорируются.

Если предоставляется только один объект, то он интерпретируется не как цель, а как источник. В таком случае как цель подразумевается объект jQuery и свойства объекта объединяются в объект jQuery.

Стоит отметить: у объединенного объекта может быть меньше свойств, чем у расширенного объекта и источника суммарно, даже если они все разные. Причина заключается в том, что функция `$.extend()` игнорирует свойства, значение которых `undefined`. Посмотрим на эту функцию на примере. Вы создаете три объекта, целевой и два источника, следующим образом:

```
var target = {a: 1, b: 2, c: 3};
var source1 = {c: 4, d: 5, e: 6};
var source2 = {c: 7, e: 8, f: 9};
```

Затем проводите над этими объектами следующую операцию с помощью `$.extend()`:

```
$.extend(target, source1, source2);
```

Этот код берет содержимое объектов-источников и объединяет их в целевой объект. Чтобы проверить код, мы создали пример кода в файле `chapter-9/$.extend.test.1.html`, который его выполняет и отображает результат на странице. Загрузка этой страницы в браузере показана на рис. 9.3.

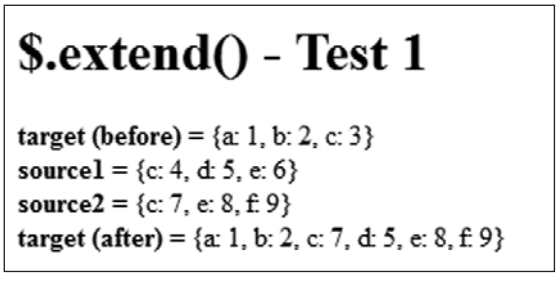


Рис. 9.3. Функция $.extend() объединяет свойства из нескольких объектов-источников без дублирования, отдавая приоритет указанным в конце списка

Как видите, все свойства объектов `source` объединены в объекте `target`. Но нужно отметить следующие важные нюансы.

- Оба объекта содержат свойство с именем `c`. Значение `c` в объекте `source1` замещает одноименное свойство в объекте `target`, которое, в свою очередь, замещено значением `c` объекта `source2`.
- И `source1`, и `source2` содержат свойство с именем `e`. Значение `e` в объекте `source2` переопределяет значение в `source1` при объединении их в `target`, что наглядно демонстрирует приоритет объектов из конца списка аргументов перед стоящими ближе к началу списка.

Прежде чем двигаться дальше, рассмотрим еще один пример ситуации, с которой вы, как и ваши дорогие авторы, сталкивались уже несколько раз. Предположим, вы хотите объединить свойства двух объектов с сохранением их обоих, а это значит,

что не хотите изменять объект `target`. Для выполнения этой операции можно передать пустой объект в качестве `target`, как показано ниже:

```
var mergedObject = $.extend({}, object1, object2);
```

Передавая пустой объект в качестве первого параметра, как `object1`, так и `object2` рассматриваются в качестве источников, таким образом, они не будут изменены.

В качестве последнего примера мы покажем результат выполнения глубокого копирования двух объектов с использованием первого параметра `$.extend()`. Предположим, у вас есть следующие объекты:

```
var target = {a: 1, b: 2};
var source1 = {b: {foo: 'bar'}, c: 3};
var source2 = {b: {hello: 'world'}, d: 4};
```

Вы оперируете с этими объектами, вызывая метод `$.extend()`:

```
$.extend(true, target, source1, source2);
```

Мы создали страницу, которая воспроизводит этот пример, так что можете поэкспериментировать с ним. Код содержится в файле `chapter-9/$.extend.test.2.html`, предоставленном с книгой. Загрузите его в своем браузере, чтобы увидеть результаты, показанные на рис. 9.4.

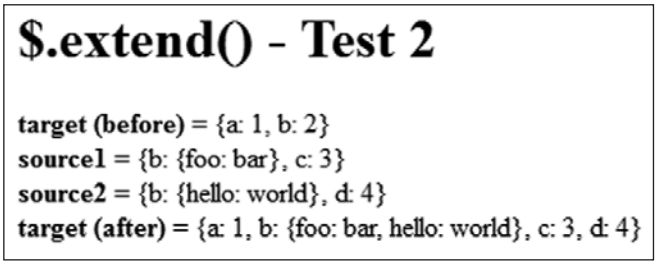


Рис. 9.4. Пример того, как функция $.extend() может объединять вложенные объекты

Данная функция очень полезна, и вы будете часто ее использовать. Теперь пора двигаться дальше, так как нас ожидает еще несколько вспомогательных функций.

9.3.7. Сериализация значений параметров

Ни для кого не секрет, что динамические высокоинтерактивные веб-приложения отправляют серверу массу запросов. Часто эти запросы являются результатом отправки форм, когда браузер формирует тело запроса, содержащее предоставленные вами параметры. Иногда придется отправлять запросы в виде адресов URL в атрибутах `href` элементов `a`. В таких случаях создание и форматирование строки

запроса, содержащей дополнительные параметры, становится полностью вашей обязанностью.

Шаблоны серверной стороны обычно содержат замечательные механизмы, помогающие конструировать корректные адреса URL. Но когда адреса создаются динамически, на стороне клиента, JavaScript в плане поддержки не слишком эффективен. Не забывайте: вы должны не только правильно расставить символы амперсанда (&) и знаки равенства (=), формирующие определения параметров в строках запросов, но и обеспечить корректное оформление имен и значений. Несмотря на то что для этих целей в JavaScript имеется удобная функция (`encodeURIComponent()`), тем не менее ответственность за форматирование строки запроса полностью ложится на ваши плечи.

И как вы, возможно, уже поняли, библиотека jQuery облегчает это бремя, предоставляя соответствующий инструмент — вспомогательную функцию `$.param()`.

Синтаксис функции: $.param

`$.param (params[, traditional])`

Сериализует переданную ей информацию в строку, пригодную для использования в качестве строки запроса. Переданное значение может быть массивом из элементов формы, объектом jQuery или объектом JavaScript. Функция формирует корректную строку запроса, кодируя в соответствии с правилами все имена и значения параметров.

Параметры

`params` (Массив|jQuery|Объект) Значение, которое будет сериализовано в строку запроса. Если функции передается массив элементов или объект jQuery, то в строку запроса будут добавлены пары «имя — значение», представляющие элементы управления форм. Если функции передается объект JavaScript, то имена и значения параметров будут сформированы из свойств этого объекта.

`traditional` (Логический тип) Необязательный флаг, который указывает, следует ли выполнять традиционную неглубокую сериализацию. Обычно влияет только на объекты, имеющие вложенные объекты. Если опущен, то по умолчанию используется значение `false`.

Возвращает

Сформированную строку запроса.

Чтобы увидеть этот метод в действии, рассмотрим следующую инструкцию:

```
$.param({
   'a thing': 'it&s=value',
   'another thing': 'another value',
   'weird characters': '!@#$%^&*()_+='
});
```

Здесь функции `$.param()` передается объект с тремя свойствами, имена и значения которых содержат символы, требующие специального кодирования для включения в строку запроса. В результате функция возвращает следующую строку:

```
a+thing=it%26s%3Dvalue&another+thing=another+value&weird+characters=!%40%23%2
     4%25%5E%26*()_%2B%3D
```

Обратите внимание, как была сформирована строка запроса и что все не алфавитно-цифровые символы в именах и в значениях свойств были закодированы. Хотя в результате получилась строка, неудобочитаемая для человека, для серверных сценариев это совершенно понятная и правильная строка!

Важный момент: если вы передаете массив элементов или объект jQuery, который содержат элементы, не являющиеся элементами форм, то в возвращаемой строке появятся такие элементы, как:

```
undefined=&
```

так как эта функция не распознает недопустимые элементы.

У вас могли появиться сомнения в полезности функции `$.param()`, так как в конечном итоге, когда значениями являются элементы формы, они будут отправлены вместе с формой браузером, который автоматически выполнит все необходимые промежуточные действия. Но не торопитесь с выводами. В главе 10 мы начнем знакомиться с технологией Ajax и увидим, что элементы формы не всегда отправляются с помощью стандартного механизма отправки формы!

Однако для нас это не будет большой проблемой: как мы увидим позже, библиотека jQuery предоставляет высокоуровневые средства (которые используют эту очень удобную функцию) для реализации подобных операций в более сложной форме.

Рассмотрим еще один пример. Представьте, что на странице есть следующая форма:

```
<form>
  <label for="name">Name:</label>
  <input id="name" name="name" value="Aurelio" />
  <label for="surname">Surname:</label>
  <input id="surname" name="surname" value="De Rosa" />
  <label for="address">Address:</label>
  <input id="address" name="address" value="Fake street 1, London, UK" />
</form>
```

Если вы вызываете вспомогательную функцию `$.param()`, передав ей объект jQuery, содержащий все элементы `input` этой формы, как показано здесь:

```
$.param($('input'));
```

то получите в результате такую строку:

```
name=Aurelio&surname=De+Rosa&address=Fake+address+123%2C+London%2C+UK
```

Данный пример прояснил, как `$.param()` работает с элементами формы. Но обсуждение этой функции еще не закончено.

Сериализация вложенных параметров. Имея многолетний опыт работы с ограничениями, которые накладывают протокол HTTP и элементы управления форм HTML, веб-разработчики привыкли представлять себе сериализованные параметры, или строки запроса, в виде простого списка пар «имя — значение» (*flat form* — «плоская форма»). Например, представьте форму, в которой вводятся имя пользователя и его адрес. Строка запроса для нее могла бы содержать параметры

с такими именами, как `firstName`, `lastName` и `city`. Сериализованная версия строки запроса может иметь следующий вид:

```
firstname=Yogi&lastname=Bear&streetaddress=123+Anywhere+Lane&city=
Austin&state=TX&postalcode=78701
```

Исходная версия этой конструкции может иметь вид:

```
{
  firstname: 'Yogi',
  lastname: 'Bear',
  streetaddress: '123 Anywhere Lane',
  city: 'Austin',
  state: 'TX',
  postalcode : '78701'
}
```

Как объект это не соответствует привычному представлению о таких данных. С точки зрения организации данных привычнее представлять указанную информацию как состоящую из двух основных элементов: имени и адреса, каждый из которых обладает своими свойствами. Например, так:

```
{
  name: {
    first: 'Yogi',
    last: 'Bear'
  },
  address: {
    street: '123 Anywhere Lane',
    city: 'Austin',
    state: 'TX',
    postalcode : '78701'
  }
}
```

Хотя такая вложенная версия элемента логически структурирована лучше, чем «плоская» версия, она не так-то просто поддается преобразованию в строку. Или все-таки просто?

Используя привычную форму записи с применением квадратных скобок, конструкцию можно выразить так:

```
name[first]=Yogi&name[last]=Bear&address[street]=123+Anywhere+
Lane&address[city]=Austin&address[state]=TX&address[postalcode]=78701
```

Здесь вложенные свойства выражаются с помощью квадратных скобок, что помогает сохранить структуру данных. Многие серверные языки, например PHP, легко справляются с аналогичными строками.

Это умное поведение, и оно не похоже на традиционный способ работы JavaScript с такими объектами. Вы могли бы ожидать нечто, подобное следующей записи:

```
name=[object+Object]&address=[object+Object]
```

которое, конечно же, не помогло бы вам.

К счастью, jQuery может иметь дело с вложенными параметрами; это позволяет решить, когда применять традиционные технологии, а когда — вот такое умное поведение. Все, что нужно сделать, — передать `true` для получения традиционного объекта и `false` (или не указывать его) — для умного поведения, как второй параметр `$.param()`.

Вы можете убедиться в наших словах, посетив лабораторную страницу `$.param()`, доступную в файле `chapter-9/lab.$.param.html`, предоставленном с книгой, а также изучив рис. 9.5.

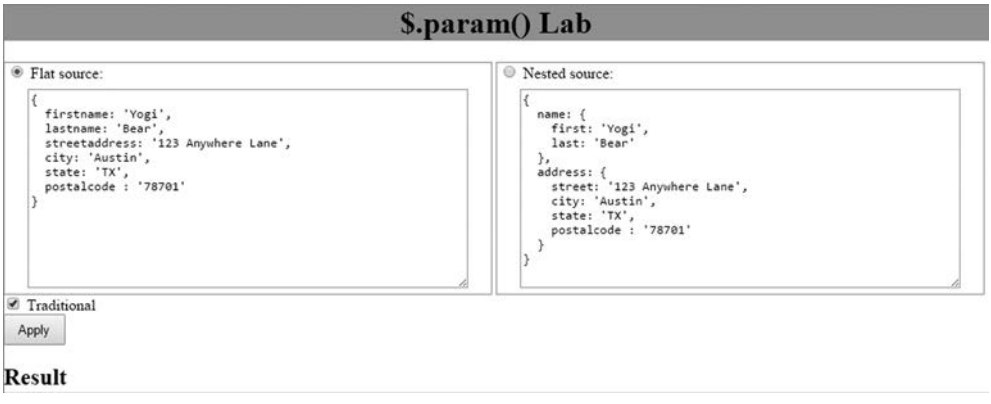


Рис. 9.5. $.param() Lab позволяет вам увидеть, как плоские и вложенные объекты сериализуются при использовании нового и традиционного алгоритмов

Эта страница позволяет увидеть, как `$.param()` будет сериализовать плоские и вложенные объекты, используя умный алгоритм, а также традиционный.

Поэкспериментируйте немного с лабораторной страницей, прежде чем перейти к следующему разделу. На ней мы установили два примера объектов, так что можно сразу же начать тестирование `$.param()`. Мы также добавили возможность редактировать их и добавлять новые свойства, так что вы можете поупражняться с различными структурами объектов, которые захотите сериализовать.

9.3.8. Проверка объектов

Вы могли обратить внимание, что многие методы jQuery и вспомогательные функции принимают достаточно гибкие списки параметров — необязательные параметры могут быть опущены без необходимости замещать их значениями `null`. Возьмем в качестве примера метод `on()`. Наиболее используемая его сигнатура такова:

```
on(eventType[, selector][, data], handler)
```

Если у вас нет селектора или не требуется передавать данные обработчику события, то можно просто вызвать метод `on()`, передав ему функцию-обработчик как второй параметр; нет никакой необходимости в заполнителях. jQuery обработает это, проверив типы аргументов, и, обнаружив, что передано всего два параметра

и во втором параметре передана функция, библиотека будет интерпретировать данную функцию как обработчик, а не как селектор или аргумент данных.

Проверка аргументов для различных типов, будь то функция или нет, становится особенно удобной, если потребуется создать собственную функцию или метод, столь же дружественный и универсальный. С этой целью jQuery предоставляет ряд вспомогательных функций проверки, которые перечислены в табл. 9.1.

Таблица 9.1. Вспомогательные функции jQuery для тестирования объектов

Функция	Описание
$.isArray(param)	Возвращает true, если param является массивом JavaScript (но не в том случае, если param является массивоподобным объектом, таким как набор jQuery), в противном случае — false
$.isEmptyObject(param)	Возвращает true, если param является объектом JavaScript без свойств, включающих любые унаследованные от prototype, в противном случае — false
$.isFunction(param)	Возвращает true, если param является функцией, в противном случае — false
$.isNumeric(param)	Возвращает true, если param является числовым значением, в противном случае — false
$.isPlainObject(param)	Возвращает true, если param является объектом JavaScript, созданным с помощью скобок {} или new Object(), в противном случае — false
$.isWindow(param)	Возвращает true, если param является объектом window, в противном случае — false
$.isXMLDoc(param)	Возвращает true, если param является документом XML или узлом в пределах документа XML, в противном случае — false

Знать эти функции — хорошо, но применять их на практике — еще лучше.

Предположим, вам хотелось бы, чтобы была функция, принимающая либо массив, либо объект в качестве первого параметра и умножающая каждый числовой элемент массива или значение объекта на заданное число, переданное в качестве второго параметра. Кроме того, вы хотите задать функцию, которая применяется после умножения элемента, в качестве третьего параметра. Второй аргумент (назовем его `factor`) и третий (назовем его `customFunction`) сделаем необязательными. Это означает следующее: можно избежать применения их обоих точно так же, как и применения одного из них. Функция должна возвращать новый объект того же типа, что и первый параметр, без изменения последнего.

На основе данного описания сигнатура функции может быть представлена так:

```
multiplier(collection[, factor][, customFunction])
```

Благодаря методам, перечисленным в табл. 9.1, вы в состоянии справиться со всеми этими случаями без особых хлопот. Реализация функции показана

в листинге 9.3. Код с некоторыми тестами также доступен в файле `chapter-9/testing.functions.html`, предоставленном с книгой, и в JS Bin (http://jsbin.com/lolub/edit?js,console).

Листинг 9.3. Тестирование вспомогательных функций

```
function multiplier(collection, factor, customFunction) {
   function calc(value) {
      return $.isFunction(factor) ?
            factor(value) :
            $.isFunction(customFunction) ?
            customFunction(value * factor) :
            value * factor;
   }
   var result = null;

   if (factor === undefined && customFunction === undefined) {
      factor = 1;
   }

   if ($.isArray(collection)) {
     result = $.map(collection, function(value) {
        if ($.isNumeric(value)) {
           return calc(value);
        }
     });
   } else if ($.isPlainObject(collection)) {
      result = {};
      for(var prop in collection) {
         if ($.isNumeric(collection[prop])) {
            result[prop] = calc(collection[prop]);
         }
      }
   }

   return result;
}
```

❶ Определяет поддерживающую функцию calc(), которая содержит ядро расчетов

❷ Если второй и третий аргументы не определены, установите factor равным 1

❸ Если имеете дело с массивом, используйте функцию $.map() для вызова calc() на значении каждого массива

❹ Если имеете дело с объектом, то используйте цикл for...in для вызова calc() на каждом значении объекта

❺ Возвращает результат расчетов

Код этого листинга весьма интересен, он использует многие функции, описанные в данной главе, в связи с чем мы опишем его более подробно.

В первой части функции вы определяете поддерживающую функцию `calc()`, которая содержит ядро расчетов ❶. Она имеет дело с переменными параметрами функции `multiplier()` и выполняет различные операции на основе значения, переданного в качестве второго и третьего аргументов. Если второй и третий аргументы `undefined` (не определены), то установите `factor` равным 1. Затем начинается реальный расчет. Если у `collection` тип массива, то функция использует вспомогательную функцию `$.map()` для вызова `calc()` на значение каждого массива,

но только если это число ❸. Если тип `collection Object`, то функция применяет цикл `for...in` для вызова `calc()` на значении каждого объекта, но только если это число ❹. Наконец, возвращается результат ❺. Тип данных результата зависит от типа данных первого аргумента (`Array` или `Object`). Если `collection` не является ни массивом, ни объектом, то возвращается `null`.

Функции, описанные в этом разделе, позволяют проверить, содержит ли переменная значение определенного типа (`Object`, `Array`, `Function` и т. д.). Но что, если информация, которую вы хотите получить, сама является типом?

Обретение типа значения. У jQuery есть одна дополнительная вспомогательная функция, которая работает с типами и называется `$.type()`.

Синтаксис данной функции представлен ниже.

Синтаксис функции: $.type

`$.type(param)`

Определяет тип значения.

Параметры

`param` (Любой) Значение для проверки.

Возвращает

Строку, описывающую тип значения.

Для демонстрации результата вызова этой функции допустим, что у вас есть следующий оператор:

```
$.type(3);
```

В данном случае вы получите:

```
"number"
```

Если такой оператор:

```
$.type([1, 2, 3]);
```

то в результате получится следующая строка:

```
"array"
```

Это важное различие по сравнению с обычным способом проверки типов в JavaScript, которое пригодится, когда мы углубимся в разработку плагинов. На самом деле, если вы выполняете тест:

```
if (typeof [1, 2, 3] === 'array')
```

то получите `false`, так как значение, возвращаемое `typeof [1, 2, 3]`, равно `"object"`.

Теперь, когда завершен обзор вспомогательных функций, которые имеют дело с типами данных, посмотрим на множество функций, позволяющих анализировать строки различных типов.

9.3.9. Анализ функций

У jQuery есть набор вспомогательных функций для синтаксического анализа нескольких форматов, начиная от JSON до XML и HTML. Современные браузеры предоставляют объект с именем `JSON`, работающий с форматом JSON. У данного объекта есть метод `parse()`, который, как и следует из названия, разбирает строку JSON. Ах да, это же современные браузеры... Как всегда, это означает, что не у всех ваших пользователей есть поддержка данной функции. К счастью, jQuery снова приходит на помощь, предоставляя вспомогательную функцию `$.parseJSON()`.

Синтаксис функции: $.parseJSON

`$.parseJSON(json)`

Анализирует переданную строку JSON, возвращая ее оценку.

Параметры

`json` (Строка) Строка JSON, которая передается.

Возвращает

Оценку строки JSON.

Вы уже несколько раз видели, что, когда браузер поддерживает собственные методы, jQuery их использует. И данный случай не является исключением. Если браузер поддерживает `JSON.parse()`, то jQuery будет его применять; в противном случае она задействует обходной прием JavaScript для выполнения оценки. Поступая так, jQuery улучшает производительность операции там, где это возможно.

Строка JSON должна быть полностью сформирована, и правила для ее сформированности значительно строже, чем нотация выражений JavaScript. Например, все имена свойств должны быть разграничены двойными кавычками, даже если они образуют действительные идентификаторы. Несоответствующая требованиям строка JSON приведет к ошибке. Посетите страницу http://www.json.org, чтобы ознакомиться с правилами формирования JSON.

JSON — не единственный формат, используемый в сети для обмена информацией. Есть еще XML. jQuery позволяет легко проанализировать строку, содержащую XML, превращая его в ее эквивалентный XML-документ с помощью вспомогательной функции `$.parseXML()`.

Синтаксис функции: $.parseXML

`$.parseXML(xml)`

Разбирает строку XML в документ XML.

Параметры

`xml` (Строка) Строка XML, которая будет разбираться.

Возвращает

Документ XML, полученный из строки.

XML-документы просто применять и перемещать, потому что их поддерживает библиотека, так что можно передать объект, возвращаемый функцией `$.parseXML()` для jQuery, а затем использовать методы, которым вы уже успели научиться. Звучит нереально? Не волнуйтесь; вскоре мы представим лабораторную страницу, где можно будет попрактиковаться с этой концепцией.

Последняя функция, которая принадлежит к данной категории, называется `$.parseHTML()`. Она задействует возможность создания нативного элемента DOM для преобразования строки, содержащей HTML-разметку для набора элементов DOM. Далее вы сможете работать на этих элементах, применяя методы jQuery.

Синтаксис функции: $.parseHTML

`$.parseHTML(html[, context][, keepScripts])`

Разбирает строку в массив узлов DOM.

Параметры

`html` (Строка) Строка HTML для разбора.

`context` (Элемент) Необязательный элемент для использования в качестве контекста, в котором будет генерироваться HTML фрагмент. Если не указан или передано `null` или `undefined`, то значением по умолчанию является (текущий) `document`.

`keepScripts` (Логический тип) В случае `true` функция удержит и включит сценарии в строку HTML. Значение по умолчанию `false`.

Возвращает

Массив элементов DOM, полученных от строки.

В описании функции мы указали, что значение по умолчанию `keepScripts` является `false`. За этим решением стоит важная проблема безопасности. При получении HTML-строки из внешнего источника, если вы включите сценарии, то позволите злоумышленникам совершить нападение на свой сайт.

Стоит отметить: в большинстве окружений, даже после избавления от элементов `script`, совершение атаки все еще возможно, поэтому вы должны убедиться, что избежите ненадежных входных сигналов через URL или cookies.

Три представленные вспомогательные функции дали возможность создать еще одну лабораторную страницу. Она доступна в файле `chapter-9/lab.parsing.html`, предоставленном с книгой, и показана на рис. 9.6.

У данной лабораторной страницы есть три предварительно собранных фрагмента кода. Первый из них представляет собой объект JSON, который можно запросить с помощью точечной нотации. Мы сделали это возможным с помощью функции `$.parseJSON()`, чтобы включить обычный текст в объект JavaScript.

Затем другая секция позволяет запрашивать XML- и HTML-код таким же образом, как это представлено на лабораторной странице главы 2. На сей раз мы будем

показывать, как много было отобрано элементов. Для данной части лабораторной страницы мы использовали `$.parseXML()` и `$.parseHTML()`.

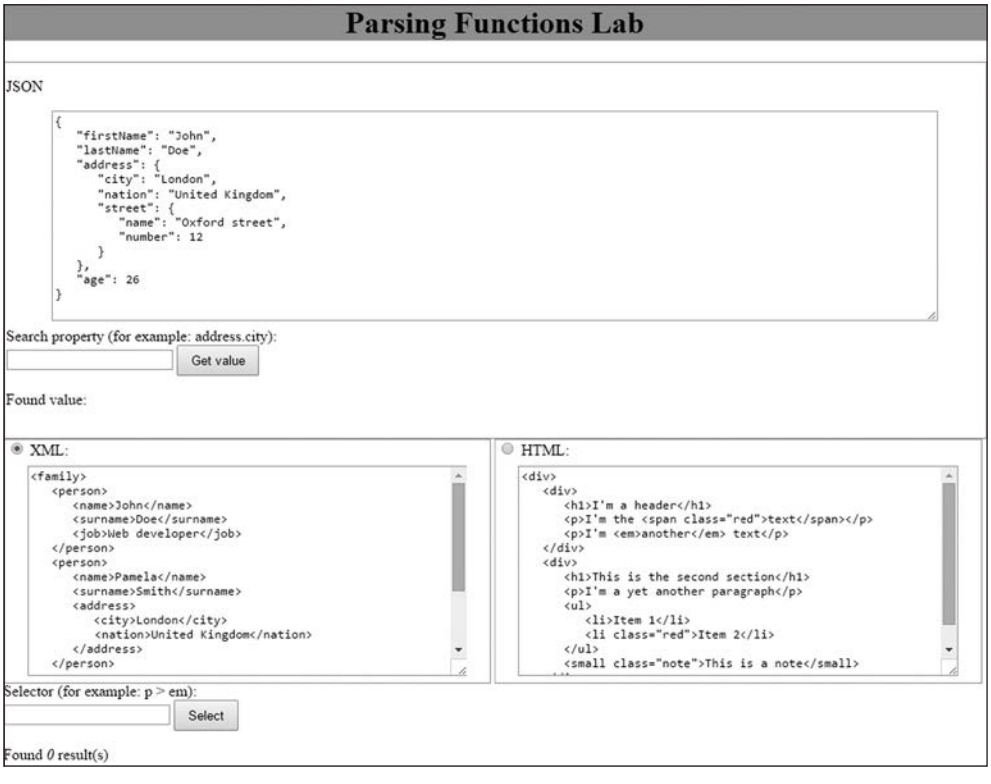


Рис. 9.6. Parsing Functions Lab позволит вам экспериментировать с функциями jQuery, которые имеют дело с тремя поддерживаемыми форматами: JSON, XML и HTML

Обратите внимание: можно изменить все три фрагмента кода, потому что они помещены в элемент `textarea`. Таким образом, вы можете проверить свой собственный объект JSON или любой XML- или HTML-код, какой хотите. Откройте лабораторную страницу и попрактикуйтесь на ней.

Разобравшись с этими методами, вы готовы узнать о других вспомогательных функциях, которые не могут быть сгруппированы в категории.

9.4. Различные вспомогательные функции

В этом разделе мы исследуем ряд вспомогательных функций; каждая из них образует собственную категорию. А начнем с функции, которая, кажется, вообще ничего не делает.

9.4.1. Пустая операция

jQuery предоставляет функцию, в буквальном смысле не делающую ничего. Ее вполне можно было бы назвать `$.uselessFunctionThatDoesNothing()` (бесполезная функция, которая ничего не делает), но такое название выглядит слишком длинным, так что она называется просто `$.noop()`. Ее синтаксис выглядит следующим образом.

Синтаксис функции: $.noop()

`$.noop()`
Ничего не делает.

Параметры

Отсутствуют.

Возвращает

`undefined`.

Хм, функция, которая ничего не принимает, ничего не делает и ничего не возвращает. Какой же в ней смысл?

Вы помните, что многие методы jQuery принимают в качестве параметров функцию обратного вызова? Функция `$.noop()` может служить значением по умолчанию, когда пользователь не указывает собственную функцию обратного вызова.

9.4.2. Проверка на вхождение

Чтобы проверить, входит ли один элемент в состав другого, jQuery предоставляет вспомогательную функцию `$.contains()`.

Синтаксис функции: $.contains

`$.contains(container,contained)`
Проверяет, входит ли элемент в состав другого в иерархии DOM.

Параметры

`container` (Элемент) Элемент DOM, который проверяется на наличие в нем другого элемента.

`contained` (Элемент) Элемент DOM, который проверяется на вхождение в состав другого элемента.

Возвращает

`true`, если элемент `contained` входит в состав элемента `container`. В противном случае — `false`.

Минуту внимания! Не кажется ли вам знакомой эта функция? Действительно, в главе 2 мы познакомились с методом `has()`, на который она так похожа.

Функция `$.contains()` часто используется внутренними механизмами jQuery, и ее очень удобно применять, когда уже имеются ссылки на элементы DOM: в этом случае не нужно тратить время на создание коллекции jQuery.

Чтобы увидеть данный метод в действии, взгляните на эту разметку:

```
<div id="wrapper">
  <p id="description">Какой-то текст</p>
</div>
<div id="empty"></div>
```

При выполнении следующих двух операторов:

```
console.log($.contains(
  document.getElementById('wrapper'),
  document.getElementById('description')
));
console.log($.contains(
  document.getElementById('empty'),
  document.getElementById('description')
));
```

получим такой результат в консоли:

```
true
false
```

Причина в том, что элемент, имеющий `description` в качестве своего идентификатора, является потомком элемента, имеющего в качестве идентификатора `wrapper`, но не элемента, идентификатор которого `empty`.

Усвоить информацию из этого и предыдущего подразделов было достаточно легко. Теперь обратим внимание на одну из наиболее оригинальных вспомогательных функций, которая позволяет получить ярко выраженный эффект от того, как вызываются слушатели событий.

9.4.3. Предварительная установка контекста функции

Как мы уже видели на протяжении всех исследований нашей библиотеки, функции и их контекст играют важную роль в программном коде, использующем jQuery. Контекст функции, на который указывает ссылка `this`, определяет, как будет вызываться функция (подробнее об этом можно прочитать в приложении). Когда необходимо вызвать какую-то определенную функцию и явно управлять ее контекстом, это можно сделать с помощью методов `call()` либо `apply()`.

Но что, если была вызвана не та функция? Что если, например, функцией является обратный вызов? В таком случае не вы ее вызываете, так что не сможете использовать упомянутые методы, чтобы воздействовать на ее контекст.

В jQuery есть вспомогательная функция, помогающая привязать к некой функции объект, который при ее вызове будет играть роль контекста. Эта вспомогательная функция называется `$.proxy()`. Синтаксис ее выглядит следующим образом.

Синтаксис функции: $.proxy

```
$.proxy(function,proxy[, argument, …, argument])
$.proxy(proxy, property[, argument, …, argument])
```

Принимает функцию и возвращает новую, которая будет иметь конкретный контекст.

Параметры

`function` (Функция) Функция, контекст которой должен быть изменен.

`proxy` (Объект) Объект, к которому должен быть установлен контекст функции (`this`).

`argument` (Любой) Аргумент, передающийся функции, на которую ссылается параметр в `function`.

`property` (Строка) Имя функции, контекст которой будет изменен (должно быть свойством объекта `proxy`).

Возвращает

Новую функцию, контекст которой привязан к объекту `proxy`.

Откройте пример в файле `chapter-9/$.proxy.html`, предоставленном с книгой, и увидите страницу, показанную на рис. 9.7.

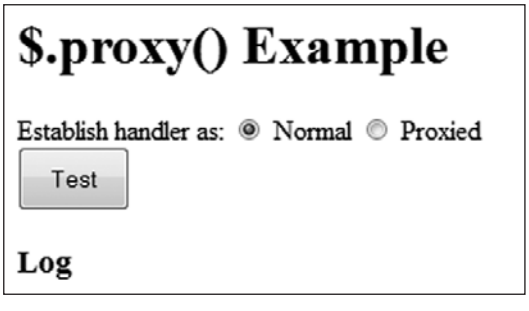


Рис. 9.7. Страница примера $.proxy поможет увидеть разницу между обычными и проксированными функциями обратного вызова

На странице примера вы найдете кнопку Test (Проверка), чьим ID является `test-button`, который нужно сохранить в переменной следующим образом:

```
var $button = $('#test-button');
```

При установке переключателя в положение Normal (Нормальный) обработчик щелчка установится на кнопке Test (Проверка) и ее контейнере:

```
$button.click(customLog);
```

Обработчик `customLog()` показывает на экране ID контекста функции (все, что ссылается на `this`):

```
function customLog() {
  $('#log').prepend(
    '<li>' + this.id + '</li>'
  );
}
```

Когда нажата кнопка, вы ожидаете вызова обработчика кнопки, чтобы получить в качестве контекста функции элемент, на котором он был установлен — кнопку Test (Проверка). Результат нажатия этой кнопки показан на рис. 9.8.

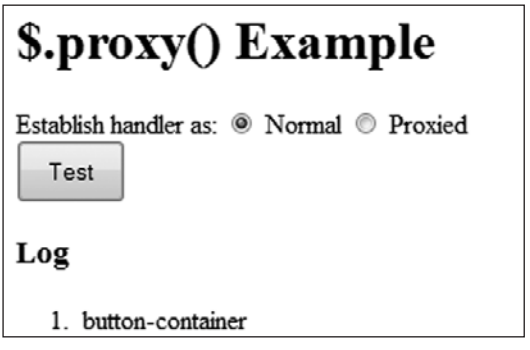


Рис. 9.8. Результат страницы примера $.proxy при использовании обычного обработчика

Но если выбрать положение переключателя Proxied (Проксированный вызов), то обработчик будет установлен, как показано ниже:

```
$button.click($.proxy(customLog, $('#control-panel').get(0)));
```

Фактически будет установлен тот же самый обработчик, что и выше, за одним исключением: на этот раз функция-обработчик передается вспомогательной функции `$.proxy()`, привязывающей объект к обработчику. Здесь мы привязали элемент со значением ID `controlPanel`. Привязываемый объект не обязательно должен быть элементом — чаще всего он и не будет им. В данном примере мы выбрали элемент лишь потому, что такой объект проще идентифицировать по значению его ID.

Если теперь нажать кнопку Test (Проверка), то картина изменится, как показано в нижней части рис. 9.9, где видно, что в качестве контекста функции был использован объект, который мы привязали к обработчику с помощью вспомогательной функции `$.proxy()`.

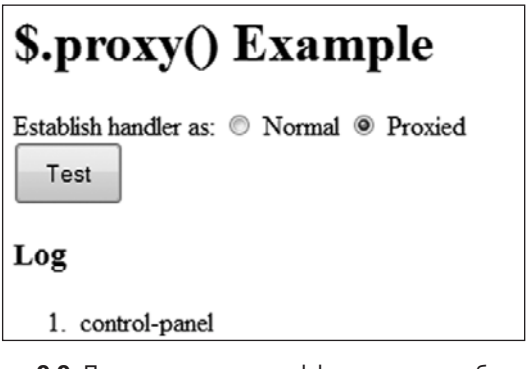


Рис. 9.9. Пример показывает эффект привязки объекта к обработчику щелчка для кнопки Test (Проверка)

Данную возможность действительно удобно использовать, когда необходимо передать ряд данных функции обратного вызова, к которым в обычной ситуации она не может иметь доступ через замыкание или через применение других механизмов.

Чаще всего необходимость в функции `$.proxy()` возникает, когда требуется установить в качестве обработчика метод объекта и передавать ему этот объект через контекст функции, как если бы метод вызывался через сам объект непосредственно. Рассмотрим следующий объект:

```
var obj = {
  id: 'obj',
  hello: function() { alert('Привет! Я ' + this.id); }
};
```

Если бы метод `hello()` вызывали через `obj.hello()`, то контекстом функции (`this`) был бы `obj`. Но если вы установите этот метод как обработчик события, например:

```
$(whatever).click(obj.hello);
```

то обнаружите, что через контекст функции методу передается текущий элемент, а не объект `obj`. И если ваш обработчик надеется получить объект `obj`, то результат его работы трудно будет предсказать. Подобную проблему можно решить с помощью вспомогательной функции `$.proxy()` и принудительно определить объект `obj` в качестве контекста функции:

```
$(whatever).click($.proxy(obj.hello, obj));
```

В качестве альтернативы с помощью второй сигнатуры функции можно достичь той же цели, написав следующее:

```
$(whatever).click($.proxy(obj, 'hello'));
```

где `obj` является объектом, которому должен быть передан контекст функции `this()` и строка `"hello"` представляет имя функции, принадлежащей `obj`, чей контекст будет изменен.

Но имейте в виду: при таком подходе обработчик всплывающего события не сможет получить информацию о текущем элементе, который при обычных условиях передается через контекст функции.

9.4.4. Выполнение выражений

Несмотря на то что применение функции `eval()` не рекомендуется многими разработчиками, тем не менее иногда бывает очень удобно ее использовать (взгляните на код страницы Parsing Functions Lab в качестве примера).

Проблема вот в чем: функция `eval()` выполняется в текущем контексте. При создании плагинов и других сценариев многократного пользования бывает необходимо, чтобы вычисления всегда выполнялись в глобальном контексте. Представляем вспомогательную функцию `$.globalEval()`.

Синтаксис функции: $.globalEval

`$.globalEval(code)`

Выполняет переданный программный код JavaScript в глобальном контексте.

Параметры

`code` (Строка) Код JavaScript, который требуется выполнить.

Возвращает

Результат выполнения программного кода на JavaScript.

Мы завершим наше исследование вспомогательных функций той из них, которую вы можете использовать, когда научитесь писать плагины jQuery, а также в других ситуациях.

9.4.5. Проброс исключений

Бывают случаи, когда нужно выбросить сообщение об ошибке в функции или плагине, автором которого вы являетесь. Например, если разработчик передает вашему плагину неожиданный параметр. Для этой цели jQuery предоставляет вспомогательную функцию `$.error()`.

Синтаксис функции: $.error

`$.error(string)`

Получает строку и выбрасывает исключение, которое ее содержит.

Параметры

`string` (Строка) Строка, определяющая сообщение для отправки.

Возвращает

`undefined`.

Этот метод существует в первую очередь для разработчиков плагинов, которые хотят его переопределить и обеспечить лучшее отображение (или больше информации) для сообщений об ошибках. Простой пример использования данной функции описан ниже.

```
function isPrime(number) {
  if (typeof number !== 'number') {
    $.error('Предоставленный аргумент не является числом');
  }
  // Остальной код здесь...
}
```

Мы вернемся к этой функции, когда будем обсуждать разработку плагинов для jQuery, но уже в другой главе.

9.5. Резюме

В данной главе мы рассмотрели функциональные возможности, которые jQuery предоставляет, помимо методов, предназначенных для выполнения операций над объектами DOM. Это разнообразные функции и несколько флагов, определенных непосредственно на уровне идентификатора `jQuery` (и его псевдонима `$`).

В первую очередь мы узнали о флагах, которые работают с анимациями. Функция `$.fx.off` позволяет полностью отключить анимации на вашем сайте так, чтобы изменения происходили немедленно. Кроме того, флаг `$.fx.interval` может изменить плавность запуска анимации.

jQuery признает, что авторам страниц иногда нужно пользоваться одновременно и другими библиотеками. Она предоставляет функцию `$.noConflict()`, которая позволяет другим библиотекам обращаться к псевдониму `$`. После ее вызова все операции JQuery должны применять идентификатор `jQuery` вместо `$`.

jQuery также предлагает набор функций, которые могут быть полезны в работе с наборами данных в массивах. Функция `$.each()` позволяет легко перемещаться по элементам в коллекциях; а `$.grep()` дает возможность создавать новые массивы путем фильтрации данных, используя какой угодно критерий фильтрации, который вы хотели бы применить. Функция `$.map()` позволяет легко применять свои собственные преобразования к источникам для получения соответствующего нового массива с преобразованными значениями.

Вы также можете обратиться к jQuery, чтобы проверить значение в массиве с помощью функции `$.inArray()`, и даже проверить, является ли само значение массивом, задействовав `$.isArray()`. Кроме того, проверить, является ли объект функцией, можно с помощью вспомогательной функции `$.isFunction()`. А проверить тип объекта можно функцией `$.type()`.

Еще один набор функций, которые предлагает библиотека, позволяет работать с различными форматами. Две такие функции, `$.parseJSON()` и `$.parseXML()`, разбирают два наиболее известных формата, используемых в Интернете для обмена информацией: JSON и XML. Еще одна, `$.parseHTML()`, позволяет разбирать HTML-разметку.

Вдобавок к информации о множестве второстепенных функций вы также узнали, как объединить объекты с помощью функции jQuery `$.extend()`. Она позволяет слить свойства любого количества исходных объектов в целевой объект. И наконец, функция `$.proxy()` позволяет изменить функцию контекста любой функции в коде.

Теперь, вооруженные сведениями о дополнительных функциональных возможностях, вы можете применять вспомогательные функции, которые предоставляет jQuery для выполнения запросов Ajax.

10 Взаимодействие с сервером по технологии Ajax

В этой главе:

- краткий обзор технологии Ajax;
- загрузка готовой HTML-разметки с сервера;
- создание запросов GET и POST;
- полное управление запросами Ajax;
- установка значений по умолчанию для свойств Ajax;
- обработка событий Ajax.

Ajax — одна из технологий, чрезвычайно изменивших Интернет. Возможность осуществления асинхронных запросов к серверу без необходимости полного обновления страниц вызвала появление целого ряда новых парадигм взаимодействия с пользователем и сделала доступными приложения на сценариях DOM.

Через несколько лет после того, как Microsoft представила Ajax, череда событий запустила эту технологию в коллективное сознание сообщества веб-разработчиков. Браузеры не от компании Microsoft внедрили стандартизированную версию в виде объекта `XMLHttpRequest` (XHR); Google начал использовать XHR; и в 2005 году Джесси Джеймс Гаррет из Adaptive Path предложил термин *Ajax* (Asynchronous JavaScript и XML).

Все как будто только и ждали, что технология получит броское название. Множество веб-разработчиков тут же заметили разрекламированную технологию Ajax, и она стала одним из основных инструментов, с помощью которых можно создавать полноценные приложения на сценариях DOM.

В этой главе мы вкратце ознакомимся с технологией Ajax (если вы уже гуру Ajax, можете сразу перейти к разделу 10.2) и с тем, как применять ее с помощью jQuery. Для начала узнаем, что представляет собой данная технология.

10.1. Знакомство с Ajax

Несмотря на то что этот раздел знакомит нас с технологией Ajax, он не является ни полным учебным пособием по Ajax, ни учебником для начинающих. Тем, кто ничего не знает о данной технологии (или, хуже того, пребывает в уверенности, что речь идет о средстве для мытья посуды или о древнегреческом мифологическом герое), рекомендуем обратиться к ресурсам, которые расскажут вам об Ajax все.

Некоторые утверждают, что термин *Ajax* применим к любым технологиям, позволяющим выполнять запросы к серверу без необходимости полностью обновлять веб-страницы (как подача запроса с помощью скрытого элемента `iframe`), но большинство связывают данный термин с использованием объекта `XMLHttpRequest` (XHR) или элемента управления ActiveX — XMLHTTP, разработанного компанией Microsoft. Диаграмма всего процесса, который мы подробно будем исследовать в этой главе, показана на рис. 10.1.

Рис. 10.1. Жизненный цикл запроса Ajax. Путь от клиента к серверу: туда и обратно

Посмотрим, как с помощью указанных объектов генерируются запросы к серверу. Начнем с создания экземпляра XHR.

10.1.1. Создание экземпляра XHR

В идеальном мире программный код, написанный для одного браузера, работал бы и в любом другом. Но мы уже знаем, что наш мир неидеален, и технология Ajax не исключение. Есть стандартный способ выполнения асинхронных запросов — с помощью объекта JavaScript XHR и собственный (уже устаревший) способ Internet Explorer — с помощью элемента управления ActiveX. В IE7 появилась обертка, имитирующая стандартный интерфейс, но IE6 все еще запрашивает расширенный код.

ПРИМЕЧАНИЕ

Реализация поддержки Ajax в библиотеке jQuery, которую мы будем рассматривать на протяжении всей главы, использует объект ActiveX, если он доступен. Это отличная новость для нас! Применяя поддержку Ajax, реализованную в jQuery, можно быть уверенными, что создатели библиотеки рассмотрели и используют наиболее удачные решения. Если вам не нужна поддержка старых версий Internet Explorer, то ваша работа станет значительно легче.

Код, выполняющий настройку, инициализацию и получение ответа на запрос, является относительно браузерно-независимым, и экземпляр XHR достаточно просто создается в любом браузере. Проблема в том, что в различных браузерах объект XHR реализован по-разному и мы вынуждены создавать его способом, нужным для текущего браузера.

Но вместо того, чтобы полагаться на определение того, какой браузер использует пользователь, для выяснения пути, которым следует идти, мы применим более предпочтительную методику, называемую *выявлением функций*, описанную в предыдущей главе. В листинге 10.1 приведен программный код, демонстрирующий типичный способ создания экземпляра XHR с применением этой техники.

Листинг 10.1. С методикой обнаружения объекта код Ajax способен выполняться в большинстве браузеров

```
var xhr;
if (window.ActiveXObject) {
  xhr = new ActiveXObject('Microsoft.XMLHTTP');

} else if (window.XMLHttpRequest) {
  xhr = new XMLHttpRequest();

} else {
  throw new Error('Ajax не поддерживается этим браузером');
}
```

Проверяется наличие ActiveX

Проверяется наличие XHR

Выбрасывается ошибка, если XHR не поддерживается

Созданный объект XHR содержит удобный набор свойств и методов, присутствующих во всех поддерживающих его браузерах. Список этих свойств и методов приведен в табл. 10.1, а наиболее часто используемые из них описаны в последующих разделах.

Таблица 10.1. Методы и свойства XMLHttpRequest (XHR)

Методы	Описание
abort()	Отменяет выполнение текущего запроса
getAllResponseHeaders()	Возвращает единую строку, содержащую имена и значения всех заголовков ответа, или null, если не был получен ни один ответ
getResponseHeader(name)	Возвращает строку, содержащую текст заголовка с указанным именем, или null, если не получен ответ, либо если в ответе нет заголовка
open(method,url [, async [, username[, password]]])	Устанавливает HTTP (GET или POST) и URL, куда должен отправляться запрос. Дополнительно запрос может быть объявлен как синхронный, и к нему могут прилагаться имя пользователя и пароль для прохождения процедуры аутентификации на сервере
overrideMimeType(mime)	Устанавливает заголовок контентного типа для ответа mime

Продолжение ⇩

Таблица 10.1 (продолжение)

Методы	Описание
send([content])	Инициирует запрос. Необязательный параметр content предоставляет тело запроса. content игнорируется, если метод запроса относится к GET или HEAD
setRequestHeader(name, value)	Устанавливает заголовок запроса с использованием указанного имени и значения
onreadystatechange	Вызывает обработчик события, когда изменяется состояние запроса
readyState	Целочисленное значение, определяющее текущее состояние запроса следующим образом: 0 — не инициализирован; 1 — открыт; 2 — заголовки получены; 3 — загружается; 4 — завершен
response	Объект для ответа в соответствии с responseType
responseText	Содержимое тела, возвращенного в ответе
responseType	Может быть установлен для изменения типа ответа. Может принимать значения " " (пустая строка), arraybuffer, blob, document, json или text.
responseXML	Если содержимое ответа представляет собой документ XML, то из такого содержимого создается XML DOM
status	Код статуса, полученный от сервера. Например: 200 — успех, 404 — страница не найдена. Полный перечень кодов статуса вы найдете в спецификации протокола HTTP (http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html#sec10)
statusText	Текстовое сообщения статуса, полученное в ответе
timeout	Количество миллисекунд, которое пройдет до того, как запрос может быть отложен. Значение по умолчанию равно 0, что означает, что тайм-аута нет.
ontimeout	Вызов обработчика события, когда истекает время ожидания запроса
upload	Процесс загрузки может отслеживаться путем добавления слушателя события к upload
withCredentials	Определяет, должны ли межсайтовые запросы Access-Control осуществляться с помощью учетных данных, таких как cookies или авторизованные заголовки. Значение по умолчанию — false

ПРИМЕЧАНИЕ

Хотите получить ту же информацию из первых рук? Спецификация объекта XHR находится по адресу http://www.w3.org/TR/XMLHttpRequest/.

А теперь, когда у вас есть созданный экземпляр XHR, посмотрим, что нужно установить и как отправить запрос серверу.

10.1.2. Инициализация запроса

Прежде чем отправить запрос серверу, вы должны выполнить следующие действия.

1. Указать метод HTTP выполнения запроса (POST или GET).
2. Предоставить URL ресурса серверной стороны, которому будет направляться запрос.
3. Указать объекту XHR, как он должен информировать вас о ходе выполнения запроса.
4. Предоставить информацию, которая будет передаваться в теле запроса POST.

Первые два пункта этого списка выполняются с помощью вызова метода `open()` объекта XHR:

```
xhr.open('GET', '/some/resource/url');
```

Обратите внимание: данный метод не отправляет запрос серверу. Он просто устанавливает URL и метод HTTP, которые должны использоваться для выполнения запроса. Методу `open()` может быть передан третий логический параметр, который определяет, будет ли запрос асинхроннным (значение `true` по умолчанию) или синхронным (`false`). Редко возникают серьезные причины для отказа от асинхронных запросов (даже если это означает, что нам не придется возиться с функциями обратного вызова). В конце концов, асинхронная природа запроса — неоспоримое преимущество.

В третьем пункте списка вы должны задать для объекта XHR способ, которым он сможет информировать нас о том, что происходит. Этот пункт выполняется присваиванием функции обратного вызова свойству `onreadystatechange` объекта XHR. Эту функцию, называемую *обработчиком события изменения состояния*, экземпляр XHR будет вызывать на разных стадиях обработки запроса. Просматривая значения различных свойств XHR, вы сможете точно определить, как протекает процесс выполнения запроса. Работу типичного обработчика события изменения состояния мы рассмотрим в следующем подразделе.

Последний шаг инициализации запроса заключается в том, чтобы предоставить содержимое для запроса POST и отправить запрос серверу. Оба этих действия выполняются с помощью метода `send()`. Для запросов GET, обычно не имеющих тела, параметр с содержимым передается так:

```
xhr.send();
```

После передачи запрашиваемых параметров другому типу запросов строка, переданная методу `send()`, должна иметь определенный формат (который про себя можно называть форматом *строки запроса*), где имена и значения должны быть преобразованы в допустимое представление (закодированы в формат URI). Рассмотрение принципов URI-кодирования выходит за рамки этого раздела (к тому же jQuery делает все необходимое без нашего участия), но, если вам интересно, в JavaScript можно использовать `encodeURIComponent()`.

Пример такого вызова метода:

```
xhr.send('a=1&b=2&c=3');
```

Теперь познакомимся поближе с обработчиком события изменения состояния.

10.1.3. Отслеживание выполнения запроса

Экземпляр XHR может информировать нас о ходе выполнения запроса с помощью обработчика события изменения состояния. Этот обработчик задается путем присваивания свойству `onreadystatechange` экземпляра XHR ссылки на функцию, играющую роль обработчика.

После инициализации запроса с помощью вызова метода `send()` данная функция обратного вызова вызывается несколько раз в процессе прохождения различных этапов выполнения запроса. Текущее состояние (статус) запроса доступно в виде числового кода в свойстве `readyState` (см. описание свойства в табл. 10.1). Это здорово, но зачастую нам просто достаточно знать, выполнен ли запрос и успешно ли. Поэтому чаще всего нам будет встречаться реализация обработчика, приведенная в листинге 10.2.

Листинг 10.2. Обработчик события, игнорирующий все промежуточные состояния, кроме DONE

```
xhr.onreadystatechange = function() {
  if (this.readyState === 4) {
    if (this.status >= 200 && this.status < 300) {
      // Успешно
    } else {
      // Проблемы
    }
  }
}
```

Игнорирует все промежуточные состояния, кроме DONE

Выполняет в случае успеха

Выполняет в случае ошибки

Ветвление по статусу ответа

Этот код игнорирует все промежуточные состояния, кроме состояния завершения выполнения запроса `DONE` (статус со значением `4`), и после его наступления проверяет значение свойства `status`, чтобы определить, был ли запрос выполнен успешно. В спецификации протокола HTTP определено, что коды статуса в диапазоне от 200 до 299 означают успешное выполнение, а от 300 и выше свидетельствуют о различных ошибках.

А сейчас посмотрим, как выполняется обработка ответа по завершении запроса.

10.1.4. Получение ответа

После того как обработчик события изменения состояния определит, что свойство `readyState` содержит признак успешного завершения запроса, можно извлечь тело ответа из объекта XHR.

Несмотря на название *Ajax* (где символ X означает XML), формат тела ответа может иметь произвольный текстовый формат — он не ограничивается XML. Это могут быть и обычный текст, и фрагмент HTML-разметки, и любые данные, представленные с помощью формата JavaScript Object Notation (JSON).

Вне зависимости от формата тело ответа доступно через свойство `responseText` экземпляра XHR (здесь мы подразумеваем завершение успешное запроса). Если в ответе указано, что он имеет формат XML, — заголовок типа содержимого, указывает тип MIME `text/xml`, `application/xml` или любой другой тип MIME, оканчивающийся последовательностью символов `+xml`, — то содержимое ответа будет анализироваться как документ XML. Доступ к полученному DOM можно получить через свойство `responseXML`. Для обработки XML DOM можно применять JavaScript (и jQuery, используя API селектора).

А сейчас вам может понадобиться освежить в памяти диаграмму всего процесса, показанную выше (см. рис. 10.1). В этом кратком обзоре технологии Ajax мы определили следующие проблемные моменты, с которыми приходится сталкиваться авторам страниц при работе с Ajax:

- при создании объекта XHR необходимо учитывать характерные особенности браузера;
- обработчики события изменения состояния вынуждены отсеивать многие изменения состояния, не представляющие для нас никакого интереса;
- содержимое ответа приходится обрабатывать самыми разными способами, в зависимости от используемого формата.

В оставшейся части главы будут описаны методы и вспомогательные функции jQuery, которые упрощают (и делают более понятным) применение технологии Ajax на ваших страницах. jQuery Ajax API содержит много функций, и мы начнем знакомство с самых простых и наиболее часто используемых инструментов.

10.2. Загрузка содержимого в элементы

Пожалуй, чаще всего технология Ajax применяется для получения от сервера порций содержимого и их добавления в ряд важных для нас мест дерева DOM. Содержимое может быть порцией фрагмента HTML, которая может быть включена в целевой контейнер, или простым текстом, который затем станет содержимым целевого элемента.

Установка для примеров

В отличие от примеров, рассматривавшихся до сих пор, код примеров данной главы требует наличия веб-сервера, который будет принимать запросы к ресурсам серверной стороны. Поскольку обсуждение работы серверных механизмов выходит далеко за рамки нашей книги, мы не будем рассматривать установку сервера.

Код, который мы будем использовать на стороне сервера, разработан в PHP, так что ваш сервер должен быть в состоянии обработать его. Если вы никогда этого не делали, то вот

список инструментов для начинающих, независимо от применяемой вами операционной системы:

- для пользователей Windows: http://www.wampserver.com/ru;
- для пользователей Mac: https://www.mamp.info.

Если вы работаете с другим языком, таким как Java или ASP.NET, то должны быть в состоянии портировать код на выбранный язык, потому что страницы очень просты. Если вы решили преобразовать страницы в Java или ASP.NET, то должны также создать веб-сервер, который в состоянии понять эти языки, такие как соответственно Tomcat и IIS. Вот список ресурсов, которые помогут вам:

- Tomcat для пользователей Windows и Linux: https://tomcat.apache.org/tomcat-7.0-doc/setup.html;
- Tomcat для пользователей Mac: http://serverfault.com/questions/183496/how-do-i-start-apache-tomcat-at-boot-on-mac-os-x;
- IIS для пользователей Windows: http://www.iis.net/learn/install/installing-iis-7/installing-iis-on-windows-vista-and-windows-7.

Предположим, вы хотите получить от сервера HTML-фрагмент, используя ресурс с именем `some-resource`, и превратить его в содержимое элемента `<div>` со значением ID, равным `elem`. Последний раз в данной главе мы посмотрим, как это можно сделать без помощи jQuery.

Используя шаблоны, установленные ранее в этой главе, вы можете написать код, приведенный в листинге 10.3. Полный HTML-файл для данного примера доступен в файле `chapter-10/listing.10.3.html`, предоставленном с книгой.

ПРИМЕЧАНИЕ

Вы должны запустить этот пример с использованием веб-сервера — недостаточно просто открыть страницу в браузере, — так, чтобы URL был http://localhost:8080/chapter-10/listing.10.3.html. Опустите указание порта :8080 в URL, если вы используете Apache, и укажите его, если задействуете Tomcat. Далее в этой главе номер порта в адресах URL будет подаваться в виде [:8080], чтобы показать, что указывать номер порта может не потребоваться, но убедитесь, что сами квадратные скобки вы не включаете в URL.

Листинг 10.3. Использование нативного XHR для получения фрагмента HTML и включения его в страницу

```
var xhr;
if (window.ActiveXObject) {
  xhr = new ActiveXObject('Microsoft.XMLHTTP');
} else if (window.XMLHttpRequest) {
  xhr = new XMLHttpRequest();
} else {
  throw new Error('Ajax не поддерживается этим браузером');
}
```

```
xhr.onreadystatechange = function() {
  if (this.readyState === 4) {
    if (this.status >= 200 && this.status < 300) {
      document.getElementById('elem').innerHTML = this.responseText;
    }
  }
}
xhr.open('GET', 'some-resource');
xhr.send();
```

Хотя здесь нет ничего сложного, программный код получился немалым — 17 строк. Эквивалентный код, который вы бы написали с использованием jQuery, выглядит так:

```
$('#elem').load('some-resource');
```

Готовы поспорить, что знаем, какой код предпочли бы писать и поддерживать вы! Познакомимся с методами jQuery, которые мы применяли в этой инструкции.

10.2.1. Загрузка содержимого с помощью jQuery

Простая инструкция в конце предыдущего раздела легко справляется с загрузкой содержимого с сервера с помощью одного из самых простых и полезных методов jQuery Ajax: `load()`. Полное описание синтаксиса этого метода приводится ниже.

Синтаксис метода: load

`load(url[, data][, callback])`

Выполняет запрос Ajax к заданному URL, передавая дополнительные данные. Если указана функция обратного вызова, то она вызывается после завершения запроса и изменения DOM. Содержимое всех элементов набора будет замещено текстом ответа.

Параметры

`url` (Строка) URL ресурса серверной стороны, которому отправляется запрос, дополнительно модифицированный с помощью селектора (объясняется ниже).

`data` (Строка|Объект|Массив) Определяет любые данные, подлежащие передаче в качестве параметров запроса. Этот параметр может быть строкой, которая будет использоваться в качестве строки запроса или тела ответа, объектом, свойства которого сериализованы, или массивом объектов, чьи свойства `name` и `value` определяют пары «имя — значение».

Если этот параметр задан как объект или как массив, то запрос будет сделан с помощью метода POST. Если же он опущен или указан в виде строки, то используется метод GET.

`callback` (Функция) Необязательная функция обратного вызова, которая вызывается после того, как данные, полученные в ответе, будут загружены в элементы набора. В качестве параметра этой функции передается текст ответа, строка статуса (обычно `"success"`) и экземпляр `jqXHR` (скоро объясним).

Эта функция будет вызвана для каждого элемента в коллекции jQuery, при этом сам элемент будет передан ей через контекст функции (`this`).

Возвращает

Коллекцию jQuery.

В описании данного метода мы представили новый объект, называемый `jqXHR`. Это имя — аббревиатура от *jQuery XMLHTTPRequest*, являющегося надстройкой над объектом `XMLHTTPRequest` (XHR). Например, он содержит свойства `responseText` и `responseXML`, а также метод `getResponseHeader()`. Он реализует интерфейс `Promise`, который мы обсудим подробнее в главе 13.

Несмотря на простоту использования, этот метод обладает некоторыми важными особенностями. Например, если параметр `data` применяется для подачи параметров запроса и переданный аргумент является объектом, то запрос делается с помощью метода POST HTTP; в противном случае инициируется запрос GET. Если вы хотите сделать запрос GET с параметрами, то можете включить их в виде строки запроса в URL. Только учтите: когда сделаете это, будете нести ответственность за правильность форматирования строки запроса и наличие URI-кодировки имен и значений параметров запроса. Для этого удобен метод JavaScript `encodeURIComponent()` либо можно использовать вспомогательную функцию jQuery `$.param()`, которую мы рассмотрели в главе 9.

Иногда бывает нужно совершить действие только после введения содержимого в один или несколько элементов. Предположим, вы отсылаете серверу запрос об обновлении статуса лондонского метро и хотите, чтобы каждый раз, когда получаете обновление, показывалось сообщение на странице. В качестве примера повторим запрос через одну секунду после того, как будет выполнена функция обратного вызова. Базовая реализация, которая удовлетворяет этим требованиям, выглядит так:

```
var updates = 1;
function pollInfo() {
  $('#container').load(
     '/check-updates',
     function(responseText, textStatus, jqXHR) {
        if (textStatus === 'success') {
           $('#status-update').text('Данные обновлены.
                        Обновление #' + updates);
           updates++;
        }
        setTimeout(pollInfo, 1000);
     }
  );
}
pollInfo();
```

Вызов функции pollInfo() через 1000 миллисекунд

Вызов функции pollInfo() первый раз

В большинстве случаев загрузку полного текста ответа в элементы объекта jQuery вы будете выполнять с помощью метода `load()`, но иногда может потребоваться отфильтровать ряд элементов, полученных в ответе. Чтобы сделать это, библиотека дает возможность определять селектор, позволяющий выбрать элементы, которые должны быть загружены в элементы набора. Для этого к строке URL можно добавить пробел, за ним и будет следовать селектор.

Например, отфильтровать из ответа все элементы, не являющиеся экземплярами `<div>`, позволит следующая инструкция:

```
$('.inject-me').load('/some-resource div');
```

Используемый селектор может быть произвольной сложности. Это значит, можно написать инструкцию, подобную данной ниже:

```
$('.inject-me').load('/some-resource div .some-class a');
```

В этом случае jQuery будет искать все элементы, соответствующие указанному селектору.

Что касается передачи данных с запросом — иногда нам придется отправлять некоторые данные, полученные динамически, но чаще всего предстоит собирать и отправлять данные, введенные пользователем в поля формы.

Как вы уже наверняка догадались, у jQuery есть кое-что и для этого случая.

Сериализация данных формы. Если данные, которые вы хотите отправить в виде параметров запроса, собираются из элементов формы, то можно воспользоваться удобным методом `serialize()`. Его синтаксис приводится ниже.

Синтаксис метода: serialize

`serialize()`

Создает правильно отформатированную и закодированную строку запроса из всех успешных элементов управления формы в коллекции jQuery.

Параметры

Отсутствуют.

Возвращает

Отформатированную строку запроса.

Метод `serialize()` настолько умный, что позволяет выбирать информацию только из элементов формы в элементах набора и только из считающихся *успешными*. Успешный элемент — тот, который был бы отправлен серверу в процессе отправки формы согласно правилам, изложенным в спецификации HTML[1]. Такие элементы, как неустановленные флажки и переключатели, списки, в которых не был выбран ни один из вариантов, а также любые неактивные элементы, не считаются успешными и не передаются серверу вместе с формой. Метод `serialize()` также проигнорирует их.

Возможность сериализовать значения для отправки их на сервер является хорошим дополнением к jQuery, но, к сожалению, наша любимая библиотека не предоставляет вспомогательной функции для выполнения противоположной операции — десериализации.

Десериализация — операция заполнения и изменения состояния полей формы, основанная на сериализованной строке. Она может пригодиться, если у вас есть сложная форма поиска и вы хотите сохранить некоторые наиболее частые запросы конкретного пользователя. В данном случае можно сохранить значения полей

[1] Спецификация HTML 4.01 Specification, раздел 17.13.2, Successful controls («Успешные элементы управления»): http://www.w3.org/TR/html401/interact/forms.html#h-17.13.2.

в cookie или базе данных, чтобы пользователь мог нажать кнопку и восстановить эти значения без заполнения формы снова и снова.

Когда jQuery не может предложить решение, динамичное сообщество вокруг проекта помогает его найти. Это еще одна причина, по которой библиотека является настолько замечательной. В ситуациях, аналогичных вышеописанной, вы можете обратиться к плагину jQuery.deserialize (https://github.com/kflorence/jquery-deserialize).

Десериализация данных с jQuery.deserialize

Чтобы использовать плагин jQuery.deserialize, сначала необходимо включить его в вашу страницу. Метод подобен тем, которые уже не раз были описаны в этой книге, — нужно поместить ссылку в библиотеку внутри тега `<script>` после библиотеки jQuery, как показано здесь:

```
<script href="path/to/jquery.js"></script>
<script href="path/to/jquery.deserialize.js"></script>
```

Установив плагин, представим, что есть такая форма:

```
<form id="my-form">
    <input name="имя" />
    <input name="фамилия" />
</form>
<button id="btn">Автозаполнение</button>
```

Есть также следующая строка — результат предыдущей сериализации:

```
name=Aurelio&surname=De+Rosa
```

Поэтому, если хотите получить форму автозаполнения, как только пользователь нажимает кнопку btn, можете написать:

```
$('#btn').click(function() {
  $('#my-form').deserialize('name=Aurelio&surname=De+Rosa');
});
```

Если нужно получить данные формы в виде массива JavaScript (в противоположность строке запроса), то jQuery предоставляет метод `serializeArray()`.

Синтаксис метода: serializeArray

`serializeArray()`

Собирает значения из всех успешных элементов формы в массив объектов, содержащих имена и значения элементов управления.

Параметры

Отсутствуют.

Возвращает

Массив с данными формы.

Массив, возвращаемый методом `serializeArray()`, состоит из литералов объектов, каждый из которых содержит свойства `name` и `value` с именем и значением каждого успешного элемента формы. Обратите внимание: массив — это один из форматов (совсем не случайно), который можно использовать для передачи параметров запроса методу `load()`.

С помощью метода `load()` можно решать задачу, с которой часто приходится сталкиваться веб-разработчикам.

10.2.2. Загрузка динамических фрагментов HTML

Довольно часто в бизнес-приложениях, особенно на сайтах интернет-магазинов, требуется получать данные от сервера в реальном времени, чтобы предоставить пользователям самую свежую информацию. В конце концов, никто не хочет вводить покупателей в заблуждение, предлагая им купить то, чего на самом деле нет, верно?

В этом подразделе мы начнем разработку страницы, которую будем расширять на протяжении всей главы. Данная страница — часть сайта вымышленной фирмы The Boot Closet («Чулан обуви»), занимающейся розничной продажей уцененных излишков мотоциклетной обуви. В отличие от фиксированных каталогов продукции в других интернет-магазинах, информация об излишках и уценке постоянно меняется в зависимости от того, какие сделки удалось заключить сегодня и что уже удалось продать. Поэтому важно предоставлять пользователям свежайшую информацию!

Для начала на странице (опустим элементы навигации по сайту и прочие типичные элементы и сосредоточимся на предмете изучения) мы предоставим клиентам раскрывающийся список, содержащий доступные в настоящий момент модели обуви, а при выборе модели будет отображаться подробная информация о ней. Сразу после открытия страница будет выглядеть так, как показано на рис. 10.2.

Рис. 10.2. Начальный вид страницы магазина с простым раскрывающимся списком, приглашающим посетителей щелкнуть на нем

После начальной загрузки страница будет содержать раскрывающийся список с перечнем моделей, доступных в настоящее время. Если модель не выбрана, то

в раскрывающемся списке будет выводиться текст -выберите модель-. Это сообщение приглашает пользователя открыть раскрывающийся список и произвести соответствующее действие. После этого вам потребуется выполнить следующие операции:

- вывести подробные сведения о выбранной модели под раскрывающимся списком;
- удалить элемент с текстом -выберите модель-, как только пользователь выберет интересующую его модель, так как дальнейшее присутствие этого элемента теряет всякий смысл.

Для начала рассмотрим HTML-разметку, которая создаст структуру страницы:

```
<body>
   <div id="banner"></div>

   <h1>Выберите вашу обувь</h1>
   <div>
      <div id="selections-pane">
         <label for="boot-chooser-control">Модель обуви:</label>
         <select id="boot-chooser-control" name="model"></select>
      </div>
      <div id="product-detail-pane"></div>
   </div>
</body>
```

Содержит раскрывающийся список ❶

Место для вывода информации о продукте ❷

Совсем немного, правда? Как и ожидалось, всю информацию о визуальном представлении мы определили во внешней таблице стилей (здесь не показываем), не включив в HTML-разметку никаких аспектов поведения, чтобы соответствовать принципам ненавязчивого JavaScript.

Наиболее интересные части в этой разметке — контейнер ❶, содержащий элемент `select`, дающий пользователю возможность выбрать модель обуви, и контейнер ❷, в котором будет выводиться информация о продукте.

Обратите внимание: прежде чем пользователь получит возможность взаимодействовать с этой страницей, нам необходимо заполнить раскрывающийся список элементами `option` с названиями моделей. Приступим к реализации поведения нашей страницы. В первую очередь необходимо отправить запрос Ajax, чтобы получить перечень доступных моделей и заполнить раскрывающийся список.

ПРИМЕЧАНИЕ

В большинстве случаев начальные значения, такие как список моделей обуви в данном примере, подготавливаются сервером еще до отправки HTML браузеру. Это значит, что даже если у ваших пользователей отключен JavaScript или не может выполняться его код, то они по-прежнему смогут использовать веб-страницу. Но в некоторых случаях предпочтительнее получать данные с применением технологии Ajax, что мы и реализуем здесь в учебных целях.

Для добавления элементов в раскрывающийся список воспользуйтесь удобным методом `load()`:

```
$('#boot-chooser-control').load('actions/fetch-boot-style-options.php');
```

Что может быть проще? Единственная сложность в этой инструкции — адрес URL, который в действительности не настолько длинный или сложный, определяющий запрос PHP-страницы серверной стороны.

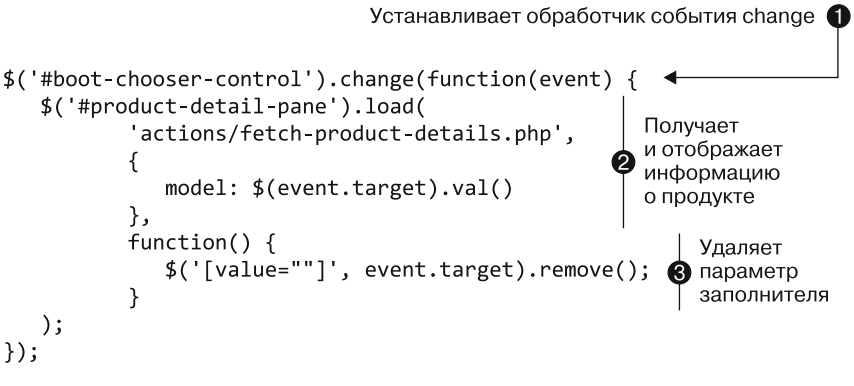
Одна из приятных особенностей применения Ajax (наряду с легкостью работы с jQuery, делающей процесс еще приятнее) состоит в полной независимости от серверных технологий. Вы создаете HTTP-запросы, иногда с дополнительными параметрами, и пока сервер отвечает на них, возвращая ожидаемую информацию, не обращаете внимания на язык, на котором он работает, — Java, Ruby, PHP или даже старый добрый CGI.

В данном конкретном случае мы ожидаем, что сервер вернет HTML-разметку, представляющую список элементов `option` с названиями моделей, по-видимому извлеченными из базы данных. Наш серверный сценарий возвращает следующий ответ:

```
<option value="">- выберите модель -</option>
<option value="7177382">Caterpillar Tradesman Work Boot</option>
<option value="7269643">Caterpillar Logger Boot</option>
<option value="7332058">Chippewa 9" Briar Waterproof Bison Boot</option>
<option value="7141832">Chippewa 17" Engineer Boot</option>
<option value="7141833">Chippewa 17" Snakeproof Boot</option>
<option value="7173656">Chippewa 11" Engineer Boot</option>
<option value="7141922">Chippewa Harness Boot</option>
<option value="7141730">Danner Foreman Pro Work Boot</option>
<option value="7257914">Danner Grouse GTX Boot</option>
```

Ответ вставляется в элемент `select`, в результате чего мы получаем полнофункциональный элемент управления.

Следующее действие — добавить в раскрывающийся список обработчик события, чтобы он мог реагировать на изменения, выполняя перечисленные выше операции. Реализация этого обработчика получилась чуть более сложной:



В этом коде вы выбираете раскрывающийся список моделей обуви и подключаете к нему обработчик события `change` ❶. В данном обработчике, вызываемом всякий раз, когда покупатель меняет выбор, вы получаете текущее значение выбора путем вызова метода jQurey `val()` на целевом событии после его обертки с помощью `$()`. В таком случае целевой элемент — тот `select`, который вызвал событие.

Далее вы применяете метод `load()` ❷ к элементу `product-detail-pane`, чтобы отправить запрос к странице `actions/fetch-product-details.php`. Этой странице мы отправляем модель обуви с помощью литерала объекта, чье единственное свойство называется `model`. И наконец, удаляем параметр заполнителя ❸ внутри функции обратного вызова метода `load()`.

После того как покупатель выберет одну из доступных моделей обуви, страница будет выглядеть так, как показано на рис. 10.3.

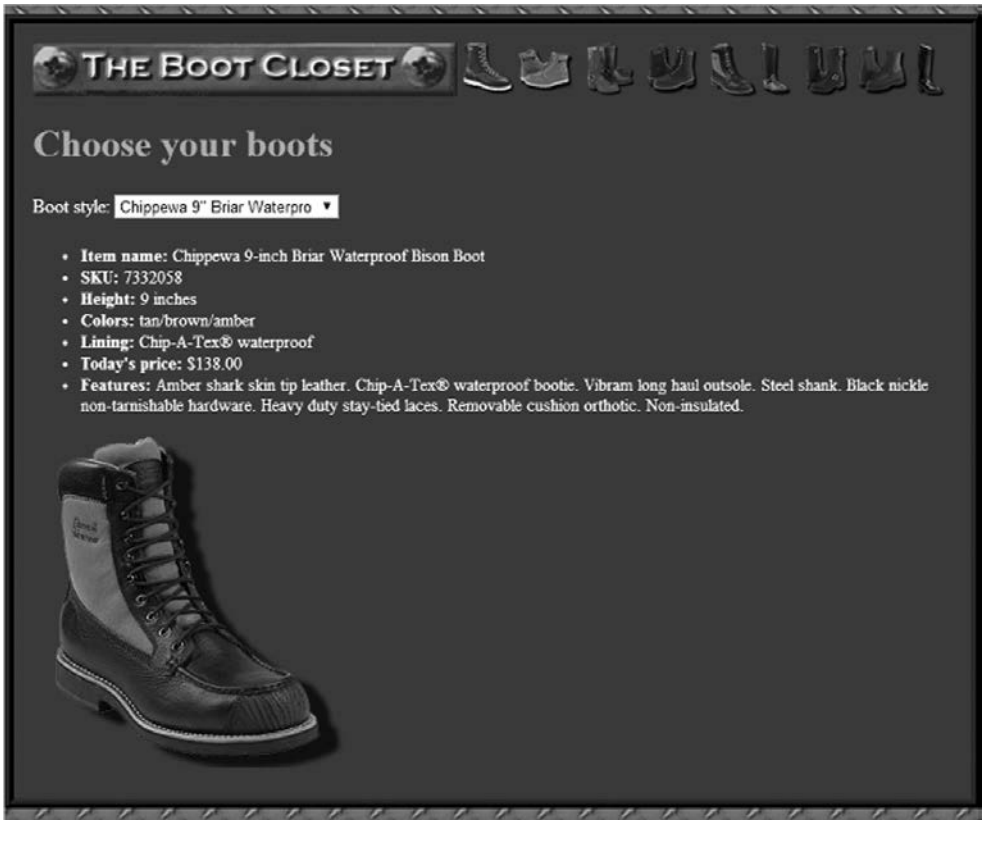


Рис. 10.3. Серверный ресурс возвращает подготовленный фрагмент HTML для отображения подробной информации об обуви

Самая примечательная выполненная операция — это использование метода `load()` для быстрой и простой загрузки с сервера фрагментов HTML и размещения их в DOM в качестве наследников существующих элементов. Данный метод

чрезвычайно удобен и прекрасно подходит для веб-приложений, приводимых в движение серверными механизмами.

В листинге 10.4 приведен полный код страницы «Чулан обуви», которую можно найти на http://localhost[:8080]/chapter-10/phase.1.html. Вы еще не раз будете возвращаться к ней на протяжении всей главы, чтобы дополнять ее новыми возможностями.

Листинг 10.4. Первая фаза страницы интернет-магазина «Чулан обуви»

```
<!DOCTYPE html>
<html>
  <head>
    <title> Чулан обуви — 1-я фаза</title>
    <link rel="stylesheet" href="../css/main.css" />
    <link rel="stylesheet" href="../css/bootcloset.css">
  </head>
  <body>
    <div id="banner"></div>
    <h1>Выберите вашу обувь</h1>
    <div>
      <div id="selections-pane">
        <label for="boot-chooser-control">Модель обуви:</label>
        <select id="boot-chooser-control" name="model"></select>
      </div>
      <div id="product-detail-pane"></div>
    </div>
    <script src="../js/jquery-1.11.3.min.js"></script>
    <script>
      $('#boot-chooser-control')
            .load('actions/fetch-boot-style-options.php')
            .change(function(event) {
                $('#product-detail-pane').load(
                        'actions/fetch-product-details.php',
                        {
                            model: $(event.target).val()
                        },
                        function() {
                            $('[value=""]', event.target).remove();
                        }
            );
          });
    </script>
  </body>
</html>
```

Метод `load()` невероятно удобно использовать, когда требуется взять фрагмент HTML, чтобы включить его в элемент (или набор элементов). Но иногда нужно получить более полный контроль над тем, как выполняется запрос Ajax, или совершить нечто более сложное с полученными данными.

Продолжим наши исследования и посмотрим, что может предложить jQuery в этих более сложных ситуациях.

10.3. Выполнение запросов GET и POST

Метод `load()` выполняет запрос GET либо POST в зависимости от того, в каком виде ему передаются (если вообще передаются) данные с параметрами запроса, но иногда вам может быть необходим чуть больший контроль над тем, каким методом HTTP производится запрос. Зачем это *вам*? Затем, что это может быть нужно вашему *серверу*.

По традиции авторы веб-страниц безответственно относились к методам GET и POST, используя то один, то другой и не задумываясь над тем, для каких целей они предназначены. Между тем у каждого из методов свой сектор ответственности.

- *Запросы GET* соблюдают *тождественность (idempotent)*; один и тот же запрос GET, выполняясь снова и снова, должен возвращать в точности одни и те же результаты (предполагается, что в это время не происходит ничего другого, что привело бы к изменению состояния сервера).
- *Запросы POST* могут быть *нетождественными (non-idempotent)*; данные, передаваемые серверу в таких запросах, могут использоваться для изменения состояния приложения, например для добавления записей в базу данных или удаления информации с сервера.

Таким образом, запросы GET должны применяться просто для получения данных, что и следует из названия метода. Для этого может потребоваться передавать некоторые данные на сервер, например чтобы идентифицировать номер модели обуви для получения информации о цвете. Но когда данные передаются на сервер для вызова изменений, следует обратиться к методу POST.

ВНИМАНИЕ

> Это не просто теория. Браузеры принимают решение о кэшировании данных на основе типа используемого запроса. Запросы GET — наиболее вероятные кандидаты на попадание в кэш. Выбор надлежащего метода HTTP может гарантировать, что вы не вступите в противоречие с ожиданиями браузера, касающимися различных типов запросов. Это всего лишь один из принципов архитектуры RESTful, которая также подразумевает использование других методов HTTP, таких как PUT и DELETE. Однако здесь мы ограничимся методами GET и POST.

Учитывая это, если вернетесь к первой фазе реализации «Чулана обуви» (см. листинг 10.4), обнаружите следующее: вы *использовали запросы неправильно*! Поскольку jQuery инициирует запрос POST, когда вы поставляете объект для параметра `data`, делаете POST там, где следовало делать запрос GET. Если вы заглянете в журнал Chrome's Developer Tools (как показано на рис. 10.4), отобразив страницу в Chrome, то увидите, что второй запрос, отправленный при выборе модели из раскрывающегося списка, в действительности был POST.

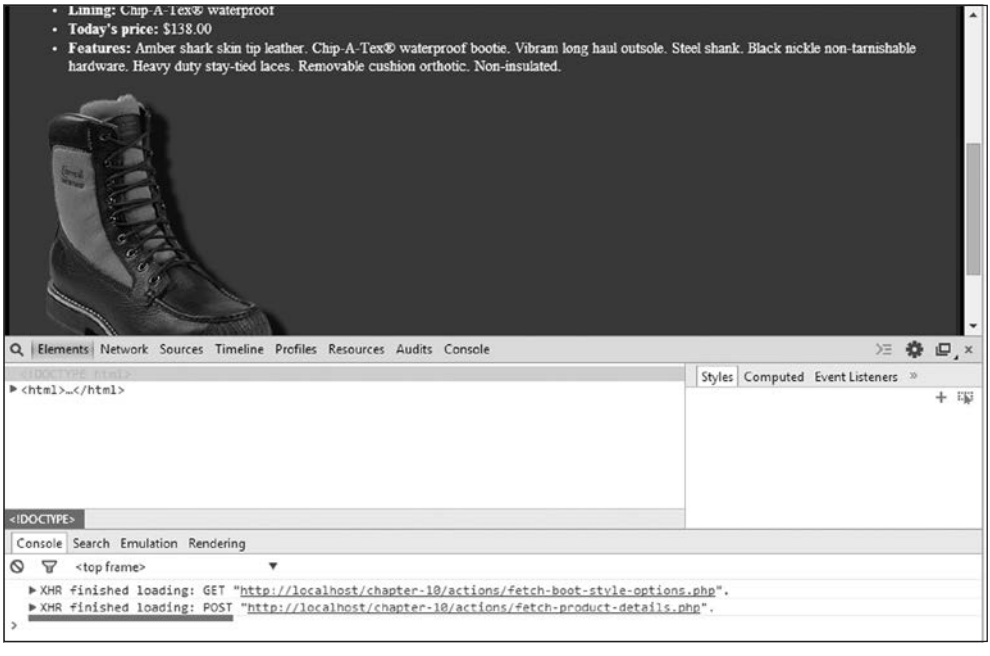


Рис. 10.4. Инспекция консоли Chrome показывает, что вы сделали запрос POST, тогда как следовало GET

ПРИМЕЧАНИЕ

Результат, который увидите в консоли, будет таким же, как и на рисунке, только в том случае, если вы включите параметр Log XMLHttpRequests. Если не хотите включать его, то можете посмотреть на вкладке Network (Сеть).

Действительно ли это имеет какое-то значение, решать вам, но если хотите использовать протокол HTTP в соответствии со спецификациями, то запрос на получение информации о модели должен выполняться с помощью метода GET, а не POST.

Инструменты разработчика

Попытка разработать приложения со сценариями DOM без помощи инструмента отладки похожа на попытку играть на концертном пианино в перчатках для сварки. Вы же не станете так делать?

В зависимости от используемого браузера существуют различные варианты проверки кода. У всех современных основных браузеров есть для этой цели набор встроенных инструментов, у каждого под своим названием. Например, в Chrome эти инструменты называются *Developer Tools* (https://developer.chrome.com/devtools/), в Internet Explorer — *F12 developer tools* (http://msdn.microsoft.com/en-us/library/bg182326(v=vs.85).aspx). У Firefox также есть собственные встроенные инструменты, но разработчики, как правило, применяют плагин *Firebug* (http://getfirebug.com). Эти инструменты позволяют

инспектировать не только консоль JavaScript, но и DOM, CSS, сценарии и многие другие аспекты вашей страницы в процессе работы над ее развитием.

Одной из наиболее актуальных возможностей для текущих целей является возможность вести журналы запросов Ajax с фиксацией и запросов, и ответной информации.

Вы могли бы просто передать параметры запроса в виде строки, а не объекта (мы рассмотрим эту возможность ниже), но сейчас воспользуемся другим способом инициализации запросов Ajax, предоставляемым jQuery.

10.3.1. Получение данных методом GET

Библиотека предоставляет несколько способов выполнения запросов GET, которые, в отличие от `load()`, реализованы в виде не методов jQuery, а вспомогательных функций. Эти функции уже упоминались в предыдущей главе, но не рассматривались подробно.

Когда необходимо получить некоторые данные с сервера и решить, что с ними делать (вместо того чтобы позволить методу `load()` загрузить их в качестве содержимого элемента HTML), можно использовать вспомогательную функцию `$.get()`.

Синтаксис функции: $.get

`$.get([url[, [, data][, callback][, dataType])`

Отправляет запрос GET к серверу, используя заданный URL и все параметры, передаваемые в виде строки запроса.

Параметры

`url` (Строка) URL ресурса на стороне сервера, которому отправляется запрос GET. Если передается пустая строка, то запрос отправляется на текущий URL во время вызова метода.

`data` (Строка|Объект) Определяет любые данные для передачи в качестве параметров в строке запроса. Этот параметр не является обязательным и может быть строкой или объектом, свойства которого сериализуются в правильно закодированные параметры, передаваемые запросу.

`callback` (Функция) Необязательная функция, вызываемая после успешного завершения запроса. Первым ее параметром передается текст ответа, интерпретация которого зависит от значения параметра `dataType`, вторым — код статуса. В третьем параметре функции будет передан объект `jqXHR`. Внутри функции обратного вызова контекст (`this`) устанавливается на объект, представляющий параметры Ajax, используемые при вызове. Этот параметр становится обязательным, если предоставляется `dataType`. Если вам не нужна функция, то можете передать `null` или `$.noop()` в качестве заполнителя.

`dataType` (Строка) Необязательный параметр, определяющий, как должен интерпретироваться текст ответа. Может иметь одно из следующих значений: `html`, `text`, `xml`, `json`, `script` или `jsonp`. Значение по умолчанию определяется jQuery в зависимости от полученного ответа и может быть одним из `xml`, `json`, `script` или `html`. Дополнительная информация дана в главе ниже, в описании функции `$.ajax()`.

Возвращает

Экземпляр `jqXHR`.

jQuery 3: добавленная сигнатура

В jQuery 3 добавлена новая сигнатура для вспомогательной функции `$.get()`:

```
$.get ([options])
```

Объект `options` может обладать многими свойствами. Чтобы узнать об этом больше, пожалуйста, обратите внимание на описание вспомогательной функции `$.ajax()` (обсуждается ниже). Стоит отметить: свойство `method`, которое может содержать объект `options`, будет автоматически установлено в положение `"GET"`.

Вспомогательная функция `$.get()` позволяет более гибко инициировать запросы GET. Вдобавок к параметрам запроса и функции обратного вызова, вызываемой после успешного ответа, теперь вы можете указывать, как интерпретировать ответ и как он будет передан функции обратного вызова. Если даже такой универсальности будет недостаточно, можете использовать еще более универсальную функцию `$.ajax()`, где подробнее изучите параметр `dataType`. Пока мы будем применять по умолчанию `html` или `xml` в зависимости от типа содержимого в ответе. Чтобы задействовать функцию `$.get()` в реализации нашей страницы «Чулан обуви», заменим вызов метода `load()`, как показано в листинге 10.5.

Листинг 10.5. Изменения в странице «Чулан обуви», использующей метод GET для получения информации о модели обуви

```
$('#boot-chooser-control')
        .change(function(event) {
           $.get(
                   'actions/fetch-product-details.php',
                   {
                      model: $(event.target).val()
                   },
                   function(response) {
                      $('#product-detail-pane').html(response);
                      $('[value=""]', event.target).remove();
                   }
           );
        });
```

Создает запрос GET ❶

Вставляет фрагмент HTML ❷

Во второй фазе в нашу страницу внесены незначительные, но очень важные изменения. Вместо метода `load()` мы вызываем функцию `$.get()` ❶, передавая ей тот же адрес URL и те же параметры запроса. Поскольку в этом случае вы используете вспомогательную функцию `$.get()`, можете убедиться, что будет выполнен запрос GET, даже если передадите объект. Функция `$.get()` не вставляет автоматически текст ответа в дерево DOM, поэтому потребовалось произвести эту операцию вручную, что было совсем несложно благодаря методу `html()` ❷.

Код этой версии страницы доступен по адресу http://localhost[:8080]/chapter-10/phase.2.html. Если открыть ее в браузере и выбрать модель в раскрывающемся списке, то можно увидеть, что запрос был выполнен методом `GET`, как показано на рис. 10.5.

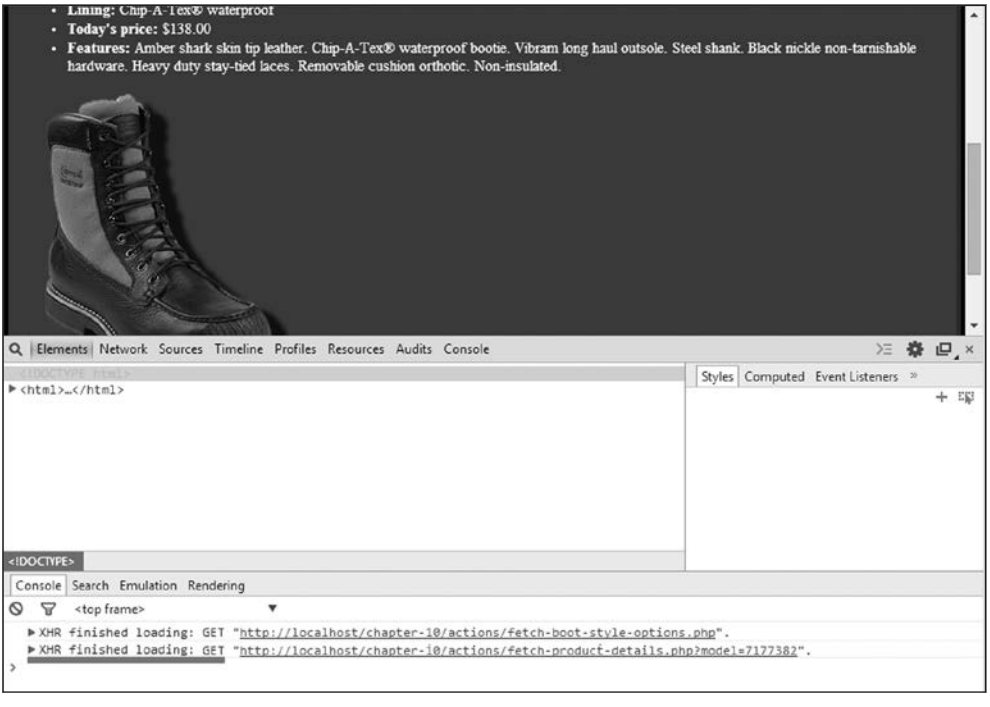


Рис. 10.5. Теперь вы можете увидеть, что второй запрос GET, а не POST, как и полагается в данном случае

В этом примере сервер возвращает фрагмент HTML, который затем вставляется в дерево DOM. Но так как у функции `$.get()` есть дополнительный параметр `dataType`, существует много возможностей для использования не только с HTML.

Познакомимся еще с одной вспомогательной функцией jQuery, которая чрезвычайно удобна в тех случаях, когда структура ваших данных предполагает, что нужно использовать формат JSON.

10.3.2. Получение данных в формате JSON

Когда XML является излишним или непригодным механизмом передачи данных, часто вместо него задействуют формат JSON. Одна из причин состоит в том, что с ним очень просто работать в клиентских сценариях, а jQuery еще больше упрощает работу.

В случаях, когда заранее известно, что форматом ответа будет JSON, можно использовать вспомогательную функцию `$.getJSON()`, которая автоматически проанализирует полученную строку JSON, а полученные данные передаст функции обратного вызова. Синтаксис `$.getJSON()` показан ниже, у функции те же параметры, что и у `$.get()`, поэтому мы не будем их повторять.

Синтаксис функции: $.getJSON

`$.getJSON(url][, data] [, callback)`

Отправляет запрос GET к серверу, применяя заданный URL и любые параметры, переданные в виде строки запроса. Ответ сервера интерпретируется как строка в формате JSON, а полученные из нее данные передаются функции обратного вызова.

Возвращает

Экземпляр `jqXHR`.

Как видите, эта функция представляет собой удобную замену вызова функции `$.get()` со значением `"json"` в аргументе `dataType`.

Функция `$.getJSON()` — далеко не единственная, в чем вы убедитесь в следующем подразделе.

10.3.3. Динамическая загрузка сценариев

В большинстве случаев внешние сценарии, необходимые для нашей страницы, загружаются из файлов сценариев с помощью тегов `<script>` внизу страницы. Но время от времени может понадобиться загрузить какой-то сценарий и во время выполнения другого сценария. Так может получиться потому, что заранее, пока пользователь не совершил некие действия, мы можем не знать, нужен ли этот сценарий, и хотим включать его только при возникшей необходимости. Может быть и так: для выбора из нескольких сценариев нам потребуется информация, отсутствующая на этапе загрузки.

Вне зависимости от того, почему вам нужно динамически загрузить новые сценарии на страницу, jQuery предоставляет вспомогательную функцию `$.getScript()`, позволяющую упростить данную работу. У этой вспомогательной функции также есть параметры, с теми же значениями, которые были описаны для `get()`, так что мы не будем их повторять.

Синтаксис функции: $.getScript

`$.getScript (url [, callback])`

Загружает сценарий, приведенный в параметре `url`, выполняя запрос GET к указанному серверу, дополнительно вызывая функцию обратного вызова в случае успеха. URL не ограничивается тем же доменом, что и содержащие страницы.

Возвращает

Экземпляр `jqXHR`.

«За кулисами» эта функция вызывает `$.get()` путем установки параметра `data` в `undefined`, а параметра `dataType` в `"script"`. В исходнике jQuery вспомогательная функция `$.getScript()` определяется так:

```
getScript: function( url, callback ) {
  return jQuery.get( url, undefined, callback, "script" );
}
```

По завершении действия данной функции сценарий в файле вычисляется, любой встроенный сценарий выполняется и любые определенные переменные или функции становятся доступными.

Посмотрим на это в действии. Рассмотрим следующий файл сценария (доступен в файле `chapter-10/external.js`, предоставленном с книгой):

```
alert('Я – встроенный сценарий!');
var someVariable = 'Значение someVariable';
function someFunction(value) {
  alert(value);
}
```

Этот простой сценарий содержит встроенную инструкцию (вывод сообщения, подтверждающего, что выполняется именно эта инструкция), объявление переменной и объявление функции для вывода сообщения, содержащего все значения, которые были переданы функции во время выполнения. Теперь создадим страницу, загружающую файл данного сценария динамически. Она приведена в листинге 10.6 и доступна в файле `chapter-10/$.getScript.html`.

Листинг 10.6. Динамическая загрузка файла сценария и проверка результатов

```
<!DOCTYPE html>
<html>
   <head>
      <title>$.getScript() Пример</title>
      <link rel="stylesheet" href="../css/main.css"/>
   </head>
   <body>
      <button id="load-button">Загрузить</button>
      <button id="inspect-button">Проверить</button>

      <script src="../js/jquery-1.11.3.min.js"></script>
      <script>
         $('#load-button').click(function() {
            $.getScript('external.js');
         });

         $('#inspect-button').click(function() {
            someFunction(someVariable);
         });
      </script>
   </body>
</html>
```

❶ Определяет тестовые кнопки

❷ Загружает сценарий по нажатию кнопки Load (Загрузить)

❸ Выводит результаты по нажатию кнопки Inspect (Проверить)

На этой странице определены две кнопки ❶, предназначенные для запуска двух разных действий. По нажатию кнопки Load (Загрузить) с помощью функции `$.getScript()` ❷ динамически загружается файл `external.js`. Нажмите эту кнопку — и, как ожидалось, выполнится встроенная в файл инструкция и появится сообщение, показанное на рис. 10.6.

Рис. 10.6. Динамическая загрузка и оценка результатов файла сценария приводит к выполнению встроенного сценария

В результате нажатия кнопки Inspect (Проверить) вызывается ее обработчик события `click` ❸, который выполняет динамически загруженную функцию `someFunction()`, передавая ей значение динамически загруженной переменной `someVariable`. Если на экране появляется сообщение, показанное на рис. 10.7, то это свидетельствует о корректной загрузке и переменной, и функции.

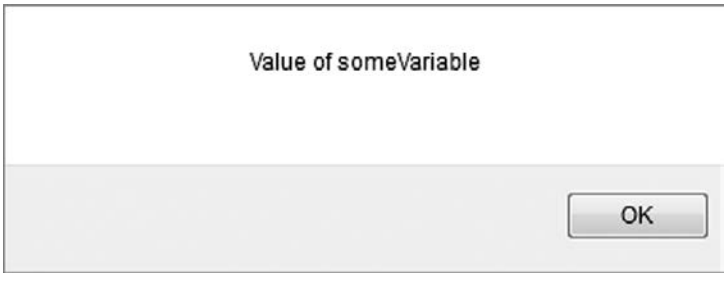


Рис. 10.7. Появление сообщения показывает, что динамическая функция загружена корректно, и корректно отображаемое значение показывает, что значение динамически загружено

Кроме того, что jQuery предоставляет описанные вспомогательные функции для выполнения запросов GET, она также позволяет делать запросы POST. Посмотрим, каким образом.

10.3.4. Создание запросов POST

Отдать предпочтение POST, а не GET можно по целому ряду причин. Во-первых, согласно спецификации протокола HTTP метод POST следует использовать для выполнения всех нетождественных запросов. То есть если запрос может вызвать изменения состояния на стороне сервера, то следует выбрать этот метод. С другой стороны, согласно общепринятой практике и соглашениям допускается применять его в случае, когда объем передаваемых данных слишком велик и его нельзя передать серверу в строке запроса URL, — ограничение зависит от используемого

браузера. Иногда серверный ресурс, с которым мы взаимодействуем, выполняет разные функции в зависимости от того, к какому методу обращается ваш запрос — GET или POST. Это только часть множества причин, почему вы можете захотеть выбрать запрос POST, а не GET.

Для случаев, когда вам необходимо выполнять запросы методом POST, jQuery предоставляет вспомогательную функцию `$.post()`. Она идентична функции `$.get()` за исключением использования метода протокола HTTP POST. По этой причине в описании синтаксиса данного метода мы не будем повторять значения всех параметров. Синтаксис функции `$.post()` следующий.

Синтаксис функции: $.post

`$.post(url[, [, data][, callback][, dataType])`

Отправляет запрос POST к серверу, применяя заданный URL и все параметры, передаваемые в теле запроса.

Возвращает

Экземпляр `jqXHR`.

jQuery 3: добавленная сигнатура

jQuery 3 добавляет новую сигнатуру для вспомогательной функции `$.post()`:

```
$.post([options])
```

Объект `options` может обладать многими свойствами. Чтобы узнать об этом больше, обратите особое внимание на описание вспомогательной функции `$.ajax()`, которая обсуждается ниже. Стоит отметить: свойство `method` объекта `options` будет автоматически установлено в положение `"POST"`.

Библиотека сама позаботится о передаче параметров в тело запроса (в противоположность формированию строки запроса) и соответственно установит метод HTTP.

Теперь вернемся к «Чулану обуви». Мы очень неплохо начали, но, чтобы купить пару ботинок, требуется намного больше, чем просто выбрать понравившуюся модель. Клиенты наверняка захотят выбрать нравящийся им цвет и, конечно, указать размер обуви. Используя эти дополнительные требования, мы продемонстрируем, как решить одну из проблем, о которой очень часто спрашивают на форумах, посвященных технологии Ajax, — и это...

10.3.5. Реализация каскадов раскрывающихся списков

Реализация каскадов раскрывающихся списков — когда содержимое каждого последующего раскрывающегося списка зависит от варианта, выбранного в предыдущем списке, — стала одним из наиболее часто используемых веб-шаблонов. В данном подразделе мы реализуем лучшее решение для нашей страницы «Чулан обуви», которое покажет, насколько просто можно это сделать с jQuery.

Мы уже видели, как легко выполняется загрузка содержимого раскрывающегося списка. Далее мы увидим, что для объединения нескольких раскрывающихся списков в каскад требуется приложить лишь немногим больше усилий.

Посмотрим, какие изменения необходимо совершить в следующей версии нашей страницы:

- добавить раскрывающиеся списки для выбора цвета и размера;
- после выбора модели раскрывающийся список выбора цвета должен заполняться цветами, доступными для выбранной модели;
- после выбора цвета раскрывающийся список выбора размера должен заполняться размерами, доступными для выбранной модели указанного цвета;
- раскрывающиеся списки всегда должны находиться в непротиворечивом состоянии: после выбора из списков должны удаляться пункты -пожалуйста, сделайте свой выбор- и необходимо обеспечить невозможность выбора пользователем недопустимого сочетания модели, цвета и размера на основе этих трех списков.

Кроме того, мы собираемся вернуться к использованию метода `load()`, но на сей раз принудительно будем заставлять его выполнять запросы GET, а не POST. Причина в том, что `load()` выглядит естественнее, когда требуется средствами Ajax организовать загрузку фрагментов разметки HTML.

Для начала рассмотрим новую HTML-разметку, которая определяет дополнительные раскрывающиеся списки. Новый контейнер с элементами управления содержит три раскрывающихся списка и метки к ним:

Выпадающее меню для цвета, изначально недоступное

Выпадающее меню для модели

```
<div id="selections-pane">
   <label for="boot-chooser-control">Модель обуви:</label>
   <select id="boot-chooser-control" name="model"></select>

   <label for="color-chooser-control">Цвет:</label>
   <select id="color-chooser-control" name="color" disabled></select>

   <label for="size-chooser-control">Размер:</label>
   <select id="size-chooser-control" name="size" disabled></select>
</div>
```

Выпадающее меню для размера, изначально недоступное

Предыдущий элемент модели `select` остается, но к нему присоединяются еще два: для цвета и размера, каждый из них первоначально пуст и недоступен (используется атрибут `disabled`).

Раскрывающийся список модели теперь должен выполнять двойную работу. Он не только должен продолжать подгружать и отображать данные о выбранной обуви после совершения выбора, но его обработчик изменений должен также заполнять и включать раскрывающийся список для выбора цвета со всеми доступными цветами для выбранной модели.

Теперь немного переделаем выбор элементов. Вам нужно использовать метод `load()`, а также принудительно применить GET, а не инициированный ранее POST. Чтобы `load()` осуществлял запрос GET, нужно передать строку, а не объект для указания данных параметров запроса. К счастью, с помощью jQuery не нужно создавать эту строку самому. Первая часть обработчика изменений для раскрывающегося списка модели переделывается так:

```
var $bootChooser = $('#boot-chooser-control');
var $colorChooser = $('#color-chooser-control');
var $sizeChooser = $('#size-chooser-control');

$bootChooser.change(function() {
   $('#product-detail-pane').load(
         'actions/fetch-product-details.php',
         $(this).serialize()
   );
   // Продолжение следует
});
```

Определяет переменные, используемые в коде

Предоставляет данные в виде строки запроса

С помощью метода `serialize()` вы создали строковое представление значения раскрывающегося списка с моделями обуви, чем вынудили метод `load()` инициировать запрос GET, как вам и хотелось.

Вторая задача, которую должен решить обработчик события `change`, заключается в загрузке допустимых значений в раскрывающийся список выбора цвета и его активизации. Рассмотрим дополнительный программный код из второй части обработчика:

```
$colorChooser.load(
       'actions/fetch-color-options.php',
       $(this).serialize(),
       function() {
          $(this).prop('disabled', false);
          $sizeChooser
                  .prop('disabled', true)
                  .html('');
       }
);
```

❶ Получает и загружает допустимые цвета

❷ Включает управление цветом

❸ Очищает и деактивизирует элемент выбора размера

Данный код должен казаться знакомым. Это еще один вариант использования метода `load()`, на сей раз загружаются данные со страницы `actions/fetch-color-options.php`, которая возвращает набор элементов `<option>`, представляющих цвета, доступные для выбранной модели ❶.

Вы также определяете функцию обратного вызова, которая будет выполняться в случае успешного выполнения запроса GET. В ней вы решаете две очень важные задачи. Во-первых, активизируете элемент выбора цвета ❷. Вызов `load()` вставляет элементы `<option>` в `<select>`, но последний после заполнения останется в неактивном состоянии, если его не активизировать вручную. Во-вторых, функция обратного вызова деактивизирует и очищает элемент выбора размера ❸. Зачем? (Задумайтесь на минутку и попробуйте догадаться сами.)

Хотя элемент выбора размера уже был деактивизирован и очищен, когда покупатель выбрал модель, что же могло случиться позже? Представьте: после выбора модели и цвета (это приведет к заполнению элемента выбора размера, как будет показано чуть ниже) покупатель изменит модель. Поскольку доступные размеры отображаются для определенной комбинации модели и цвета, список размеров, который отображался ранее, может не соответствовать действительности. Поэтому всякий раз, когда выбирается другая модель, необходимо очистить список доступных размеров и вернуть элемент выбора в начальное состояние.

Прежде чем празновать окончание работы, необходимо сделать еще кое-что. Все еще требуется обработать событие выбора цвета и использовать выбранную модель и цвет для загрузки данных в раскрывающийся список выбора размера. Реализация этой задачи выполняется по уже знакомому нам шаблону:

```
$colorChooser.change(function() {
  $sizeChooser.load(
          'actions/fetch-size-options.php',
          $colorChooser
                  .add($bootChooser)
                  .serialize(),
          function() {
            $(this).prop('disabled', false);
          }
  );
});
```

В обработчике события `change` выполняется попытка загрузить информацию о доступных размерах путем обращения к странице `actions/fetch-size-options.php`, которой передаются выбранные модель и цвет, после чего активизируется элемент управления выбором размера.

И здесь тоже необходимо сделать кое-что. После первоначального заполнения каждого раскрывающегося списка в нем присутствует элемент `option` с пустым значением и с текстом -выберите что-то-. Вы наверняка помните, что в предыдущих версиях этой страницы вы добавляли код, удаляющий данный элемент из раскрывающегося списка моделей после выбора.

Ну, вы могли бы добавить данный код в обработчики изменения события для раскрывающихся списков моделей и цветов и реализовать обработчик изменения события для раскрывающегося списка размеров (которого пока нет), куда также добавить этот код. Но попробуем поступить чуть хитрее.

Одна из особенностей модели событий, о которой часто забывают многие веб-разработчики, — наличие механизма *всплытия событий*. Авторы страниц зачастую все свое внимание концентрируют исключительно на целевых элементах, в которых возникают события, и забывают, что события всплывают вверх по дереву DOM, где можно предусмотреть их более обобщенную обработку, чем на уровне целевых элементов.

Если вы вспомните, что удаление элемента выбора с пустым значением из любого раскрывающегося списка выполняется совершенно одинаково, независимо от того, в каком из них возникло событие, то сможете избежать повторения одного

и того же программного кода в трех разных местах, подключив единственный обработчик к элементу, расположенному выше в дереве DOM, который будет перехватывать и обрабатывать события изменения. Этот прием должен напомнить об обсуждении делегирования событий в главе 6.

В структуре документа три раскрывающихся списка содержатся внутри элемента `<div>` с ID `selectionsPane`. Благодаря этому у вас имеется возможность удалять временный параметр для всех трех раскрывающихся списков в единственном обработчике, как показано ниже:

```
$('#selections-pane').change(function(event){
  $('[value=""]', event.target).remove();
});
```

Данный обработчик будет вызываться всякий раз при возникновении в каком-либо из раскрывающихся списков события `change` и будет удалять `option` с пустым значением из элемента, на который ссылается контекст функции (то есть из раскрывающегося списка, где возникло событие).

На этом мы завершаем работу над третьей фазой «Чулана обуви», в которую был добавлен каскад из трех раскрывающихся списков, как показано на рис. 10.8. Вы можете использовать подобные методы на любых страницах, где содержимое одного раскрывающегося списка зависит от вариантов, выбранных в других списках. Эту версию страницы можно найти на http://localhost[:8080]/chapter-10/phase.3.html.

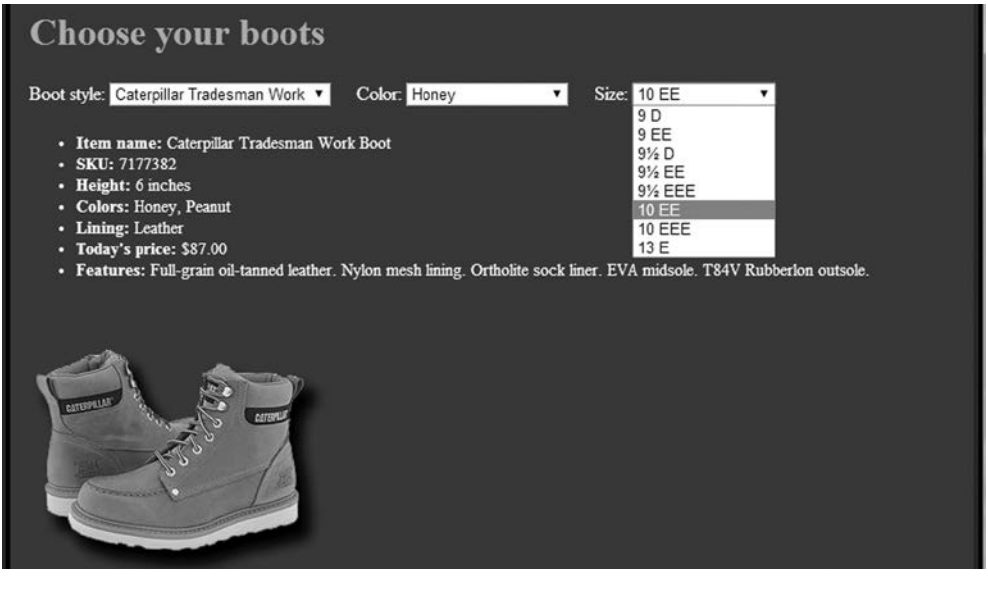


Рис. 10.8. Третья фаза «Чулана обуви» показывает, насколько просто реализовать каскад раскрывающихся списков

Полный код этой страницы приводится в листинге 10.7.

Листинг 10.7. «Чулан обуви», теперь с каскадом раскрывающихся списков!

```
<!DOCTYPE html>
<html>
  <head>
    <title>Чулан обуви - Фаза 3</title>
    <link rel="stylesheet" href="../css/main.css" />
    <link rel="stylesheet" href="../css/bootcloset.css" />
  </head>
  <body>
    <div id="banner"></div>
    <h1>Выберите вашу обувь</h1>
    <div>
    <div id="selections-pane">
      <label for="boot-chooser-control">Модель:</label>
      <select id="boot-chooser-control" name="model"></select>
      <label for="color-chooser-control">Цвет:</label>
      <select id="color-chooser-control" name="color" disabled></select>
      <label for="size-chooser-control">Размер:</label>
      <select id="size-chooser-control" name="size" disabled></select>
    </div>
    <div id="product-detail-pane"></div>
    <script src="../js/jquery-1.11.3.min.js"></script>
    <script>
      var $bootChooser = $('#boot-chooser-control');
      var $colorChooser = $('#color-chooser-control');
      var $sizeChooser = $('#size-chooser-control');
      $bootChooser
              .load('actions/fetch-boot-style-options.php')
              .change(function() {
                $('#product-detail-pane').load(
                  'actions/fetch-product-details.php',
                  $(this).serialize()
                );
                $colorChooser.load(
                        'actions/fetch-color-options.php',
                        $(this).serialize(),
                        function() {
                          $(this).prop('disabled', false);
                          $sizeChooser
                                  .prop('disabled', true)
                                  .html('');
                        }
                );
              });
      $colorChooser.change(function() {
        $sizeChooser.load(
              'actions/fetch-size-options.php',
```

```
            $colorChooser
                    .add($bootChooser)
                    .serialize(),
                      function() {
                        $(this).prop('disabled', false);
                      }
        );
      });
      $('#selections-pane').change(function(event){
        $('[value=""]', event.target).remove();
      });
    </script>
  </body>
</html>
```

Как видите, используя метод `load()` и различные функции библиотеки jQuery, выполняющие запросы GET и POST, можно в некоторой степени контролировать то, как инициируется запрос и способ получения информации о результатах запроса. Но для случаев, когда необходим полный контроль над запросами Ajax, jQuery предлагает средства, позволяющие вам делать запрос настолько подробным, насколько вы захотите этого сами.

10.4. Полное управление запросами Ajax

Функции и методы, с которыми мы познакомили, удобны во многих случаях, но часто бывает так, что требуется иметь в своих руках полное управление, до последней детали. Например, может понадобиться получать свежие данные каждый раз при выполнении запроса Ajax, то есть обходя кэш браузера. Использование методов нижнего уровня также может потребоваться, когда необходимо выполнить запрос Ajax, но его результат имеет значение только в случае, если он получен в определенный отрезок времени. Последний пример, который мы хотим привести: вы иногда можете получить результат вызова Ajax в определенном формате, например в виде обычного текста, но предпочитаете, чтобы он был преобразован в другой, например HTML или XML.

В данном разделе мы посмотрим, как jQuery обеспечивает эти возможности.

10.4.1. Выполнение запросов Ajax со всеми настройками

Для случаев, когда требуется иметь полный контроль над выполнением запросов Ajax, jQuery предоставляет универсальную вспомогательную функцию: `$.ajax()`. Прочие методы и функции библиотеки jQuery, основанные на технологии Ajax, в конечном счете для инициации запроса используют именно эту функцию. Ее синтаксис представлен ниже.

Синтаксис функции: $.ajax

```
$.ajax(url[, options])
$.ajax([options])
```

Осуществляет запрос Ajax, используя URL и переданные параметры для управления передачей запроса и обращения к функции обратного вызова. Во второй версии этой вспомогательной функции URL указывается в параметрах. Если не указаны параметры, то запрос отправляется на текущую страницу.

Параметры

`url` (Строка) Строка, содержащая URL, на который отправляется запрос.

`options` (Объект) Объект, свойства которого определяют параметры выполнения операции. Подробности приведены в табл. 10.2.

Возвращает

Экземпляр `jqXHR`.

Казалось бы, все просто? Но не спешите с выводами. Параметр `options` может определять широкий диапазон значений, которые могут быть использованы для настройки действий этих функций, включая URL, куда отправляется запрос. Эти параметры (в порядке убывания их важности и вероятности их использования) приведены в табл. 10.2.

Таблица 10.2. Параметры вспомогательной функции $.ajax()

Имя	Описание
url	(Строка) Строка, содержащая URL, на который отправится запрос. Если передана пустая строка, то запрос отправляется на текущий URL во время вызова метода
method	(Строка) Применяемый метод HTTP. Обычно POST или GET. Если он опущен, то по умолчанию используется GET. В версиях jQuery до 1.9.0 это же свойство называется type
data	(Строка|Объект|Массив) Определяет значения, которые будут переданы серверу. Если запрос выполняется с помощью метода GET, то эти данные передаются в виде строки запроса. Если POST, то данные передаются в теле запроса. В любом случае кодирование значений выполняется вспомогательной функцией $.ajax(). Этот параметр может быть строкой, которая будет использоваться как строка запроса или как тело ответа, объектом, свойства которого сериализованы, или массивом объектов, у которых свойства name и value определяют пары имя/значение параметров запроса
dataType	(Строка) В своей основной форме это ключевое слово, идентифицирующее тип данных, которые ожидается получить в ответе. Данное значение определяет, каким образом должны быть обработаны данные, если это потребуется, прежде чем будут переданы функции обратного вызова. Допустимы следующие значения: xml — текст ответа анализируется как документ XML, функции обратного вызова передается полученное дерево DOM XML; html — текст ответа передается функции обратного вызова без предварительной обработки. Все блоки <script> внутри полученного фрагмента HTML выполняются;

Продолжение ⇩

Таблица 10.2 (продолжение)

Имя	Описание
	json — текст ответа анализируется как строка JSON, функции обратного вызова передается полученный объект; jsonp — аналогично формату json, за исключением того, что допускает удаленное создание сценариев (предполагается, что сервер поддерживает данную возможность); script — текст ответа передается функции обратного вызова, но перед этим ответ обрабатывается как инструкция или инструкции JavaScript; text — предполагается, что ответ содержит обычный текст. Ресурс на сервере отвечает за установку соответствующего заголовка ответа content-type. Значение по умолчанию определяется jQuery в зависимости от полученного ответа и будет одним из xml, json, script или html. Значение этого параметра может быть также строкой значений, разделенных пробелами. В этом случае библиотека преобразует тип данных в другой. Например, если получен ответ в виде текста и вы хотите, чтобы он был в формате XML, напишите "text xml "
cache	(Логическое значение) Если имеет значение false, то ответ не будет кэшироваться браузером. Это работает корректно только с запросами HEAD и GET. По умолчанию принимает значение true, кроме случая, когда указанный параметр dataType — script или jsonp
context	(Объект|Элемент) Определяет объект или элемент DOM, который будет использоваться в качестве контекста для всех функций обратного вызова, ассоциированных с запросом. По умолчание контекстом является объект, представляющий настройки Ajax, используемые в вызове
timeout	(Число) Устанавливает предельное время ожидания ответа на запрос в миллисекундах. Время ожидания начинается при вызове ajax(). Если запрос не завершен в течение указанного времени, то его выполнение прерывается с вызовом функции обработки ошибок (если определена)
global	(Логическое значение) Если false, то запрещает выполнение глобальных событий Ajax. Это нестандартные события, которые могут вызываться на различных этапах или при некоторых условиях в процессе выполнения запроса Ajax. Мы подробно рассмотрим их в следующем подразделе. Если параметр опущен, то по умолчанию (true) выполнение глобальных событий разрешено
contentType	(Строка) Тип содержимого в запросе. Если опущен, то по умолчанию предполагается тип application/x-www-form-urlencoded; charset=UTF-8; этот же тип используется по умолчанию при отправке форм
success	(Функция|Массив) Функция или массив, вызываемые в случае, если код статуса в ответе сообщает об успехе. Тело ответа передается этой функции в виде первого параметра и оценивается в соответствии со значением параметра dataType. Вторым параметром является строка, содержащая значение статуса, — в данном случае всегда строка "success". Третий параметр предоставляет ссылку на экземпляр jqXHR
error	(Функция|Массив) Функция или массив, вызываемые в случае, если код статуса в ответе сообщает об ошибке. Функции передаются три аргумента: экземпляр jqXHR, строка сообщения о состоянии (в данном случае одно из следующих значений: "error", "timeout", "abort" или "parseerror") и необязательный объект-исключение, иногда возвращаемый экземпляром jqXHR, если таковой есть. Этот обработчик не вызывается для междоменных сценариев и междоменных запросов JSONP

Имя	Описание
complete	(Функция|Массив) Функция, вызываемая по завершении запроса. Передаются два аргумента: экземпляр jqXHR и строка сообщения о состоянии — "success", "error", "notmodified", "timeout", "abort" или "parseerror". Если указаны также функции обратного вызова для успешного или неудачного выполнения, то данная функция будет вызвана после них
beforeSend	(Функция) Функция, вызываемая перед инициацией запроса. Ей передается экземпляр jqXHR, и она может использоваться для установки дополнительных заголовков или выполнения других предварительных операций. Если эта функция вернет значение false, то выполнение запроса будет прервано
async	(Логическое значение) Если задано значение false, то запрос выполняется как синхронный. По умолчанию задано значение true и выполняется асинхронный запрос. Междоменные запросы и dataType: "jsonp" не поддерживают синхронные операции
processData	(Логическое значение) Если установлено false, то кодирование передаваемых данных в формат URL не производится. По умолчанию данные кодируются в формат URL, применяемый при передаче запросов типа application/x-www-form-urlencoded
contents	(Объект) Объект пар «строка/регулярное выражение», определяющее, как jQuery будет анализировать ответ, на основе типа контента
converters	(Объект) Объект, содержащий преобразователи dataType-в-dataType. Каждое значение преобразователя — это функция, возвращающая преобразованное значение ответа
crossDomain	(Логическое значение) Устанавливается в true для отправки crossDomain-запроса в том же домене. По умолчанию значение false для внутридоменных запросов и true — для междоменных запросов
headers	(Объект) Объект дополнительных пар ключ/значение заголовка, которые отправляются с запросами. По умолчанию значением является пустой объект
dataFilter	(Функция) Функция обратного вызова, которая вызывается для фильтрации данных в ответе. Этой функции передается необработанный ответ и значение параметра dataType. Предполагается, что она вернет «очищенные» данные
ifModified	(Логическое значение) При заданном значении true запрос считается успешным, только если содержимое ответа не изменилось с момента последнего запроса, в соответствии с заголовком Last-Modified. Если опущен, то заголовок не проверяется. По умолчанию имеет значение false
jsonp	(Строка) Определяет имя параметра запроса, который будет подставлен в запрос типа jsonp, в параметр с именем callback
jsonpCallback	(Строка|Функция) Определяет имя функции обратного вызова для запроса JSONP. Это значение будет использовано вместо случайно сгенерированного jQuery имени
username	(Строка) Имя пользователя, которое будет использоваться в случае запроса HTTP на аутентификацию
password	(Строка) Пароль, который будет использоваться в случае запроса HTTP на аутентификацию
scriptCharset	(Строка) Кодировка символов для использования в запросах типа script и jsonp, когда на стороне сервера и на стороне клиента используются различные кодировки символов для представления контента

Продолжение ⇩

Таблица 10.2 (продолжение)

Имя	Описание
xhr	(Функция) Функция, создающая пользовательскую реализацию экземпляра XHR
xhrFields	(Объект) Объект пар «имя — значение» для установки на нативный объект XHR. По умолчанию — пустой объект
accepts	(Объект) Тип контента, передаваемого в заголовке запроса, который сообщает серверу, какого рода он будет принимать ответ. По умолчанию его значение зависит от типа данных dataType
mimeType	(Строка) Тип mime для переопределения типа XHR mime
traditional	(Логическое значение) Если имеет значение true, то используется традиционный алгоритм сериализации параметров. См. описание функции $.param() в главе 9

Не пугайтесь, увидев этот длинный список. Мы знаем, что здесь довольно много параметров, но, во-первых, не нужно запоминать их все (всегда можно прочесть в данной книге или официальной документации), а во-вторых, маловероятно, что в каждом отдельном запросе будет использоваться сразу множество параметров.

Что такое JSONP

JSON представляет собой легкий и часто используемый формат обмена данными. Сайты, как правило, получают данные в таком формате, выполняя запросы Ajax с помощью объекта XHR. Этот механизм придерживается политики одного источника, который предписывает, что передача определенных типов данных должна быть ограничена и происходить только в случае, если домен целевого ресурса идентичен странице, сделавшей запрос. Чтобы обойти это ограничение, в декабре 2005 года Боб Ипполито предложил новый механизм JSONP, который был опубликован в его статье «Удаленный JSON — JSONP» (http://bob.ippoli.to/archives/2005/12/05/remote-json-jsonp/).

JSONP (аббревиатура от JSON with padding — «JSON с дополнением») работает путем создания элемента `script` (либо в HTML-разметке, либо вставленного в DOM с помощью JavaScript) со ссылкой на ресурс, возвращающий данные в формате JSON, обернутом функцией, объявленной на странице, выполняющей запрос, имя которого предоставляется элементом `script`. Как правило, имя функции передается с помощью параметра `callback`. Например, можно создать следующий элемент `script`:

```
<script src="http://www.example.com/data?callback=myFunction"></script>
```

В этом случае `myFunction()` — функция, определенная на странице, выполняющая запрос, который должен обрабатывать возвращаемый JSON. Сервер, который может работать с такими запросами, обычно отвечает, как показано ниже:

```
myFunction({"name": "jQuery in Action"});
```

Это приводит к вызову функции `myFunction()` с данными, возвращаемыми сервером, которые переданы в качестве аргумента.

Чтобы узнать больше о JSONP, посетите сайт www.json-p.org.

«И никаких примеров использования `$.ajax()`?» — удивитесь вы. Не беспокойтесь: следующая глава будет посвящена созданию проектов, задействующих возможности Ajax.

Иногда удобно установить значения по умолчанию для параметров, представленных в табл. 10.2 (см. выше), для страниц, где планируется сделать большое количество запросов. Узнаем, как это можно сделать.

10.4.2. Настройка запросов, используемых по умолчанию

jQuery предоставляет способ определить значения свойств Ajax по умолчанию. Это поможет упростить страницы, выполняющие множество однотипных запросов Ajax. Функция для установки значений по умолчанию называется `$.ajaxSetup()`.

Синтаксис функции: $.ajaxSetup

`$.ajaxSetup(options)`

Устанавливает переданный набор параметров как значения по умолчанию для всех последующих вызовов функции `$.ajax()` или производных методов, таких как `$.get()` и `$.post()`, включая те, которые выполняются сторонними библиотеками или плагинами.

Параметры

`options` (Объект) Объект, свойства которого определяют значения свойств Ajax по умолчанию. Это те же самые свойства, используемые функцией `$.ajax()` и описанные в табл. 10.2.

Данная функция не должна использоваться для определения обработчиков `success`, `error` и `completion`. (Как устанавливать эти обработчики с помощью альтернативных средств, будет показано в следующем подразделе.)

Возвращает

`undefined`.

Эту функцию можно использовать в любом месте сценария для всех последующих вызовов функции `$.ajax()`. Данный метод нужно применять с осторожностью, ведь он может изменить поведение плагинов и других библиотек, которые вы задействуете на своих веб-страницах, осуществляющих вызовы Ajax помощью `$.ajax()` и аналогичных методов.

ПРИМЕЧАНИЕ

Набор значений по умолчанию, задаваемый этой функцией, не применяется к методу load(). Для вспомогательных функций $.get() и $.post() нельзя переопределить используемый ими метод HTTP. Например, установка параметра type в значение GET не приведет к тому, что функция $.post() будет использовать метод HTTP GET.

Предположим, вы настраиваете страницу, где для большей части запросов (выполняемых функциями, отличными от метода `load()`) хотите установить некоторые значения по умолчанию, чтобы не определять их при каждом вызове. Тогда первую инструкцию в элементе `script` можно записать так:

```
$.ajaxSetup({
  type: 'POST',
  timeout: 5000,
  dataType: 'html'
});
```

Она гарантирует следующее: каждый последующий запрос (кроме уже упоминавшихся) по умолчанию будет применять эти значения, если не переопределить их явно в параметрах, передаваемых используемым вспомогательным функциям Ajax. В частности, вы устанавливаете, что все запросы будут POST, максимальное время обработки запроса — пять секунд (5000 миллисекунд) и ожидается ответ в формате HTML.

А теперь поговорим об упоминавшихся выше *глобальных событиях*, которыми управляет параметр `global`.

10.4.3. Обработка событий Ajax

В процессе выполнения запросов Ajax jQuery вызывает последовательность пользовательских событий, для обработки которых можно устанавливать обработчики, чтобы они информировали о различных этапах выполнения запроса и дали возможность предпринимать какие-либо действия на различных этапах. jQuery подразделяет эти события на локальные и глобальные.

Локальные события обрабатываются функциями обратного вызова, которые можно устанавливать непосредственно, с помощью параметров `beforeSend`, `success`, `error` и `complete`, передаваемых функции `$.ajax()`, или опосредованно, передавая функции обратного вызова соответствующим методам (которые, в свою очередь, вызывают функцию `$.ajax()` для фактического выполнения запроса). Вы совершаете обработку локальных событий, даже не подозревая об этом, всякий раз, когда передаете функцию обратного вызова той или иной функции Ajax в jQuery.

Глобальные события — события, которые выполняются для любого запроса Ajax на веб-странице. Вы можете установить обработчики события для них с помощью метода `on()` (как любое другое событие), используемого в `document` (прикрепление их к любому другому элементу не будет работать). Глобальные события, многие из которых являются лишь отражением локальных событий, — это `ajaxStart`, `ajaxSend`, `ajaxSuccess`, `ajaxError`, `ajaxStop` и `ajaxComplete`.

Прикрепленные обработчики получают три параметра: экземпляр `jQuery.Event`, экземпляр `jqXHR` и объект, содержащий параметры, переданные `$.ajax()`. Исключения в этом списке параметров отмечены в табл. 10.3, где перечислены события Ajax в порядке их следования в процессе выполнения запроса.

Таблица 10.3. Типы событий Ajax в jQuery

Имя события	Тип	Описание
ajaxStart	Глобальное	Выполняется при запуске запроса Ajax, если к этому моменту не было других активных запросов. При одновременном выполнении нескольких запросов это событие выполняется только для первого запроса. Передается только экземпляр jQuery.Event
beforeSend	Локальное	Вызывается перед отправкой запроса для предоставления возможности внести какие-либо изменения в экземпляр XHR. Вы можете отменить запрос, вернув значение false
ajaxSend	Глобальное	Выполняется перед отправкой запроса для разрешения модификации экземпляра XHR
success	Локальное	Вызывается после того, как запрос получает успешный ответ
ajaxSuccess	Глобальное	Выполняется после того, как запрос вернется от сервера с успешным ответом
error	Локальное	Вызывается после того, как запрос вернет ответ с сообщением об ошибке
ajaxError	Глобальное	Выполняется после того, как запрос вернется от сервера, и ответ будет содержать код статуса, свидетельствующий об ошибке. Четвертым необязательным параметром передается ссылка на ошибку, если таковая имеется
complete	Локальное	Вызывается после того, как запрос вернется от сервера, независимо от кода состояния. Эта функция вызывается даже для синхронных запросов
ajaxComplete	Глобальное	Выполняется после того, как запрос вернется от сервера, независимо от кода состояния. Эта функция обратного вызова вызывается даже для синхронных запросов
ajaxStop	Глобальное	Выполняется после того, как завершатся все этапы обработки запроса и при отсутствии других параллельно выполняемых активных запросов. Передается только экземпляр jQuery.Event

Чтобы убедиться, что все понятно, еще раз обратим внимание: локальные события представляют собой функции обратного вызова, передающиеся функции `$.ajax()` (и родственным ей), а глобальные — это пользовательские события, которые могут обрабатываться устанавливаемыми обработчиками (в `document`), как и любые другие типы событий.

В табл. 10.3 мы сообщали: `ajaxStart` и `ajaxStop` получают экземпляр `jQuery.Event` в качестве единственного параметра. У данного параметра нет реального применения, так что этот аспект не приведен в официальной документации. Но мы все еще хотим о нем сообщить для большей точности (и еще потому, что мы будем использовать его в следующем демонстрационном примере). Прочитать больше на эту тему можно на https://github.com/jquery/api.jquery.com/issues/478.

Вдобавок к использованию `on()` для создания обработчиков событий jQuery предоставляет несколько удобных функций для установки обработчиков, синтаксис которых выглядит следующим образом.

Синтаксис функций: установка обработчиков глобальных событий Ajax

```
ajaxComplete(callback)
ajaxError(callback)
ajaxSend(callback)
ajaxStart(callback)
ajaxStop(callback)
ajaxSuccess(callback)
```

Подключает переданную функцию ко всем элементам набора для обработки события Ajax, соответствующего имени метода.

Параметры

`callback` (Функция) Функция, которая будет установлена как обработчик события Ajax. Через контекст функции (`this`) передается элемент DOM, к которому подключен обработчик. Параметры, подлежащие передаче, перечислены в табл. 10.3.

Возвращает

Коллекцию jQuery.

Рассмотрим небольшой пример, демонстрирующий использование некоторых из этих методов для отслеживания хода выполнения запросов Ajax. Тестовая страница (слишком простая, чтобы назвать ее лабораторной) изображена на рис. 10.9 и доступна на http://localhost[:8080]/chapter-10/ajax.events.html.

Рис. 10.9. Первоначальное отображение страницы, которую мы будем использовать для исследования событий Ajax в jQuery, запуская несколько событий и наблюдая работу обработчиков

На данной странице мы определили три элемента управления: поле счетчика `count`, кнопки Good request (Успешный запрос) и Bad request (Ошибочный запрос). Эти кнопки выполняют множество запросов, количество которых определяется полем счетчика `count`. Кнопка Good request (Успешный запрос) выполняет запросы к существующему ресурсу, а Bad request (Ошибочный запрос) — к ошибочному ресурсу, что приводит к неудачному выполнению этих запросов.

В коде этой страницы устанавливаются несколько обработчиков событий, как показано ниже:

```
var $log = $('#log');
$(document).on(
  'ajaxStart ajaxStop ajaxSend ajaxSuccess ajaxError ajaxComplete',
  function(event) {
    $log.text($log.text() + event.type + '\n');
  }
);
```

Данная инструкция устанавливает обработчик на объект `document` для каждого типа событий Ajax. Обработчик выводит сообщение, показывающий тип события, которое было инициализировано в элементе `textarea`, содержащем `log` в качестве ID.

Оставив значение счетчика запросов равным `1`, нажмите кнопку Good request (Успешный запрос) и понаблюдайте за результатами. Вы увидите, что все события Ajax выполняются в том порядке, в каком они перечислены в табл. 10.3. Но чтобы понять отличительные особенности событий `ajaxStart` и `ajaxStop`, установите счетчик равным `2` и нажмите кнопку Good request (Успешный запрос). Вы увидите результаты, показанные на рис. 10.10.

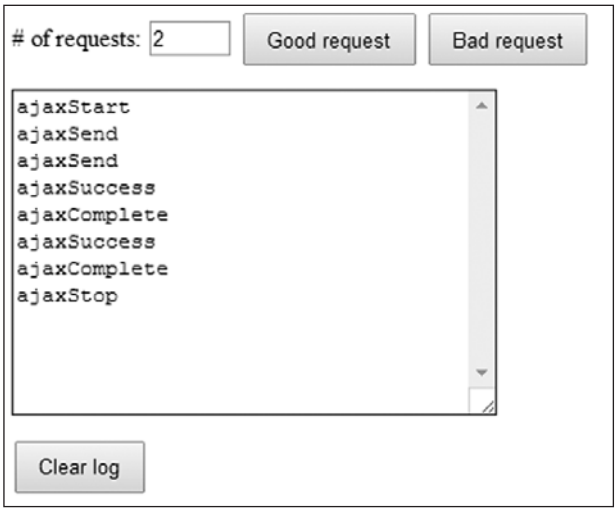


Рис. 10.10. Когда активны несколько запросов, события ajaxStart и ajaxStop вызываются для всего набора запросов, а не для каждого из них

Здесь видно, что при наличии нескольких активных запросов события `ajaxStart` и `ajaxStop` вызываются только один раз и для всего набора параллельно выполняемых запросов, тогда как другие события вызываются для каждого отдельного запроса.

Теперь попробуйте нажать кнопку Bad request (Ошибочный запрос), чтобы сгенерировать недопустимый запрос, и понаблюдайте за поведением обработчика. Вы получите результат, показанный на рис. 10.11, который доказывает, что в этот раз запущено событие `ajaxError`.

Рис. 10.11. Результат ошибочного запроса показывает, что вызывается обработчик ajaxError

Как вы уже успели заметить, функция `$.ajax()` очень эффективна благодаря своей гибкости, но бывают моменты (нечасто, если честно), когда хочется сделать еще больше. Например, может понадобиться обработать запросы на основе некоторых вариантов или изменения существующих до того, как был сделан запрос, либо управлять передачей данных вызова Ajax. Возможен такой вариант использования, который предполагает предотвращение вызова Ajax в некоторые домены (подробно обсудим это в следующем подразделе). Посмотрим, что предлагает jQuery в таких ситуациях.

10.4.4. Продвинутые вспомогательные функции Ajax

Помимо всех методов и вспомогательных функций, которые мы обсуждали до сих пор, у jQuery есть и другие «фишки». Маловероятно, что вы часто будете располагать возможностью увидеть эти две вспомогательные функции в действии, но, поскольку они все же существуют, мы хотим познакомить с ними. Встречайте: `$.ajaxPrefilter()` и `$.ajaxTransport()`!

Функция `$.ajaxPrefilter()` может быть использована для предотвращения запросов Ajax, основанных на некоторых пользовательских установленных параметрах при вызове `$.ajax()`. Ее синтаксис выглядит следующим образом.

Синтаксис функции: $.ajaxPrefilter

`$.ajaxPrefilter([dataTypes,] callback)`

Управляет пользовательскими параметрами Ajax или модифицирует существующие параметры перед отправлением запроса и перед обработкой последующими вызовами `$.ajax()`.

Параметры

`dataTypes` (Строка) Необязательная строка, содержащая один или несколько разделенных пробелами `dataType`, как описано для функции `$.ajax()` в табл. 10.2. Если этот параметр передан, то обработчик вызывается только при соответствии `dataTypes` запроса.

`callback` (Функция) Функция для установки значений по умолчанию для будущих запросов Ajax. Принимает три параметра: `options`, содержащие параметры запроса, `originalOptions`, который хранит параметры, предоставленные вызову `$.ajax()` (без значений по умолчанию из `ajaxSettings`), и `jqXHR`, являющийся объектом `jqXHR` объекта.

Возвращает

`undefined`.

В качестве конкретного примера использования данной функции представьте: вы хотите отменить все запросы типа XML, направленные на определенный набор доменов. Вы можете сделать это, так как знаете, что они никогда не будут выполняться успешно.

Чтобы достичь цели, можно написать код, подобный следующему, доступному также в файле http://localhost:8080/chapter-10/$.ajaxPrefilter.html, и в JS Bin (http://jsbin.com/bitiv/edit?js,console):

```
$.ajaxPrefilter('xml', function(options, originalOptions, jqXHR) {
   if ($.inArray(options.url,
      originalOptions.deniedDomains) !== -1) {
      console.log('Запрос Ajax на ' + options.url + ' отменен');
      jqXHR.abort();
   } else {
      console.log('Запрос Ajax выполнен');
   }
});

$.ajax(
   'http://www.google.com',
   {
      dataType: 'xml',
      deniedDomains: [
         'http://www.google.com',
         'http://www.manning.com'
      ]
   }
);
```

Если домен не разрешен, то запрос отменяется

Осуществляется запрос Ajax

Префильтрует запросы, основываясь на указанном dataType и на наборе недопустимых доменов

Применение данной функции не ограничивается изменением поведения вызова Ajax на основе установленных параметров. Она также может быть использована в тех случаях, когда нужно переадресовывать запрос от изначального `dataType` к какому-нибудь другому, что достигается путем возвращения нужного вам `dataType`.

Другая не очень известная функция, которую мы хотим здесь упомянуть, — это `$.ajaxTransport()`. Это функция низкого уровня, что позволяет взять под свое управление передачу функцией `$.ajax()` запрашиваемых данных. Ее синтаксис показан ниже.

Синтаксис функции $.ajaxTransport

`$.ajaxTransport([dataType,] callback)`

Создает объект, который обрабатывает фактическую передачу данных Ajax.

Параметры

`dataType` (Строка) Необязательная строка, содержащая используемый тип данных. Если этот параметр передан, то обработчик вызывается только при совпадении `dataType` запроса.

`callback` (Функция) Функция для возврата нового транспортного объекта, который будет использоваться с предоставленным `dataType`. Принимает три параметра: `options`, содержащий параметры запроса, `originalOptions`, хранящий параметры, предоставленные вызову `$.ajax()` (без значения по умолчанию из `ajaxSettings`), и `jqXHR`, который является объектом `jqXHR` запроса.

Возвращает

`undefined`.

Функция `callback` этого метода должна возвращать новый транспортный объект — объект JavaScript, поддерживающий два метода: `send()` и `abort()`, которые используются внутренне с помощью `$.ajax()`.

Функция `send()` принимает два параметра: `headers` и `completeCallback`. Первая из них является объектом пар «ключ — значение» заголовков запроса, которые может передать транспорт, если он их поддерживает, а второй — это обратный вызов, используемый для уведомления `$.ajax()` о завершении запроса.

На этой последней и немного сложной вспомогательной функции завершим наш обзор методов и функций, которые библиотека предоставляет для взаимодействия с Ajax. Приведенные примеры могут послужить хорошим началом, чтобы с головой окунуться в эту тему. Тем не менее вы, дорогой читатель, заслуживаете еще большего.

Цель следующей главы состоит в том, чтобы показать реальный пример, использующий все возможности Ajax для решения распространенной проблемы, с которой вы можете столкнуться, — и не исключено, что уже сталкивались.

10.5. Резюме

Технология Ajax — это ключевая составляющая современных веб-приложений, и jQuery не остается в стороне, предлагая широкий спектр инструментов для работы с ней.

Метод `load()` предоставляет простой способ загрузки фрагментов HTML в элементы DOM, получая контент со стороны сервера и превращая его в контент с любым набором элементов. Выбор метода POST или GET определяется типом предоставленного параметра `data`.

Когда требуется применить метод GET, jQuery предоставляет вспомогательные функции `$.get()` и `$.getJSON()`. Последняя удобна, когда сервер возвращает данные в формате JSON. Принудительно использовать метод POST позволяет вспомогательная функция `$.post()`.

Если требуется максимальная гибкость, то можно воспользоваться вспомогательной функцией `$.ajax()`, которая, обладая богатым набором параметров, позволит управлять большинством характеристик Ajax-запросов. Все остальные функции Ajax, реализованные в jQuery, обращаются к этой функции.

Для снижения трудоемкости управления параметрами библиотека jQuery предлагает вспомогательную функцию `$.ajaxSetup()`, позволяющую устанавливать значения по умолчанию для любых параметров, часто используемых функцией `$.ajax()` (и всеми остальными функциями поддержки Ajax, прибегающими к услугам `$.ajax()`).

Заканчивая рассмотрение функциональных возможностей Ajax, заметим, что jQuery позволяет отслеживать ход выполнения запросов Ajax, вызывая события на разных этапах их выполнения, и устанавливать обработчики этих событий. Для подключения обработчиков можно использовать метод `on()` или удобные методы установки обработчиков конкретных событий, такие как `ajaxStart()`, `ajaxSend()`, `ajaxSuccess()`, `ajaxError()`, `ajaxComplete()` и `ajaxStop()`.

Благодаря впечатляющей коллекции инструментов Ajax, которыми вы можете вооружиться, легко реализовать возможности полнофункциональных веб-приложений. Помня об этом, займемся реальным демонстрационным примером.

11 Демонстрационный пример: форма контакта, основанная на технологии Ajax

В этой главе:

- эффекты с jQuery;
- вспомогательные функции jQuery;
- создание запросов Ajax;
- создание удобной формы.

В предыдущей главе мы рассмотрели несколько тем, не относящихся к основам jQuery. Помимо обсуждения методов, вспомогательных функций и флагов, мы показали множество фрагментов кода, демонстрационных и лабораторных страниц. Все эти примеры должны были придать вам больше уверенности в работе с аргументами.

В главе 7 мы разработали демо, чтобы показать эффективность jQuery на реальном примере. Было использовано много методов и приемов, которые вы уже выучили к тому моменту. Затем мы предоставили больше дополнительных тем, таких как эффекты и анимации, вспомогательные функции и, что еще важнее, Ajax. Последнее — это ключевое понятие, а также замечательная технология, удобная для применения на ваших веб-страницах.

В этой главе мы коснемся другой реальной задачи, с которой многие из вас рано или поздно столкнутся, — создание формы контактов. Полагаясь на приобретенные знания, вы создадите не просто полностью работающую форму контактов, но еще и такую, которой не понадобится обновление страницы, чтобы проинформировать пользователя о неудаче или успехе при отправке сообщения. Равно как и в предыдущей главе, в данном демонстрационном примере вы будете использовать PHP как язык для разработки на серверной стороне. Если вы не знаете о PHP, то не беспокойтесь. Код будет настолько простым и хорошо описанным, что не составит труда понять его.

11.1. Функциональные возможности проекта

Прежде чем углубиться непосредственно в разработку вашего проекта, рассмотрим требования. В демонстрационном примере нужны только две страницы: одна будет содержать форму (назовем ее `index.html`), а вторая — бизнес-логику на стороне сервера (которую мы назовем `contact.php`). Можете поэкспериментировать с данным примером, найдя его в папке `chapter-11` с исходным кодом, предоставленным с книгой.

Чтобы пример был максимально простым, но одновременно интересным, создадим форму, которая содержит четыре поля: полное имя, электронный адрес, заголовок сообщения и само сообщение. Сделаем все эти поля обязательными для заполнения. Наконец, нужно, чтобы адрес электронной почты был верно отформатированным, а остальные поля содержали хотя бы четыре символа.

Форма будет высокоинтерактивной и станет проверять, соответствуют ли все поля установленным ограничениям без необходимости обновлять страницу. Для достижения этой цели будут использованы способы, которые вы уже изучили в предыдущих главах, и в особенности понятия, описанные в посвященной Ajax главе, связанные с отправкой запросов к серверу.

Проверка данных, введенных пользователем, будет осуществляться всякий раз при нажатии им кнопки Submit (Отправить). Кроме того, проверка будет выполняться каждый раз, когда поле теряет фокус. Если значение не будет допустимым, то вы будете предупреждать пользователя с помощью информационного сообщения под полем.

Для чего нужны проверки серверной стороны

Некоторые из вас могут поинтересоваться, почему мы решили задействовать проверки серверной стороны вместо выполнения задачи на клиентской стороне, используя только JavaScript. Причина в том, что каждая проверка, проводимая JavaScript, ненадежна и небезопасна, так как пользователь может просто отключить JavaScript. Таким образом, вы в конце концов позволите пользователям отправить недействительные или потенциально опасные данные или будете вынуждены делать такую же проверку и на серверной стороне. Чтобы избежать этих проблем, вы будете применять проверку на стороне сервера, оставляя страницу интерактивной с помощью Ajax.

Рисунок 11.1 показывает пример описанного функционала.

Рис. 11.1. Пример недействительного поля и показанное сообщение об ошибке

Помимо этой обратной связи после отправки пользователем формы, вы покажете ему диалоговое окно с сообщением либо об ошибке (рис. 11.2, *а*), либо об успешной отправке (см. рис. 11.2, *б*).

а *б*

Рис. 11.2. Диалоговое окно показывается, когда: *а* — форма содержит одну или несколько ошибок; *б* — все поля форм действительны и сообщение успешно отправлено. В сообщение об успешной отправке включено имя отправителя (в данном случае Аурелио де Роза)

Показанные сообщения будут возвращены страницей PHP благодаря использованию объекта JSON и введены в диалог с помощью jQuery. Структура возвращаемого JSON показана на рис. 11.3.

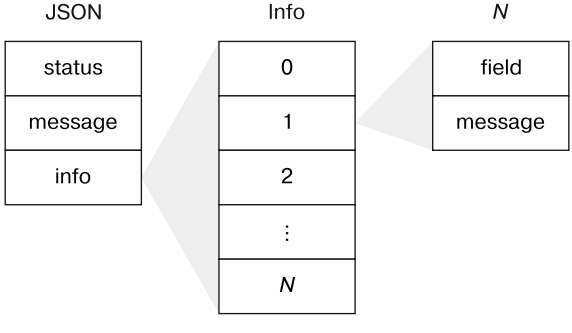


Рис. 11.3. Структура объекта JSON, возвращенного страницей PHP, которая проверяет введенные пользователем данные и отправляет сообщения по электронной почте

Структура объекта JSON довольно проста. Она состоит из трех свойств: `status`, `message` и `info`. Свойство `status` — строка, используемая для указания, все ли написанные пользователем значения корректны. Если да, то значение должно быть `success`, в противном случае — `error`. Свойство `message` содержит строку, которую нужно показать в диалоговом окне. В случае проверки отдельной строки, если значение последней действительно, сообщение будет пустой строкой. Свойство `info` предоставляет массив, содержащий объекты, связанные с недействительными полями формы. Каждый из этих объектов выставляет два поля: `field` и `message`. Задача `field` — указать имя поля, содержащего ошибку. Роль `message` совпадает

с описанной выше, но в этот раз она используется для предоставления информации о конкретной ошибке недействительного поля.

Теперь перейдем к следующему шагу — разметке формы.

11.2. Создание разметки

В предыдущем разделе мы обсудили ограничения, которые нужно будет применить к полям формы. Эти ограничения могут быть установлены с использованием новых атрибутов и типов HTML5. Например, чтобы получить поле электронного адреса, которое будет хорошо отформатировано и обязательно к заполнению, можно применить следующий `input` в форме:

```
<input type="email" name="email" id="email" required />
```

В современных браузерах, как только пользователь отправит форму, данный `input` запустит нужные вам проверки. К сожалению, старые браузеры, такие как Internet Explorer 9 и ниже, не распознают атрибут типа `email` и атрибут `required`. Если что-то не распознается, то поведение браузера по умолчанию — проигнорировать это. В данном случае это подобно тому, как если бы вы определили `type="text"` и не указали атрибут `required`. Так что с устаревшими браузерами вы не сможете решить описанную проблему, просто указав атрибуты.

Если вы хотите применить эти новые атрибуты и типы HTML5 и затем вернетесь обратно к элементам управления для старых браузеров, то можете прибегнуть к методу, который изучили в разделе 4.1:

```
if (!('required' in document.createElement('input'))) {
  // Установите свои элементы управления
}
```

В данном примере, чтобы все максимально упростить, вы будете делать вид, что забыли HTML5. Вы всегда будете использовать собственные элементы управления и избегать применения атрибутов HTML5. С учетом этих соображений посмотрим на код формы, показанный в листинге 11.1.

Листинг 11.1. Разметка формы контактов

```
<form id="contact-form" name="contact-form" class="box" method="post"
      action="contact.php">
  <div class="form-field">
      <label for="name">Полное имя:</label>
      <input name="name" id="name" />
      <span class="error"></span>
  </div>
  <div class="form-field">
      <label for="email">Электронный адрес:</label>
      <input name="email" id="email" />
      <span class="error"></span>
  </div>
```

```
  <div class="form-field">
      <label for="subject">Заголовок:</label>
      <input name="subject" id="subject" />
      <span class="error"></span>
  </div>
  <div class="form-field">
      <label for="message">Сообщение:</label>
      <textarea name="message" id="message"></textarea>
      <span class="error"></span>
  </div>
  <input type="submit" value="Отправить"/>
  <input type="reset" value="Отменить"/>
</form>
```

Как видите, каждое поле состоит из трех элементов: `label`, `input` (`textarea` для сообщения) и `span`. Элемент `label` используется для указания имени поля, у него есть атрибут `for`, установленный для улучшения доступности формы. Элементы `input` и `textarea` позволяют пользователю ввести необходимую информацию. Наконец, `span` служит для демонстрации обратной связи, проиллюстрированной выше (см. рис. 11.1). Помимо этих элементов, у вас есть кнопки Submit (Отправить) и Reset (Сбросить) внизу формы.

Контактная форма — не единственный компонент страницы. Вам также понадобится диалоговое окно для отображения сообщений. Разметка последних довольно проста, поскольку ей нужны только заголовок, абзац и кнопка для закрытия диалога. Эти три элемента обернуты в контейнер, так что можно рассматривать их как уникальный компонент.

HTML-код данного диалога таков:

```
<div class="dialog-box">
  <h2 class="title"></h2>
  <p class="message"></p>
  <button>OK</button>
</div>
```

Указанным фрагментом мы завершаем обзор HTML-кода `index.html`. Если вы запустите демонстрационный пример в текущем виде, то он не будет ни интерактивным, ни сколько-нибудь полезным, поскольку не делает вообще ничего.

Исправим это, добавив код на серверной стороне. Не беспокойтесь, если не понимаете PHP; мы осветим только ключевые моменты.

11.3. Реализация серверной части на PHP

В зоне ответственности серверной части вашего проекта два аспекта: проверка введенных пользователем данных и отправка электронной почты. Первый из них вам наиболее интересен, поскольку, основываясь на спецификации проекта, должны иметь дело с двумя разными случаями. Первый представляет собой частичный запрос, происходящий в результате потери фокуса поля. Второй — запрос, содержащий значения всех полей, происходящий от нажатия кнопки Submit (Отправить).

Чтобы различить два этих случая, вы создадите пользовательский параметр, названный `partial`, для частичных запросов (расскажем более подробно, когда будем обсуждать код JavaScript). В таких ситуациях страница PHP должна пропустить все проверки, кроме относящейся к полю, с которого убрали фокус. Затем она должна вернуть объект JSON в соответствии с результатами теста. Данный результат сохранится в переменной, называемой `$result` (такое вот у нас воображение!), и вернется (с применением конструкции языка `echo`) посредством функции PHP, называемой `json_encode()`, как показано здесь:

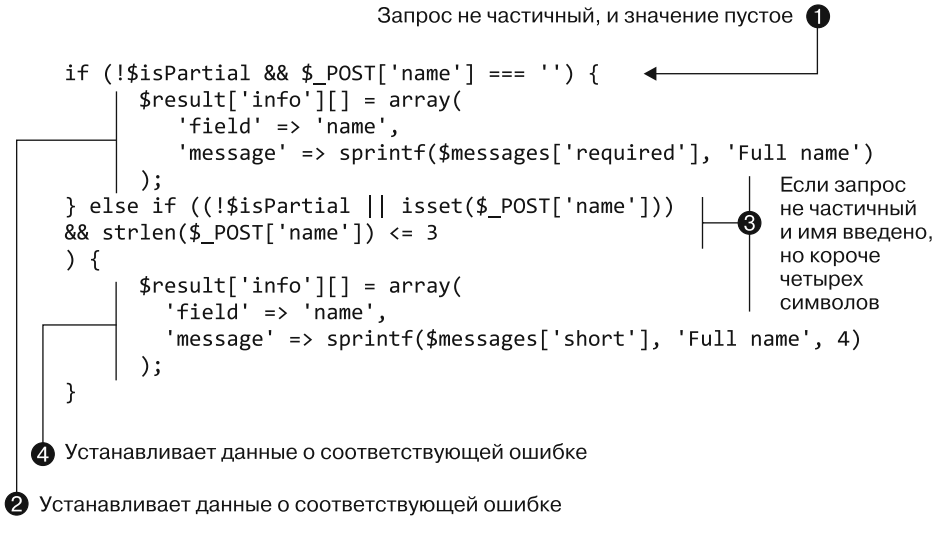
```
echo json_encode($result);
```

Чтобы отличить частичный запрос от полного, используйте переменную, названную `$isPartial`, значение которой устанавливается так:

```
$isPartial = empty($_POST['partial']) ? false : true;
```

Чтобы помочь понять код серверной стороны, проанализируем часть кода, относящуюся к имени пользователя, показанную в листинге 11.2.

Листинг 11.2. Код для проверки поля имени



Наиболее интересная часть данного листинга — два условия `if`. В первом из них вы проверяете, не является ли текущий запрос частичным (отрицание переменной `$isPartial`) и не является ли значение пустым (`$_POST['name'] === ''`) 1. Если оба определяются как `true`, то добавляете сообщение об ошибке, указывающее, что поле обязательно к заполнению 2.

А вот второй `if` 3 немного сложнее. Вы хотите проверить размер поля, и если оно короче четырех символов (`strlen($_POST['name']) <= 3`), то возвращаете ошибку. Но когда вы хотите произвести проверку, все становится интереснее. Нужно проверить длину, если запрос не частичный (`!$isPartial`), и при этом проверить все поля, а если запрос частичный, то он будет касаться поля имени

(`$isPartial && isset($_POST['name'])`). Основываясь на данном обсуждении, вы можете подумать, что окончательное условие, которое будете использовать, таково:

```
!$isPartial || ($isPartial && isset($_POST['name']))
```

Но часть справа оператора OR будет вычислена, только если левая часть будет `false` — то есть если запрос частичный. Полагаясь на эту информацию, можно сократить условие, получив следующее:

```
!$isPartial || isset($_POST['name'])
```

Если условие определяется как `true`, то устанавливаете сообщение о соответствующей ошибке ❹.

Понять это условие может быть сложновато на первых порах, но, перечитав код внимательно и несколько раз, вы убедитесь, что он написан правильно. Если все еще не уверены, то можете посмотреть диаграмму на рис. 11.4.

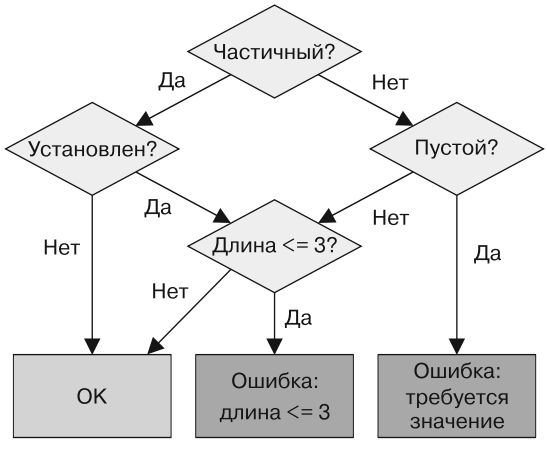


Рис. 11.4. Процесс проверки пользовательского ввода

Проверка других полей аналогична описанной, так что мы ее опустим.

Теперь, когда мы рассмотрели код серверной части, пришло время вникнуть в наиболее интересную часть нашего проекта — код JavaScript. В следующем разделе вы узнаете, как можно использовать jQuery, чтобы связать воедино созданные фрагменты и оживить эту страницу.

11.4. Проверка полей с использованием Ajax

Ваша форма на месте, и есть PHP-страница, способная обрабатывать входящие запросы. Раньше, лет 15 назад, этого было бы достаточно. Пользователь заполняет форму и нажимает кнопку Submit (Отправить), чтобы отправить данные на сервер; тот использует серверный язык для обработки данных, их проверки и последующего формирования страницы для вывода. Но сейчас пользователи ожидают, что

сайты должны быть высокоинтерактивными, и можно удовлетворить их ожидания, задействуя Ajax.

Первая функция, которую вы разработаете, — возможность уведомить пользователя о проверке поля без необходимости отправлять форму. Это действительно хорошее усовершенствование: оно экономит трафик ваших пользователей, поскольку не нужно обновлять абсолютно все.

Идея заключается в следующем: предложить быстрый ответ пользователю, как только данное поле теряет фокус. Для ее реализации необходимо отправить на сервер запрос Ajax, содержащий значение поля, которое только что потеряло фокус. Затем, если оно недействительно, сервер возвращает ошибку. Если она появляется, то вы захотите вывести ее в консоль, чтобы можно было отладить проект. Код, реализующий такое поведение, показан в листинге 11.3.

Листинг 11.3. Проверка данных во время заполнения пользователем формы

```
$('input, textarea', '#contact-form').blur(function() {
   var $this = $(this);

   $.ajax(
      'contact.php',
      {
         method: 'POST'
      dataType: 'json',
      data: $this.serialize() + '&partial=true',
      success: function(data) {
         if (data.status === 'error') {
            $this
               .next('.error')
               .text(data.info[0].message);
         }
      },
      error: function(data) {
         console.log(data);
       }
     }
   );
});
```


Первое, что нужно сделать в этом фрагменте кода, — прослушать событие `blur`, вызванное элементами `input` или `textarea` вашей формы ❶. В обработчике для данного события вы отправляете запрос серверу, используя вспомогательную функцию `$.ajax()`, так как нужен больший контроль над запросом ❷.

Первый параметр, передаваемый функции, — страница, которая получит данные, то есть `contact.php` ❸. Вдобавок к ней вы передаете объект как второй параметр для уточнения запроса. Исходя из выученного в предыдущей главе, если хотите использовать методы HTTP-запроса корректно, нужно применить POST-запрос. Вы устанавливаете свойство `method` как POST ❹. Второе свойство, которое определяете, — `dataType`. Вы знаете, что PHP-страница вернет объект JSON; следовательно, назначаете ожидаемому типу данных соответствующее значение (`json`) ❺. Вы также сериализуете поле и его значение, добавляя специальный параметр (`partial=true`), чтобы отметить запрос как частичный ❻. После установки всех этих свойств пришло время назначить функцию обратного вызова для двух основных статусов: `success` и `error`.

В функции обратного вызова `success` ❼ проверяете значение свойства `status` возвращаемого объекта. Если это `error`, то устанавливаете следующий элемент `span` (описанный в разделе 11.2) значению свойства `message` первого элемента свойства `info` ❽. Внутри функции обратного вызова статуса `error`, показывающего на ошибку в запросе, выводите в консоль полученную ошибку ❾.

С помощью данного кода вы сможете сообщить пользователям о правильности их ввода. Когда сообщение об ошибке будет показано под полем, пользователь наверняка захочет вернуться к этому полю, чтобы исправить ошибку. По мере того как на поле снова переключится фокус, вы должны скрыть ошибку. Реализация этого функционала проста. Все, что нужно сделать, — прикрепить обработчик на событие фокуса для того же набора элементов, выбранных в предыдущем коде. Внутри обработчика вы скрываете элемент `span`. Поскольку нужно работать с тем же набором элементов, можно сэкономить несколько строк кода, соединив код, написанный для реализации этой функциональной возможности (выделено полужирным шрифтом), с представленным в листинге 11.3. Соответствующий код для этого изменения таков:

```
$('input, textarea', '#contact-form').blur(function() {
  // Здесь код пропущен...
})
.focus(function() {
  $(this)
    .next('.error')
    .text('');
});
```

В данном фрагменте и в листинге 11.3 вы применяете несколько методов, описанных в книге: метод `blur()` — для прикрепления обработчика к событию `blur` и вспомогательную функцию `$.ajax()` — для отправки асинхронного запроса на сервер. Метод `serialize()` позволит получить `span` сразу после потерявшего фокус поля. Наконец, с помощью метода `next()` вы устанавливаете текст элемента `span`. Это лишь примеры того, как несколько методов jQuery и вспомогательных функций помогут создать интересные возможности для ваших веб-страниц.

Вне зависимости от результатов проверки введенных значений пользователь может нажать кнопку Submit (Отправить). При нынешней реализации это действие запускает классический синхронный запрос, которого вы хотите избежать. Посмотрим, как можно изменить данное поведение по умолчанию.

11.5. Еще больше веселья с Ajax

Способность обеспечить быструю обратную связь пользователям — это здорово. Однако вам нужно, чтобы форма знала, что делать в случае, когда пользователь нажимает кнопку Submit (Отправить).

Чтобы сделать это, нужно прикрепить обработчик к событию `submit`, вызываемому формой. Обработчик подает Ajax-запрос `contact.php`, отправляя все введенные данные. Когда сервер возвращает результат, нужно проанализировать его. Если он содержит ошибки, то следует показать каждое сообщение об ошибке в соответствующем поле `span`, а затем продемонстрировать диалоговое окно, содержащее общее сообщение о запросе об успехе или неудаче. Код данного функционала показан в листинге 11.4.

Листинг 11.4. Обработчик для управления запросом submit через запрос Ajax

```
$('#contact-form').submit(function(event) {
   event.preventDefault();

   $.post(
      'contact.php',
      $(this).serialize(),
      function (data) {
         if (data.status === 'error') {
            $.each(data.info, function(index, elem) {
               $('#' + elem.field)
               .next('.error')
               .text(elem.message);
            });
         }

         var $dialogBox = $('.dialog-box');

         $dialogBox
            .children('.title')
            .text(data.status);
         $dialogBox
            .children('.message')
            .text(data.message);
         $dialogBox
            .finish()
            .show();
      },
      'json'
   );
});
```

1 Прикрепляет обработчик к событию submit формы

2 Предотвращает поведение по умолчанию

3 Отправляет асинхронный POST-запрос серверу

4 Устанавливает сообщения об ошибке для каждого поля

5 Устанавливает заголовок и сообщение для диалогового окна

6 Останавливает запущенный анимационный эффект и показывает диалоговое окно

Этот код не очень отличается от представленного выше. Здесь вы устанавливаете слушатель для обработчика `submit` формы 1. В нем предотвращаете поведение по умолчанию — то есть отправляете синхронный POST-запрос, — используя метод

`preventDefault()`. Вы применяете его, поскольку не хотите, чтобы POST-запрос отправлялся в любом случае после выполнения кода ❷.

Следующая операция — отправка запроса на сервер. Еще раз: это должен быть POST-запрос. На сей раз не нужно указывать функцию обратного вызова для сообщения об ошибке, так что можно использовать вспомогательную функцию `$.post()`, поскольку ее аргументов достаточно для ваших потребностей ❸. Первый переданный аргумент — страница, которая получит данные, `contact.php`. Затем вы сериализуете форму, чтобы значения, содержащиеся в полях, были отправлены в теле запроса. Третий параметр — функция обратного вызова в случае успеха. Наконец, указывается тип ожидаемых данных (`json`).

Внутри обработчика успешной операции `$.post()` проверьте значение свойства `status` возвращаемого объекта. Если это `error`, то итерируйте по свойству `info` для установки сообщения об ошибке для каждого поля, содержащего ошибки ❹.

Вне зависимости от подтверждения правильности заполнения полей нужно показать общую обратную связь для пользователей в созданном диалоговом окне. Для этого установите заголовок и сообщение диалогового окна ❺ и затем покажите его ❻. Прежде чем показать диалоговое окно, также убедитесь, что все предыдущие анимационные эффекты остановлены с помощью вызова метода jQuery `finish()`.

При использовании такого кода при нажатии кнопки Submit (Отправить) запускается обработчик, который мы только что рассмотрели. Затем показывается диалоговое окно с сообщением об успехе или неудаче. И здесь взаимодействие прекращается. Можете представить почему? Подумайте чуть-чуть об этом.

Причина в том, что вы не предоставили кнопке диалога инструкцию о закрытии (скрытии) диалога после того, как пользователь нажал ее. Исправим это.

Скрытие диалогового окна. Закрытие диалогового окна — не столь сложное дело на данном этапе. Прежде чем углубиться в код, проведем небольшую оптимизацию.

В обработчике запроса Ajax, показанном в листинге 11.4, вы определили переменную `$dialogBox`:

```
var $dialogBox = $('.dialog-box');
```

Поскольку снова нужно проводить определенные действия над данным элементом на странице, можно сэкономить пару нажатий клавиш и немного оптимизировать производительность кода, перенеся инструкции в его начало (на верхнюю часть элемента `script`).

Чтобы скрыть диалоговое окно, когда в нем нажата кнопка, следует прикрепить слушатель события `click` самой кнопки. Это можно сделать так:

```
$dialogBox.children('button').click(function() {
  $(this)
    .parent()
    .hide();
});
```

Благодаря этому дополнению демонстрационный пример делает все запланированное. Если вам понравилась простота, с которой была сделана эта контактная форма, и вы хотите использовать ее в вашем следующем проекте — продолжайте.

Для тех же, кто хочет сделать еще кое-что, — посмотрим, как можно улучшить проект, применяя некоторые эффекты.

11.6. Улучшение пользовательского опыта с использованием анимационных эффектов

Эффекты и анимации — не та часть приложения, без которой нельзя обойтись, — в том смысле, что и с ними, и без них люди имеют возможность решать необходимые задачи. Однако зачастую бывает полезно улучшить пользовательский опыт. В этом проекте можно сделать так, чтобы сообщения об ошибках и диалоговое окно появлялись и исчезали постепенно, а не мгновенно.

Первый добавленный эффект влияет на обработчик кнопки диалогового окна. Можно обновить код так, чтобы данное окно скрывалось медленно. Вместо использования `hide()` можете задействовать метод jQuery `slideUp()`. Вы не станете передавать ему какие-либо параметры, так что эффект будет длиться 400 миллисекунд (это значение по умолчанию). Код выглядит следующим образом:

```
$dialogBox.children('button').click(function() {
  $(this)
    .parent()
    .slideUp();
});
```

Если хотите управлять длительностью, то просто передайте методу соответствующий параметр.

Таким же образом, как добавили эффект скрытия диалога, можно добавить и эффект его появления. Для этой цели подойдет `slideDown()`, но для разнообразия передадим аргумент `show()`. Как было указано в подразделе 8.2.1, можно передать либо число, определяющее количество миллисекунд, в течение которых будет происходить эффект, либо строку со значением `slow`, `normal` или `fast`. Пользователи не должны ждать появления диалога слишком долго, так что передайте строку `fast`, и эффект будет длиться 200 миллисекунд.

Другие эффекты могут быть добавлены в сообщения об ошибках, но это оставим вам для самостоятельной работы.

Добавленные эффекты могут нравиться одним пользователям, но раздражать других. Пытаясь обеспечить удобство для максимально широкой аудитории, вы должны учитывать множество мнений, насколько это возможно. Посмотрим, что можно сделать для тех, кому анимационные эффекты не нравятся.

Переключение анимационных эффектов. В главе 8 мы рассказали о флагах jQuery. Помимо всего прочего, обсуждали флаг `fx.off`, позволяющий глобально запрещать все анимации. Чтобы дать пользователю такую возможность, необходимо предоставить ему HTML-элемент для выбора. В этом демонстрационном примере вы создадите окно выбора с двумя вариантами: `On` (Включить) и `Off` (Выключить), но можете применить любой элемент HTML, который удовлетворяет данному требованию, например флажок или переключатель.

Код элемента `select` следующий:

```
<div class="animations-box">
  <label for="animations">Анимационные эффекты:</label>
  <select id="animations">
    <option value="true" selected>Включены</option>
    <option value="false">Выключены</option>
  </select>
</div>
```

Поместите эту разметку прямо над вашей формой. Затем следует добавить логику, чтобы указанный `select` производил какие-то действия. Для этого нужно прослушивать изменения (используя событие `change`) выбранной `option` и обновлять в соответствии с этим флаг `fx.off`. Данной цели служит такой код:

```
$('#animations').change(function() {
  $.fx.off = $(this).val() === 'false';
})
.change();
```

Как многие из вас заметили, вы не только прослушиваете событие `change`, но также и запускаете его после установки обработчика. Это обеспечивает установку флага вне зависимости от значения `<select>` по умолчанию. Данный прием будет удобным, если вы захотите, чтобы параметр Off (Выключить) был значением по умолчанию, без необходимости обновлять код JavaScript (устанавливающий флагу `fx.off` значение `true`).

Прежде чем завершить проект, мы хотим обсудить еще один последний момент.

11.7. Замечание о доступности

JavaScript — эффективный и повсеместно используемый язык, позволяющий решать невероятное количество задач. Как вы уже видели в этой книге, jQuery позволяет вывести ваш код на следующий уровень, делая многое с меньшим количеством кода. Однако при разработке веб-приложений нужно учитывать, что не все можно и даже не все разрешается загружать и выполнять с помощью JavaScript. У некоторых пользователей данный язык может быть отключен на компьютере. Или их сервер может прекратить обслуживание библиотеки JavaScript, модуля или файла в целом. Для подобных случаев у вас, как веб-профессионалов, должен быть план Б.

В этом проекте вы применяли JavaScript ради улучшения интерфейса для ваших пользователей. Если ваш код JavaScript не сможет загрузиться по какой-либо причине, то ваша форма контактов сможет работать в любом случае. Кнопка Submit (Отправить) сможет отправлять данные формы на сервер. Это возможно, поскольку вы разработали данную возможность в виде надстройки над естественным поведением HTML. Единственный недостаток — процесс превратится в синхронный, и ваши пользователи увидят новую страницу (или ту же с другим контентом), обслуживаемую сервером. Так ли это? Действительно ли пользователи увидят страницу?

Вы разработали страницу `contact.php` так, что она всегда обслуживает объект JSON. Если код JavaScript не загрузится и пользователь отправил форму, то он

увидит непонятную и трудночитаемую строку, представляющую собой объект JSON. Какой ужас! Что мы сможем сделать в данном случае?

Оказывается, можно слегка обновить проект для решения этой проблемы. Нужно всего лишь отредактировать `contact.php`, чтобы он смог отличить, является ли запрос Ajax . Если да, то он может быть обслужен как объект JSON. В противном случае он должен использовать собранные данные, чтобы создать полную страницу и отправить ее пользователю. Это выглядит как простое изменение, но может повысить доступность приложения и избавить ваших пользователей от разочарования. У представленного подхода, называемого *прогрессивным усовершенствованием*, есть много преимуществ, и он подходит не только для данного проекта.

Прогрессивное усовершенствование

Прогрессивное усовершенствование — это методология, которая подчеркивает доступность, семантическую HTML-разметку, внешние таблицы стилей и технологии сценариев. Выражение было придумано Стивеном Чампеоном в серии статей и презентаций для Webmonkey и конференции SXSW Interactive в 2003 году.

Данная методология пропагандирует создание веб-страниц, чтобы каждый мог получить доступ к основному содержанию и функциональным возможностям, а затем предоставление расширенной версии для тех, кто использует более совершенные технологии (например, современный браузер). Это повышает не только доступность веб-страниц, но и их ранг в результатах поиска (SERP — Search Engine Result Pages), так что применяйте данное усовершенствование в ваших последующих проектах.

Такое проектирование для случаев со сбоями JavaScript — не единственный способ, с помощью которого можно улучшить доступность веб-страниц. Вы можете и должны применять также и WAI-ARIA (http://www.w3.org/TR/waiaria/). Объяснение этих деталей выходит за рамки нашей книги, но дадим общее представление: она обеспечивает онтологию ролей, состояний и свойств, определяющих доступность элементов пользовательского интерфейса. Это улучшает доступность и функциональную совместимость веб-контента и приложений. Одна из ролей, `dialog`, идеально подходит вашему диалоговому окну. Для использования ее добавьте некоторые атрибуты (выделенные полужирным) в разметку следующим образом:

```
<div class="dialog-box" role="dialog" aria-labelledby="dialog-title"
    aria-describedby="dialog-desc">
  <h2 id="dialog-title" class="title"></h2>
  <p id="dialog-desc" class="message"></p>
  <button>OK</button>
</div>
```

Другое улучшение может быть осуществлено с помощью атрибута HTML5 `required`, который уже упоминался в этой главе. Его применение позволит пользовательским почтовым агентам, поддерживающим HTML5, предоставлять информацию об обязательности заполнения данного поля. Ассистивные технологии (Assistive Technologies, ATs) не всегда работают с той же скоростью, что и обычные

(Chrome, Firefox и т. д.). Для восполнения этого пробела можно установить атрибут WAI-ARIA `aria-required` ко всем обязательным к заполнению элементам, как показано здесь:

```
<input name="name" id="name" required aria-required="true" />
```

Даже если пользовательский агент поддерживает HTML5, то добавление отдельных атрибутов WAI-ARIA не повредит.

Усовершенствования, которые мы обсудили, — малая часть того, что можно сделать для улучшения доступности вашей формы контактов. Но эти изменения должны побудить вас рассмотреть вопрос о доступности в вашем следующем проекте.

11.8. Резюме

В этой главе мы воспользовались нашими знаниями при разработке простой, но полнофункциональной Ajax-формы контактов. При разработке данного демонстрационного примера мы коснулись множества тем, обсуждавшихся в нашей книге. Мы использовали селекторы, включая параметр `context`, который вы изучили в главе 2. Такие методы, как `parent()`, `next()` и `find()`, представленные в главе 3, применялись для усовершенствования выбора элементов. Метод `text()`, обсуждавшийся в главе 5, послужил для обновления текста `span`, содержащего ошибки. Мы задействовали несколько связанных с событиями методов, о которых говорилось в главе 6, для их прослушивания и запуска (например, `blur()`, `click()` и `focus()`). Мы добавили некоторые эффекты, описанные в главе 9, помогли вам итерировать по массиву ошибок. Наконец, мы обратились к вспомогательным функциям `$.ajax()` и `$.post()`, обсуждавшимся в главе 10, для осуществления асинхронных запросов к серверу.

Это резюме показывает: все представленное нами не только существует в теории, но и находит практическое применение в реальном мире. В демонстрационном примере мы использовали по меньшей мере по одному понятию из каждой главы.

Мы надеемся, что, изучая материалы этих страниц, вы поняли, насколько важна каждая часть библиотеки jQuery для достижения определенной цели (формы контактов в нашем случае) и как, сочетая их, можно получить невероятные возможности. Надеемся, вам понравилось разрабатывать данный проект и вы уже более уверенно используете функции и методы, которые мы рассмотрели на текущий момент.

Этим примером мы завершаем вторую часть книги. Начиная со следующей главы, перейдем к более серьезным задачам, таким как создание плагинов и модульное тестирование кода.

Часть III
Дополнительные возможности

Во второй части этой книги мы рассмотрели невероятное количество селекторов, методов и вспомогательных функций jQuery. Если вы освоили их, затратив немного времени и приложив чуточку усердия, то теперь при желании сможете реализовывать любые возможности. В последней главе мы доказали, что это правда и при наличии достаточных знаний единственным ограничением является ваша фантазия.

Библиотека jQuery — мощный инструмент, однако в ней может не оказаться методов и функций, нужных вам для проекта. Чтобы восполнить этот пробел, в jQuery предусмотрены широкие возможности расширяемости, что позволит веб-разработчикам использовать свой функционал, как если бы он был частью ядра библиотеки. В следующих главах вы научитесь писать плагины для jQuery. Затем мы обсудим объект `Deferred` и его методы. Объект `Deferred` относится к ядру jQuery, но мы решили рассмотреть его отдельно, поскольку знания о нем достаточно сложны для понимания.

За исключением случаев разработки совсем небольшого проекта для единоличного пользования, вы будете писать код, который придется каким-то образом перерабатывать, обновлять и изменять. В таких ситуациях нужно быть уверенными, что весь код, работавший до изменений, не сломается после обновления. Чтобы добиться таких гарантий, проект нужно тестировать. Поскольку речь зашла о тестировании, почему бы не воспользоваться специальным инструментом для модульного тестирования под названием QUnit, который был создан командой разработчиков jQuery для тестирования самой библиотеки? Звучит разумно, именно этим мы и займемся в главе 14.

Наконец, в последней главе книги мы познакомим вас с некоторыми инструментами, поделимся методами, советами и приемами, которые пригодятся при разработке больших проектов, и покажем, как jQuery вписывается в них.

Все упомянутые темы отличают «умеренно опытного» разработчика, знающего, как использовать библиотеку jQuery, от настоящего профессионала, который способен улучшить и оптимизировать код, не забывая одновременно закладывать основы для его совершенствования в будущем (благодаря тестированию). Об этом и пойдет речь в следующих главах.

Не будем тратить впустую драгоценное время, пора переходить к самым интересным темам этой книги.

12 jQuery не справляется? Помогут плагины!

В этой главе:

- зачем расширять jQuery с помощью дополнительного кода;
- использование плагинов сторонних разработчиков;
- рекомендации по написанию эффективных плагинов jQuery;
- написание вспомогательных функций;
- написание методов для объектов jQuery.

Прочитав бóльшую часть данной книги, вы уже поняли, что jQuery предоставляет широкий набор полезных методов и утилит. Вы также научились комбинировать эти инструменты, чтобы обеспечить желаемое поведение страницы. Иногда код реализует распространенные приемы и наборы операций, которые вы намерены применять снова и снова. В таких случаях имеет смысл сохранить их в виде многоразового инструментария и добавить его к исходной библиотеке. В данной главе мы покажем, как сохранить часто используемые фрагменты кода в виде *плагинов* jQuery. Они бывают двух видов: методы и коллекции (как `find()` и `animate()`) или вспомогательные функции (как `$.grep()` и `$.extend()`). Мы рассмотрим оба вида.

При разработке проекта у вас вряд ли будет время на то, чтобы писать все необходимые инструменты с нуля, особенно если данный код уже разработал кто-то другой. Поэтому мы также познакомим вас с рядом популярных плагинов, которые могут пригодиться. Но прежде всего мы ответим на вопрос: зачем сохранять код в виде плагинов jQuery?

12.1. Зачем нужны плагины jQuery

Если до сих пор вы внимательно читали эту книгу, то, конечно же, заметили, что использование jQuery при разработке веб-страниц оказывает огромное влияние на то, как выглядит сценарий на странице.

Библиотека задает определенный стиль кода: обычно он заключается в формировании коллекций jQuery, к которым затем применяется метод jQuery или цепочка таких методов. Вы можете писать собственный код как захотите, но наиболее опытные

разработчики сходятся во мнении, что весь или почти весь код сайта должен быть оформлен в едином стиле. Это хорошая практика, и мы ее тоже рекомендуем. Одна из веских причин структурировать код, создавая плагины jQuery, заключается в следующем: это позволит поддерживать единый стиль кода для всего сайта. Еще одна важная причина — многоразовыми компонентами сможете пользоваться не только вы в будущих проектах, но и другие разработчики, если опубликуете свой код в Интернете.

Наконец, последняя причина, на которую мы обратим внимание (хотя, возможно, есть и другие), — создавая плагины jQuery, можно опираться на уже существующий базовый код, доступный благодаря нашей библиотеке. Например, при создании новых методов jQuery автоматически наследуется действенный механизм селекторов jQuery и кросс-браузерная совместимость, реализованная в этой библиотеке. Зачем писать все с нуля, если можно опереться на такой эффективный инструментарий?

Теперь, зная эти причины, вы понимаете, почему написание многоразовых компонентов в виде плагинов jQuery — разумный подход к работе. В данной главе мы рассмотрим рекомендации и подходы, позволяющие создавать такие плагины, и сами создадим несколько образцов.

Но прежде, чем начнем учиться разрабатывать собственные плагины, посмотрим, где искать, как выбирать и использовать плагины, созданные другими разработчиками.

12.2. Где искать плагины

После нескольких месяцев напряженного труда 16 января 2013 года команда jQuery объявила в своем официальном блоге о выпуске нового (конечно же, улучшенного) реестра плагинов jQuery (http://blog.jquery.com/2013/01/16/announcing-the-jquery-plugin-registry/). Будучи доступным на http://plugins.jquery.com/, он заменил старый, доставлявший много проблем. Но примерно через два года новый реестр был переведен в режим «только для чтения» — это значило, что новые версии плагинов не разрабатывались. В качестве замены новому реестру команда jQuery рекомендует использовать npm (https://www.npmjs.com/).

По данному URL вы найдете страницу с простым интерфейсом. Это панель поиска, с помощью которой можно искать нужные плагины. Если щелкнуть на имени плагина в списке результатов поиска, то можно получить много дополнительной информации о нем, в том числе ссылку на загружаемый архив.

Канал npm является рекомендованным, но не единственным для поиска плагинов. Другой вариант — страница Unheap (http://www.unheap.com/) с приятным интерфейсом, но довольно скудным выбором.

Если ни один из вариантов не нравится или вы не нашли нужный инструмент, помните: всегда есть Google. Только не забудьте верифицировать исходный код. В конце концов, это же ваш сайт и вы собираетесь включить в его состав один или несколько файлов JavaScript! Если же и этого мало, то подождите немного — ниже мы покажем, как создавать плагины.

Одного знания, где искать плагины jQuery, мало. Вы же не хотите, чтобы продуманный до последних мелочей проект наполнился некачественным кодом или, хуже того, стал медленно работать из-за некорректно написанного плагина. В следующем подразделе мы дадим несколько советов, как оценить качество плагина.

12.2.1. Как использовать хорошо написанный плагин

Опираться на плагины сторонних разработчиков при работе над проектом — разумный способ сэкономить время, поскольку вам не придется создавать, тестировать и обслуживать данный код. Но так ли это? Как показывает наш опыт, не всегда.

Включить в проект плагин означает добавить в него зависимость. Выбор плагина — важное решение, поскольку на нем будет основан целый проект, поэтому нужно не пожалеть времени и выяснить ряд моментов. Часть из них касаются непосредственно кода, другие — чисто внешние. Рассмотрев их в комплексе, вы составите представление о стабильности и качестве выбранного компонента. Начнем с анализа некоторых факторов, не имеющих прямого отношения к коду.

Факторы, не связанные с кодом

Вы решили довериться компонентам сторонних разработчиков, чтобы освободить себя от необходимости создавать одну или несколько функций с нуля. Но если не уделить должного внимания используемым компонентам, то можно потерять больше времени, чем вы рассчитывали сэкономить, — пока будете исправлять ошибки или разбираться, как применять выбранное. И хотя здесь мы обсуждаем плагины jQuery, помните: все эти нюансы актуальны и для других инструментов, библиотек и вообще для любого программного обеспечения от сторонних разработчиков.

ПРИМЕЧАНИЕ

Более основательное обсуждение этой темы и сопутствующих вопросов вы найдете в главе Николаса Закаса «Пишем удобный в сопровождении, расширяемый код» (Writing Maintainable, Future-Friendly Code) в четвертой электронной книге Smashing под названием New Perspectives on Web Design (2013,https://shop.smashingmagazine.com/products/smashing-book-4-ebooks).

Первое, на что следует обратить внимание, — когда в последний раз обновлялся плагин. По этой дате вы поймете, сколько внимания уделяет автор данному проекту. Избегайте использовать плагин, который давно не обновлялся. Прежде чем его применять, не пожалейте времени на чтение журнала внесенных изменений. Дата последнего обновления не всегда хороший показатель. Плагин мог быть написан на старой версии jQuery, а позже могли быть выпущены промежуточная версия или патч (подробнее об этих терминах читайте на http://semver.org/), и он все еще работает благодаря обратной совместимости. Но в данном случае очередное обновление может сделать плагин неработоспособным. И это подводит нас к следующему пункту.

Второй фактор, который необходимо учитывать, — скорость устранения неполадок автором. Идеального программного обеспечения не существует, проблемы возникают по самым разным причинам. Важно, насколько быстро автор их исправляет. Если используемый вами плагин перестанет работать из-за обновления jQuery, а получить оперативный ответ не удастся, то придется либо продолжать пользоваться старой версией jQuery, либо исправлять ошибки самостоятельно, что сведет к нулю все преимущества внешних функций.

Третий важный фактор — версия плагина. Если его автор не новичок, то версия библиотеки имеет точное значение (описанное по приведенной ранее ссылке). Никогда не используйте плагин версии 0.1.0, если только не собираетесь поразвлечься. Программное обеспечение, не достигшее версии 1.0.0, часто находится на стадии активных изменений, и в нем отсутствует обратная совместимость.

Четвертый фактор — автор. Кто это — компания или студент-одиночка? Какая репутация у данной компании или разработчика? Как правило, компоненты, созданные компаниями, хорошо поддерживаются, поскольку компании могут себе позволить вкладывать деньги в поддержку своих продуктов, в то время как одиночные разработчики обычно уделяют таким проектам свое свободное время. Впрочем, если автор — разработчик-одиночка, но с хорошей репутацией, то, возможно, плагин стоит использовать.

Пятый фактор — документация. Если плагин слабо документирован или документации нет вообще, советуем продолжить поиски. Иначе вам придется потратить много времени на попытки понять, как все работает и как его использовать.

Подведем итоги. Помните: качество плагинов jQuery бывает разным и именно вам необходимо приложить все свои знания, чтобы проверить его как следует.

Рассмотренные в данном разделе особенности очень важны, но представляют собой только одну сторону вопроса. Чтобы правильно оценить плагин, необходимо также как следует изучить его код. Это не значит, что мы советуем пойти в Интернет и читать весь код каждого плагина, который вы намерены использовать, — на это ушло бы слишком много рабочих часов. Идея в другом: получить представление о качестве исходного кода, прочитав его фрагмент и посмотрев, нет ли там ошибок. Но для этого нужно знать принципы сборки плагина. Еще несколько страниц — и вы получите пошаговую инструкцию, как создать плагин.

А сейчас поговорим о том, как использовать плагины сторонних разработчиков.

Использование плагина

После того как вы нашли подходящий плагин и убедились, что он заслуживает внимания, можете добавить его в проект. Если он хорошо написан, то пользоваться им, как правило, легко. Все, что нужно сделать, — это сохранить его в папке, доступной веб-серверу, и добавить его в проект после библиотеки jQuery.

Для начала рассмотрим плагин jQuery Easing (https://github.com/gdsmith/jquery.easing) — мы уже упоминали его в разделе 8.3, когда обсуждали функции сглаживания анимации. После загрузки сохраните его в папке, которая будет доступна с вашей страницы, например в `javascript`. Затем добавьте его на страницу с помо-

щью элемента `script` после библиотеки jQuery. Если поместить плагин до jQuery, то получите ошибку и весь JavaScript на странице перестанет работать. На вашей странице должна быть разметка, подобная этой:

```
<script src="javascript/jquery.1.11.3.min.js"></script>
<script src="javascript/jquery.easing.min.js"></script>
```

По готовности разметки ваши следующие действия будут зависеть от конкретного плагина. В данном случае не понадобится вызывать какие-либо методы jQuery или другие вспомогательные функции. jQuery Easing вставляет функции сглаживания в ядро jQuery, позволяя использовать их так, как будто они там были с самого начала.

Этот плагин — особый случай. Как правило, плагины требуют добавить на вебстраницу ту или иную разметку, назначить элементам определенные классы или даже ID. Чтобы изучить один из таких плагинов на практике, посмотрите на slick (https://github.com/kenwheeler/slick) — плагин jQuery для создания каруселей. В следующем примере мы напишем код для карусели из изображений.

Первое, что нужно сделать для использования плагина slick, — вставить его файл JavaScript после библиотеки jQuery. Если вы сохранили его в папке `javascript`, расположенной на одном уровне с HTML-страницей, то разметка будет выглядеть так:

```
<script src="javascript/jquery.1.11.3.min.js"></script>
<script src="javascript/slick.min.js"></script>
```

После подключения файла JavaScript нужно также подключить файл CSS, входящий в состав slick. Как мы уже знаем из главы 1, файлы JavaScript следует размещать перед закрывающим тегом `</body>`, в то время как файлы CSS должны располагаться внутри тега `<head>`.

Если вы сохранили файл CSS в папке с именем `css`, то получится такой код:

```
<head>
  <link rel="stylesheet" href="css/slick.css" />
```

Когда вы с этим закончите, нужно будет внести изменения в разметку страницы. Поскольку мы хотим создать карусель из изображений, нужно заключить изображения в контейнерный элемент (в данном случае мы использовали `<div>`), как показано здесь:

```
<div class="carousel">
    <img src="images/image1.jpg" />
    <img src="images/image2.jpg" />
    <img src="images/image3.jpg" />
    <img src="images/image4.jpg" />
</div>
```

Когда этот код будет готов, останется только вызвать волшебный метод `slick()`, чтобы все завертелось:

```
<script>
    $('.carousel').slick();
</script>
```

Это выражение полагается на конфигурацию плагина по умолчанию, но при необходимости можно изменить его параметры. Если вы захотите ближе познакомиться с данным плагином, то загляните в его репозиторий и почитайте документацию.

Изучив два примера, вы, вероятно, уже получили представление, чего следует ожидать от сторонних компонентов, встроенных в ваши веб-страницы. Теперь, прежде чем приступить к созданию плагинов, рассмотрим вкратце несколько популярных и полезных готовых плагинов jQuery.

12.2.2. Отличные плагины для ваших проектов

В этом подразделе представлен краткий список ряда наиболее популярных и полезных плагинов jQuery, которые вы можете использовать в своих проектах для выполнения часто возникающих задач. Данный список не является исчерпывающим, но для начала вполне подойдет.

Первый из предлагаемых плагинов — typeahead.js (https://github.com/twitter/typeahead.js). Это быстрое и полнофункциональное расширение для автодополнения от Twitter. Можно передать ему набор данных и получить поле `<input>`, в котором будут появляться предложения для пользователей по мере ввода текста.

Следующий заслуживающий внимания плагин jQuery — isotope (https://github.com/metafizzy/isotope). Он позволяет фильтровать и сортировать элементы пользовательского интерфейса, используя красивую анимацию и разные варианты размещения. Среди доступных вариантов — в строку, в столбик и в виде кирпичной кладки.

Еще один очень интересный плагин — pickadate.js (https://github.com/amsul/pickadate.js). Это предусмотренный для мобильных устройств, адаптивный и легкий элемент выбора даты и времени. Плагин добавляет к тегу `<input>` виджет, выводящий указатель даты или времени, чтобы упростить выбор пользователю, когда он переходит к данному элементу.

Четвертый плагин — Chosen (https://github.com/harvesthq/chosen). Эта библиотека позволяет сделать длинные, громоздкие списки выбора более красивыми и удобными в применении.

Velocity (https://github.com/julianshapiro/velocity) — плагин jQuery, переопределяющий метод jQuery `animate()`. В новом варианте метода повышена производительность и появились новые функции.

Наконец, еще два плагина, на которые мы советуем обратить внимание, — jCarousel (https://github.com/jsor/jcarousel) и Magnific Popup (https://github.com/dimsemenov/Magnific-Popup). Первый позиционируется как расширение, способное создавать карусель не только из изображений, но и из других объектов. Magnific Popup — легкий и адаптивный плагин для создания всплывающих окон. Особое внимание в нем уделено производительности.

Перечисленные плагины стали популярными благодаря тому, что красиво и эффективно решают типичные задачи, или же потому, что стали делать это раньше

других. Как бы то ни было, мы уверены: вы захотите и сами создавать такие же успешные плагины, как представленные здесь. Для этого нужно научиться разработке хороших плагинов, чему и посвящен следующий раздел.

12.3. Руководство по созданию плагинов jQuery

В данном разделе содержится набор рекомендаций, которые помогут структурировать плагин и выбрать для него хорошее название. Следуя этим советам, вы создадите код, который не только правильно встроится в архитектуру jQuery, но также будет хорошо совмещаться с другими плагинами jQuery и даже другими библиотеками JavaScript. Вкратце основные принципы и правила разработки плагина заключаются в следующем.

Плагин jQuery может принимать одну из двух форм:

- методы, обрабатывающие коллекцию jQuery (которые мы называем методами jQuery);
- вспомогательные функции, определенные непосредственно через `$` (псевдоним для `jQuery`).

В оставшейся части этого раздела мы подробно рассмотрим некоторые рекомендации, типичные для обоих вариантов, а остальные разделы посвятим каждому из них в отдельности.

Чтобы упростить процесс изучения, будем проверять каждое новое рассмотренное правило на примере. Нашей целью будет построить Jqia Context Menu (Jqia — сокращенное jQuery in Action), плагин jQuery, который будет показывать специальное контекстное меню для одного или нескольких выбранных нами элементов страницы. Контекстное меню — то, что появляется на экране компьютера, если щелкнуть на странице правой кнопкой мыши или нажать кнопку меню, когда страница активна.

Для создания контекстного меню плагин будет использовать элемент страницы (обычно это список), который по умолчанию скрыт. Данный элемент надо создать и где-то разместить. Элемент страницы, играющий роль меню, будет вызываться по ID. Наш плагин будет предусматривать два действия и, соответственно, иметь два метода: для инициализации расширения и для отмены эффекта. При инициализации плагин будет переопределять стандартное поведение при щелчке правой кнопкой мыши (при котором появляется обычное контекстное меню) и показывать специальное меню. При отмене — очищать ресурсы и восстанавливать стандартное поведение.

Наконец, чтобы было интереснее, мы позволим разработчикам, которые будут использовать наш плагин, переопределить щелчок левой кнопкой мыши. В таком случае специальное меню будет отображаться независимо от того, какой кнопкой был выполнен щелчок. По умолчанию данный вариант будет отключен.

Теперь, когда мы изложили наш план, примемся за работу — у нас ее много.

12.3.1. Соглашения об именах файлов и функций

Первое решение, которое необходимо принять при разработке плагина, касается имени. Выбирая его, необходимо избежать конфликтов. Важно, чтобы имя нового плагина не совпадало с именами других файлов и расширений, иначе у авторов веб-страниц будет множество проблем.

На этот случай у команды jQuery есть простая, но эффективная рекомендация — следовать такому формату:

- имя должно быть коротким, но достаточно понятным;
- перед именем файла должен стоять префикс jquery;
- дополнительно можно добавить название компании или пакета;
- затем идет имя плагина;
- затем, при желании, номер версии плагина;
- в конце должно стоять `.js`.

Если выполнять эти рекомендации, то имя файла с нашим плагином Jqia Context Menu должно выглядеть так: `jquery.jqia.contextMenu-1.0.0.js`.

Префикс jquery в идеале поможет избежать конфликта имен с файлами, предназначенными для использования с другими библиотеками. В сущности, разработчику, пишущему плагины для других библиотек, ни к чему добавлять к своим файлам данный префикс. Впрочем, и само имя оставляет поле для дискуссий внутри сообщества jQuery. Поэтому в нашем примере имя плагина состоит из нескольких слов (в настоящее время многие слова уже использованы в других именах). Соединяя слова, мы применили правило верблюжьего регистра, но можно писать все слова строчными буквами, разделить их точками или дефисами. Какой вариант выберете — дело вкуса, мы лишь советуем выбрать один и всегда ему следовать.

Один из способов убедиться, что имена файлов вашего плагина, *скорее всего* (никогда нельзя быть уверенными на 100 %), не конфликтуют с другими, — добавить к ним второй префикс в виде уникального имени. Это может быть ваше имя, название вашей организации или пакета. Например, если бы мы хотели создать второй префикс для плагинов, создаваемых в данной книге, то могли бы использовать префикс jquery.jqia, как в предыдущем примере.

Третий пункт списка — необязательный, и на то есть важная причина. Предположим, некие разработчики применяли опубликованный нами плагин. Все шло хорошо, и он работал как следует. Мы собрались выпустить следующую версию с новыми функциями, и тут возникла проблема. Наш файл назывался `jquery.jqia.contextMenu-1.1.0.js`. Таким образом, разработчики, использующие наш плагин, должны не только обновить файл JavaScript, но и изменить разметку, чтобы в имени файла стояла новая версия. Было бы гораздо проще, если бы плагин назывался `jquery.jqia.contextMenu.js`, а версия была бы указана внутри файла, в комментариях. Тогда разработчикам было бы достаточно только заменить файл JavaScript, не меняя разметки.

Теперь, прочитав пояснения, вы, как мы надеемся, поняли, зачем нужны все эти правила, и мы можем начать их применять при разработке нашего плагина.

Мы проигнорируем необязательное правило относительно версии и создадим файл с именем `jquery.jqia.contextMenu.js`.

В данном разделе мы показали, как важно выбрать правильное имя файла, поскольку мы не можем предполагать, что еще будут использовать разработчики других сайтов. То же самое касается нашего любимого псевдонима `$`. Присмотримся к нему внимательнее.

12.3.2. Осторожно: $

Написав немало строк кода jQuery, вы уже наверняка оценили удобство использования псевдонима `$` вместо слова `jQuery`. Но когда созданные вами плагины окажутся на страницах других людей, вы можете оказаться не единственным, кто применяет такое сокращение. Как авторы плагина, мы не можем знать, будет ли разработчик веб-страницы задействовать функцию `$.noConflict()` (которая обсуждалась в разделе 9.2), чтобы к псевдониму `$` могли прибегать другие библиотеки (самая известная из них — Prototype). Мы могли бы использовать имя `jQuery` вместо псевдонима `$`, но нам *нравится* `$`.

В разделе 9.2 был представлен шаблон проектирования IIFE (Immediately-Invoked Function Expression, выражение немедленно вызываемой функции). Более подробно этот шаблон описан в приложении. Он часто используется для того, чтобы убедиться, что псевдоним `$` в данном контексте соответствует имени `jQuery` и не влияет на остальную часть страницы. Шаблон IIFE можно и нужно применять при создании плагинов jQuery таким образом:

```
(function($){
//
// Определение плагина
//
})(jQuery);
```

Передавая имя `jQuery` функции, которая принимает параметр `$`, вы гарантируете, что внутри функции использование `$` будет означать обращение к `jQuery`. Теперь можно спокойно пользоваться `$` при описании плагина.

Приняв к сведению это новое знание, откройте файл `jquery.jqia.contextMenu.js` и поместите в него приведенный код (комментарии можно пропустить).

Теперь рассмотрим следующую рекомендацию — она касается параметров.

12.3.3. Укрощение сложных списков параметров

Большинство плагинов стремятся к простоте и используют мало параметров либо вообще обходятся без них. Они предусматривают варианты по умолчанию, когда необязательные параметры можно пропустить, а порядок следования параметров зависит от того, какие из необязательных параметров пропущены.

Хороший пример такого поведения — метод jQuery `on()`; если не указан необязательный параметр `data`, то функция-обработчик, которая обычно является четвертым параметром, становится третьим. Если также пропущен параметр `selector`, то обработчик становится вторым аргументом. Динамическая природа JavaScript

позволяет писать подобный гибкий код, но такое положение вещей может привести к сбоям и сложностям (и для веб-разработчиков, и для авторов плагинов), когда параметров становится много. И чем больше необязательных параметров, тем выше вероятность сбоя.

Рассмотрим функцию с такой сигнатурой:

```
function complex(p1, p2, p3, p4, p5, p6, p7) {
    // Код здесь...
}
```

Данная функция имеет семь параметров. Предположим, все они, кроме первого, — необязательные. Слишком много необязательных параметров, чтобы прийти к какому-либо логичному выводу о том, что имел в виду разработчик, когда вызывал функцию без этих параметров. Если при вызове пропущены только последние параметры, то проблема невелика, поскольку можно считать, что пропущенные параметры имеют значение `undefined`. Но если при вызове мы хотим задать `p7` и считать, что все параметры от `p2` до `p6` имеют значения по умолчанию? А если некоторые из пропущенных параметров принимают данные одного типа (и их отсутствие не дает возможности распознать, что именно пропущено, по типу данных)? Тогда при вызове приходится использовать заполнители для всех отсутствующих параметров и писать так:

```
complex(valueA, null, null, null, null, null, valueB);
```

Отвратительно! Хуже этого — только такое:

```
complex(valueA, null, valueC, valueD, null, null, valueB);
```

Чтобы использовать такую функцию, веб-разработчику приходится внимательно следить за количеством нулей и последовательностью параметров. К тому же код становится трудно читать и понимать. Но как поступить, если мы хотим, чтобы при вызове функции было действительно много вариантов?

Здесь на помощь снова приходит гибкая природа JavaScript. Шаблон, позволяющий укротить весь этот хаос, родился в сообществах авторов веб-страниц и называется *options hash* (хеш вариантов). Согласно данному шаблону, необязательные параметры объединяются в один под видом экземпляра объекта JavaScript. Его пары «имя — значение» как раз и играют роль этих необязательных параметров.

Используя эту технологию, наш первый пример можно переписать так:

```
complex(valueA, {p7: valueB});
```

А второй — так:

```
complex(valueA, {
  p3: valueC,
  p4: valueD,
  p7: valueB
});
```

Так гораздо лучше!

Теперь не надо заменять пропущенные параметры значениями `null` и считать их. У каждого необязательного параметра есть имя, по которому к нему удобно обращаться: вы точно знаете, что он представляет (особенно если заменить имена `p1` — `p7` на более удачный вариант).

ПРИМЕЧАНИЕ

Разработчики некоторых API выполняют это соглашение, объединяя все необязательные параметры в один параметр `options` (при этом обязательные параметры задаются отдельно). Другие объединяют в один объект весь набор параметров, включая в него как обязательные, так и необязательные. Нам больше нравится второй подход. Это перспективный вариант, с учетом того, что со временем количество обязательных параметров может увеличиться.

Конечно, для пользователя таких сложных функций это большое преимущество. Но как вам, автору плагина, справиться со всеми описанными модификациями? Оказывается, jQuery поддерживает механизм, позволяющий собрать все параметры воедино и назначить им значения по умолчанию, и мы с данным механизмом уже знакомы. Посмотрим еще раз на нашу функцию с одним обязательным и шестью дополнительными параметрами. Ее новая упрощенная сигнатура выглядит так:

```
complex(p1, options)
```

Удобная сервисная функция `$.extend()` обеспечивает слияние заданных параметров и значений по умолчанию. Рассмотрим такой код:

```
function complex(p1, options) {
  var settings = $.extend({
      p2: defaultValue1,
      p3: defaultValue2,
      p4: defaultValue3,
      p5: defaultValue4,
      p6: defaultValue5,
      p7: defaultValue6
  },
  options || {}
);
// Остальной код функции
}
```

При слиянии значений, переданных веб-разработчиком в параметре `options`, с объектом, содержащим все доступные варианты с их значениями по умолчанию, получается переменная `settings`, в которой значения по умолчанию заменены явно заданными значениями, переданными разработчиком.

СОВЕТ

Вместо того чтобы создавать новую переменную settings для сбора значений, можно использовать ссылку на сам параметр options и тем самым сэкономить одну ссылку в стеке. Но сейчас мы оставим все как есть — для более понятного кода.

В предыдущем примере мы застраховались от появления в объекте `options` значений `null` и `undefined` с помощью кода `|| {}`. Он заменяет `options` на пустой объект, если `options` имеет значение `false` (как мы знаем, `null` и `undefined` эквивалентны `false`). Просто, универсально и удобно для вызова.

Теперь, когда мы продвинулись еще на один шаг вперед в изучении того, как создавать изящные и хорошо написанные плагины, применим новые знания в нашем проекте. Возвращаясь к описанию Jqia Context Menu, вспомним, что нам нужен необязательный параметр, определяющий, будет ли контекстное меню также появляться при нажатии левой кнопки мыши. Кроме того, нужно задать ID элемента, который будет играть роль меню. И хотя нужны всего два параметра (один из них обязательный, а второй — нет), мы воспользуемся концепцией передачи плагину одного объекта. Причиной, как уже отмечалось выше, является то, что такая концепция является более перспективной.

Превратив это описание в код и поместив его в наш рабочий файл JavaScript вместо его предыдущего содержимого, получим такой результат:

```
(function($) {
  var defaults = {
      idMenu: null,
      bindLeftClick: false
  };
})(jQuery);
```

В этом коде мы определяем объект с именем `defaults`. Он содержит свойство, определяющее ID меню (`idMenu`), и другое, от значения которого зависит, будет ли переопределено нажатие левой кнопки мыши (`bindLeftClick`).

Пока что мы не сделали ничего выдающегося — всего лишь определили объект с двумя свойствами. Пора изучить рекомендации о том, как разрабатывать методы плагина.

12.3.4. Единое пространство имен

Как и в случае с именами файлов, необходимо убедиться, что имена всех функций — как сервисных функций, так и методов для коллекций jQuery, — не конфликтуют с методами других плагинов, которые может использовать разработчик.

При создании плагинов для собственных нужд вы обычно знаете, какие еще компоненты применяете; в своем проекте, если только он не гигантских размеров, легко избежать конфликтов имен. Но как быть с плагинами, предназначенными для открытого доступа? Если плагин, который вы поначалу писали для себя, окажется таким полезным, что вы захотите поделиться им с другими разработчиками?

Чтобы лучше понять эту идею, рассмотрим наглядный пример. Как мы уже говорили, плагину Jqia Context Menu нужны два метода: `init()` и `destroy()`. Возможно, у вас уже возник соблазн добавить в файл JavaScript такой код:

```
var init = function(options) {
  // Код функции
};
var destroy = function() {
  // Код функции
};
```

К сожалению, это нам не подходит. Одна из главных концепций IIFE — создание такой среды, в которой переменные и функции, объявленные в рамках IIFE, не будут доступны извне. Такое поведение действительно полезно для стандартных переменных, которые не должны быть видимыми вне плагина — но не для методов. В рамках IIFE нам понадобится способ сделать методы доступными из внешнего мира.

Создаваемый нами плагин работает с коллекциями jQuery. Чтобы спроектировать новый метод для коллекций jQuery, нужно назначить их свойству с именем `$.fn`. После внесения соответствующих изменений в файл JavaScript получим код, показанный в листинге 12.1.

Листинг 12.1. Первая версия плагина Jqia Context Menu

```
(function($) {
  var defaults = {
    idMenu: null,
    bindLeftClick: false
  };
  $.fn.init = function() {
    // Код функции
  };
  $.fn.destroy = function() {
    // Код функции
  };
})(jQuery);
```

Код, представленный в этом листинге, ничего не делает — ведь мы не определили тела функций `init()` и `destroy()`. Тем не менее их уже можно вызывать как обычные методы ядра jQuery.

Играть с новыми игрушками всегда интересно, так что вы можете попробовать добавить в оба метода по выражению `console.log()` (просто чтобы убедиться в их выполнении), подключить библиотеку jQuery и файл `jquery.jqia.contextMenu.js` к веб-странице и написать примерно такое выражение, чтобы испытать наш проект в действии:

```
$('p').init();
```

К сожалению, если вы откроете страницу, то сразу после вывода `console.log()` получите тревожное сообщение об ошибке. Причина в следующем: вы выбрали для метода слишком распространенное имя. Оно настолько часто встречается, что есть даже в ядре jQuery. Наш метод вступил в конфликт с определенным ранее методом jQuery `init()`, и эта проблема подводит нас к важному моменту — к пространствам имен методов.

Правильный выбор пространства имен для плагина — важный этап процесса разработки. Он позволит свести к минимуму вероятность конфликта с другими плагинами и с методами ядра jQuery.

Приняв во внимание данный совет, вы могли бы переименовать методы в `jqiaCustomMenuInit()` и `jqiaCustomMenuDestroy()`. Теперь можно без помех

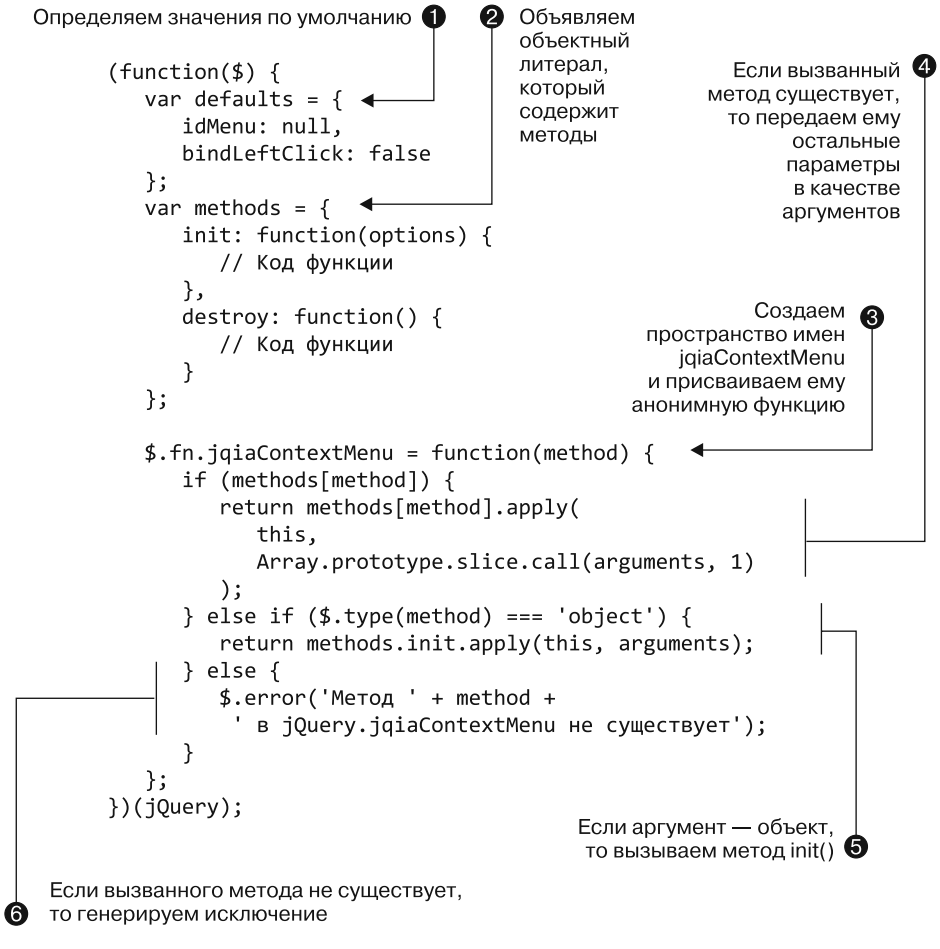
вызывать оба, поскольку ни один метод jQuery не будет с ними конфликтовать. Данное изменение действительно работает, но jQuery не рекомендует объявлять несколько пространств имен, чтобы не загромождать `$.fn`. Есть другое решение, и оно реализовано во многих популярных плагинах: собрать все методы плагина в один объектный литерал (обычно с именем `methods`) и вызывать их, используя один общий метод, получающий в качестве параметра строку с именем того частного метода, который надо выполнить.

Чтобы вы поняли, как все работает, предположим: этот наш главный метод называется `jqiaContextMenu`. Используя его, можно выполнить метод `destroy()` таким образом (на время забудем о том, что у него должны быть параметры):

```
$('#element').jqiaContextMenu('destroy');
```

Следуя данному правилу, можем переделать код из листинга 12.1 и получим листинг 12.2.

Листинг 12.2. Пересмотренный вариант плагина Jqia Context Menu



В самой внешней анонимной функции мы задаем значения по умолчанию для параметров плагина ❶. Затем объявляем объектный литерал, содержащий методы, которые нам нужны, пока что пустые ❷.

Следующая часть кода — самая интересная и действительно красивая. Прежде всего мы присваиваем анонимную функцию новому свойству `$.fn` с именем `jqiaContextMenu` ❸. Таким образом, создаем всего одно имя для всех методов, вместо того чтобы присваивать уникальное имя каждому из них в отдельности — в соответствии с рекомендациями. В этой функции мы проверяем, является ли первый из переданных ей аргументов, `method`, значением одного из свойств переменной `methods` ❹. Если да, то применяем специальную технологию JavaScript с добавлением функций `apply()` и `call()` для вызова нужного метода. Функция `apply()` используется для присвоения контексту функции (`this`) набора элементов из коллекции jQuery и для передачи вызванному методу всех полученных параметров плагина, кроме первого (поскольку первый параметр — имя вызываемого метода). Поскольку аргументы не являются настоящим массивом (это так называемый массивоподобный объект), можно задействовать метод разделения массива `slice()` и метод `call()`, чтобы удалить первый аргумент из списка.

ПРИМЕЧАНИЕ

Если вы не знаете или забыли, что такое методы apply() и call(), то почитайте о них в приложении.

Если первое условие не выполнено, то проверяем, является ли первый параметр объектом ❺, и если да, то вызываем метод `init()`, которому перенаправляем все параметры, переданные в `jqiaContextMenu()`. Мы исходим из того, что если пользователь вызвал функцию `jqiaContextMenu()` и передал ей объект с параметрами, то он, скорее всего, хотел инициализировать плагин и вызвать его действие по умолчанию. В этом случае также вызывается метод `apply()`, чтобы назначить контексту функции набор элементов из коллекции jQuery и передать аргументы вызванному методу. Наконец, если ни одно условие не выполнено, то генерируем исключение с помощью сервисной функции `$.error()` ❻.

Как видите, изменения, внесенные нами в код, эффективны и позволяют использовать только одно пространство имен (`jqiaContextMenu`) для всех методов. Правило, подобное рассмотренному в данной главе, действует и для событий, связанных с нашим плагином, а также для хранящихся в нем данных. Обсудим это подробнее.

12.3.5. Пространства имен для событий и данных

О пространстве имен событий мы узнали в главе 6. Данное свойство особенно полезно при создании авторского плагина. Если в нем есть обработчики событий, то считается хорошим тоном создавать для них отдельное пространство имен. Следуйте этому правилу — и впоследствии при необходимости сможете отменить обработку какого-либо события, не опасаясь повлиять на остальные плагины, которые, возможно, обрабатывают это же событие.

Помимо обработки событий, ряд плагинов хранят данные в одном или нескольких элементах веб-страницы. Это бывает полезно, если нужно отслеживать состояние элемента или проверить, вызывался ли уже плагин для данного элемента. Указанная операция выполняется с помощью метода jQuery `data()`, описанного в главе 4, — данные легко получить и легко удалить.

Выполняя все описанные ранее рекомендации, мы в итоге получим отлично структурированный плагин. Но нам до сих пор не хватает самого важного — тела методов. Начнем с `init()`.

Метод init() плагина Jqia Context Menu

Метод `init()` выполняет следующие действия.

1. Проверяет, переданы ли плагину параметры, особенно обязательные.
2. Объединяет переданные параметры с их значениями по умолчанию.
3. Проверяет, инициализирован ли плагин для выбранных элементов.
4. Сохраняет параметры в элементах коллекции jQuery.
5. Отслеживает событие щелчка правой кнопкой мыши — мы его назвали `contextmenu` — для элементов из коллекции jQuery. Если событие происходит, то показывает специальное контекстное меню. Дополнительно может отслеживать также событие нажатия левой кнопки мыши (`click`).
6. Скрывает специальное меню, если событие `click` возникает *за пределами* элементов из коллекции jQuery.

Для выполнения первого пункта необходимо убедиться, что свойство `idMenu`, содержащее ID элемента, который будет играть роль специального меню, задано и данный элемент существует на странице. Это делается с помощью следующего кода.

```
if (!options.idMenu) {
  $.error('Меню не определено');
} else if ($('#' + options.idMenu).length === 0) {
  $.error('Данное меню не существует');
}
```

В коде мы использовали свойство `length`, чтобы проверить, существует ли на странице данный элемент.

Реализовать второй пункт тоже легко. Для слияния значений нужно лишь вызвать сервисную функцию jQuery `extend()`:

```
options = $.extend(true, {}, defaults, options);
```

Как видите, в этом выражении параметр `options` применен дважды, чтобы не создавать дополнительную (лишнюю) переменную.

Третий и четвертый пункты тесно взаимосвязаны. После того как плагин инициализирован для выбранных элементов, мы используем метод jQuery `data()` для хранения параметров под одним и тем же именем. В данном случае мы также используем этот метод, чтобы проверить, был ли элемент уже инициализирован нашим плагином. Можно применить сохраненную информацию и другую функ-

циональность, которую вы, возможно, захотите добавить, последующее изменение конфигурации для данного элемента.

Для сохранения данных можно написать следующее выражение:

```
this.data('jqiaContextMenu', options);
```

При первом запуске плагина на странице вы точно знаете: ни один элемент еще не инициализирован. Но что произойдет, если запустить Jqia Context Menu второй раз для тех же элементов? Двойная инициализация элемента относится к тем ситуациям, которых лучше избегать, — иначе, например, один и тот же обработчик события может быть добавлен дважды. Необходимо убедиться, что ни у одного элемента из набора еще нет сохраненных данных, использующих пространство имен нашего плагина (`jqiaContextMenu`). Эту задачу выполняет следующий код:

```
if (
  this.filter(function() {
    return $(this).data('jqiaContextMenu');
  }).length !== 0
) {
  $.error('Плагин уже инициализирован');
}
```

Представленный короткий фрагмент тем не менее дает возможность подчеркнуть важный момент — значение `this` в плагине. Когда функция прикреплена к `$.fn`, ключевое слово `this` означает экземпляр jQuery (коллекцию jQuery, для которой вызван плагин). Можно использовать любой метод библиотеки непосредственно, не оборачивая его в метод `$()` (например, `$(this)`). Это справедливо и для функций, определенных в объекте `methods`, поскольку вы изменяете их контекст с помощью `apply()`. Если не применять `apply()`, то внутри `init()` ключевое слово `this` будет означать объект `methods`.

В зависимости от того, как вы структурируете плагин, в функции обратного вызова, выполняемой в плагине, ключевое слово `this` означает определенный элемент DOM. В своем коде вы можете задействовать метод jQuery `filter()`, перебирающий все элементы набора. При первом вызове `this` в функции обратного вызова будет ссылаться на первый элемент набора, при втором — на второй элемент и т. д. Именно поэтому в анонимной функции, переданной в `filter()`, `this` передается как аргумент `$()`: чтобы использовать метод jQuery `data()`.

Теперь, когда вы лучше понимаете внутреннее содержание плагины, jQuery, продолжим изучение метода `init()`.

Пятый пункт — ядро нашего проекта. Чтобы его выполнить, нужно назначить функцию обратного вызова для события `contextmenu`, которое обычно будет возникать при нажатии пользователем правой кнопки мыши. Как вы, наверное, помните, мы также предусматриваем возможность отслеживать событие `click`, которое происходит при нажатии левой кнопки мыши. В зависимости от параметров, переданных разработчиком, может потребоваться отслеживать оба события — и `contextmenu`, и `click`.

В самой функции обратного вызова нужно отменить стандартное поведение — иначе, кроме нашего контекстного меню, будет появляться и стандартное.

Когда это будет готово, нужно будет назначить позицию нашего меню в зависимости от положения указателя мыши в момент нажатия кнопки (данная информация хранится в объекте `Event`, передаваемом в функцию обратного вызова). И последнее, что надо будет сделать, — вывести само меню.

Код, выполняющий все эти действия, представлен ниже:

```
this.on(
    'contextmenu.jqiaContextMenu' +
      (options.bindLeftClick ? ' click.jqiaContextMenu' : ''),
    function(event) {
      event.preventDefault();
      $('#' + options.idMenu)
          .css({
              top: event.pageY,
              left: event.pageX
          })
          .show();
    }
);
```

В этом коде для проверки того, нужно ли также отслеживать нажатие левой кнопки, используется тернарный оператор. Объект передается методу jQuery `css()`, который назначает позицию меню (отдельно нужно в файле CSS назначить для меню стиль `position: absolute`). Мы не указываем единицы измерения (в данном случае пикселы), поскольку по умолчанию jQuery предполагает, что значения заданы в пикселах.

И последнее: нужно скрыть специальное контекстное меню при нажатии любой кнопки мыши за пределами элементов, инициализированных плагином Jqia Context Menu. То есть надо назначить всем элементам страницы, кроме инициализированных нашим плагином, обработчик события, который будет скрывать наше меню. Назначить обработчик события для каждого элемента страницы — значит создать серьезные проблемы с производительностью, поэтому мы воспользуемся преимуществом делегирования событий. Мы назначим только один обработчик для корня документа — элемента `html`:

```
$('html').on(
  'contextmenu.jqiaContextMenu click.jqiaContextMenu',
  function() {
    $('#' + options.idMenu).hide();
  }
);
```

Когда данный код займет свое место в файле, вам может показаться, что метод `init()` готов. Но это не так.

Наш проект в его теперешнем состоянии содержит серьезную ошибку. Когда пользователь нажимает кнопку мыши на инициализированном элементе, открывается специальное контекстное меню. Затем по технологии всплытия событие распространяется дальше в сторону корня дерева DOM. Когда оно дойдет до элемента `html`, сработает прикрепленная к нему функция обратного вызова и меню будет скрыто. В результате наше меню появится всего на несколько миллисекунд (вы даже не успе-

ете его увидеть). Чтобы исправить эту ошибку, нужно вызвать `event.stopPropagation()` из обработчика событий для инициализированных элементов.

Когда вы внесете соответствующие изменения, метод `init()` будет готов. Теперь займемся разработкой метода `destroy()`. (Если вам интересно, как выглядит готовый плагин, перейдите к листингу 12.3.)

Метод destroy() плагина Jqia Context Menu

Данный метод отвечает за освобождение ресурсов, использованных нашим плагином, — данных, сохраненных в инициализированных элементах, и назначенных им обработчиков событий, в том числе и элементу `html`. Кроме того, прежде чем завершить работу метода `destroy()`, надо убедиться, что наше контекстное меню скрыто — иначе оно будет отображаться до следующего обновления страницы.

Один из возможных вариантов кода, реализующего все эти задачи, выглядит так:

```
this
  .each(function() {
    var options = $(this).data('jqiaContextMenu');
    if (options !== undefined) {
      $('#' + options.idMenu).hide();
    }
  })
  .removeData('jqiaContextMenu')
  .add('html')
  .off('.jqiaContextMenu');
```

Прежде всего мы перебираем все элементы набора и скрываем их контекстные меню (если соответствующий элемент был инициализирован нашим плагином). Затем удаляем данные, сохраненные в каждом элементе. На этот раз не требуется перебирать элементы по одному, чтобы выполнить дополнительную проверку, так что можно воспользоваться методом jQuery `removeData()`.

Последняя операция — удаление всех обработчиков, связанных с событиями из пространства имен `jqiaContextMenu`. Добавляем элемент `html` к нашему набору jQuery и вызываем для этого расширенного набора метод `off()`, передавая ему строку `".jqiaContextMenu"`.

Наблюдательные читатели могли заметить, что в приведенных фрагментах кода неоднократно в разных целях использовалась строка `"jqiaContextMenu"`. Во избежание таких повторений можно сохранить ее в частной переменной (доступной только внутри плагина) и затем внести в код соответствующие изменения.

Если такое выражение добавить под определение переменной `defaults`:

```
var namespace = 'jqiaContextMenu';
```

то тело метода `destroy()` можно переписать так:

```
this
  .each(function() {
    var options = $(this).data(namespace);
    if (options !== undefined) {
```

```
      $('#' + options.idMenu).hide();
    }
  })
  .removeData(namespace)
  .add('html')
  .off('.' + namespace);
```

Теперь наш плагин работает. Но есть еще два изменения, которые следует внести.

12.3.6. Поддержка цепочек

В этой книге мы постоянно применяем цепочки jQuery, позволяющие выполнять несколько операций в одном выражении. Но методы нашего плагина не возвращают значений, так что по умолчанию возвращается `undefined`. Из-за этого разработчик, использующий наш плагин, после вызова `jqiaContextMenu()` не сможет вызвать ни один метод.

Наш плагин должен занять свое место в цепочке. Для этого потребуется внести в него маленькое, но бесценное изменение. Нужно всего лишь сделать так, чтобы его методы всегда возвращали ключевое слово `this`. В применении к методу `destroy()` после:

```
.off('. ' + namespace);
```

требуется добавить строку:

```
return this;
```

Подобное изменение необходимо внести и в метод `init()`.

Благодаря этому мы обеспечим возможность продолжения цепочки после применения Jqia Context Menu и последующие методы того же выражения будут оперировать тем же набором элементов.

12.3.7. Перенос стандартных значений в открытый доступ

У нашего плагина не так уж много параметров, но по мере его усложнения может оказаться, что приходится многократно передавать разным элементам один и тот же большой набор значений.

Можно внести следующее улучшение: выставить настройки по умолчанию так, чтобы разработчик, использующий плагин, мог их переопределить. Благодаря этому изменению разработчику нужно будет только передать объект с различными параметрами при каждом вызове плагина.

Для реализации этого усовершенствования наш проект нуждается всего в двух изменениях. Первое из них касается переменной `defaults`. Чтобы сделать ее доступной извне, нужно назначить ее свойству `$.fn`. Напоминаем: нельзя нарушать

правило единого пространства имен для всего плагина, поэтому сделаем объект, содержащий стандартные значения, свойством `jqiaContextMenu`. Таким образом, вместо:

```
var defaults = {
  // Список параметров
};
```

получим:

```
$.fn.jqiaContextMenu.defaults = {
  // Список параметров
};
```

Нам также нужно добавить определение стандартной конфигурации после оператора, создающего пространство имен (`$.fn.jqiaContextMenu = function(method) {}`).

После этих изменений больше нельзя будет получить стандартные значения через переменную `defaults`. Нужно во всем коде заменить `defaults` на `$.fn.jqiaContextMenu.defaults`. В нашем проекте такое обращение только одно, в теле метода `init()`, где мы объединяем параметры, переданные плагину, с теми, что задаются по умолчанию. Измененное выражение будет выглядеть так:

```
options = $.extend(true, {}, $.fn.jqiaContextMenu.defaults, options);
```

Как можно будет использовать стандартные значения после внесения всех изменений? Предположим, мы хотим, чтобы плагин Jqia Context Menu всегда реагировал на щелчок левой кнопкой мыши. Тогда можно написать так:

```
$.fn.jqiaContextMenu.defaults.bindLeftClick = true;
```

Теперь можно вызвать плагин, передав ему только значение свойства `idMenu`.

После внесения этих изменений наш проект готов. Его окончательный код представлен в листинге 12.3.

Листинг 12.3. Окончательная версия Jqia Context Menu

```
(function($) {
  var namespace = 'jqiaContextMenu';
  var methods = {
    init: function(options) {
      if (!options.idMenu) {
        $.error('Меню не задано');
      } else if ($('#' + options.idMenu).length === 0) {
        $.error('Данное меню не существует');
      }
      options = $.extend(
        true,
        {},
        $.fn.jqiaContextMenu.defaults,
        options
      );
```

```
    if (
      this.filter(function() {
        return $(this).data(namespace);
        }).length !== 0
        ) {
        $.error('Плагин уже инициализирован');
    }
    this.data(namespace, options);
    $('html').on(
      'contextmenu.' + namespace + ' click.' + namespace,
      function() {
        $('#' + options.idMenu).hide();
      }
    );
    this.on(
      'contextmenu.' + namespace +
         (options.bindLeftClick ? ' click.' + namespace : ''),
      function(event) {
        event.preventDefault();
        event.stopPropagation();
        $('#' + options.idMenu)
          .css({
            top: event.pageY,
            left: event.pageX
          })
          .show();
      }
    );
    return this;
  },
  destroy: function() {
    this
      .each(function() {
        var options = $(this).data(namespace);
        if (options !== undefined) {
          $('#' + options.idMenu).hide();
        }
      })
      .removeData(namespace)
      .add('html')
      .off('.' + namespace);
    return this;
  }
};
$.fn.jqiaContextMenu = function(method) {
  if (methods[method]) {
    return methods[method].apply(
      this,
      Array.prototype.slice.call(arguments, 1)
    );
  } else if ($.type(method) === 'object') {
    return methods.init.apply(this, arguments);
  } else {
```

```
      $.error('Метод ' + method +
        ' в плагине jQuery.jqiaContextMenu не существует'
      );
    }
  };
  $.fn.jqiaContextMenu.defaults = {
    idMenu: null,
    bindLeftClick: false
  };
})(jQuery);
```

Код доступен в файле `js/jquery.jqia.contextMenu.js` в исходном коде к книге. Чтобы плагин стал полностью рабочим, необходимо подключить таблицу стилей. Вы найдете ее в файле `css/jquery.jqia.contextMenu.css`. Мы также создали для вас пример (доступен в файле `chapter-12/jqia.contextMenu.html`, предоставленном с книгой), чтобы вы могли поэкспериментировать с использованием плагина. На рис. 12.1 показан результат щелчка правой кнопкой мыши на элементе страницы, инициализированном плагином Jqia Context Menu.

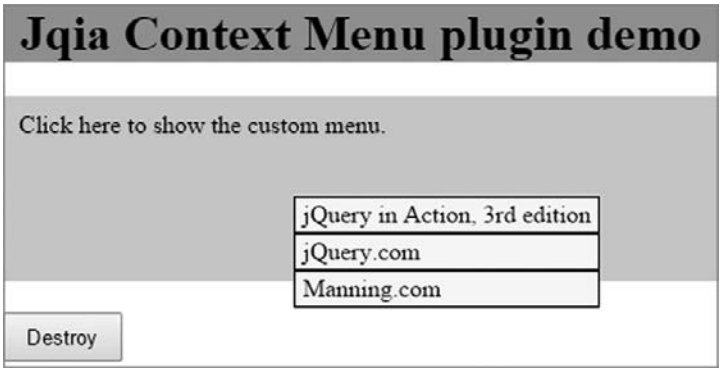


Рис. 12.1. Результат щелчка правой кнопкой мыши на элементе, инициализированном плагином

Нравится ли вам результат? Надеемся, да. Мы также надеемся, что вы получили удовольствие от процесса разработки этого маленького проекта. В следующем разделе мы создадим более сложный плагин jQuery, следуя уже известным рекомендациям.

12.4. Демонстрационный пример: создание слайд-шоу в виде плагина jQuery

В качестве более сложного примера мы разработаем метод jQuery, который позволит веб-программистам быстро создавать страницы со слайд-шоу. Это будет плагин jQuery с именем Jqia Photomatic, и мы создадим тестовую страницу, на которой испытаем его возможности. Когда все будет готово, тестовая страница будет выглядеть так, как показано на рис. 12.2.

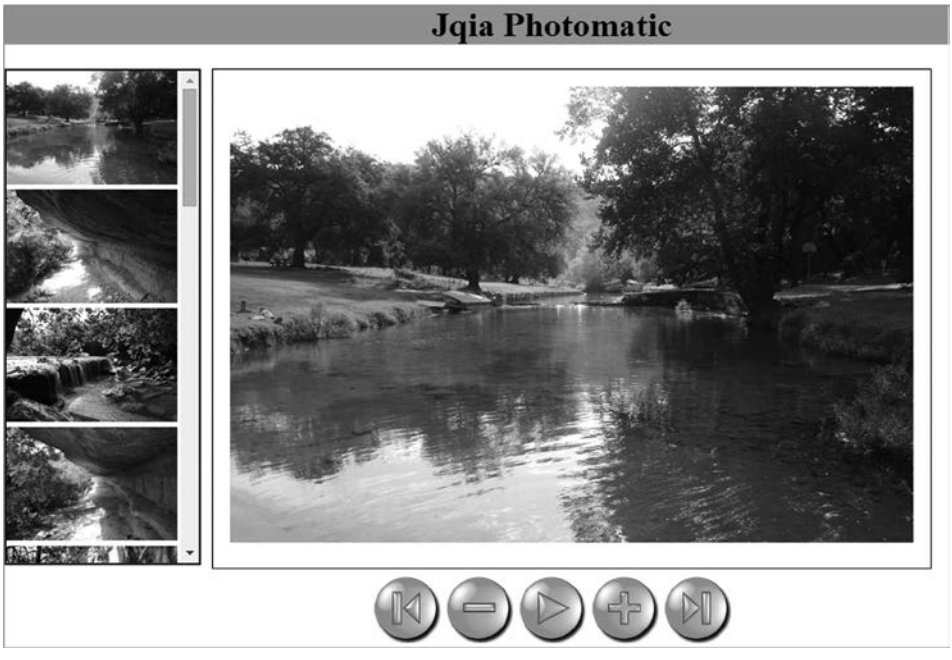


Рис. 12.2. Тестовая страница, на которой мы проверим все возможности плагина Jqia Photomatic

Эта страница будет состоять из таких компонентов, как:

- набор мини-слайдов;
- полноразмерный вариант изображения одного из мини-слайдов;
- набор кнопок для ручного управления слайдами, а также для запуска и остановки слайд-шоу в автоматическом режиме.

Поведение страницы будет таким:

- если щелкнуть на любом мини-слайде, то откроется соответствующее полноразмерное изображение;
- если щелкнуть на полноразмерном изображении, то откроется следующее изображение;
- если нажать кнопку, то выполнится следующая операция:
  - First (В начало) — показать первую картинку;
  - Previous (Раньше) — показать предыдущую картинку;
  - Next (Дальше) — показать следующую картинку;
  - Last (В конец) — показать последнюю картинку;
  - Play (Воспроизвести) — запустить автоматический показ изображений до следующего нажатия кнопки;
- когда достигается конец списка, при любой попытке просмотреть следующее изображение открывается первое изображение, а при достижении начала спи-

ска при любой попытке просмотреть предыдущее — открывается последнее. Если нажать кнопку Next (Дальше), когда открыто последнее изображение, то откроется первое. Если нажать кнопку Previous (Раньше), когда открыто первое изображение, — откроется последнее.

Мы построим плагин так, чтобы разработчики могли назначать элементы любым удобным способом и затем сообщали плагину, какой элемент страницы и для каких целей будет использован. Более того, для предоставления разработчикам как можно большей свободы действий мы определим плагин так, чтобы в роли мини-слайдов могла выступать любая коллекция jQuery, содержащая собранные вместе изображения, — как это сделано на нашей тестовой странице.

Для начала представляем синтаксис метода плагина Jqia Photomatic.

Синтаксис метода: jqiaPhotomatic

`jqiaPhotomatic(options)`

Создает набор мини-слайдов, а также назначает элементы страницы, заданные в хеш-объекте `options`, элементами управления Jqia Photomatic.

Параметры

`options` (Объект) Хеш-объект, определяющий свойства Jqia Photomatic. Подробнее см. табл. 12.1.

Возвращает

Коллекцию jQuery.

Поскольку количество параметров для управления плагином Jqia Photomatic больше нуля (часть из них могут быть не заданы), мы используем для их передачи хеш вариантов, который обсуждался в подразделе 12.3.3. Его возможные значения представлены в табл. 12.1.

Таблица 12.1. Параметры плагина Jqia Photomatic

Параметр	Описание
firstControl	(Селектор) Селектор jQuery, идентифицирующий элементы DOM, которые будут использоваться в качестве элемента управления (кнопки) First (В начало). Если не задан, то такой элемент управления не создается
LastControl	(Селектор) Селектор jQuery, идентифицирующий элементы DOM, которые будут использоваться в качестве элемента управления Last (В конец). Если не задан, то такой элемент управления не создается
nextControl	(Селектор) Селектор jQuery, идентифицирующий элементы DOM, которые будут использоваться в качестве элемента управления Next (Дальше). Если не задан, то такой элемент управления не создается
photoElement	(Селектор) Селектор jQuery, идентифицирующий элемент img, который будет использоваться для показа полноразмерного изображения. Если не задан, то по умолчанию используется селектор jQuery img.photomatic-photo

Продолжение ⇩

Таблица 12.1 (продолжение)

Параметр	Описание
playControl	(Селектор) Селектор jQuery, идентифицирующий элементы DOM, которые будут использоваться в качестве элемента управления Play (Воспроизвести). Если не задан, то такой элемент управления не создается
previousControl	(Селектор) Селектор jQuery, идентифицирующий элементы DOM, которые будут использоваться в качестве элемента управления Previous (Раньше). Если не задан, то такой элемент управления не создается
transformer	(Функция) Функция, используемая для преобразования URL мини-слайда в URL соответствующего полноразмерного изображения. Если не задана, то по умолчанию в исходном URL все фрагменты thumbnail заменяются на photo
delay	(Число) Интервал между переходами для автоматического воспроизведения. Измеряется в миллисекундах. По умолчанию равен 3000

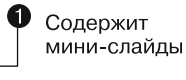
Прежде чем углубляться в разработку самого плагина Jqia Photomatic, сначала создадим тестовую страницу для него.

12.4.1. Создание разметки

Первое, что нужно сделать при создании нашего плагина, — создать страницу, на которой он будет использоваться. Код этой страницы, доступный в файле `chapter-12/jqia.photomatic.html`, предоставленном с книгой, дается в листинге 12.4.

Листинг 12.4. Код тестовой страницы Photomatic, показанной на рис. 12.2

```
<!DOCTYPE html>
<html>
  <head>
     <meta charset="utf-8" />
     <title>Jqia Photomatic - jQuery в действии, 3-е издание</title>
     <link rel="stylesheet" href="../css/main.css"/>
     <link rel="stylesheet" href="../css/jquery.jqia.photomatic.css"/>
  </head>
  <body>
     <h1 class="header">Jqia Photomatic</h1>

     <div id="thumbnails-pane">
        <img src="../images/thumbnails/img_1.jpg" alt="" />
        <img src="../images/thumbnails/img_2.jpg" alt="" />
        <img src="../images/thumbnails/img_3.jpg" alt="" />
        <img src="../images/thumbnails/img_4.jpg" alt="" />
        <img src="../images/thumbnails/img_5.jpg" alt="" />
        <img src="../images/thumbnails/img_6.jpg" alt="" />
        <img src="../images/thumbnails/img_7.jpg" alt="" />
```



```
        <img src="../images/thumbnails/img_8.jpg" alt="" />
        <img src="../images/thumbnails/img_9.jpg" alt="" />
        <img src="../images/thumbnails/img_10.jpg" alt="" />
        <img src="../images/thumbnails/img_11.jpg" alt="" />
        <img src="../images/thumbnails/img_12.jpg" alt="" />
    </div>

    <div>
        <img id="photo-display" src=""
             title="Нажмите, чтобы увидеть следующее фото" />
    </div>

    <div id="button-bar">
        <img src="../images/button.first.png" id="first-button"
             alt="Первое фото" />
        <img src="../images/button.previous.png" id="previous-button"
             alt="Следующее фото" />
        <img src="../images/button.play.png" id="play-button"
             alt="Запустить/остановить слайд-шоу" />
        <img src="../images/button.next.png" id="next-button"
             alt="Следующее фото" />
        <img src="../images/button.last.png" id="last-button"
             alt="Последнее фото" />
    </div>

    <script src="../js/jquery-1.11.3.min.js"></script>
    <script src="../js/jquery.jqia.photomatic.js"></script>
    <script>
        $('#thumbnails-pane img').jqiaPhotomatic({
            photoElement: '#photo-display',
            previousControl: '#previous-button',
            nextControl: '#next-button',
            firstControl: '#first-button',
            lastControl: '#last-button',
            playControl: '#play-button',
            delay: 1000
        });
    </script>
</body>
</html>
```

❷ Определяет элемент img для полноразмерных фото

Содержит элементы, которые будут использоваться для управления ❸

Вызывает расширение Photomatic ❹

Если применить принципы ненавязчивого JavaScript и вынести все стили во внешнюю таблицу стилей, то разметка будет аккуратной и простой.

Разметка HTML состоит из контейнера, содержащего мини-слайды ❶, элемента `img` (сначала без атрибута `src`) для полноразмерного фото ❷ и набора изображений ❸, которые будут использоваться для управления слайд-шоу. Все остальное обеспечит наш новый плагин. Размещенный на странице код сценария состоит всего из одного выражения; в нем вызывается плагин, и ему передается несколько параметров ❹.

Теперь, когда разметка готова, можно заняться непосредственно плагином.

12.4.2. Разработка Jqia Photomatic

В начале разработки этого плагина нам понадобится такой же «скелет», как и тот, что мы использовали при создании Jqia Context Menu, только с другим пространством имен:

```
(function($){
  var methods = {
    init: function() {
    }
  };
  $.fn.jqiaPhotomatic = function(method) {
    if (methods[method]) {
      return methods[method].apply(
        this,
        Array.prototype.slice.call(arguments, 1)
      );
    } else if ($.type(method) === 'object') {
      return methods.init.apply(this, arguments);
    } else {
      $.error('Метод ' + method +
        ' не существует в jQuery.jqiaPhotomatic'
      );
    }
  };
})(jQuery);
```

В этом плагине нужна только одна функция — инициализации. Но чтобы оставить возможность в будущем расширить его, мы используем переменную `methods`, как в предыдущем проекте.

Внутри функции `init()` объединим параметры, полученные при вызове плагина, со значениями по умолчанию, описанными в табл. 12.1. Результат записывается в общий объект `options`, к которому и выполняются обращения в остальной части функции.

При вызове плагина может быть полезно переопределить некоторые значения, заданные по умолчанию (например, свойство `delay`), так что мы сделаем их доступными извне, как показано здесь:

```
$.fn.jqiaPhotomatic.defaults = {
      photoElement: 'img.photomatic-photo',
      transformer: function(name) {
        return name.replace('thumbnail', 'photo');
      },
      nextControl: null,
      previousControl: null,
      firstControl: null,
      lastControl: null,
      playControl: null,
      delay: 3000
};
```

Подобно тому, как это было сделано в плагине Jqia Context Menu, выполняем слияние с помощью метода jQuery `$.extend()`:

```
options = $.extend(true, {}, $.fn.jqiaPhotomatic.defaults, options);
```

Когда будет выполнено данное выражение, переменная `options` будет содержать значения по умолчанию из встроенного хеш-объекта, переопределенные значениями, полученными при вызове плагина.

Следует также проследить еще за несколькими моментами. Чтобы наш плагин «понимал», что такое *следующее* и *предыдущее* изображение, нужен не только список мини-слайдов, но также индикатор, указывающий *текущее* изображение.

Список мини-слайдов представляет собой коллекцию jQuery. Именно ее обрабатывает наш метод — или, во всяком случае, должен обрабатывать. Мы не можем знать, что именно разработчики сайта поместят в эту коллекцию, поэтому отфильтруем из нее только элементы `img`. Эту операцию можно выполнить с помощью селектора и метода jQuery `filter()`. Но где хранить эти два результата?

Мы могли бы без труда ввести еще одну переменную, но, чтобы держать исходные параметры в одном месте, лучше создадим дополнительные свойства объекта `options`. Для этого нужно немного изменить вызов метода `extend()`:

```
options = $.extend(
  true,
  {},
  $.fn.jqiaPhotomatic.defaults,
  options,
  {
     current: 0,
    $thumbnails: this.filter('img')
  }
);
```

Обратите внимание, как мы разместили объект, содержащий текущее изображение, и список мини-слайдов как последний аргумент метода `extend()`, поскольку он определяет приоритет свойств при слиянии. Свойство, в котором хранится список, мы назовем `$thumbnails`, поскольку его значением является коллекция jQuery.

Теперь, когда начальное состояние задано, можно перейти к самой главной части плагина — описанию свойств элементов управления, мини-слайдов и полноразмерного фото.

ПРИМЕЧАНИЕ

Все это положение вещей возможно благодаря замыканиям. Они нам уже встречались, но если вы до сих пор не чувствуете себя с ними уверенно, то советуем заглянуть в приложение. Вам следует хорошо понимать, что такое замыкания. Не только для того, чтобы закончить реализацию плагина Jqia Photomatic, но и для создания других, хоть сколько-нибудь сложных плагинов.

Теперь нужно назначить обработчики событий элементам управления и другим элементам плагина, созданным нами. Поскольку, когда объявляются функции, соответствующие этим обработчикам, переменная `options` находится в области видимости, каждый обработчик будет частью замыкания, включающего в себя и переменную `options`. Вы можете быть уверены, что, даже если она станет временной, состояние, которое она описывает, будет доступно для всех созданных нами обработчиков событий.

Кстати об обработчиках. Вот список обработчиков событий `click`, которые нам надо назначить различным элементам.

- При нажатии мини-слайда должно открываться его полноразмерное фото.
- При нажатии полноразмерного снимка должно показываться следующее фото.
- При нажатии элемента, которому назначена функция кнопки Previous (Раньше), должно показываться предыдущее изображение. Если перед этим было показано первое изображение списка, то после нажатия Previous (Раньше) должно показываться последнее изображение.
- При нажатии элемента управления Next (Дальше) должно показываться следующее изображение. Если перед этим было показано последнее изображение списка, то после нажатия Next (Дальше) должно быть показано первое изображение.
- При нажатии элемента управления First (В начало) должно быть показано первое изображение в списке.
- При нажатии элемента управления Last (В конец) должно быть показано последнее изображение в списке.
- При нажатии элемента управления Play (Воспроизвести) должно запускаться автоматическое воспроизведение слайд-шоу, в котором переход от одного фото к другому осуществляется с задержкой, заданной в параметрах плагина. При следующем нажатии этого элемента управления автоматическое воспроизведение слайд-шоу должно прекратиться.

Просматривая данный список, вы, вероятно, сразу обратили внимание на то, что у всех этих обработчиков событий есть общая черта: они отображают полноразмерное фото того или иного мини-слайда. Будучи хорошими разработчиками, вы, конечно же, захотите вынести общий код в отдельную функцию, чтобы не повторять каждый раз одно и то же.

Вы не захотите нарушать пространство имен — ни глобальное, ни пространство имен `$` — ради функции, которая будет вызываться только из нашего кода. Здесь снова поможет преимущество JavaScript как функционального языка — он позволит определить новую функцию внутри функции плагина. Таким образом мы ограничим ее область видимости только функцией плагина (чего мы и хотели).

По этой причине мы определим функцию с именем `showPhoto()` внутри плагина, но вне метода `init()`. Функция будет принимать два параметра: первый — полу-

ченный при вызове плагина, второй — индекс мини-слайда, который должен быть показан в полном размере. Код функции будет таким:

```
function showPhoto(options, index) {
  $(options.photoElement).attr(
    'src',
    options.transformer(options.$thumbnails[index].src)
  );
  options.current = index;
}
```

Эта новая функция, получая индекс мини-слайда, чье полноразмерное фото должно быть показано, использует значения объекта `options` для выполнения следующих действий.

1. Находит атрибут `src` мини-слайда с заданным индексом.
2. Пропускает это значение через функцию `transformer`, чтобы преобразовать его из URL мини-слайда в URL полноразмерного фото.
3. Присваивает результат преобразования атрибуту `src` элемента с полноразмерным изображением.
4. Записывает индекс отображенного фото как новый текущий индекс.

Теперь, когда у нас есть эта удобная функция, можно описать перечисленные выше обработчики событий. Начнем с функциональности самих мини-слайдов, для которых достаточно просто отображать соответствующие полноразмерные фото:

```
options.$thumbnails.click(function() {
  showPhoto(options, options.$thumbnails.index(this));
});
```

В данном обработчике событий мы получаем значение индекса мини-слайда, передавая методу jQuery `index()` тот элемент, на котором произошел щелчок кнопкой мыши.

Чтобы при нажатии фото показывалось следующее по списку изображение, используем столь же простой код:

```
$(options.photoElement + ', ' + options.nextControl).click(function() {
  showPhoto(options, (options.current + 1) % options.$thumbnails.length);
});
```

Мы создаем здесь обработчик события `click`, в котором вызываем функцию `showPhoto()` и передаем ей параметры — объект `options` и значение следующего индекса. Обратите внимание, как мы использовали оператор JavaScript `%` (остаток от деления), чтобы вернуть список к началу после того, как будет достигнут его конец.

Наблюдательный читатель мог заметить: в выражении использован другой селектор. Мы сделали так по той причине, что поведение кнопки Next (Далее) — точно такое же. Поэтому мы оптимизировали инструкцию, объединив два селектора в один с помощью запятой.

Обработчики событий для элементов управления First (В начало), Previous (Ранее) и Last (В конец) строятся по одному и тому же шаблону: определить индекс мини-слайда, полноразмерное фото которого надо показать, и вызвать для этого индекса функцию showPhoto():

```
$(options.previousControl).click(function() {
  showPhoto(
    options,
    options.current === 0 ?
      options.$thumbnails.length - 1 :
      options.current - 1
  );
});
$(options.firstControl).click(function() {
  showPhoto(options, 0);
}).triggerHandler('click');
$(options.lastControl).click(function() {
  showPhoto(options, options.$thumbnails.length - 1);
});
```

Особого внимания в данном коде заслуживает одна строка — та, где используется triggerHandler(). Мы вызываем этот метод в процессе выполнения плагина для загрузки начального фото в контейнер изображений.

Реализовать элемент управления Play (Воспроизвести) несколько сложнее. Вместо того чтобы показывать определенное фото, данный элемент запускает последовательный просмотр всего набора изображений, который при следующем нажатии кнопки должен быть остановлен. Рассмотрим код, выполняющий эти действия:

```
var tick;
$(options.playControl).click(function() {
  var $this = $(this);
  if ($this.attr('src').indexOf('play') !== -1) {
    tick = window.setInterval(
      function() {
        $(options.nextControl).triggerHandler('click');
      },
      options.delay
    );
    $this.attr(
      'src',
      $this.attr('src').replace('play', 'pause')
    );
  } else {
    window.clearInterval(tick);
    $this.attr(
      'src',
      $this.attr('src').replace('pause', 'play')
    );
  }
});
```

Вначале мы используем атрибут изображения `src`, чтобы решить, какую операцию выполнять. Если в атрибуте `src` содержится фрагмент `play`, то надо запустить слайд-шоу, иначе — остановить его.

В теле конструкции `if` задействуем метод JavaScript `setInterval()`, чтобы функция непрерывно запускалась через заданные интервалы. Ссылку на данный таймер сохраняем в переменной `tick` для последующего применения. В анонимной функции, передаваемой методу `setInterval()`, мы имитируем нажатие кнопки мыши на элементе управления Next (Дальше) для перехода к следующему фото. Это происходит при каждом вызове функции в `setInterval()`.

Последнее выражение внутри `if` обновляет атрибут `src` элемента управления, чтобы заменить картинку на пиктограмму паузы. Данное изменение позволяет перейти к разделу кода `else`, где выполняются операции кнопки Play (Воспроизвести) для ее второго режима работы.

В разделе `else` нужно остановить слайд-шоу. Для этого мы обнуляем интервал задержки с помощью метода `clearInterval()`, передав туда `tick`, и восстанавливаем исходную пиктограмму на кнопке Play (Воспроизвести).

И последнее, что остается сделать, — позволить продолжать цепочку методов после использования нашего плагина. Для этого он должен возвращать исходный набор элементов. Мы достигнем результата с помощью одной строки:

```
return this;
```

Можно воспользоваться моментом и сплясать победный танец — мы наконец-то закончили! Вы и не думали, что это будет так просто, правда? Готовый код плагина описан в листинге 12.5 и доступен в файле `js/jquery.jqia.photomatic.js`, предоставленном с книгой.

Листинг 12.5. Полная реализация плагина Jqia Photomatic

```
(function($) {
  function showPhoto(options, index) {
    $(options.photoElement).attr(
      'src',
      options.transformer(options.$thumbnails[index].src)
    );
    options.current = index;
  }
  var methods = {
    init: function(options) {
      options = $.extend(
        true,
        {},
        $.fn.jqiaPhotomatic.defaults,
        options,
        {
          current: 0,
          $thumbnails: this.filter('img')
        }
      );
```

```
    options.$thumbnails.click(function() {
      showPhoto(options, options.$thumbnails.index(this));
    });
    $(options.photoElement + ', ' + options.nextControl).click(
      function() {
        showPhoto(
          options,
          (options.current + 1) % options.$thumbnails.length
        );
      }
    );
    $(options.previousControl).click(function() {
      showPhoto(
        options,
        options.current === 0 ?
          options.$thumbnails.length - 1 :
          options.current – 1
      );
    });
    $(options.firstControl).click(function() {
      showPhoto(options, 0);
    }).triggerHandler('click');
    $(options.lastControl).click(function() {
      showPhoto(options, options.$thumbnails.length - 1);
    });
    var tick;
    $(options.playControl).click(function() {
      var $this = $(this);
      if ($this.attr('src').indexOf('play') !== -1) {
        tick = window.setInterval(
          function() {
            $(options.nextControl).triggerHandler('click');
          },
          options.delay
        );
        $this.attr(
          'src',
          $this.attr('src').replace('play', 'pause')
        );
      } else {
        window.clearInterval(tick);
        $this.attr(
          'src',
          $this.attr('src').replace('pause', 'play')
        );
      }
    });
    return this;
  }
};
$.fn.jqiaPhotomatic = function(method) {
  if (methods[method]) {
    return methods[method].apply(
```

```
        this,
        Array.prototype.slice.call(arguments, 1)
      );
    } else if ($.type(method) === 'object') {
      return methods.init.apply(this, arguments);
    } else {
      $.error(
        'Метод ' + method +
        ' не существует в jQuery.jqiaPhotomatic'
      );
    }
  };
  $.fn.jqiaPhotomatic.defaults = {
    photoElement: 'img.photomatic-photo',
    transformer: function(name) {
      return name.replace('thumbnail', 'photo');
    },
    nextControl: null,
    previousControl: null,
    firstControl: null,
    lastControl: null,
    playControl: null,
    delay: 3000
  };
})(jQuery);
```

Это типичный плагин с использованием jQuery — в нем множество действий уместилось в несколько строк компактного кода. И он демонстрирует важный набор приемов — применение замыканий, позволяющих сохранить состояние внутри пространства имен плагина jQuery и создавать приватные функции, которые можно задействовать внутри плагина, не выходя за пределы пространства имен.

Обратите внимание: поскольку мы позаботились о том, чтобы данные не вышли за пределы плагина, можно размещать на странице сколько угодно виджетов Jqia Photomatic, не опасаясь, что они будут мешать друг другу (надо только проследить, чтобы в разметке не дублировались значения ID).

Но закончена ли наша работа? Судите сами и учтите такие возможности.

- Сейчас переход между изображениями происходит мгновенно. Используя знания об анимации и эффектах, можно изменить плагин так, чтобы изображения плавно перетекали одно в другое.
- Пойдем дальше: почему бы не позволить разработчикам самим задавать анимационные эффекты?
- Для обеспечения максимальной гибкости при создании этого плагина мы исходили из соображения, что все элементы HTML уже созданы пользователем. Как насчет того, чтобы создать другой плагин, с меньшей свободой отображения, но в котором все элементы HTML генерируются динамически?

Теперь, когда вы знаете, как реализовать новый метод jQuery, можно узнать больше о том, как создавать сервисные функции.

12.5. Создание сервисных функций

В этой книге понятие «сервисная функция» означает функцию, определенную как свойство `jQuery` (и, следовательно, `$`). Такие функции обычно предназначены для работы с объектами JavaScript, не являющимися элементами, а также для выполнения ряда других операций, которые вообще не требуют каких-либо объектов. Примерами сервисных функций являются уже знакомые нам `$.each()` и `$.noConflict()`. В данном разделе мы узнаем, как создавать собственные сервисные функции.

Добавить функцию в виде свойства объекта или другой функции — то же самое, что создать функцию и затем присвоить ее переменной. Вот код простейшей сервисной функции:

```
$.say = function(what) {
  alert('Я говорю ' + what);
};
```

Это действительно так просто, как кажется. Но в данном способе определения сервисных функций есть свои подводные камни. Если разработчик разместит подобную функцию на странице, где используется Prototype, и вызовет `$.noConflict()` вместо того, чтобы добавить плагин jQuery, то такой разработчик создаст метод функции Prototype `$()`. (Если концепция метода функции вам не совсем ясна, то загляните в приложение.) Как видите, в отличие от плагинов, которые оперируют набором элементов, сервисные функции назначаются `$`, но не `$.fn`.

Эту проблему и ее решение мы уже рассмотрели в подразделе 12.3.2 (намекаем: создайте IIFE). Обсуждать похожую сервисную функцию не имеет смысла. Лучше реализуем и проверим в действии другую, менее тривиальную.

Создание форматировщика даты. Если вы раньше занимались серверным программированием, то одна из вещей, по которой вы будете скучать, — это простой форматировщик даты, у объекта JavaScript `Date` такого нет. Поскольку такая функция оперировала бы не с элементами DOM, а с объектом `Date`, то она — идеальный кандидат на роль сервисной функции. Напишем такую функцию, использующую следующий синтаксис.

Синтаксис функции: $.formatDate

`$.formatDate(date, pattern)`

Форматирует переданную дату в соответствии с заданным шаблоном. В шаблоне применяются такие обозначения:

`yyyy`: четырехзначный год;

`yy`: двузначный год;

`MMMM`: полное название месяца;

`MMM`: сокращенное название месяца;

`MM`: двузначный номер месяца с лидирующими нулями;

`M`: номер месяца;

`dd`: двузначный день месяца с лидирующими нулями;

`d`: день месяца;

EEEE: полное название дня недели;
EEE: сокращенное название дня недели;
a: время относительно полудня (AM или PM);
HH: 24-часовое время суток в виде двузначного числа с лидирующими нулями;
H: 24-часовое время суток;
hh: 12-часовое время суток в виде двузначного числа с лидирующими нулями;
h: 12-часовое время суток;
mm: количество минут в виде двузначного числа с лидирующими нулями;
m: количество минут;
ss: количество секунд в виде двузначного числа с лидирующими нулями;
s: количество секунд;
S: количество миллисекунд в виде трехзначного числа с лидирующими нулями.

Параметры

`date`	(Дата) Дата, которую надо отформатировать.
`pattern`	(Строка) Шаблон форматирования даты. Символы, не подходящие к шаблону, копируются в результат без изменений.

Возвращает

Отформатированную дату.

Реализация этой функции представлена в листинге 12.6. Мы не будем чересчур углубляться в детали, касающиеся алгоритма форматирования, поскольку он не является темой данной главы. Воспользуемся реализацией, чтобы обратить ваше внимание на ряд интересных подходов, которые можно использовать при создании более сложных сервисных функций.

Листинг 12.6. Реализация сервисной функции $.formatDate()

```
(function($) {
   var patternParts =
/^(yy(yy)?|M(M(M(M)?)?)?|d(d)?|EEE(E)?|a|H(H)?|h(h)?|m(m)?|s(s)?|S)/;
   var patternValue = {
      yy: function(date) {
         return toFixedWidth(date.getFullYear(), 2);
      },
      yyyy: function(date) {
         return date.getFullYear().toString();
      },
      MMMM: function(date) {
         return $.formatDate.monthNames[date.getMonth()];
      },
      MMM: function(date) {
         return $.formatDate.monthNames[date.getMonth()].substr(0, 3);
      },
      MM: function(date) {
         return toFixedWidth(date.getMonth() + 1, 2);
```

❶ Определяется регулярное выражение, которому будут соответствовать элементы шаблонов

❷ Определяется объект, содержащий функцию форматирования, которая будет использоваться для найденных соответствий

```
    },
    M: function(date) {
       return date.getMonth() + 1;
    },
    dd: function(date) {
       return toFixedWidth(date.getDate(), 2);
    },
    d: function(date) {
       return date.getDate();
    },
    EEEE: function(date) {
       return $.formatDate.dayNames[date.getDay()];
    },
    EEE: function(date) {
       return $.formatDate.dayNames[date.getDay()].substr(0, 3);
    },
    HH: function(date) {
       return toFixedWidth(date.getHours(), 2);
    },
    H: function(date) {
       return date.getHours();
    },
    hh: function(date) {
       var hours = date.getHours();
       return toFixedWidth(hours > 12 ? hours - 12 : hours, 2);
    },
    h: function(date) {
       return date.getHours() % 12;
    },
    mm: function(date) {
       return toFixedWidth(date.getMinutes(), 2);
    },
    m: function(date) {
       return date.getMinutes();
    },
    ss: function(date) {
       return toFixedWidth(date.getSeconds(), 2);
    },
    s: function(date) {
       return date.getSeconds();
    },
    S: function(date) {
       return toFixedWidth(date.getMilliseconds(), 3);
    },
    a: function(date) {
       return date.getHours() < 12 ? 'AM' : 'PM';
    }
};
function toFixedWidth(value, length, fill) {
   var result = (value || '').toString();
```

❸ Функция форматирует переданное значение как поле фиксированной длины и заданного размера

```
      fill = fill || '0';                    ❹ Присваивается значение по умолчанию
      var padding = length - result.length;  ❺ Вычисляется отступ
      if (padding < 0) {
         result = result.substr(-padding);   ❻ При необходимости усекается
      } else {
         for (var n = 0; n < padding; n++) {
         result = fill + result;             ❼ Составляется результат
      }
   }
   return result;                            ❽ Возвращается готовый результат
  }
  $.formatDate = function(date, pattern) {   ❾ Реализуется основное тело функции
     var result = [];
     while (pattern.length > 0) {
        patternParts.lastIndex = 0;
        var matched = patternParts.exec(pattern);
        if (matched) {
           result.push(patternValue[matched[0]].call(this, date));
           pattern = pattern.slice(matched[0].length);
        } else {
           result.push(pattern.charAt(0));
           pattern = pattern.slice(1);
        }
     }
     return result.join('');
  };
  $.formatDate.monthNames = [                ❿ Названия месяцев
    'January', 'February', 'March', 'April', 'May', 'June', 'July',
    'August', 'September', 'October', 'November', 'December'
  ];
  $.formatDate.dayNames = [                  ⓫ Названия дней недели
    'Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday',
    'Saturday'
  ];
})(jQuery);
```

Самый интересный аспект этой реализации, не считая нескольких трюков JavaScript, использованных для сокращения кода, — то, что анонимной функции, назначенной `$.formatDate` ❾, требуется для работы несколько вспомогательных массивов, объектов и функций, а именно:

- регулярное выражение для поиска соответствий элементам шаблона ❶;
- список английских названий месяцев ❿;
- список английских названий дней недели ⓫;
- набор подфункций, извлекающих из предоставленной даты соответствия для каждого элемента шаблона ❷;
- частная функция, форматирующая предоставленное значение в виде поля фиксированной длины и заданного размера ❸.

В данной сервисной функции определены несколько переменных и функций. Некоторые из них являются приватными (объявлены с помощью ключевого слова `var`), в то время как другие доступны извне (объявлены как свойства `$.formatDate`). С одной стороны, переменная `patternParts` нужна только внутри нашей функции, так что нет смысла делать ее общедоступной, поэтому она объявлена как приватная. С другой стороны, переменные `monthNames` (10) и `dayNames` (11) могут быть переопределены, чтобы перевести названия месяцев и дней недели на другие языки, поэтому мы сделали их доступными за пределами функции. Помните: функции JavaScript — объекты первого класса и у них могут быть свои свойства. В том числе свойствами могут служить и другие объекты JavaScript.

Наша сервисная функция опирается на вспомогательную функцию `toFixedWidth()`. Передаваемое ей значение преобразуется в строковый эквивалент, а символы заполнения определяются либо передаваемым параметром, либо по умолчанию равны `0` (4). Затем вычисляется требуемая величина отступа (5).

Если отступ получится отрицательным (результат длиннее, чем переданная длина поля), то начало результата обрезается до достижения заданного размера (6); иначе в начало результата добавляется соответствующее количество символов заполнения (7). Затем он возвращается как результат функции (7).

А как же сам алгоритм форматирования? Если коротко, то он работает так.

1. Создается массив для хранения фрагментов результата.
2. Просматривается шаблон, переданный сервисной функции в качестве второго аргумента, в нем идентифицируются подходящие фрагменты и символы, не входящие в шаблон, до тех пор, пока не будет достигнут конец строки.
3. На каждой итерации просмотра регулярное выражение (хранящееся в переменной `patternParts`) восстанавливается путем присвоения его свойству `lastIndex` значения `0`.
4. Проверяется, есть ли в регулярном выражении фрагмент, находящийся в данный момент в начале шаблона.
5. Вызывается функция из коллекции функций преобразования `patternValue`, чтобы получить соответствующее значение из экземпляра `Date` в случае, если обнаружится такое соответствие. Это значение помещается в конец массива результатов, а обнаруженный фрагмент удаляется из начала шаблона.
6. Если фрагмент, находящийся в начале шаблона, не соответствует регулярному выражению, то первый символ шаблона удаляется и добавляется в конец массива результатов.
7. Когда весь шаблон пройден, массив результатов объединяется в строку, и ее значение возвращается как значение функции.

Код этого примера доступен в файле `js/jquery.jqia.formatDate.js`, а простая страница для его тестирования — в файле `chapter-12/jqia.formatDate.html`. Оба файла предоставлены с книгой.

12.6. Резюме

В данной главе мы научились писать многократно используемый код, дополняющий возможности нашей библиотеки. Создание плагинов jQuery дает целый ряд преимуществ. Это не только согласованность кода во всех веб-приложениях, независимо от того, относится ли он к jQuery API, или вы написали его сами. Это также возможность применять в вашем коде все преимущества jQuery.

Вы узнали, как соблюдение правил именования позволяет избежать конфликтов между именами файлов и кодом других плагинов, а также проблем, связанных с именем `$`, если на странице, использующей ваш плагин, оно будет переназначено другому объекту. Кроме того, вы научились создавать плагины, не нарушающие цепочки jQuery.

Вы узнали, что создавать сервисные функции — то же самое, что добавить новое свойство-функцию к объекту `$`, а новые методы jQuery создаются в виде свойств `$.fn`.

Если вам понравилось создавать плагины, то мы настоятельно рекомендуем загрузить и тщательно проанализировать код нескольких уже существующих, чтобы изучить, как реализованы их функции.

Теперь, когда знаний о jQuery стало еще больше, посмотрим, как можно применять библиотеку, чтобы улучшить управление асинхронными функциями.

13 Предотвращение появления callback hell с помощью Deferred

В этой главе:

- что такое промисы и почему они важны;
- объект `Deferred`;
- как управлять несколькими асинхронными операциями;
- разрешение и отклонение промиса.

Долгое время разработчики на JavaScript прибегали к функциям обратного вызова для выполнения нескольких задач, таких как работа операторов после определенного количества времени (применяя `setTimeout()`), или при регулярных интервалах (задействуя `setInterval()`), или для ответа на данное событие (`click`, `keypress`, `load` и т. д.). Мы широко обсуждали и использовали функции обратного вызова для совершения асинхронных операций; например, в главе 6 сосредоточились на событиях, в главе 8 описали анимационные эффекты, в главе 10 рассмотрели Ajax. Функции обратного вызова просты и делают свою работу, но как только нужно запустить много асинхронных операций — либо параллельно, либо последовательно, — они становятся неуправляемыми. Ситуация, когда у вас есть множества вложенных функций обратного вызова или индивидуальные функции обратного вызова, которые нужно синхронизировать, часто называется callback hell («ад обратного вызова»).

Сейчас сайты и веб-приложения зачастую используют больше кода JavaScript, чем кода серверной стороны (пришла эра сервисов, управляемых API, верно?). По этой причине разработчикам нужен лучший способ управления асинхронными операциями и их синхронизации, чтобы сохранить читаемость и поддерживаемость своего кода.

В данной главе мы обсудим промисы: что это такое и как jQuery позволяет их использовать. Наша любимая библиотека реализует концепцию промисов с помощью двух объектов: `Deferred` и `Promise`. Как jQuery реализует промисы — тема для дискуссий, критики и множества изменений, что станет известно из следующих страниц.

По мере изучения данной главы вы заметите, что терминология может немного сбивать с толку, и что у понятия промиса нет взаимно однозначного со-

ответствия объекту `Promise`, и это, как все признают, немного странно. Кроме того, нужно изучить немного терминов. Но не пугайтесь. Текст будет дополнен несколькими примерами и подробными объяснениями, которые помогут в процессе изучения.

13.1. Введение в промисы

В реальном мире, вне компьютеров (да, там еще что-то есть) мы часто говорим об обещаниях. Вы просите людей пообещать нечто, или об этом просят вас. Интуитивно понятно, что обещание представляет собой обязательство, проистекающее из просьбы, адресованной вами человеку, которое он должен выполнить в какой-то момент в будущем. Иногда действие может быть произведено очень скоро; иногда нужно ждать долгое время. В идеальном мире все обещания выполняются. К сожалению, наш мир далек от идеала, поэтому случается, что обещания остаются так и не выполненными по какой-либо причине. Вне зависимости от времени, которое оно займет, и конечного результата, обещание не препятствует какой-либо иной деятельности в это же время — чтению, приготовлению пищи или работе.

В JavaScript *promise* (промис, обещание) именно это и делает. Вы запрашиваете ресурс с другого сайта, результат длительного расчета либо вашего сервера, либо функции JavaScript, либо ответа сервиса REST (это примеры промисов), а выполняете другие задачи, пока ждете результата. Когда последний становится доступным (промис *разрешен/выполнен*) или запрос завершился неудачно (промис *завершился неудачно/отклонен*), действуете соответственно.

Предыдущее описание, надеемся, поможет понять, что такое промисы. Но у них есть и более формальное определение. Промисы широко обсуждались, и в результате дискуссий были разработаны два предложения: Promises/A и Promises/A+. Мы рассмотрим оба варианта до углубления в то, как jQuery работает с промисами, чтобы вы лучше понимали их как понятие.

Спецификация Promises/A+ находится на http://promisesaplus.com/, но в двух словах они определяют промис так.

> Промис представляет собой конечный результат асинхронной операции. Основной способ взаимодействия с промисом — через метод then, регистрирующий функции обратного вызова, получающие либо конечное значение промиса, либо причину, по которой промис не может быть выполнен.

Спецификации Promises/A+

Метод `then()`, описанный в предложении Promises/A, — основа промисов. Он принимает две функции: одну, если промис выполнен, и вторую — если отклонен. Когда промис находится в одном из этих состояний, независимо от того, в каком из них, он считается *урегулированным*. Если промис не выполнен, не отклонен (например, если вы все еще ждете ответа от сервера, производящего расчеты), то он *в ожидании*.

Даже если формальное определение упоминает асинхронные операции, то стоит заметить, что промисы могут разрешаться или отклоняться и синхронными операциями, поэтому можно использовать их не только для запросов Ajax (обсудим это подробнее позже).

Иногда операция, синхронная или асинхронная, может быть уже завершена, так что возвращаемое значение или причина отклонения промиса могут быть уже доступны. В подобном случае, если вы регистрируете функцию с помощью метода `then()`, она выполнится немедленно. Это еще одно важное отличие работы промиса от работы функции обратного вызова при ответе на событие. Помните: если вы добавляете обработчик к событию, которое уже запущено, то обратный вызов не будет выполняться.

Теперь, когда вы прочитали о промисах и методе `then()`, важно понять, чем промисы так хороши и как они могут помочь в работе. В качестве доказательства обсудим и реализуем простой сценарий, аналогичный многим, с которыми вы сталкивались в вашей работе. Сначала мы подойдем к задаче, используя уже обретенные знания, такие как функции обратного вызова, а затем переберем код, чтобы найти лучшее решение с помощью промисов — в отношении как читаемости, так и поддерживаемости кода.

Предположим, вы создаете веб-страницу, используя сторонний веб-сервис Randomizer (генератор случайных чисел). Сервис предоставляет API, который возвращает случайный номер при каждом вызове. Вы хотите, чтобы ваша веб-страница получала два числа и суммировала их. После завершения хотите показать результат в элементе, у которого `content` в виде ID. Для достижения этой цели нужно выполнить два запроса Ajax, синхронизировать функции обратного вызова каким-то образом (вы задействуете функцию поддержки) и затем показать результат. Наиболее сложная часть кода — вторая: синхронизация двух независимых запросов Ajax.

Код веб-страницы, который мы обсуждали, показан в листинге 13.1. Пожалуйста, обратите внимание: запуск этой страницы не даст никакого результата, так как вы осуществляете запрос к несуществующему сервису Randomizer, но написанный код позволит понять важность промисов.

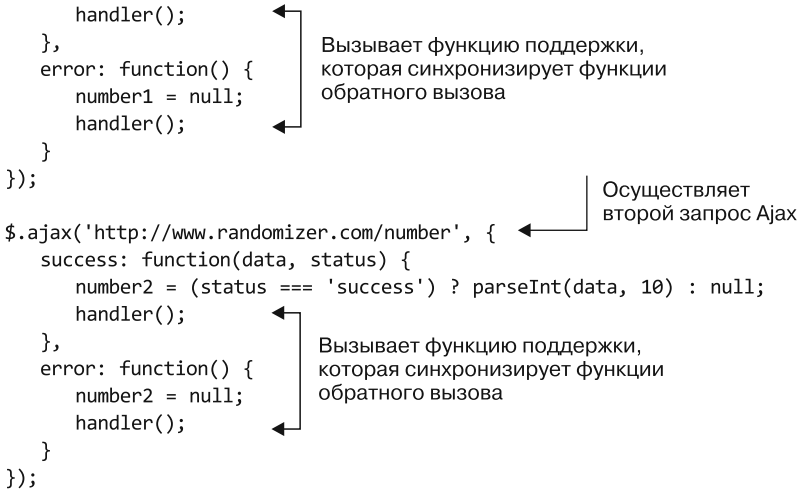
Листинг 13.1. Реализация нескольких асинхронных операций с помощью функций обратного вызова

```
var number1, number2;

function handler() {
   var $content = $('#content');

   if (number1 === null || number2 === null) {
      $content.text('An error has occurred, try again later.');
   } else if (number1 !== undefined && number2 !== undefined) {
      $content.text(number1 + number2);
   }
}
$.ajax(http://www.randomizer.com/number', {
   success: function(data, status) {
      number1 = (status === 'success') ? parseInt(data, 10) : null;
```

Определяет функцию, которую будут вызывать обе функции обратного вызова Ajax

Выводит результат на странице

Осуществляет первый запрос Ajax



Данный листинг совсем простой, но имеет одну большую проблему: нужно предоставить переменную для каждой функции обратного вызова. Здесь нужны только две переменные, но проект растет, ситуация может стать неуправляемой. К тому же представим, что предыдущий код был телом функции, вызываемой другой функцией, названной `foo`. Как можно вернуть результат двух запросов Ajax в `foo`? Используя текущий подход, это нельзя сделать без глобальных переменных. Наконец, что если нужно было сделать два запроса Ajax и второй должен был отправиться после завершения первого? В таком случае вам нужен обратный вызов внутри обратного вызова. Со все нарастающим количеством обратных вызовов в вашем коде наступит полный хаос, вот почему эту ситуацию называют «адом обратного вызова».

Данный пример дает представление о том, какие проблемы могут возникнуть при работе с функциями обратного вызова. Можно улучшить код с помощью промисов, и это даст ряд преимуществ. ECMA International (http://www.ecma-international.org/), группа стандартизаторов JavaScript, решила представить промисы в следующей версии JavaScript (ECMAScript 2015, также известной как ECMAScript 6), и придерживаться предложения Promises/A+. Некоторые современные браузеры уже поддерживают их, но не все, а старые браузеры и не будут. Так что, если вы хотите писать чистый, читаемый и поддерживаемый код, используя промисы, то придется рассчитывать на полифилл или библиотеку при желании задействовать больше функций, чем предлагается по стандарту.

jQuery поможет избежать указанных проблем с браузерами, но, в зависимости от версии библиотеки, это может происходить по-разному. jQuery предоставляет два объекта, `Deferred` и `Promise`, которые можно применять. Но прежде, чем мы их опишем, нужно помочь вам понять концепции данной главы.

Реализация jQuery's 1.x и 2.x следует предложению CommonJS Promises/A (http://wiki.commonjs.org/wiki/Promises/A), который используется как основа для Promises/A+. Таким образом, есть разница между тем, как можно задействовать

промисы в чистом JavaScript и в jQuery 1.x и 2.x. Более того, поскольку библиотека следует разным предложениям в этих ветках, она несовместима с другими. Предложение Promises/A дает промисам такое определение:

Промис представляет конечное значение, возвращаемое после одиночного выполнения операции. Промис может быть в одном из трех состояний: не выполнен, выполнен, завершился неудачно. Состояние промиса может меняться только от невыполненного к выполненному или от невыполненного до завершенного неудачно.

Спецификация Promises/A

Как видите, терминология для определения объектов `Promise`, показанная на рис. 13.1, немного отличается. Предложение Promises/A определяет состояния «не выполнен», «выполнен» и «завершен неудачно», а Promises/A+ — состояния «в ожидании», «выполнен» и «отклонен».

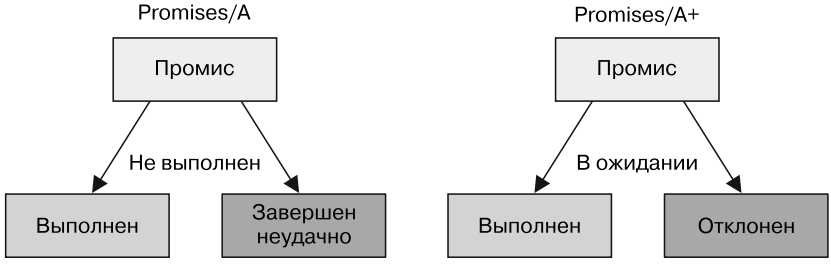


Рис. 13.1. Различия в терминологии между предложениями Promises/A и Promises/A+

Эти предложения обрисовали поведение промисов, а не реализацию, так что у библиотек, реализующих промисы, есть общий метод `then()`, но они могут отличаться другими предлагаемыми методами.

В jQuery 3.x была улучшена функциональная совместимость со стандартным промисом (как реализованном в ECMAScript 2015, следующим Promises/A+). Сигнатура метода `then()` по-прежнему немного отличается для обратной совместимости, но поведение в большей степени соответствует стандарту. Если вас это смущает — не беспокойтесь: мы осветим все различия и дадим множество примеров.

В следующем разделе вы узнаете больше об объектах `Deferred` и `Promise`. Там, где это необходимо, мы расскажем о различиях между предложениями Promises/A и Promises/A+.

13.2. Объекты Deferred и Promise

Объект `Deferred` был представлен в jQuery 1.5 как встраиваемая в цепочку вспомогательная функция, используемая для регистрации нескольких функций обратного вызова в очереди обратных вызовов, вызывая очереди обратных вызовов и передавая статусы успешного или неудачного выполнения любых синхронных или асинхронных функций. За это время он успел побывать темой множества дискуссий,

определенной критики и многократно менялся[1]. Данный объект можно применять для многих асинхронных операций, например для запросов Ajax и анимационных эффектов, но также и для функций распределения по времени в JavaScript. С объектом `Promise` он представляет собой реализацию промисов jQuery.

Объект `Promise` создается, начиная от объекта `Deferred` или объекта jQuery, и у него есть поднабор методов объекта `Deferred` (`always()`, `done()`, `fail()`, `state()` и `then()`). Объекты `Deferred` обычно используются при написании собственной функции, работающей с асинхронными функциями обратного вызова. Таким образом, ваша функция — *поставщик* значения и вы хотите запретить пользователям менять состояние `Deferred`.

В главе 10 мы рассмотрели вспомогательные функции, которые предоставляет jQuery для работы с запросами Ajax, но не упомянули (чтобы облегчить восприятие материала) вот о чем: возвращаемое значение этих функций, являющихся объектом `jqHXR` (если помните), реализует интерфейс `Promise`. По данной причине они иногда называются `Promise`-совместимыми объектами и можно вызывать все методы обычных объектов `Promise` на них. Это позволит писать более чистый и читаемый код.

А сейчас обсудим, как работать с `Deferred` и `Promise`.

13.3. Методы Deferred

В jQuery объект `Deferred` создается путем вызова конструктора `$.Deferred()`. Синтаксис этой функции следующий.

Синтаксис функции: $.Deferred

`$.Deferred([beforeStart])`

Функция-конструктор, возвращающая объект-утилиту, который можно использовать в цепочке вызовов, содержащая методы для регистрации функций обратного вызова, помещения их в очередь, инициализации выполнения очередей обратных вызовов и передачи статуса успешного или неудачного выполнения любой синхронной или асинхронной функции. Она принимает дополнительную функцию для вызова перед возвратом конструктора.

Параметры

`beforeStart` (Функция) Функция, которая вызывается перед возвратом конструктора. Принимает объект `Deferred`, используемый в виде контекста (`this`) функции.

Возвращает

Объект `Deferred`.

---

[1] http://bugs.jquery.com/ticket/11010.

You're Missing the Point of Promises by Domenic Denicola: http://domenic.me/2012/10/14/youremissing-the-point-of-promises/.

JavaScript promises and why jQuery implementation is broken by Valerio Gheri: https://thewayofcode.wordpress.com/tag/jquery-deferred-broken/.

https://github.com/jquery/jquery/issues/1722.

Объект Deferred позволяет выполнять работу в цепочке, как и многие другие методы jQuery, которые мы рассматривали, но имеет свои собственные методы. Вы никогда не сможете написать такую инструкцию:

```
$.Deferred().html('Промисы — это круто!');
```

Создание нового объекта Deferred не будет иметь какой-либо пользы, если вы не знаете, как его использовать. Начиная со следующего подраздела, мы рассмотрим методы, предоставляемые этим объектом.

13.3.1. Выполнение или отклонение Deferred

Одно из первых представленных в данной главе понятий — состояние, в котором могут находиться промисы. В jQuery промис может быть выполнен (если выполнен успешно), отклонен (если возникла ошибка) или в ожидании (не выполнен и не отклонен). Эти состояния могут быть достигнуты двумя способами. Первый определен кодом, который написали вы или другой разработчик, с явным вызовом таких методов, как deferred.resolve(), deferred.resolveWith(), deferred.reject() или deferred.rejectWith(). Указанные методы, как мы скоро обсудим, позволяют либо выполнить, либо отклонить промис. Второй определен успешным или неудачным выполнением функции jQuery (например, $.ajax()) — в этом случае уже нет надобности вызывать самостоятельно какой-либо из упоминавшихся выше методов.

Прежде чем писать код, использующий объект Deferred, нужно познакомиться с этими методами. Синтаксис метода deferred.resolve() выглядит следующим образом.

Синтаксис метода: deferred.resolve

deferred.resolve([argument, ..., argument])

Разрешает Deferred выполнение всех определенных успешных функций обратного вызова, передавая любой данный аргумент. Метод принимает произвольное количество аргументов.

Параметры

argument	(Любой) Необязательный аргумент любого типа, передающийся функциям обратного вызова, которые определены и выполняются в случае успеха.

Возвращает

Объект Deferred.

Синтаксис метода deferred.resolveWith() таков.

Синтаксис метода: deferred.resolveWith

deferred.resolveWith(context[, argument, ..., argument])

Разрешает объекту Deferred выполнение любого установленного успешного обратного вызова, передает любой argument и устанавливает context как их контекст.

Параметры

`context` (Объект) Объект, который устанавливается в качестве контекста успешных обратных вызовов.

`argument` (Любой) Необязательный аргумент любого типа, передающийся функциям обратного вызова, которые определены и выполняются в случае успеха.

Возвращает

Объект `Deferred`.

Иногда необходимо отклонить промис. Для таких случаев можно воспользоваться методом `deferred.reject()`. Его синтаксис выглядит следующим образом.

Синтаксис метода: deferred.reject

```
deferred.reject([argument, ..., argument])
```

Запрещает объекту `Deferred` выполнение любого установленного неудачного обратного вызова, передает любой `argument`. Метод принимает произвольное количество аргументов.

Параметры

`argument` (Любой) Необязательный аргумент любого типа, передающийся функциям обратного вызова, которые определены и выполняются в случае неудачи.

Возвращает

Объект `Deferred`.

Аналогично тому, как jQuery определяет метод для установки контекста для успешных обратных вызовов, вы можете установить его для неудачных обратных вызовов с помощью метода `deferred.rejectWith()`. Его синтаксис показан ниже.

Синтаксис метода: deferred.rejectWith

```
deferred.rejectWith(context[, argument, ..., argument])
```

Запрещает объекту `Deferred` выполнение любого установленного неудачного обратного вызова, передает любой `argument` и устанавливает `context` как их контекст.

Параметры

`context` (Объект) Объект, который устанавливается в качестве контекста неудачных обратных вызовов.

`argument` (Любой) Необязательный аргумент любого типа, передающийся функциям обратного вызова, которые определены и выполняются в случае неудачи.

Возвращает

Объект `Deferred`.

Теперь вы знаете, как создать `Deferred` и выполнить его или отклонить. Но самое интересное происходит при написании кода, реагирующего на изменение состояния `Deferred`. Посмотрим, как это делается.

13.3.2. Выполнение функций после разрешения или отклонения

Обычно вам нужно знать, когда выполнен `Deferred`, чтобы совершить определенные действия. Для этого можно использовать метод `deferred.done()`. Он принимает один или несколько аргументов, каждый из которых может быть либо отдельной функцией, либо массивом функций. Эти функции обратного вызова выполняются в порядке их добавления. Синтаксис метода показан ниже.

Синтаксис метода: deferred.done

```
deferred.done(callbacks[, callbacks, ..., callbacks])
```

Добавляет обработчики, которые вызываются после выполнения объекта `Deferred`. Метод принимает произвольное количество функций обратного вызова, начиная с одной.

Параметры

`callbacks` (Функция|Массив) Функция или массив функций, которые вызываются, если `Deferred` разрешен.

Возвращает

Объект `Deferred`.

Аналогично тому, как вы можете выполнить операции после того, как объект `Deferred` разрешен, можно запустить функции в случае его отклонения. Для этого используется метод `deferred.fail()`. Его синтаксис показан ниже.

Синтаксис метода: deferred.fail

```
deferred.fail(callbacks[, callbacks, ..., callbacks])
```

Добавляет обработчики, которые вызываются после отклонения объекта `Deferred`. Метод принимает произвольное количество функций обратного вызова, начиная с одной.

Параметры

`callbacks` (Функция|Массив) Функция или массив функций, которые вызываются, если `Deferred` отклонен.

Возвращает

Объект `Deferred`.

В данном случае, когда `Deferred` отклонен, функции обратного вызова также выполняются в порядке их добавления.

Теперь, когда мы представили эти методы, можно показать код, демонстрирующий, как все происходит. Для начала модифицируем пример, обсуждавшийся в начале данной главы, и перепишем его, используя `Deferred`. На сей раз мы хотим предоставить рабочий демонстрационный пример, так что добавим простую PHP-страничку, названную `integer.php`, моделирующую работу сервиса Randomizer. Итоговый код показан в листинге 13.2, а также доступен в файле `chapter-13/randomizer.1.html`, предоставленном с книгой.

Листинг 13.2. Использование Promise с запросами Ajax, версия 1

```
<!DOCTYPE html>
<html>
   <head>
      <meta charset="utf-8" />
      <title>Использование промисов с запросами Ajax,
             версия 1</title>
      <link rel="stylesheet" href="../css/main.css" />
   </head>
   <body>
      <p>
         Случайно созданные числа - <span id="number1"></span>
         и<span id="number2"></span>.
         Их сумма равна<span id="sum"></span>
      <p>

      <script src="../js/jquery-1.11.3.min.js"></script>
      <script>
         var number1, number2;

         function handler() {
            var $sum = $('#sum');

            if (number1 === null || number2 === null) {
               $sum.text('Произошла ошибка. Повторите позже.');
            } else if (number1 !== undefined &&
                       number2 !== undefined) {
               $sum.text(number1 + number2);
            }
         }

         var promise1 = $.ajax('integer.php');
         promise1.done(function(data, status, jqXHR) {
            number1 = (status === 'success') ?
                       parseInt(data, 10) : null;
            $('#number1').text(number1);
            handler();
         })

         .fail(function() {
            number1 = null;
            handler();
         });

         var promise2 = $.ajax('integer.php');
         promise2.done(function(data, status, jqXHR) {
            number2 = (status === 'success') ?
                       parseInt(data, 10) : null;
            $('#number2').text(number2);
            handler();
```

1. Сохраняет совместимый с Promise объект, возвращенный $.ajax() в переменной

Добавляет функцию обратного вызова, которая выполнится после выполнения первого промиса (2)

3. Добавляет функцию обратного вызова, которая выполнится, если первый промис будет отклонен

4. Сохраняет другой совместимый с Promise объект, возвращенный $.ajax() в переменной

Добавляет функцию обратного вызова, которая выполнится после выполнения второго промиса (5)

```
            })
            .fail(function() {
               number2 = null;
               handler();
            });
         </script>
      </body>
   </html>
```

Добавляет функцию обратного вызова, которая выполнится, если второй промис будет отклонен ❻

Код в этом листинге назван версией 1, поскольку вы будете работать с ним позже и перерабатывать, пока не получите чистый и элегантный код, решающий поставленную задачу благодаря использованию промисов. Как видите, код уже отличается от представленного в листинге 13.1.

Первое понятие — вы можете хранить объект `jqXHR`, возвращенный функцией `$.ajax()`, в двух переменных, названных `promise1` ❶ и `promise2` ❹. Как мы уже упоминали, объект `jqXHR` совместим с `Promise`. Это значит, можно вызывать на нем такие методы, как `done()` и `fail()`, напрямую, но мы хотим прояснить, какой объект возвращается `$.ajax()` через имя переменных.

Затем вы прикрепляете те же функции обратного вызова успешного и неудачного завершений, которые были разработаны в листинге 13.1 (см. выше). Функции обратного вызова, вызываемые при успешном выполнении, добавлены с помощью вызова метода `done()` на переменных `promise1` ❷ и `promise2` ❺. То же самое происходит с функциями обратного вызова при неудачном завершении, они добавляются путем вызова метода `fail()` на `promise1` ❸ и `promise2` ❻.

При описании `done()` и `fail()` мы хотели подчеркнуть, что при добавлении функции обратного вызова успеха или неудачи *после* изменения состояния `Deferred` на «выполнено» или «отклонено» функция обратного вызова выполняется сразу. Рассмотрим следующий код:

```
var promise1 = $.ajax('integer.php');
setTimeout(function() {
  promise1.done(function(data, status, jqXHR) {
    // Код здесь
  })
  .fail(function() {
    // Код здесь
  });
}, 5000);
```

В данном случае вы задерживаете операцию, которая добавляет функции обратного вызова, на пять секунд, чтобы имитировать достаточно длительное время для выполнения страницы `integer.php` и возврата ответа (это не гарантируется, но для нашего примера достаточно). Основываясь на приведенном допущении, в то время, когда `done()` и `fail()` вызываются для добавления функций обратного вызова, состояние `promise1` уже определено. Вы вполне могли ожидать, в силу опыта работы со слушателями, добавленными к событиям, что ни один из них не будет запущен. Причина в следующем: «событие» изменения состояния промиса уже произошло. Но одна из двух функций все еще работает, в этом важное отличие.

Другое интересное отличие состоит в том, что можно добавить сколько угодно функций обратного вызова с помощью одной инструкции. Допустим, запрос Ajax прошел успешно, вы хотите выполнить две функции, `foo()` и `bar()`. При традиционном подходе можно было бы написать такой код:

```
$.ajax('integer.php', {
  success: function(data, status, jqXHR) {
    foo(data, status, jqXHR);
    bar(data, status, jqXHR);
  }
);
```

Используя метод `done()`, можно переписать это, задействуя всего одну строку:

```
$.ajax('integer.php').done(foo, bar);
```

Или то же самое (обратите внимание, здесь вы передаете массив функций):

```
$.ajax('integer.php').done([foo, bar]);
```

Намного лучше, правда?

Листинг 13.2 использует несколько новых методов, которые вы изучили, но по-прежнему неполноценен из-за применения неудобного подхода к синхронизации. Посмотрим, как можно улучшить ситуацию.

13.3.3. Метод when()

Чтобы отредактировать листинг 13.2 и довести до финальной версии, можно воспользоваться другой вспомогательной функцией — `$.when()`. Она позволяет просто выполнять функции обратного вызова, основанные на одном или более объектах, обычно совместимых с `Deferred` или `Promise`, представляющих асинхронные события. Это именно то, что нужно коду, поскольку у вас есть два объекта, совместимых с `Promise`, возвращаемые двумя вызовами `$.ajax()`. Но прежде, чем ее использовать, посмотрим, какие параметры она принимает.

Синтаксис функции: $.when

`$.when(object[, object, ..., object])`

Предоставляет способ запустить функции обратного вызова, основанные на одном или нескольких объектах, обычно совместимых с `Deferred` или `Promise`, представляющих асинхронные события. Эта функция принимает произвольное количество объектов, начиная от одного.

Параметры

`object` (Deferred|Promise|Объект) Объект `Deferred`, `Promise`, или совместимый с `Promise`, или объект JavaScript. Если передается объект JavaScript, то он обрабатывается как разрешенный `Deferred`.

Возвращает

Объект `Promise`.

У вспомогательной функции `$.when()` есть интересная особенность: она возвращает объект не `Deferred`, а `Promise`. То, что возвращает этот метод, нельзя выполнить или отклонить; вы можете лишь вызвать `done()`, `fail()` и еще некоторые другие методы. Стоит отметить: в ECMAScript 2015 есть аналогичный метод, названный `Promise.all()`.

Если вы передадите отдельный объект `Deferred` в `$.when()`, то вернется его объект `Promise`; в остальных случаях будет создан новый `Promise`, начиная от «основного» `Deferred`, который будет следить за состоянием всех объектов (`Promise`, совместимый с `Promise`, `Deferred` и т. д.), переданных `$.when()`.

`$.when()` вызывает выполнение всех успешных функций обратного вызова (то есть функций, которые выполняются в случае события успешного выполнения промиса), когда и если все объекты, переданные этой вспомогательной функции, выполнены (в случае с объектами `Deferred`, `Promise` и совместимым с `Promise`), или могут быть рассмотрены как выполненные (в случае с другими типами объектов). И наоборот, вызывается выполнение неудачных функций обратного вызова, когда один из `Deferred` отклонен или один из объектов `Promise` или совместимых с `Promise` в состоянии «отклонен». Аргументы, переданные функциям обратного вызова, успешным или неудачным, передаются `resolve()` или `reject()`, в зависимости от конкретного случая.

Прежде чем погрязнуть в этом море информации, рассмотрим конкретный пример. Как уже упоминалось, мы перепишем код листинга 13.2 и попробуем улучшить его, используя `Deferred`. Конечный результат показан в листинге 13.3, а также доступен в файле `chapter-13/randomizer.2.html`.

Листинг 13.3. Использование Promise с запросами Ajax, версия 2

```
<!DOCTYPE html>
<html>
   <head>
      <meta charset="utf-8" />
      <title>Использование промисов с запросами Ajax,
             версия 2</title>
      <link rel="stylesheet" href="../css/main.css" />
   </head>
   <body>
   <p>
      Случайно созданные числа — <spanid="number1"></span>
      и<span id="number2"></span>.
      Их сумма равна <span id="sum"></span>
   <p>

   <script src="../js/jquery-1.11.3.min.js"></script>
   <script>
      function success(params1, params2) {
         var number1 = (params1[1] === 'success') ?
               parseInt(params1[0], 10) : null;

         var number2 = (params2[1] === 'success') ?
```

❶ Определяет функцию обратного вызова для случая успеха

```
            parseInt(params2[0], 10) : null;

        if (number1 === null || number2 === null) {
           fail();
           return;
        }

        $('#number1').text(number1);
        $('#number2').text(number2);
        $('#sum').text(number1 + number2);
      }
      function fail() {
        $('#sum').text('Произошла ошибка. Повторите позже.');
      }

      $.when($.ajax('integer.php'), $.ajax('integer.php'))
        .done(success)
        .fail(fail);
        </script>
    </body>
</html>
```

❷ Определяет функцию обратного вызова для случая неудачи

❹ Устанавливает функцию обратного вызова success(), используя метод deferred.done()

❺ Устанавливает функцию обратного вызова fail(), используя метод deferred.fail()

❸ Создает новый промис, начиная от «основного» Deferred, созданного внутренне $.when(), который основывается на вызовах $.ajax()

Посмотрев на код, вы почувствуете себя счастливым. Благодаря вспомогательной функции `$.when()` мы в высшей степени упростили код листинга 13.2, сделали его читаемым и управляемым. Проанализируем его.

Ключевой момент — использование `$.when()`, которую вы задействовали для решения проблемы синхронизации результатов запросов Ajax ❸. Делая это, вам не нужно устанавливать функции для успешного или неудачного завершения для каждого из них. Фактически вы только устанавливаете их на объекте `Promise`, возвращенном `$.when()` путем использования `done()` ❹ и `fail()` ❺. Еще раз: объект `Promise` создан, начиная от объекта `Deferred` или объекта jQuery (в данном случае он создается внутренне методом `$.when()`), и обладает поднабором методов `Deferred` (`always()`, `done()`, `fail()`, `state()` и `then()`).

Функция `success()` выполняется, когда оба совместимых с `Promise` объекта выполнены. Это поведение не поменялось по сравнению с предыдущей версией, но есть интересная деталь. Вы определяете два параметра для данной функции, `params1` и `params2`, потому что это количество совместимых с `Promise` объектов, которые вы используете ❶. Каждый из этих параметров является массивом, содержащим обычные параметры, переданные обратным вызовам для успешного обратного вызова функции `$.ajax()`, `$.get()` или `$.post()`: `data`, `statusText` и `jqXHR`. Стоит заметить: если передаваемое значение было бы простым объектом, то оно не было бы обернуто.

Последняя функция, определенная в листинге, — `fail()` ❷. Она взята из функции `handler()` из предыдущего листинга и выполняется после того, как запрос Ajax завершается неудачно.

Помимо выполнения или отклонения `Deferred`, можно также получить и уведомление о прогрессе процесса.

13.3.4. Уведомление о прогрессе Deferred

Иногда у вас может быть код, которому нужно знать о состоянии `Deferred`. Например, при асинхронном получении данных вы можете захотеть знать, сколько процентов задачи выполнено. Если процесс основывается на промисах, то у вас может быть функция, ожидающая выполнения или отклонения этого промиса, с помощью которой вы хотите информировать о текущем состоянии. Для таких случаев можно задействовать `deferred.notify()`. Синтаксис данного метода следующий.

Синтаксис метода: deferred.notify

`deferred.notify([argument, ..., argument])`

Запускает выполнение любого прогресса определенного обратного вызова, передавая любые заданные аргументы. Этот метод принимает произвольное число аргументов.

Параметры

`argument` (Любой) Необязательный аргумент любого типа, который будет передан в определенные функции обратного вызова, созданные для обработки текущего выполнения хода запроса.

Возвращает

Объект `Deferred`.

Если вы хотите заставить выполниться контекст функции обратного вызова, то можете использовать `deferred.notifyWith()`.

Синтаксис метода: deferred.notifyWith

`deferred.notifyWith (context[, argument, ..., argument])`

Запускает выполнение любого прогресса определенного обратного вызова, передавая любые заданные аргументы и устанавливая `context` в качестве их контекста.

Параметры

`context` (Объект) Объект, устанавливаемый как контекст прогресса функций обратного вызова.

`argument` (Любой) Необязательный аргумент любого типа, который будет передан в определенные функции обратного вызова, созданные для обработки текущего выполнения хода запроса.

Возвращает

Объект `Deferred`.

Задействуя эти методы, можно посмотреть, как выполнить некоторые действия в процессе работы функции.

13.3.5. Следим за прогрессом

Используя методы, описанные в предыдущем разделе, можно известить о прогрессе асинхронной операции. Но данное действие совершенно бесполезно, если вы не можете мониторить эти обновления. Познакомьтесь с методом `deferred.progress()`.

Синтаксис метода: deferred.progress

`deferred.progress(callbacks[, callbacks, ..., callbacks])`

Добавляет обработчики, которые вызываются, когда объект `Deferred` создает уведомления о прогрессе. Этот метод принимает произвольное количество функций обратного вызова, начиная от одной.

Параметры

`callbacks` (Функция|Массив) Функция или массив функций, вызываемый, когда объект `Deferred` создает уведомления о прогрессе.

Возвращает

Объект `Deferred`.

Теперь, когда известно, как работает метод, перейдем к примеру. Допустим, вы хотите создать анимацию для индикатора выполнения и иметь возможность следить за прогрессом анимации для отображения процента выполнения. Для этого можно использовать методы `deferred.progress()` и `deferred.notify()`.

ПРИМЕЧАНИЕ

Пример, который мы собираемся разработать, неидеален. Лучшим подходом было бы вернуть объект Promise используемого Deferred и позволить методу, вызывающему функцию анимации, обновлять процент. Мы изменим это ниже, но в данный момент хотим использовать максимально понятное и простое решение и двигаться дальше небольшими шажками, так что потерпите немного.

Код, который реализует эти требования, опубликован в листинге 13.4, а также доступен в файле `chapter-13/deferred.progress.1.html`, предоставленном с книгой, и в JS Bin (http://jsbin.com/yohiho/edit?html,css,js,output). .

Листинг 13.4. Использование deferred.progress(), версия 1

```
<!DOCTYPE html>
<html>
   <head>
      <meta charset="utf-8" />
      <title>Использование deferred.progress(), версия 1</title>
      <link rel="stylesheet" href="../css/main.css" />
      <style>
         .bar,
         .inner-bar
         {
            height: 50px;
```

```
            border-radius: 3px;
         }
         .bar
         {
            position: relative;
            border: 1px solid #000000;
            background-color: #CCCCCC;
         }
         .inner-bar
         {
            width: 0%;
            background-color: #F72F39;
         }
         .progress
         {
            position: absolute;
            font-size: 30px;
            width: 100%;
            text-align: center;
            top: 10px;
         }

         button
         {
            margin-top: 15px;
         }
      </style>
   </head>
   <body>
      <div class="bar">
         <div class="inner-bar"></div>
         <span class="progress">0%</span>
      </div>

      <button id="run-button">Запуск анимации</button>

      <script src="../js/jquery-1.11.3.min.js"></script>
      <script>
      function animate(milliseconds) {
         var $innerBar = $('.inner-bar').width(0);
         var $barWidth = $('.bar').width();
         var step = $barWidth / 100;
         var deferred = $.Deferred().progress(function (value) {
            $('.progress').text(Math.floor(value) + '%');
         });
         var idInterval = setInterval(
            function () {
               // Не можем использовать width() для получения
               // ширины из-за ошибки #1724
               // (https://github.com/jquery/jquery/issues/1724)
               var nextWidth =
                  parseFloat($innerBar.get(0).style.width) + step;
```

❶ Определяет простой индикатор выполнения

❷ Определяет функцию, которая анимирует индикатор выполнения

❸ Создает новый Deferred, добавляет функцию обратного вызова прогресса

```
            deferred.notify(nextWidth * 100 / $barWidth);
            $innerBar.width(nextWidth);
            if (nextWidth >= $barWidth) {
               deferred.resolve();
               clearInterval(idInterval);
            }
         },
            milliseconds / 100
         );
      }

      $('#run-button').click(function() {
         animate(1000);
      });
   </script>
</body>
</html>
```

Выполняет Deferred после завершения анимации ❺

Информирует Deferred на каждом шагу анимации, передавая процент выполнения ❹

В этом примере вы создаете индикатор, который показывает процент выполнения ❶. Внутри элемента `script` определяете функцию, названную `animate()`, анимирующую данный индикатор. Она принимает количество миллисекунд, которые должна длиться анимация. Внутри нее создаете новый экземпляр `Deferred` и добавляете функцию обратного вызова прогресса, обновляющую процент выполнения ❸.

Используя функцию JavaScript `setInterval()`, вы устанавливаете основу функции `animate()`, когда подсчитываете следующий шаг анимации, который всегда будет должен добавить еще один `$barWidth/100` к ширине индикатора и уведомить `Deferred`. Наконец, когда анимация будет завершена, выполните `Deferred` ❹, применяя метод `deferred.resolve()` ❺.

Мы показали пример использования метода `deferred.progress()`, но вы можете пытаться понять, почему бы не поместить инструкцию для обновления процента выполнения внутрь функции, переданной `setInterval()`, и избавиться от объекта `Deferred` совсем. Вы действительно могли бы сделать это, но данный листинг дал нам возможность постепенно привести вас к ключевой концепции: когда нужно применять объект `Deferred` или `Promise` и почему.

13.3.6. Использование объекта Promise

Для овладения в полной мере `Deferred` и `Promise` нужно понимать, когда использовать первый, а когда — второй. Чтобы помочь понять эту тему, предположим, что вы хотите реализовать основанную на промисах функцию тайм-аута. Вы — *производитель* функции. В этом случае *потребитель* функции не должен заботиться о том, выполнена они или отклонена. Потребитель только должен иметь возможность добавить обработчики для выполнения операций при выполнении, при неудаче или в процессе выполнения `Deferred`. И более того, вы хотите удостовериться, что потребитель не получит возможность перевести состояние `Deferred` в «выполнено» или «отклонено» по собственному усмотрению.

Чтобы это сделать, нужно вернуть объект `Promise` объекта `Deferred`, созданного в вашей функции тайм-аута, а не `Deferred` сам по себе. Позволит так поступить другой метод, названный `deferred.promise()`.

Синтаксис метода: deferred.promise

`deferred.promise([target])`

Возвращает объект Promise объекта Deferred.

Параметры

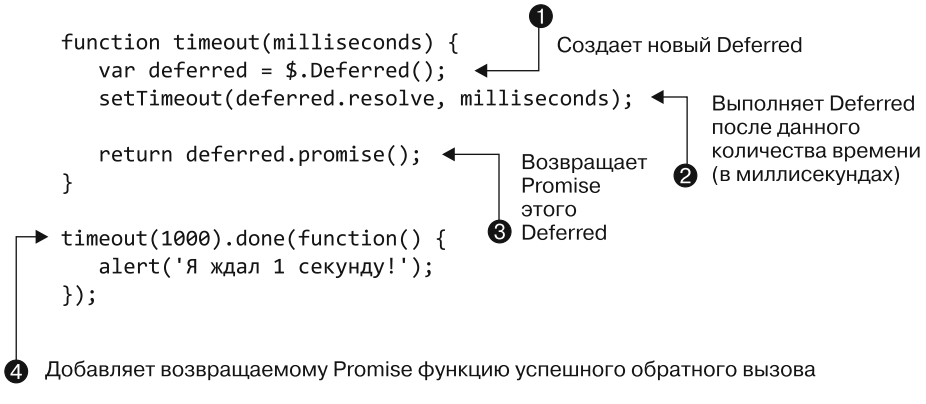
`target` (Объект) Объект, к которому прикрепляются методы промиса. Если этот параметр предоставлен, то метод прикрепляет к нему поведение `Promise` и возвращает данный объект, а не создает новый.

Возвращает

Объект `Promise`.

Теперь, когда вы узнали о существовании этого метода, напишем немного кода с его использованием. Код показан в листинге 13.5, доступен в файле `chapter-13/promise.timer.html`, предоставленном с книгой, и в JS Bin (http://jsbin.com/kefaza/edit?js,output).

Листинг 13.5. Таймер, основанный на промисе



В этом листинге вы определяете функцию, названную `timeout()`, которая оборачивает функцию JavaScript `setTimeout()`. Внутри `timeout()` создаете новый объект `Deferred` (вы производитель) ❶ и устанавливаете, что `setTimeout()` выполняет этот `Deferred` после данного количества миллисекунд ❷. По окончании возвращаете объект `Promise` этого `Deferred` ❸. Таким образом вы обеспечиваете следующее: потребитель, вызывающий вашу функцию ❹, не сможет выполнить или отклонить объект `Deferred`, а только добавит функции обратного вызова, такие как `deferred.done()` и `deferred.fail()`.

Предыдущий пример довольно прост и, возможно, недостаточно прояснил, когда использовать объект `Deferred`, а когда — `Promise`. Если у вас еще остались

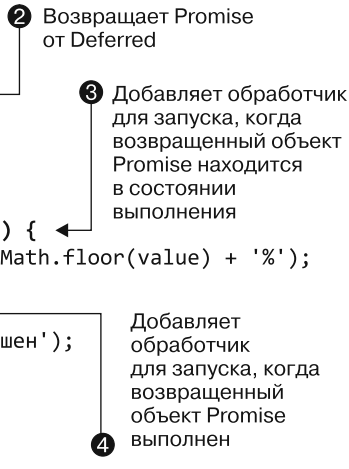
какие-либо сомнения, то обратимся к еще одному примеру. Мы пересмотрим нашу работу с индикатором выполнения так, что функция `animate()` будет заниматься только самим индикатором, а не обновлением текста, показывающего процент выполнения.

Новая версия этого кода показана в листинге 13.6 и также доступна в файле `chapter-13/deferred.progress.2.html`, предоставленном с книгой, и в JS Bin (http://jsbin.com/cefece/edit?html,css,js,output). Обратите внимание: в листинге мы опустили стилизацию страницы, чтобы сосредоточиться именно на коде. Кроме того, мы выделили полужирным шрифтом те части, которые изменились по сравнению с предыдущей версией.

Листинг 13.6. Использование deferred.progress(), версия 2

```
<!DOCTYPE html>
<html>
   <head>
      <meta charset="utf-8" />
      <title>Использование deferred.progress(), версия 2</title>
      <link rel="stylesheet" href="../css/main.css" />
   </head>
   <body>
      <div class="bar">
         <div class="inner-bar"></div>
         <span class="progress">0%</span>
      </div>

      <button id="run-button">Запуск анимации</button>

      <script src="../js/jquery-1.11.3.min.js"></script>
      <script>
         function animate(milliseconds) {
            var $innerBar = $('.inner-bar').width(0);
            var $barWidth = $('.bar').width();
            var step = $barWidth / 100;

            var deferred = $.Deferred();  <-- 1 Создает новый Deferred
            var idInterval = setInterval(
               function () {
                  // Не можем использовать width() для
                  // получения ширины из-за ошибки #1724
                  // (https://github.com/jquery/jquery/issues/1724)
                  var nextWidth =
                     parseFloat($innerBar.get(0).style.width) + step;
                  deferred.notify(nextWidth * 100 / $barWidth);
                  $innerBar.width(nextWidth);
                  if (nextWidth >= $barWidth) {
                     deferred.resolve();
                     clearInterval(idInterval);
                  }
```

```
            },
            milliseconds / 100
        );

        return deferred.promise();
    }

    $('#run-button').click(function() {
        animate(1000)
                .progress(function(value) {
                    $('.progress').text(Math.floor(value) + '%');
                })
                .done(function() {
                    alert('Процесс завершен');
                });
    });
  </script>
 </body>
</html>
```



Как видите, в данной версии функции `animate()` вы создаете только `Deferred` ❶. После того как код, запускающий анимацию, установлен, возвращаете его `Promise`, чтобы вызывающая этот метод функция могла добавлять функции обратного вызова ❷. Еще раз: вы возвращаете `Promise`, поскольку хотите разрешить вызывающей функции добавлять функции обратного вызова и при этом не разрешить менять состояние `Deferred`. В таком случае только функция, которая вызвала `Deferred`, может выполнить или отклонить его.

Наконец, внутри обработчика, прикрепленного к событию `click` кнопки, вы определяете функцию обратного вызова, которая выполняется в течение прохождения анимации, используя `deferred.progress()` ❸, а затем функцию обратного вызова для выполнения, когда `Deferred` выполнен, используя `deferred.done()` (этого не было в предыдущей версии) ❹.

Надеемся, благодаря примеру вы закрепили описанные понятия и методы. Самые наблюдательные могли заметить, что мы еще не рассмотрели единственный метод, упоминаемый в спецификациях и Promises/A, и Promises/A+, — `then()`. Исправим это.

13.3.7. Упростим код, используя then()

В разделе 13.1 мы в общих чертах рассказали о промисах, принятых в предложениях Promises/A и Promises/A+. Обе спецификации упоминают метод `then()`, но предложение Promises/A указывает, что он должен обладать третьим аргументом, который будет использован как обработчик для события прогресса, а предложение Promises/A+ такого аргумента не предполагает. Реализация данного метода в jQuery перед версией 3 (ветки 1.x и 2.x) отличаются от обоих этих предложений. Библиотека определяет первый аргумент, функцию обратного вызова для

успешного завершения как обязательную, и остальные две — функции обратного вызова для неудачного завершения и для работы в процессе — как необязательные. Для сравнения: предложения Promises/A и Promises/ A+ указывают, что все аргументы не являются обязательными.

Метод `deferred.then()` может использоваться как короткая запись для вызовов `deferred.done()`, `deferred.fail()` и `deferred.progress()`, когда нужно выполнить один обработчик для каждого или некоторых из этих методов. Если нужно больше обработчиков того же типа, то можно сделать цепочку из вызовов `then()`, `done()`, `fail()` и `progress()`. Более того, если не нужен обработчик данного типа, то можете просто передать `null`. Например, можно написать такую инструкцию:

```
$.Deferred()
  .then(success1)
  .then(success2, null, progress1)
  .done(success3);
```

Теперь, когда вы знаете, что может делать этот метод, изучим его синтаксис.

Синтаксис метода: deferred.then

`deferred.then(resolvedCallback[, rejectedCallback [, progressCallback]])`

Определяет обработчики, запускаемые после выполнения, отклонения или при выполнении работы объекта `Deferred`. Если какой-либо из параметров не нужен, то можно передать `null`.

Параметры

`resolvedCallback` (Функция) Функция, которая запускается после выполнения `Deferred`.

`rejectedCallback` (Функция) Функция, которая запускается после отклонения `Deferred`.

`progressCallback` (Функция) Функция, которая запускается, когда `Deferred` выполняется.

Возвращает

Объект `Promise`.

Метод `deferred.then()` возвращает объект `Promise` вместо `Deferred`. После вызова этого метода вы не сможете выполнить или отклонить использованный вами `Deferred`, если не сохранили ссылку на него. Ссылаясь на обработчик события `click` в кнопке, определенной в листинге 13.6, используя метод `deferred.then()`, можно переписать код следующим образом, получая тот же результат:

```
animate(1000).then(
  function() {
    alert('Процесс завершен');
  },
  null,
  function (value) {
    $('.progress').text(Math.floor(value) + '%');
  }
);
```

Вот что делает этот метод еще интереснее — он может перенаправлять значение, переданное как параметр, другому вызову `deferred.then()`, `deferred.done()`, `deferred.fail()` или `deferred.progress()`, определенному после него.

Прежде чем начать плакать в отчаянии, рассмотрим простой пример:

```
var deferred = $.Deferred();
deferred
  .then(function(value) { return value + 1; })
  .then(function(value) { return value + 2; })
  .done(function(value) { alert(value); });
deferred.resolve(0);
```

Данный код создает новый `Deferred` и затем добавляет три функции, когда он выполнен: две с использованием метода `deferred.then()` и одну — с использованием `deferred.done()`. Последняя строка выполняет `Deferred` со значением `0` (ноль), вызывая выполнение трех определенных функций (в порядке их добавления).

Внутри функций, добавленных с помощью `deferred.then()`, вы возвращаете новое значение, созданное начиная от принятого значения. Первая функция получает значение `0`, потому что это значение, переданное `deferred.resolve()`, добавляет к нему `1` и возвращает результат. Результат суммы передается следующей функции, добавленной путем `deferred.then()`. Вторая функция получает в качестве аргумента `1`, а не `0`. К данному значению (`1`) добавляется `2`, и возвращенный результат (`3`) применяется следующим обработчиком. На сей раз обработчик добавляется с использованием `deferred.done()`, у которого нет такой возможности, так что в сообщении вы получаете конечное значение `3`.

Если вы добавите еще одну функцию, задействуя `deferred.done()`, в цепочку в предыдущем примере и вернете измененное значение от третьего в цепочке (добавленного с помощью `deferred.done()`), то новый обработчик получит то же значение, что и третий. Следующий код выведет сообщение со значением `3` дважды:

```
var deferred = $.Deferred();
deferred
  .then(function(value) { return value + 1; })
  .then(function(value) { return value + 2; })
  .done(function(value) { alert(value); return value + 5; })
  .done(function(value) { alert(value); });
deferred.resolve(0);
```

В библиотеках, совместимых с Promises/A и Promises/A+ (например, jQuery 3.x), выброшенное исключение переводится в отклонение и функция обратного вызова для неудачи вызывается с помощью исключения. В jQuery 1.x и 2.x непойманные исключения остановят выполнение программы. Рассмотрим следующий код:

```
var deferred = $.Deferred()
deferred
  .then(function(value) {
    throw new Error('Сообщение об ошибке');
  })
  .fail(function(value) {
    alert('Ошибка');
  });
deferred.resolve();
```

В jQuery 3.x вы увидите сообщение «Ошибка». А у jQuery 1.x и 2.x исключение может уйти к `window.onerror`. Если функция для этого не определена, то в консоли вы увидите Uncaught Error: Сообщение об ошибке.

Вы можете продолжать исследовать эту проблему, чтобы лучше понять различия в поведении. Рассмотрим следующий код:

```
var deferred = $.Deferred();
deferred
  .then(function() {
    throw new Error('Сообщение об ошибке');
  })
  .then(
    function() {
      console.log('Первая успешная функция');
    },
    function() {
      console.log('Первая неудачная функция');
    }
  )
  .then(
    function() {
      console.log('Вторая успешная функция');
    },
    function() {
      console.log('Вторая неудачная функция');
    }
  );
deferred.resolve();
```

В jQuery 3.x данный код выведет в консоль сообщения `Первая неудачная функция` и `Вторая успешная функция`. Причина этого, как мы упоминали, в том, что выбрасываемое исключение следует перевести в отклонение и должна быть вызвана функция обратного вызова для случая неудачного завершения с исключением. Вдобавок после управления исключением (в нашем примере с помощью неудачной функции обратного вызова, переданной второму `then()`, следующие успешные функции должны быть вызваны (в этом случае успешная функция обратного вызова передается третьему `then()`).

В jQuery 1.x и 2.x ничего, кроме первой функции (выбрасывающей ошибку) не вызывается. Вы только увидите сообщение в консоли Uncaught Error: Сообщение об ошибке.

jQuery 3: добавленный метод

В jQuery 3 к объектам `Deferred` и `Promise` добавлен новый метод, названный `catch()`. Это метод для определения обработчика, выполняющегося, когда объект `Deferred` отклоняется или состояние его объекта `Promise` «отклонен». Его сигнатура следующая:

```
deferred.catch(rejectedCallback)
```

Данный метод — не что иное, как сокращенный вариант `then(null, rejectedCallback)` и добавлен в jQuery 3 для лучшего соответствия спецификациям ECMAScript 2015, которые включают метод с таким именем.

Несмотря на отличия библиотеки jQuery от спецификаций, `Deferred` остается невероятно эффективным инструментом, доступным для вас. Как профессиональный разработчик с возрастающей сложностью проектов, вы будете использовать эту функцию довольно часто.

Иногда, вне зависимости от состояния `Deferred`, вы захотите выполнять одно или несколько действий. У jQuery есть метод и для такого случая.

13.3.8. Всегда запускайте обработчик

Иногда вы можете захотеть запустить один обработчик или сразу несколько вне зависимости от состояния `Deferred`. Это можно сделать, воспользовавшись методом `deferred.always()`.

Синтаксис метода: deferred.always

`deferred.always(callbacks[, callbacks, ..., callbacks])`

Добавляет обработчики, которые вызываются, когда объект `Deferred` либо выполнен, либо отклонен. Метод принимает произвольное количество функций обратного вызова, начиная от одной.

Параметры

`callbacks` (Функция|Объект). Функция или массив функций, который вызывается, когда `Deferred` либо выполнен, либо отклонен.

Возвращает

Объект `Deferred`.

Рассмотрим следующий код.

```
var deferred = $.Deferred();
deferred
  .then(
    function(value) {
      console.log('успешно: ' + value);
    },
    function(value) {
      console.log('неудачно: ' + value);
    }
  )
  .always(function() {
    console.log('Я всегда появляюсь');
  });
deferred.reject('ошибка');
```

При выполнении этого кода вы увидите два сообщения в консоли. Первое — `Неудачно: ошибка`, поскольку `Deferred` был отклонен. Второе — `Я всегда появляюсь`, потому что функции обратного вызова, добавленные с помощью `deferred.always()`, выполняются вне зависимости от выполнения или отклонения `Deferred`.

Осталось обсудить один метод.

13.3.9. Определение состояния Deferred

При написании кода, использующего `Deferred`, вы можете захотеть проверить его текущее состояние. Для этого пригодится метод `deferred.state()`. Он делает именно то, что вы и предполагаете: возвращает строку, которая указывает текущее состояние `Deferred`. Его синтаксис представлен ниже.

Синтаксис метода: deferred.state

`deferred.state()`

Определяет текущее состояние объекта `Deferred`. Возвращает строку, которая может быть `"resolved"`, `"rejected"` или `"pending"`.

Возвращает

Строку, соответствующую состоянию `Deferred`.

Данный метод будет особенно полезен для модульного тестирования кода. Например, вы хотите написать такую инструкцию:

```
assert.equal(deferred.state(), 'resolved');
```

Метод, используемый в предыдущей инструкции, пришел из фреймворка для модульного тестирования QUnit, о котором мы поговорим в следующей главе. Он просто проверяет, что первый параметр равен второму, и, если это так, тест пройден.

13.4. Промисификация всего на свете

В предыдущих разделах мы сосредоточились на объектах `Deferred` и `Promise` и их методах, но у нас есть еще небольшой сюрприз — метод `promise()`. Он отличается от метода `deferred.promise()`, рассмотренного немного выше. Метод `promise()` разрешает преобразовать объект jQuery в объект `Promise`, позволяя тем самым добавлять обработчики, используя методы, обсуждавшиеся в этой главе. Синтаксис данного метода показан ниже.

Синтаксис метода: promise

`promise([type][, target])`

Возвращает динамически созданный объект `Promise`, который выполняется, когда все действия определенного типа, объединенные в коллекцию, по очереди или нет, завершены. По умолчанию используется тип `fx`. Это значит, что возвращенный `Promise` разрешен, когда все анимации у выбранных элементов завершены.

Параметры

`type` (Строка) Тип очереди, которая должна соблюдаться. Значение по умолчанию — `fx`, представляющий собой очередь по умолчанию для эффектов.

`target` (Объект) Объект, которому должны быть добавлены методы `promise`.

Возвращает

Объект `Promise`.

На основании описания этого метода, если у применяемого объекта jQuery нет используемых анимационных эффектов, он выполняется сразу. Таким образом, любой прикрепленный обработчик запускается немедленно. Рассмотрим следующий код:

```
$('p')
  .promise()
  .then(function(value) { console.log(value); });
```

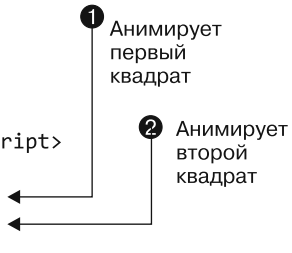
Если абзацы на странице не запускают анимации, то функция, добавленная с помощью `then()`, выполняется немедленно и переданное ей значение — это сам объект jQuery.

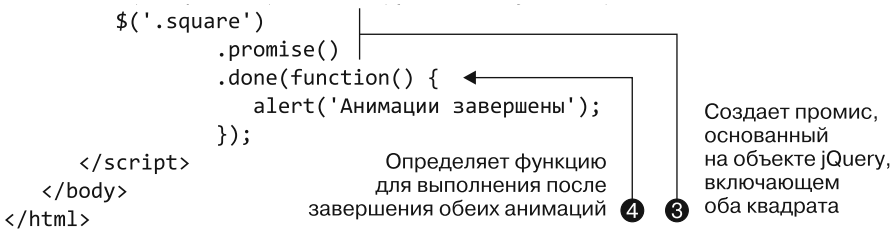
Теперь рассмотрим чуть более сложный пример, где задействованы анимации с использованием метода `animate()`, который вы изучили в главе 8. В данном примере вы создаете два квадрата и двигаете их слева направо с разной скоростью, так что анимации закончатся в разное время. После завершения обеих анимаций вы получите сообщение об успешном выполнении.

Код для достижения этой цели показан в листинге 13.7, а также доступен в файле `chapter-13/promisify.html`, предоставленном с книгой, и в JS Bin (http://jsbin.com/wuhiqa/edit?js,output).

Листинг 13.7. Промисификация коллекции jQuery

```
<!DOCTYPE html>
<html>
   <head>
      <meta charset="utf-8" />
      <title>Промисификация объекта jQuery</title>
      <link rel="stylesheet" href="../css/main.css" />
      <style>
          .square
          {
             position: relative;
             width: 25px;
             height: 25px;
             background-color: #1E39BC;
             margin: 10px 0;
          }
      </style>
   </head>
   <body>
      <div id="square1" class="square"></div>
      <div id="square2" class="square"></div>

      <script src="../js/jquery-1.11.3.min.js"></script>
      <script>
         $('#square1').animate({left: 500}, 1500);
         $('#square2').animate({left: 500}, 3000);
```





В коде данного листинга вы анимируете оба квадрата, но первая анимация длится 12 500 миллисекунд ❶, а вторая — 3000 ❷. После определения этих анимаций выбираете оба квадрата, используя имя их класса, `square`, и создаете объект `Promise`, используя рассмотренный в этом разделе метод `promise()` ❸. Наконец, добавляете функцию, которая будет выполнена после завершения обеих анимаций ❹. Замечательно, не так ли?

13.5. Резюме

В данной главе мы познакомили вас с промисами, лучшим шаблоном для работы с асинхронным кодом. Промисы позволяют избегать использования сложных приемов с синхронизацией параллельных асинхронных функций и избавляют от необходимости вкладывать функцию обратного вызова внутри функции обратного вызова внутри еще одной функции обратного вызова.

Мы вкратце описали предложения Promises/A и Promises/A+ и сравнили их с реализацией промисов в jQuery. Кроме того, проиллюстрировали различие между двумя предложениями, обратив внимание на принятую терминологию и количество параметров, заданных для метода `then()`, ключевого в промисах.

Реализация промисов в jQuery включает два объекта: `Deferred` и `Promise`. Как сказано выше, последний создан из объекта `Deferred` или jQuery и обладает поднабором методов от `Deferred`.

Мы углубились в множество методов, предлагаемых `Deferred`. Последний может быть выполнен или отклонен, в первом случае с помощью метода `deferred.resolve()`, во втором — `deferred.reject()`. Вы будете также извещены о ходе выполнения асинхронных операций благодаря использованию `deferred.notify()`. У всех этих методов есть родственные методы, которые принимают дополнительный параметр, позволяющий вам установить контекст запущенных функций: `deferred.resolveWith()`, `deferred.rejectWith()` и `deferred.notifyWith()`.

Методы `deferred.done()`, `deferred.fail()` и `deferred.progress()` позволяют разрешить добавлять обработчики для запуска, когда `Deferred` выполнен, отклонен или находится в процессе выполнения.

Еще одно ключевое понятие — вспомогательная функция `$.when()`, позволяющая просто синхронизировать несколько асинхронных или даже синхронных функций.

Мы познакомили вас с реализацией метода jQuery `then()`, подчеркивая различия, существующие между jQuery и другими библиотеками, придерживающимися либо предложения Promises/A, либо Promises/A+.

Эта глава также описывает другие методы, предлагаемые объектом `Deferred`, такие как `deferred.always()` и `deferred.state()`.

Наконец, мы обсудили `promise()`, метод для преобразования объекта jQuery в объект `Promise`. Данный метод очень полезен, поскольку позволяет использовать возможности `Deferred` простыми объектами jQuery.

Теперь мы закончили наш обзор библиотеки jQuery. Надеемся, благодаря прочитанным главам вы изучили все плюсы и минусы этой библиотеки и теперь можете считать себя jQuery-ниндзя! Поздравляем!

В следующей главе мы постараемся сделать шаг вперед, обсуждая ключевое понятие, которое должен усвоить каждый профессиональный разработчик, — модульное тестирование. Переверните страницу и продолжайте процесс изучения.

14 Модульное тестирование с jQuery

В этой главе:

- почему тестирование — это важно;
- что такое модульное тестирование;
- установка QUnit;
- тестирование вашего кода JavaScript с QUnit;
- как создать полный тестовый комплект с QUnit.

Предыдущая глава была последней, где мы обсуждали непосредственно основы jQuery. `Deferred` и `Promise` было, возможно, непросто изучить, но надеемся, что с нашей помощью этот процесс прошел гладко.

А теперь пришло время еще больше улучшить навыки. В данной главе вы изучите другие инструменты и методы, которые должен знать каждый профессионал. Вы будете применять их к коду, написанному с использованием jQuery, но можете прибегать к этим знаниям при работе с любым кодом, написанным на JavaScript. Тестирование — одно из важнейших понятий, которое нужно усвоить, если вы работаете в команде или делаете что-то не только для себя.

Эти важные темы, которые вы изучите в данной главе, — тесты и создание утверждений. Утверждение — способ проверить, что ваш код работает корректно и его поведение соответствует ожидаемому. Иначе говоря, утверждение проверяет, возвращает ли функция значение, которое вы ожидаете получить на основе конкретного набора входных данных, или значение переменной, или свойства объекта, которые вы ожидаете получить после выполнения определенных операций с ним.

Наконец, после краткого введения о том, что такое модульное тестирование и для чего оно может понадобиться, мы обсудим QUnit. Это одна из библиотек, которой в сообществе JavaScript пользуются для проверки кода, написанного на JavaScript. Причина, почему мы поговорим о QUnit, а не о какой-то другой библиотеке, состоит в том, что QUnit — фреймворк, используемый всеми проектами jQuery (jQuery, jQuery UI и jQuery Mobile) для модульного тестирования кода.

А сейчас поговорим о тестировании, в особенности модульном, и о QUnit.

14.1. Почему тестирование важно

Чтобы объяснить, почему тестирование ПО важно, начнем с примера. Какой ответ вы бы дали, если бы мы предложили проехать на машине, которую никогда не тестировали, на гонках «Формулы-1»? Вы бы рискнули жизнью, используя автомобиль, прочность которого никто ни разу не проверил, не зная, не разнесет ли его на первом повороте и действительно ли тормоза работают? Конечно же, вы бы на этот вопрос ответили категорическим «Нет!». Такой же принцип применим и к ПО. Никто не хочет использовать программы, которые «слетают» каждые две минуты, или работают не так, как нужно, или, хуже того, не соответствуют требованиям и ожиданиям пользователя. Именно поэтому на помощь приходит тестирование ПО.

Тестирование ПО — процесс оценки части программы, чтобы определить различия между ожидаемым и фактическим результатами с имеющимися входными данными. Оно помогает создать безопасное, надежное и стабильно работающее ПО. Тестирование также служит показателем того, как ваш продукт отличается от спецификаций, ожиданий заказчиков, предыдущих версий этого продукта и множества других критериев. Другой основной целью является выявление дефектов кода, ошибок, которые часто называют багами (англ. bug — «жук»; «технический дефект»).

Говоря о выявлении ошибок, мы не имеем в виду, что тестирование позволит проверить все возможные условия и найти абсолютно все ошибки, это просто невозможно ни в какой реальной системе. В рамках тестирования следует сделать следующее: проверить ваш код, продукт и систему в определенных условиях — нужно убедиться, что все работает так, как предполагается. При тестировании ошибки будут всегда. Их трудно искоренить, даже используя практический опыт или знания, так как они присущи абсолютно каждой сложной системе. Вам нужно признать: ошибки в ПО существуют не потому, что вы или другие разработчики плохо работаете, а потому, что любое реальное ПО настолько сложное, что вы не можете предугадать, а следовательно, и исправить каждый источник возможного сбоя.

Тестирование — неотъемлемая часть жизни каждого разработчика. Ну, так должно быть, во всяком случае. К сожалению, многие разработчики боятся тестирования. Это часть, воспринимается как дополнительная деятельность, как что-то, вынуждающее вас тратить время — и много времени. Конечно, вы не должны бросаться в крайности. Если вы разрабатываете действительно небольшой кусок кода для себя с целью автоматизации процесса, который вы выполните один раз в жизни, то тестирование может и не стоить усилий. Но опытные разработчики могут подтвердить: если вы разрабатываете даже небольшой проект или библиотеку, которые будете использовать в вашей повседневной работе и которыми будете делиться с вашей командой, другими разработчиками или всем сообществом JavaScript, то вам лучше их протестировать.

Тестирование — невероятно обширная дисциплина. Вы можете проверить многие аспекты проекта (например, тестирование совместимости и регрессионное тестирование), используя различные методы (например, визуальное тестирование и «тестирование методом черного ящика») и на разных уровнях (например, модульное и интеграционное тестирование). Но цель данного раздела состоит не в обучении вас всем премудростям тестирования ПО. Данная тема настолько обширна, что нам потребовалось бы написать отдельную книгу, чтобы эту тему осветить. Мы хотим обсудить здесь важность тестирования и причины, по которым вам следует тестировать ваши программы, если вы этого еще не делаете.

В следующем подразделе мы дадим обзор одного из типов тестирования, упомянутого только что, — модульного.

14.1.1. Почему модульное тестирование?

Модульное тестирование — это метод тестирования ПО, когда ПО рассматривается как набор отдельных кусков кода, называемых *модулями (unit)*. Можно проверить каждый модуль по отдельности, чтобы удостовериться, что все работает так, как полагается. При модульном тестировании каждый набор тестов, касающихся отдельного модуля, должен быть независим от других. Обычно модуль определяется функцией или методом, в зависимости от типа принятого языка программирования.

Вкратце основные преимущества модульного тестирования таковы:

- проверяет, возвращает ли код ожидаемый результат на основании определенных входных данных;
- определяет максимальное количество дефектов на ранней стадии (основываясь на предыдущем пункте);
- улучшает проектирование кода;
- определяет модули, которые являются слишком сложными.

Придерживаясь принципов модульного тестирования, на основании функции вашего ПО и набора входных данных вы можете определить, возвращает ли функция ожидаемые результаты. Данный процесс обычно автоматизирован и включает фреймворк для модульного тестирования. Оно состоит из написания функций, передающих при выполнении набор входных данных, которые вы определили для целевой функции (той, которую вы тестируете). Затем эти функции проверяют, соответствует ли для каждого набора входных данных результат ожидаемому (тому, который вы определили в тестовой функции). Используя указанный метод, можно также проверить следующий момент: вы передаете неправильные входные данные целевой функции, а последняя корректно справляется с ними (это также может означать выброс ожидаемой ошибки или исключения). Когда один или несколько тестов завершаются неудачно, вы знаете, что в коде модуля ошибка и нужно ее исправить. Процесс повторяется, пока все тесты не будут пройдены (с результатами, соответствующими ожидаемым).

Цель модульного тестирования состоит в поиске максимально возможного количества дефектов ПО на ранней стадии процесса разработки. Еще одно преимущество в том, что после создания всех тестов для модуля можно уверенно улучшать код (функцию или метод). Если вы допустите ошибку при обновлении кода, один или несколько тестов (обычно) завершатся неудачно и, таким образом, сразу станет ясно: что-то пошло не так. Вы сможете менять свой код увереннее, зная, что не нарушите правильно работавшую функцию.

Еще одно преимущество, которое всегда ассоциируется с модульным тестированием, — оно помогает понять и улучшить проектирование кода. Вместо того чтобы писать код, выполняющий некие операции, вы начинаете с обрисовывания всех условий, с которыми он должен столкнуться, и ожидаемым результатом. Это понятие обычно связывают с методологией, называемой *test-driven development*, TDD (разработкой через тестирование).

Разработка через тестирование

Разработка через тестирование — это процесс разработки ПО, основанный на написании тестов перед написанием кода (модуля), который будет тестироваться. Первый шаг в TDD — написать первоначально завершающийся неудачей тестовый случай (test case). Затем разработчик должен написать код, реализующий функционал, пока все тесты не будут завершены. Наконец, код перерабатывается, пока не будет соответствовать приемлемому стандарту качества.

Последнее преимущество, которое мы хотим подчеркнуть, — модульное тестирование помогает определить слишком сложные модули. Например, вы можете увидеть, что метод делает больше, чем должен в соответствии со своей основной целью. Помните об одном из основополагающих принципов ООП — принципе единственной ответственности (single responsibility principle, SRP). Иногда при написании тестового кода наступает момент, когда кажется, что он слишком сложен или становится чересчур громоздким. Это говорит о том, что методы должны быть разделены или улучшены.

Если проект разрабатывается с помощью JavaScript, то есть еще одна причина воспользоваться тестированием — несовместимость браузеров. Хотя все крупные браузеры, включая Internet Explorer, с каждым днем все больше и больше придерживаются веб-стандартов, их поведение по-прежнему во многих случаях очень и очень разнится. Наличие надежных тестов — один из способов избежать проблем с развертыванием кода, который работает в одних определенных браузерах и ломает ваши веб-страницы в других. Вы можете запустить те же тесты в различных браузерах, чтобы убедиться, что во всех они будут выполнены.

Теперь, когда вы представляете, что такое тестирование в общем и модульное в частности и хорошо понимаете, почему должны его использовать , пришло время посмотреть на фреймворки для модульного тестирования, доступные для вашего JavaScript-кода.

14.1.2. Фреймворки для модульного тестирования JavaScript

Возможно, вы знаете эту шутку, довольно известную среди JavaScript-разработчиков: вы должны загадать слово, поискать в Google `<слово>.js`, и если библиотека с таким именем существует, то выпить. Если не знали, то теперь уже знаете. Суть шутки не в том, чтобы быстро напиться, а в том, чтобы обратить внимание на наличие в мире огромного количества JavaScript-библиотек, фреймворков и плагинов. То же можно было бы сделать с фреймворками для модульного тестирования.

Сообщество JavaScript предлагает множество библиотек, которые можно использовать для модульного тестирования проектов. Но, как и само ПО, библиотеки для тестирования приходят и уходят. Прежде чем выбрать свою, убедитесь, что она по-прежнему поддерживается. В этой главе мы дадим краткий обзор ряда наиболее популярных фреймворков для модульного тестирования в JavaScript.

QUnit (http://qunitjs.com/) — первый фреймворк для модульного тестирования, который мы хотим представить. Он был разработан для тестирования jQuery, но затем превратился в отдельный фреймворк для модульного тестирования. Был задействован во всех остальных проектах, организованных командой jQuery, но может работать с любым кодом, основанным на JavaScript. QUnit поддерживает те же браузеры, что и jQuery 1.x. Одним из преимуществ этого фреймворка является то, что он предоставляет простой в применении набор инструментов для тестирования проекта. Помимо использования обычных методов утверждения, QUnit позволяет тестировать асинхронные методы.

Mocha (http://mochajs.org/) — многофункциональный фреймворк тестирования JavaScript, работающий под Node.js и браузером. Mocha запускает тесты последовательно, что позволяет гибко и точно сообщать о событиях, в то же время сопоставляя неперехваченные исключения с правильными тестовыми сценариями.

Jasmine (http://jasmine.github.io/) — фреймворк для JavaScript с открытым исходным кодом, в основе которого лежит *разработка через функционирование* (behavior-driven development, BDD). Его синтаксис — чистый и удобный для чтения.

Разработка через функционирование

Разработка через функционирование — процесс разработки ПО, эволюционировавший из разработки через тестирование. При использовании этого процесса вы тестируете не только код в исходном состоянии, применяя модульные тесты, но и приложение с помощью приемочных тестов. BDD указывает, что тесты любого модуля ПО должны быть определены с точки зрения желаемого поведения модуля.

Другие фреймворки, о которых можно прочитать в Интернете и о которых мы хотели бы упомянуть, — YUI Test (http://yuilibrary.com/yui/docs/test/) и Selenium (http://docs.seleniumhq.org/). К какому фреймворку прибегнуть — зависит от ваших

предпочтений, от навыков, которыми владеете вы и ваша команда, от того, какой подход хотите выбрать (TDD или BDD). В оставшейся части этой главы мы будем обсуждать QUnit, потому что, как мы уже упоминали, данный фреймворк поддерживается и используется командой jQuery. Если они доверяют QUnit, то почему бы не доверять и вам? Посмотрим, как можно начать его применять.

14.2. Начинаем работу с QUnit

Пожалуй, лучшая особенность QUnit состоит в том, что использовать его легко. Начало работы с этим фреймворком — дело трех простых шагов.

Первый шаг — загрузить фреймворк. QUnit можно скачать несколькими разными способами. Первый — зайти на его сайт и загрузить файлы JavaScript и CSS в последней доступной версии.

Файл JavaScript содержит модуль, запускающий тест (код, отвечающий за запуск теста) и фактический тестовый фреймворк (набор методов, применяемых для тестирования кода); файлы CSS стилизуют страницу тестового комплекта, используемую для отображения результатов тестов. Ссылки на эти файлы можно найти в правой верхней части главной страницы, как показано на рис. 14.1.

Рис. 14.1. Главная страница фреймворка QUnit

Второй шаг — переместить файлы в папку, где вы также создадите HTML-файл. Он должен содержать ссылку на файлы CSS и JavaScript, равно как и обязательный элемент (обычно `<div>`), имеющий ID `qunit`. Внутри него фреймворк создает элементы, которые составляют пользовательский интерфейс, используемый для группировки тестов и отображения результатов. Полученный HTML-код должен выглядеть так, как показано в листинге 14.1.

Листинг 14.1. Минимальная установка фреймворка QUnit

```
<!DOCTYPE html>
<html>
   <head>
      <meta charset="utf-8" />
      <title>Тесты QUnit</title>
      <link rel="stylesheet" href="qunit-1.18.0.css" />
   </head>
   <body>
      <div id="qunit"></div>
      <script src="qunit-1.18.0.js"></script>
   </body>
</html>
```

Включает таблицу стилей фреймворка

Оборачивает пользовательский интерфейс QUnit

Добавляет JavaScript-файл QUnit

И последний шаг — открыть HTML-файл в выбранном вами браузере. После этого вы увидите страницу, подобную представленной на рис. 14.2.

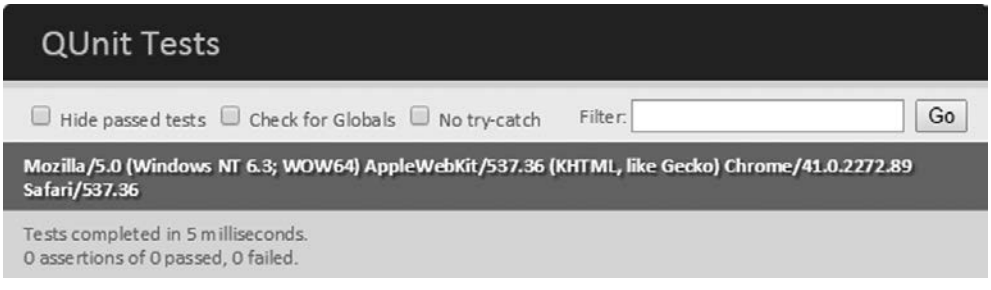


Рис. 14.2. Пользовательский интерфейс фреймворка QUnit

Достаточно просто, правда? Проанализируем компоненты этого пользовательского интерфейса.

Верхняя часть интерфейса — не что иное, как содержание элемента `title`, размещенного в `head` страницы (QUnit Tests). Под заголовком находится зеленая полоса. Ее цвет обусловлен тем, что все тесты пройдены (отсутствие тестов подразумевает корректную работу). Эта полоса становится красной, если один или несколько определенных вами тестов прошли неудачно.

Во втором разделе находятся три флажка: Hide passed tests, Check for Globals (иногда называется noglobals) и No try-catch. Когда установлен первый флажок, QUnit будет скрывать успешно пройденные тесты (те, которые вернули ожидаемый результат). Второй флажок позволит проверять, добавлено ли свойство к объекту `window`, сравнивая его до и после каждого теста, в случае чего тест завершится неудачей, показывая различия. Последний флажок может быть использован для проверки, выбрасывает ли код исключение, когда вы даете QUnit команду запустить тест вне блока `try-catch`. Помимо этих флажков, есть элемент `input`, который будет использован для фильтрации результатов теста, поиска определенного теста.

В третьем разделе можно увидеть значение свойства `window.navigator.userAgent`. Оно возвращает строку пользовательского агента для браузера, который обращается к странице.

Нижняя часть показывает время, затраченное QUnit на совершение определенных тестов. Там вы можете прочитать, сколько *утверждений* определено, сколько прошло и сколько завершилось неудачно.

Что такое утверждение

Утверждение проверяет, что инструкция возвращает `true`. Это полезно для тестирования того, что ваш код возвращает ожидаемый результат. Важно отметить: утверждение должно использоваться только для тестирования *значимого* кода. Мы имеем в виду, что нужно проверить только основы вашего кода — и ничего более. Например, если код применяет встроенную функцию JavaScript, то тестировать ее совершенно бессмысленно. Следует предполагать, что у функции JavaScript (например, `getElementById()`) нет ошибок, даже несмотря на то, что иногда это не совсем так.

Как видите, все эти значения равны нулю. Причина в том, что мы еще не определили ни один тест. У нас вообще нет утверждений — ни прошедших, ни не прошедших. Когда вы пишете какие-то тесты и утверждения, QUnit перечисляет здесь утверждения, сгруппированные по тесту.

Если вы в процессе загрузки файлов уже успели просмотреть официальную документацию, то могли заметить, что в ней указывается другой элемент как обязательный в минимальной конфигурации установки фреймворка. Это элемент (обычно `<div>`), у которого ID `qunit-fixture`. Его цель состоит в том, чтобы предотвратить успешное или неудачное выполнение ряда тестов как побочный эффект от предыдущего запуска тестов, таких как удаление или добавление элементов в DOM. Данный элемент полезен, поскольку фреймворк сбрасывает элементы внутри него после каждого теста. Таким образом, это не обязательно, но вам нужно включить его в любой реальный проект.

Другие способы получения QUnit

QUnit можно получить по-разному. Первый возможный способ — включение нужных файлов, используя jQuery CDN. Чтобы сделать это с помощью версии 1.18.0 (текущая на момент написания книги), нужно включить в вашу страницу следующий код:

```
<link rel="stylesheet" href="//code.jquery.com/qunit/qunit-1.18.0.css"/>
<script src="//code.jquery.com/qunit/qunit-1.18.0.js"></script>
```

Еще один способ — загрузить QUnit через Bower, запустив следующую команду:

```
bower install --save-dev qunit
```

Наконец, можно получить фреймворк через npm:

```
npm install --save-dev qunitjs
```

С установленным фреймворком вы готовы к тестированию кода. Но прежде, чем изучить, как это делается, нужно внести пару изменений. Первое — добавить на вашу страницу код или файл JavaScript, который хотите проверить (например, `code.js`). Можно разместить его где угодно при условии, что он идет перед кодом, который вы написали для его проверки. Второе изменение — добавить код или файл, содержащий тесты. Как правило, они размещаются в другом JavaScript-файле, нередко названном `tests.js`. Данный файл должен быть включен после элемента `script`, который вы использовали для включения JavaScript-файла, принадлежащего QUnit. У вас должен быть код, выглядящий так:

```
<script src="code.js"></script>
<script src="qunit-1.18.0.js"></script>
<script src="tests.js"></script>
```

Содержимое этих файлов также должно быть встраиваемым (с содержимым внутри элемента `script`). В примерах данной главы мы будем встраивать тестовый код и иногда даже код, который нужно проверить, из-за его простоты, но мы всегда настоятельно рекомендуем использовать внешний файл, когда вы применяете QUnit в реальном проекте.

Усвоив последнее замечание, вы готовы узнать, что может предложить этот фреймворк.

14.3. Создание тестов для синхронного кода

С помощью QUnit вы сможете проверять и синхронный, и асинхронный код. Сейчас мы сосредоточимся на тестировании синхронного кода, поскольку так проще углубиться в мир QUnit.

Для создания теста в QUnit вы должны использовать метод, который называется `QUnit.test()`. Его синтаксис показан ниже.

Синтаксис метода: QUnit.test

`QUnit.test(name, test)`

Добавляет тест для запуска.

Параметры

`name`	(Строка) Имя для идентификации созданного теста.
`test`	(Функция) Функция, содержащая утверждения для запуска. Фреймворк передает аргумент `assert` этой функции. Он предоставляет все методы утверждения QUnit.

Возвращает

`undefined`.

Чтобы создать новый тест, напишите следующий код:

```
QUnit.test('Мой первый тест', function(assert) {
  // Код здесь...
});
```

Если вы поместите этот тест в упоминавшийся уже файл `test.js` или встроите его и затем откроете HTML-файл в браузере, то увидите ошибку. Фреймворк напишет, что вы определили тест без единого утверждения. Какой смысл в определении теста, если в нем нет вообще никакого тестового кода?

При создании теста хорошей манерой считается установка количества утверждений, которое вы ожидаете выполнить. Это можно сделать с помощью метода `expect()` параметра `assert`, описанного при обсуждении `QUnit.test()`. Если вы работаете только с синхронным кодом, использование `expect()` кажется бесполезным — вы можете полагать, что единственной причиной, по которой утверждение не запустится, может быть ошибка, вызванная фреймворком. Это действительно так, пока вы не учитываете асинхронный код. На данный момент просто доверьтесь нам и применяйте `expect()`.

Синтаксис данного метода представлен ниже.

Синтаксис метода: expect

`expect(total)`

Устанавливает количество утверждений, выполнение которых ожидается в тесте. Если количество фактически выполненных утверждений не соответствует параметру `total`, то тест завершается неудачно.

Параметры

`total` (Число) Количество утверждений, которое планируется выполнить в тесте.

Возвращает

`undefined`.

Теперь, вооружившись знаниями о методе `expect()`, вы можете изменить предыдущий тест для установки ожидания запуска нулевого количества тестов:

```
QUnit.test('Мой первый тест', function(assert) {
  assert.expect(0);
});
```

Обновив HTML-страницу, вы заметите исчезновение предыдущего сообщения об ошибке. Причина заключается в следующем: вы явно установили количество выполненных утверждений равным нулю, и теперь QUnit полагает, что это именно то, что вы и хотели.

Использовав `expect()`, вы исправили ошибку на HTML-странице, но по-прежнему нет утверждений. Рассмотрим, что может предложить QUnit.

14.4. Тестирование кода с использованием утверждений

Утверждения — это основа тестирования ПО, ведь они позволяют проверить, работает ли код так, как ожидается. QUnit предоставляет множество методов для проверки ваших ожиданий, все они доступны в тесте через параметр `assert`, передаваемый функции, указанной в качестве параметра `QUnit.test()`.

Начнем наш обзор методов утверждений с рассмотрения четырех из них.

14.4.1. equal(), strictEqual(), notEqual() и notStrictEqual()

В этом подразделе мы рассмотрим четыре метода, предоставляемых QUnit. Первый метод, о котором хотим рассказать, — `equal()`.

Синтаксис метода: equal

`equal(value, expected[, message])`

Проверяет эквивалентность параметра `value` параметру `expected`, используя нестрогое сравнение (==).

Параметры

`value`	(Любое) Значение, возвращаемое функцией или методом, или хранимое в переменной, которое нужно проверить.
`expected`	(Любое) Значение, с которым проводится сравнение.
`message`	(Строка) Необязательное описание утверждения. Если оно пропущено, то показывается сообщение okay в случае успешного и failed в случае неудачного прохождения теста.

Возвращает

`undefined`.

Описание утверждения, параметр `message`, необязателен, но мы рекомендуем всегда его использовать.

Чтобы дать представление о том, как применяется этот метод, посмотрим на такой пример. Допустим, вы создали функцию, которая суммирует два числа, передаваемых как аргументы. Определение данной функции в JavaScript будет следующим:

```
function sum(a, b) {
  return a + b;
}
```

Теперь, написав функцию `sum()`, вы хотите проверить корректность ее работы. Чтобы сделать это, используя метод `equal()`, можно написать такой тест:

```
QUnit.test('Мой первый тест', function(assert) {
  assert.expect(3);
  assert.equal(sum(2, 2), 4, 'Сумма двух положительных чисел');
```

```
  assert.equal(sum(-2, -2), -4, 'Сумма двух отрицательных чисел');
  assert.equal(sum(2, 0), 2, 'Сумма положительного числа и нейтрального
              элемента');
});
```

Вы можете запустить тест, открыв файл `chapter-14/test.1.html`, предоставленный с книгой, или через соответствующий JS Bin (http://jsbin.com/towoxa/edit?html,output).

Запустив предыдущий тест, вы будете уверены в том, что ваш код работает правильно, функция написана верно и работает без сбоев. Но так ли это? Оказывается, нет. А если вы добавите вот такие утверждения к тесту? (Помните, что нужно также обновить `assert.expect()`.)

```
assert.equal(sum(-1, true), 0, 'Сумма отрицательного элемента и true');
```

Это утверждение будет верным, поскольку JavaScript — слабо типизированный язык, так что `true` равен `1`. Таким образом, `-1` плюс `true` дадут в результате `0`.

Кроме проблемы с передачей логического типа, что будет, если вы передадите число или строку вроде `"foo"` в `sum()`? В данной ситуации функция будет воспринимать оба параметра как строки и конкатенирует их. Иногда это может оказаться правильным результатом, но лучше такие случаи обрабатывать явно. Например, вы хотите сгенерировать исключение, если один или несколько параметров не являются типом `Number`, или преобразовывать их перед суммированием. Прежде чем мы обсудим, как работать с исключениями и учитывать сложные случаи, познакомимся еще с одним методом утверждения, который может предложить QUnit, — `strictEqual()`.

Метод `strictEqual()` похож на `equal()`, за исключением того, что осуществляет строгое сравнение фактического и ожидаемого результатов. Его синтаксис показан ниже.

Синтаксис метода: strictEqual

`strictEqual(value, expected[, message])`

Проверяет эквивалентность параметра `value` параметру `expected`, используя строгое сравнение (`===`).

Параметры

`value`	(Любое) Значение, возвращаемое функцией или методом или хранимое в переменной, которое нужно проверить.
`expected`	(Любое) Значение, с которым проводится сравнение.
`message`	(Строка) Необязательное описание утверждения. Если оно пропущено, то показывается сообщение okay в случае успешного и failed в случае неудачного прохождения теста.

Возвращает

`undefined`.

Ради использования `strictEqual()` с простой функцией `sum()`, которую вы создали, заменим предыдущее утверждение на то, где вы сравниваете сумму двух чисел, логическим значением `false` (можете поменять и все остальные ваши утверждения на `strictEqual()`):

```
assert.strictEqual(sum(-2, 2), false, 'Сумма отрицательного и положительного
                  чисел равна false');
```

Обновление HTML-страницы покажет ошибку. Данный тест может обнаружить неравенство фактического и ожидаемого значения. Но при использовании этого вида утверждений (что не будет полезно для тестирования функций) ваш тест завершится неудачей, а это не то, что вы хотите. Нужно проверить, не равны ли оба числа логическому значению (`false` в данном случае), и если это так, то утверждение пройдет успешно.

Для ситуаций, когда вы хотите утвердить, что значение не равно или строго не равно другому, можно использовать дополняющие методы: `notEqual()` и `notStrictEqual()`.

Обновите предыдущее утверждение, чтобы увидеть успешность теста:

```
assert.notStrictEqual(sum(-2, 2), false, 'Сумма отрицательного
                        и положительного чисел не равна false');
```

На этот раз обновление HTML-страницы корректно выполнит все четыре утверждения и тест закончится успешно.

Полная и обновленная версия этой страницы показана в листинге 14.2. Он доступен в файле `chapter-14/test.2.html`, предоставленном с книгой, и в JS Bin (http://jsbin.com/suroya/edit?html,output).

Листинг 14.2. Использование strictEqual() и notStrictEqual()

```
<!DOCTYPE html>
<html>
   <head>
      <meta charset="utf-8" />
      <title>QUnit - Тест 2</title>
      <link rel="stylesheet"
            href="../css/qunit-1.18.0.css" />
   </head>
   <body>
      <div id="qunit"></div>
      <div id="qunit-fixture"></div>
      <script>
         function sum(a, b) {
            return a + b;
         }
      </script>
      <script src="../js/qunit-1.18.0.js"></script>
      <script>
```

Таблица стилей CSS QUnit

Функция для проверки

Файл JavaScript QUnit

```
        QUnit.test('Мой первый тест', function(assert) {
           assert.expect(4);
           assert.strictEqual(
              sum(2, 2),
              4,
              'Сумма двух положительных чисел'
           );
           assert.strictEqual(
              sum(-2, -2),
              -4,
              'Сумма двух отрицательных чисел'
           );
           assert.strictEqual(
              sum(2, 0),
              2,
              'Сумма положительного числа
               и нейтрального элемента'
           );
           assert.notStrictEqual(
              sum(-2, 2),
              false,
              'Сумма отрицательного и положительного
               чисел не равна false'
           );
        });
     </script>
     </body>
</html>
```

Код для тестирования функции sum()

До настоящего момента мы обсуждали, как тестировать числа в JavaScript. Но изученные вами методы можно задействовать для тестирования других типов данных, таких как `Array`, `Object`, `String` и т. д. Кроме того, хотя QUnit полезен для тестирования любого кода JavaScript, эта книга по-прежнему о jQuery. Так что посмотрим некоторые примеры применения всех описанных выше методов применительно к коду, написанному с использованием методов и вспомогательных функций jQuery.

```
assert.equal($.trim(' '), '',
  'Обрезка строки, содержащей только пробелы, возвращает пустую строку');
assert.strictEqual($('input:checked').length,
  $('input').filter(':checked').length,
  'Фильтрация элементов до и после дает такое же количество элементов');
assert.notEqual($('input:checked'), $('input').filter(':checked'),
  'Два объекта jQuery различны, если они не указывают на один
   и тот же адрес памяти');
assert.notStrictEqual(new Array(1, 2, 3), [1, 2, 3],
  'Два массива различны, если они не указывают на один
   и тот же адрес памяти');
```

Как видите, тестирование различных типов данных не отличается от тестирования чисел. Перейдем к другим методам утверждений.

14.4.2. Другие методы утверждений

Другие методы утверждений, предоставляемые QUnit, схожи по области видимости и принимаемым параметрам; так что мы решили для компактности разместить их в табл. 14.1.

Таблица 14.1. Обзор других методов утверждения, предоставляемых QUnit

Метод	Описание
deepEqual(value, expected[, message])	Рекурсивное, строгое сравнение, работающее для всех типов JavaScript. Утверждение проходит успешно, если value и expected идентичны с точки зрения свойств и значений и у value и expected один и тот же прототип
notDeepEqual(value, expected[, message])	То же, что и deepEqual(), но тестируется неравенство хотя бы одного свойства или значения
propEqual(value, expected[, message])	Строгое сравнение свойств и значений объекта. Утверждение проходит успешно, если все свойства и значения равны при строгом сравнении
notPropEqual(value, expected[, message])	То же, что и propEqual(), но тестируется неравенство хотя бы одного свойства или значения
ok(value[, message])	Утверждение, которое проходит, если первый аргумент истинный

Чтобы увидеть эти новые методы в действии, создадим функцию, которая тестирует четность числа:

```
function isEven(number) {
  return number % 2 === 0;
}
```

Для проверки функции `isEven()` подойдет такая запись:

```
assert.strictEqual(isEven(4), true, '4 – четное число');
```

Но, используя метод `ok()`, можно упростить утверждение:

```
assert.ok(isEven(4), '4 – четное число');
```

Намного лучше, правда?

Разница между методами утверждения `deepEqual()` и `propEqual()` незначительная, но важная. Чтобы понять ее, определим объектный литерал, называемый `human`, со свойством `fullName`, инициализированным в `null`. Кроме того, определим функцию `Person`, которая должна использоваться как конструктор, принимающий в качестве единственного параметра строку. Она определяет свойство `fullName` для созданного объекта. Код для создания функции и объектного литерала выглядит следующим образом.

```
function Person(fullName) {
  this.fullName = fullName;
}
var human = {
  fullName: null
};
```

Теперь создайте объект типа `Person`, установив `"John Doe"` как значение `fullName`, а также установите свойство `fullName` объектного литерала `human`. Затем протестируйте их равенство, используя `deepEqual()` и `propEqual()`, чтобы выяснить их разницу. Соответствующий код показан ниже, а также доступен в файле `chapter-14/test.3.html`, предоставленном с книгой, и в JS Bin (http://jsbin.com/tecihe/edit?html,output):

```
QUnit.test('Тестирование propEqual() и deepEqual()', function(assert) {
  assert.expect(2);
  var person = new Person('John Doe');
  human.fullName = 'John Doe';
  assert.propEqual(person, human, 'Тест пройден. Одинаковые свойства
                   и значения');
  assert.deepEqual(person, human, 'Тест не пройден. Одинаковые свойства
                   и значения, но разные прототипы');
});
```

Запустив данный тест, вы увидите, что первое утверждение проходит успешно, а второе — нет. Причина состоит в следующем: у объектов `person` и `human` одни и те же свойства и значения, но различные *прототипы* (у `person` прототип `Person`, а у `human` — `Object`). Этого оказывается достаточно, чтобы утверждение `propEqual()` прошло, а `deepEqual()` — нет.

Осталось обсудить еще один, последний метод утверждения.

14.4.3. Метод утверждений throws()

Мы откладывали обсуждение метода утверждения `throws()`, потому что он немного отличается от остальных. Его синтаксис представлен ниже.

Синтаксис метода: throws

`throws(function[, expected][, message ])`

Проверяет, выбрасывает ли функция обратного вызова исключение, и дополнительно сравнивает выброшенную ошибку.

Параметры

`function`	(Функция) Запускаемая функция.
`expected`	(Объект|Функция|Регулярное выражение) Объект `Error`, или функция `Error` (конструктор), или регулярное выражение `RegExp`, которое соответствует (или частично соответствует) представлению строки, или функция обратного вызова, которая должна вернуть `true` для передачи проверки утверждения.
`message`	(Строка) Необязательное описание утверждения. Если оно пропущено, то показывается сообщение okay в случае успешного и failed — в случае неудачного прохождения теста.

Возвращает

`undefined`.

Данный метод отличается от остальных тем, что принимает в качестве своего первого аргумента для тестирования не значение, а функцию, которая должна выдать ошибку, так как ожидаемое значение не является обязательным. Посмотрим на это в действии, изменив функцию `isEven()` так, чтобы она выдавала ошибку, если переданный параметр не типа `Number`:

```
function isEven(number) {
  if (typeof number !== 'number') {
    throw new Error('Передаваемый аргумент не является числом');
  }
  return number % 2 === 0;
}
```

Чтобы проверить выданную ошибку, можно написать:

```
assert.throws(
  function() {
    isEven('test');
  },
  new Error('Передаваемый аргумент не является числом'),
  'При передаче строки выбрасывается ошибка'
);
```

В данном случае вы передаете ожидаемое значение в форму того же экземпляра `Error`, который должен быть выброшен. В качестве альтернативы можно проверить, совпадает ли сообщение об ошибке с ожидаемым вами, изменив предыдущее утверждение на то, которое будет использовать регулярное выражение:

```
assert.throws(
  function() {
    isEven('test');
  },
  /Переданный аргумент не является числом/,
  'При передаче строки выбрасывается ошибка'
);
```

Методом `throws()` мы завершили обзор методов утверждений, предоставляемых QUnit для тестирования синхронного кода. Для тестирования асинхронного кода, например функций обратного вызова, передаваемых функциям jQuery Ajax, нужно изучить дополнительный метод. Обсудим это.

14.5. Как тестировать асинхронные задачи

Иногда нужно выполнить определенное действие или повторять его снова и снова спустя определенное время. В других случаях вы захотите получать информацию от сервера без обновления страницы. Это ситуации, когда вам может понадобиться асинхронное выполнение одной или нескольких функций.

Для тестирования асинхронных функций можно использовать метод для создания теста и методы утверждения, описанные выше. Но также вам понадобится

механизм информирования модуля, запустившего тест, что вы ждете завершения работы асинхронного метода. Посмотрим на метод `async()`. Он принадлежит тому же параметру `assert`, который уже упоминался в этой главе, и его синтаксис показан ниже.

Синтаксис метода: async

`async()`

Инструктирует QUnit ждать завершения асинхронной операции.

Параметры

Отсутствуют.

Возвращает

Уникальную функцию разрешения обратного вызова.

Как видите, данный метод не принимает никаких параметров и возвращает функцию. Эта функция уникальна, должна быть использована только один раз и вызвана внутри асинхронной функции, которую вы хотите протестировать.

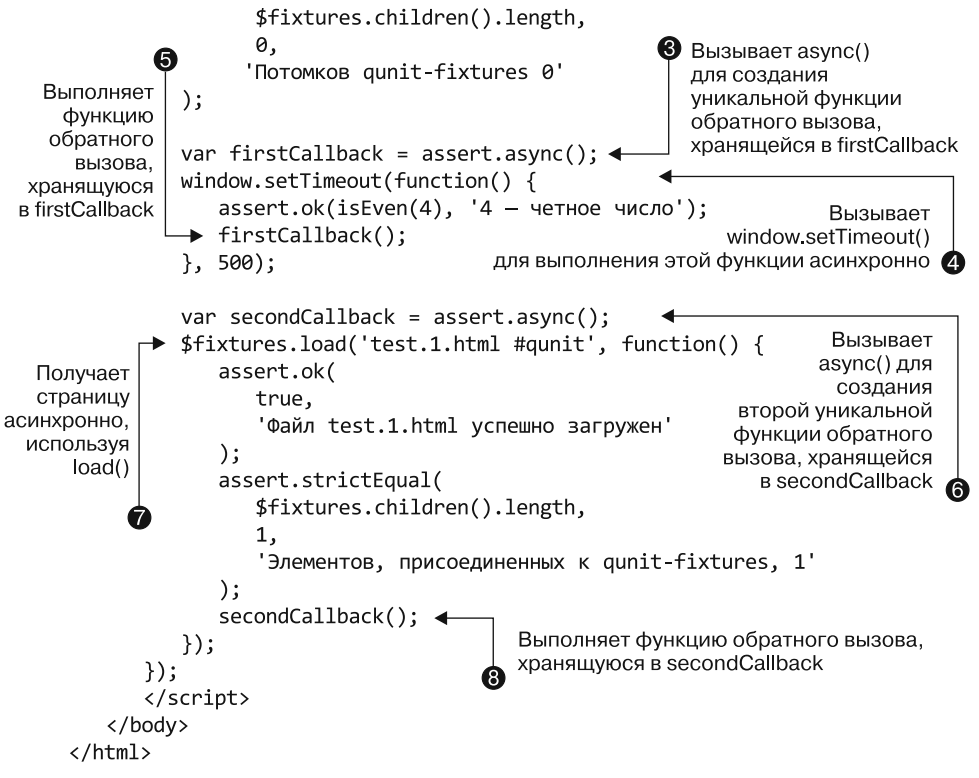
Чтобы понять лучше, как работает метод `async()`, проанализируем код, показанный в листинге 14.3, также доступном в файле `chapter-14/asynchronous.test.html`, предоставленном с книгой.

Листинг 14.3. Асинхронный тест с QUnit

```
<!DOCTYPE html>
<html>
   <head>
      <meta charset="utf-8" />
      <title>QUnit – Асинхронный тест</title>
      <link rel="stylesheet" href="../css/qunit-1.18.0.css" />
   </head>
   <body>
      <div id="qunit"></div>
      <div id="qunit-fixture"></div>

      <script src="../js/jquery-1.11.3.min.js"></script>
      <script>
         function isEven(number) {
            return number % 2 === 0;
         }
      </script>
      <script src="../js/qunit-1.18.0.js"></script>
      <script>
         QUnit.test('Тестирование асинхронного кода',
         function(assert) {
            var $fixtures = $('#qunit-fixture');
            assert.expect(4);

            assert.strictEqual(
```

❶ Определяет функцию isEven()

❷ Получает элемент с ID qunit-feature



В этом коде вы создаете разметку для использования QUnit и добавляете jQuery. Определяете также функцию `isEven()`, которая будет проверять четность числа ❶. Затем, после включения модуля запуска теста QUnit, создаете асинхронный тест, применяя обычный метод `QUnit.test()`.

Внутри теста вы получаете элемент с ID `qunit-feature` ❷, который задействуете для введения в отдельные элементы позже в этом коде. Затем устанавливаете количество утверждений, которые запустите, и создаете первое утверждение.

Затем вы вызываете метод `assert.async()` для создания первой уникальной функции обратного вызова, хранящейся в переменной `firstCallback` ❸. Первая асинхронная операция, которую вы выполняете, — запуск функции с 500-миллисекундной задержкой, используя метод `window.setTimeout()` ❹. Внутри определенной функции обратного вызова тестируете четность числа 4 и выполняете функцию обратного вызова, хранящуюся в `firstCallback` ❺. Выполнение данной функции имеет ключевое значение, поскольку, если этого не сделать, модуль запуска теста будет неопределенное время ждать вызова функции, блокируя тем самым прохождение всех остальных тестов.

В последующей части кода вы вновь запускаете `async()` для создания второй уникальной функции обратного вызова ❻. Она хранится в переменной `secondCallback`. Вторая асинхронная операция кода использует метод jQuery `load()`. Вы пытаетесь получить файл `chapter-14/test.1.html` и затем ввести только

элемент с ID qunit (7). Внутри функции обратного вызова, переданной load(), проверяете, загружен ли файл корректно, с помощью метода assert.ok() и того единственного элемента, который был вставлен в элемент с ID qunit-fixture. После завершения выполняете функцию обратного вызова, хранящуюся в secondCallback (8).

Запуск асинхронных тестов прост, как видно в этом примере. Единственный аспект, который нужно помнить, — вызов функций обратного вызова с использованием метода async().

Иногда при использовании функций Ajax в jQuery вам придется загружать ресурсы с сервера. Возможна ситуация, что вы будете разрабатывать код JavaScript до того, как написан код серверной стороны. Или вы не захотите полагаться на корректность работы такого кода, чтобы убедиться, что ваш код для клиентской стороны работает ожидаемо. В таких случаях можно создавать фиктивные Ajax-запросы, используя jQuery Mockjax (https://github.com/jakerella/jquery-mockjax) или Sinon.js (http://sinonjs.org/).

Ранее в этой главе мы упоминали три флажка на пользовательской стороне, предоставленные QUnit при запуске тестов. Поговорим чуть больше о двух из них, меняющих поведение QUnit.

14.6. noglobals и notrycatch

QUnit предлагает два флажка: noglobals (обозначенный как Check for Globals) и notrycatch (No try-catch), с помощью которых вы можете устанавливать или снимать отметки, если нужно изменить поведение всех тестов, выполняемых на странице.

Флаг noglobals приведет тест к неудачному завершению, если в запускаемом коде появится новая глобальная переменная (по сути, то же, что и добавление свойства к объекту window). Следующий пример показывает тест, который завершится неудачно при установленном флажке noglobals:

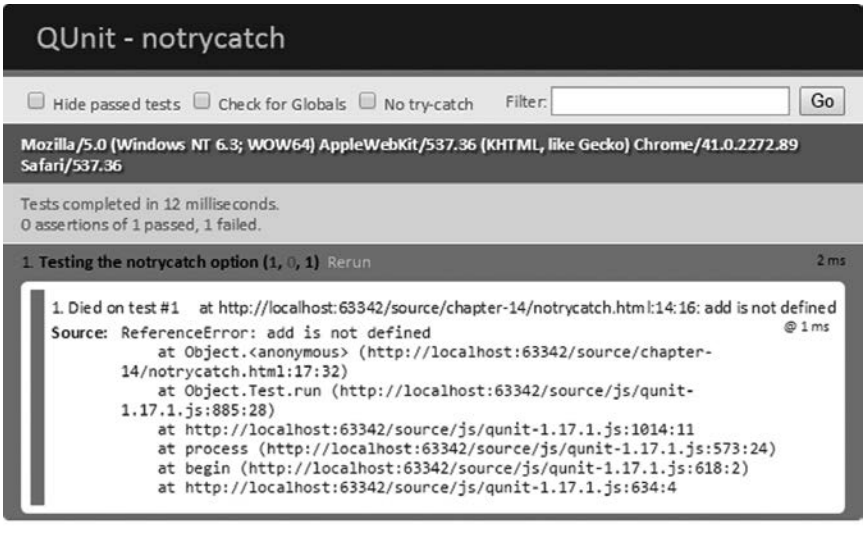
```
QUnit.test('Тестирование параметра noglobals', function(assert) {
  assert.expect(1);
  window.bookName = 'jQuery в действии';
  assert.strictEqual(bookName, 'jQuery в действии', 'Строки равны');
});
```

Причина неудачного завершения теста в том, что код добавил свойство bookName к объекту window. Этот тест не завершился бы неудачно, если бы код только изменил существующее свойство объекта window, такое как name.

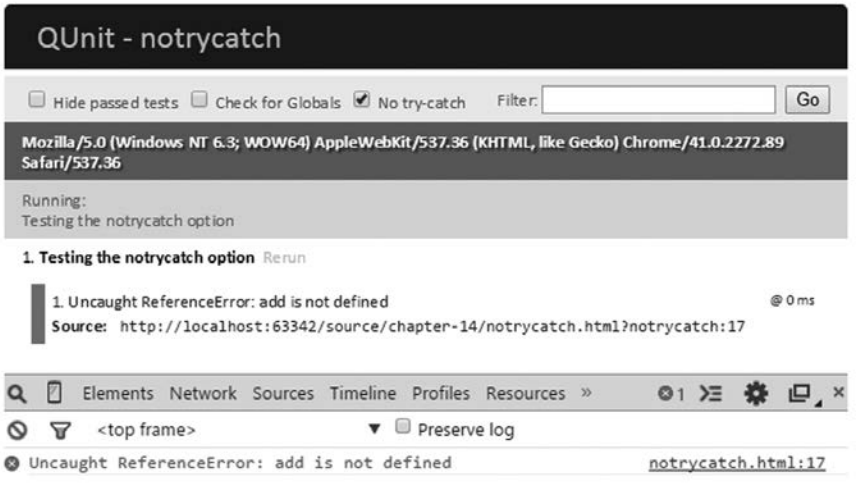
Флаг notrycatch разрешает запускать QUnit без заключения в блок try-catch. Это не даст QUnit поймать ошибку и вывести ее, но часто помогает при более глубокой отладке проблемы. Следующий пример, который вы можете найти в файле chapter-14/notrycatch.html, предоставленном с книгой, показывает этот параметр в действии:

```
QUnit.test('Тестирование параметра notrycatch', function(assert) {
  assert.expect(1);
  assert.strictEqual(add(2, 2), 4, 'Сумма 2 и 2 равна 4');
});
```

В этом коде функция `add()` не определена, так что вызов ее приведет к ошибке (`ReferenceError`, если сказать конкретнее). Если вы запустите код с установленным флажком `notrycatch`, то фреймворк не поймает ошибку и не выведет ее в журнал консоли. Разница показана на рис. 14.3, *а* и *б*.



а



б

Рис. 14.3. Запуск теста: *а* — без флага notrycatch; *б* — с активизированным флагом notrycatch

При работе с крупными приложениями нужно делать тесты логически структурированными, чтобы иметь возможность выполнить конкретную группу тестов без запуска полного комплекта тестов. Здесь в игру вступают модули.

14.7. Группировка тестов в модулях

Когда плагин или библиотека увеличивается в размерах, для улучшения поддерживаемости можно разделить исходный код на модули. Подобный принцип вы видели, когда мы обсуждали структуру jQuery, чей код состоит из более чем 10 000 строк. Аналогичный принцип применим и к коду, который вы пишете для тестирования проекта.

У QUnit есть простой метод для группировки тестов в модули, называемый `QUnit.module()`. Его синтаксис следующий.

Синтаксис метода: QUnit.module

`QUnit.module(name[, lifecycle])`

Группирует набор соответствующих тестов в отдельный модуль.

Параметры

`name`	(Строка) Имя для идентификации модуля.
`lifecycle`	(Объект) Объект, содержащий две необязательные функции для запуска перед (`beforeEach`) и после (`afterEach`) каждого теста. Каждая из этих функций получает аргумент, названный `assert`, который предоставляет все методы утверждения QUnit.

Возвращает

`undefined`.

Посмотрев на сигнатуру этого метода, вы можете догадаться, как определить принадлежность тестов данному модулю. Ответ заключается в том, что тесты, принадлежащие модулю, — те, которые определены после вызова `QUnit.module()`, но до того, как будет найден другой вызов `QUnit.module()` (если таковой есть). Следующий код прояснит ситуацию:

```
QUnit.module('Core');
QUnit.test('Первый тест', function(assert) {
  assert.expect(1);
  assert.ok(true);
});
QUnit.module('Ajax');
QUnit.test('Второй тест', function(assert) {
  assert.expect(1);
  assert.ok(true);
});
```

В данном фрагменте мы выделили жирным шрифтом вызовы `QUnit.module()`. В этом случае тест, отмеченный как «Первый тест», принадлежит модулю Core, а тест, отмеченный как «Второй тест», — модулю Ajax.

Если выполнить предыдущий код, то вы увидите, что именам тестов предшествуют имена модулей этих тестов. Кроме того, можно задать определенное имя модуля, чтобы выбрать тесты для запуска из выпадающего меню в правом верхнем углу страницы. Описанные данные представлены на рис. 14.4.

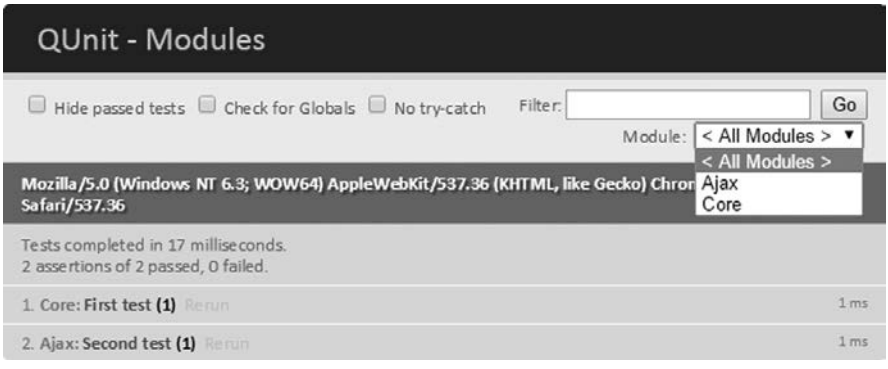


Рис. 14.4. Выполнение тестового комплекта, упорядоченного в модули

Теперь, когда вы знаете, как правильно организовать тестовый комплект, с удобством для дальнейшей поддержки, пришло время изучить определенные свойства конфигурации фреймворка QUnit.

14.8. Конфигурирование QUnit

Подобно тому как у jQuery есть большое количество разумных значений по умолчанию для многих методов, у QUnit есть предустановленная конфигурация. Иногда вам может понадобиться настроить эту конфигурацию для нужд проекта.

Фреймворк позволяет переопределять эти значения по умолчанию, разместив их в свойстве, названном `config`. В табл. 14.2 показаны все свойства, доступные через объект `QUnit.config`.

Таблица 14.2. Свойства конфигурации QUnit.config

Имя	Описание
altertitle	(Логический тип) QUnit меняет document.title, добавляя галочку или крестик, чтобы показать, успешно или неудачно завершен тестовый комплект. Установка этого свойства в false (значение по умолчанию — true) отключает это. Может быть полезным, если ваш код работает с document.title
autostart	(Логический тип) QUnit запускает тесты, когда событие load запускается в window. Если вы загружаете тесты в асинхронном режиме, то можете установить значение в false (по умолчанию это true), и затем вызывать QUnit.start(), когда все загружено
hidepassed	(Логический тип) QUnit показывает все тесты, включая те, которые прошли успешно. Установив этому свойству значение true, вы будете видеть только те тесты, которые завершились неудачно
moduleFilter	(Строка) Определяет отдельный модуль для запуска с помощью указания его имени. Значение по умолчанию — undefined, в этом случае QUnit запускает все загруженные модули

Продолжение ⇩

Таблица 14.2 (продолжение)

Имя	Описание
reorder	(Логический тип) Фреймворк сначала запускает те тесты, которые завершились неудачно при предыдущем запуске. Если вы хотите изменить это поведение, то установите значение false
requireExpects	(Логический тип) Установите этому свойству true, если хотите заставить использовать метод assert.expect()
testId	(Массив) Это свойство позволяет QUnit запускать определенные блоки тестов с помощью хеша строки, состоящей из имени модуля и имени теста. Значение по умолчанию — undefined
testTimeout	(Число) Устанавливает максимальное время выполнения, после которого все тесты завершатся неудачно. Значение по умолчанию — undefined, означающее, что лимита по времени нет
scrolltop	(Логический тип) Установите этому свойству false, если хотите предотвратить с помощью QUnit возвращение в верхнюю часть страницы после выполнения всех тестов
urlConfig	(Массив) Управляет элементами панели инструментов QUnit (та панель, где находятся флаги noglobals и notrycatch). Расширив этот массив, можно добавлять дополнительные флажки и списки

Пользовательская конфигурация тестового комплекта должна быть размещена после JavaScript-файла QUnit. Можно определить конфигурацию во внешнем файле, как показано в следующем коде, или встраивать его.

```
<script src="qunit-1.18.0.js"></script>
<script src="qunit-config.js"></script>
```

Если у вас есть огромный набор тестов, то можете спрятать успешно пройденные тесты, чтобы обращать внимание только на те, которые прошли неудачно. Для этого используйте свойство `hidepassed`, описанное в табл. 14.2, как показано здесь:

```
QUnit.config.hidepassed = true;
```

Если хотите заставлять себя и свою команду указывать количество утверждений, то можете применять свойство `requireExpects`:

```
QUnit.config.requireExpects = true;
```

QUnit определяет и другие методы, которых здесь нет и которые мы не будем рассматривать в этой главе, так как они второстепенны. Темы, охваченные на данный момент, позволят создать полный набор тестов, достаточный в большинстве случаев.

В следующем разделе вы создадите такой набор, чтобы применить все полученные знания о QUnit в действии. Будет интересно!

14.9. Пример набора тестов

Здесь вы создадите полный набор тестов для проекта, а что может быть лучше тестирования чего-то, что вы разработали в этой самой книге? Вот потому вы и создадите тестовый комплект для Jqia Context Menu, плагина jQuery, который вы создали в главе 12, показывающего пользовательское контекстное меню на одном или нескольких определенных элементах страницы.

Первый шаг, нужный для создания комплекта, — создание новой страницы с необходимой установкой для QUnit, где также нужно добавить библиотеку QUnit и файлы, относящиеся к Jqia Context Menu (`jquery.jqia.contextMenu.css` и `jquery.jqia.contextMenu.js`). Затем разместите ненумерованный список, который будет выступать в качестве пользовательского меню внутри элемента с ID `qunit-fixture`. Вы также будете применять этот элемент как тот, где вы будете отображать пользовательское меню.

Вы можете заставить себя и других разработчиков, принимающих участие в написании тестового набора, всегда использовать метод `assert.expect()`. Чтобы это сделать, нужно изменить значение по умолчанию свойства конфигурации QUnit `requireExpects`, чтобы QUnit выдавал ошибку, если данный метод не вызывается.

Плагин Jqia Context Menu состоит из одного JavaScript-файла, так что в идеале не нужно группировать тесты в модули. Но следует убрать все данные, прикрепляемые к элементам с ID `qunit-fixture` каждый раз, когда тест завершается. Таким образом, нужно использовать `afterEach` следующим образом:

```
QUnit.module('Core', {
  afterEach: function() {
    $('#qunit-fixture').removeData();
  }
});
```

После объявления модуля нужно определить тесты. В итоге вы создадите пять тестов, разделив их так, как описано ниже.

- *Основные требования* — содержит утверждения, которые проверяют успешность загрузки jQuery и плагина Jqia Context Menu. Кроме того, проверяется, корректно ли плагин выставляет значения по умолчанию, чтобы можно было их настроить.
- *Неверные параметры* — определяет утверждения для проверки, может ли расширение работать с непредусмотренными типами параметров или пропущенными обязательными свойствами (как `idMenu`).
- *Инициализации* — указывает утверждения для проверки, обрывает ли плагин цепочку и не вызывает ли он ошибки, когда передаются правильные параметры. Кроме того, тестирует, не может ли плагин быть инициализирован два раза или более на том же элементе.

- *Функции обратного вызова* — содержит утверждения для проверки, отображается или скрывается меню на основании запущенных событий и какой элемент запустил событие.
- *Разрушение* — проверяет, сохранилась ли цепочка и корректно ли метод удаляет все данные, прикрепленные к элементу. Кроме того, проверяет, скрывается ли меню при отмене эффекта плагина.

Код описанных тестов — в листинге 14.4.

Листинг 14.4. Полный тестовый комплект для Jqia Context Menu

```
QUnit.test('Основные требования', function(assert) {
  assert.expect(4);

  assert.ok($, 'jQuery загружен');
  assert.ok($.fn.jqiaContextMenu,
            'Расширение загружено корректно');
  assert.ok($.fn.jqiaContextMenu.defaults,
            'Значения по умолчанию выставлены');
  assert.propEqual(
    $.fn.jqiaContextMenu.defaults,
    {
       idMenu: null,
       bindLeftClick: false
    },
    'Выставленные значения по умолчанию корректны'
  );
});

QUnit.test('Неверные параметры', function(assert) {
  assert.expect(6);
  var $fixture = $('#qunit-fixture');

  assert.throws(
     function() {
        $fixture.jqiaContextMenu('нет метода');
     },
     /Method .*? не существует/,
     'Вызов неопределенного метода'
  );

  assert.throws(
     function() {
        $fixture.jqiaContextMenu(100);
     },
     /Method .*? не существует/,
     'Неправильный тип аргумента: number'
  );

  assert.throws(
```

Создает тест для проверки основных требований

Тесты, с которыми будет работать расширение, передают ему неправильные параметры

```
    function() {
      $fixture.jqiaContextMenu(null);
    },
    /Method .*? does not exist/,
    'Неправильный тип аргумента: null'
  );

  assert.throws(
    function() {
      $fixture.jqiaContextMenu([]);
    },
    /Method .*? does not exist/,
    'Неправильный тип аргумента: array'
  );

  assert.throws(
    function() {
      $fixture.jqiaContextMenu({});
    },
    /Меню не указано/,
    'Меню не указано'
  );

  assert.throws(
    function() {
      $fixture.jqiaContextMenu({idMenu: неизвестный id'});
    },
    /Указанное меню не существует/,
    'Неизвестное меню'
  );
});
QUnit.test('Initialization', function(assert) {
  assert.expect(5);
  var $fixture = $('#qunit-fixture');
  var $fixtureInitialized = $fixture.jqiaContextMenu({
    idMenu: 'context-menu'
  });

  assert.ok($fixtureInitialized, 'Меню инициализировано');
  assert.notEqual(
    $fixtureInitialized.data('jqiaContextMenu'),
    {},
    'Используется правильное пространство имен'
  );
  assert.strictEqual(
    $fixture.length,
    $fixtureInitialized.length,
    'Сохранение цепочки'
  );
  assert.strictEqual(
```

Проверяет, сохраняет ли jqiaContextMenu() цепочку и устанавливает ли атрибуты data-* и т. д. при передаче корректного параметра

```
      $fixture,
      $fixtureInitialized,
      'Возврат того же объекта'
   );
   assert.throws(
      function() {
         $fixture.jqiaContextMenu({idMenu: 'context-menu'});
      },
      /Расширение уже было инициализировано/,
      'Расширение уже проинициализировано на элементе'
   );
});

QUnit.test('Callbacks', function(assert) {
   assert.expect(3);
   var $fixture = $('#qunit-fixture').jqiaContextMenu({
      idMenu: 'context-menu'
   });
   var $menu = $('#context-menu');

   assert.strictEqual(
      $menu.css('display'),
      'none',
      'Меню скрыто'
   );
   $fixture.trigger('contextmenu');
   assert.strictEqual(
      $menu.css('display'),
      'block',
      'Меню отображается после нажатия'
   );
   $('html').click();
   assert.strictEqual(
      $menu.css('display'),
      'none',
      'Меню скрыто после нажатия других элементов'
   );
});

QUnit.test('Destroy', function(assert) {
   assert.expect(5);
   var $fixture = $('#qunit-fixture').jqiaContextMenu({
      idMenu: 'context-menu'
   });
   var $fixtureDestroyed = $fixture.jqiaContextMenu('destroy');
   var $menu = $('#context-menu');

   assert.strictEqual(
      $fixture.length,
      $fixtureDestroyed.length,
```

Проверяет, показывается/скрывается ли меню после запуска события (щелчка на целевых элементах или вне их)

Проверяет, сохраняет ли destroy() цепочку, удаляет ли атрибуты data-* и т. д.

```
    'Keep chainability'
  );
  assert.strictEqual(
    $fixture,
    $fixtureDestroyed,
    'Возврат того же объекта'
  );
  assert.strictEqual(
    $menu.css('display'),
    'none',
    'Меню скрыто'
  );
  assert.strictEqual(
    $fixture.data('jqiaContextMenu'),
    undefined,
    'Данные пространства имен очищены'
  );
  $fixture.trigger('contextmenu');
  assert.strictEqual(
    $menu.css('display'),
    'none',
    'Меню по-прежнему скрыто после нажатия'
  );
});
```

Если хотите запустить этот тестовый набор и поэкспериментировать с ним немного (например, добавив еще несколько нужных тестов), то можете найти его в файле `chapter-14/test.suite.html`, предоставленном с книгой.

Этот набор создан так, что все тесты пройдут успешно, но для лучшего понимания кода, написанного для тестирования плагина, вы можете захотеть посмотреть на то, как некоторые тесты пройдут неудачно. Для этого можно попробовать поменять код плагина неожиданным образом. Если нужна подсказка — попробуйте убрать инструкцию `return this;` в методах `init()` и `destroy()`. Таким образом вы разрушите возможность создания цепочки в плагине и соответственные утверждения завершатся неудачно.

Последний демонстрационный пример показал не только как применять большинство методов, описанных в данной главе, но и то, как выглядит полный комплект тестов. Надеемся, вы последуете нашему искреннему совету и начнете тестировать свой код чаще, если еще не делаете этого.

14.10. Резюме

В данной главе мы описали фундаментальные понятия тестирования ПО и то, почему модульное тестирование вашего кода так важно. Тестирование позволяет быть уверенными, что код работает корректно и (практически) не содержит ошибок.

Мы предоставили обзор фреймворков, доступных для модульного тестирования проектов на JavaScript, обращая особое внимание на QUnit. Этот фреймворк,

поддерживаемый той же командой, которая предложила любимую библиотеку jQuery, предоставляет простой в использовании набор методов для тестирования вашего кода.

После описания способов создания тестов с помощью `QUnit.test()` мы познакомили вас с несколькими методами утверждения, используемыми для проверки того, соответствуют ли возвращаемые значения ваших функций и методов ожидаемым. Вдобавок вы изучили важность установки количества утверждений, которые предполагаете запустить, задействуя метод `assert.expect()`.

Затем вы узнали, как тестировать функции, работающие асинхронно, с помощью метода `assert.async()`.

Наконец, вы изучили, как организовать набор тестов в модули и установить нужную для проекта конфигурацию. Обладая этими знаниями, вы создали полный и работающий тестовый комплект. Надеемся, что начиная с завтрашнего или даже сегодняшнего дня вы станете тестировать свой код, получив возможность использовать и менять его с большей уверенностью.

В следующей, и последней, главе этой книги вы откроете для себя ряд полезных инструментов, которые помогут применять jQuery в больших проектах.

15 jQuery в больших проектах

В этой главе:

- оптимизация селекторов для повышения производительности;
- разделение кода на модули;
- загрузка модулей через RequireJS;
- управление зависимостями с помощью Bower;
- создание SPA с помощью Backbone.js.

Мы надеемся, что, прочитав все предыдущие главы, вы научились писать красивый и лаконичный код, используя jQuery. Вы также умеете создавать плагины на базе нашей библиотеки и знаете, как тестировать код с помощью модульных тестов. Теперь, когда вы хорошо изучили jQuery, пора узнать, в каких случаях этого знания недостаточно и когда нужно применять и другие библиотеки или даже фреймворки.

В последней главе этой книги мы раздвинем горизонты и обратим внимание еще на несколько инструментов, фреймворков и шаблонов. Они не связаны напрямую с jQuery, но могут быть использованы для создания быстрого, надежного и красивого кода.

Главное назначение нашей библиотеки — помочь в манипуляциях с DOM. Операции с деревом DOM обычно медленные, поэтому нужно понимать, как повысить производительность кода jQuery, чтобы операции выполнялись с максимально возможной скоростью. Вам также следует научиться легко интегрировать jQuery в крупные проекты и правильно структурировать код, разделяя его на модули для удобного обслуживания впоследствии.

Одной из наиболее важных задач, которую приходится решать разработчикам, является создание высокопроизводительного кода. Эту проблему многие недооценивают, но оптимизация кода JavaScript всегда стоит потраченного на нее времени. В некоторых ситуациях улучшить производительность мешает плохо написанный код, который нуждается в глубоком рефакторинге, но иногда это так же просто, как верный выбор элементов. В первом разделе мы подробно обсудим, как можно повысить производительность кода, написанного с использованием jQuery, правильно выбирая элементы.

При работе над крупными проектами необходима более эффективная организация кода. Если не управлять им должным образом, то по прошествии времени в результате добавления новых функций и рефакторинга образуется полная неразбериха. Один из способов решить эту проблему — использовать несколько известных и надежных шаблонов. Практика разделения кода на модули, которая будет следующей темой этой главы, позволяет лучше представлять себе проект в целом, а также легко разделять его на части, чтобы каждый программист мог работать со своим модулем.

Разделение проекта на модули — хороший способ содержать проект в порядке, но он связан с одной проблемой. Работа каждого модуля может зависеть от других, поэтому нужно внимательно следить за последовательностью подключения модулей на странице. Пока их мало, ими легко управлять, но при разработке больших проектов задача усложняется. Когда задействовано много модулей, плагинов, библиотек и фреймворков, у каждого из которых есть свои зависимости, необходим профессиональный и надежный метод подключения их к проекту в правильной последовательности. Одно из возможных решений этой задачи — использование RequireJS, библиотеки, с которой мы познакомимся в разделе 15.3.

В предыдущем абзаце мы упомянули плагины, библиотеки и фреймворки. Написание кода с нуля требует слишком много времени, поэтому обычно при разработке проекта опираются на программные продукты сторонних производителей, такие как jQuery и Modernizr. Чтобы включить указанные компоненты в проект, вы обычно посещаете соответствующие сайты, загружаете оттуда требуемые файлы и помещаете их в папку проекта. Данный процесс медленный и скучный, хотя и работающий. Вдобавок на вас ложится ответственность вручную проверять и устанавливать новые версии. Для автоматизации этой процедуры можно воспользоваться менеджером пакетов Bower, о котором мы расскажем в разделе 15.4.

Наконец, в последнем разделе этой главы мы познакомимся с Backbone.js. Мы не претендуем на исчерпывающее руководство к данному фреймворку, но хотим дать представление о том, куда двигаться дальше по пути обучения и как интегрировать jQuery с такими фреймворками, как Backbone.js, при создании сложных приложений.

15.1. Повышение производительности селекторов

Достижение высокой производительности — огромная проблема, и сегодня больше, чем когда-либо. Об этом следует заботиться при разработке любого веб-проекта с самого начала во избежание появления страниц, загружающихся по десять секунд.

Быстрый код нужен не только для того, чтобы хвастаться перед друзьями; он привлечет к вам довольных пользователей. В данном разделе мы изучим некоторые приемы и хитрости, позволяющие оптимизировать код, правильно выбрав элементы с помощью jQuery.

15.1.1. Избегайте универсального селектора

Первый и простейший совет по улучшению производительности: используйте универсальный селектор только там, где он совершенно необходим. Если в коде встречается такой селектор:

```
$('form :checkbox');
```

то он эквивалентен этому:

```
$('form *:checkbox');
```

Как вы, возможно, помните, если селектор перед фильтром не указан, то неявно применяется универсальный селектор. Чтобы повысить производительность предыдущего селектора, можно преобразовать его в следующий:

```
$('form input:checkbox');
```

А еще лучше, помня о параметре `context`, сделать так:

```
$('input:checkbox', 'form');
```

Последний вариант в большинстве случаев работает быстрее, чем представленный в начале этого раздела.

Стоит избегать универсального селектора при выборе всех прямых потомков элемента. Простое решение данной задачи с применением универсального селектора выглядит так:

```
$('form > *');
```

Но можно сделать лучше! Более разумный подход — использовать в селекторе имя тега, а потом применить метод jQuery `children()`:

```
$('form').children();
```

Это решение лучше, поскольку позволяет jQuery вызвать нативную — и очень быструю — функцию JavaScript `getElementsByTagName()`.

Принцип, показанный в последнем примере, можно применять и в других случаях. Помните: наилучшая производительность достигается при наличии возможности вызвать из jQuery собственные функции JavaScript, такие как `getElementById()` (самая быстрая функция из себе подобных), `getElementsByTagName()` и т. п.

15.1.2. Дополняйте селектор Class

Изучив эту книгу, вы узнали, что jQuery по возможности использует собственные функции JavaScript для ускорения выполнения операций.

Чтобы выбрать элементы по имени класса, библиотека незаметно для пользователя использует функцию `getElementsByClassName()` тех браузеров, которые ее поддерживают, — IE9+, Firefox 3+, Chrome, Safari, Opera 9.5+ и многих других. В Internet Explorer до версии 9 jQuery тоже обеспечивает ожидаемый результат,

полагаясь на собственную реализацию данной функции. Поэтому при содержании на странице большого количества элементов их выбор может быть медленным.

Если вы хотите повысить производительность для Internet Explorer версий 6–8 — например, если ориентируетесь именно на эти браузеры (некоторые организации как будто застыли в прошлом), — то можете оптимизировать поиск элементов путем объединения селектора для класса с селектором для элемента. А именно, можно поставить имя элемента перед именем интересующего вас класса.

Например, если нужно выбрать все элементы `p` класса `description` и сохранить их в переменной, то можно написать такой код:

```
var $elements = $('p.description');
```

Это хорошее начало повышения производительности кода, но можно сделать больше.

15.1.3. Не злоупотребляйте параметром context

В главе 2 был представлен второй параметр метода `jQuery()` под названием `context`. Мы сообщили тогда, что использование данного параметра обычно позволяет повысить производительность селектора, ограничив выбор одним или несколькими поддеревьями DOM, в зависимости от конкретного селектора.

Но иногда применение `context` не повышает производительность. Например, если выбрать элемент по ID, то вы ничего не выиграете, задав `context`. Наоборот, производительность снизится. Поэтому избегайте выражений, подобных представленному ниже:

```
var $element = $('#test', 'div');
```

поскольку их производительность хуже по сравнению с таким:

```
var $element = $('#test');
```

Первое решение медленнее, так как библиотеке приходится сначала отбирать все теги `<div>` (которых, скорее всего, много) и потом проверять их потомков вместо того, чтобы сразу воспользоваться преимуществами функции `getElementById()`. Если хотите узнать, насколько медленно может работать первый селектор, то посмотрите на поразительные результаты теста, представленные на http://jsperf.com/jquery-context-parameter, а также на диаграмме на рис. 15.1.

Данная диаграмма была создана с помощью jsPerf (http://jsperf.com), службы, позволяющей создавать и публиковать сценарии тестирования. Это хорошая альтернатива, если вы не хотите запускать тест на своем компьютере или если намерены поделиться с кем-то результатами.

Из представленного примера следует еще один важный вывод. Чтобы позволить jQuery использовать функцию `getElementById()`, никогда не следует ставить перед ID имя тега, так что избегайте селекторов вроде `$('p#test')`.

Применение параметра `context` часто повышает производительность в тех случаях, когда не указан ID. Но это правило выполняется не всегда. Вообще,

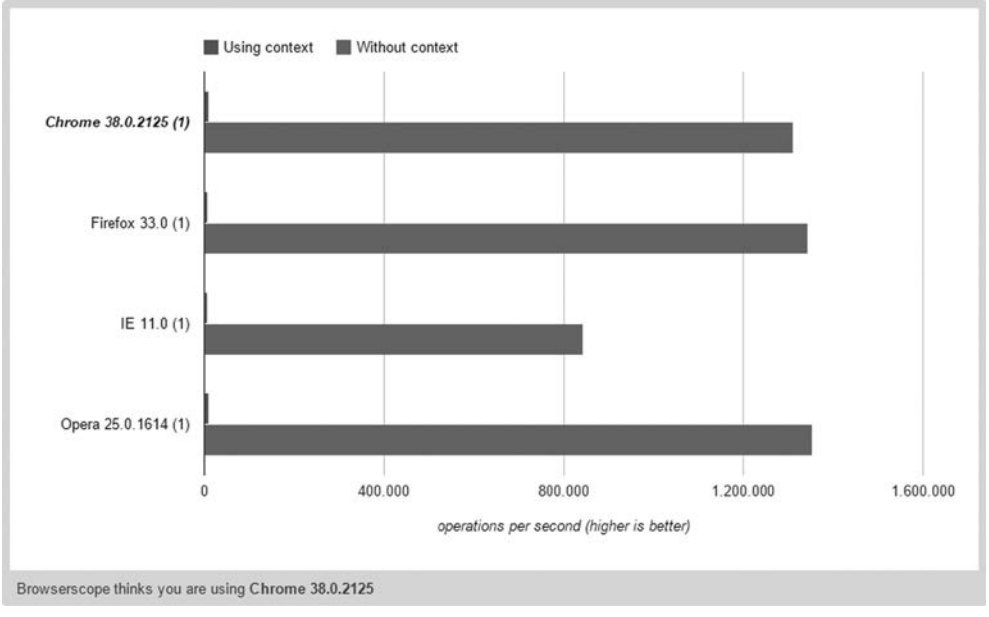


Рис. 15.1. Тест производительности при выборе элемента по его ID с использованием параметра context и без него (чем длиннее полоска, тем выше производительность)

когда речь идет о производительности, ни одно правило не является абсолютным, каждый случай надо тестировать отдельно.

Производительность зависит от многих факторов, таких как количество и тип элементов на странице в браузере. Общее правило следующее: тестировать, тестировать и снова тестировать селекторы, чтобы найти тот, который будет лучше работать в каждом конкретном случае.

Однако это еще не все возможные способы оптимизации. Посмотрим, что можно сделать с помощью фильтров.

15.1.4. Оптимизация с помощью фильтров

Многие фильтры, поддерживаемые jQuery, не являются частью спецификации CSS и поэтому не пользуются преимуществами производительности, предоставляемыми такими методами, как `querySelectorAll()`. Для некоторых, например `:input`, `:visible` и др., лучше вначале выбрать элементы с помощью «чистого» селектора CSS и затем отфильтровать нужные с помощью метода `filter()`. Например, вместо кода:

```
$('p:visible');
```

можно написать:

```
$('p').filter(':visible');
```

Для других фильтров, таких как `:image`, `:password`, `:reset` и т. п., можно воспользоваться преимуществом выбора по атрибуту. Предположим, нам надо выбрать все имеющиеся на странице кнопки сброса. Мы могли бы воспользоваться фильтром `:reset` и написать:

```
$(':reset');
```

но лучше оптимизировать этот селектор, превратив его в такой:

```
$('[type="reset"]');
```

Здесь мы используем универсальный селектор, которого, как мы говорили ранее, следует избегать. Чтобы улучшить данное выражение, можно поставить в начале селектор элемента:

```
$('input[type="reset"]');
```

В нашей книге мы уже встречались с фильтрами позиционирования, такими как `:eq()`, `:lt()` и `:gt()`. Подобно другим описанным здесь фильтрам, они представляют собой плагины jQuery и не поддерживаются CSS. Чтобы повысить производительность, можно вместо использования фильтра `:eq()` сначала выбрать элементы, а затем применить к ним метод `eq()`. Таким образом мы позволим библиотеке задействовать преимущества собственных методов JavaScript. Вместо `:lt()` и `:gt()` можно применить метод `slice()`.

С учетом этих рекомендаций, если мы хотим выбрать первые два элемента списка, вместо такого кода:

```
$('#my-list li:lt(2)');
```

можно написать:

```
$('#my-list li').slice(0, 2);
```

И последняя оптимизация, о которой мы хотим рассказать, касается фильтров `:not()` и `:has()`. В браузерах, поддерживающих метод `querySelectorAll()`, первый из них можно заменить на функцию jQuery `not()`, а второй — на метод jQuery `has()`.

Например, выражение:

```
$('input[placeholder!="Имя"]');
```

можно заменить на:

```
$('input').not('[placeholder="Имя"]');
```

Это был последний совет, касающийся оптимизации фильтров. Но есть еще одна мудрость, которой мы хотели бы поделиться с вами.

15.1.5. Не перегружайте селекторы

jQuery опирается на механизм селекторов, называемый Sizzle, который разбирает селекторы справа налево. Таким образом, чтобы ускорить выбор элементов, следует давать более конкретные указания в правой части и менее конкретные — в левой.

Для наглядности предположим: нам надо выбрать все теги `<span>` с классом `value` внутри тега `<table>` с классом `revenue`. Составляя селектор, вместо кода:

```
var $values = $('table.revenue .value');
```

обычно лучше писать так:

```
var $values = $('.revenue span.value');
```

После знакомства с приведенными примерами у вас мог возникнуть соблазн перегрузить селекторы, особенно в правой части. Не делайте этого! Если тот же набор элементов можно получить, используя меньшее количество селекторов, то уберите все лишнее. Например, старайтесь не писать следующих выражений:

```
var $values = $('table.revenue tr td span.value');
```

если те же самые элементы можно выбрать так:

```
var $values = $('.revenue span.value');
```

Предпоследнее выражение труднее читать, и оно не поможет повысить производительность.

Совет, который мы дали в этом разделе, должен помочь оптимизировать селекторы. Теперь посмотрим, как улучшить структуру проекта, разделив его на модули.

15.2. Разделение кода на модули

При работе над большими проектами нужно уделять особое внимание правильной организации кода. Главные приоритеты здесь — следить за чистотой глобального пространства имен и логической организацией модулей, в том числе для кода, написанного с использованием jQuery. Вы должны заботиться о его структуре точно так же, как могли бы делать это для любого другого кода, написанного без применения нашей библиотеки.

Чтобы продемонстрировать эти правила, начнем с простого кода JavaScript. Мы объявим несколько функций и объектов, как показано здесь:

```
function foo() {};
function bar() {};
function baz() {};
var obj = {};
var anotherObj = {};
```

Проблема данного кода в том, что все указанные функции и объекты являются глобальными (доступными как свойства объекта `window`), в связи с чем рано или поздно одна из библиотек переопределит какой-нибудь из них. Кроме того, невозможно сделать часть этих данных приватными — эту проблему мы уже обсуждали, когда изучали плагины jQuery и структурирование нашей библиотеки.

Еще одна проблема заключается в том, что если эти функции и объекты играют разные роли и используются в разных частях проекта, то единственный способ узнать ту или иную роль — распознать ее по имени объекта. Предположим, у нас есть

код JavaScript для интернет-магазина и объект `obj` и функция `bar()` применяются в части приложения, отвечающей за платежи, а функция `baz()` и объект `anotherObj` нужны для создания корзины. Как понять, что для чего предназначено?

Говоря техническим языком, наше приложение состоит из двух *модулей*: платежей и корзины. Модули играют важную роль в построении надежной архитектуры приложения и, как мы видим, помогают разделять и упорядочивать части кода. В JavaScript есть несколько средств для реализации модулей: AMD (Asynchronous Module Definition — асинхронное определение модуля, описанное в разделе 15.3), ECMAScript 2015 (также известный как ECMAScript 6), объектные литералы, CommonJS и др.

В следующих подразделах мы познакомимся с некоторыми шаблонами организации кода в модули.

15.2.1. Шаблон объектных литералов

Один из простейших способов разделения кода на модули заключается в использовании объектных литералов. Чтобы облегчить объяснение этого подхода, разобьем материал обсуждения на несколько этапов.

Первый этап позволит нам избежать засорения глобального пространства имен. Для этого мы создадим «внутреннее» пространство имен для функций, объектов и других переменных, объявленных с помощью объектного литерала:

```
var myService = {
  foo: function() {},
  bar: function() {},
  baz: function() {},
  obj: {},
  anotherObj: {}
}
```

Маленькое изменение обеспечит доступ к тем же самым функциям и объектам, но через единую точку входа. Благодаря этому вероятность того, что какая-то библиотека переопределит код, снижается. Однако остается проблема разделения функций и объектов в зависимости от их ролей. Чтобы ее решить, нужно дополнить код таким образом:

```
var myService = {
   foo: function() {},
   payment: {
      obj: {},
      bar: function() {}
   },
   basket: {
      anotherObj: {},
      baz: function() {}
   }
};
```

Определяет только функцию, принадлежащую ядру

Определяет модуль платежей

Определяет модуль корзины

Теперь, чтобы вызвать функцию `baz()` из модуля `basket`, нужно написать следующий код:

```
myService.basket.baz();
```

После такого логического разделения кода можно даже поместить каждый модуль в отдельный файл. Так, на следующем этапе можно создать файл `basket.js`, содержащий модуль `basket`, определенный таким образом:

```
myService.basket = {
  anotherObj: {},
  baz: function() {}
};
```

Благодаря разделению на файлы каждый разработчик получит возможность работать над своим модулем, не вмешиваясь в дела коллег.

Данный подход решает ряд проблем, но не очень подходит для обеспечения приватности данных, объявленных внутри модуля, — они по-прежнему доступны глобально. Нам нужен способ создания функций и объектов, доступных только внутри модуля, но не вне его.

(Внутри функции вы можете создавать данные, доступные только в этой функции.) В следующем подразделе мы представим улучшенный метод, позволяющий решить и эту задачу.

15.2.2. Шаблон Module

Шаблон Module был реализован в JavaScript для эмуляции концепции приватных методов и переменных, объявленных внутри объектов, подобно объектно-ориентированным языкам, например Java и C#. Мы использовали здесь термин «эмуляция», поскольку, строго говоря, в JavaScript нет модификаторов доступа, таких как `private` и `public`.

Данный шаблон состоит из двух основных частей: IIFE (подробнее об этом понятии см. в приложении) и объекта для возврата или приращения. Чтобы дать общее представление о шаблоне Module, рассмотрим простой пример кода с его реализацией:

```
var myFirstModule = (function() {
  return {
    foo: function() {},
    bar: function () {},
    obj: {}
  }
})();
```

Здесь создается функция IIFE, возвращающая объектный литерал. Он содержит функции и объекты, которые мы хотим сделать доступными публично.

На первый взгляд этот шаблон мало отличается от предыдущего. Но не спешите делать выводы, сначала дочитайте раздел до конца. Благодаря описанному подходу у нас есть возможность создавать переменные и функции, доступные внутри модуля, но недоступные извне. Предположим, мы хотим добавить к предыдущему коду «приватную» переменную с именем `count` для подсчета количества вызовов функции `foo()`. Мы также хотим создать «приватную» функцию `doSomethingPrivate()`, которая будет вызываться каждый раз при выполнении функции `bar()`. Код для этого выглядит так:

```
var myFirstModule = (function() {
   var count = 0;
   function doSomethingPrivate() {};

   return {
      obj: {},
      foo: function() { count++; },
      bar: function () { doSomethingPrivate(); }
  }
})();
```

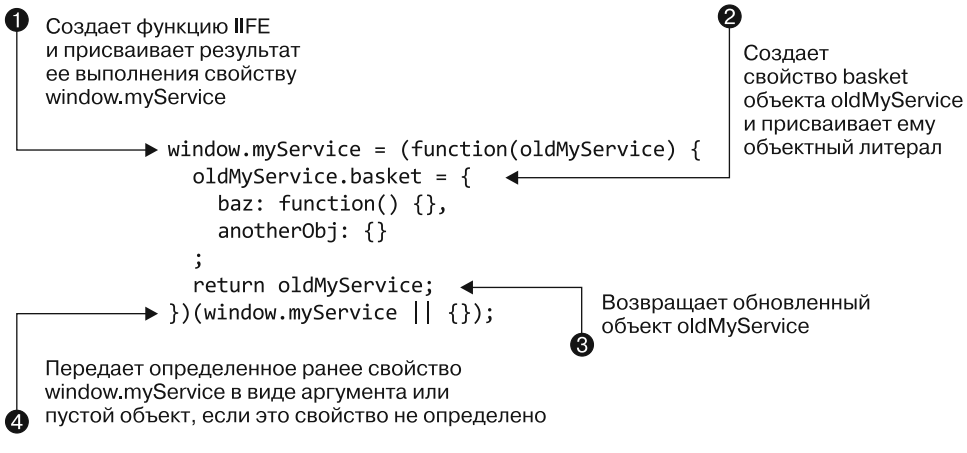
Объявляет «приватную» переменную

Объявляет «приватную» функцию

Увеличивает значение count

Вызывает функцию doSomethingPrivate()

Теперь, когда мы ближе познакомились с шаблоном Module, можно рассмотреть одну из его модификаций. Она позволяет добавить модуль `basket`, о котором мы говорили в предыдущем подразделе.



В этом коротком фрагменте кода реализовано несколько интересных приемов. Прежде всего, мы присвоили то, что возвращает функция IIFE, свойству `myService`, созданному для объекта `window` ❶. Для IIFE определен только один параметр, названный `oldMyService`, — он будет получать значение описанного ранее свойства `window.myService` или же пустой объект, если данное свойство не определено (другими словами, если его значение *ложно*) ❹. Таким образом можно увеличить свойство `myService` на один модуль (если уже есть хотя бы один) или создать первый модуль.

Внутри функции IIFE мы создали свойство этого объекта `basket`, представляющее наш модуль, и определили в нем функции и свойства, которые должны быть доступны публично ❷. Наконец, мы возвращаем дополненный объект ❸. Данное выражение необходимо, если значение `window.myService` ложно и функции IIFE был передан пустой объектный литерал.

На этом последнем примере мы заканчиваем краткий обзор разделения проекта на модули. Есть и другие шаблоны, которые мы не рассматриваем здесь, но сейчас обратим внимание на то, как сделать код чистым и более управляемым. В следующем разделе обсудим еще один шаблон создания модулей, называемый AMD, и познакомимся с RequireJS — библиотекой, предназначенной для загрузки модулей с учетом их зависимостей.

15.3. Загрузка модулей с помощью RequireJS

В предыдущем разделе мы обсудили два простых способа разделения кода на модули. Но у них есть большой недостаток: приходится вручную управлять зависимостями каждого модуля. Предположим, метод одного модуля использует свойство другого объекта. Для решения данной проблемы необходимо обратить внимание на то, в каком порядке подключаются модули на веб-странице. Но если в проекте десятки или сотни модулей, то это становится трудным и приводит к ошибкам. Задача еще более усложняется тем, что некоторые модули могут опираться на сторонние плагины, библиотеки или фреймворки.

Один из методов решения этой проблемы — определить зависимости модулей и затем использовать «что-нибудь», автоматически выстраивающее правильную последовательность подключений. Здесь-то и приходит на помощь асинхронное определение модулей и RequireJS (http://requirejs.org/).

Асинхронное определение модуля — интерфейс API JavaScript, который представляет собой механизм определения модулей, так что модуль и его зависимости загружаются асинхронно.

RequireJS — загрузчик файлов и модулей JavaScript, оптимизированный для использования в браузерах, но его также можно применять и в других средах JavaScript, таких как Rhino и Node.js. Эта удобно настраиваемая библиотека обеспечивает большую гибкость, но для решения простейших задач ее можно задействовать почти без настройки. Здесь мы не будем описывать ее во всех подробностях, но дадим достаточно информации, чтобы вы могли начать с ней работать.

RequireJS загружает каждую зависимость в виде тега `script` внутри элемента страницы `head`. Затем библиотека ждет, когда все зависимости загрузятся, и вычисляет правильную последовательность вызова функций, определяющих модули. После этого она вызывает функции определения модулей в правильном порядке.

Теперь, когда вы узнали, что такое библиотека RequireJS и для чего она нужна, начнем ее использовать.

15.3.1. Начало работы с RequireJS

Первое, что надо сделать, — это загрузить библиотеку. Откройте страницу http://requirejs.org/docs/download.html и загрузите самую свежую версию.

Прежде чем задействовать библиотеку RequireJS на наших веб-страницах, необходимо обсудить несколько ее важных принципов. Первый из них — функция `define()`, определенная по методу AMD.

Синтаксис функции: define

`define([[id, ] dependencies,] factory)`

Определяет новый модуль с возможными зависимостями и идентификатор.

Параметры

`id`	(Строка) Идентификатор модуля.
`dependencies`	(Массив) Массив, содержащий имя модулей, от которых зависит новый модуль.
`factory`	(Объект|Функция) Объектный литерал или функция, определяющая новый модуль. Если это функция, то она принимает в качестве параметров зависимости в том порядке, в котором они заданы.

Возвращает

`undefined`.

Для определенности предположим, что у нас есть объект `Person`, описанный в файле `Person.js`. Единственное свойство этого объекта называется `name` и не имеет зависимостей. Используя функцию `define()`, можно создать его таким образом:

```
define({
  name: 'John Doe'
});
```

Как видите, мы не объявили ни ID, ни зависимостей для этого модуля.

Кроме `Person`, у нас есть объект `Car`, хранящийся в файле `Car.js`. У данного объекта есть метод `getOwner()`, внутри которого используется свойство `name` объекта (литерала) `Person`; таким образом, у него есть зависимость от модуля `Person`. Модуль `Car` может быть описан так:

```
define(['Person'], function(Person) {
  function Car() {
    this.getOwner = function() {
      return 'The owner is ' + Person.name;
    };
  }
  return Car;
});
```

Здесь мы указали `Person` как зависимость и затем создали функцию `Car`, которая играет роль конструктора. У данной функции есть метод `getOwner()` — он возвращает простое сообщение, использующее свойство `name` объекта `Person`. В итоге возвращается объект `Car`, доступный в виде модуля. Теперь мы определили два модуля, но еще не применили их. Для этой цели существует функция `require()`.

Функция `require()`, как и `define()`, определяет модуль, но, кроме того, еще и выполняет его. Это значит, что зависимые модули загружаются и запускаются перед выполнением функции. Обычно в приложении используется единственная функция `require()` как главная точка входа, а другие модули определяются через `define()`.

Для завершения примера предположим, что у нас есть файл `main.js`, играющий роль точки входа в приложение, и в данном файле мы выводим сообщение о владельце машины. Для этого мы воспользуемся функцией `require()` и определим `Car` как ее зависимость:

```
require(['Car'], function(Car) {
  var car = new Car();
  alert(car.getOwner());
});
```

Если вы внимательно читали этот подраздел, то, наверное, уже задаетесь вопросом: как RequireJS из простой строки вроде `"Car"` узнает, какой модуль надо загрузить? Ответ: библиотека создает модуль с тем же именем, что и файл, в котором находится его описание. В нашем примере, несмотря на то что мы не задали ID для модулей, есть три имени модулей: `main`, `Car` и `Person`, так как у нас три файла JavaScript: `main.js`, `Car.js` и `Person.js`.

Пойдем дальше. Если у нас есть файл `Basket.js`, который хранится в папке `cart`, то модуль будет называться `cart/Basket`.

На последнем этапе предоставим RequireJS выполнить оставшуюся работу, подключив эту библиотеку на HTML-странице. Для этого можно использовать элемент `script` с атрибутом `data-main`. Данный атрибут применяется, чтобы задать точку входа приложения (файла с `require()`). Если библиотека RequireJS и все созданные ранее модули находятся в папке `scripts`, то можно создать демонстрационную страницу, написав такой код:

```
<script data-main="scripts/main" src="scripts/require.min.js"></script>
```

Работающий пример, в котором реализованы все созданные нами в этой главе фрагменты кода, находится в папке `chapter-15/requirejs`, предоставленной с книгой.

Теперь, когда мы познакомились с некоторыми основными принципами, научимся использовать RequireJS в сочетании с jQuery.

15.3.2. Использование RequireJS с jQuery

В главе 12 мы обсудили, что такое плагины jQuery и как их создавать, дополняя таким образом возможности ядра этой библиотеки. Будучи ее плагинами, они, естественно, зависят от нее. А код, который вы напишете с использованием методов этих плагинов, в свою очередь, зависит от jQuery и самих плагинов. Отличная ситуация для применения RequireJS в нашем проекте.

Наши плагины с самого начала создавались с помощью AMD. Поэтому на первый взгляд кажется, что единственное решение, которое позволит нам приспособить их к использованию `define()`, — объявить jQuery в качестве зависимости. К счастью, есть и лучший вариант, который мы рассмотрим ниже.

Плагины jQuery, использующие AMD

Если вы разрабатываете плагин с нуля и рассчитываете использовать RequireJS, то можете разместить определение плагина внутри вызова `define()`, объявив jQuery как зависимость:

```
define(['jquery'], function($) {
  $.fn.jqia = function() {
    // Код плагина
  };
});
```

Как видите, нам не нужно возвращать модуль, поскольку мы дополняем исходный объект jQuery. Более того, не нужно размещать определение плагина внутри IIFE, поскольку у нас уже есть функция, в которой оно уже размещено, и объект jQuery будет предоставлен RequireJS. Предыдущий код в том виде, в каком он есть сейчас, не будет работать, поскольку RequireJS не распознает строку `jquery`, заданную как зависимость в библиотеке jQuery. У данной проблемы есть простое решение — в соответствии с соглашениями имен RequireJS переименовать файл jQuery в `jquery.js` и разместить его в той же папке, что и файл с точкой входа в приложение. После этого все готово для использования плагина.

Теперь предположим, что плагин, помещенный внутри `define()`, хранится в файле `jquery.jqia.js`, а точка входа проекта — в файле `main.js`, который зависит от jQuery и от нашего плагина. С учетом этого файл `main.js` будет выглядеть так:

```
require(['jquery', 'jquery.jqia'], function($) {
  // Код, использующий jQuery и плагин
});
```

В данном примере есть две детали, заслуживающие особого внимания. Первая: поскольку код опирается на плагин, то оно объявлено как зависимость. Вторая: для использования плагина второй параметр не нужен, поскольку после загрузки оно становится свойством исходного объекта jQuery.

Использование существующих плагинов jQuery

Когда новый проект только начинается, легко спланировать структуру его модулей с расчетом на использование AMD. Но часто приходится обслуживать старые библиотеки или стороннее программное обеспечение, которое создавалось без учета AMD. В таких случаях можно создать для RequireJS конфигурационный файл, который позволит применять `define()`, не меняя эти файлы:

```
requirejs.config({
  shim: {
    'jquery.jqia': ['jquery']
  }
});
```

Данная конфигурация, в которой использовано свойство `shim`, должна быть помещена перед вызовом `require()`. Свойство `shim` представляет собой объект со значением и позволяет задать зависимости для плагинов jQuery, не вызывающих

`define()`. Объектный литерал, назначенный `shim`, должен содержать имена модулей в качестве свойств, и массив их зависимостей в качестве их значений.

С учетом представленного описания окончательный код `main.js` должен выглядеть так:

```
requirejs.config({
  shim: {
    'jquery.jqia': ['jquery']
  }
});
require(['jquery', 'jquery.jqia'], function($) {
  // Код, использующий jQuery и плагин
});
```

В этом последнем примере мы увидели, как можно применить RequireJS в проектах, в которых используется библиотека jQuery и ее плагин. Описанные здесь принципы не претендуют на полное описание RequireJS, много вопросов остаются открытыми — такие как оптимизатор и различные свойства для настройки конфигурации. Но представленного материала достаточно, чтобы начать использовать эту библиотеку для управления зависимостями проекта.

Помимо удобного способа управления зависимостями модуля и порядком их подключения к проектам, нам также нужен улучшенный и более быстрый способ установки, обновления и даже удаления стороннего программного обеспечения, используемого в проекте. Именно это мы обсудим в следующем разделе.

15.4. Управление зависимостями с помощью Bower

При разработке веб-проекта для ускорения процесса часто применяют сторонние компоненты. Если таких компонентов один или два, то ими легко управлять вручную, и все так делали еще несколько лет назад. Но по мере усложнения проектов разработчики стали нуждаться в более надежном способе установки и управления зависимостями.

За последние несколько лет появилось множество инструментов для решения этой проблемы. Один из них — Bower (http://bower.io/). В данном разделе мы познакомимся с ним и с его основными функциями, обращая особое внимание на интеграцию jQuery в проекты с помощью Bower.

15.4.1. Начало работы с Bower

Менеджер пакетов Bower был создан Twitter в 2012 году. С тех пор много разработчиков внесли свой вклад в данный проект, и сейчас это один из самых известных инструментов разработки клиентской части веб-приложений. Bower позиционируется как «менеджер пакетов для веб-разработки» — другими словами, это менеджер зависимостей для JavaScript, CSS и т. п., в том числе для веб-шрифтов. Пакетом может быть библиотека JavaScript (такая как jQuery, jQuery UI или QUnit), файл

CSS (например, `Reset.css` или `Normalize.css`), пиктограммный шрифт (такой как FontAwesome), фреймворк (такой как Bootstrap), плагин jQuery (например, jQuery Easing или jQuery File Upload) и любой другой файл, который разработчик намерен представить как сторонний компонент веб-проекта.

Как ни удивительно, но у Bower есть несколько собственных зависимостей и их необходимо подключить перед его использованием. Это Node.js (http://nodejs.org/), уже несколько раз упоминавшаяся в нашей книге платформа, позволяющая применять JavaScript как язык программирования серверной стороны; менеджер пакетов npm (https://www.npmjs.com/), который устанавливается вместе с Node.js; и клиент Git (http://git-scm.com/downloads). Когда все это будет установлено, вы будете готовы погрузиться в мир Bower.

Bower создает *файл манифеста* с именем `bower.json`. Это файл в формате JSON, в котором хранится информация о проекте — название, автор (-ы), актуальная версия и используемые пакеты. Данный файл очень полезен при работе в группе, поскольку позволяет делиться указанной информацией с участниками. Это удобно, так как они могут установить все зависимости проекта одной командой (скоро мы обсудим этот вопрос подробнее).

Когда все зависимости Bower будут установлены, можно установить и сам Bower, введя в интерфейсе командной строки (command-line interface, CLI) такую команду:

```
npm install -g bower
```

Этот процесс займет несколько минут. После выполнения команды Bower будет готов к использованию в ваших проектах.

Теперь предположим, что мы разрабатываем новый проект и хотим применить Bower для управления его зависимостями. Для этого прежде всего надо открыть папку проекта и создать внутри нее файл `bower.json`. Данный файл можно создать вручную или с помощью Bower. В нашем примере мы рассмотрим второй вариант. Откройте CLI, перейдите в папку проекта и введите команду:

```
bower init
```

Bower запросит информацию о проекте, как показано на рис. 15.2.

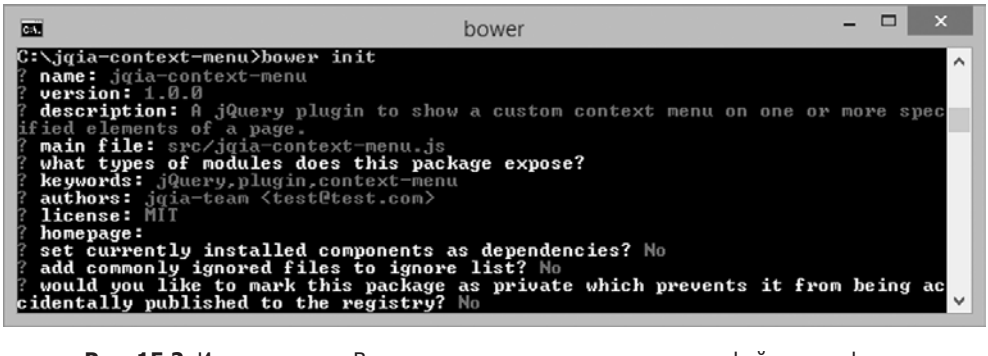


Рис. 15.2. Использование Bower с целью создания для проекта файла манифеста

Когда вы заполните все поля и подтвердите ввод информации, в папке проекта будет создан файл манифеста (`bower.json`). Пример такого файла показан в листинге 15.1.

Листинг 15.1. Пример файла bower.json

```
{
  "name": "jqia-context-menu",
  "version": "1.0.0",
  "authors": [
    "jqia-team <test@test.com>"
  ],
  "description": "Плагин jQuery для вывода собственного контекстного меню
  для одного или нескольких элементов веб-страницы.",
  "main": "src/jqia-context-menu.js",
  "keywords": [
    "jQuery",
    "plugin",
    "context-menu"
  ],
  "license": "MIT"
}
```

Теперь наш проект готов к использованию Bower, но мы пока еще не сделали ничего особо впечатляющего или полезного. В следующих двух подразделах мы удовлетворим ваш интерес.

15.4.2. Поиск пакетов

Пакетами в Bower называются любые компоненты, подключаемые к проекту. Bower может управлять не всеми библиотеками и фреймворками, доступными через Сеть, но поскольку в состав Bower входит более 34 000 пакетов, вы можете быть вполне уверены в доступности всего нужного для проекта.

Чтобы узнать, доступен ли данный пакет, его можно поискать с использованием Bower. Для этого откройте CLI и введите такую команду:

```
bower search <имя_пакета>
```

где `<имя_пакета>` — имя пакета, который мы ищем.

И разве может быть лучший пример, чем поиск jQuery? Чтобы найти jQuery с использованием Bower, введем команду:

```
bower search jquery
```

В результате выполнения этой команды получим не только саму библиотеку jQuery, но и все остальные пакеты, в имени, описании или в ключевых словах которых есть фрагмент `"jquery"`.

После того как найдено точное имя нужного пакета, его можно установить.

15.4.3. Установка, обновление и удаление пакетов

Перед инсталляцией пакета необходимо принять важное решение: будет ли эта зависимость присутствовать в готовом продукте или нужна только на стадии разработки?

Например, jQuery — пакет, необходимый в готовом продукте: едва ли не весь наш код JavaScript опирается на jQuery. То же самое касается компонентов jQuery UI или Backbone.js. Другие пакеты, такие как фреймворк тестирования (например, QUnit или Mocha), нужны только при разработке проекта для обеспечения надлежащего качества и надежности кода. Никаким другим частям проекта — по крайней мере тем, что будут разработаны, — указанные пакеты не нужны. Это важное отличие, от которого будет зависеть способ подключения пакетов.

Чтобы установить пакет с помощью Bower, нужно выполнить такую команду:

```
bower install <имя_пакета> <--production-or-development>
```

где `<имя_пакета>` — имя пакета, а `<--production-or-development>` — флаг, определяющий, нужен пакет только на стадии разработки (`--save-dev`) или нет (`--save`).

Чтобы установить jQuery как зависимость для готового продукта, откройте CLI и перейдите в папку проекта, на тот же уровень, где находится файл `bower.json`. После этого введите команду:

```
bower install jquery --save
```

Предположим, мы также хотим установить QUnit, поскольку желаем провести модульное тестирование проекта. Для установки этого пакета как зависимости на этапе разработки нужна такая команда:

```
bower install qunit --save-dev
```

При первом выполнении команды `install` создается папка `bower_components`. В ней размещаются загруженные Bower пакеты. Одновременно имена этих пакетов записываются в файл `bower.json` — в раздел `dependency` или `devDependency` в зависимости от выбранного режима установки.

После того как зависимость — например, jQuery — загружена, нужно включить ее в проект. Предположим, у нас есть файл `index.html`, расположенный на том же уровне, что и папка `bower_components`. Тогда в этот файл надо вставить строку:

```
<script src="bower_components/jquery/dist/jquery.min.js"></script>
```

Путь зависит от проекта, но структура тега обычно одна и та же.

Когда все необходимые зависимости будут установлены, можно приступать к разработке функций проекта.

Процесс разработки обычно занимает много времени. Пока вы пишете код, может выйти новая версия одного или нескольких использованных вами пакетов. В новых версиях часто исправляются серьезные ошибки, поэтому важно вовремя обновлять зависимости.

Обновление пакетов

Обновить пакет очень легко. Нужно лишь перейти в корень проекта и ввести в CLI следующую команду:

```
bower update <имя_пакета>
```

Так, для обновления jQuery нужно написать:

```
bower update jquery
```

Иногда требуется обновить сразу все пакеты. Bower позволяет сделать это с помощью одной команды:

```
bower update
```

Бывает и так, что зависимость больше не нужна и ее требуется удалить. Ниже мы покажем, как это сделать.

Удаление пакетов

Чтобы удалить зависимость с помощью Bower, можно воспользоваться такой командой:

```
bower uninstall <имя_пакета> <--production-or-development>
```

Здесь значение двух параметров такое же, как было показано ранее.

Предположим, мы решили испробовать возможности QUnit в проекте, но результат не понравился и мы захотели писать тесты с помощью Mocha. Нам нужно удалить библиотеку QUnit, которая была установлена как зависимость для этапа разработки. Для этого нужно ввести в CLI такую команду:

```
bower uninstall qunit --save-dev
```

Данным примером мы завершаем обзор Bower. У этого менеджера пакетов есть и другие команды, которые мы не рассматривали, но для начала представленного материала достаточно, чтобы ускорить разработку ваших проектов.

Благодаря jQuery, Bower, RequireJS и другому программному обеспечению, описанному в этой книге, вы уже готовы создавать серьезные, профессиональные сайты и веб-приложения. Но данная глава была бы неполной без рассказа об одностраничных приложениях и их создании с помощью фреймворков MVC.

15.5. Создание одностраничных приложений с помощью Backbone.js

Как уже говорилось во вступлении, при работе над большими проектами очень важна правильная организация кода. Представьте на минуту, что было бы, если бы программные продукты, создаваемые такими компаниями, как Google или Microsoft, имели плохо организованный код! В проектах, где постоянно

появляются новые и обновляются старые функции, плохо структурированный код приводит к тому, что невероятное количество времени тратится впустую. А, как мы знаем, время — деньги.

Один из самых распространенных шаблонов структурирования программного кода называется MVC (Model — View — Controller, Модель — Вид — Контроллер). Это шаблон программной архитектуры, в котором код разделяется на три основные части: модель, вид и контроллер. Модель представляет данные приложения, такие как зарегистрированный пользователь или сайт. Вид — компонент, который занимается отображением данных, то есть отвечает за то, как данные будут отображаться на веб-страницах, если вы используете этот шаблон в Интернете. Контроллер обновляет состояние модели и передает данные виду (например, изменения в адресе пользователя).

Если бы вы были PHP-разработчиком, то знали бы о Symphony, Laravel или Zend Framework; если бы вы были разработчиком на Java, то наверняка слышали бы о Spring Web MVC или Struts. У JavaScript тоже есть свои фреймворки. Один из этих фреймворков, реализующих шаблон MVC, называется Backbone.js.

Теоретически jQuery можно было бы использовать для создания одностраничных приложений. Однако поскольку это не является главным назначением библиотеки, то часто приводит к сложному коду, который трудно писать и поддерживать. Для создания таких приложений нужен фреймворк, специально предназначенный для указанных целей, такой как Backbone.js. Он опирается на библиотеки Underscore.js и jQuery. Таким образом, уже полученные вами знания пригодятся при его изучении.

Многие современные веб-приложения опираются на JavaScript и Ajax для улучшенного взаимодействия с пользователем и для создания одностраничных приложений (single-page applications, SPA). Эти приложения после загрузки в браузере выполняют все HTTP-запросы без обновления всей страницы. В отношении производительности это большое преимущество, поскольку браузер не загружает заново все ресурсы (файлы JavaScript и CSS, шрифты, и т. п.), а только маленькие порции данных, полученные от сервера, и затем вставляет их в DOM.

У такого подхода есть своя цена. Обычно SPA дольше загружаются, поскольку им требуется загрузить больше кода, чем другим приложениям. Поэтому нужно обращать внимание на размер кода, необходимого для работы приложения. Страницы, загружающиеся слишком долго, рискуют потерять посетителей.

В следующих подразделах мы обсудим свойства шаблона, реализованного в этом фреймворке, а также кратко рассмотрим его главные концепции, такие как модели, виды и маршрутизаторы. Мы также разработаем простое, но полезное приложение по построению списков текущих дел (здесь и далее будем называть его Todos manager). Мы выбрали это приложение, поскольку на его примере часто изучают новые фреймворки. Его назначение — сохранять задачи, которые вам нужно выполнить, так, чтобы о них можно было быстро вспомнить. Обратите внимание: данный раздел не претендует на исчерпывающее руководство по Backbone.js — это лишь поверхностное описание. Если вам нужен полный учебник, то советуем купить книгу, посвященную фреймворку.

15.5.1. Зачем нужны MV*-фреймворки

Подобно многим другим вещам в жизни, программирование развивается по спирали. Стоит решить проблему теми средствами, которыми вы располагаете в данный момент, как появляются новые библиотеки и фреймворки, позволяющие улучшить код и ваши решения. Потом появляются новые задачи, и все повторяется сначала.

При появлении первых SPA многие разработчики начали использовать jQuery (и подобные библиотеки) в сочетании с функциями обратного вызова и множеством вызовов Ajax, чтобы синхронизировать пользовательский ввод с данными, хранящимися на сервере. По мере усложнения этих приложений их код становился все менее обслуживаемым и масштабируемым. Разработчики испытывали все более острую потребность во фреймворке, который позволил бы создавать хорошо структурированный и обслуживаемый код. Так появились Backbone.js, AngularJS, Ember и многие другие подобные им фреймворки. Обычно их называют MV*-фреймворками, поскольку они не вполне соответствуют шаблонам MVC, MVP (Model — View — Presenter, Модель — Вид — Представитель) и MVVM (Model — View — ViewModel, Модель — Представление — Модель представления).

Иллюстрация, представляющая шаблон MVC и его реализацию во фреймворке Backbone.js, представлена на рис. 15.3.

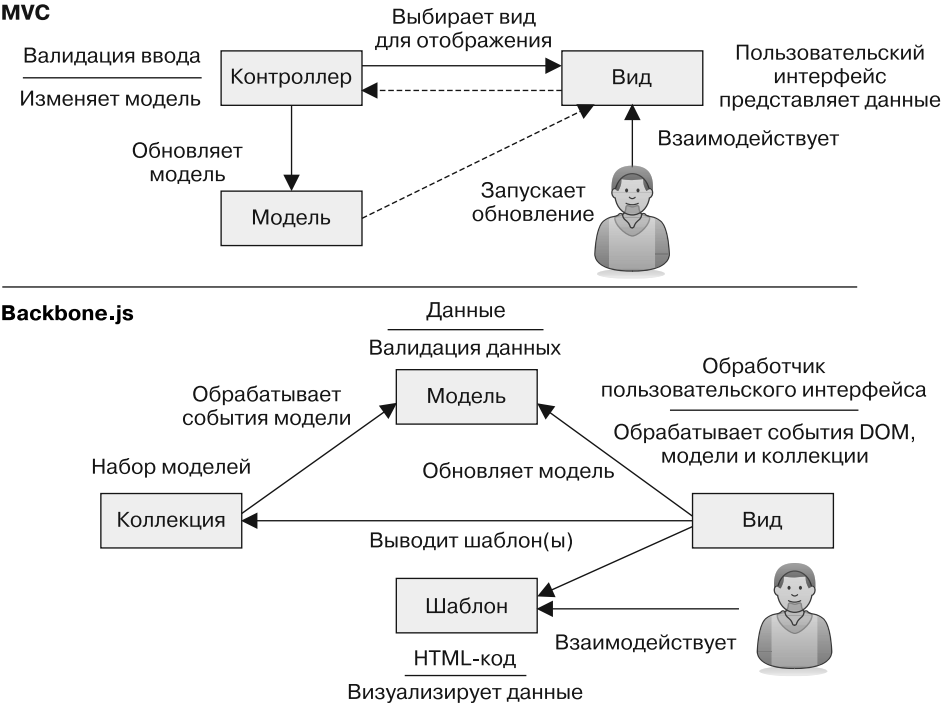


Рис. 15.3. Сравнение шаблона MVC и его реализации в Backbone.js

Как видите, главным различием между классическим шаблоном MVC и его реализацией в Backbone.js является компонент «Вид». Он также играет роль контроллера, так как отвечает и за построение пользовательского интерфейса, представленного блоком «Шаблон», и за обновление модели.

Теперь, когда вы познакомились с моделью, реализованной в Backbone.js, можно внимательно рассмотреть каждый компонент по очереди, проанализировать его назначение и функции.

15.5.2. Начало работы с Backbone.js

Как мы уже знаем, Backbone.js позволяет разработчикам разделить код на меньшие фрагменты. Рассмотрим вкратце его компоненты.

Модель

Модели — это объекты, представляющие данные приложения. Свойствами модели могут быть атрибуты, описывающие объект. Здесь обычно размещаются методы валидации данных, инициализации их свойств и уведомления сервера об изменениях в модели.

Чтобы понять, что такое модель, представим себе приложение Todos manager, которое мы намерены разработать. Одна из моделей нашего приложения (и, в сущности, единственная) — запись Todo, выполняющая единственное действие. У каждой из записей Todos будет заголовок, позиция в списке и свойство, показывающее, выполнено задание или нет. Эти три атрибута объекта являются свойствами модели.

Моделям ничего не известно о том, как будет отображаться информация, которую они содержат. Каждая из них может быть соединена с одним или несколькими видами, отслеживающими изменения. Таким образом мы гарантируем, что отображаемая информация синхронизирована с данными, описанными с помощью модели. В Backbone.js модели создаются путем расширения объекта `Backbone.Model`. Кроме того, можно группировать модели в единую сущность под названием *«коллекция»* — о них мы поговорим ниже.

Вот пример модели Todo с описанными выше свойствами:

```
var todo = Backbone.Model.extend({
  position: 1,
  title: '',
  done: false
});
```

Как видите, простейший вариант модели — обычный набор свойств, объявленных как объект и переданных методу `Backbone.Model.extend`.

Коллекции

Коллекция — это набор моделей, который используется для их организации и выполнения операций над ними. При создании коллекции необходимо задать свойство, определяющее тип создаваемой коллекции.

Коллекции позволяют не отслеживать вручную экземпляры каждой модели в отдельности. В приложении Todos manager вам понадобится способ группировки записей, поскольку необходимо представлять их как уникальный список. Для этого воспользуемся коллекцией.

В Backbone.js для создания коллекции используется расширение объекта `Backbone.Collection`. У нее может быть один или несколько видов, отслеживающих ее изменения.

В следующем примере показано, как создать коллекцию записей Todo, применяя модель, показанную выше:

```
var todoList = Backbone.Collection.extend({
  model: todo
});
```

Как видите, коллекция может представлять собой просто объект с единственным свойством, определяющим содержащуюся в нем модель.

Виды

Вид — компонент, отвечающий за обработку событий DOM. Для этого он выполняет один или несколько методов в зависимости от потребностей. Как правило, вид привязан к определенной модели. Виды помогают синхронизировать DOM с данными. Именно в видах выполняется вся логика, управляющая представлением данных.

Этот компонент не содержит код HTML. Данный код находится в *шаблонах*, управляемых другими библиотеками JavaScript, такими как Mustache.js или Underscore.js. Поскольку Backbone.js опирается на Underscore.js, в демонстрационном проекте мы тоже будем ее использовать.

В нашем примере — Todos manager — вид представляет собой объект, который позволит отслеживать события DOM и выполнять один или несколько соответствующих методов. Для наглядности представим, что событие DOM — нажатие кнопки Add Todo (Добавить запись) или добавление новой модели в список записей Todo.

Самое важное свойство вида — `el`, ссылка на элемент DOM. Такая ссылка должна быть у каждого вида, она связывает объект `View` с элементом DOM. Нам часто будут встречаться операции и методы jQuery, работающие с `el`, поэтому в Backbone.js определено удобное свойство `$el` — обычный объект jQuery, в который обернуто свойство `el`.

Есть два способа ассоциировать свойство `el` с элементом DOM. Первое — создать новый элемент для вида и затем добавить его в DOM. В таком случае новый элемент создается фреймворком и ссылка на него присваивается автоматически. Кроме того, можно использовать другие свойства, чтобы назначить элементу DOM основные атрибуты. Эти свойства — `tagName`, определяющее имя тега (`div`, `p`, `li` и т. п.), `id` и `className`, которое позволяет назначить имя класса CSS. Второй способ — сослаться на элемент, уже существующий на странице.

Еще одно важное, хотя и необязательное свойство вида — метод `render()`. В нем описывается логика построения шаблона — операции, которые будут выполнены, чтобы построить HTML, представляющий модель. В этом методе обычно присутствует объект JSON, представляющий модель, ассоциированную с видом. Объект

передается в шаблон, чтобы наполнить его данными и представить пользователю готовый HTML. Точнее, шаблон компилируется в функцию с помощью метода Underscore `_.template()`, после чего этой функции передается объект JSON. Предположим, у нас есть такой код:

```
var TodoView = Backbone.View.extend({
  render: function() {
    this.$el.html(this.template(this.model.toJSON()));
    return this;
  }
});
```

В этом примере вводится элемент `li` — первым из рассмотренных способов, чтобы связать свойство `el` с элементом DOM. Создается и шаблон:

```
var TodoView = Backbone.View.extend({
  tagName: 'li',
  className: 'todo',
  template: _.template($('#todo-template').html())
});
```

Шаблон — фрагмент HTML-кода, содержащий ряд шаблонных тегов, которые впоследствии будут заменены на данные, хранящиеся в модели. Предположим, мы хотим отобразить заголовок записи Todo. Для этого можно создать такой шаблон:

```
<script type="text/template" id="todo-template">
  <span class="todo-title"><%- title %></span>
</script>
```

Подведем итоги. Пользователи взаимодействуют с разметкой HTML. Эта разметка содержится в шаблонах, управляемых видами. Виды отвечают за уведомление модели о событиях и в конечном счете за изменение своей части HTML-кода.

Виды также отвечают за передачу моделей шаблонам. В шаблонах содержатся метки-заполнители, вместо которых подставляются значения атрибутов. Во время обработки шаблона эти заполнители заменяются на действительные значения моделей. Можно использовать и другие структуры, такие как условные выражения, применяющие переданные данные для принятия решения, будет ли показан тот или иной HTML-элемент или атрибут.

Как мы уже видели на примере фрагмента шаблона, шаблоны создаются путем вставки их содержимого в теги `<script>`. Эти теги, как правило, имеют атрибут `type="text/template"` и ID, по которому их легко получить с помощью jQuery или другой аналогичной библиотеки. Пример шаблона — показанный ранее тег `<%- title %>`, где `<%- %>` используется для подстановки значения переменной и ее экранирования в HTML, а `title` — имя свойства, которое будет передано из вида.

Маршрутизатор

Маршрутизатор — это способ привязать части проекта к определенному URL и отслеживать состояния приложения. Он строит путь к функции. Обычно работает с одной или несколькими моделями и затем обновляет вид. На рис. 15.4 схематически показано, какое место маршрутизатор занимает в общей архитектуре Backbone.js.

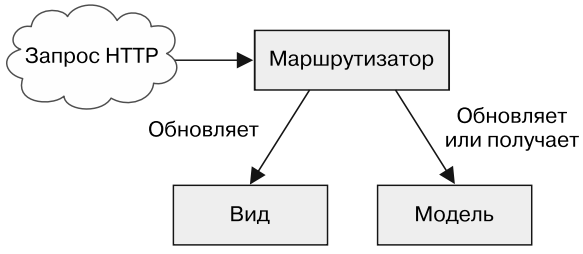


Рис. 15.4. Схема взаимодействия маршрутизатора с другими компонентами Backbone.js

Маршрутизаторы преобразуют URL или хеш-фрагменты URL в состояние приложения. Это означает следующее: они необходимы, если требуется предоставить другим пользователям доступ к состоянию приложения или дать возможность его запоминать. Если обнаруживается, что один из путей, определенных в маршрутизаторе, соответствует заданному URL, то выполняется соответствующая функция.

Чтобы проиллюстрировать эту идею, снова обратимся к Todos manager. Предположим, мы хотим предоставить пользователю возможность открывать выбранные записи Todo и просматривать их описания. Для этого нам потребуется маршрутизатор. Можно создать маршрутизатор и ассоциировать URL — например, `todo/MY-TODO-ID` — с функцией, которая будет обновлять страницу, убирать с нее список записей Todo и показывать вместо него подробное описание выбранной записи.

Маршрутизаторы создаются путем расширения объекта `Backbone.Router`. Как правило, в каждом приложении есть только один маршрутизатор, хотя теоретически их можно создать сколько угодно. Пример маршрутизатора показан в листинге 15.2.

Листинг 15.2. Пример маршрутизатора в Backbone.js

```
var TodoRouter = Backbone.Router.extend({
    routes: {
        "todo/:id": "getTodo",
        "search/:string": "searchTodo"
    },

    getTodo: function(id) {
        // Код функции
    },

    searchTodo(string) {
        // Код функции
    }
});
```

❶ Определяем маршруты и связываем их с функциями

❷ Объявляем функции, которые будут выполняться для различных маршрутов

В показанном примере маршрутизатора созданы два маршрута ❶, `todo/:id` и `search/:string`. Набор маршрутов представляет собой объект, назначенный свойству `property`. Ключами этого объекта являются шаблоны маршрутов, а значениями — имена функций, которые будут выполняться при найденном соответствии

между заданным URL и маршрутом-ключом (например, `todo/2` соответствует первому из заданных маршрутов). В данном примере маршрутам `todo/:id` и `search/:string` отвечают функции `getTodo` и `searchTodo` соответственно. Тела функций определены в оставшейся части объектного литерала ❷.

Как было показано ранее, если найдено соответствие заданного URL с одним из определенных в маршрутизаторе путей, то выполняется соответствующая функция. В качестве аргументов ей передаются одна или несколько переменных, определенных в пути. Переменные являются частью заданного пути, перед ними ставится двоеточие, например `:id`.

Разъяснив этот вопрос, мы завершаем анализ компонентов фреймворка. Настало время приступить к разработке Todos manager.

15.5.3. Создание приложения Todos manager с помощью Backbone.js

При изучении нового фреймворка большинство людей сходится во мнении, что один из самых эффективных способов усвоить его понятия — разработать небольшой проект. Цель данного подраздела — помочь вам создать простое приложение Todos manager (рис. 15.5), реализующее базовый набор операций CRUD (Create Read Update Delete — «создание, чтение, обновление, удаление»). Чтобы проект был максимально простым, мы не будем подключать веб-службы для передачи и хранения данных, а вместо этого воспользуемся интерфейсом Web Storage API (http://www.w3.org/TR/webstorage/) на базе адаптера Backbone.js под названием Backbone.localStorage (https://github.com/jeromegn/Backbone.localStorage). Полный код приложения вы найдете в наборе исходных кодов для этой книги в папке `chapter-15/todos-manager`.

Рис. 15.5. Пример приложения Todos manager

У приложения Todos manager будет только одна модель, соответствующая записи Todo. У каждой записи Todo будет свойство `title`, в котором будет храниться описание запланированной задачи, положение в списке в свойстве `position` и логическое значение, определяющее, выполнена ли эта задача, — в свойстве

done. Мы также создадим одну коллекцию, чтобы упростить сортировку моделей. Наконец, возможно вопреки вашим ожиданиям, у нас будет два вида. Мы реализуем шаблон «Контроллер элемента», состоящий из двух видов: первый из них управляет набором элементов, а экземпляры второго — каждым элементом в отдельности.

Структура проекта, показанная на рис. 15.6, весьма проста. Есть веб-страница index.html с разметкой HTML и шаблоны, используемые приложением. Есть также папка css с основными файлами CSS, обеспечивающими лучший вид и поведение страницы, папка js со всеми подключенными библиотеками (такими как jQuery и Backbone.js) в подпапке vendor и файлом app.js, содержащим собственный код проекта. Для простоты мы поместим весь код в один файл, но при работе с большими проектами лучше хранить каждый объект в отдельном файле. Файлы следует разделить по подпапкам, имена которых соответствуют компонентам проекта: models, collections и views.

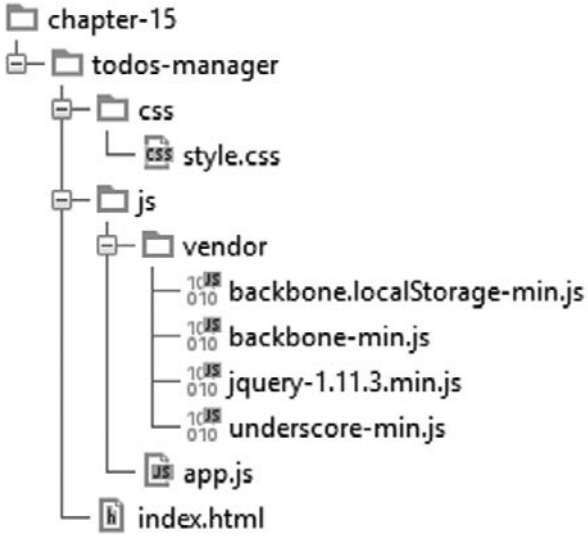


Рис. 15.6. Структура файлов и папок для приложения Todos manager

Теперь, когда мы определились с функциями проекта, можно приступать к его разработке.

Создание HTML-кода

Ни одно веб-приложение, взаимодействующее с пользователями, не может обойтись без интерфейса. Поэтому нашим первым шагом станет создание разметки HTML. Вся разметка HTML и шаблоны будут находиться в файле index.html. Интерфейс простой, поскольку нам требуется всего два компонента: первый — место, где пользователь будет создавать новые записи (заголовок для Todo) и добавлять их в список, а второй — сам список записей Todo.

Хорошим тоном при разработке веб-приложения считается предоставить пользователю возможность обратной связи на случай сбоя одной любой из выполняемых операций. Для этого мы создадим элемент DOM, который при необходимости будет показывать сообщения об ошибках.

Реализация задуманного выглядит в HTML так:

```
  <div id="todo-sheet">
  <input id="new-todo" type="text" placeholder="Разместите здесь свою запись" />
  <button id="new-todo-save">Сохранить</button>
  <span class="error-message"></span>
  <ul class="todos">
  </ul>
</div>
```

В этом коде мы создали элемент (`ul`), в который будут вставляться записи Todo, но еще не решили, каким образом они будут отображаться. Нам нужен шаблон для элементов `<li>`, содержащих информацию о записях Todo. Как вы это сделаете, зависит от ваших дизайнерских предпочтений, но мы тем не менее дадим один совет. Как уже говорилось во вступлении, у каждой записи Todos есть свойства `title`, `position` и `done`. Первые два из них могут быть представлены в виде простого элемента `span`, а для последнего лучшим вариантом будет флажок, позволяющий пользователю отметить запись Todo как выполненную.

СОВЕТ

> В показанной разметке мы пропустили элемент label, связанный с полем input, чтобы сконцентрироваться на коде, непосредственно связанном с проектом. Однако при работе с элементами форм хорошим стилем считается использовать элементы label, поскольку они делают более удобной работу с элементами интерфейса.

Элемент `<span>`, содержащий заголовок Todo, можно будет редактировать на месте благодаря атрибуту `contenteditable`. Кроме того, пользователи смогут удалять Todo с помощью кнопки, текстом которой является прописная буква *X*. Чтобы предоставить пользователю обратную связь о выполненных пунктах Todo, мы будем назначать таким элементам класс, стилем которого является зачеркнутый текст.

Ранее, в пункте «Виды», мы обратили внимание на то, что шаблоны содержат заполнители, показывающие, в каких местах нужно подставить значения модели, а также другие структуры, такие как условные выражения.

Ниже показан шаблон для Todo:

```
<script type="text/template" id="todo-template">
  <span class="todo-position"><%- position %></span>.
  <input class="todo-done" type="checkbox"
        <%= done ? checked="checked" : '' %> title="Сделано" />
```

```
  <span class="todo-title <%= done ? 'todo-stroked' : '' %>"
        contenteditable="true"><%- title %></span>
  <button class="todo-delete" title="Удалить">X</button>
</script>
```

Кроме заполнителей, мы использовали условное выражение, проверяющее, выполнена ли данная запись.

Теперь, когда код HTML построен, нужно подключить библиотеки, которые позволят нам запустить приложение.

Установка Backbone.js

У Backbone.js всего одна жесткая зависимость — библиотека Underscore.js (версии 1.7.0 и выше). Это значит, что необходимо подключить эту библиотеку раньше, чем Backbone.js, иначе фреймворк не будет работать. Подключить фреймворк и его зависимость так же легко, как и jQuery, — нужно всего лишь добавить их на страницу, используя тег `<script>`.

Наш проект опирается на следующие библиотеки: jQuery (это все еще книга о jQuery, не так ли?), Backbone.js и ее зависимость Underscore.js, а также адаптер Backbone.localStorage. Чтобы их подключить, мы добавим теги `<script>` со ссылками на них в конец файла `index.html`, перед закрывающим тегом `<body>`, как показано в листинге 15.3.

Листинг 15.3. Подключение библиотек на веб-странице

```
<!DOCTYPE html>
<html>
  <head>
    ...
  </head>
  <body>
    ...
    <script src="js/vendor/jquery-1.11.3.min.js"></script>
    <script src="js/vendor/underscore-min.js"></script>
    <script src="js/vendor/backbone-min.js"></script>
    <script src="js/vendor/backbone.localStorage-min.js"></script>
    <script src="js/app.js"></script>
  </body>
</html>
```

Из кода видно, как просто подключается фреймворк.

ПРИМЕЧАНИЕ

Если вы захотите улучшить наш демонстрационный проект, можете загрузить все библиотеки с помощью Bower и управлять порядком их подключения с помощью RequireJS.

На главной странице фреймворка указано следующее:

> «[...] для совместимости с RESTful, поддержки истории путем Backbone.Router и управления DOM с помощью Backbone.View подключить jQuery; подключить json2.js для поддержки ранних версий Internet Explorer».

Разметка выглядит хорошо, но ведь наше приложение еще ничего не делает. Исправим это, разработав модель проекта.

Модель Todo

Единственная модель в Todos manager описывает единичную запись Todo, представляющую собой некую задачу, которую необходимо выполнить (рис. 15.7). У каждого экземпляра этого объекта есть свойства `title`, `position` и `done`. Вместо того чтобы добавлять их напрямую, как показано в примере в подразделе «Модели», можно разместить их внутри объекта, назначенного свойству с именем `defaults`. В таком случае при создании экземпляра модели любому неопределенному свойству объекта Todo будет присвоено значение по умолчанию. Данный прием необязателен, но его преимущество в том, что у каждого свойства модели гарантированно есть значение по умолчанию.

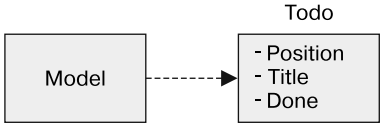


Рис. 15.7. Представление объекта Todo, единственной модели в приложении Todos manager

В нашей модели также будут созданы два метода: `initialize()` и `validate()`. Оба они необязательны, но вы скоро обнаружите, что постоянно используете их в своих проектах. Метод `initialize()` также применяется для коллекций и видов, он выполняется всякий раз при создании нового экземпляра модели. Здесь мы будем отслеживать одно или несколько событий, при наступлении которых будут выполняться соответствующие обработчики.

Метод `validate()` по умолчанию вызывается перед сохранением объекта. Он должен возвращать ошибку в виде строки или объекта в случае неудачи, а при успешном выполнении — ничего не возвращать. В случае неудачи находящаяся в хранилище модель не будет обновлена, независимо от того, хранится ли она на сервере или локально (как в нашем проекте). Этот метод очень важен, поскольку одновременно с возвращением ошибки он генерирует событие `invalid`, которое можно отслеживать и в случае его наступления совершать одно или несколько специальных действий.

В нашем проекте в методе `initialize()` мы будем отслеживать это и другие события и выводить соответствующую информацию в консоль. Таким образом мы сможем наблюдать за тем, что происходит с приложением.

Прежде чем углубиться в код, нужно выполнить простую подготовительную операцию. Читая эту книгу, вы уже поняли, как важно не засорять глобальное пространство имен. Все модели, виды и коллекции, которые мы намерены создать, будут размещены в общем пространстве имен, назовем его `app`. Нашей первой строкой кода будет:

```
window.app = {};
```

Теперь, когда у нас есть пространство имен, посмотрим на код модели Todo, показанный в листинге 15.4.

Листинг 15.4. Модель Todo

```
app.Todo = Backbone.Model.extend({
   defaults: {
      position: 1,
      title: '',
      done: false
   },
   initialize: function() {
      this
         .on('invalid', function(model, error) {
            console.log(error);
         })
         .on('add', function(model, error) {
            console.log(
                'Запись под заголовком "' +
                 model.get('title') + '" создана.'
        );
      })
      .on('remove', function(model, error) {
      {
            console.log(
                'Запись под заголовком "' +
                 model.get('title') + '" удалена.'
        );
      })
      .on('change', function(model, error) {
            console.log(
                'Запись под заголовком "' +
                model.get('title') + '" обновлена.'
        );
      });
   },

  validate: function(attributes) {
     if(!attributes.title) {
        return 'Заголовок не может быть пустым';
   }
```

1. Создаем расширение объекта Backbone.Model
2. Задаем значения по умолчанию для его свойств
3. Добавляем обработчик для события invalid
4. Объявляем метод для валидации свойств

```
    if(
       attributes.position === undefined ||
       parseInt(attributes.position, 10) < 1
    ) {
       return 'Позиция должна быть положительной';
    }
  }
});
```

В начале отслеживания событий мы создаем новый объект, расширяя объект `Backbone.Model` ❶, и присваиваем результат свойству `Todo`, принадлежащему созданному ранее (не показанному в листинге) объекту `window.app`. Мы также определяем объектный литерал, ключами которого являются имена атрибутов, которые хотим создать, а значениями — значения свойств по умолчанию ❷.

Как уже отмечалось, мы также создаем метод `initialize()` и размещаем в нем обработчики нескольких событий, таких как `invalid`. Наконец, переопределяем метод `validate()`, в котором проверяем, не является ли значение `title` ложным (пустая строка, `null`, `undefined` и т. п.), убеждаемся, что значение `position` больше нуля, и в случае необходимости выводим сообщение об ошибке.

Здесь мы построили модель, описывающую записи Todo для нашего приложения. Но мы также планировали объединить модели в коллекцию. Посмотрим, как это делается.

Коллекция записей Todo

Мы намерены объединить модели и представить их как общий список. Для этого воспользуемся коллекцией. Мы создадим ее как расширение объекта `Backbone.Collection` и зададим ее тип, присвоив значение свойству `model`. Нужно также сделать список упорядоченным, поскольку мы хотим выводить его в соответствии с номерами позиций, заданными в каждой записи Todo, так что нам нужно создать блок сравнения, который бы сортировал модели. Для этого мы создадим внутри коллекции свойство `comparator`. В его качестве может выступить метод, определяемый разработчиком, или же строка, определяющая имя атрибута, который будет использоваться для сортировки объектов. Как уже говорилось, мы хотим сортировать записи Todo по атрибуту `position`, поэтому значением свойства `comparator` у нас будет `position`.

Нам нужно, чтобы при создании или удалении пользователем записи Todo в списке каждый раз восстанавливалась правильная последовательность записей. Поэтому нужно отслеживать события `add` и `remove` и каждый раз вызывать функцию, которую мы назовем `collectionChanged`. Данная функция будет отвечать за восстановление правильной последовательности записей. Backbone.js передает созданную или удаленную модель обработчику в виде аргумента. Это важно, поскольку сначала мы будем проверять, является ли модель валидной, с помощью метода `isValid()`, и только затем, если проверка будет пройдена, обновлять позиции остальных моделей. Код, реализующий коллекцию моделей Todo, показан в листинге 15.5.

Листинг 15.5. Коллекция моделей Todo

```
app.todoList = new (Backbone.Collection.extend({
   model: app.Todo,

   localStorage: new Backbone.LocalStorage('todo-list'),

   comparator: 'position',

   initialize: function() {
      this.on('add remove', this.collectionChanged);
   },

   collectionChanged: function(todo) {
      if (todo.isValid()) {
         this.each(function(element, index) {
            element.save({
               position: index + 1
            });
         });
         this.sort();
      }
   }
}));
```

Обозначаем, что это коллекция моделей Todo

Создаем локальное хранилище моделей Todo

Элементы обновляются только в том случае, если модель валидна

В этом подразделе мы описали объекты, используемые для хранения данных и для группировки записей Todo. Но мы до сих пор так ничего и не показали пользователю. Пора заполнить этот пробел, разработав виды.

Виды Todo

В приложении Todos manager два вида: один для отдельных записей Todo, а второй — для всей коллекции. Сейчас мы обсудим первый из них, а второй — несколько позже.

Группа Todos представляет собой список, элементы которого — отдельные записи Todo. В терминах HTML это значит, что у нас есть один элемент `ul` и много элементов `li`. Внутри `<li>` мы будем отображать данные, ассоциированные с моделью `title`, `position` и `done`. Чтобы создать элемент `li`, мы определим свойство `tagName` объекта `view` и, хотя это и не обязательно, свойство `className` — так назначать стили элемента будет проще. Для отображения данных модели мы воспользуемся шаблоном, описанным в пункте «Создание HTML-кода», и переопределим метод `render()`.

Данный вид также отвечает за реакцию на события, касающиеся отдельных записей Todo, таких как удаление и добавление. Чтобы решить эту задачу, мы воспользуемся хешем событий Backbone. Это обычный объект, назначенный свойству вида `events` и содержащий набор пар «ключ — значение». Ключ представлен в виде `имяСобытия селектор`, а значение является именем функции обратного вызова, которая будет запускаться при возникновении события.

Код, реализующий данный вид, представлен в листинге 15.6.

Листинг 15.6. Вид Todo

```
app.TodoView = Backbone.View.extend({
   tagName: 'li',
   className: 'todo',

   template: _.template($('#todo-template').html()),

   events: {
      'blur .todo-position': 'updateTodo',
      'change .todo-done': 'updateTodo',
      'keypress .todo-title': 'updateOnEnter',
      'click .todo-delete': 'deleteTodo'

   },
   initialize: function() {
      this.listenTo(this.model, 'change', this.render);
      this.listenTo(this.model, 'destroy', this.remove);
   },

   deleteTodo: function() {
      this.model.destroy();
   },

   updateTodo: function() {
      this.model.save({
         title: $.trim(this.$title.text()),
         position: parseInt(this.$position.text(), 10),
         done: this.$done.is(':checked')
      });
   },

   updateOnEnter: function(event) {
      if (event.which === 13) {
         this.updateTodo();
      }
   },

   render: function() {
      this.$el.html(this.template(this.model.toJSON()));
      this.$title = this.$('.todo-title');
      this.$position = this.$('.todo-position');
      this.$done = this.$('.todo-done');

   return this;
   }
});
```

❶ Кэшируем шаблон

Хэш событий: ассоциируем события с функциями обратного вызова

❷ Удаляем модель из хранилища

Сохраняем модель

Обновляем модель после нажатия клавиши Enter при редактировании записи Todo

❸ Визуализируем готовый шаблон

В первых строках листинга мы создаем тег для вида, определяем имя класса для этого элемента и кэшируем шаблон. Затем создаем хеш событий ❶, в нем ряд со-

бытий ассоциирован с функциями, которые мы создадим в виде — например, для случая, когда пользователь нажимает кнопку Delete (Удалить), чтобы удалить модель из списка ❷. Наконец, в функции `render()` отображаем созданный ранее шаблон ❸ и возвращаем код HTML, который заменит предыдущее содержимое вида.

Теперь рассмотрим второй вид приложения.

Вид приложения

Вид приложения, `appView`, отвечает за создание новых записей Todo и за отображение их списка.

В отличие от вида Todo, в нашем коде HTML уже есть элемент DOM, к которому будет обращаться этот вид, так что нам не потребуется задавать `tagName` и `className`. Данный элемент — `<div>` с ID `todo-sheet`. Мы сделаем его значением свойства `el` в объекте `appView`. Когда вид будет инициализирован, также потребуется извлечь из памяти список записей Todo и представить его пользователю. Для этого вызовем из метода `initialize()` метод `fetch()`.

С учетом описанных операций единственное событие DOM, которое нас интересует в данном виде, — `click` кнопки Save (Сохранить). При его возникновении мы будем вызывать функцию `createTodo()`, чтобы создать и сохранить новый элемент Todo, записанный пользователем. Это идеальная ситуация для хеша событий. Кроме упомянутого события DOM, наш вид будет отслеживать изменения в списке Todos, чтобы вовремя обновлять код HTML, представляющий этот список. В листинге 15.7 показан код, реализующий все, что здесь обсуждалось.

Листинг 15.7. Вид приложения

```
app.appView = Backbone.View.extend({
   el: '#todo-sheet',

   events: {
      'click #new-todo-save': 'createTodo'
   },

   initialize: function() {
      this.$input = this.$('#new-todo');
      this.$list = this.$('ul.todos');

      this.listenTo(app.todoList, 'reset sort destroy',
                    this.showTodos);
      this.listenTo(app.todoList, 'invalid', this.showError);

      app.todoList.fetch();

   },

   createTodo: function() {
      app.todoList.create(
         {
            title: this.$input.val().trim()
         },
```

❶ Кэшируем список и элемент input

❷ Извлекаем модели из хранилища

❸ Создаем новую запись Todo

```
        {
            at: 0,
            validate: true
        }
    );

    this.$input.val('');
},

showError: function(collection, error, model) {
    this
        .$('.error-message')
        .finish()
        .html(error)
        .fadeIn('slow')
        .delay(2000)
        .fadeOut('slow');
    },

showTodo: function(todo) {
    if (todo.isValid()) {
        var view = new app.TodoView({ model: todo });
        this.$list.prepend(view.render().el);
    }
},

showTodos: function() {
    this.$list.empty();
        var todos = app.todoList.sortBy(function(element) {
            return -1 * parseInt(element.get('position'), 10);
        });
    for(var i = 0; i < todos.length; i++) {
        this.showTodo(todos[i]);
    }
}
});
```

❹ Размещаем запись Todo вверху списка и запускаем ее валидацию

❺ Выводим сообщение об ошибке

❻ Добавляем элемент, только если модель валидна

В методе `initialize()` мы кэшируем элемент `ul`, а также элемент `input`, в котором возможна активность пользователя (редактирование заголовка Todo) ❶. Затем назначаем обработчики для всех интересующих событий. В конце извлекаем модели из локального хранилища ❷.

В методе `createTodo()` мы создаем экземпляр модели Todo, передавая в нее только заголовок и рассчитывая, что остальные свойства получат значения по умолчанию ❸. Затем помещаем ее в начало списка Todos, используя параметр `at`, и запускаем процедуру валидации с помощью параметра `validate` ❹.

В случае обнаружения ошибки функция `showError()` выводит сообщение о ней в элементе с классом `error-message` ❺.

Для построения списка Todos создаем метод `showTodos()` и вспомогательный метод `showTodo()`, отвечающий за отображение отдельной записи. В методе

`showTodos()` сначала, используя метод jQuery `empty()`, проверяем, является ли элемент `ul`, содержащий Todos, пустым. Затем сортируем список в обратном порядке, поскольку хотим, чтобы последний сохраненный элемент находился в голове списка — последняя сделанная запись должна быть вверху, не так ли?

В завершение перебираем отсортированный в обратном порядке список, вызывая метод `showTodo()` и передавая ему в виде аргумента текущую запись Todo. Метод `showTodo()` также проверяет, является ли данная запись Todo валидной ❻. Если запись валидна, то создается соответствующий вид и код вставляется в начало элемента `ul`.

Теперь весь код на месте и можно запустить приложение. Для этого надо написать в конце файла такое выражение:

```
new app.appView();
```

Итак, проект закончен, можно откупоривать шампанское. Готовый код вы найдете в числе других исходных кодов для этой книги в папке `chapter-15/todos-manager`. Чтобы запустить приложение, откройте файл `index.html` в браузере.

Этот раздел, посвященный Backbone.js, всего лишь знакомит с данным фреймворком, и созданное нами приложение очень простое. Но вы, вероятно, заметили, как тесно интегрирована в фреймворк библиотека jQuery и как широко она используется в функциях, которые Backbone.js позволяет строить. Это хорошее подтверждение того, насколько широко применяется функциональность и гибкость jQuery. Мы надеемся, что благодаря представленному краткому введению вы знаете о потенциале Backbone.js больше и достаточно заинтригованы, чтобы двигаться дальше. В качестве заключительного испытания, для проверки ваших знаний, приглашаем вас модифицировать проект и задействовать в нем Bower, RequireJS и QUnit. Наслаждайтесь!

15.6. Резюме

В первой части этой главы мы показали, как можно повысить производительность кода, применяя jQuery и правильно выбирая элементы. Мы обсудили, когда следует пользоваться преимуществами параметра `context` функции `jQuery()`, а когда его лучше не задействовать. Мы также научились избегать универсального селектора. Затем узнали, как добиться лучшей производительности в старых браузерах, создавая селекторы, позволяющие jQuery вызывать собственные функции JavaScript, такие как `getElementById()` и `getElementsByClassName()`.

Ни одна библиотека или фреймворк не являются магическим средством, способным решить все проблемы. Помните: используя сторонние программные продукты, даже такие эффективные, как jQuery, вы до некоторой степени упрощаете себе жизнь, но главная ответственность все равно лежит на вас.

Во второй половине главы мы обсудили, как важно, чтобы весь код приложения был ясно написан и хорошо структурирован. Мы показали, что такое модуль,

и познакомили с некоторыми шаблонами, позволяющими разделить код, написанный с помощью jQuery (и не только), на модули. Среди прочих преимуществ этого подхода — возможность создавать «приватные» переменные и функции, избегая засорения глобального пространства имен.

Мы познакомили вас с инструментом RequireJS — загрузчиком файлов JavaScript и других модулей в различных средах. Эта библиотека освобождает разработчика от бремени самому выбирать порядок подключения модулей, библиотек и фреймворков в соответствии с их зависимостями. В данном разделе мы научились разрабатывать модули с использованием преимуществ RequireJS и показали, как адаптировать к этой библиотеке уже существующий код JavaScript. Мы обратили особое внимание на то, как применять готовый плагин jQuery с RequireJS, не меняя его исходный код, но создав простой файл конфигурации.

Еще один инструмент, описанный в главе, — Bower. Это менеджер пакетов для Сети, который берет на себя управление файлами JavaScript, CSS и другими зависимостями, включая jQuery. Мы показали, как можно находить нужные для приложения пакеты, используя CLI. Затем научились устанавливать, обновлять и удалять пакеты с помощью Bower.

В заключительной части этой главы мы познакомились с Backbone.js, одним из MV*-фреймворков для JavaScript. Backbone.js позволяет создавать одностраничные приложения — тип приложений, широко распространенных сегодня и позволяющих разработчикам свести к минимуму объем серверной части сложных приложений. В сущности, главная часть бизнес-логики написана на JavaScript и находится на клиентской стороне. В разделе, посвященном Backbone.js, мы показали, куда можно двигаться дальше после освоения jQuery. Кроме того, данный фреймворк хорошо интегрирован с jQuery и позволяет создавать прекрасные приложения.

Мы также обсудили архитектуру и главные концепции Backbone.js: модели, маршрутизаторы, виды, шаблоны и коллекции. Затем мы объединили все это и разработали Todos manager — простейшее приложение для хранения списка ежедневных задач.

15.7. Конец

Подумать только! Как много времени прошло с тех пор, как вы начали читать эту книгу! Для нас это был неоценимый опыт — предложить вам лучший из возможных ресурсов, и мы от души надеемся, что достигли цели. Мы уверены: для вас это тоже было незабываемое путешествие, хотя бывали моменты, когда попытки запомнить огромное количество поступающей информации приводили вас в отчаяние. Если вы не можете вспомнить каждую из описанных здесь функций, не беда, в этом нет ничего плохого. Опыт, практика и постоянно открытая в браузере страница документации jQuery помогут справиться с данной проблемой.

Проект jQuery продолжает развиваться. Постоянно выходят новые обновления и дополнения. Одни функции появляются, другие устаревают и даже удаляются — вы могли в этом убедиться, читая об изменениях в jQuery 3. Иногда за новостями трудно успеть, команда jQuery сообщает об обновлениях и устраненных ошибках при каждом следующем выпуске библиотеки и документации.

В нашей книге мы сделали все возможное, чтобы предоставить наиболее свежую информацию о функциях и свойствах jQuery, а также о лучших приемах работы, принятых в веб-сообществе, и о некоторых передовых технологиях программирования.

Надеемся, вы получили удовольствие от книги и не остановитесь в своем обучении. Мы также желаем вам здоровья и счастья, и пусть все ваши ошибки легко исправляются!

Приложение. То, что стоит знать и чего вы, возможно, не знаете о JavaScript!

Это приложение содержит следующую информацию:

- важные концепции JavaScript для эффективного использования jQuery;
- основные сведения об объекте `Object` JavaScript;
- функции как объекты верхнего уровня;
- что такое IIFE;
- что такое `this` и как им управлять;
- что такое замыкания.

Одно из самых больших преимуществ применения jQuery при разработке веб-приложений — возможность запрограммировать значительную часть поведения страницы и не писать при этом весь код с нуля. jQuery самостоятельно реализует все внутренние детали, так что разработчик может сконцентрироваться на непосредственных задачах приложения.

Для понимания примеров, приведенных в первых главах книги, вам было достаточно элементарных навыков программирования на JavaScript. В последующих главах, посвященных обработке событий, анимации и Ajax, для эффективного использования библиотеки jQuery уже требовалось понимать ряд фундаментальных концепций JavaScript. Возможно, тогда вы обнаружили, что какие-то моменты в JavaScript, которые вы считали чем-то само собой разумеющимся (или принимали на веру), начали обретать смысл.

Мы не собираемся подробно рассматривать все концепции JavaScript — книга предназначена не для этого. Цель данного приложения — дать вам представление об основных концепциях JavaScript, которые позволят более эффективно использовать jQuery.

Главная из этих концепций следующая: функции в JavaScript являются объектами верхнего уровня — таков результат специфического механизма создания и обработки функций в данном языке. Понять, что значит для функции быть объектом вообще и объектом верхнего уровня в частности, вы сможете, если понимаете, что такое в принципе объект JavaScript.

Основные сведения об объектах в JavaScript

В большинстве объектно-ориентированных (ОО) языков программирования определен основной тип данных `Object`, от которого наследуются все остальные объекты. В JavaScript базовый `Object` тоже является основой для остальных объектов, но на этом общие черты заканчиваются. По своей природе `Object` JavaScript имеет мало общего с основным типом данных `Object`, определенным в большинстве других ОО-языков.

Сразу после создания `Object` JavaScript не имеет данных и мало что может в отношении семантики. Но даже эта ограниченная семантика имеет большой потенциал. Сейчас мы узнаем, какой именно.

Как создаются объекты

Создать объект в JavaScript можно несколькими способами. Первый из них использует оператор `new` в паре с конструктором `Object`. Создать объект этим способом легко — нужно лишь написать такое выражение:

```
var shinyAndNew = new Object();
```

Его даже можно упростить (и мы скоро покажем вам как), но пока остановимся на этом.

Созданный новый объект пуст. Он не содержит ни информации, ни сложной семантики — вообще ничего. Чтобы он стал нам интересен, надо его чем-нибудь наполнить, а именно свойствами.

Свойства объектов

Объекты JavaScript могут содержать данные и обладать методами (своего рода), которые при необходимости можно добавлять динамически. Рассмотрим такой фрагмент кода:

```
var ride = new Object();
ride.make = 'Yamaha';
ride.model = 'XT660R';
ride.year = 2014;
ride.purchased = new Date(2015, 4, 10);
```

Здесь создается новый экземпляр `Object` и присваивается переменной `ride`. Затем эта переменная наполняется некоторым количеством свойств разных типов: две строки, число и экземпляр объекта `Date`.

ПРИМЕЧАНИЕ

В объекте Date отсчет месяцев начинается с 0: январю соответствует 0, февралю — 1, марту — 2 и т. д.

Не нужно объявлять эти свойства до их присвоения. Они создаются сразу же, как только им присваивается значение. Это необыкновенно эффективное средство, обеспечивающее высокую гибкость. Но за гибкость всегда нужно платить!

Предположим, в следующем коде нужно изменить значение даты покупки:

```
ride.purchased = new Date(2015, 7, 21);
```

Легко! Если только по ошибке не написать так:

```
ride.purcahsed = new Date(2015, 7, 21);
```

Компилятора, способного предупредить вас об ошибке, нет. От вашего имени будет создано новое свойство `purcahsed`, а позже вы будете недоумевать, почему *не принимается* новая дата, когда вы обращаетесь к свойству с правильно написанным именем?

Большие возможности — большая ответственность (где вы слышали это раньше?), так что будьте внимательны, когда пишете код!

На примере видно, что экземпляр JavaScript `Object`, который мы здесь и далее будем называть *объектом*, в действительности представляет собой коллекцию свойств. Каждое из этих свойств состоит из *имени* и *значения*. Имя свойства — строка, а значение может принадлежать к любому типу данных JavaScript: `Number`, `String`, `Boolean`, `Object` и т. п.

Это значит, что главное назначение экземпляра `Object` — служить контейнером для именованной коллекции данных других типов.

Свойством объекта может быть другой экземпляр `Object` со своим набором свойств, которые, в свою очередь, тоже могут быть объектами, и так далее, до любого уровня, имеющего смысл для тех данных, чью модель вы намерены построить.

Предположим, мы создали новое свойство для экземпляра `ride`, которое описывает владельца транспортного средства. Это свойство — еще один объект JavaScript. Он содержит такие свойства, как имя и род занятий владельца:

```
var owner = new Object();
owner.name = 'Спайк Шпигель';
owner.occupation = 'охотник за головами';
ride.owner = owner;
```

Чтобы получить доступ к вложенному свойству, нужно написать так:

```
var ownerName = ride.owner.name;
```

Ограничений по количеству уровней вложенности не существует (кроме тех, что накладываются здравым смыслом). Окончательная — на этот момент — иерархия объекта показана на рис. П1.

Обратите внимание: каждое значение на рисунке является отдельным экземпляром типа JavaScript.

ПРИМЕЧАНИЕ

Вообще говоря, объекты-посредники, подобные owner, не нужны, мы создали такой объект в иллюстративных целях. Скоро вы увидите, что есть более эффективный и компактный способ объявления объектов и их свойств.

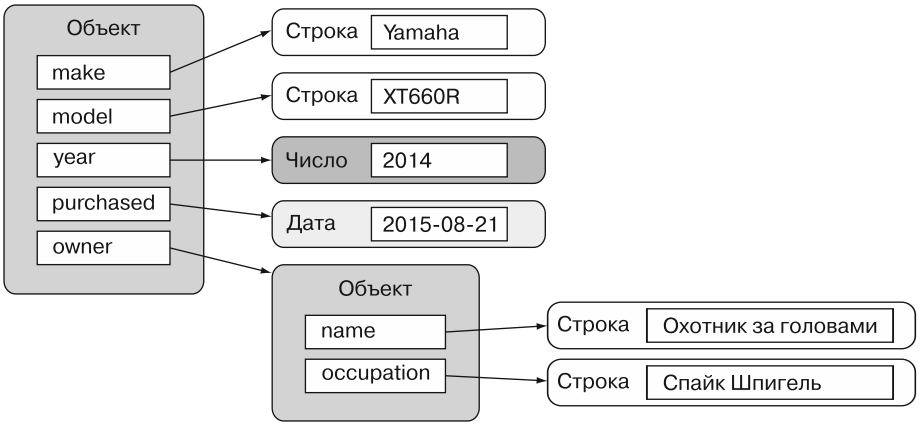


Рис. П1. Как видно по иерархии объекта, экземпляры Object представляют собой контейнеры для именованных ссылок на другие объекты и встроенные типа JavaScript

До сих пор мы обращались к свойствам объекта, используя оператор «точка» (`.`). Теперь подумаем: что произойдет, если создать свойство `color.scheme` (с точкой в середине имени)? В таком случае интерпретатор JavaScript будет искать в объекте `color` вложенное свойство `scheme`.

«Так не будем этого делать!» — скажете вы. А как насчет пробелов? Как быть с другими символами, которые могут быть введены по ошибке и распознаны как разделители, в то время как на самом деле они являются частью имени? И, что более важно, как быть, если вы даже не знаете заранее имени свойства, так как получили его как значение другой переменной или в результате вычисления выражения?

В таких случаях оператор «точка» неприменим. Тогда для доступа к свойствам объекта нужно использовать более общий вариант — оператор *квадратные скобки* (`[]`):

```
object[выражение_имени_свойства]
```

Здесь `выражение_имени_свойства` — выражение JavaScript. Его результатом является строка, соответствующая имени свойства, к которому вы обращаетесь. Например, все три следующих ссылки эквивалентны:

```
ride.make
ride['make']
ride['m' + 'a' + 'k' + 'e']
```

Как и эта запись:

```
var p = 'make';
ride[p];
```

Оператор «квадратные скобки» — единственный способ сослаться на свойства, имена которых не являются валидными идентификаторами JavaScript, например:

```
ride["a property name that's rather odd!"]
```

Этот оператор содержит символы, недопустимые в идентификаторах JavaScript, а также имена, уже использованные для других переменных.

Построение объектов путем создания новых экземпляров с помощью оператора `new` и присвоение значений каждому свойству в отдельности с помощью операторов присваивания — дело довольно утомительное. В следующем разделе мы рассмотрим более компактную и удобную для чтения запись, применяемую для объявления объектов и их свойств.

Объектные литералы

В предыдущем разделе мы создали объект, моделирующий некоторые свойства мотоцикла, и присвоили его переменной `ride`. Для этого мы использовали два оператора `new`, промежуточную переменную `owner` и ряд операторов присваивания. Такой набор действий утомителен и чреват ошибками. Кроме того, при быстром просмотре кода будет трудно визуально понять структуру объекта.

К счастью, существует более компактная и легкая для восприятия запись. Рассмотрим такой оператор:

```
var ride = {
  make: 'Yamaha',
  model: 'XT660R',
  year: 2014,
  purchased: new Date(2015, 7, 21),
  owner: {
    name: 'Спайк Шпигель',
    occupation: 'охотник за головами'
  }
};
```

В этом фрагменте использован *объектный литерал*. Здесь создан такой же объект `ride`, что и раньше, с операторами присваивания, но в виде единой, компактной записи. Большинство разработчиков веб-страниц предпочитают именно такую.

Данная структура очень проста; объект обозначен парой фигурных скобок, внутри которых через запятую перечислены его свойства. Каждое свойство записывается как пара из имени и значения, разделенных двоеточием. Как видите по объявлению свойства `owner`, объявления объектов могут быть вложенными.

Аналогично можно объявлять массивы с помощью *массивов литералов*. Такая запись представляет собой разделенный запятыми список элементов, заключенный в квадратные скобки:

```
var someValues = [2, 3, 5, 7, 11, 13, 17];
```

В примерах, представленных в этом разделе, ссылки на объекты часто хранятся в переменных, которые являются свойствами других объектов. Рассмотрим один особый случай такого подхода.

Объекты как свойства window

До сих пор нам были известны два способа сохранить ссылку на объект JavaScript: в виде переменной и в виде свойства. Как видите, эти два способа подразумевают разную запись:

```
var aVariable = 'Это текст.';
someObject.aProperty = 'Это тоже текст.';
```

В обоих случаях здесь присваивается строка: в первом операторе — переменной, во втором — свойству объекта. Но чем отличаются выполняемые при этом операции? Оказывается, ничем!

Использование ключевого слова `var` на верхнем уровне, вне тела какой-либо функции, — всего лишь удобный для программиста способ записи, обозначающей свойство предопределенного объекта JavaScript `window`. Любая ссылка, созданная на верхнем уровне, неявно указывает на экземпляр `window`. Это значит, что следующие операции, выполняемые на верхнем уровне (вне тела функций), эквивалентны:

```
var foo = 'bar';
window.foo = 'bar';
foo = 'bar';
```

Независимо от выбранного способа записи создается свойство `foo` объекта `window` (если такого свойства еще не было), и ему присваивается значение `bar`. Данная концепция может показаться простой и понятной, но по мере углубления внутрь функций правила видимости становятся более сложными.

На этом мы заканчиваем обзор JavaScript `Object`. Вот главные выводы, которые следуют из обсуждения:

- `object` в JavaScript — это неупорядоченная коллекция свойств;
- свойства представляют собой пары из имени и значения;
- объект можно объявить с помощью объектного литерала;
- массивы можно объявлять с помощью массивов литералов;
- *переменные* верхнего уровня — это свойства объекта `window`.

Теперь обсудим, что мы имели в виду, когда ссылались на функции JavaScript как на *объекты первого класса*.

Функции как объекты первого класса

Во многих традиционных ОО-языках объекты могут содержать данные и методы. В этих языках данные и методы обычно представляют собой разные концепции. Разработчики JavaScript пошли по другому пути.

В JavaScript функции — такие же объекты, как и данные других типов, определенных в JavaScript, таких как `String`, `Number` и `Date`. Подобно другим объектам,

функции создаются с помощью конструктора JavaScript — в данном случае, `Function`. Функцию можно:

- присвоить переменной;
- сделать свойством другого объекта;
- передать другой функции как параметр;
- вернуть как результат другой функции;
- создать, используя литералы.

Поскольку с функциями можно обращаться так же, как и с другими объектами языка, то можно сказать, что функции являются *объектами первого класса*.

В JavaScript функции служат разным целям и создаются разными способами. Узнаем об этом больше.

Функциональные выражения и объявления функций

Может показаться странным, но функции — обычные переменные, которые, кроме того, могут быть вызваны как функции, и скоро мы это докажем. Один из способов определить функцию называется *объявлением функции*. Рассмотрим такой код:

```
function doSomethingWonderful() {
  alert('Делает что-то интересное');
}
```

Объявление функции состоит из ключевого слова `function`, после него стоит имя функции, ее параметры, заключенные в скобки, и тело функции. В приведенном примере мы объявили функцию с именем `doSomethingWonderful`, у которой нет параметров. При вызове функции выполняется ее тело, в данном случае состоящее только из вызова `alert()`. Может показаться, что функция не возвращает значения. Но в JavaScript, если значение не возвращается явно, по умолчанию функция возвращает `undefined`.

Браузеры и имена функций

В браузерах, в которых полностью или же частично реализованы спецификации ECMAScript 6, у функций есть свойство `name`, хранящее имя функции. Подробнее об этом свойстве читайте в статье https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/name.

Как мы уже выяснили, все переменные верхнего уровня — свойства объекта `window`. Объекты `Function` не являются исключением. Если предыдущее объявление функции сделано на верхнем уровне, то при этом создано свойство объекта

window с именем, соответствующим имени функции. Таким образом, следующие выражения эквивалентны:

```
function hello() { alert('Привет!'); }
hello = function hello() { alert('Привет!'); }
window.hello = function hello() { alert('Привет!'); }
```

В браузерах, частично или полностью соответствующих спецификациям ECMAScript 6, можно получить имя функции, обратившись к свойству name самой функции.

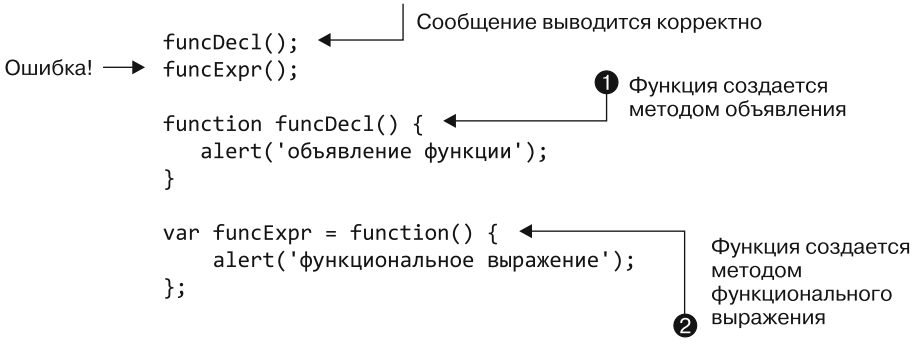
В JavaScript функции могут быть объявлены как часть выражения — такой способ называется *функциональными выражениями*. Значением функционального выражения является объект Function. Рассмотрим следующий код:

```
var myFunc = function() {
  alert('это функция');
};
```

Как видите, здесь создается переменная myFunc, которой присваивается функция. Обратите внимание: поскольку это выражение, то в его конце, после закрывающей фигурной скобки, стоит точка с запятой. У данной функции нет имени (так что значением ее свойства name будет пустая строка), и мы не можем вызвать ее по имени. Однако поскольку мы присвоили ее переменной, то можем выполнить эту функцию так:

```
myFunc();
```

Это не все, чем отличается функциональное выражение от объявления функции. Еще одно важное отличие: объявление функции может быть *«поднято»* (hoisted), а функциональное выражение — нет. Чтобы понять, что это означает на практике, рассмотрим такой пример:



В этом примере создаются две функции: funcDecl() 1 и funcExpr() 2. Но мы пытаемся их вызвать прежде, чем они созданы в действительности. Первый вызов (funcDecl();) успешен, но второй (funcExpr();) приводит к ошибке. Разница в поведении вызвана тем, что funcDecl() «поднимается», а funcExpr() — нет.

Функциональное выражение можно присвоить не только переменной, но и свойству объекта:

```
var myObj = {
  bar: function() {}
};
```

Как видите, функции можно присваивать переменным и свойствам. А как насчет передачи параметров функции? Посмотрим, как это делается и почему.

Функции как обратные вызовы

Когда на странице используются события и таймеры или выполняются запросы Ajax, ее код по своей природе становится асинхронным. Одна из самых распространенных концепций асинхронного программирования — понятие функций *обратного вызова*.

Рассмотрим в качестве примера таймер. Чтобы он сработал через определенное время — допустим, пять секунд, — нужно передать соответствующее значение длительности методу `window.setTimeout()`. Но каким образом таймер узнает, что именно нужно сделать по истечении времени ожидания? Для этого используется функция, которая передается таймеру.

Рассмотрим такой код:

```
function hello() { alert('Привет!'); }
setTimeout(hello, 5000);
```

Мы объявили функцию `hello` и установили таймер на пять секунд, передав ему в качестве второго параметра значение 5000 миллисекунд. В качестве первого параметра метода `setTimeout()` вы передаете ссылку на функцию. Передача функции в качестве параметра ничем не отличается от передачи любого другого значения — например, числа.

По истечении заданного времени будет вызвана функция `hello`. Поскольку метод `setTimeout()` отправляет вызов *обратно* к функции из вашего собственного кода, такая функция называется *функцией обратного вызова*.

Опытным программистам, работающим на JavaScript, такой пример кода может показаться наивным: задавать имя `hello` не обязательно, поскольку данная функция вызывается только один раз. Если вы не планируете вызывать эту функцию в другом месте страницы, то не нужно создавать для нее отдельное свойство `hello` объекта `window`. Если требуется временно сохранить экземпляр `Function`, только чтобы передать его в виде параметра обратного вызова, можно воспользоваться более элегантным решением:

```
setTimeout(function() { alert('Привет!'); }, 5000);
```

Здесь описание функции находится непосредственно в списке параметров (такие функции называются *встроенными анонимными функциями*) и ненужное имя не создается. Такой прием часто встречался в коде jQuery, когда не было необходимости создавать экземпляр функции и назначать его свойству верхнего уровня.

Функции, которые мы создавали в примерах, являются либо функциями верхнего уровня (которые, как мы знаем, есть свойства объекта верхнего уровня `window`), либо параметрами, заданными в функциях вызова. Но экземпляры `Function` могут быть и свойствами объектов. Рассмотрим, как это делается.

Что такое this

В ОО-языках внутри каждого метода автоматически создается ссылка на экземпляр объекта, для которого данный метод вызван. В таких языках, как Java и C#, указателем на текущий экземпляр объекта является переменная `this`. В JavaScript тоже существует аналогичная концепция, и даже используется то же ключевое слово, предоставляющее доступ к связанному с функцией объекту. Но реализация JavaScript несколько отличается от того, как это сделано в других языках.

В ОО-языках, построенных на основе классов, `this` обычно представляет собой ссылку на экземпляр класса, которому принадлежит данный метод. В JavaScript, где функции выступают объектами первого класса и не являются частью чего-либо, `this` — здесь он называется *контекстом функции* — ссылается не на объект, в котором объявлена данная функция, а на объект, для которого она *вызвана*.

Это значит, что у *одной и той же* функции могут быть разные контексты, в зависимости от того, как она вызвана. На первый взгляд такой подход может показаться причудливым, но в действительности он весьма полезен.

По умолчанию контекст (`this`) вызова функции — объект, свойство которого содержит ссылку, использованную для вызова функции. Для демонстрации этой идеи вернемся к нашему примеру с мотоциклом и изменим процедуру создания объекта (дополнения выделены жирным шрифтом):

```
var ride = {
  make: 'Yamaha',
  model: 'XT660R',
  year: 2014,
  purchased: new Date(2015, 7, 21),
  owner: {
    name: 'Спайк Шпигель',
    occupation: 'Охотник за головами'
  },
  whatAmI: function() {
    return this.year + ' ' + this.make + ' ' + this.model;
  }
};
```

К коду первоначального примера добавлено свойство `whatAmI`, которое ссылается на экземпляр объекта `Function`. Новая иерархия объекта, с экземпляром `Function`, назначенным свойству `whatAmI`, показана на рис. П2.

При вызове функции через ссылку на свойство подобно этому:

```
var bike = ride.whatAmI();
```

контекст функции (`this`) указывает на экземпляр объекта, на который ссылается `ride`. В результате переменной `bike` присваивается строка `'2014 Yamaha XT660R'`,

поскольку функция получает свойства объекта, для которого она была вызвана, через `this`.

Все описанное касается и функций верхнего уровня. Напоминаем: эти функции — свойства объекта `window`, так что их функциональным контекстом, когда они вызываются как функции верхнего уровня, является объект `window`.

Несмотря на то что может быть обычным и неявное поведение, JavaScript позволяет явно контролировать то, что используется в качестве контекста функции. Можно присвоить функциональный контекст чему угодно, для чего мы хотим вызывать функцию через методы объекта `Function` — `call()` и `apply()`. Это выглядит безумно, но у функций, как у объектов первого класса, есть методы, определенные конструктором `Function`.

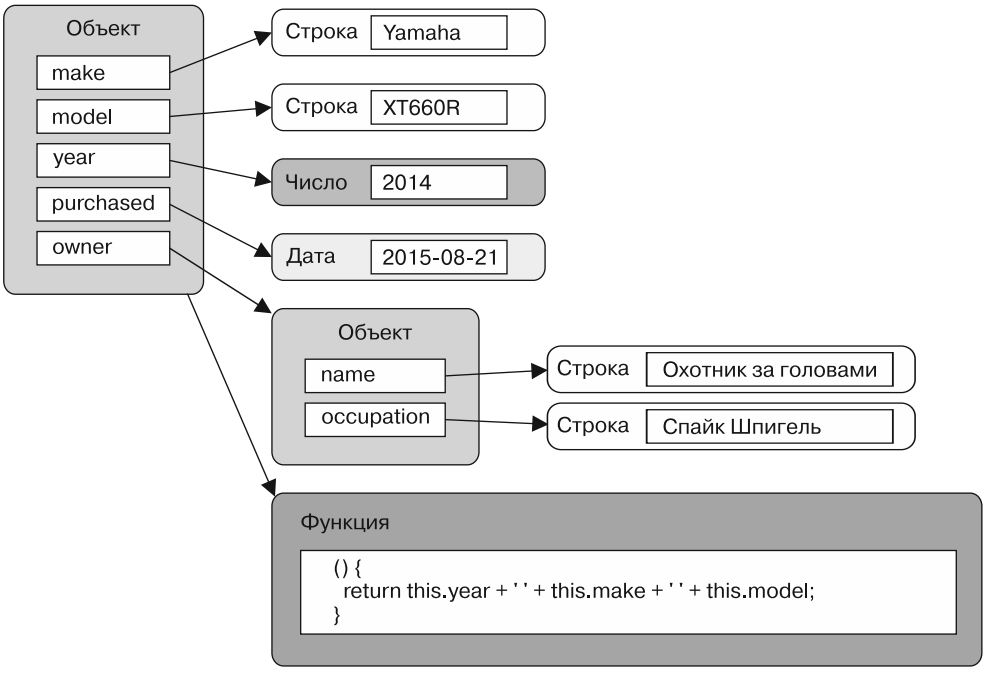


Рис. П2. На этой модели ясно видно, что функция не является частью Object — в свойстве Object под именем whatAmI хранится только ссылка на нее

Метод `call()` вызывает функцию, принимая в качестве первого параметра объект, служащий ее функциональным контекстом. Остальные параметры становятся параметрами вызываемой функции — второй параметр `call()` становится первым аргументом вызываемой функции и т. д. Метод `apply()` работает подобным образом, за исключением того, что его вторым параметром должен быть массив объектов, которые становятся аргументами вызываемой функции.

Чтобы лучше понять эту концепцию, воспользуемся примером. Рассмотрим код листинга П1 (доступен в файле `appendix-a/function.context.html`, предоставленном с книгой, и в JS Bin (http://jsbin.com/dumac/edit?html,js,output)).

Листинг П1. Значение функционального контекста зависит от того, как была вызвана функция

```
<!DOCTYPE html>
<html>
   <head>
      <meta charset="utf-8" />
      <title>Function Context Example</title>
   </head>
   <body>
      <script>
         var obj1 = { handle: 'obj1' };
         var obj2 = { handle: 'obj2' };
         var obj3 = { handle: 'obj3' };
         var value = 'test';
         window.handle = 'window';

         function whoAmI(param) {
            return this.handle + ' ' + param;
         }

         obj1.identifyMe = whoAmI;

         alert(whoAmI(value));
         alert(obj1.identifyMe(value));
         alert(whoAmI.call(obj2, value));
         alert(whoAmI.apply(obj3, [value]));
      </script>
   </body>
</html>
```

1 Определяем три объектных литерала с одним свойством; имена свойств одинаковые, но значения разные

2 Определяем функцию с одним параметром

3 Присваиваем функцию свойству объекта ob1

4 Вызываем функцию whoAmI()

5 Вызываем функцию whoAmI(), используя метод apply()

6 Вызываем функцию whoAmI(), используя ссылку, сохраненную в obj1.indentifyMe

7 Вызываем функцию whoAmI(), используя метод call()

В данном коде мы создали три простых объекта. У каждого из них есть свойство `handle`, по которому легко различать ссылки на эти объекты 1. Мы также создали свойство `handle` для объекта `window`, который теперь тоже легко идентифицировать.

Затем мы создали функцию верхнего уровня, возвращающую значение свойства `handle` объекта, являющегося ее функциональным контекстом 2. Мы присвоили ссылку на *эту же* функцию свойству `identifyMe` объекта `obj1` 3. Можно сказать, что тем самым мы создали для объекта `obj1` метод `identifyMe`, хотя важно отметить: функция была создана независимо от этого объекта.

Наконец, мы вывели четыре сообщения, по одному для разных механизмов вызова одной и той же функции. При загрузке страницы в браузере будет выведена последовательность из четырех сообщений, показанная на рис. П3.

Эта последовательность иллюстрирует следующее.

Если вызвать функцию непосредственно как функцию верхнего уровня, то ее функциональным контекстом является объект `window` 4.

Если вызвать функцию как ссылку на объект (в данном случае `obj1`), то ее функциональным контекстом будет данный объект 5. Можно сказать, что функция ведет себя как *метод* этого объекта, как обычно в ОО-языках. Но не очень-то полагайтесь на эту аналогию. Если не быть внимательными, то она может ввести в заблуждение, как покажут последующие результаты примера.

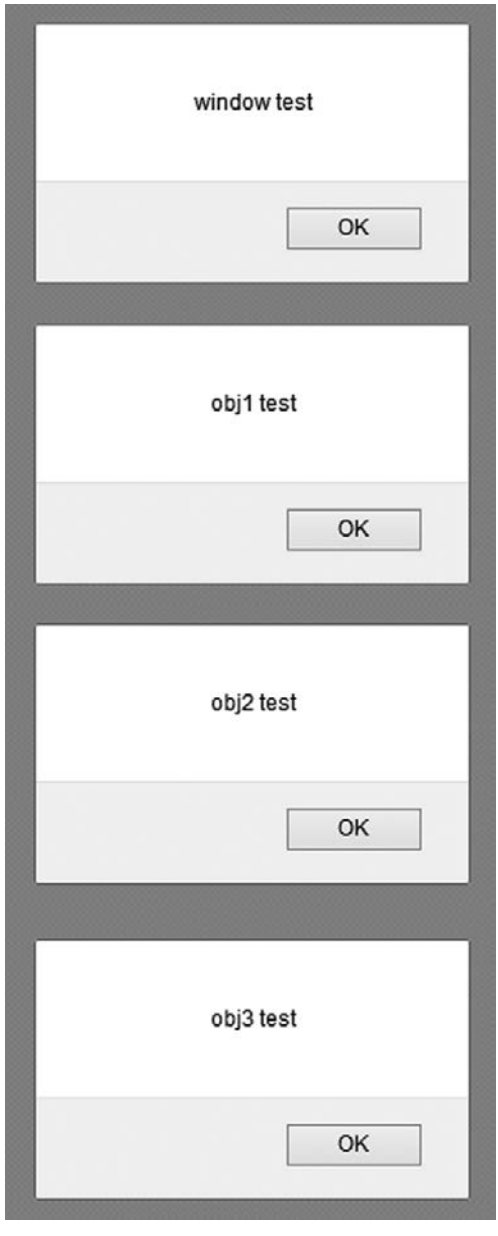


Рис. П3. Объект, играющий роль функционального контекста, меняется в зависимости от способа, которым была вызвана функция

Применение метода `call()` объекта `Function` приводит к тому, что функциональным контекстом становится объект, который был передан в качестве первого параметра `call()`, — в данном случае, `obj2` ❻. В этом примере функция действует

как метод объекта `obj2`, несмотря на то что она никак — даже через свойство — не связана с данным объектом. Здесь также видно, как можно передавать параметры с помощью `call()`.

Как и в случае с `call()`, применение метода `apply()` объекта `Function` приводит к тому, что функциональным контекстом становится объект, переданный в качестве первого параметра (7). Различия между этими двумя методами становятся заметными только тогда, когда функции передаются параметры. При использовании `apply()` все параметры должны быть представлены как элементы единого массива, который передается как второй аргумент.

В примере наглядно показано: функциональный контекст определяется непосредственно при вызове функции и одна и та же функция может быть вызвана для любого объекта, который будет играть роль ее контекста. По этой причине, вероятно, некорректно говорить, что функция является методом объекта. Правильнее говорить так.

> Функция func действует как метод объекта obj в том случае, когда obj является функциональным контекстом при вызове func.

В качестве дальнейшей иллюстрации этой концепции посмотрим, что получится, если добавить к нашему примеру такое выражение:

```
alert(obj1.identifyMe.call(obj3));
```

Несмотря на то что ссылка на функцию является свойством объекта `obj1`, функциональным контекстом при данном вызове является `obj3` — напомним, что функциональный контекст функции зависит не от того, как она объявлена, а от того, как она вызвана.

Теперь, когда вы понимаете, как функции могут играть роль методов объектов, обратим внимание еще на одну тему, касающуюся расширенных возможностей функций и играющую важную роль в эффективном использовании jQuery, — замыкания.

Замыкания

Если максимально все упростить, то *замыкание* — объект `Function` и локальные переменные его окружения, необходимые для его выполнения. После объявления функции у нее появляется возможность обращаться к любым переменным, находящимся в том же пространстве имен, что и точка, в которой была объявлена функция. Такое поведение ожидаемо и не является сюрпризом для любого разработчика, независимо от его опыта. Но в случае с замыканиями эти переменные относятся к функции, даже когда точка ее объявления находится вне пространства имен, замыкая, таким образом, объявление функции.

Возможность обращаться к локальным переменным через обратный вызов функции, после того как они были объявлены, — важный инструмент для написания

эффективного кода на JavaScript. Воспользуемся таймером и приведем наглядный пример в листинге П2. Этот код доступен в файле `appendix-a/closure.html`, предоставленном с книгой, и в JS Bin (http://jsbin.com/jurub/1/edit?html,js,output).

Листинг П2. Замыкания обеспечивают доступ к пространству имен внутри объявления функции

```
<!DOCTYPE html>
<html>
   <head>
      <meta charset="utf-8" />
      <title>Пример замыкания</title>
   </head>
   <body>
      <div id="display"></div>

      <script src="../js/jquery-1.11.1.min.js"></script>
      <script>
         function timer() {
            var local = 1;
            window.setInterval(
               function() {
                  $('#display').append(
                     '<div>В ' + new Date() +
                     ' local=' + local + '</div>'
                  );
                  local++;
               },
               2000
            );
          }

          timer();
      </script>
   </body>
</html>
```

❶ Объявляем элемент, в котором будет выводиться текущее время

❷ Присваиваем переменной local значение 1

❸ Объявляем функцию, которая выполняется каждые две секунды

❹ Увеличиваем значение local на 1

❺ Выполняем функцию timer()

В данном примере мы создали функцию `timer()`, выполняющуюся сразу после объявления ❺. В функции `timer()` объявили локальную переменную `local` ❷ и присвоили ей числовое значение `1`. Затем использовали метод `window.setInterval()`, чтобы установить таймер, который будет срабатывать каждые две секунды ❸. В качестве функции обратного вызова для таймера задали встроенную функцию, которая обращается к переменной `local`. Эта функция показывает текущее время и значение `local`, добавляя элемент `div` к расположенному в теле страницы элементу `display` ❶. В процессе выполнения функции обратного вызова значение переменной `local` также увеличивается на `1` ❹.

Если вы не имели опыта работы с замыканиями, то могли подумать вот о чем: поскольку функция обратного вызова вызывается через две секунды после вызова

функции `timer()`, во время выполнения функции обратного вызова значение `local` будет неопределенным. Но если вы загрузите страницу и оставите ее открытой на некоторое время, то увидите картину, показанную на рис. П4.

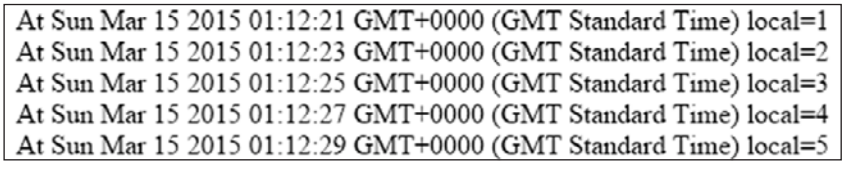


Рис. П4. Замыкания делают переменные окружения доступными для функций обратного вызова, даже если время жизни этого окружения уже истекло

К тому времени, когда сработала функция обратного вызова, время жизни блока, в котором объявлена переменная `local`, истекло. Но замыкание, созданное при объявлении функции, в которой создана `local`, по-прежнему существует, поэтому пример работает.

ПРИМЕЧАНИЕ

Вы могли заметить, что это замыкание, как и вообще замыкания в JavaScript, создано неявно: в отличие от других языков программирования, также поддерживающих замыкания, здесь нет необходимости прибегать к явному синтаксису. Это палка о двух концах: с одной стороны, мы можем легко создавать замыкания (хотим мы этого или нет!), с другой — их потом бывает трудно найти в коде. Случайные замыкания могут приводить к непредсказуемым последствиям. Например, замкнутые сами на себя ссылки приводят к потерям памяти. Классический пример — создание элементов DOM, которые ссылаются на переменные замыкания, не позволяя удалить эти переменные.

Еще одно важное свойство замыканий — функциональный контекст никогда не является частью замыкания. Например, следующий код выполняется не так, как можно было бы ожидать:

```
...
this.id = 'someID';
$('*').each(function() {
  alert(this.id);
});
```

Напомним: каждый вызов функции определяет свой функциональный контекст. Поэтому в приведенном коде функциональный контекст для функции обратного вызова, переданной в `each()`, — элемент из коллекции jQuery (элемент DOM), а не свойство внешней функции, которому присвоено значение `'someID'`. При каждом обращении к функции обратного вызова будет выводиться сообщение с ID очередного объекта из коллекции jQuery.

Если во внутренней функции нужно обратиться к объекту, играющему роль функционального контекста для внешней функции, то следует поступить обычным способом — создать копию ссылки `this` в виде локальной переменной и включить ее в замыкание. Рассмотрим такое изменение нашего примера:

```
this.id = 'someID';
var outer = this;
$('*').each(function() {
  alert(outer.id);
});
```

Переменная `outer` (чаще выбирают имя `that`) становится частью замыкания, поскольку к ней обращается функция обратного вызова, и, следовательно, она из нее доступна. Переменной `outer` присвоена ссылка на объект, который является контекстом за пределами функции обратного вызова. В данном примере, если предыдущий код разместить внутри функции `foo`, переменная `outer` будет ссылаться на функциональный контекст `foo`. Если разместить предыдущий код на HTML-странице вне какой-либо функции, то переменная `outer` будет ссылаться на объект `window`.

После этих изменений код будет показывать сообщение, содержащее строку `'someID'` независимо от того, сколько раз и для каких элементов коллекции jQuery он будет вызван.

Вы скоро поймете, что замыкания незаменимы для создания элегантного кода с использованием команд jQuery, применяющих асинхронный обратный вызов, особенно когда речь идет о запросах Ajax и обработке событий.

Прежде чем закончить приложение, мы намерены обсудить еще одну концепцию, которую активно применяли в этой книге: *функциональные выражения немедленного вызова* (Immediately-Invoked Function Expression, IIFE).

Функциональные выражения немедленного вызова

Так называется шаблон программирования на JavaScript, подразумевающий создание функционального выражения, которое сразу же запускает выполнение функции. Этот термин придумал Бен Алман в статье с одноименным названием (http://benalman.com/news/2010/11/immediately-invoked-function-expression/).

Прежде чем обсудить преимущества использования IIFE, посмотрим, как они реализуются:

```
(function() {
  // Код функции...
})();
```

В данном коде создается анонимная функция, которая сразу же и выполняется благодаря оператору `()` в конце. Функция заключена в скобки — так мы даем понять синтаксическому анализатору, что это функциональное выражение, а не объявление функции.

Шаблон программирования IIFE удобен в нескольких случаях. Его можно использовать для создания «приватных» переменных и функций, недоступных за

пределами функции. Еще одно преимущество: создавая такие «приватные» переменные и функции, мы не засоряем глобальное пространство имен.

Применяя данный шаблон, можно передавать аргументы функции так же, как если бы это была обычная функция. Например, можно написать следующее:

```
var i = 10;
(function(index) {
  // Код функции...
})(i);
```

В этом примере мы объявили переменную `i` и передали ее в IIFE. Внутри функции можно выполнять со значением `i` любые операции, обращаясь к ней как к параметру `index`.

Функциональные выражения немедленного вызова часто используются, когда надо получить доступ к переменным из внешнего пространства имен внутри обработчика события. Предположим, на странице есть три кнопки:

```
<button id="button-1">Кнопка 1</button>
<button id="button-2">Кнопка 2</button>
<button id="button-3">Кнопка 3</button>
```

Мы хотим назначить обработчик каждой из них, чтобы при нажатии выводился индекс соответствующей кнопки (`1` для `button-1`, `2` для `button-2`, `3` для `button-3`). На первый взгляд кажется, задачу решит следующий код:

```
for (var i = 1; i <= 3; i++) {
  document.getElementById('button-' + i).addEventListener(
    'click',
    function() { alert(i); }
  );
}
```

К сожалению, он не работает так, как ожидалось. Независимо от того, какая кнопка нажата, будет всегда выводиться номер 3. Дело в том, что к моменту срабатывания обратного вызова цикл `for` уже завершил работу и переменная `i`, доступная через замыкание, имеет значение `3`.

Чтобы решить эту проблему, можно использовать IIFE:

```
for (var i = 1; i <= 3; i++) {
  (function(index) {
    document.getElementById('button-' + index).addEventListener(
      'click',
      function() { alert(index); });
  })(i);
}
```

В данном коде на каждой из трех итераций цикла создается новое замыкание. Благодаря этому каждая функция получает собственное значение параметра `index`, которому, в свою очередь, присваивается значение переменной `i`.

Как видите, шаблон IIFE чрезвычайно полезен. Мы уверены, что вы будете часто применять его в своем коде.

Резюме

JavaScript — это язык, широко используемый в веб-программировании. К сожалению, авторы многих страниц хоть и задействуют его, но знают недостаточно *глубоко*. В данном приложении мы представили ряд аспектов языка, которые необходимо знать, чтобы эффективно работать с jQuery.

Если у вас есть опыт объектно-ориентированного программирования, то думать об экземпляре `Object` как о неупорядоченном наборе пар «имя — значение» может показаться немного непривычным по сравнению с вашими обычными представлениями об объекте. Но это важная концепция, которую необходимо понимать, если вы пишете на JavaScript даже не очень сложный код.

Функции в JavaScript являются объектами первого класса. Их можно объявлять и ссылаться на них так же, как на объекты других типов. Создавать функции можно путем их объявления или создания функциональных выражений, которые затем присваиваются переменным или свойствам объектов или даже передаются другим функциям как параметры для обратного вызова.

Понятие «функциональный контекст» описывает объект, на который указывает переменная `this` при выполнении функции. Функция может быть создана в расчете на то, чтобы работать как метод объекта, когда этот объект передается ей в виде функционального контекста. Однако функцию нельзя объявить как метод конкретного объекта. Функциональный контекст определяется способом вызова функции (возможно, под явным контролем вызывающего объекта) в момент вызова.

Мы также узнали, каким образом объявление функции и ее окружение могут формировать замыкание, позволяющее функции, когда она впоследствии будет вызвана, получить доступ к локальным переменным, которые являются частью этого замыкания.

Наконец, мы обсудили шаблон программирования на JavaScript, называемый IIFE. Он позволяет создавать «приватные» переменные и функции, избегая, таким образом, засорения глобального пространства имен. Это полезно при использовании функций обратного вызова во время обработки событий, когда может потребоваться создать замыкание, чтобы гарантировать передачу указанным функциям правильного значения переменной.

Только прочно усвоив эти понятия, вы будете готовы писать эффективный код на JavaScript, используя на своих страницах jQuery.

Беэр Бибо, Иегуда Кац, Аурелио де Роза

jQuery в действии. 3-е издание

Перевел с английского *А. Тумаркин*

Заведующая редакцией	*Ю. Сергиенко*
Руководитель проекта	*О. Сивченко*
Ведущий редактор	*Н. Гринчик*
Литературный редактор	*Н. Хлебина*
Художественный редактор	*С. Заматевская*
Корректоры	*Т. Курьянович, Е. Павлович*
Верстка	*Г. Блинов*

Изготовлено в России. Изготовитель: ООО «Питер Пресс».
Место нахождения и фактический адрес: 192102, Россия, город Санкт-Петербург,
улица Андреевская, дом 3, литер А, помещение 7Н. Тел.: +78127037373.
Дата изготовления: 07.2017. Наименование: книжная продукция. Срок годности: не ограничен.
Налоговая льгота — общероссийский классификатор продукции ОК 034-2014,
58.11.12 — Книги печатные профессиональные, технические и научные.
Подписано в печать 23.06.17. Формат 70x100/16. Бумага офсетная. Усл. п. л. 42,570. Тираж 1000. Заказ 0000

КНИГА-ПОЧТОЙ

ЗАКАЗАТЬ КНИГИ ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР» МОЖНО ЛЮБЫМ УДОБНЫМ ДЛЯ ВАС СПОСОБОМ:

- на нашем сайте: **www.piter.com**
- по электронной почте: **books@piter.com**
- по телефону: **(812) 703-73-74**

ВЫ МОЖЕТЕ ВЫБРАТЬ ЛЮБОЙ УДОБНЫЙ ДЛЯ ВАС СПОСОБ ОПЛАТЫ:

- Наложенным платежом с оплатой при получении в ближайшем почтовом отделении.
- С помощью банковской карты. Во время заказа вы будете перенаправлены на защищенный сервер нашего оператора, где сможете ввести свои данные для оплаты.
- Электронными деньгами. Мы принимаем к оплате Яндекс.Деньги, Webmoney и Kiwi-кошелек.
- В любом банке, распечатав квитанцию, которая формируется автоматически после совершения вами заказа.

ВЫ МОЖЕТЕ ВЫБРАТЬ ЛЮБОЙ УДОБНЫЙ ДЛЯ ВАС СПОСОБ ДОСТАВКИ:

- Посылки отправляются через «Почту России». Отработанная система позволяет нам организовывать доставку ваших покупок максимально быстро. Дату отправления вашей покупки и дату доставки вам сообщат по e-mail.
- Вы можете оформить курьерскую доставку своего заказа (более подробную информацию можно получить на нашем сайте www.piter.com).
- Можно оформить доставку заказа через почтоматы (адреса почтоматов можно узнать на нашем сайте www.piter.com).

ПРИ ОФОРМЛЕНИИ ЗАКАЗА УКАЖИТЕ:

- фамилию, имя, отчество, телефон, e-mail;
- почтовый индекс, регион, район, населенный пункт, улицу, дом, корпус, квартиру;
- название книги, автора, количество заказываемых экземпляров.

БЕСПЛАТНАЯ ДОСТАВКА:

- курьером по Москве и Санкт-Петербургу при заказе на сумму **от 2000 руб.**
- почтой России при предварительной оплате заказа на сумму **от 2000 руб.**

ИЗДАТЕЛЬСКИЙ ДОМ ПИТЕР®
WWW.PITER.COM

ИЗДАТЕЛЬСКИЙ ДОМ «ПИТЕР» предлагает профессиональную, популярную и детскую развивающую литературу

Заказать книги оптом можно в наших представительствах

РОССИЯ

Санкт-Петербург: м. «Выборгская», Б. Сампсониевский пр., д. 29а
тел./факс: (812) 703-73-83, 703-73-72; e-mail: sales@piter.com

Москва: м. «Электрозаводская», Семеновская наб., д. 2/1, стр. 1, 6 этаж
тел./факс: (495) 234-38-15; e-mail: sales@msk.piter.com

Воронеж: тел.: 8 951 861-72-70; e-mail: hitsenko@piter.com

Екатеринбург: ул. Толедова, д. 43а; тел./факс: (343) 378-98-41, 378-98-42;
e-mail: office@ekat.piter.com; skype: ekat.manager2

Нижний Новгород: тел.: 8 930 712-75-13; e-mail: yashny@yandex.ru; skype: yashny1

Ростов-на-Дону: ул. Ульяновская, д. 26
тел./факс: (863) 269-91-22, 269-91-30; e-mail: piter-ug@rostov.piter.com

Самара: ул. Молодогвардейская, д. 33а, офис 223
тел./факс: (846) 277-89-79, 277-89-66; e-mail: pitvolga@mail.ru,
pitvolga@samara-ttk.ru

БЕЛАРУСЬ

Минск: ул. Розы Люксембург, д. 163; тел./факс: +37 517 208-80-01, 208-81-25;
e-mail: og@minsk.piter.com

Издательский дом «Питер» приглашает к сотрудничеству авторов:
тел./факс: (812) 703-73-72, (495) 234-38-15; e-mail: ivanovaa@piter.com
Подробная информация здесь: http://www.piter.com/page/avtoru

Издательский дом «Питер» приглашает к сотрудничеству зарубежных торговых партнеров или посредников, имеющих выход на зарубежный рынок: *тел./факс: (812) 703-73-73; e-mail: sales@piter.com*

Заказ книг для вузов и библиотек:
тел./факс: (812) 703-73-73, доб. 6243; e-mail: uchebnik@piter.com

---

Заказ книг по почте: *на сайте www.piter.com; тел.: (812) 703-73-74, доб. 6216; e-mail: books@piter.com*

Вопросы по продаже электронных книг: *тел.: (812) 703-73-74, доб. 6217; e-mail: kuznetsov@piter.com*