HIGHTECH

БЕР БИБО
ИЕГУДА КАЦ

jQuery

Подробное руководство
по продвинутому JavaScript

СИМВОЛ®

jQuery
in Action

Bear Bibeault, Yehuda Katz

HIGH TECH

jQuery

Подробное руководство по продвинутому JavaScript

Бер Бибо, Иегуда Кац

Санкт-Петербург — Москва
2009

Серия «High tech»

Бер Бибо, Иегуда Кац

jQuery. Подробное руководство по продвинутому JavaScript

Перевод А. Киселева

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Выпускающий редактор	*П. Щеголев*
Научный редактор	*Б. Попов*
Редактор	*Т. Темкина*
Корректор	*С. Минин*
Верстка	*Д. Орлова*

Бибо Б., Кац И.

jQuery. Подробное руководство по продвинутому JavaScript. – Пер. с англ. – СПб.: Символ-Плюс, 2009. – 384 с., ил.

ISBN-10: 5-93286-135-5

ISBN-13: 978-5-93286-135-6

Издание представляет собой введение и справочное руководство по jQuery – мощной платформе для разработки веб-приложений. Подробно описывается, как выполнять обход документов HTML, обрабатывать события, добавлять поддержку технологии Ajax в свои веб-страницы, воспроизводить анимацию и визуальные эффекты. Уникальные «лабораторные страницы» помогут закрепить изучение каждой новой концепции на практических примерах. Рассмотрены вопросы взаимодействия jQuery с другими инструментами и платформами и методы создания модулей расширения для этой библиотеки.

Книга предназначена для разработчиков, знакомых с языком JavaScript и технологией Ajax и стремящихся создавать краткий и понятный программный код. Уникальная способность jQuery составлять «цепочки» из команд позволяет выполнять несколько последовательных операций над элементами страницы, в результате чего код сокращается втрое.

ISBN-10: 5-93286-135-5

ISBN-13: 978-5-93286-135-6

ISBN: 1-933988-35-5 (англ)

Издательство «Символ-Плюс». 199034, Санкт-Петербург, 16 линия, 7,
тел. (812) 324-5353, www.symbol.ru. Лицензия ЛП N 000054 от 25.12.98.
Налоговая льгота – общероссийский классификатор продукции
ОК 005-93, том 2; 953000 – книги и брошюры.
Подписано в печать 30.10.2008. Формат 70x100¹/16. Печать офсетная.
Объем 24 печ. л. Тираж 2000 экз. Заказ № 693
Отпечатано с готовых диапозитивов в ГУП «Типография «Наука»
199034, Санкт-Петербург, 9 линия, 12.

Оглавление

Предисловие

Эта книга о простоте. Зачем веб-разработчику писать сложный программный код объемом с хорошую книгу, если все, что требуется, – это реализовать достаточно простые взаимодействия? Нигде не сказано, что программный код веб-приложений непременно должен быть сложным.

Еще только обдумывая создание jQuery, я решил сосредоточиться на небольшом и простом программном коде, подходящем для всех практических приложений, с которыми веб-разработчики имеют дело каждый день. Прочитав книгу «jQuery in Action», я был приятно удивлен, поскольку увидел в ней яркое проявление принципов, заложенных в библиотеку jQuery.

Благодаря своей практической направленности, выразительности представленного программного кода и удачной структуре книга «jQuery in Action» послужит идеальным источником информации для тех, кто стремится познакомиться с этой библиотекой.

Больше всего в этой книге мне понравилось то внимание, которое проявили Бер (Bear) и Иегуда (Yehuda) к деталям внутренних механизмов библиотеки. Они очень тщательно исследовали и описали jQuery API. Ежедневно я бывал удостоен электронного письма или мгновенного сообщения с просьбой пояснить что-нибудь, с сообщением о новой ошибке, найденной в библиотеке, или с рекомендациями по ее улучшению. Вы можете быть уверены, что эта книга – это один из самых продуманных и проработанных источников знаний о библиотеке jQuery.

Меня приятно удивило то обстоятельство, что в книге описаны подключаемые модули (plugins), а также тактика и теория, лежащие в основе разработки модулей jQuery. Причина неизменной простоты jQuery заключается в ее модульной архитектуре. Библиотека предоставляет множество документированных точек расширения, позволяющих наращивать ее функциональность путем подключения к ним модулей. Добавляемые функциональные возможности, хотя и полезны, но зачастую недостаточно универсальны, чтобы включить их в саму библиотеку jQuery, поэтому и важна модульная архитектура. Некоторые из подключаемых модулей, например Forms, Dimension и LiveQuery, получили очень широкое распространение по вполне очевидным причинам: они грамотно написаны, хорошо документированы и обеспечены технической поддержкой. Обязательно обратите особое внимание

на то, как создаются и как используются модули, поскольку на них базируется jQuery.

С такими источниками информации, как эта книга, проект jQuery будет и дальше успешно развиваться. Раз уж вы решились изучать и применять jQuery, надеюсь, эта книга будет полезна и вам.

Джон Ресиг (John Resig),
создатель библиотеки jQuery

Введение

Один из авторов книги – седой ветеран, начавший программировать, когда FORTRAN стал бомбой, а второй – не по годам сообразительный специалист в данной области, не заставший то время, когда не было Интернета. Что объединило этих двух человек, с таким разным жизненным опытом, для совместной работы над книгой?

Ответ очевиден – библиотека *jQuery*.

Мы пришли к этому, одному из наиболее интересных инструментов создания клиентских приложений путями разными, как день и ночь.

Я (Бер), впервые услышал о jQuery, когда работал над книгой «Ajax in Practice»[1]. Заключительный, и самый лихорадочный, этап работы над книгой называется *редактированием* – кроме всего прочего, технический редактор проверяет грамматическую правильность текста и ясность изложения информации. Это самый напряженный, по крайней мере для меня, период, когда меньше всего мне хотелось бы услышать: «Определенно, здесь нужен еще один раздел».

Одна из глав, написанных мной для книги «Ajax in Practice», рассматривает несколько библиотек поддержки технологии Ajax на стороне клиента, с одной из которых я уже был очень близко знаком (Prototype), а с другими (Dojo Toolkit и DWR) мне пришлось познакомиться очень быстро.

Жонглируя множеством задач (все время оставаясь в курсе дел, руководя своим бизнесом и решая семейные проблемы), технический редактор Валентин Греттаз (Valentin Crettaz) случайно обронил: «А почему у вас нет раздела о jQuery?»

«О джей чём?», – спросил я.

И тут же прослушал лекцию о том, как прекрасна эта совершенно новая библиотека и что ее непременно следует включать в любое современное исследование библиотек, обеспечивающих поддержку технологии Ajax на стороне клиента. Я поспрашивал вокруг – не слыхал ли кто о библиотеке jQuery?

---

[1] Дейв Крейн, Бер Бибо, Джордон Сонневельд «Ajax на практике». – Пер. с англ. – К.: Вильямс, 2008.

Многие ответили положительно, с восторгом подтвердив, что jQuery действительно превосходная библиотека. Дождливым воскресным днем я провел четыре часа на сайте jQuery, изучая документацию и пробуя писать маленькие тестовые программы. Затем я написал новый раздел и отправил его техническому редактору, чтобы убедиться в том, что правильно понял суть библиотеки.

Раздел был принят на ура, и заключительный этап работы над книгой «Ajax in Practice» продолжился. (Добавлю, что раздел о jQuery был опубликован в электронном журнале «Dr. Dobb's Journal».)

Когда пыль улеглась, на задворках сознания пустила ростки моя суматошная попытка понять суть jQuery. Мне понравилось то, что я увидел во время своего стремительного погружения, и захотелось познакомиться с библиотекой поближе. Я стал использовать jQuery в веб-проектах. То, что я увидел, понравилось мне еще больше. Я пробовал заменять старый программный код предыдущих проектов, чтобы увидеть, насколько проще становятся веб-страницы благодаря jQuery. И мне *действительно* понравился полученный результат.

Восхищенный своим новым открытием и гонимый желанием поделиться им с другими, я послал в издательство Manning предложение написать книгу «jQuery in Action». Разумеется, я должен был их убедить. (Из чувства мести за такой переполох, я предложил своему техническому редактору, с которого все началось, стать техническим редактором и *этой* книги. Клянусь, *это* послужило ему хорошим уроком!)

И тут редактор Майк Стефенс (Mike Stephens) спросил: «А не хотели бы вы работать над книгой вместе с Иегудой Кацем (Yehuda Katz)?»

«С Иегентой кем?», – спросил я...

Иегуда пришел в этот проект совершенно другим путем; он был знаком с библиотекой jQuery еще в те дни, когда она даже не имела номера версии.[1] Наткнувшись на модуль Selectables, он заинтересовался основной библиотекой jQuery. Несколько разочарованный недостатком (в то время) электронной документации, он обыскал различные wiki-ресурсы[2] и основал сайт Visual jQuery (*visualjquery.com*).

---

[1] В оригинале – «имела номер версии», без частицы «не», однако после знакомства с разделом history на сайте jquery.com, я убедился, что библиотека получила первый номер версии почти год спустя после начала разработки. – *Примеч. пер.*

[2] Те, кого интересует происхождение и история wiki-проектов, могут заглянуть, например, сюда: *wiki.org/wiki.cgi?WhatIsWiki* – что такое Wiki, *c2.com/cgi/wiki?WikiWikiWeb* – распространенный термин, *c2.com/cgi/wiki?InformalHistoryOfProgrammingIdeas* – «ты помнишь, как все начиналось...» – *Примеч. науч. ред.*

В скором времени он дал толчок появлению отличной документации, стал оказывать помощь проекту jQuery и следить за модульной архитектурой и экосистемой, попутно пропагандируя jQuery в сообществе пользователей Ruby.

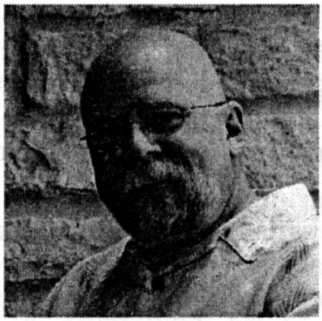
И вот настал день, когда ему позвонили из издательства Manning (издателю его порекомендовал друг), предложив поработать над книгой о jQuery вместе с парнем по имени Бер...

Несмотря на разницу в возрасте, опыте и путях, которыми мы пришли в этот проект, из нас получилась отличная команда и мы прекрасно провели время, работая над книгой. Даже географическая разобщенность (я живу в сердце Техаса, а Иегуда – на побережье Калифорнии) не стала для нас препятствием благодаря электронной почте и системе обмена мгновенными сообщениями!

Думаем, что комбинация наших знаний и талантов позволила сделать добротную и информативную книгу для вас. Надеемся, что, читая ее, вы получите столько же удовольствия, сколько и мы, работая над ней.

Рекомендуем только выбрать наиболее подходящее для чтения время.

Об авторах



Бер Бибо разрабатывает программное обеспечение уже тридцать лет, начав с создания программы Tic-Tac-Toe на суперкомпьютере Control Data Cyber, где в качестве устройства ввода применялся телетайп со скоростью обмена 100 бод. Поскольку у Бера два электротехнических диплома, он должен был бы проектировать антенны или что-то вроде того, но начав работать в Digital Equipment Corporation, он все больше тянулся к программированию. Бер успел поработать на Lightbridge Inc., BMC Software, Dragon Systems и даже служил в вооруженных силах США, где обучал солдат-пехотинцев взрывать танки. (Угадайте, чем он больше всего увлекался!) Теперь Бер – разработчик архитектур программных комплексов и технический менеджер в компании, которая занимается разработкой и сопровождением крупных финансовых веб-приложений, обслуживающих бухгалтерию многих компаний из списка Fortune 500.

Кроме повседневной работы, Бер пишет книги, руководит небольшой собственной фирмой, которая создает веб-приложения и предлагает другие электронные услуги (только не видеосъемку свадеб, никакой свадебной видеосъемки!), а также помогает поддерживать порядок на сайте *JavaRanch.com* под псевдонимом sheriff. Когда не сидит за компьютером, Бер любит приготовить *много* еды (чем и объясняется размер его джинсов), увлекается монтажом фото- и видеофильмов, катается на своем мотоцикле Yamaha V-Star и носит футболки с тропическими картинками.

Живет и работает он в городе Остин (Austin), штат Техас, который нежно любит, за исключением аболютно ненормальных водителей.

Иегуда Кац за последние несколько лет поучаствовал в разработке нескольких проектов с открытым исходным кодом. Он не только член основной команды проекта jQuery, но также участник проекта Merb, альтернативы Ruby on Rails (также реализованной на языке Ruby).

Иегуда родился в штате Миннесота, рос в Нью Йорке, а теперь живет в солнечной Калифорнии, в городе Санта-Барбаре. Он занимался разработкой веб-сайтов для New York Times, Allure Magazine, Architectural Digest, Yoga Journal и других известных клиентов. Программирует на таких языках, как Java, Ruby, PHP и JavaScript.

В свободное от работы время помогает поддерживать *VisualjQuery.com* и отвечает на вопросы начинающих пользователей jQuery на канале IRC и в официальной почтовой рассылке jQuery.

Благодарности

Вас не удивляет длинный список имен, пробегающих по экрану в конце фильма? Неужели для создания фильма действительно требуется столько людей?

В работе над книгой тоже участвует столько людей, что большинство из вас удивится. Многие должны вложить в книгу свой труд и талант, чтобы у вас в руках оказался этот томик (или электронная книга, которую вы читаете с экрана).

Специалисты издательства Manning усердно трудились вместе с нами, чтобы книга получилась хорошей, и мы благодарны им за это. Без них эта книга не состоялась бы. В «заключительные титры» этой книги попали имена не только нашего издателя Марьян Бас (Marjan Bace) и редактора Майка Стефенса, но и следующих сотрудников: Дуглас Падник (Douglas Pudnick), Андреа Кучер (Andrea Kaucher), Карен Тегтмайер (Karen Tegtmayer), Кэти Тенант (Katie Tenant), Меган Йоки (Megan Yockey), Дотти Марсико (Dottie Marsico), Мэри Пиргис (Mary Piergies), Тиффани Тейлор (Tiffany Taylor), Денис Далинник (Denis Dalinnik), Габриель Добреску (Gabriel Dobrescu) и Рон Томич (Ron Tomich).

Трудно переоценить труд наших рецензентов, которые помогали довести книгу до совершенства, отыскивали опечатки, исправляли ошибки в терминологии и в программном коде и даже помогли организовать структуру глав в книге. Каждый цикл рецензирования существенно улучшал книгу. За время, потраченное на рецензирование книги, мы хотим поблагодарить Джонатана Блумера (Jonathan Bloomer), Валентина Греттаза, Дениса Куриленко (Denis Kurilenko), Раму Кришна Вавилала (Rama Krishna Vavilala), Филипа Халлсторма (Philip Hallstrom), Джея Бланчарда (Jay Blanchard), Джеффа Каннингхема (Jeff Cunningham), Эрика Паскарело (Eric Pascarello), Джоша Хейера (Josh Heyer), Грегга Болингера (Gregg Bolinger), Эндрю Симера (Andrew Siemer), Джона Тайлера (John Tyler) и Теда Годдарда (Ted Goddard).

Отдельное спасибо Валентину Греттазу, техническому редактору книги. Помимо проверки всех примеров программного кода в разных окружениях он предложил массу ценных рекомендаций по повышению технической точности текста.

Бер Бибо (Bear Bibeault)

Это моя третья публикация, и список тех, кого я хотел бы поблагодарить, включая всех членов и персонал *javaranch.com*, довольно велик. Без моей причастности к проекту JavaRanch я никогда не смог бы начать писать и поэтому я искренне благодарен Полу Уитону (Pol Wheaton) и Кэти Сиерра (Kathy Sierra), с кого все это началось, а также другим штатным сотрудникам, которые подбадривали и поддерживали меня, включая (как минимум) Эрика Паскарелло (Eric Pascarello), Бена Соузера (Ben Souther), Эрнеста Фридмана Хилла (Ernest Friedman Hill), Марка Хершберга (Mark Herschberg) и Макса Хаббиби (Max Habbibi).

Выражаю свою благодарность Валентину Греттазу – он не только был техническим редактором, но и первым ввел меня в мир jQuery, а также моему коллеге Дэниелу Хедрику (Daniel Hedrick), который добровольно предлагал мне примеры программного кода на языке PHP для последней части книги.

Мой партнер Джей (Jay) и собаки – Литл Бер (не могли же мы назвать его просто *Бером*?) и Козмо, – спасибо вам за ваше незаметное присутствие и за то, что делили со мной крышу над головой, почти не задевая клавиатуру MacBook Pro в течение всех месяцев, пока я работал над этой книгой.

Наконец, я хочу поблагодарить моего соавтора Иегуду Каца, без которого этот проект не смог бы воплотиться.

Иегуда Кац (Yehuda Katz)

Для начала я хотел бы выразить свою благодарность моему соавтору Беру Бибо за его огромный опыт в работе над книгами. Рождению этой книги в огромной мере способствовали его писательский талант и удивительная способность преодолевать препятствия, возникающие на пути профессиональной публикации.

Заговорив о тех, благодаря кому все стало возможным, считаю своим долгом поблагодарить мою любимую жену Леа (Leah), которая проводила без меня столько долгих ночей и рабочих выходных, что заранее я не решился бы попросить об этом. Ее вклад в эту книгу сравним с моим собственным; как всегда, благодаря ей я смог перенести самый сложный этап проекта. Я люблю тебя, Леа.

Совершенно очевидно, что без библиотеки jQuery не было бы и книги «jQuery in Action». Я хочу выразить свою благодарность создателю jQuery Джону Ресигу за то, что он изменил процесс разработки клиентских сценариев, сняв гору с плеч веб-разработчиков всей планеты (видите ли, у нас есть довольно большие группы разработчиков в Китае, Японии, Франции и многих других странах). Я также считаю его своим другом и талантливым автором, помогавшим мне подготовиться к выполнению моих тяжелых обязательств.

Никакой библиотеки jQuery не было бы без огромного сообщества пользователей и членов основной команды, включая разработчиков Брендона Аарона (Brandon Aaron) и Йерна Заффферера (Jörn Zaefferer), популяризаторов Рея Банго (Rey Bango) и Карла Шведберга (Karl Swedberg), Пауля Бакауса (Paul Bakaus), возглавляющего проект jQuery UI, а также Клауса Хартла (Klaus Hartl) и Майка Алсупа (Mike Alsup), работающих в команде разработки модулей вместе со мной. Эта большая группа программистов помогала продвигать платформу jQuery от простого набора базовых операций до библиотеки JavaScript мирового уровня с поддержкой (модульной) практически всего, что только можно пожелать, полностью реализованной усилиями пользователей. Я не привожу полный список тех, кто участвовал в работе, – вас так много! Достаточно сказать, что меня не было бы здесь без уникального сообщества, сплотившегося вокруг этой библиотеки, и у меня не хватает слов, чтобы выразить свою благодарность.

Наконец, я хотел бы поблагодарить свою семью, которую я почти не вижу с тех пор, как начались мои поездки по стране. По мере взросления я и мои братья впитали дух товарищества, а вера семьи в меня всегда придавала мне сил и уверенности. Мама, Никки (Nikki), Эби (Abie) и Иаков (Yaakov): спасибо вам, я люблю вас.

Об этой книге

Делай больше меньшими усилиями.

Эти слова просто и понятно выражают цель этой книги: помочь вам узнать, как наполнить веб-страницы большей функциональностью при меньшем объеме сценариев. Авторы книги, один из которых сотрудник и проповедник (evangelist[1]) jQuery, а другой – увлеченно-восторженный пользователь, полагают, что jQuery – лучшая библиотека из доступных на сегодняшний день, которая позволит вам сделать это.

Эта книга поможет вам быстро и эффективно овладеть jQuery и, надеемся, сделает это занятие увлекательным. Здесь рассматривается собственно ядро jQuery API, каждый метод прикладного программного интерфейса описан в удобном для восприятия формате, включающем описание входных параметров и возвращаемых значений. В книге присутствуют небольшие примеры использования библиотеки, а для так называемых *важных концепций* мы предусмотрели так называемые *лабораторные страницы* (*lab pages*). Эти забавные проверочные страницы представляют собой прекрасный способ увидеть нюансы использования методов jQuery без необходимости писать много своего программного кода.

Все исходные тексты примеров и лабораторных страниц можно загрузить с сайта *www.manning.com/bibeault*.

Мы могли бы еще долго рассказывать, как хороша эта книга, подпуская при этом рекламные словечки, но вряд ли кому приятно тратить время на чтение этой чепухи, правда? Чего вам на самом деле хочется – погрузиться в биты и байты, ведь так?

Так что же вас держит? Вперед!

Для кого эта книга

Книга адресована как начинающим, так и опытным веб-разработчикам, желающим взять на себя управление сценариями JavaScript на

---

[1] Наличие таких людей важно для успешного проекта. К сожалению, не всегда «верхний менеджмент» организации понимает это. Но, например, в компании Sun Microsystems специально введен почетный и поощряемый статус Евангелиста – проповедника и проводника философии Java. – *Примеч. науч. ред.*

своих веб-страницах и создавать настоящие, интерактивные, богатые возможностями интернет-приложения без необходимости писать весь клиентский программный код, что обычно требуется при создании приложений на пустом месте.

Пользу из этой книги извлекут все веб-разработчики, стремящиеся создавать с помощью библиотеки jQuery удобные веб-приложения, которые бы восхищали, а не раздражали пользователей.

При чтении некоторых разделов начинающий веб-разработчик может почувствовать себя несмышленышем, но это не должно его останавливать. Мы включили в книгу приложение с описанием основных понятий JavaScript, помогающих освоить весь потенциал jQuery. Поняв основные концепции, такой читатель обнаружит, что сама библиотека jQuery очень дружелюбна к новичкам и при этом достаточно мощна для более опытных веб-разработчиков.

Как новичкам, так и ветеранам разработки веб-приложений полезно включить jQuery в свой арсенал. Мы ручаемся, что эта книга поможет вам быстро изучить все, что для этого нужно.

Структура книги

Книга организована так, чтобы вы смогли овладеть jQuery максимально быстро и эффективно. Она начинается с введения в основы jQuery и сразу переходит к фундаментальным концепциям прикладного программного интерфейса jQuery. После этого мы продемонстрируем вам различные области применения, где jQuery позволяет писать невероятно продуктивный клиентский программный код, от обработки событий до выполнения запросов к серверу с применением технологии Ajax. Затем вашему вниманию будет представлен обзор наиболее популярных расширений jQuery.

В главе 1 мы рассмотрим философию jQuery и то, как она согласуется с такими современными принципами программирования, как «ненавязчивый JavaScript». Мы исследуем причины, по которым было бы желательно использовать jQuery, вкратце рассмотрим принцип ее действия и такие базовые концепции, как обработчики события готовности документа, вспомогательные функции, создание элементов объектной модели документа (Document Object Model, DOM) и расширений для jQuery.

В главе 2 представлена концепция обернутых наборов элементов – базовая концепция jQuery, определяющая принцип ее действия. Вы узнаете, как создавать обернутый набор элементов – коллекцию элементов DOM, участвующих в операции как единое целое, – за счет выбора элементов документа страницы с помощью богатой коллекции мощных *селекторов* jQuery. Вы увидите, как эти селекторы наращивают возможности, предлагающие нам стандартные способы применения каскадных таблиц стилей (CSS).

В главе 3 мы узнаем, как можно использовать обернутые наборы jQuery для управления структурой DOM страницы. Мы рассмотрим вопросы изменения стилей и атрибутов элементов, содержимого элементов, а также перемещения элементов в пределах страницы и изучим принципы работы с элементами формы.

В главе 4 показано, как с помощью jQuery существенно упростить обработку событий в веб-страницах. В конце концов, именно обработка пользовательских событий делает интернет-приложения полнофункциональными, и все, кому уже приходилось иметь дело с запутанными механизмами обработки событий в разных броузерах, порадуются простоте данной задачи с jQuery.

Мир анимации и визуальных эффектов станет предметом обсуждения главы 5. Здесь мы увидим, что jQuery позволяет создавать анимационные эффекты не только просто, но и эффективно, и даже забавно.

В главе 6 мы познакомимся со вспомогательными функциями и флагами, которые jQuery предоставляет в распоряжение не только авторов страниц, но и всех, кто пишет расширения и модули для jQuery.

О том, как пишутся расширения и модули, будет рассказано в главе 7. Мы увидим, насколько просто пишутся расширения для jQuery, – для этого не потребуется писать хитроумный программный код JavaScript или знать устройство jQuery. Поэтому есть смысл всякий программный код многократного использования оформлять в виде расширения для jQuery.

Глава 8 заинтересует вас одной из наиболее важных областей в разработке полнофункциональных интернет-приложений: выполнение запросов Ajax. Здесь мы увидим, насколько проще применять технологию Ajax с помощью jQuery, и как эта библиотека ограждает нас от ловушек, которыми чревато введение поддержки технологии Ajax в страницы, существенно упрощая реализацию наиболее распространенных видов взаимодействий Ajax (таких, как возврат данных в формате JSON[1]).

Наконец, в главе 9 мы познакомимся с наиболее популярными и мощными модулями jQuery и узнаем, где отыскать информацию о множестве подобных модулей расширения. Мы исследуем модули, позволяющие обрабатывать формы, и более мощные возможности Ajax, чем предполагает jQuery, а также модули, позволяющие реализовать механизм перетаскивания (drag-and-drop) на страницах.

В приложении описаны такие ключевые концепции JavaScript, как *контексты функции* и *замыкания* (*closers*), составляющие основу для максимально эффективного использования библиотеки jQuery в наших

[1] JSON (JavaScript Object Notation) – формат обмена данными; основан на подмножестве языка программирования JavaScript, определенного в стандарте ECMA-262 3rd Edition, December 1999 (*json.org/json-ru.html*). – *Примеч. науч. ред.*

страницах. Это приложение предназначено для тех, кто хочет освежить свои знания об этих концепциях.

Соглашения по оформлению примеров программного кода

Программный код в листингах примеров или в тексте выполнен моноширинным шрифтом, как, например, `этот текст`, чтобы отделить его от обычного текста. Имена методов и функций, свойств, элементов XML и атрибутов также выделяются этим шрифтом.

В некоторых случаях первоначальный программный код может быть отформатирован с целью уместить его на книжной странице. Вообще, программный код изначально был написан с учетом ограниченной ширины книжной страницы, но иногда вы можете заметить незначительные различия между оформлением листингов и исходных программных кодов примеров, доступных для загрузки. В нескольких редких случаях, когда длинная строка не может быть отформатирована без изменения ее смысла, в листинге появится маркер продолжения строки. Большая часть листингов сопровождается описанием, в котором подчеркиваются наиболее важные концепции. Часто в листингах присутствуют нумерованные маркеры для ссылок из пояснительного текста.

Загрузка программного кода примеров

Исходный программный код всех приведенных в книге примеров (а также ряд дополнительных примеров, не попавших в книгу) доступен для загрузки на странице *http://www.manning.com/jQueryinAction* или *http://www.manning.com/bibeault*.

Примеры программного кода организованы так, чтобы его проще было использовать с локальным веб-сервером. Распакуйте загруженный архив с примерами в любую папку и назначьте ее корневой папкой документов веб-сервера. Начальная страница, откуда можно запускать примеры, находится в корневой папке, в файле *index.html*.

За исключением примеров для главы 8 и части примеров для главы 9, примерам не требуется локальный веб-сервер, и можно загружать их непосредственно в броузер. Инструкции по установке программного продукта Tomcat – веб-сервера для примеров из главы 8 – приведены в файле *chapter8/tomcat.pdf*.

Все примеры были опробованы в разных типах броузеров, включая Internet Explorer 7, Firefox 2, Opera 9, Safari 2 и Camino 1.5. Примеры также вполне успешно работают в Internet Explorer 6, но при этом могут наблюдаться некоторые проблемы с расположением элементов на странице. Обратите внимание: программный код библиотеки jQuery работает в IE6 безупречно – все проблемы с расположением элементов обусловлены применением стилей CSS. Поскольку большинство читателей этой книги – профессиональные веб-разработчики, предполага-

ется, что у каждого читателя есть несколько разных броузеров, в которых будут опробоваться примеры.[1]

Форум Author Online

Покупка книги «jQuery в действии» дает право свободного доступа к частному форуму, который поддерживается издательством Manning Publications, где можно оставлять свои комментарии о книге, задавать вопросы технического характера и получать помощь от авторов книги и от других пользователей. Чтобы получить доступ к форуму, откройте в броузере страницу *http://www.manning.com/jQueryinAction* или *http://www.manning.com/bibeault*. Здесь вы найдете информацию о том, как попасть на форум после регистрации, какого рода помощь будет доступна, и о правилах поведения в форуме.

Издательство Manning обязуется предоставить своим читателям место встречи, где может завязаться диалог между читателями и авторами книги. Но издательство не гарантирует присутствие всех или части авторов на форуме – их содействие форуму книги остается добровольным (и неоплачиваемым). Рекомендуем задавать такие вопросы, которые могли бы вызвать интерес у авторов!

Форум Author Online и архивы предыдущих обсуждений доступны на веб-сайте издателя.

Об этой серии

Введения, обзоры и примеры в книгах серии «...in Action» комбинируются так, чтобы способствовать изучению *и* запоминанию. Согласно исследованиям в когнитивистике лучше запоминается информация, полученная при наличии заинтересованности в ее получении.

В издательстве Manning нет ни одного специалиста в области когнитивистики, тем не менее, мы убеждены, что для лучшего запоминания процесс обучения должен пройти через стадии исследования, игры и, что довольно интересно, пересказа изученного материала. Люди склонны запоминать новые сведения, которые дает им преподаватель, только активно исследовав их. Мы учимся *в действии*. Квинтэссенцией книг «...in Action» являются примеры. Они побуждают читателя опробовать новый программный код, поиграть с ним и исследовать новые идеи.

Есть и другая, более прозаическая причина такого названия книг: наши читатели – занятые люди. Они читают книги, чтобы выполнить

---

[1] Авторы подготовили примеры с помощью jQuery 1.2.1. Ко времени подготовки данного перевода для загрузки появилась версия 1.2.6 (*http://docs.jquery.com/Downloading_jQuery*). Отличия версий описаны в разделах Release Notes сайта проекта: *http://docs.jquery.com/Release:jQuery 1.2.6*. – *Примеч. науч. ред.*

свою работу или решить проблему. Им требуются книги, позволяющие легко находить информацию и изучать только то, что нужно, и только когда это нужно. Им нужны книги, которые помогут им *в действии*. Книги этой серии предназначены именно для таких читателей.

Об иллюстрации на обложке

Иллюстрация на обложке книги «jQuery in Action» называется «The Watchman» (Дозорный). Она взята из французской книги «Encyclopedie des Voyages» Дж. Г. С. Саво (J. G. St. Saveur), опубликованной в 1796 году. Путешествия для удовольствия были в то время относительно новым явлением, и большой популярностью пользовались разные путеводители, такие как эта книга. Обычного путешественника или того, кто любит путешествовать сидя дома, эта книга знакомила с жителями других стран мира, а также с национальными костюмами и униформой французских солдат, государственных служащих, торговцев и крестьян.

Разнообразные рисунки в «Encyclopedie des Voyages» – яркое свидетельство того, как уникальны и неповторимы были города мира всего 200 лет тому назад. Это было время, когда особенности одежды жителей двух областей, разделенных всего несколькими десятками миль, позволяли безошибочно определить, в какой из двух областей проживает человек. Путеводитель позволяет почувствовать разобщенность людей того времени и любого другого исторического периода, за исключением нынешнего.

С тех пор стиль одежды сильно изменился, и разнообразие, обусловленное разным происхождением, исчезло. Теперь по одежде сложно определить, на каком континенте проживает тот ли иной человек. Пожалуй, с оптимистичной точки зрения, мы пожертвовали культурным и внешним разнообразием в угоду разнообразию личной жизни. Или для более разнообразной и интересной интеллектуальной и технической деятельности.

Мы, издательство Manning, прославляем изобретательность, инициативу и радость работы с компьютерами, воскрешая на обложках наших книг иллюстрации из путеводителя двухвековой давности, отражающие богатое многообразие тогдашней жизни.

В этой главе:

- Для чего нужна библиотека jQuery
- Что такое «ненавязчивый JavaScript»
- Основы jQuery
- jQuery и другие библиотеки JavaScript

1

Введение в jQuery

Большую часть своей жизни считавшийся среди серьезных веб-разработчиков «игрушечным», язык JavaScript обрел былой авторитет на волне интереса к полнофункциональным интернет-приложениям и технологиям Ajax. Языку пришлось очень быстро повзрослеть, так как разработчики клиентских сценариев отказались от приема копирования и вставки программного кода JavaScript в пользу полноценных библиотек, которые позволяют решать сложные проблемы, связанные с различиями между броузерами разных типов, и реализуют новые и улучшенные парадигмы разработки веб-приложений.

Несколько запоздало появившаяся в мире библиотек JavaScript jQuery покорила сообщество веб-разработчиков, быстро завоевав поддержку крупных сайтов, таких как MSNBC, и была высоко оценена такими открытыми проектами, как SourceForge, Trac и Drupal.

В отличие от других инструментов, сконцентрированных на применении сложных методик JavaScript, jQuery стремится изменить представление веб-разработчиков о принципах создания полнофункциональных интернет-приложений. Вместо того чтобы тратить время на жонглирование непростыми возможностями языка JavaScript, разработчики могли бы, применив знание каскадных таблиц стилей (Cascading Style Sheets, CSS), расширенного языка разметки гипертекста (eXtensible Hypertext Markup Language, XHTML) и старого доброго JavaScript, напрямую манипулировать элементами страницы, воплотив мечту о быстрой разработке веб-приложений.

В этой книге мы во всех подробностях рассмотрим, что может предложить jQuery нам как авторам полнофункциональных интернет-приложений для страниц. Для начала узнаем, что в действительности может дать jQuery при разработке веб-страниц.

1.1. Почему jQuery?

Потратив некоторое время на попытки привнести динамическую функциональность в ваши страницы, вы обнаружите, что постоянно следуете одному и тому же шаблону: сначала отбирается элемент или группа элементов, а затем над ними выполняются некоторые действия: вы могли бы скрывать или показывать элементы, добавлять к ним класс CSS, создавать анимационные эффекты или изменять атрибуты.

С обычным JavaScript для решения каждой из задач потребуются десятки строк программного кода. Создатель jQuery разработал свою библиотеку именно для того, чтобы сделать наиболее общие задачи тривиальными. Например, чтобы создать таблицу с разным цветом фона для четных и нечетных строк, дизайнеру потребуется написать до 10 строк кода на языке JavaScript. А вот как тот же эффект достигается с помощью jQuery:

```
$("table tr:nth-child(even)").addClass("striped");
```

Не волнуйтесь, если сейчас эта строка кажется вам странной. Вскоре вы поймете, как она работает, и сами будете использовать короткие, но мощные инструкции jQuery, чтобы добавить живости своим страницам. Давайте коротко рассмотрим, как работает этот фрагмент кода.

Эта инструкция отыскивает все четные строки (элемент `<tr>`) во всех таблицах на странице и добавляет к ним класс CSS `striped`. Применяя желаемый цвет фона к этим строкам через правило CSS класса `striped`, единственная строка JavaScript может улучшить эстетику всей страницы.

В результате таблицы на странице будут выглядеть, как показано на рис. 1.1.

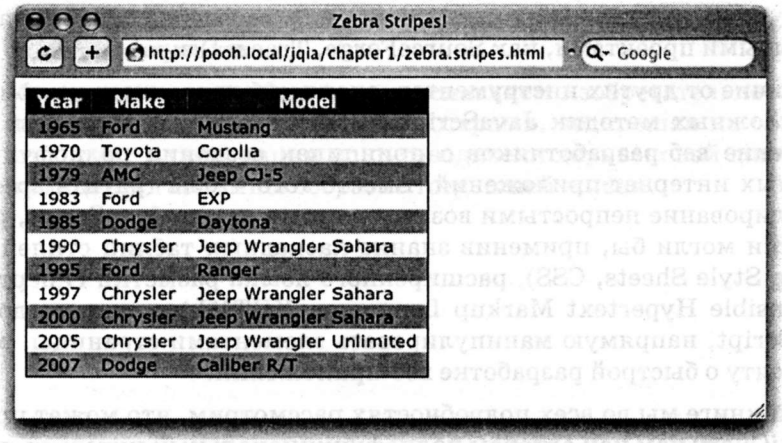


***Рис. 1.1.** Библиотека jQuery позволяет добавить расцветку строк таблицы с помощью единственной инструкции!*

Истинная мощь этой инструкции jQuery заложена в *селекторе* – выражении идентификации элементов, позволяющем идентифицировать и отбирать нужные элементы страницы. В данном случае – каждый четный элемент `<tr>` во всех таблицах. Полный исходный код этой страницы вы найдете среди загружаемых примеров этой книги, в файле *chapter1/zebra.stripes.html*.

Позднее мы узнаем, как легко создавать эти селекторы, но сначала посмотрим, как изобретатели jQuery представляют себе эффективное использование JavaScript в наших страницах.

1.2. Ненавязчивый JavaScript

Помните суровые времена до появления CSS, когда мы были вынуждены смешивать в HTML-страницах стилевую разметку со структурой документа?

Эти воспоминания почти наверняка заставят содрогнуться любого, кто создавал тогда страницы. Добавление CSS в арсенал инструментов веб-разработчика позволяет отделить стилистическую информацию от структуры документа и проводить на заслуженный отдых такие теги, как `<font>`. Отделение стиля от структуры не только упрощает управление документами, но и придает им гибкость полного изменения стиля отображения страницы простой заменой таблицы стилей.

Немногие из нас захотели бы вернуться в прошлое, когда стили отображения применялись непосредственно к HTML-элементам. И все же до сих пор очень популярна разметка вроде этой:

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

Здесь видно, что стиль элемента-кнопки, включая шрифт надписи на ней, определяется не тегом `<font>` или другой нежелательной разметкой стиля, а задается правилами CSS, действующими в пределах страницы. В этом объявлении не смешивается разметка стиля и структура, однако здесь налицо смешение *поведения* и структуры за счет включения кода JavaScript, выполняемого по щелчку на кнопке, в код разметки элемента `<button>` (в данном случае щелчок на кнопке вызывает окрашивание в красный цвет некоторого элемента объектной модели документа (Document Object Model, DOM) с именем `xyz`).

По тем же причинам, по которым желательно отделять стиль от структуры HTML-документа, точно так же (если не более) желательно было бы отделить *поведение* элементов от структуры.

Такого рода отделение известно под названием *ненавязчивый JavaScript*, и разработчики jQuery сосредоточили свои усилия на том, чтобы

помочь авторам страниц следовать ему. С точки зрения концепции ненавязчивого JavaScript и библиотеки jQuery наличие выражений или инструкций на языке JavaScript в теге <body> HTML-страниц, в атрибутах HTML-элементов (например, onclick) либо в блоках сценариев в теле страницы считается неправильным.

Вы можете спросить: «Как же пользоваться кнопкой без атрибута onclick?» Рассмотрим такое определение кнопки:

```
<button type="button" id="testButton">Click Me</button>
```

Оно гораздо проще! Но теперь кнопка просто *ничего не делает*.

Вместо того чтобы встраивать определение поведения кнопки в ее разметку, мы поместим его в блок сценария в разделе <head> страницы, за пределами тела документа, как показано ниже:

```
<script type="text/javascript">
  window.onload = function() {
    document.getElementById('testButton').onclick = makeItRed;
  };

  function makeItRed() {
    document.getElementById('xyz').style.color = 'red';
  }
</script>
```

В обработчике события страницы onload мы связываем функцию makeItRed() с атрибутом onclick элемента-кнопки. Обработчик события onload (а не встроенный код) используется потому, что нам нужно, чтобы элемент-кнопка уже *существовал* к тому моменту, как мы попытаемся манипулировать им. (В разд. 1.3.3 будет показано, что jQuery предоставляет лучший способ размещения такого программного кода.)

Если в этом примере что-то покажется вам непонятным, не пугайтесь! В приложении приведен обзор понятий JavaScript, которые помогут вам эффективно использовать jQuery. Кроме того, в оставшейся части главы мы узнаем, как jQuery позволяет писать код, аналогичный предыдущему, более короткий, более простой и одновременно более гибкий.

Ненавязчивый JavaScript – это мощная методика, позволяющая разделить обязанности в веб-приложениях, но за это приходится платить. Возможно, вы уже обратили внимание, что для достижения поставленной цели нам потребовалось добавить немного больше строк кода, чем в случае, когда код JavaScript помещался непосредственно в код разметки. Ненавязчивый JavaScript не только может привести к увеличению объема программного кода, но также требует определенной дисциплины и применения хорошо зарекомендовавших себя шаблонов программирования клиентских сценариев.

Но в этом нет ничего плохого – все, что побуждает нас с заботой и уважением писать свой клиентский код, обычно размещаемый на сервере, нам только на пользу. Но без jQuery для этого пришлось бы *выполнить* лишнюю работу.

Как уже говорилось, разработчики jQuery сосредоточили свои усилия на том, чтобы упростить нам применение методик ненавязчивого JavaScript при программировании страниц, не платя за это лищним программным кодом. Мы считаем, что эффективное использование jQuery позволяет нам добиться больших возможностей за счет меньшего объема сценариев.

А теперь спокойно рассмотрим, как библиотека jQuery позволяет расширять функциональность страниц без лишних усилий и головной боли.

1.3. Основы jQuery

Суть jQuery в том, чтобы отбирать элементы HTML-страниц и выполнять над ними некоторые операции. Если вы знакомы с CSS, то хорошо понимаете, насколько удобны селекторы, которые описывают группы элементов, объединенные по каким-либо атрибутам или по местоположению в документе. Благодаря jQuery вы сможете, используя свои знания, значительно упростить код JavaScript.

Библиотека jQuery в первую очередь обеспечивает непротиворечивую работу программного кода во всех основных типах броузеров, решая такие сложные проблемы JavaScript, как ожидание загрузки страницы перед тем, как выполнить какие-либо операции.

На тот случай, если в библиотеке обнаружится недостаток функциональности, разработчики предусмотрели простой, но весьма действенный способ ее расширения. Многие начинающие программисты jQuery обнаруживают эту гибкость на практике, расширяя возможности jQuery в первый же день.

Но для начала давайте посмотрим, как знание CSS помогает создать краткий, но мощный программный код.

1.3.1. Обертка jQuery

С введением CSS в веб-технологии для отделения представления от содержимого понадобился способ, позволяющий ссылаться на группы элементов страницы из внешних таблиц стилей. В результате был разработан метод, основанный на использовании *селекторов*, которые представляют элементы на основе их атрибутов или местоположения в HTML-документе.

Например, селектор

```
p a
```

ссылается на все ссылки (элементы <a>), вложенные в элементы <p>. Библиотека jQuery использует те же самые селекторы и поддерживает не только обычные селекторы, применяемые сегодня в CSS, но и другие, еще не полностью реализованные в большинстве броузеров. Селектор

nth-child из примера с полосатой таблицей, приведенного выше, – отличный пример применения селектора, определенного в спецификации CSS3.

Синтаксис отбора группы элементов прост:

```
$(selector)
```

или

```
jQuery(selector)
```

Функция $(), на первый взгляд, необычна, но большинство пользователей быстро начинают применять ее благодаря ее краткости.

Например, чтобы получить группу ссылок, вложенных в элементы <p>, можно использовать следующий код:

```
$("p a")
```

Функция $() (псевдоним функции jQuery()) возвращает специальный объект JavaScript, который содержит массив элементов DOM, соответствующих указанному селектору. У этого объекта много удобных предопределенных методов, способных воздействовать на группу элементов.

На языке программирования такого рода конструкция называется *оберткой* (*wrapper*), потому что она «обертывает» отобранные элементы дополнительной функциональностью. Мы будем использовать термин *обертка jQuery*, или *обернутый набор*, ссылаясь на группы элементов, управлять которыми позволяют методы, определенные в jQuery.

Предположим, нам требуется реализовать постепенное исчезновение всех элементов с классом CSS notLongForThisWorld. jQuery позоляет сделать это так:

```
$("div.notLongForThisWorld").fadeOut();
```

Особенность многих из этих методов, часто называемых *командами* jQuery, состоит в том, что по завершении своих действий (например, действия, обеспечивающего постепенное исчезновение) они возвращают ту же самую группу элементов, готовую к выполнению другой операции. Предположим, что после того, как элементы исчезнут, к ним нужно добавить класс CSS removed. Записать это можно так:

```
$("div.notLongForThisWorld").fadeOut().addClass("removed");
```

Такую *цепочку* (*chain*) команд jQuery можно продолжать до бесконечности. Вы без труда сможете отыскать в Интернете примеры цепочек jQuery, состоящих из десятков команд. А поскольку каждая функция работает сразу со всеми элементами, соответствующими указанному селектору, вам не потребуется выполнять обход массива элементов в цикле. Все нужное происходит за кулисами!

Но даже при том, что группа отобранных элементов представлена довольно сложным объектом JavaScript, в случае необходимости мы мо-

жем обращаться к ней как к обычному массиву элементов. В итоге следующие две инструкции дают идентичные результаты:

```
$("#someElement").html("Добавим немного текста");
```

или

```
$("#someElement")[0].innerHTML =
  "Добавим немного текста";
```

Поскольку здесь мы использовали селектор по атрибуту ID, ему будет соответствовать один элемент. В первом случае используется метод библиотеки jQuery – `html()`, который замещает содержимое элемента DOM некоторой HTML-разметкой. Во втором случае с помощью jQuery извлекается массив элементов, выбирается первый элемент массива с индексом 0 и затем его содержимое замещается с помощью самого обычного метода JavaScript.

Если мы захотим получить тот же результат с помощью селектора, который может отобрать несколько элементов, то можно использовать любой из двух следующих способов, дающих идентичные результаты:

```
$("div.fillMeIn")
  .html("Добавим немного текста в группу элементов");
```

или

```
var elements = $("div.fillMeIn");
for(i=0;i<elements.length;i++)
  elements[i].innerHTML =
    "Добавим немного текста в группу элементов";
```

С увеличением сложности поставленных задач способность jQuery создавать цепочки команд по-прежнему способствует уменьшению числа строк программного кода, необходимого для получения желаемых результатов. Библиотека jQuery поддерживает не только селекторы, которые вы уже знаете и любите, но и более сложные селекторы, определенные как часть спецификации CSS, и даже некоторые нестандартные селекторы.

Вот несколько примеров.

```
$("p:even");
```

Этот селектор отбирает все четные элементы <p>.

```
$("tr:nth-child(1)");
```

Этот селектор отбирает первые строки во всех таблицах.

```
$("body > div");
```

Этот селектор отбирает элементы , являющиеся прямыми потомками элемента <body>.

```
$("a[href$=pdf]");
```

Этот селектор отбирает ссылки на файлы PDF.

```
$("body > div:has(a)")
```

Этот селектор отбирает элементы , которые являются прямыми потомками элемента <body> и содержат ссылки.

Мощная штука!

Для начала вы можете использовать свое знание CSS, а затем изучите более сложные селекторы, поддерживаемые библиотекой. Очень подробно селекторы будут рассматриваться в разд. 2.1, а их полный перечень вы найдете по адресу *http://docs.jquery.com/Selectors*.

Выбор элементов DOM для выполнения операций – это самое обычное дело для наших страниц, но в некоторых случаях требуется выполнить какие-либо действия, никак не связанные с элементами DOM. Давайте коротко рассмотрим, что еще может предложить jQuery помимо манипулирования элементами.

1.3.2. Вспомогательные функции

Даже при том, что обертывание элементов для выполнения операций над ними – основное применение функции $() в библиотеке jQuery, это не единственная ее возможность. Одна из ее дополнительных возможностей – выступать в качестве префикса *пространства имен* (*namespace*) для вспомогательных функций общего назначения. Обертка jQuery, получаемая в результате вызова $() с селектором, предоставляет авторам страниц максимум возможностей, поэтому большинство авторов страниц использует другие возможности, предоставляемые вспомогательными функциями, сравнительно редко. Мы не будем подробно рассматривать эти функции до главы 6, где описаны подготовительные действия для создания подключаемых модулей jQuery. Но некоторые из этих функций *встретятся* вам в следующих разделах, поэтому мы опишем их здесь.

Поначалу способ обращения к этим функциям может казаться немного странным. Давайте рассмотрим пример использования вспомогательной функции, которая усекает строки. Вызов этой функции выглядит так:

```
$.trim(someString);
```

Если префикс $. кажется вам странным, запомните, что $ – это обычный идентификатор JavaScript, такой же, как и любой другой. Вызов той же самой функции с использованием идентификатора jQuery выглядит более знакомым:

```
jQuery.trim(someString);
```

Здесь совершенно очевидно, что функция trim() принадлежит пространству имен jQuery с псевдонимом $.

Примечание

В документации эти элементы называются вспомогательными *функциями*, но совершенно очевидно, что в действительности они являются *методами* функции $(). Не углубляясь в технические детали, мы также будем называть эти методы *вспомогательными функциями*, чтобы терминология соответствовала электронной документации.

Одну такую вспомогательную функцию, позволяющую расширять jQuery, мы рассмотрим в разд. 1.3.5, а другую, позволяющую jQuery мирно сосуществовать с другими клиентскими библиотеками, – в разд. 1.3.6. Но для начала рассмотрим еще один важный аспект функции $.

1.3.3. Обработчик готовности документа

Методика ненавязчивого JavaScript позволяет отделить поведение элементов от структуры документа, поэтому мы будем манипулировать с элементами страницы за пределами разметки документа, где создаются эти элементы. Для этого нам нужен механизм, позволяющий дождаться окончания загрузки элементов DOM страницы и только после этого выполнить необходимые операции. В примере с полосатой таблицей мы должны дождаться момента, когда будет загружена вся таблица, и только потом изменить цвет фона ее строк.

Традиционно для этих целей используется обработчик `onload` объекта `window`, который выполняет инструкции после того, как страница будет загружена целиком. Обычно он вызывается так:

```
window.onload = function() {
  $("table tr:nth-child(even)").addClass("even");
};
```

Тем самым программный код раскрашивания таблицы вызывается только после того, как документ полностью загружен. К сожалению, броузер задерживает выполнение обработчика `onload` не только до момента создания полного дерева DOM, но также ждет, пока будут загружены все изображения и другие внешние ресурсы и страница отобразится в окне броузера. В результате посетитель может заметить задержку между тем моментом, когда он впервые увидит страницу, и тем, когда будет выполнен сценарий `onload`.

Хуже того, если изображение или другой ресурс загружается достаточно долго, посетитель вынужден ждать окончания его загрузки, прежде чем дополнительные особенности поведения элементов станут доступными. Во многих реальных применениях это могло бы обречь идею ненавязчивого JavaScript на неудачу.

Намного лучший подход заключается в том, чтобы задерживать запуск сценариев, обеспечивающих дополнительные особенности поведения, *только* до момента окончания загрузки структуры документа,

когда HTML-код страницы будет преобразован броузером в дерево DOM. Добиться этого независимым от типа броузера способом достаточно сложно, но библиотека jQuery предоставляет простой способ запуска программного кода сразу после загрузки дерева DOM, но до загрузки внешних изображений. Формальный синтаксис определения такого кода (из примера с таблицей):

```
$(document).ready(function() {
  $("table tr:nth-child(even)").addClass("even");
});
```

Сначала производится обертывание экземпляра документа в функцию `jQuery()`, а затем применяется метод `ready()`, которому функция передается для исполнения после того, как документ станет доступным для манипуляций.

Этот синтаксис мы назвали *формальным* потому, что гораздо чаще используется сокращенная форма его записи:

```
$(function() {
  $("table tr:nth-child(even)").addClass("even");
});
```

Вызывая функцию `$()`, мы тем самым предписываем броузеру дождаться, пока дерево DOM (и только дерево DOM) будет полностью загружено, прежде чем выполнить этот код. Более того, в одном и том же HTML-документе мы можем применять этот прием многократно, и броузер выполнит все указанные функции в порядке следования объявлений. Напротив, методика на базе обработчика `onload` объекта `window` позволяет указать единственную функцию. Это ограничение чревато трудноуловимыми ошибками, если какой-либо сторонний сценарий использует механизм `onload` для собственных нужд (что никак нельзя признать хорошей практикой).

Мы познакомились со вторым назначением функции `$()`. Давайте посмотрим, что еще она может нам предложить.

1.3.4. Создание элементов DOM

Совершенно очевидно, что авторы jQuery избегали вводить глобальные идентификаторы в пространство имен JavaScript, сделав функцию `$()` (которая является обычным псевдонимом функции `jQuery()`) достаточно универсальной для выполнения множества операций. Итак, у этой функции есть еще одна особенность, которую мы хотим исследовать.

Передавая функции `$()` строку с кодом разметки HTML-элементов дерева DOM, мы можем создавать эти элементы на лету. Например, так можно создать новый элемент-абзац:

```
$("<p>Hi there!</p>")
```

Но в создании ни с чем не связанных элементов DOM (или иерархии элементов) нет никакого смысла. Обычно после создания иерархии элементов задействуются другие функции jQuery для манипулирования деревом DOM.

Давайте исследуем в качестве примера листинг 1.1.

Листинг 1.1. Создание элементов HTML на лету

```
<html>
  <head>
    <title>Follow me!</title>
    <script type="text/javascript" src="../scripts/jquery-1.2.js">
    </script>
    <script type="text/javascript">
      $(function(){
        $("<p>Hi there!</p>").insertAfter("#followMe");
      });
    </script>
  </head>

  <body>
    <p id="followMe">Follow me!</p>
  </body>
</html>
```

❶ Обработчик готовности документа, создающий элементы HTML

❷ Существующий элемент, за которым будет вставлен новый

В этом примере в теле документа определяется существующий HTML-элемент абзаца с именем `followMe` ❷. В сценарии, который находится в разделе `<head>`, определяется сценарий обработчика события готовности документа ❶. Этот сценарий добавляет вновь создаваемый абзац в дерево DOM сразу вслед за существующим элементом:

```
$("<p>Hi there!</p>").insertAfter("#followMe");
```

Результат показан на рис. 1.2.

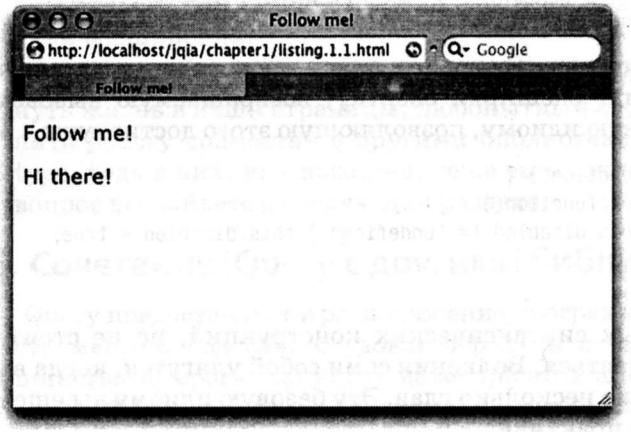


Рис. 1.2. Элемент, созданный динамически и добавленный в документ

Полный комплект функций манипулирования деревом DOM мы рассмотрим в главе 2, где вы увидите, что jQuery предоставляет много функций манипулирования DOM, позволяющих придать документу желаемую структуру.

Теперь, когда вы познакомились с базовым синтаксисом jQuery, давайте взглянем на одну из самых мощных особенностей этой библиотеки.

1.3.5. Расширение jQuery

Функция-обертка jQuery обеспечивает доступ к многим полезным функциям, которые мы постоянно будем использовать в страницах. Но ни одна библиотека не в состоянии удовлетворить все потребности без исключения. Можно даже утверждать, что ни одна библиотека не должна стремиться удовлетворить все возможные потребности, иначе появится огромная неуклюжая масса программного кода, содержащая редко используемые функции, которые только мешают работе!

Понимая это, авторы библиотеки jQuery прилагают усилия для выявления функциональных особенностей, востребованных большинством авторов страниц, чтобы включать в библиотеку только такие особенности. Ввиду того что у каждого автора страниц могут быть собственные неповторимые потребности, библиотека jQuery была создана легко расширяемой.

Но зачем расширять jQuery – ведь для восполнения недостатка функциональности можно писать автономные функции!

Все просто: расширяя библиотеку, мы можем использовать ее мощные особенности, в частности возможность выбора элементов.

Рассмотрим конкретный пример: в библиотеке jQuery отсутствует предопределенная функция, которая деактивировала бы элементы форм. Если во всех приложениях используются формы, пригодилась бы возможность применять, к примеру, такой синтаксис:

```
$("form#myForm input.special").disable();
```

К счастью, архитектура jQuery позволяет легко расширять имеющийся набор функций, расширяя обертку, возвращаемую вызовом $(). Рассмотрим базовую идиому, позволяющую этого достигнуть:

```
$.fn.disable = function() {
  return this.each(function() {
    if (typeof this.disabled != "undefined") this.disabled = true;
  });
}
```

Здесь много новых синтаксических конструкций, но не стоит из-за них сильно волноваться. Волнения сами собой улягутся, когда вы прочитаете следующие несколько глав. Эту базовую идиому вы еще много раз примените на практике.

Во-первых, конструкция `$.fn.disable` означает, что мы расширяем обертку `$` функцией с именем `disable`. Внутри этой функции коллекцию обернутых элементов DOM, над которыми будет выполняться операция, представляет идентификатор `this`.

Во-вторых, метод `each()` в этой обертке вызывается для обхода всех элементов коллекции. Подробнее этот и подобные ему методы рассматриваются в главе 2. Внутри функции, передаваемой методу `each()`, ключевое слово `this` является ссылкой на конкретный элемент DOM в текущей итерации. Пусть вас не смущает тот факт, что внутри вложенной функции ссылка `this` указывает на различные объекты. Когда вы напишете несколько функций расширения, это станет привычным и естественным.

Для каждого элемента выполняется проверка – есть ли у текущего элемента атрибут `disabled`, и если такой атрибут имеется, ему присваивается значение `true`. Результат метода `each()` (обертка) возвращается в вызывающую программу, чтобы обеспечить возможность составления цепочек нашим новым методом `disable()`, как это делают многие методы библиотеки jQuery. После этого можно написать такую инструкцию:

```
$("form#myForm input.special").disable().addClass("moreSpecial");
```

С точки зрения кода нашей страницы все выглядит так, как если бы наш новый метод `disable()` был встроен непосредственно в библиотеку! Этот прием обладает такой мощью, что большинство пользователей, начинающих изучать jQuery, обнаруживают в себе способность создавать небольшие расширения для jQuery, как только начали применять библиотеку.

Более того, наиболее инициативные пользователи расширили возможности jQuery наборами полезных функций, которые называются *подключаемыми модулями*, *модулями расширения* (*plugins*). Мы еще не раз поговорим об этом способе расширения jQuery, а в главе 9 представим вашему вниманию свободно распространяемые официальные модули расширения.

Прежде чем углубиться в тонкости использования jQuery, чтобы вдохнуть жизнь в наши страницы, любопытный спросит, можно ли применять jQuery совместно с другими библиотеками, такими как Prototype, ведь в них, как известно, тоже есть сокращение `$`. Ответ на этот вопрос вы найдете в следующем разделе.

1.3.6. Сочетание jQuery с другими библиотеками

jQuery предоставляет в распоряжение программиста набор мощных инструментов, способных удовлетворить насущные потребности большинства авторов страниц, но даже при этом иногда в странице требуется задействовать возможности нескольких библиотек JavaScript. Такие ситуации могут возникать, например, во время перевода приложения

на использование библиотеки jQuery или когда необходимо использовать в наших страницах другую библиотеку совместно с jQuery.

Разработчики jQuery четко понимают потребности пользователей в своем сообществе и не стремятся блокировать применение других библиотек, создавая все условия для обеспечения мирного сосуществования jQuery с другими библиотеками в ваших страницах.

В первую очередь, по общепринятой рекомендации, они стремятся избегать «загрязнения» глобального пространства имен идентификаторами, которые могут вызывать конфликты не только с другими библиотеками, но, возможно, и с именами из ваших собственных сценариев. Идентификатор `jQuery` и его псевдоним $ – это единственные имена, внедряемые в глобальное пространство имен. Определение вспомогательных функций, о которых говорилось в разд. 1.3.2, как части пространства имен `jQuery` – это прекрасный пример такой заботы.

Весьма маловероятно, чтобы какая-то другая библиотека имела достаточно веские причины объявить глобальный идентификатор `jQuery`, но здесь есть и более удобный идентификатор – псевдоним $. Другие библиотеки JavaScript, в частности библиотека Prototype, используют имя $ для своих собственных целей. А поскольку применение идентификатора $ в таких библиотеках является ключом к их функциональности, это чревато серьезным конфликтом.

Авторы jQuery постарались избежать этого конфликта с помощью вспомогательной функции, которая так и называется `noConflict()`. В любой момент, как только будут загружены конфликтующие библиотеки, можно вызвать функцию

```
jQuery.noConflict();
```

которая установит значение идентификатора $ в соответствие с требованиями библиотеки, отличной от jQuery.

Тонкости использования этой вспомогательной функции мы подробнее рассмотрим в разд. 7.2.

1.4. Итоги

В этом кратком введении в jQuery мы рассмотрели множество подготовительных сведений, чтобы легко и быстро приступить к разработке полнофункциональных интернет-приложений.

Библиотека jQuery будет полезна для любой страницы, которая должна выполнять какие-либо действия кроме простейших операций JavaScript и позволит авторам страниц применять методику ненавязчивого JavaScript. При таком подходе поведение элементов отделяется от структуры документа, так же как CSS позволяет отделить информацию о представлении от структуры, за счет чего улучшается организация страниц и увеличивается гибкость программного кода.

jQuery вводит в пространство имен JavaScript всего два новых идентификатора – функцию `jQuery` и ее псевдоним `$`, тем не менее, через эту функцию библиотека предоставляет массу новых возможностей, что делает ее весьма гибким инструментом. Любая операция, выполняемая этой функцией, основана на ее параметрах.

Как показано выше, функция `jQuery()` служит для:

- отбора и обертывания элементов DOM, участвующих в операции;
- исполнения роли пространства имен для глобальных вспомогательных функций;
- создания элементов DOM из кода разметки HTML;
- определения программного кода, выполняемого после того, как дерево DOM будет готово к манипуляциям.

Библиотека jQuery ведет себя на странице как добропорядочный гражданин, не только минимизируя свое вторжение в глобальное пространство имен, но и обеспечивая возврат официального значения идентификатора `$`, еще больше минимизируя вторжение в глобальное пространство имен в случаях, когда это имя может вызвать конфликт, а именно, когда какая-нибудь другая библиотека, например Prototype, задействует идентификатор `$` для своих нужд. Как вам *такое* отношение к пользователю?

Последнюю версию jQuery можно получить на сайте проекта по адресу *http://jquery.com/*. Версия библиотеки jQuery, с которой тестировался программный код примеров из этой книги (1.2.1), включена в состав загружаемого пакета с примерами.

В следующих главах мы рассмотрим все, что может предложить jQuery нам как авторам полнофункциональных интернет-приложений для страниц. Путешествие, которые мы начнем в следующей главе, покажет, как вдохнуть жизнь в страницы путем манипулирования деревом DOM.

2

В этой главе:

- Отбор элементов для включения в обертку с применением селекторов jQuery
- Создание и размещение новых элементов HTML в дереве DOM
- Манипуляции с обернутым набором элементов

Создание обернутого набора элементов

В предыдущей главе мы рассматривали различные способы применения функции `$()` из библиотеки jQuery. Возможности этой функции широки – от отбора элементов DOM до определения функций, которые должны выполняться только после того, как дерево DOM будет полностью загружено.

В этой главе мы подробнее исследуем, как элементы DOM идентифицируются для выполнения операций, рассмотрев две наиболее мощные и часто используемые возможности функции `$()`: отбор элементов DOM посредством *селекторов* и создание новых элементов DOM.

Множество возможностей, необходимых для реализации полнофункциональных интернет-приложений, достигается за счет манипулирования элементами DOM, составляющими страницы. Но прежде чем ими манипулировать, необходимо их идентифицировать и отобрать. Давайте приступим к подробному изучению тех способов, которыми jQuery позволяет нам определять элементы, которые должны быть отобраны для выполнения манипуляций.

2.1. Отбор элементов для манипуляции

Первое, что необходимо сделать перед использованием практически всех методов jQuery (которые часто называют *командами* jQuery), – это выбрать некоторые элементы страницы для выполнения операции. Иногда набор элементов, которые требуется отобрать, описать достаточно легко, например: «все элементы-абзацы на странице». Но иногда описание выглядит более сложно, например: «все элементы списка, которые имеют класс CSS `listElement` и содержат ссылку».

К счастью, jQuery обеспечивает достаточно мощный синтаксис задания селекторов. Мы можем кратко и элегантно определить практически любой набор элементов. Скорее всего, вы уже поняли, что синтаксис селекторов, в основном, следует уже известному и любимому нами синтаксису CSS, расширяя и дополняя его некоторыми нестандартными методами выбора элементов, что позволяет нам решать как простые, так и сложные задачи.

Чтобы помочь вам изучить принципы выбора элементов, мы включили в состав загружаемого пакета с примерами для этой книги лабораторную страницу Selectors Lab (лаборатория селекторов). Если вы еще не загрузили примеры, сейчас самое время сделать это. Загрузка примеров описана в начале книги.

Эта лабораторная страница позволит вам вводить строки селекторов jQuery и наблюдать (в реальном времени!), какие элементы DOM они отбирают. Лабораторная страница находится в файле *chapter2/lab.selectors.html* среди примеров кода.

В окне броузера эта страница должна выглядеть, как показано на рис. 2.1 (если панели в окне не располагаются в одну линию, возможно, вам следует расширить окно броузера).

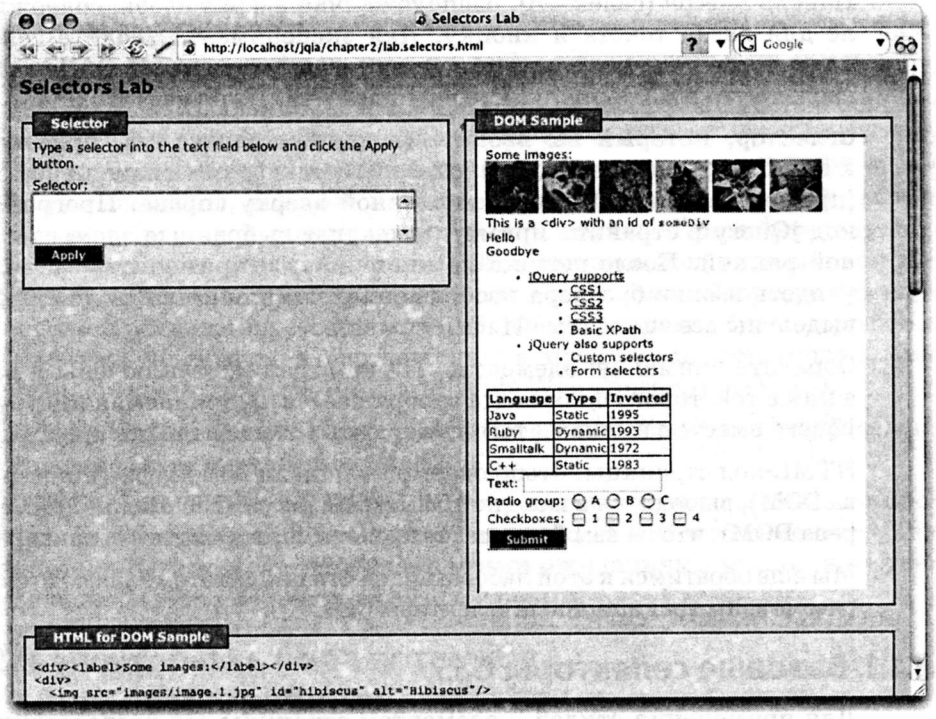


Рис. 2.1. Страница Selectors Lab позволяет исследовать поведение любых селекторов в реальном времени

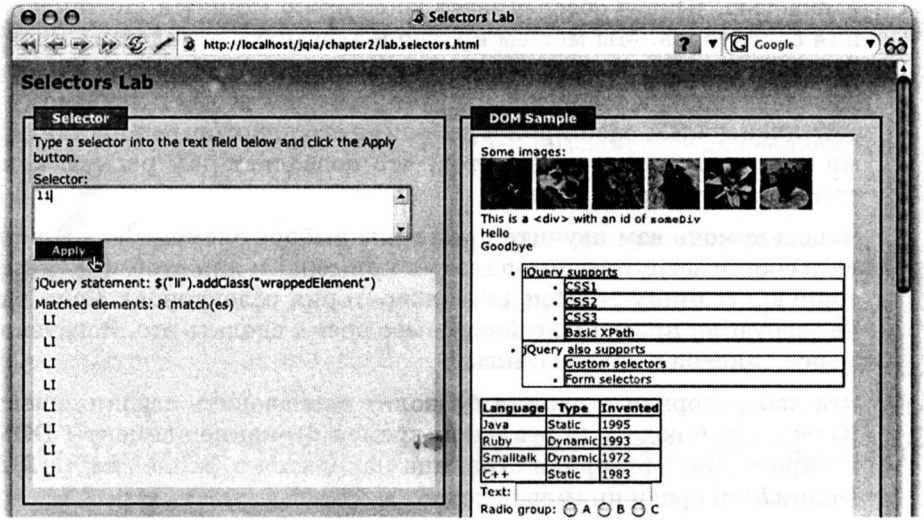


***Рис. 2.2.** Селектору со значением li соответствуют все элементы <li>, как это видно по результату применения этого селектора*

Панель Selector (селектор), расположенная вверху слева, содержит поле для ввода текста и кнопку. Для эксперимента введите селектор в текстовое поле и щелкните на кнопке Apply (применить). Для начала введите строку `li` и щелкните на кнопке Apply (применить).

Селектор, который вы вводите (в данном случае `li`), применяется к HTML-странице, загруженной в элемент `<iframe>` в панели DOM Sample (пример дерева DOM), расположенной вверху справа. Программный код jQuery в странице примера выделяет выбранные элементы красной рамкой. После щелчка на кнопке Apply (применить) вы должны увидеть в окне броузера изображение, как показано на рис. 2.2, где выделены все элементы `<li>` на странице.

Обратите внимание: элементы `<li>` на странице заключены в рамку, а ниже текстового поля ввода отображается выполняемая инструкция jQuery вместе с именами тегов выбранных элементов.

HTML-код страницы, отображаемой в панели DOM Sample (пример дерева DOM), выводится в панели HTML for DOM Sample (HTML для примера дерева DOM), чтобы вам было легче экспериментировать с селекторами.

Мы еще обратимся к этой лабораторной странице в этой главе. Но сначала рассмотрим традиционные селекторы CSS.

2.1.1. Базовые селекторы CSS

Для применения стилей к элементам страницы веб-разработчики используют несколько мощных и удобных методов выбора, работающих во всех типах броузеров. Это методы выбора элементов по идентифика-

тору (ID), имени класса CSS, имени тега и по положению элементов в иерархии дерева DOM.

Ниже приводятся несколько примеров, которые помогут вам освежить память.

- `a` – этому селектору соответствуют все элементы-ссылки (<a>).
- `#specialID` – этому селектору соответствуют элементы, имеющие идентификатор `specialID`.
- `.specialClass` – этому селектору соответствуют элементы с классом CSS `specialClass`.
- `a#specialID.specialClass` – этому селектору соответствуют ссылки с идентификатором `specialID` и классом `specialClass`.
- `p a.specialClass` – этому селектору соответствуют ссылки с классом `specialClass`, объявленные внутри элементов <p>.

Подбирая и смешивая различные базовые типы селекторов, модно отбирать группы элементов с очень высокой степенью точности. Великолепный внешний вид самых необычных и интересных веб-сайтов Интернета создан с применением комбинаций этих базовых возможностей.

Сразу после установки можно использовать jQuery с уже привычными селекторами CSS. Чтобы выбрать элементы с помощью jQuery, нужно обернуть селектор функцией `$()`, например:

```
$("p a.specialClass")
```

За некоторыми исключениями библиотека jQuery полностью совместима с CSS3, поэтому операция выбора элементов не содержит в себе сюрпризов – механизм селекторов библиотеки jQuery отоберет те же самые элементы, которые могли быть отобраны реализацией таблиц стилей в броузерах, совместимых со стандартами. Примечательно, что jQuery не зависит от реализации CSS в броузере, под управлением которого выполняется программный код. Даже если в броузере нет корректной реализации селекторов CSS, jQuery все равно будет корректно выбирать элементы в соответствии с правилами, установленными стандартами консорциума World Wide Web Consortium (W3C).

Для тренировки поэкспериментируйте с различными базовыми селекторами CSS на странице Selectors Lab.

Эти базовые селекторы обладают широкими возможностями, но иногда нам требуется еще более высокая точность выбора элементов. Библиотека jQuery в соответствии с этими ожиданиями расширяет стандартный набор улучшенными селекторами.

2.1.2. Селекторы выбора потомков, контейнеров и атрибутов

В качестве улучшенных селекторов jQuery использует следующее поколение каскадных таблиц стилей (CSS), поддерживаемых Mozilla

Firefox, Internet Explorer 7, Safari и другими современными броузерами. Эти селекторы позволяют выбирать прямых потомков некоторых элементов, элементы, следующие в дереве DOM за заданными, а также элементы, значения атрибутов которых соответствуют определенным условиям.

Иногда требуется выбрать только прямых потомков определенного элемента. Например, таким способом можно выбрать элементы определенного списка, но не элементы вложенных списков. Рассмотрим фрагмент разметки HTML из примера Selectors Lab:

```
<ul class="myList">
  <li><a href="http://jquery.com">jQuery supports</a>
    <ul>
      <li><a href="css1">CSS1</a></li>
      <li><a href="css2">CSS2</a></li>
      <li><a href="css3">CSS3</a></li>
      <li>Basic XPath</li>
    </ul>
  </li>
  <li>jQuery also supports
    <ul>
      <li>Custom selectors</li>
      <li>Form selectors</li>
    </ul>
  </li>
</ul>
```

Предположим, нам нужно выбрать ссылку на удаленный сайт jQuery, но не ссылки на различные локальные страницы с описанием спецификаций CSS. С помощью базовых селекторов можно было бы сконструировать селектор `ul.myList li a`. К сожалению, этот селектор выберет все ссылки, потому что все они входят в состав списка.

Попробуйте ввести на странице Selectors Lab селектор `ul.myList li a` и щелкнуть на кнопке Apply. Результат должен получиться таким, как показано на рис. 2.3.

Улучшенный вариант заключается в использовании *селектора потомков*, в котором родительский элемент и его прямой потомок разделены правой угловой скобкой (>):

```
p > a
```

Этому селектору соответствуют только те ссылки, которые являются *прямыми* потомками элемента <p>. Если ссылка вложена глубже, например находится внутри элемента , вложенного в элемент <p>, она не будет выбрана.

Вернемся к примеру и рассмотрим следующий селектор:

```
ul.myList > li > a
```

Этот селектор выберет только ссылки, являющиеся прямыми потомками элементов списка, которые, в свою очередь, являются прямыми

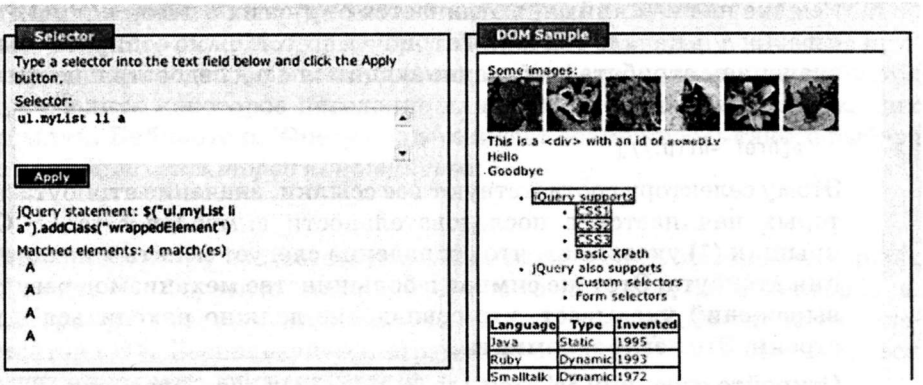


***Рис. 2.3.** Селектор ul.myList li a выбирает все якорные теги в элементах <li> независимо от глубины вложенности*

потомками элементов <ul> с классом myList. Ссылки во вложенных списках выбираться не будут, потому что элементы <ul>, выступающие в качестве родительских элементов для элементов подсписков <li>, не принадлежат классу myList (рис. 2.4).

Возможности *селекторов атрибутов* чрезвычайно широки. Представьте, что нам требуется определить специальное поведение только для ссылок, указывающих на внешние сайты. Вот фрагмент страницы Selectors Lab, уже рассмотренный выше:

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

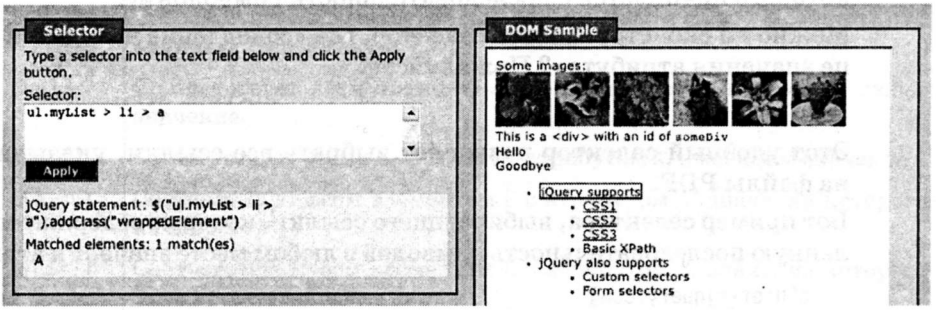


***Рис. 2.4.** Селектор ul.myList > li > a выбирает только прямых потомков родительских узлов*

Ссылка на внешний сайт отличается от других ссылок наличием строки `http://` в начале значения атрибута `href`. Можно отбирать ссылки со значением атрибута `href`, начинающимся с последовательности символов `http://`:

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

Этому селектору соответствуют все ссылки, значение атрибута `href` которых начинается с последовательности символов `http://`. Символ крышки (^) указывает, что совпадение следует искать в начале значения атрибута. Этот же символ в большинстве механизмов регулярных выражений указывает, что совпадение должно находиться в начале строки. Это легко запомнить.

Откройте страницу Selectors Lab, откуда был взят фрагмент HTML-разметки, введите в текстовое поле селектор `a[href^=http://]` и щелкните на кнопке Apply. Обратите внимание: выделилась только ссылка на сайт jQuery.

Есть и другие способы применения селекторов атрибутов. Выбрать элемент с определенным атрибутом независимо от его значения позволяет селектор

```
form[method]
```

Этому селектору соответствуют любые элементы `<form>`, в которых явно определен атрибут `method`.

Выбрать элементы с определенным значением атрибута позволяет селектор:

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

Этому селектору соответствуют все элементы ввода типа `text`.

Мы уже рассматривали селектор, которому соответствуют элементы, где «совпадение находится в начале значения атрибута». Вот еще один такой селектор:

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

Этот селектор выбирает все элементы `<div>` с атрибутом `title`, значение которого начинается с последовательности символов `my`.

Можно ли сконструировать селектор, где «совпадение находится в конце значения атрибута»? Пожалуйста:

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

Этот удобный селектор позволяет выбрать все ссылки, указывающие на файлы PDF.

Вот пример селектора, выбирающего ссылки, которые содержат определенную последовательность символов в любом месте значения атрибута:

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

Как вы наверняка уже поняли, этому селектору соответствуют все элементы `<a>`, которые ссылаются на сайт jQuery.

Помимо атрибутов иногда бывает необходимо выбрать элемент, только если он содержит некоторый другой элемент. Возьмем в качестве примера предыдущий фрагмент со списком и предположим, что требуется определить некоторое поведение для элементов списка, содержащих ссылки. Библиотека jQuery поддерживает возможность такого выбора с помощью *селекторов контейнеров*:

```
li:has(a)
```

Этому селектору соответствуют все элементы <li>, которые содержат элемент <a>. Обратите внимание: это не то же самое, что селектор `li a`, которому соответствуют все элементы <a>, содержащиеся внутри элементов <li>. Воспользуйтесь страницей Selectors Lab, чтобы убедиться в различии этих двух селекторов.

Селекторы CSS, которые можно использовать с библиотекой jQuery, приведены в табл. 2.1.

Таблица 2.1. Базовые селекторы CSS, поддерживаемые библиотекой jQuery

Селектор	Описание
*	Соответствует любому элементу
E	Соответствует всем элементам с именем тега E
E F	Соответствует всем элементам с именем тега F, вложенным в элемент с именем тега E
E>F	Соответствует всем элементам с именем тега F, являющимся прямыми потомками элементов с именем тега E
E+F	Соответствует всем элементам F, которым непосредственно предшествует любой элемент E на том же уровне вложенности
E~F	Соответствует всем элементам F, которым предшествует любой элемент E на том же уровне вложенности
E:has(F)	Соответствует всем элементам с именем тега E, имеющим хотя бы один вложенный элемент с именем тега F
E.C	Соответствует всем элементам E с именем класса C. В отсутствие E эквивалентен селектору *.C
E#I	Соответствует элементу E с идентификатором (id) I. В отсутствие E эквивалентен селектору *.I
E[A]	Соответствует всем элементам E с атрибутом A, имеющим любое значение
E[A=V]	Соответствует всем элементам E с атрибутом A, имеющим значение V
E[A^=V]	Соответствует всем элементам E с атрибутом A, значение которого начинается с V
E[A$=V]	Соответствует всем элементам E с атрибутом A, значение которого заканчивается на V
E[A*=V]	Соответствует всем элементам E с атрибутом A, значение которого содержит V

Не забывайте, что поддерживается только один уровень вложенности. Хотя есть возможность вложить еще *один* уровень, например:

```
foo:not(bar:has(baz))
```

но дополнительные уровни вложенности, такие как:

```
foo:not(bar:has(baz:eq(2)))
```

не поддерживаются.

Теперь, обладая этими сведениями, вернитесь к странице Selectors Lab и проведите еще несколько экспериментов с различными селекторами из табл. 2.1. Попробуйте, например, выбрать элементы , содержащие текст *Hello* и *Goodbye* (подсказка: для этого вам придется сконструировать комбинацию из нескольких селекторов).

На случай, если рассмотренных выше селекторов окажется недостаточно, есть дополнительная возможность нарезать страницу на еще более тонкие ломтики вдоль и поперек.

2.1.3. Выбор элементов по позиции

Иногда требуется выбрать элементы по их положению на странице или по размещению относительно других элементов. К примеру, если мы хотим выбрать первую ссылку на странице или каждый второй абзац, или последний пункт каждого списка. Библиотека jQuery поддерживает механизмы, позволяющие производить подобную выборку.

Например:

```
a:first
```

Данному селектору соответствует первый элемент <a> на странице.

А как быть с селектором, который выбирает каждый второй элемент?

```
p:odd
```

Этому селектору соответствует каждый нечетный элемент параграфа. Аналогичным образом выбирается каждый четный элемент:

```
p:even
```

Другая форма селектора

```
li:last-child
```

выбирает последний дочерний элемент родительского элемента. В этом примере селектору соответствует последний дочерний элемент <li> в каждом элементе <ul>.

Таких селекторов целая уйма, и они могут обеспечить весьма элегантное решение довольно сложных задач. В табл. 2.2 приведен список таких позиционных селекторов.

Таблица 2.2. Дополнительные селекторы, поддерживаемые jQuery: выбирают элементы на основе их местоположения в дереве DOM

Селектор	Описание
`:first`	Первое совпадение на странице. Селектор `li a:first` выберет первую ссылку, которая при этом находится в элементе списка
`:last`	Последнее совпадение на странице. Селектор `li a:last` выберет последнюю ссылку, которая при этом находится в элементе списка
`:first-child`	Первый дочерний элемент. Селектор `li:first-child` выберет первый элемент каждого списка
`:last-child`	Последний дочерний элемент. Селектор `li:last-child` выберет последний элемент каждого списка
`:only-child`	Выберет все элементы, являющиеся единственными дочерними элементами
`:nth-child(n)`	Выберет *n*-й дочерний элемент. Селектор `li:nth-child(2)` выберет второй элемент в каждом списке
`:nth-child(even|odd)`	Четные (`even`) или нечетные (`odd`) дочерние элементы. Селектор `li:nth-child(even)` выберет четные дочерние элементы в каждом списке
`:nth-child(Xn+Y)`	*n*-й дочерний элемент, порядковый номер которого вычисляется в соответствии с представленной формулой. Если значение `Y` равно `0`, его можно опустить. Селектор `li:nth-child(3n)` выберет каждый третий элемент, тогда как селектор `li:nth-child(5n+1)` выберет элемент, следующий за каждым пятым элементом.
`:even` и `:odd`	Четные (`even`) или нечетные (`odd`) элементы на странице. Селектор `li:even` выберет каждый четный элемент списка
`:eq(n)`	*n*-й элемент
`:gt(n)`	Выберет элементы, расположенные за *n*-м элементом (кроме него)
`:lt(n)`	Выберет элементы, расположенные перед *n*-м элементом (кроме него)

Здесь есть одна особенность. Селектор `nth-child` начинает отсчет элементов с 1, тогда как остальные селекторы – с 0. Счет с 1 в селекторе `nth-child` реализован из соображений совместимости с CSS, но дополнительные нестандартные селекторы библиотеки jQuery следуют общепринятым соглашениям программирования, начиная отсчет с 0. Накопив некоторый опыт работы с селекторами, вы легко запомните правило «кто есть кто», но в первое время это может стать источником ошибок.

Попробуем еще углубиться.

Рассмотрим таблицу, содержащую список некоторых языков программирования с некоторой дополнительной информацией о них:

```
<table id="languages">
  <thead>
    <tr>
      <th>Language</th>
      <th>Type</th>
      <th>Invented</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>Java</td>
      <td>Static</td>
      <td>1995</td>
    </tr>
    <tr>
      <td>Ruby</td>
      <td>Dynamic</td>
      <td>1993</td>
    </tr>
    <tr>
      <td>Smalltalk</td>
      <td>Dynamic</td>
      <td>1972</td>
    </tr>
    <tr>
      <td>C++</td>
      <td>Static</td>
      <td>1983</td>
    </tr>
  </tbody>
</table>
```

Предположим, нам нужно выбрать все ячейки таблицы, содержащие названия языков программирования. Так как все они являются первыми ячейками в строках, можно использовать селектор:

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

Можно также использовать такой селектор:

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

Но первый вариант выглядит более очевидным и элегантным.

Выбрать ячейки с описанием типизации, применяемой в языке программирования, можно с помощью селектора `:nth-child(2)`, а ячейки с годами появления языков – с помощью селектора `:nth-child(3)` или `:last-child`. Если потребуется выбрать самую последнюю ячейку таблицы (в которой указан год *1983*), можно использовать селектор `td:last`. Кроме того, селектор `td:eq(2)` выберет ячейку с текстом *1995*,

td:nth-child(2) выберет все ячейки с описанием типизации, применяемой в языке программирования. Не забывайте, что селектор :eq начинает отсчет элементов с 0, а селектор :nth-child – с 1.

Прежде чем двинуться дальше, вернитесь к странице Selectors Lab и попробуйте выбрать в списке вторую и четвертую строки. Затем попробуйте выбрать ячейку таблицы с текстом *1972* тремя разными способами. А потом прочувствуйте разницу между селектором nth-child и другими селекторами выбора элементов по абсолютной позиции.

Селекторы CSS, которые мы рассмотрели, обладают невероятно широкими возможностями, но сейчас мы попробуем выжать из селекторов jQuery еще больше.

2.1.4. Нестандартные селекторы jQuery

Селекторы CSS обеспечивают высокую гибкость и широкие возможности при выборе требуемых элементов DOM, но иногда требуется выбирать элементы на основе характеристик, не предусмотренных спецификацией CSS.

Например, может потребоваться выбрать все флажки (check boxes), отмеченные пользователем. При выборке по значению атрибута будут учитываться только начальные значения элементов управления, как они определены в HTML-коде, поэтому jQuery предлагает свой, нестандартный фильтр :checked, выбирающий только те элементы, которые были отмечены. Например, селектор input выберет все элементы <input>, а селектор input:checked – только отмеченные элементы <input>. Нестандартный селектор :checked работает на манер селектора атрибута CSS (например, [foo=bar]), в том смысле, что оба выбирают элементы в соответствии с некоторым критерием. Комбинирование нестандартных селекторов может еще больше повысить их гибкость; примером могут служить комбинации селекторов :radio:checked и :checkbox:checked.

Как уже говорилось, jQuery поддерживает все селекторы CSS, а также привносит множество своих нестандартных селекторов. Эти нестандартные селекторы приведены в табл. 2.3.

Многие нестандартные селекторы jQuery имеют отношение исключительно к формам и позволяют достаточно элегантно выбирать элементы определенного типа или в определенном состоянии. Эти селекторы-фильтры также можно комбинировать друг с другом. Например, выбрать только активные и отмеченные флажки можно было бы так:

```
:checkbox:checked:enabled
```

Попробуйте поэкспериментировать с этими фильтрами на странице Selectors Lab, пока не ощутите, что достаточно полно представляете себе их возможности.

Эти фильтры – полезное дополнение к уже известным нам селекторам, но что можно сказать о фильтре *инвертирования* условия?

Таблица 2.3. Нестандартные селекторы-фильтры jQuery, позволяющие еще точнее идентифицировать нужные элементы

Селектор	Описание
:animated	Выбирает элементы, в настоящий момент управляемые механизмом воспроизведения анимационных эффектов. Подробнее об анимации и визуальных эффектах рассказывается в главе 5
:button	Выбирает все кнопки (input[type=submit], input[type=reset], input[type=button] или button)
:checkbox	Выбирает только элементы-флажки (input[type=checkbox])
:checked	Выбирает только отмеченные флажки или переключатели (поддерживается средствами CSS)
:contains(foo)	Выбирает только элементы, содержащие текст *foo*
:disabled	Выбирает только элементы форм, находящиеся в неактивном состоянии (поддерживается средствами CSS)
:enabled	Выбирает только элементы форм, находящиеся в активном состоянии (поддерживается средствами CSS)
:file	Выбирает все элементы типа file (input[type=file]).
:header	Выбирает только элементы, являющиеся заголовками, например, элементы от <h1> до <h6>
:hidden	Выбирает только скрытые элементы
:image	Выбирает изображения в формах (input[type=image])
:input	Выбирает только элементы форм (input, select, textarea, button)
:not(filter)	Инвертирует значение указанного фильтра filter
:parent	Выбирает только элементы, у которых имеются вложенные (дочерние) элементы (включая простой текст), но не выбирает пустые элементы
:password	Выбирает только элементы ввода пароля (input[type=password])
:radio	Выбирает только переключатели (input[type=radio])
:reset	Выбирает только кнопки сброса (input[type=reset] или button[type=reset])
:selected	Выбирает элементы option, которые были выбраны
:submit	Выбирает кнопки submit (button[type=submit] или input[type=submit])
:text	Выбирает только элементы ввода текста (input[type=text])
:visible	Выбирает только видимые элементы

Фильтр :not

Инвертировать условие выбора, например, чтобы выбрать все элементы ввода, представленные *не* отмеченными флажками, позволит фильтр

:not, который поддерживает как фильтры CSS, так и нестандартные селекторы-фильтры jQuery.

Выбор неотмеченных флажков (элементов <input>):

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

Важно понимать различие между селекторами-фильтрами, которые ограничивают количество возможных совпадений, применяя дополнительные критерии выбора (как те, что были продемонстрированы выше), и *селекторами поиска*. Селекторы поиска, такие как селектор

Как говорится, «постойте, это еще не всё!»

Мы подчеркивали и продолжаем подчеркивать, что доля успеха jQuery заключается в простоте расширения функциональных возможностей посредством подключаемых модулей. Если вы уже знакомы с применением языка XML Path Language (XPath) для выбора элементов внутри документов XML (eXtensible Markup Language), можете считать, что вам повезло. Есть подключаемый модуль для библиотеки jQuery, обеспечивающий некоторую базовую поддержку XPath для совместного использования селекторов CSS и jQuery. Найти этот модуль можно на странице *http://jquery.com/plugins/project/xpath*.

Имейте в виду, что поддерживаются только самые основы XPath, но этого должно быть достаточно (в комбинации со всем остальным, что доступно в jQuery), чтобы обеспечить гибкость выбора.

Во-первых, подключаемый модуль поддерживает самые обычные селекторы / и //. Например, /html//form/fieldset выберет все элементы <fieldset>, непосредственно вложенные в элемент <form>.

Можно также использовать селектор * для представления любых элементов, например, /html//form/*/input выбирает все элементы <input>, вложенные точно в один элемент, который вложен в элемент <form>.

Подключаемый модуль XPath также поддерживает селектор выбора родительского элемента .., который выбирает родителей, указанных в селекторе элементов. Например, //div/.. выбирает все элементы, являющиеся прямыми родителями элемента .

Также поддерживаются селекторы атрибутов XPath (//div[@foo=bar]) и селекторы контейнеров (//div[@p], который выбирает элементы , содержащие по меньшей мере одни элемент p). Модуль также поддерживает функцию position() через позиционные селекторы jQuery, описанные выше. Например, position()=0 соответствует селектору :first, а position()>n соответствует селектору :gt(n).

выбора вложенных элементов (символ пробела), селектор выбора прямых потомков (>) и селектор выбора элементов на том же уровне вложенности (+), отыскивают *все* элементы, имеющие какое-либо отношение к тем, что уже отобраны, но не ограничивают возможность совпадения какими-либо критериями.

Фильтр `:not` можно применять только к селекторам-фильтрам, но не к селекторам поиска. Селектор

```
div p:not(:hidden)
```

является корректным, а `div :not(p:hidden)` – нет.

В первом случае выбираются все не скрытые элементы <p>, вложенные в элемент . Во втором случае селектор является некорректным, поскольку пытается применить условие `:not` к селектору, а не к фильтру (`p` в `p:hidden` не является фильтром).

Чтобы упростить вам жизнь, заметим: селекторы-фильтры легко распознаются по начальному символу двоеточия (`:`) или по символу квадратной скобки (`[`). Все остальные селекторы нельзя использовать внутри фильтра `:not`.

Как мы уже видели, jQuery дает нам обширный набор инструментов, позволяющих выбирать элементы, присутствующие в странице, для выполнения манипуляций посредством методов jQuery, которые будут рассматриваться в главе 3. Но прежде чем перейти к рассмотрению этих методов, нам необходимо узнать, как с помощью функции `$()` создавать новые элементы HTML для включения в выбранные наборы.

2.2. Создание новых элементов HTML

Иногда требуется создать новый фрагмент HTML и вставить его в страницу. С помощью jQuery это делается легко и просто, как мы уже видели в главе 1. Помимо выбора имеющихся элементов страницы функция `$()` может создавать новые элементы HTML. Рассмотрим инструкцию:

```
$("<div>Hello</div>")
```

Это выражение создает новый элемент , готовый к добавлению в страницу. Мы можем манипулировать вновь созданным фрагментом с помощью любых команд jQuery, применяемых к обернутым наборам существующих элементов. На первый взгляд, эта возможность может показаться не столь существенной, но когда речь пойдет об обработчиках событий, применении технологий Ajax и создании визуальных эффектов (в последующих главах), мы убедимся, насколько она удобна.

Обратите внимание: создать пустой элемент можно с помощью более краткой записи:

```
$("<div>")          <-------- Идентично $("<div></div>") и $("<div/>")
```

Как это часто бывает, здесь не все так просто: мы не можем использовать этот прием для создания элементов <script>. Но у нас есть масса способов, позволяющих обойти ситуации, требующие создания элементов <script>.

Чтобы почувствовать, что вы сможете делать позднее (не волнуйтесь, если что-то останется непонятным), взгляните на следующий фрагмент:

```
$("<div class='foo'>I have foo!</div><div>I don't</div>")
  .filter(".foo").click(function() {
    alert("I'm foo!");
  }).end().appendTo("#someParentDiv");
```

В этом отрывке сначала создаются два элемента , один с классом foo и второй без класса. Затем мы ограничиваем выбор только элементом с классом foo и присоединяем к нему обработчик события, который в результате щелчка мышью запускает диалог. В заключение используется метод end() (разд. 2.3.6) для возврата к полному набору, включающему оба элемента , и этот набор присоединяется к дереву DOM, в конец элемента со значением атрибута id=someParentDiv.

Для единственной инструкции выполняется достаточно много работы, но этот пример наглядно показывает, что необходимости писать длинные сценарии нет.

Страница HTML, в которой реализован этот пример, включена в состав загружаемого пакета примеров в виде файла *chapter2/new.divs.html*. Открыв этот файл в броузере, вы должны получить результат, показанный на рис. 2.5.

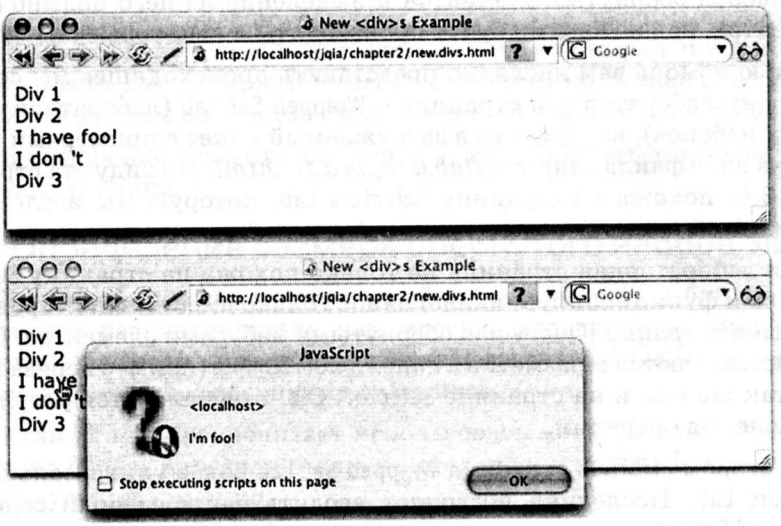


***Рис. 2.5.** Из сценариев можно создавать новые элементы HTML и определять для них дополнительные атрибуты, такие как обработчики событий, причем все это в единственной инструкции jQuery*

При первой загрузке страницы (как показано в верхней половине рис. 2.5) создается новый элемент `<div>` и добавляется в дерево DOM (поскольку пример фрагмента страницы был помещен в обработчик события готовности документа) сразу за элементом, содержащим текст *Div 2* (который имеет значение атрибута `id=someParentDiv`). В нижней половине рис. 2.5 показано, как выглядит диалог, открываемый щелчком мышью на вновь созданном элементе `<div>`.

Не волнуйтесь пока слишком сильно, мы еще не рассмотрели многое из необходимого для полного понимания предыдущего примера. До всего этого мы доберемся достаточно скоро. А сейчас перейдем к обсуждению темы манипулирования обернутым набором элементов, включая команду `filter()`, использованную в этом примере.

2.3. Манипулирование обернутым набором элементов

Как только у нас появился обернутый набор элементов, полученный из существующего дерева DOM с помощью селекторов или созданный из фрагментов HTML-разметки (или в результате применения обоих способов сразу), мы можем манипулировать этими элементами с помощью всевозможных команд jQuery. Эти команды будут рассматриваться в следующей главе. Но что, если мы еще не совсем готовы к этому? Что если *далее* нам потребуется уточнить набор элементов, обернутых функцией jQuery?

В этом разделе мы исследуем различные способы уточнения, расширения набора обернутых элементов и выделение из него подмножества элементов, над которыми будут производиться некоторые операции.

С целью помочь вам наглядно представить происходящее мы создали еще одну лабораторную страницу – Wrapped Set Lab (лаборатория обернутых наборов), включив ее в загружаемый пакет с примерами к этой главе в виде файла *chapter2/lab.wrapped.set.html*. По виду эта страница (рис. 2.6) похожа на страницу Selectors Lab, которую мы исследовали ранее в этой главе.

Новая лабораторная страница не только похожа на страницу Selectors Lab, она и функционирует аналогично. Только вместо селекторов здесь вводятся *операции* jQuery над обернутыми наборами элементов. Операции применяются к разметке в панели DOM Sample (пример дерева DOM), где, так же как и на странице Selectors Lab, отображаются результаты выполнения операций.

В некотором смысле страница Wrapped Set Lab более универсальна, чем Selectors Lab. Последняя позволяет вводить единственный селектор, а Wrapped Set Lab дает возможность ввести любое выражение, которое будет воздействовать на обернутый набор элементов. Так как jQuery позволяет составлять цепочки команд, вводимое выражение может так-

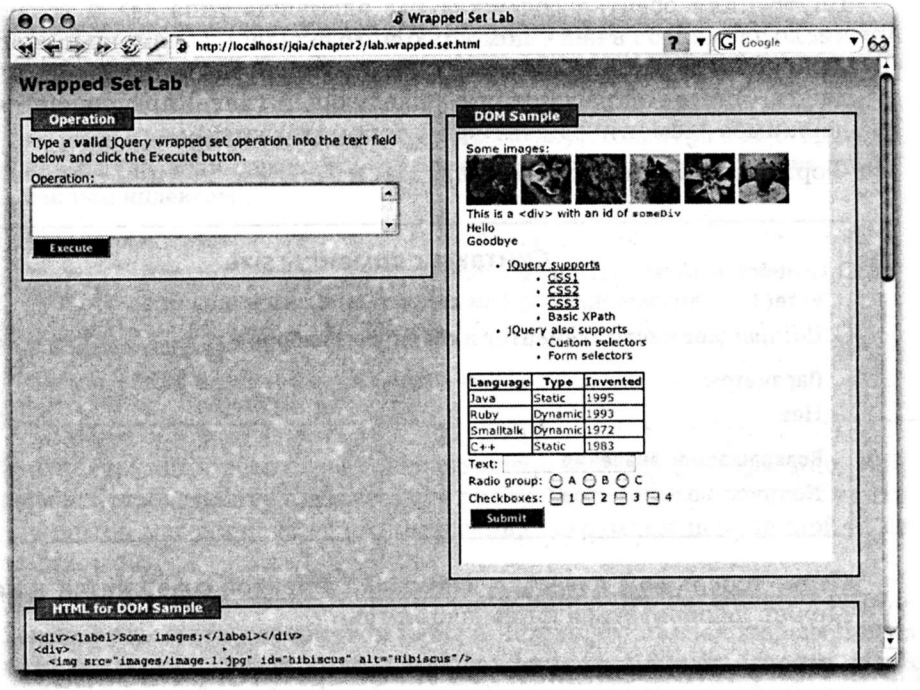


Рис. 2.6. Страница Wrapped Set Lab поможет увидеть, как создаются обернутые наборы элементов и как можно манипулировать ими

же включать команды, что делает эту страницу более мощной в смысле исследования операций jQuery. Учтите, что вводимые команды должны иметь допустимый синтаксис, так же как выражения, которые в результате дают обернутые наборы элементов. В противном случае вы столкнетесь с ошибками выполнения программного кода JavaScript.

В следующем разделе мы увидим, как работает эта новая лабораторная страница.

2.3.1. Определение размера обернутого набора элементов

Ранее говорилось, что обернутый набор элементов в библиотеке jQuery может рассматриваться как массив. Это сходство включает также свойство `length`, присутствующее в массивах JavaScript и определяющее количество обернутых элементов.

Если кто-то предпочитает использовать метод, а не свойство, библиотека jQuery определяет еще и метод `size()`, который возвращает ту же самую информацию.

Рассмотрим следующую инструкцию:

```
$('#someDiv')
  .html('There are '+$('a').size()+' link(s) on this page.');
```

Внутренняя обертка содержит все элементы типа <a> и возвращает число элементов в ней с помощью метода size(). С помощью этого числа конструируется текстовая строка, которая затем посредством метода html() (его мы рассмотрим в следующей главе) превращается в содержимое элемента с идентификатором (id) someDiv.

Формальный синтаксис метода size():

Синтаксис команды size

size()

Возвращает число элементов в обернутом наборе

Параметры

Нет

Возвращаемое значение

Количество элементов.

Итак, теперь вам известно, сколько элементов содержится в наборе. Можно ли обратиться к ним напрямую?

2.3.2. Получение элементов из обернутого набора

Обычно, получив обернутый набор элементов, мы тут же можем выполнить над ним какую-либо операцию с помощью команд jQuery. Например, можно скрыть их все с помощью метода hide(). Но вполне возможна ситуация, когда нам потребуется получить прямой доступ к элементам для выполнения на ними низкоуровневых операций средствами языка JavaScript.

Так как jQuery позволяет рассматривать обернутый набор как массив JavaScript, можно просто использовать индексы массива, чтобы получить любой элемент в обернутом наборе по его индексу. Например, получить первый элемент в наборе всех элементов <img> на странице с атрибутом alt можно так:

```
$('img[alt]')[0]
```

Если кто-то предпочитает вместо индекса массива использовать метод, для тех же целей jQuery определяет метод get().

Фрагмент

```
$('img[alt]').get(0)
```

эквивалентен предыдущему примеру, в котором использована операция получения элемента массива по индексу.

Метод get()также служит для получения обычного массива Javascript, содержащего все обернутые элементы. Например:

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

Синтаксис команды get

`get(index)`

Получает один или все элементы в обернутом наборе. Если параметр `index` не указан, возвращаются все элементы обернутого набора в виде массива JavaScript. Если параметр `index` определен, возвращается элемент с указанным индексом.

Параметры

`index` (число) Индекс единственного возвращаемого элемента. Если опущен, возвращается весь набор в виде массива.

Возвращаемое значение

Элемент DOM или массив элементов DOM.

Эта инструкция обертывает все элементы `<button>` на странице, которым непосредственно предшествуют элементы `<label>`, и создает из этих элементов массив JavaScript, присваивая его затем переменной `allLabeledButtons`.

Мы можем использовать обратную операцию поиска индекса определенного элемента в обернутом наборе. Допустим, по некоторым причинам нам нужно узнать индекс изображения с идентификатором `findMe` в обернутом наборе всех изображений на странице. Это значение можно получить так:

```
var n = $('img').index($('img#findMe')[0]);
```

Синтаксис команды `index()`:

Синтаксис команды index

`index(element)`

Отыскивает заданный элемент в обернутом наборе и возвращает его индекс в наборе. Если указанный элемент отсутствует в наборе, возвращается значение `-1`.

Параметры

`element` (элемент) Ссылка на элемент, индекс которого требуется определить.

Возвращаемое значение

Порядковый номер указанного элемента в обернутом наборе или `-1`, если элемент в наборе отсутствует.

Теперь узнаем, можно ли вместо одного элемента получить уточненный набор элементов, который был обернут.

2.3.3. Получение срезов обернутого набора элементов

Получив обернутый набор элементов, мы можем захотеть уточнить этот набор, добавив или убрав часть элементов первоначального множества. Библиотека jQuery предоставляет множество методов для управления набором обернутых элементов. Прежде всего рассмотрим возможность добавления элементов в набор.

Добавление дополнительных элементов в обернутый набор

Нередки ситуации, когда требуется добавить дополнительные элементы в имеющийся обернутый набор. Эта возможность полезнее прочих, если после применения каких-либо команд к первоначальному набору нужно добавить дополнительные элементы. Не забывайте про способность jQuery объединять команды в цепочки, что позволяет выполнить огромный объем работы единственной инструкцией.

Для начала рассмотрим простейшую ситуацию. Предположим, нам нужно отыскать все элементы `<img>`, в которых определен атрибут `alt` или `title`. Возможности селекторов jQuery позволяют выразить это в виде единственного селектора:

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

Но чтобы проиллюстрировать добавление с помощью метода `add()`,получим тот же самый набор иначе:

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

Такое использование метода `add()` позволяет объединять несколько селекторов по условию *ИЛИ*, создавая объединение элементов, соответствующее сразу обоим селекторам. Такие методы, как `add()`, можно

Синтаксис команды add

`add(expression)`

Добавляет в обернутый набор элементы, определяемые параметром `expression`. Параметр `expression` может быть селектором, фрагментом HTML-разметки, элементом DOM или массивом элементов DOM.

Параметры

`expression`	(строка | элемент | массив) Определяет, что должно быть добавлено в набор. Этот параметр может быть селектором jQuery, в этом случае любые соответствующие ему элементы будут добавлены в набор. Если это фрагмент HTML-разметки, соответствующие элементы будут созданы и затем добавлены в набор. Если это элемент DOM или массив элементов DOM, они будут добавлены в набор.

Возвращаемое значение

Обернутый набор элементов.

использовать вместо селекторов, в том смысле, что в таком случае можно будет с помощью метода `end()` (который мы исследуем в разд. 2.3.6) исключить из набора элементы, добавленные методом `add()`.

Откройте страницу Wrapped Set Lab в броузере, введите предыдущий пример (в точности так, как показано) и щелкните на кнопке Execute (выполнить). В результате должна быть выполнена операция jQuery, результатом которой будут все изображения с установленным атрибутом `alt` или `title`.

Исследовав исходный код HTML-разметки в панели DOM Sample, видим, что все изображения цветов имеют атрибут `alt`, изображения щенков имеют атрибут `title`, а изображение кофейника не имеет ни того, ни другого атрибута. Таким образом, можно было бы ожидать, что все изображения, за исключением кофейника, станут частью обернутого набора. На рис. 2.7 приведен снимок экрана с полученными результатами.

Мы видим, что в обернутый набор добавлены пять изображений из шести (то есть все, за исключением кофейника). (Красная рамка может быть плохо заметна в книге, где рисунки напечатаны в черно-белом варианте.)

Теперь давайте рассмотрим более реалистичный пример использования метода `add()`. Допустим, нам требуется окружить жирной рамкой все элементы `<img>` с атрибутом `alt`, а затем сделать полупрозрачными все элементы `<img>` с атрибутом `alt` или `title`. Здесь нам не поможет оператор «запятая» (`,`) селекторов CSS, потому что сначала нужно выполнить операцию над обернутым набором и только *потом* добавить в него дополнительные элементы. Этого легко можно было бы добиться с помощью нескольких инструкций, но эффективнее и элегантнее использовать мощь jQuery, решив ту же задачу с помощью цепочки команд, объединенных в единую инструкцию:

```
$('img[alt]').addClass('thickBorder').add('img[title]')
➥ .addClass('seeThrough')
```

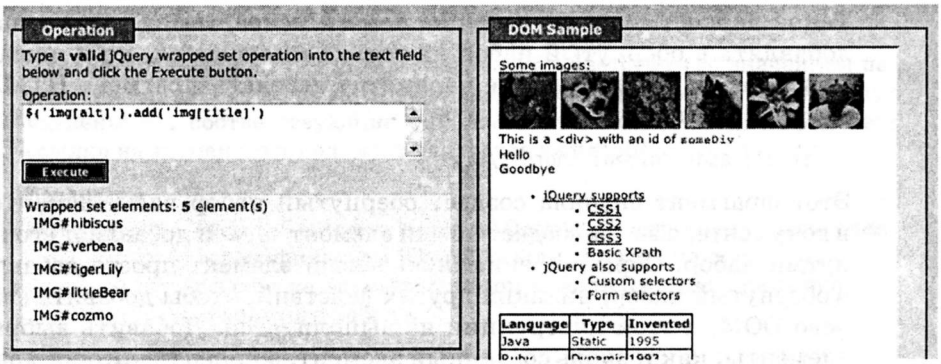


Рис. 2.7. Как и ожидалось, выражению jQuery соответствуют только изображения с атрибутом alt или title

Эта инструкция создает обернутый набор всех элементов <img> с атрибутом alt, применяет к нему предопределенный класс CSS, задающий жирную рамку, добавляет в набор элементы <img> с атрибутом title и в заключение применяет к расширенному набору класс CSS, который определяет степень полупрозрачности.

Введите эту инструкцию на странице Wrapped Set Lab (в которой уже предопределены указанные классы) – результат должен выглядеть, как показано на рис. 2.8.

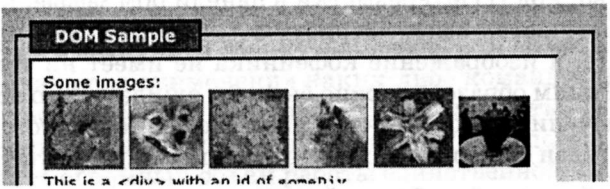


Рис. 2.8. Способность jQuery объединять команды в цепочки позволяет выполнять сложные операции с помощью единственной инструкции, о чем свидетельствует данный результат

В полученном результате можно заметить, что изображения с цветами (которые имеют атрибут alt) окружены жирными рамками, а все изображения, за исключением кофейника (единственное изображение, не имеющее ни атрибута alt, ни атрибута title), стали полупрозрачными в результате применения правила opacity.

Метод add()также позволяет добавлять в имеющийся обернутый набор элементы, заданные ссылками на них. Если методу add() передается ссылка на элемент или массив ссылок на элементы, он добавит эти элементы в обернутый набор. Предположим, что в переменной с именем someElement хранится ссылка на элемент; тогда его можно добавить в набор всех изображений, имеющих атрибут alt с помощью инструкции:

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

Такой гибкости уже достаточно, тем не менее, метод add() позволяет добавлять в обернутый набор не только имеющиеся, но и новые элементы, если ему в качестве параметра передать фрагмент HTML-разметки. Например:

```
$('p').add('<div>Hi there!</div>')
```

Этот фрагмент сначала создает обернутый набор всех элементов <p> в документе, а затем создает новый элемент и добавляет его в обернутый набор. Обратите внимание: новый элемент просто добавляется в обернутый набор, никаких других действий, чтобы добавить его в дерево DOM, в этой инструкции не выполняется. Добавить выбранные элементы, как и вновь созданный элемент, в конец некоторого раздела DOM можно с помощью метода append() из библиотеки jQuery (наберитесь терпения, мы поговорим об этих методах достаточно скоро).

Расширить обернутый набор с помощью метода `add()` легко и просто. А теперь рассмотрим методы библиотеки jQuery, позволяющие исключать элементы из обернутого набора.

Уменьшение содержимого обернутого набора

Мы уже видели, насколько просто в jQuery создаются обернутые наборы применением нескольких селекторов с помощью цепочки методов `add()`, формирующей объединение по условию *ИЛИ*. Точно так же с помощью метода `not()` можно организовать объединение селекторов в цепочку для *исключения* лишних элементов. По действию этот метод напоминает фильтр `:not`, рассмотренный ранее, но подобно методу `add()` может использоваться для удаления элементов из обернутого набора в любом месте цепочки команд jQuery.

Предположим, требуется отобрать все элементы `<img>` на странице, имеющие атрибут `title`, за исключением тех, что содержат в значении этого атрибута строку *puppy*. Можно было бы сконструировать единственный селектор, отвечающий поставленным условиям (а именно, `img[title]:not([title*=puppy])`, но представим, что мы забыли о существовании фильтра `:not`. С помощью метода `not()`, который удаляет из обернутого набора любые элементы, соответствующие заданному выражению селектора, мы можем выразить тип объединения *за исключением*. Реализацию описанного выше условия можно записать так:

```
$('img[title]').not('[title*=puppy]')
```

Введите это выражение на странице Wrapped Set Lab и выполните его. Вы увидите, что будет отобрано только изображение коричневого щенка. Изображение черного щенка, включенное в первоначальный обернутый набор из-за наличия атрибута `title`, будет исключено из набора вызовом метода `not()`, потому что значение атрибута `title` этого элемента содержит строку *puppy*.

Синтаксис команды not

`not(expression)`

Удаляет элементы из обернутого набора в соответствии со значением параметра `expression`. Если параметр является селектором-фильтром, будут удалены все соответствующие ему элементы. Если методу передается ссылка на элемент, этот элемент удаляется из набора.

Параметры

`expression` (строка | элемент | массив) Этот параметр может быть селектором-фильтром jQuery, ссылкой на элемент или массивом ссылок на элементы, которые должны быть исключены из обернутого набора.

Возвращаемое значение

Обернутый набор элементов.

Обратите внимание: методу not() можно передавать только *селекторы-фильтры*, не допускающие использование ссылок на элементы (подразумевая, что тип элемента может быть любым). Если бы мы передали методу not() более явный селектор img[title*=puppy], то не смогли бы получить ожидаемый результат, потому что селекторы элементов этим методом не поддерживаются.

Подобно методу add() с помощью метода not() можно исключать из обернутого набора отдельные элементы, для чего нужно передать методу ссылку на элемент или массив ссылок на элементы. Последний способ особенно интересен, потому что, если помните, любой обернутый набор можно использовать как массив ссылок на элементы.

Если сложно или невозможно выразить условие в виде селектора, то можно фильтровать обернутые наборы. В таких случаях нам придется реализовать программное управление фильтрацией обернутого набора элементов. Например, можно было бы обойти в цикле все элементы в наборе и с помощью метода not(element) удалить те из них, которые не соответствуют критериям отбора. Но разработчики jQuery избавили нас от необходимости вручную выполнять эти действия, предусмотрев метод filter().

Если передать методу filter() некоторую функцию, он вызовет ее для каждого элемента в наборе и удалит те элементы, для которых эта функция вернет значение false. Эта функция получает доступ к текущему элементу набора через контекст функции (this).

Предположим, что по некоторым причинам нам требуется создать обернутый набор всех элементов <td>, содержащих числовые значения. Даже при всей гибкости селекторов jQuery, с их помощью невозможно выразить такое условие. В подобных ситуациях можно использовать метод filter():

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

Данная инструкция jQuery создает обернутый набор всех элементов <td> и затем для каждого элемента вызывает функцию, переданную методу filter(), с текущим элементом в качестве значения this. С помощью регулярного выражения функция определяет, соответствует ли содержимое элемента заданному шаблону (последовательность из одной или более цифр), и возвращает значение false, если это не так. Каждый элемент, для которого функция вернет значение false, будет удален из обернутого набора.

Вновь вернитесь к странице Wrapped Set Lab, введите предыдущее выражение и выполните его. Вы увидите, что будут выделены только ячейки столбца Invented (год появления).

Методу filter() можно также передать селектор, на который накладываются ограничения, описанные ранее для метода not(), а именно: могут использоваться только селекторы-фильтры с подразумеваемым типом элемента. При таком использовании селектор имеет обратное зна-

Синтаксис команды filter

`filter(expression)`

Отфильтровывает элементы из обернутого набора с помощью переданного ей выражения селектора или функции фильтрации.

Параметры

`expression` (строка | функция) Этот параметр может быть селектором jQuery, используемым для удаления всех не соответствующих ему элементов, или функцией, принимающей решение об исключении. Данная функция вызывается для каждого элемента в наборе, с текущим элементом в качестве контекста вызова функции. Любые элементы, для которых функция возвращает значение `false`, удаляются из набора.

Возвращаемое значение

Обернутый набор элементов.

чение по сравнению с методом `not()`, то есть удаляются все элементы, *не* соответствующие селектору. Это не очень мощный метод, так как обычно проще воспользоваться более ограничивающим селектором, но он удобен при использовании в цепочках команд jQuery. Рассмотрим следующий пример:

```
$('img').addClass('seeThrough').filter('[title*=dog]')
➥ .addClass('thickBorder')
```

Эта цепочка команд отбирает все элементы `<img>`, применяет к ним класс `seeThrough`, затем оставляет в наборе только те элементы, значение атрибута `title` которых содержит строку *dog*, и к оставшимся элементам применяет класс `thickBorder`. В результате все изображения становятся полупрозрачными, и только изображение коричневой собаки окружается жирной рамкой.

Методы `not()` и `filter()` предоставляют нам широкие возможности уточнения набора обернутых элементов на лету на основе произвольного критерия выбора обернутых элементов. Мы можем также создавать подмножество обернутого набора на основе позиций элементов в наборе. Давайте рассмотрим методы, позволяющие сделать это.

Получение подмножества обернутого набора

Иногда требуется получить подмножество обернутого набора на основе позиций элементов в этом наборе. Для этих целей библиотека jQuery предоставляет метод `slice()`. Эта команда создает и возвращает *новый* набор из любой непрерывной части или срез первоначального обернутого набора. Синтаксис этой команды представлен на стр. 66.

Получить обернутый набор с единственным элементом из первоначального набора по известному индексу требуемого элемента в этом

Синтаксис команды slice

`slice(begin, end)`

Создает и возвращает новый обернутый набор, содержащий непрерывную область первоначального набора.

Параметры

`begin` (число) Позиция первого элемента (отсчет начинается с нуля) области, которая должна быть включена в возвращаемый срез.

`end` (число) Необязательный индекс первого элемента (отсчет начинается с нуля), который не должен быть включен в возвращаемый срез, или позиция элемента, стоящего сразу же за последним включаемым в срез элементом. Если этот параметр опущен, конец возвращаемого среза совпадает с концом первоначального набора.

Возвращаемое значение

Вновь созданный обернутый набор элементов.

наборе можно с помощью метода `slice()`, передав ему местоположение элемента в обернутом наборе. Например, получить третий элемент можно так:

```
$('*').slice(2,3);
```

Эта инструкция отберет все элементы на странице и затем сгенерирует новый набор, содержащий третий элемент первоначального набора.

Обратите внимание: эта инструкция не равнозначна инструкции `$('*').get(2)`, которая вернет третий *элемент*, а не набор, содержащий третий элемент.

Таким образом, следующая инструкция:

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

выберет все элементы на странице и затем создаст набор из первых четырех элементов.

Выбрать элементы из конца обернутого набора можно с помощью такой инструкции:

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

Она отберет все элементы на странице и затем вернет набор – все элементы, за исключением четырех первых.

Но и это еще не все! jQuery также позволяет получать подмножества обернутого набора на основе взаимоотношений между элементами в дереве DOM. Давайте посмотрим, как это делается.

2.3.4. Получение обернутого набора с учетом взаимоотношений

Библиотека jQuery позволяет получать новые обернутые наборы из имеющихся на основе взаимоотношений между обернутыми элементами в дереве DOM. Примечательно, что по действию методы этой группы несколько отличаются от большинства рассмотренных ранее методов, которые модифицируют передаваемый им обернутый набор. Как и метод `slice()`, рассматриваемые в этом разделе методы возвращают *новый* обернутый набор, не изменяя первоначальный набор.

В табл. 2.4 приведены эти методы и их описания. Каждый из перечисленных методов принимает селектор в качестве необязательного параметра, с помощью которого производится выборка требуемых элементов. При отсутствии параметра с селектором выбираются все допустимые элементы.

Эти методы обеспечивают большую широту выбора элементов из дерева DOM на основе их взаимоотношений с другими элементами. Но обсуждение на этом не заканчивается. Давайте посмотрим, как jQuery работает с обернутыми наборами.

Таблица 2.4. Методы получения нового обернутого набора на основе взаимоотношений между элементами

Метод	Описание
`children()`	Возвращает обернутый набор, состоящий из единственных потомков обернутых элементов.
`contents()`	Возвращает обернутый набор содержимого элементов (который может включать текстовые узлы) первоначального обернутого набора. (Часто используется, чтобы получить содержимое элементов `<iframe>`.)
`next()`	Возвращает обернутый набор, состоящий из единственных следующих соседних элементов в первоначальном обернутом наборе.
`nextAll()`	Возвращает обернутый набор, содержащий все последующие соседние элементы в первоначальном обернутом наборе.
`parents()`	Возвращает обернутый набор, содержащий единственные прямые родительские элементы в первоначальном обернутом наборе.
`prev()`	Возвращает обернутый набор, состоящий из единственных предшествующих соседних элементов в первоначальном обернутом наборе.
`prevAll()`	Возвращает обернутый набор, содержащий все предшествующие соседние элементы в первоначальном обернутом наборе.
`siblings()`	Возвращает обернутый набор, содержащий единственные соседние элементы в первоначальном обернутом наборе.

Все методы в табл. 2.4, за исключением `contents()`, принимают параметр, содержащий строку, которую можно использовать для фильтрации результата.

2.3.5. Дополнительные способы использования обернутого набора

Всего вышесказанного, пожалуй, достаточно, тем не менее, в рукаве у jQuery есть еще несколько приемов, позволяющих определять наши коллекции обернутых объектов.

Метод `find()` служит для поиска в существующем обернутом наборе, возвращая новый набор, который содержит все элементы, соответствующие заданному селектору. Допустим, обернутый набор хранится в переменной `wrappedSet`, тогда с помощью следующей инструкции мы сможем получить другой обернутый набор всех цитат (элемент `<cite>`) из абзацев:

```
wrappedSet.find('p cite')
```

Обратите внимание: если все требовалось бы реализовать в единственной инструкции, это можно было бы сделать, передав параметр контекста селектору jQuery:

```
$('p cite',wrappedSet)
```

Как и многие другие методы jQuery для работы с обернутыми наборами, метод `find()` обретает дополнительную мощь при использовании внутри цепочки операций.

Синтаксис команды find

`find(selector)`

Возвращает новый обернутый набор, содержащий все элементы из первоначального набора, соответствующие заданному селектору.

Параметры

`selector` (строка) Селектор jQuery, которому должны соответствовать элементы в возвращаемом наборе.

Возвращаемое значение

Вновь созданный обернутый набор элементов.

В дополнение к возможности поиска в обернутом наборе элементов, соответствующих указанному селектору, библиотека jQuery предоставляет метод поиска элементов, содержащих определенную строку. Метод `contains()`[1] возвращает новый обернутый набор, который состоит

[1] Метод `contains()` удален из jQuery 1.2. Чтобы добавить метод, работающий, как в старой версии, на сайте книги предлагается использовать следующую конструкцию (*примеч. науч. ред.*):

```
jQuery.fn.contains = function(text) { return $(this).filter(":contains
("+text+")"); }
```

из элементов, содержащих заданную строку в любом месте в теле их содержимого. Например:

```
$('p').contains('Lorem ipsum')
```

Эта инструкция возвращает обернутый набор всех абзацев, содержащих строку *Lorem ipsum*. Обратите внимание: вхождение заданной строки проверяется во всех компонентах элемента, составляющих его тело, включая разметку и значения атрибутов дочерних элементов, но разметка и значения атрибутов самого элемента не проверяются.

Синтаксис команды contains

`contains(text)`

Возвращает новый обернутый набор, сконструированный из элементов, содержащих текстовую строку, переданную методу в качестве параметра `text`.

Параметры

`text` (строка) Текст, который должен содержать элемент, чтобы быть включенным в возвращаемый набор.

Возвращаемое значение

Вновь созданный обернутый набор элементов.

Последним в этом разделе мы исследуем один из методов, позволяющих проверять обернутый набор на наличие, по крайней мере, одного элемента, соответствующего заданному селектору. Метод `is()` возвращает значение `true`, если селектору соответствует хотя бы один элемент, и `false` – в противном случае. Например:

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

Эта инструкция устанавливает значение переменной `hasImage` равным `true`, если в текущей странице присутствует хотя бы один элемент `<img>`.

Синтаксис команды is

`is(selector)`

Определяет, имеются ли в обернутом наборе элементы, соответствующие заданному селектору.

Параметры

`selector` (строка) Селектор, проверяющий наличие искомых элементов в обернутом наборе.

Возвращаемое значение

`true` – если заданному селектору соответствует хотя бы один элемент, и `false` – в противном случае.

2.3.6. Управление цепочками команд jQuery

Мы проделали грандиозную работу (и продолжаем делать, потому что это *действительно* грандиозная работа) по изучению возможности методов jQuery для работы с обернутыми наборами объединяться в цепочки, чтобы выполнить больше операций в одной инструкции. Способность формировать цепочки не только позволяет кратко записывать мощные операции, но и повышает их эффективность, потому что благодаря такой возможности ликвидируется необходимость снова и снова извлекать один и тот же набор элементов для применения к нему нескольких команд.

В зависимости от методов, из которых конструируется цепочка команд, может быть сгенерировано несколько обернутых наборов. Например, при использовании метода `clone()` (который мы подробнее рассмотрим в главе 3) генерируется новый обернутый набор, представляющий собой точную копию первого набора. Если бы у нас не было никакой возможности сослаться на первоначальный набор сразу после его создания, наши возможности по конструированию гибких цепочек команд jQuery были бы весьма ограниченны.

Рассмотрим инструкцию:

```
$('img').clone().appendTo('#somewhere');
```

Внутри этой инструкции создается два обернутых набора: первоначальный набор всех элементов `<img>` на странице и второй, обернутый набор с копиями этих элементов. Метод `clone()` возвращает этот второй набор как результат своей работы и именно к этому набору применяется команда `appendTo()`.

Но как быть, если впоследствии нам потребуется применить команду, например добавления имени класса, к первоначальному обернутому набору уже *после* того, как получена его копия? Мы не можем добавить новую команду в конец цепочки, потому что она будет воздействовать на копию, а не на первоначальный набор изображений.

Для решения этой проблемы jQuery предоставляет команду `end()`. Данный метод при использовании внутри цепочки команд jQuery выполняет откат к предыдущему обернутому набору и возвращает его в качестве возвращаемого значения, в результате все последующие операции будут применяться к предыдущему набору.

Например:

```
$('img').clone().appendTo('#somewhere').end().addClass('beenCloned');
```

Метод `appendTo()` возвращает набор новых копий, но после вызова `end()` мы получаем предыдущий обернутый набор (оригинальных изображений), на который затем воздействует команда `addClass()`. Без команды `end()` команда `addClass()` воздействовала бы на набор копий.

Проще всего представлять себе этот механизм как стек, в который попадают обернутые наборы, производимые командами jQuery в цепочке. При вызове команды `end()` самый верхний (то есть самый последний) обернутый набор выталкивается из стека, открывая последующим командам доступ к предыдущему набору.

Синтаксис команды end

`end()`

Используется в цепочках команд jQuery для отката к предыдущему обернутому набору элементов.

Параметры

нет

Возвращаемое значение

Предыдущий обернутый набор.

Другой удобный метод jQuery, который изменяет стек обернутых наборов, называется `andSelf()`. Этот метод объединяет два самых верхних в стеке набора в единый обернутый набор.

Синтаксис команды andSelf

`andSelf()`

Объединяет два предыдущих обернутых набора в цепочках команд.

Параметры

нет

Возвращаемое значение

Объединенный обернутый набор.

2.4. Итоги

В этой главе все внимание было сконцентрировано на создании и уточнении наборов элементов (в этой и следующих главах называемых *обернутыми наборами*) с помощью множества методов, которые библиотека jQuery предоставляет для идентификации элементов на странице HTML.

Библиотека jQuery обеспечивает много гибких и мощных *селекторов* с кратким и гибким синтаксисом, напоминающим селекторы CSS, для идентификации элементов внутри страницы документа. Среди них не только синтаксические конструкции CSS2, поддерживаемые большинством современных броузеров, но и конструкции CSS3, а также

дополнительные селекторы. Кроме того, с помощью подключаемого модуля обеспечивается базовая поддержка селекторов XPath.

Библиотека jQuery также позволяет создавать и расширять обернутые наборы, создавая на лету новые элементы с помощью фрагментов HTML-разметки. Этими ни к чему не привязанными элементами можно манипулировать, как любыми другими элементами в обернутом наборе, в конечном счете присоединяя их к тем или иным частям документа страницы.

Библиотека jQuery предоставляет ряд надежных методов для усечения обернутого набора, как сразу же после его создания, так и в ходе выполнения цепочки команд. Применение фильтрующих критериев к уже имеющемуся набору позволяет легко создавать новые обернутые наборы.

В общем и целом jQuery предоставляет множество инструментов, позволяющих нам легко и точно идентифицировать элементы страницы для выполнения манипуляций над ними.

В этой главе мы исследовали значительную часть фундамента, фактически *не выполняя* никаких операций над элементами DOM страницы. Зато сейчас мы знаем, как выбрать элементы, на которые нужно воздействовать, и мы готовы вдохнуть жизнь в наши страницы с помощью команд jQuery.

В этой главе:

- Получение и установка атрибутов элементов
- Манипулирование именами классов элементов
- Установка содержимого элементов
- Получение доступа к значениям элементов форм
- Изменение дерева DOM

3

Вдыхаем жизнь в страницы с помощью jQuery

Помните дни (не столь давние), когда неопытные авторы страниц пытались взбодрить свои страницы с помощью всякой гадости вроде рамок, мерцающего текста, броского фона, затрудняющего чтение текста на странице, раздражающих анимированных GIF-изображений и, что, пожалуй, хуже всего, фонового звука, сопровождающего загрузку страницы (и позволяющего проверить, насколько быстро пользователь закроет броузер)?

Много воды утекло с тех пор.

Сегодняшние веб-разработчики и дизайнеры многому научились и направляют мощь динамического HTML (Dynamic HTML, DHTML) на *улучшение* условий для пользователя, а не на создание витрин с раздражающими эффектами.

Будь то пошаговое раскрытие содержимого или создание элементов ввода, превосходящих по своим возможностям стандартные элементы HTML, или предоставление пользователям возможности настраивать страницы в соответствии со своими предпочтениями, DHTML – или возможность манипулирования деревом DOM – позволяет многим веб-разработчикам поражать (но не раздражать) своих пользователей.

Практически ежедневно мы встречаем веб-страницы, которые делают нечто заставляющее нас воскликнуть: «Ого! Я и не подозревал, что это возможно!» И будучи профессионалами (а не просто «жутко любознательными»), мы тут же бросаемся изучать исходный код, чтобы понять, *как* это делается.

Вместо того чтобы писать все нужные сценарии вручную, можно использовать возможности jQuery по управлению деревом DOM, создавая те самые страницы «Ого!» с минимальным объемом программного кода. В предыдущей главе мы изучили множество способов отбора элементов DOM в обернутые наборы, а в этой главе узнаем о мощных инструментах jQuery, позволяющих выполнять операции над этими наборами и вдыхать жизнь и тот самый неуловимый фактор «Ого!» в наши страницы.

3.1. Манипулирование свойствами и атрибутами элементов

Что касается элементов DOM, главное, чем мы можем управлять, – это *свойства* и *атрибуты* этих элементов. Первоначальные значения, присвоенные свойствам и атрибутам элементов DOM в результате синтаксического анализа HTML-разметки, можно изменять в сценариях.

Чтобы сверить и уточнить терминологию, взгляните на фрагмент HTML-разметки с описанием элемента `<img>`:

```
<img id="myImage" src="image.gif" alt="An image" class="someClass"
➥title="This is an image"/>
```

В этой разметке элемента именем тега является последовательность символов `img`, а разметка `id`, `src`, `alt`, `class` и `title` представляет собой атрибуты элемента, каждый из которых состоит из имени и значения. Разметка этого элемента воспринимается и интерпретируется броузером, в результате чего создается объект JavaScript, представляющий данный элемент в дереве DOM. В дополнение к разного рода атрибутам этот объект обладает множеством *свойств*, включая те, что являются значениями атрибутов (есть даже свойство, которое содержит список всех атрибутов элемента). На рис. 3.1 представлена упрощенная схема этого процесса.

Броузер транслирует HTML-разметку в элемент DOM, представляющий изображение. Создается объект `NodeList` (один из контейнерных типов, определяемых в объектной модели документа) и присваивается свойству `attributes` элемента как значение. Между атрибутами и соответствующими им свойствами (которые мы называем *свойствами атрибутов*) существует динамическая связь. Изменение атрибута приводит к изменению соответствующего свойства атрибута и наоборот. Тем не менее, значения не всегда могут быть идентичны. Например, установка атрибута `src` в значение `image.gif` приведет к тому, что в свойство `src` будет записан полный абсолютный адрес URL изображения.

Большинство имен свойств атрибутов JavaScript совпадают с именами соответствующих им атрибутов, но в некоторых случаях они могут отличаться. Например, атрибут `class` в этом примере представлен свойством атрибута с именем `className`.

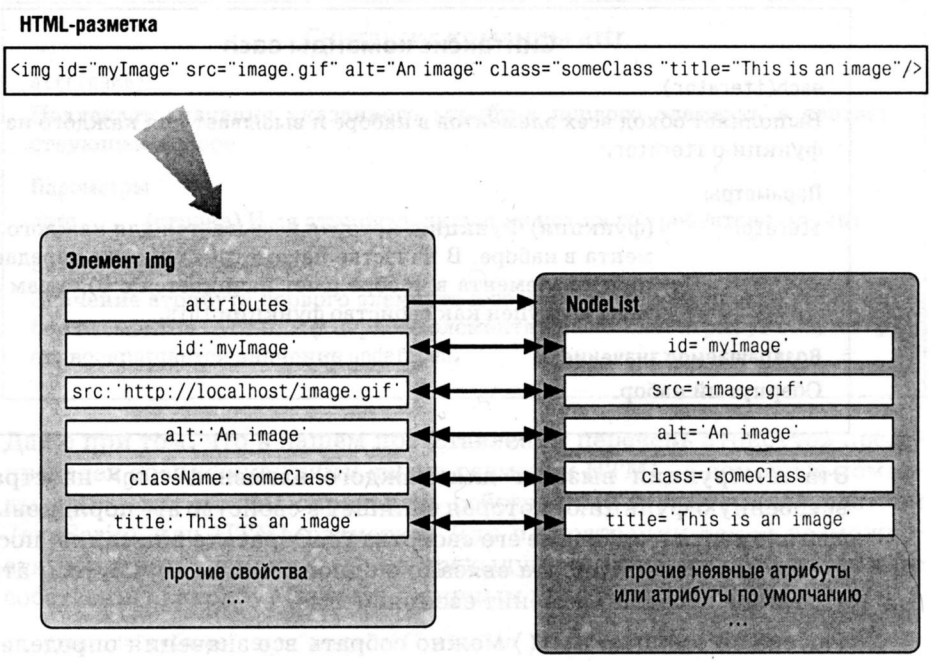


Рис. 3.1. HTML-разметка транслируется в элементы DOM, включая атрибуты тега и свойства, созданные из них

Библиотека jQuery предоставляет средства, упрощающие манипулирование атрибутами элемента, и обеспечивает доступ к самому элементу, чтобы и мы могли изменять его свойства. Какой из двух способов выбрать для выполнения манипуляций, зависит от того, что и как нужно сделать.

Для начала рассмотрим приемы получения и установки свойств элемента.

3.1.1. Манипулирование свойствами элементов

У библиотеки jQuery нет специальной команды для получения или изменения значений свойств элемента. Для доступа к свойствам и их значениям следует использовать обычные средства JavaScript. Вся хитрость в том, чтобы получить ссылки на элементы. Самый простой способ получить или изменить элементы компонентов в наборе предоставляет команда `each()`. Ее синтаксис представлен на стр. 76.

Эта команда позволяет установить значение свойства для всех элементов набора. Например:

```
$('img').each(function(n){
  this.alt='This is image['+n+'] with an id of '+this.id;
});
```

Синтаксис команды each

`each(iterator)`

Выполняет обход всех элементов в наборе и вызывает для каждого из них функцию `iterator`.

Параметры

`iterator`	(функция) Функция, которая вызывается для каждого элемента в наборе. В качестве параметра функции передается индекс элемента в наборе (счет начинается с 0), а сам элемент доступен как свойство функции `this`.

Возвращаемое значение

Обернутый набор.

Эта инструкция вызовет для каждого элемента `<img>` на странице встроенную функцию, которая запишет в свойство `alt` порядковый номер элемента и значение его свойства `id`. Обратите внимание: поскольку это свойство атрибута связано с одноименным атрибутом, атрибут `alt` также косвенно изменит свое значение.

Так же с помощью `each()` можно собрать все значения определенного свойства в массив:

```
var allAlts = new Array();
$('img').each(function(){
  allAlts.push(this.alt);
});
```

Если все, что нам нужно, – это получить значение свойства единственного элемента, вспомните, что к соответствующему набору можно обращаться как к массиву JavaScript. Значение свойства можно получить, например, так:

```
var altValue = $('#myImage')[0].alt;
```

Работать с атрибутами немного сложнее, чем со свойствами в JavaScript, поэтому jQuery предоставляет для этого дополнительные возможности. Посмотрим, как это делается.

3.1.2. Извлечение значений атрибутов

Как и многие другие команды библиотеки jQuery, команда `attr()` позволяет выполнять как чтение, так и запись. Способ использования команды jQuery, предназначенной для таких совершенно разных операций, определяется количеством и типами передаваемых ей параметров.

С помощью команды `attr()` можно извлечь значение атрибута первого элемента в соответствующем наборе либо установить значение атрибута для всех элементов набора.

Синтаксис команды `attr()` для извлечения значения:

Синтаксис команды attr

`attr(name)`

Извлекает значение указанного атрибута первого элемента в соответствующем наборе.

Параметры

`name` (строка) Имя атрибута, значение которого требуется получить.

Возвращаемое значение

Значение атрибута первого элемента в соответствующем наборе. Если набор элементов пуст или у первого элемента указанный атрибут отсутствует, возвращается значение `undefined`.

Даже при том, что в нашем представлении перечень атрибутов предопределен спецификациями языка разметки HTML, с помощью команды `attr()` мы можем определять собственные атрибуты посредством JavaScript или HTML-разметки. Для иллюстрации такой возможности исправим элемент `<img>` из предыдущего примера, добавив в него собственный атрибут (выделен жирным шрифтом):

```
<img id="myImage" src="image.gif" alt="An image" class="someClass"
➥title="This is an image" custom="some value"/>
```

Заметим, что свой атрибут элемента мы, не мудрствуя лукаво, так и назвали – `custom`[1]. Мы можем получать значение этого атрибута, как если бы это был один из стандартных атрибутов, например:

```
$("#myImage").attr("custom")
```

Предупреждение

Использование нестандартных имен атрибутов, таких как `custom`, несмотря на то что это вполне безобидный трюк, приведет к тому, что ваша разметка будет считаться недопустимой – она не пройдет тестирование при проверке допустимости. Как следствие, это может привести к проблемам доступности и вызывать ошибки в программах, которые ожидают, что ваш сайт написан с использованием допустимой разметки HTML или XHTML.

Имена атрибутов в разметке HTML нечувствительны к регистру символов. Независимо от того, как атрибут вроде `title` объявлен в разметке, мы сможем получить доступ к атрибуту (или установить его значение, о чем мы вскоре поговорим) при любом варианте написания его имени – `Title`, `TITLE`, `TiTlE` или в любой другой эквивалентной комбинации регистров. Даже в XHTML, где имена атрибутов должны записываться символами в нижнем регистре, мы сможем получить доступ к атрибуту при любом варианте написания его имени.

[1] Custom – свой, собственный, пользовательский. – *Примеч. пер.*

Вы можете спросить: «Зачем вообще иметь дело с атрибутами, если доступ к свойствам так прост (как показано в предыдущем разделе)?»

Ответ на этот вопрос заключается в том, что команда `attr()` – это не просто обертка вокруг методов JavaScript `getAttribute()` и `setAttribute()`. В дополнение к возможности получить доступ к набору атрибутов элемента, библиотека jQuery предоставляет доступ к некоторым наиболее часто используемым свойствам, традиционно служившим источником раздражения для авторов страниц из-за различий между броузерами.

Набор нормализованных имен приведен в табл. 3.1.

Таблица 3.1. Нормализованные имена атрибутов для команды attr( )

Нормализованное имя	Исходное имя
`class`	ClassName
`cssFloat`	`styleFloat` для IE, `cssFloat` для остальных броузеров (использующих файлы *.css*)
`float`	`styleFloat` для IE, `cssFloat` для остальных броузеров (использующих файлы *.css*)
`for`	HtmlFor
`maxlength`	MaxLength
`readonly`	ReadOnly
`styleFloat`	`styleFloat` для IE, `cssFloat` для остальных броузеров (использующих файлы *.css*)

3.1.3. Установка значений атрибутов

jQuery предоставляет два способа установки значений атрибутов элементов в обернутом наборе. Начнем с наиболее простого способа, позволяющего однократно установить значение единственного атрибута (для всех элементов в обернутом наборе). Синтаксис этой операции:

Синтаксис команды attr

`attr(name,value)`

Устанавливает значение `value` атрибута `name` для всех элементов в обернутом наборе.

Параметры

`name` (строка) Имя атрибута, значение которого требуется установить.

`value` (строка | объект | функция) Определяет значение атрибута. Это может быть выражение JavaScript, результатом которого является некоторое значение, или функция. Порядок работы с этим параметром описан ниже в этой главе.

Возвращаемое значение

Обернутый набор.

Хотя этот вариант команды attr() на первый взгляд очень прост, он достаточно сложен в применении.

В наиболее типичном случае использования этого метода, когда параметр value является выражением JavaScript, результатом которого является некоторое значение (включая массив), вычисленное значение выражения присваивается указанному атрибуту как значение.

Гораздо интереснее случай, когда параметр value является ссылкой на функцию. В такой ситуации функция вызывается для *каждого* элемента в обернутом наборе и возвращаемое ею значение устанавливается как значение атрибута. При вызове функции передается единственный параметр, содержащий индекс элемента в наборе (отсчет начинается с нуля). Кроме того, функции передается сам элемент в виде значения переменной this, что позволяет функции подстраиваться под каждый конкретный элемент, и в этом – главное достоинство такого способа применения функций.

Рассмотрим инструкцию:

```
$('*').attr('title',function(index) {
  return 'I am element ' + index + ' and my name is ' +
         (this.id ? this.id : 'unset');
});
```

Эта команда выполнит обход всех элементов страницы и запишет в атрибут title каждого элемента строку, сконструированную из индекса элемента в дереве DOM и значения атрибута id этого конкретного элемента.

Такой прием подходит, если значение устанавливаемого атрибута не просто является каким-то значением, а зависит от других характеристик элемента.

Второй вариант команды attr() удобно использовать, когда требуется установить значения сразу нескольких атрибутов.

Синтаксис команды attr

attr(attributes)

Устанавливает значения атрибутов, передаваемые функции в виде объекта, для всех элементов в обернутом наборе.

Параметры

attributes	(объект) Объект, свойства которого копируются в значения атрибутов всех элементов в обернутом наборе.

Возвращаемое значение

Обернутый набор.

Данный формат вызова позволяет легко и быстро установить множество атрибутов для всех элементов обернутого набора. Параметром может

быть ссылка на любой объект, обычно – объект-литерал, свойства которого определяют имена и значения устанавливаемых атрибутов. Например:

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

Эта инструкция записывает пустую строку в атрибут `value` элементов `<input>` и строку `Пожалуйста, введите значение` в атрибут `title`.

Заметим, что если значением какого-либо свойства объекта, передаваемого в параметре `value`, является ссылка на функцию, она будет действовать, как описано выше, – функция будет вызвана для каждого отдельного элемента в соответствующем наборе.

Предупреждение

Броузер Internet Explorer не позволяет изменять атрибут `name` элементов `<input>`. Если потребуется изменить значение атрибута `name` элементов `<input>` в Internet Explorer, вы должны будете заменить имеющийся элемент новым, с нужным именем.

Теперь мы знаем, как получать и устанавливать значения атрибутов. Но что если нам потребуется вообще избавиться от них?

3.1.4. Удаление атрибутов

Для удаления атрибута элемента DOM jQuery предоставляет команду `removeAttr()`. Ее синтаксис:

Синтаксис команды removeAttr

`removeAttr(name)`

Удаляет атрибут, заданный параметром `name`, у всех элементов набора.

Параметры

`name` (строка) Имя удаляемого атрибута.

Возвращаемое значение

Обернутый набор.

Примечательно, что удаление атрибута не приводит к удалению соответствующего ему свойства из JavaScript-элемента DOM, хотя эта операция может привести к изменению значения свойства. Например, удаление атрибута `readonly` из элемента приведет к тому, что значение свойства `readOnly` изменится с `true` на `false`, но само свойство останется у элемента.

Теперь рассмотрим несколько примеров применения только что полученных знаний в наших страницах.

3.1.5. Игры с атрибутами

Допустим, мы хотим, чтобы каждая ссылка, указывающая на внешний сайт, открывалась в новом окне. Эта задача становится достаточно тривиальной, если есть полный контроль над разметкой, например:

```
<a href="http://external.com" target="_blank">Some External Site</a>
```

Это конечно хорошо, но как быть, если мы пользуемся системой управления содержимым (Content Management System, CMS) или wiki, где конечный пользователь может добавить новое содержимое без гарантии того, что добавит и атрибут `target="_blank"` во все внешние ссылки? Для начала попробуем точно определить, что нам нужно. Мы хотим, чтобы каждая ссылка, значение атрибута `href` которой начинается с последовательности символов *http://*, открывалась в новом окне (для чего надо установить атрибут `target` в значение `_blank`).

Решить эту задачу позволит описанный в этой главе прием:

```
$("a[href^=http://]").attr("target","_blank");
```

Сначала выбираются все ссылки, у которых значение атрибута `href` начинается со строки `http://` (указание на то, что ссылка является внешней). Затем их атрибут `target` устанавливается в значение `_blank`. Задание выполнено всего одной строкой программного кода jQuery!

Другой яркий пример использования возможностей jQuery для работы с атрибутами – решение наболевшего для всех веб-приложений вопроса «страшной проблемы двойной отправки» (Dreaded Double Submit Problem). Для веб-приложений типична ситуация: если отправка формы задерживается до нескольких секунд или более, нетерпеливый пользователь щелкает на кнопке отправки несколько раз, провоцируя неприятности для программного кода на стороне сервера.

Чтобы решить эту проблему, мы переопределим обработчик события `submit` и будем деактивизировать кнопку отправки после первого нажатия. В этом случае пользователь не сможет нажать кнопку повторно и к тому же получит визуальный сигнал (предполагается, что неактивные кнопки и в броузере выглядят неактивными) о том, что передача формы выполняется. Не вникая пока во все тонкости обработки события в следующем примере (в главе 5 будет дана вся необходимая информация по этой теме), сконцентрируйтесь на применении команды `attr()`:

```
$("form").submit(function() {
  $(":submit",this).attr("disabled", "disabled");
});
```

В теле обработчика события с помощью селектора `:submit` мы выбираем все кнопки отправки формы и записываем в атрибут `disabled` значение `"disabled"` (официально рекомендованное консорциумом W3C для атрибута). Обратите внимание: при выборке соответствующего набора мы передаем значение контекста `this` (второй параметр). Как мы узнаем

в главе 5 при обсуждении вопросов обработки событий, внутри обработчика события этот указатель всегда ссылается на элемент страницы, с которым связан обработчик.

Предупреждение

Деактивизация кнопки отправки таким способом совершенно не снимает с программного кода на стороне сервера ответственность за проверку факта двойной отправки или какие-либо другие проверки. Добавление этой особенности к клиентскому программному коду лишь повышает уровень удобства для пользователя и предотвращает двойную отправку при обычных обстоятельствах. Она не защищает от нападений или попыток взлома, поэтому программный код на стороне сервера по-прежнему должен обеспечивать максимальную защиту.

Ранее в этой главе, обсуждая различия в именах свойств и атрибутов, мы упоминали свойство `className`, однако истины ради следует отметить, что имена классов – это еще один особый случай, и обработка этих особенностей также предусматривается библиотекой jQuery. В следующем разделе описаны более удобные способы работы с именами классов, чем прямой доступ к свойству `className` или применение команды `attr()`.

3.2. Изменение стиля отображения элемента

Если нам нужно изменить стиль визуального представления элемента, есть два пути. Можно добавлять или удалять классы CSS, вынуждая механизм каскадных таблиц стилей изменить стиль элемента в соответствии с его новым классом, или воздействовать непосредственно на элемент DOM, применяя стили напрямую.

Давайте посмотрим, как jQuery упрощает операцию изменения классов стилей в элементах.

3.2.1. Добавление и удаление имен классов

Атрибуты и свойства элементов DOM с именем класса имеют уникальный формат и семантику и играют важную роль в создании функциональных пользовательских интерфейсов. Добавление и удаление имен классов из элементов – одно из основных средств, позволяющих динамически изменять визуальное представление элементов.

Одна из особенностей имен классов, которая делает их уникальными и осложняет работу с ними, – это то, что каждому элементу может быть присвоено любое число имен классов. В HTML список имен применяемых классов, разделенных пробелами, определяется с помощью атрибута `class`. Например:

```
<div class="someClass anotherClass yetAnotherClass"></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! То есть всякий раз, когда захочется добавить или удалить имя класса из элемента, в котором уже присутствуют имена классов, нам придется анализировать строку, чтобы вычленить отдельные имена при чтении и обеспечить корректный формат строки при записи.

Хотя создать программный код, делающий все это, – не наша главная цель, неплохо было бы всегда скрывать внутренние механизмы за функциями прикладного интерфейса. К счастью, в библиотеке jQuery уже реализовано все необходимое.

Добавить имена классов ко всем элементам соответствующего набора достаточно просто с помощью команды `addClass()`:

Синтаксис команды addClass

`addClass(names)`

Добавляет указанное имя класса (или имена классов) ко всем элементам в обернутом наборе.

Параметры

`names` (строка) Строка, содержащая имя добавляемого класса, или, в случае добавления нескольких имен классов, – строка с именами классов, разделенными пробелами.

Возвращаемое значение

Обернутый набор.

Так же просто позволяет удалить имена классов команда `removeClass()`:

Синтаксис команды removeClass

`removeClass(names)`

Удаляет указанное имя класса (или имена классов) у всех элементов в обернутом наборе.

Параметры

`names` (строка) Строка, содержащая имя удаляемого класса, или, в случае удаления нескольких имен классов, – строка с именами классов, разделенными пробелами.

Возвращаемое значение

Обернутый набор.

Часто требуется переключать наборы стилей туда-обратно, например, чтобы обозначить переход между двумя состояниями или по каким-то другим причинам, имеющим значение для нашего интерфейса. Такую возможность обеспечивает команда `toggleClass()` библиотеки jQuery.

Синтаксис команды toggleClass

`toggleClass(name)`

Добавляет указанное имя класса, если оно отсутствует в элементе, и удаляет имя у тех элементов, где указанное имя класса уже присутствует. Примечательно, что каждый элемент проверяется отдельно, поэтому в некоторые элементы имя класса может добавляться, а из других – удаляться.

Параметры

`name` (строка) Строка, содержащая имя переключаемого класса.

Возвращаемое значение

Обернутый набор.

Чаще всего команда `toggleClass()` применяется для легкого и быстрого переключения визуального представления элементов. Помните пример с полосатой таблицей (см. рис. 1.1)? Представьте, что у нас появились достаточно серьезные причины при определенных событиях заменять цвет фона нечетных строк цветом фона четных строк (и, возможно, наоборот). Команда `toggleClass()` позволяет решить эту задачу элементарно: добавить имя класса к каждой второй строке и удалить его из всех остальных.

Попробуем это сделать. В файле *chapter3/zebra.stripes.html* вы найдете копию той же страницы из главы 1, но с небольшими изменениями. В элемент `<script>` в заголовке страницы мы добавили функцию:

```
function swap() {
  $('tr').toggleClass('striped');
}
```

Эта функция использует команду `toggleClass()`, чтобы переключать имя класса `striped` во всех элементах `<tr>`. Мы также добавили вызов этой функции в атрибуты `onmouseover` и `onmouseout` таблицы:

```
<table onmouseover="swap();" onmouseout="swap();">
```

В результате всякий раз, когда указатель мыши пересекает границу таблицы, из всех элементов `<tr>` с именем класса `striped` оно удаляется, а ко всем элементам `<tr>`, не имеющим имени класса, оно добавляется. Этот эффект (довольно раздражающий) показан на рис. 3.2.

Манипулирование визуальным представлением элементов через имена классов CSS – очень мощное средство, но иногда приходится иметь дело с базовыми стилями, как если бы они были объявлены непосредственно в элементах. Посмотрим, что может нам предложить jQuery для решения этой задачи.

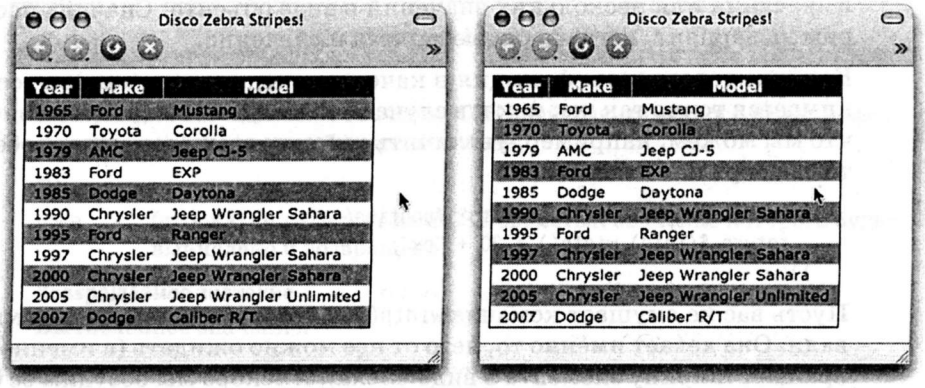


***Рис. 3.2.** Наличие или отсутствие класса striped переключается всякий раз, когда указатель мыши пересекает границу таблицы*

3.2.2. Получение и установка стилей

Изменяя класс элемента, мы можем выбирать предопределенный набор стилей, который должен быть применен, но иногда нам требуется отменить действие таблицы стилей. Применение стилей непосредственно к элементу автоматически отменяет действие таблицы стилей, что позволяет более тонко управлять отдельными элементами и их визуальным представлением.

Метод `css()` по действию напоминает метод `attr()`, позволяя нам устанавливать значения отдельных свойств CSS путем указания имени

Синтаксис команды css

`css(name, value)`

Устанавливает CSS-свойство `name` в значение `value` для всех элементов в обернутом наборе.

Параметры

`name` (строка) Имя CSS-свойства, значение которого требуется установить.

`value` (строка | число | функция) Строка, число или функция, содержащие значение свойства. Если в этом параметре передается функция, она будет вызвана для каждого элемента в обернутом наборе, а ее возвращаемое значение будет использовано в качестве значения CSS-свойства. В свойстве `this` функции передается сам элемент.

Возвращаемое значение

Обернутый набор.

и значения или нескольких значений в виде объекта. Сначала посмотрим на вариант, когда передаются имя и значение.

Как уже говорилось, функция в качестве входного параметра воспринимается точно так же, как и в случае с командой `attr()`. Это означает, что мы можем, например, увеличить ширину всех элементов в обернутом наборе на 20 пикселов:

```
$("div.expandable").css("width",function() {
  return $(this).width() + 20 + "px";
});
```

Пусть вас не смущает команда `width()`, которую мы еще не рассматривали. Она делает именно то, чего от нее можно ожидать (а именно, возвращает ширину элемента в виде числа), и вскоре мы обсудим ее более подробно. Тут хочется отметить кое-что интересное: свойство `opacity`, обычно вызывающее проблемы, отлично работает во всех типах броузеров при передаче значений в диапазоне от 0,0 до 1,0 – не надо больше путаться с альфа-фильтрами IE вроде `moz-opacity`!

Теперь рассмотрим сокращенную форму команды `css()`, которая по своему действию напоминает сокращенную версию команды `attr()`.

Синтаксис команды css

`css(properties)`

Устанавливает CSS-свойства, имена которых определены как ключи в переданном объекте, в связанные с ними значения для всех элементов в соответствующем наборе.

Параметры

`properties`	(объект) Объект, свойства которого копируются в CSS-свойства всех элементов в обернутом наборе.

Возвращаемое значение

Обернутый набор.

Как и в сокращенной версии команды `attr()`, здесь мы можем использовать функции в качестве значений для любых CSS-свойств в объекте, передаваемом через параметр `properties`. Они будут вызываться для каждого элемента в обернутом наборе, чтобы определить значения, которые должны применяться.

Наконец, мы можем передать команде `css()` одно только имя свойства, чтобы получить вычисленный стиль, ассоциированный с этим именем. Говоря «вычисленный стиль», мы подразумеваем стиль, который получается после применения всех внешних, внутренних и встроенных стилей CSS. Эта команда отлично работает во всех типах броузеров, даже со свойством `opacity`, для которого возвращается строка, представляющая число в диапазоне от 0,0 до 1,0.

Синтаксис команды css

`css(name)`

Возвращает вычисленное значение CSS-свойства, заданного именем `name`, для первого элемента в обернутом наборе.

Параметры

`name` (строка) Определяет имя CSS-свойства, значение которого будет вычислено и возвращено.

Возвращаемое значение

Вычисленное значение.

Имейте в виду, что этот вариант команды `css()` всегда возвращает строку, поэтому, если вам требуется число или значение какого-то другого типа, вы должны проанализировать возвращаемое значение самостоятельно.

Для некоторых CSS-свойств, к которым обращаются наиболее часто, библиотека jQuery реализует удобные команды, упрощающие получение значений этих свойств и их преобразование в наиболее часто используемые типы. В частности, мы можем получать (или изменять) ширину и высоту элемента в числовом виде с помощью удобных команд `width()` и `height()`. Чтобы установить ширину или высоту:

Синтаксис команд width и height

`width(value)`
`height(value)`

Устанавливают ширину или высоту всех элементов в соответствующем наборе.

Параметры

`value` (число) Устанавливаемое значение в пикселах.

Возвращаемое значение

Обернутый набор.

Имейте в виду, что эти команды являются сокращенными вариантами более универсальной команды `css()`, поэтому инструкция

```
$("div.myElements").width(500)
```

эквивалентна инструкции

```
$("div.myElements").css("width","500px")
```

Чтобы получить ширину или высоту:

Синтаксис команд width и height

`width()`

`height()`

Возвращают ширину или высоту первого элемента в обернутом наборе.

Параметры

Нет

Возвращаемое значение

Вычисленное значение ширины или высоты.

Тот факт, что значение ширины и высоты эти функции возвращают в виде чисел, – не единственное удобство этих команд. Определяя ширину или высоту элемента по его свойствам `style.width` или `style.height`, вы могли столкнуться с тем обстоятельством, что значения этих свойств устанавливаются только при наличии соответствующего атрибута `style` данного элемента – чтобы иметь возможность определять размеры элемента через эти свойства, прежде необходимо установить их. Вот уж не эталон удобства!

С другой стороны, команды `width()` и `height()` вычисляют и возвращают размеры элемента. И хотя знание точных размеров элементов редко требуется в простых страницах, позволяющих элементам располагаться как придется, но для полнофункциональных интернет-приложений очень важно уметь определять размеры, чтобы можно было правильно располагать активные элементы, такие как контекстные меню, всплывающие подсказки, дополнительные элементы управления и прочие динамические компоненты.

Давайте посмотрим на них в работе. На рис. 3.3 показан пример с двумя основными элементами: объектом исследований является элемент `<div>`, содержащий абзац текста (с рамкой и выделяющим его цветом фона), во втором элементе `<div>` выводятся размеры.

Размеры исследуемого элемента заранее неизвестны из-за отсутствия стилевых правил, задающих размеры. Ширина элемента определяется

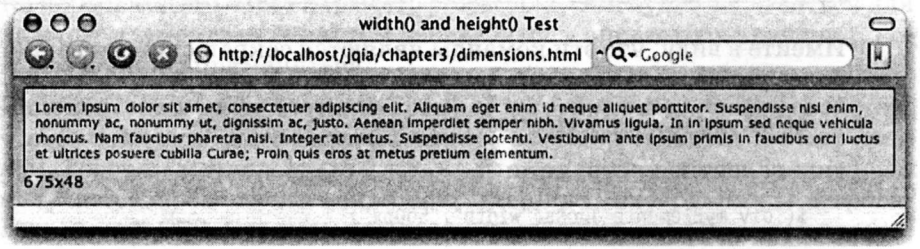


***Рис. 3.3.** Ширина и высота исследуемого элемента – не фиксированные значения и зависят от ширины окна броузера*

шириной окна броузера, а его высота зависит от того, как много места потребуется для отображения содержащегося в нем текста. При изменении размеров окна броузера изменятся и размеры элемента.

В нашей странице мы определяем функцию, которая с помощью команд `width()` и `height()` будет выяснять размеры объекта исследований – элемента `<div>` (с именем `testSubject`) и выводить полученные значения во втором элементе `<div>` (с именем `display`).

```
function report() {
  $('#display').html(
    $('#testSubject').width()+'x'+$('#testSubject').height()
  );
}
```

Функция вызывается в обработчике события готовности страницы, в результате на экране появляются числа 675 и 48, соответствующие размерам в окне броузера, как показано на рис. 3.3.

Мы также добавили вызов функции в атрибут `onresize` элемента `<body>`:

```
<body onresize="report();">
```

Результат изменения размеров окна броузера показан на рис. 3.4.

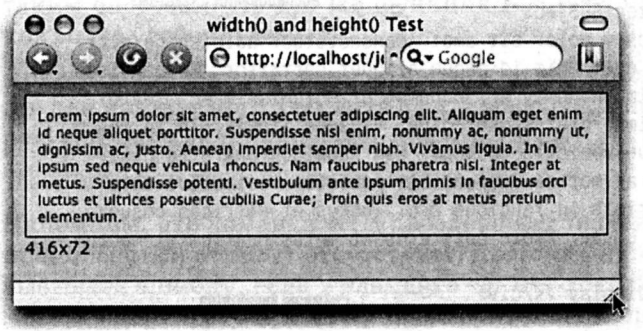


***Рис. 3.4.** Изменение размеров окна броузера приводит к изменению размеров объекта исследований, что отражается на вычисляемых значениях*

Эта способность определять вычисленные размеры элемента в любой момент неоценимо важна для точного размещения динамических элементов на страницах.

Полный исходный код этой страницы приведен в листинге 3.1. Также его можно найти в загружаемом пакете с примерами, в файле *chapter3/dimensions.html*.

***Листинг 3.1.** Динамическое слежение за размерами элемента*

```
<html>
  <head>
    <title>width() and height() Test</title>
    <link rel="stylesheet" type="text/css" href="../common.css">
```

```
<script type="text/javascript"
        src="../scripts/jquery-1.2.1.js"></script>
<script type="text/javascript">
  $(function(){
    report();
  });
  function report() {
    $('#display').html(
      $('#testSubject').width()+'x'+$('#testSubject').height()
    );
  }
</script>
<style>
  #testSubject {
    background-color: plum;
    border: 1px solid darkmagenta;
    padding: 8px;
    font-size: .85em;
  }
</style>
</head>
<body onresize="report();">
  <div id="testSubject">
    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>
</body>
</html>
```

Определение размеров элемента в момент готовности страницы

Вывод ширины и высоты исследуемого элемента

Стили, применяемые к исследуемому объекту

Определение размеров элемента при изменении размеров окна

Объявление объекта исследований с текстом

Размеры выводятся в этой области

Вероятно, вы заметили, что в этом примере мы встроили программный код прямо в разметку HTML, вопреки правилам ненавязчивого JavaScript. Пока в этом нет ничего страшного, но в следующей главе мы продемонстрируем более удачный способ привязки обработчиков событий.

Теперь, когда мы рассмотрели возможность манипулирования стилями элементов обернутого набора, давайте обсудим некоторые связанные со стилями действия, которые вам могут пригодиться.

3.2.3. Дополнительные команды работы со стилями

Часто нужно определить, имеется ли в элементе определенный класс. Библиотека jQuery предлагает для этого функцию `hasClass()`.

```
$("p:first").hasClass("surpriseMe")
```

Она возвращает значение true, если хотя бы у одного из элементов в соответствующем наборе есть указанный класс.

Синтаксис команды hasClass

`hasClass(name)`

Проверяет, есть ли класс с указанным именем хотя бы у одного элемента в соответствующем наборе.

Параметры

name (строка) Проверяемое имя класса.

Возвращаемое значение

Возвращает значение true, если хотя бы у одного из элементов в обернутом наборе имеется указанный класс, и false – в противном случае.

Напомним, что тот же результат можно получить с помощью команды is(), которая рассматривалась в главе 2:

```
$("p:first").is(".surpriseMe")
```

На самом деле в недрах jQuery функция hasClass() реализована именно таким способом! Но все же функция hasClass() делает программный код более удобочитаемым.

Другая часто востребованная возможность – получение списка классов для конкретного элемента в виде массива, а не в виде строки, которую еще надо анализировать. Мы можем сделать это так:

```
$("p:first").attr("class").split(" ");
```

Напомним, что команда attr() возвращает значение undefined, если запрашиваемый атрибут отсутствует, поэтому данная инструкция будет вызывать ошибку, если в элементе <p> нет хотя бы одного класса. Мы можем решить эту проблему предварительной проверкой наличия атрибута, заключив все это в удобное для многократного использования расширение jQuery:

```
$.fn.getClassNames = function() {
  if (name = this.attr("className")) {
    return name.split(" ");
  }
  else {
    return [];
  }
};
```

Пусть вас не смущают особенности синтаксиса расширений jQuery – мы подробно обсудим эту тему в главе 7. Сейчас важно то, что мы можем использовать функцию getClassNames() в любом месте сценария, получая массив имен классов или пустой массив, если элемент не имеет ни одного класса. Лихо!

Узнав, как получать и изменять значения стилей элементов, можно перейти к обсуждению способов изменения содержимого элементов.

3.3. Установка содержимого элемента

Когда дело доходит до изменения содержимого элементов, начинаются споры о том, каким способом это лучше делать: с применением методов DOM API – или за счет изменения HTML-разметки. В большинстве случаев изменить HTML-разметку элемента оказывается проще и эффективнее, поэтому jQuery предлагает ряд методов, реализующих именно этот способ.

3.3.1. Замена HTML-разметки или текста

Первая команда в этой группе – `html()`. Она позволяет извлекать содержимое элемента в виде HTML-разметки при вызове без параметров и, как и другие уже рассмотренные функции jQuery, изменяет содержимое элемента при вызове с параметром.

Так можно получить содержимое элемента в виде HTML-разметки:

Синтаксис команды html

`Html()`

Возвращает содержимое первого элемента в соответствующем наборе в виде HTML-разметки.

Параметры

Нет

Возвращаемое значение

Содержимое первого элемента в соответствующем наборе в виде разметки HTML. Возвращаемое значение идентично значению свойства `innerHTML` этого элемента.

Так можно установить содержимое в виде HTML-разметки для всех элементов в наборе:

Синтаксис команды html

`html(text)`

Устанавливает фрагмент HTML-разметки как содержимое всех элементов соответствующего набора.

Параметры

`text` (строка) Фрагмент HTML-разметки, который должен быть установлен как содержимое элементов.

Возвращаемое значение

Обернутый набор.

Кроме того, можно получать или устанавливать только текстовое содержимое элементов. Команда `text()` при вызове без параметров возвращает строку – результат конкатенации всех текстовых узлов. Допустим, у нас имеется следующий фрагмент HTML:

```
<ul id="theList">
  <li>One</li>
  <li>Two</li>
  <li>Three</li>
  <li>Four</li>
</ul>
```

Тогда инструкция

```
var text = $('#theList').text();
```

запишет в переменную `text` строку `OneTwoThreeFour`.

Синтаксис команды text

`text()`

Объединяет путем конкатенации текстовое содержимое всех обернутых элементов и возвращает полученный текст в качестве результата.

Параметры

Нет

Возвращаемое значение

Объединенная строка.

Также с помощью команды `text()` можно изменить текстовое содержимое обернутых элементов. Синтаксис этого варианта команды:

Синтаксис команды text

`text(content)`

Устанавливает содержимое строки `content` как текстовое содержимое всех обернутых элементов. Если строка `content` содержит угловые скобки (< и >), они замещаются эквивалентными HTML-элементами.

Параметры

`content` (строка) Новое текстовое содержимое всех обернутых элементов. Все угловые скобки замещаются эквивалентными HTML-элементами.

Возвращаемое значение

Обернутый набор.

Обратите внимание: эти команды замещают старую HTML-разметку или текст внутри элементов содержимым входного параметра, поэтому

использовать их следует с особой осторожностью. Если вы не хотите уничтожить прежнее содержимое элементов, то лучше используйте методы, которые оставляют старое содержимое нетронутым и только добавляют новое содержимое или изменяют окружение элемента. Давайте рассмотрим их.

3.3.2. Перемещение и копирование элементов

Добавить новое содержимое в конец уже имеющегося можно с помощью команды `append()`.

Синтаксис команды append

`append(content)`

Добавляет фрагмент HTML или элементы из параметра `content` ко всем соответствующим элементам.

Параметры

`content` (строка | элемент | объект) Строка, элемент или обернутый набор, которые будут добавлены к элементам в обернутом наборе. Подробности описаны ниже.

Возвращаемое значение

Обернутый набор.

Эта функция принимает строку, содержащую фрагмент HTML, ссылку на существующий или вновь созданный элемент DOM либо обернутый набор элементов.

Рассмотрим следующий простой пример:

```
$('p').append('<b>какой-либо текст<b>');
```

Эта инструкция добавляет фрагмент HTML, созданный из строки, в конец существующего содержимого всех элементов <p> на странице.

Вот более сложный пример применения команды с добавлением уже имеющихся элементов дерева DOM:

```
$("p.appendToMe").append($("a.appendMe"))
```

Эта инструкция добавляет все ссылки с классом `appendMe` в элементы <p> с классом `appendToMe`. Расположение добавляемых элементов зависит от числа элементов, принимающих добавку. Если добавление производится в единственный элемент, добавляемый элемент удаляется из своего прежнего местоположения – то есть выполняется операция *перемещения* элемента в новое местоположение. При добавлении в несколько элементов добавляемый элемент остается в своем прежнем местоположении и копируется в конец каждого принимающего элемента – то есть выполняется операция *копирования*.

Вместо полноценного обернутого набора можно указать ссылку на определенный элемент DOM:

```
var toAppend = $("a.appendMe")[0];
$("p.appendToMe").append(toAppend);
```

Тип операции – перемещение или копирование элемента toAppend – снова зависит от количества элементов, идентифицированных выражением $("p.appendToMe"): в случае единственного элемента в наборе будет выполнена операция перемещения, а если принимающих элементов больше, то операция копирования.

Если нам нужно переместить или скопировать элемент из одного места в другое, проще всего сделать это с помощью команды appendTo(), которая позволяет получить элемент и переместить его в дереве DOM.

Синтаксис команды appendTo

`appendTo(target)`

Перемещает все элементы в обернутом наборе в конец содержимого элемента, заданного параметром target.

Параметры

target (строка | элемент) Строка содержит селектор jQuery или элемент DOM. Каждый элемент обернутого набора будет добавлен в конец по этому указанию. Если селектору в строке соответствует больше одного элемента, исходный элемент будет скопирован и добавлен к каждому элементу, соответствующему селектору.

Возвращаемое значение

Обернутый набор.

Все функции, представленные в этом разделе, похожи в том смысле, что элемент *перемещается*, если в качестве места назначения идентифицирован единственный элемент. Если в качестве места назначения идентифицировано несколько элементов, то исходный элемент остается в прежнем местоположении и будет *скопирован* в каждый принимающий элемент.

Прежде чем перейти к другим командам, работающим сходным образом, для окончательного прояснения этой очень важной концепции рассмотрим пример. Мы создали лабораторную страницу, несколько элементов которой выступают в качестве исходных элементов команды appendTo(), а другие несколько элементов служат местом назначения. Сразу после открытия в окне броузера лабораторная страница Move and Copy Laboratory Page выглядит, как показано на рис. 3.5.

HTML-разметка испытуемых объектов в виде двух групп элементов выглядит так:

```
<fieldset id="source">
  <legend>Source elements</legend>
  <img id="flower" src="flower.png"/>
  <img id="car" src="car.png"/>
```

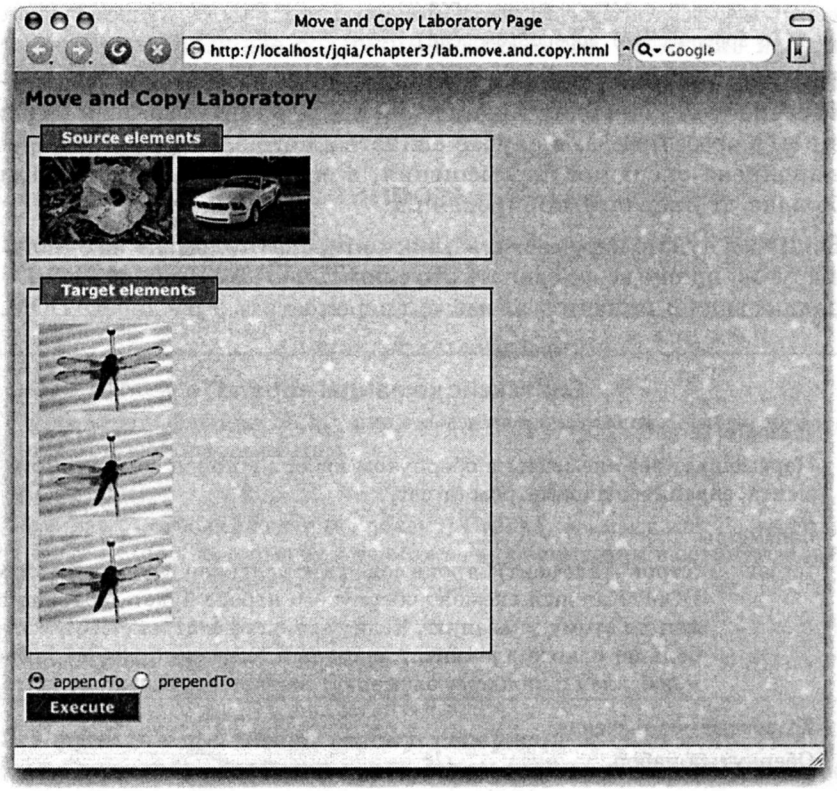


Рис. 3.5. *Лабораторная страница Move and Copy Laboratory Page иллюстрирует операции, выполняемые командами appendTo и prependTo*

```
</fieldset>

<fieldset id="targets">
  <legend>Target elements</legend>
  <p><img src="dragonfly.png"/></p>
  <p><img src="dragonfly.png"/></p>
  <p><img src="dragonfly.png"/></p>
</fieldset>
```

Группа-источник содержит два изображения: одно со значением атрибута `id="flower"`, а второе – со значением атрибута `id="car"`. Эти элементы-изображения служат для команд источниками. Группа-приемник содержит три элемента `<p>`, каждый из которых содержит изображение. Эти элементы-абзацы служат для наших команд местом назначения.

Откройте в броузере эту страницу, которая находится в файле *chapter3/lab.move.and.copy.html*. Оставьте отмеченным переключатель appendTo и щелкните на кнопке Execute (выполнить). В результате будет выполнен программный код, эквивалентный следующему:

```
$('#flower').appendTo('#targets p')
$('#car').appendTo('#targets p:first')
```

Первая инструкция выполнит команду appendTo() для изображения цветка, в качестве места назначения этой команды определены три абзаца. Так как здесь больше одного места назначения, можно ожидать, что изображение цветка будет скопировано. Вторая инструкция выполнит ту же самую команду для изображения автомобиля, но в качестве места назначения определен единственный абзац, поэтому можно ожидать, что изображение будет перемещено.

Снимок экрана (рис. 3.6), сделанный после щелчка на кнопке Execute, показывает, что эти ожидания оправдались.

Этот результат наглядно показал, что если мест назначения несколько, то исходный элемент копируется, а если одно, то перемещается.

Многие родственные команды работают так же, как команды append() и appendTo():

- prepend() и prependTo() – Работают подобно функциям append() и appendTo(), но вставляют элемент-источник перед содержимым эле-

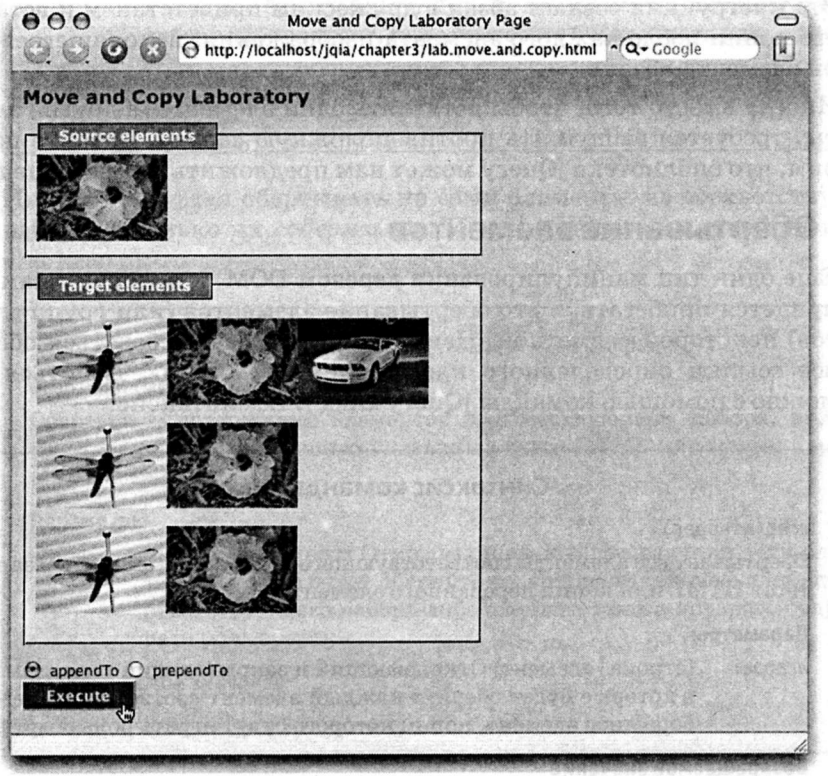


Рис. 3.6. *После выполнения команд видно, что изображение автомобиля было перемещено, а изображение цветка – скопировано*

мента места назначения, а не после. Действие этих команд также демонстрируется на лабораторной странице Move and Copy Laboratory Page, надо лишь выбрать переключатель prependTo перед щелчком на кнопке Execute.

- `before()` и `insertBefore()` – Вставляет элемент-источник перед самим указанным элементом, а не перед первым его дочерним элементом.
- `after()` и `insertAfter()` – Вставляет элемент-источник после самого указанного элемента, а не после его последнего дочернего элемента.

Так как синтаксис этих команд очень похож на синтаксис команд `append()` и `appendTo()`, мы не приводим его. За информацией о синтаксисе этих команд обращайтесь к описанию команд `append()` и `appendTo()`.

Прежде чем двинуться дальше, хочется добавить еще кое-что.

Помните, как в предыдущей главе мы рассматривали порядок создания новых фрагментов HTML с помощью функции-обертки `$()`? Этот прием становится по-настоящему полезным в соединении с командами `appendTo()`, `prependTo()`, `insertBefore()` и `insertAfter()`. Например:

```
$('<p>Hi there!</p>').insertAfter('p img');
```

Эта инструкция создает абзац с дружеским приветствием и вставляет его копии после каждого элемента-изображения, находящегося внутри элемента-абзаца.

Иногда вместо того, чтобы вставлять одни элементы в другие элементы, требуется выполнить противоположную задачу. Давайте посмотрим, что библиотека jQuery может нам предложить для ее решения.

3.3.3. Обертывание элементов

Еще один тип манипулирования деревом DOM, к которому нам часто придется прибегать, – это обертывание элементов (или групп элементов) некоторой разметкой. Например, может потребоваться обернуть все ссылки определенного класса элементом `<div>`. Добиться этого можно с помощью команды jQuery `wrap()`. Ее синтаксис:

Синтаксис команды wrap

`wrap(wrapper)`

Обертывает все элементы соответствующего набора с помощью указанных тегов HTML или копии переданного элемента.

Параметры

`wrapper` (строка | элемент) Открывающий и закрывающий теги элемента, в которые будет обернут каждый элемент соответствующего набора либо элемент, копии которого будут играть роль обертки.

Возвращаемое значение

Обернутый набор.

Обернуть каждую ссылку с классом `surprise` элементом `<div>` с классом `hello` можно так:

```
$("a.surprise").wrap("<div class='hello'></div>")
```

А если требуется обернуть ссылку копией первого на странице элемента `<div>`, это делается так:

```
$("a.surprise").wrap($("div:first")[0]);
```

Если в обернутом наборе несколько элементов, команда `wrap()` обрабатывает каждый из них по отдельности. Обернуть все элементы набора как единое целое поможет метод `wrapAll()`:

Синтаксис команды wrapAll

`wrapAll(wrapper)`

Обертывает элементы соответствующего набора как единое целое с помощью указанных тегов HTML или копии переданного элемента.

Параметры

`wrapper` (строка | элемент) Открывающий и закрывающий теги элемента, в которые будут обернуты сразу все элементы соответствующего набора либо элемент, копия которого будет играть роль обертки.

Возвращаемое значение

Обернутый набор.

Иногда требуется обертывать не сами элементы из соответствующего набора, а только их *содержимое*. Именно на такой случай есть метод `wrapInner()`:

Синтаксис команды wrapInner

`wrapInner(wrapper)`

Обертывает содержимое элементов соответствующего набора, включая текстовые узлы, с помощью указанных тегов HTML или копии переданного элемента.

Параметры

`wrapper` (строка | элемент) Открывающий и закрывающий теги элемента, в которые будет обернуто содержимое каждого элемента соответствующего набора либо элемент, копии которого будут играть роль обертки.

Возвращаемое значение

Обернутый набор.

Узнав, как создавать, обертывать, копировать и перемещать элементы, можно заняться вопросом их удаления.

3.3.4. Удаление элементов

Очистить или удалить набор элементов можно с помощью команды `remove()`. Ее синтаксис:

Синтаксис команды remove

`remove()`

Удаляет все элементы обернутого набора из страницы.

Параметры

Нет

Возвращаемое значение

Обернутый набор.

Обратите внимание: как и многие другие команды библиотеки jQuery, этот метод также возвращает обернутый набор. Элементы, удаленные из дерева DOM, по-прежнему доступны через обернутый набор (по этой причине они не могут быть удалены из памяти «сборщиком мусора») и могут дальше участвовать в операциях в других командах jQuery, включая `appendTo()`, `prependTo()`, `insertBefore()`, `insertAfter()` и все другие сходные команды.

Очистить содержимое элементов DOM можно с помощью команды `empty()`. Ее синтаксис:

Синтаксис команды empty

`empty()`

Удаляет содержимое всех элементов DOM в соответствующем наборе.

Параметры

Нет

Возвращаемое значение

Обернутый набор.

Обычно с помощью команды `remove()` и `after()` выполняется замена. Например:

```
$("div.elementToReplace").after("<p>I am replacing the div</p>").remove();
```

Так как здесь функция `after()` возвращает первоначальный обернутый набор, содержащий элемент `<div>`, мы можем просто удалить его, в результате элемент `<div>` окажется замещен вновь созданным элементом `<p>`.

Если окажется, что вы применяете эту операцию снова и снова, вспомните, что такие полезные инструкции всегда можно оформить в виде расширений jQuery:

```
$.fn.replaceWith = function(html) {
  return this.after(html).remove();
};
```

Тогда операцию из предыдущего примера можно реализовать так:

```
$("div.elementToReplace").replaceWith("<p>I am replacing the div</p>");
```

Вот мы и создали еще одно расширение jQuery. Очень важный момент здесь – возврат обернутого набора, чтобы после вызова расширения цепочка команд могла быть продолжена (если цель расширения не в том, чтобы вернуть другие данные). Вы должны приобретать навыки расширения jQuery, но не стоит пока волноваться, если вам что-то непонятно – более полное описание вы найдете позже в этой книге (точнее, в главе 7).

Иногда требуется не переместить элементы, а скопировать их.

3.3.5. Копирование элементов

Один из множества способов манипулирования деревом DOM – копирование элементов с их последующим присоединением к другим участкам дерева DOM. Для выполнения этой операции библиотека jQuery предоставляет удобный метод-обертку – команду `clone()`.

Синтаксис команды clone

`clone(copyHandlers)`

Создает копии элементов обернутого набора и возвращает обернутый набор копий. Копируются как сами элементы, так и все вложенные в них элементы. В зависимости от значения параметра `copyHandlers` также могут копироваться обработчики событий.

Параметры

`copyHandlers` (логическое значение) Если параметр имеет значение `true`, вместе с элементами будут скопированы обработчики событий. Если параметр имеет значение `false`, обработчики событий копироваться не будут.

Возвращаемое значение

Новый обернутый набор.

Копировать существующие элементы с помощью `clone()` мало проку, если после ничего с этими копиями не делать. Обычно вслед за получением обернутого набора с копиями элементов выполняется другая команда библиотеки jQuery, которая присоединяет этот набор в какое-нибудь другое место дерева DOM. Например:

```
$('img').clone().appendTo('fieldset.photo');
```

Данная инструкция создает копии всех элементов-изображений и добавляет их ко всем элементам <fieldset> с классом photo.

Вот чуть более сложный пример:

```
$('ul').clone().insertBefore('#here');
```

Эта цепочка команд выполняет похожую операцию, только здесь создаются копии всех элементов <ul>, включая вложенные элементы (скорее всего, каждый элемент <ul> содержит несколько вложенных элементов <li>).

И последний пример:

```
$('ul').clone().insertBefore('#here').end().hide();
```

Эта инструкция выполняет ту же операцию, что и в предыдущем примере, но после вставки копий выполняется команда end(), которая осуществляет возврат к первоначальному обернутому набору (исходные элементы), и затем элементы первоначального набора делаются невидимыми с помощью команды hide(). Это наглядно показывает, что в ходе выполнения операции копирования создается новый набор элементов в новой обертке.

Обсудив способы манипулирования элементами DOM, рассмотрим возможности манипулирования особой группой элементов – элементами форм.

3.4. Обработка значений элементов форм

У элементов форм есть особые свойства, поэтому библиотека jQuery содержит ряд удобных функций для получения и изменения их значений, для их сериализации и для выбора элементов на основе свойств формы. Их вполне можно использовать в простых случаях, однако не забудем о подключаемом модуле Form Plugin – он официально одобрен и разрабатывается членами группы jQuery Core Team, – который предоставляет намного более широкие возможности. Модуль Form Plugin рассматривается в главе 9.

Примечание

Элементами формы мы называем элементы, которые могут появляться внутри форм и имеют атрибуты name и value, передаваемые серверу в виде параметров запроса при отправке формы. Обрабатывать такие элементы в сценариях не всегда просто, и не только потому, что элементы могут быть деактивизированы, но и потому, что они могут оказаться в состоянии *неуспеха* (*unsuccessful*), определенном консорциумом W3C для элементов управления. Это состояние определяет элементы, которые должны игнорироваться при передаче, что осложняет их обработку.

Теперь рассмотрим одну из наиболее типичных операций, применяемых к элементам форм: получение доступа к их значениям. Команда `val()` учитывает большинство возможных ситуаций и возвращает атрибут `value` элемента формы для первого элемента в обернутом наборе. Ее синтаксис:

Синтаксис команды val

`val()`

Возвращает свойство `value` первого элемента в соответствующем наборе. Если элемент предоставляет возможность множественного выбора, возвращаемое значение является массивом всех выбранных вариантов.

Параметры

Нет

Возвращаемое значение

Извлеченное значение или значения.

У этой команды, хотя и весьма полезной, есть ряд ограничений, из-за чего ее следует применять особенно осторожно. Если первый элемент в обернутом наборе не является элементом формы, это вызовет ошибку JavaScript. Кроме того, эта команда не различает отмеченные и не отмеченные состояния флажков (checkboxes) и переключателей (radio-buttons), и возвратит значение атрибута `value` этих элементов независимо от того, были они отмечены или нет.

При работе с переключателями сэкономить время на отладке поможет совместное применение селекторов jQuery и метода `val()`. Рассмотрим форму с группой переключателей (объединенных общим названием) под именем `radioGroup` и следующее выражение:

```
$('[name=radioGroup]:checked').val()
```

Это выражение вернет значение единственного отмеченного переключателя (или значение `undefined`, если ни один из них не отмечен). Это намного проще, чем обходить все переключатели в цикле, чтобы отыскать отмеченный элемент, правда? Поскольку команда `val()` работает только с первым элементом в обернутом наборе, этот прием не так полезен при работе с группой флажков, где отмеченными могут оказаться сразу несколько элементов управления.

Если требуется получить значения, с которыми элементы управления будут отправлены на сервер вместе с формой, то гораздо лучше подойдет команда `serialize()` (рассматривается в главе 8) или официальный модуль расширения Form Plugin.

Еще одна типичная операция, которую нам придется выполнять, – установка значения элемента формы. Это также делается с помощью

команды `val()`, но при этом ей передается требуемое значение. Синтаксис команды:

Синтаксис команды val

`val(value)`

Устанавливает значение параметра `value` как значение всех соответствующих элементов формы.

Параметры

`value` (строка) Строка, которая устанавливается как значение свойства `value` каждого элемента формы в обернутом наборе.

Возвращаемое значение

Обернутый набор.

Как и в случае получения значения, у этой функции есть ограничения. Она не может устанавливать несколько значений, например, в списках, допускающих множественный выбор. Это одна из причин, по которым предпочтительнее использовать мощный модуль расширения Form Plugin. Кроме устранения упомянутых ограничений, этот модуль способен выполнять такие операции, как получение массива значений группы флажков, сериализация элементов в обернутом наборе, очистка полей ввода и даже преобразование дерева DOM формы в формат, совместимый с технологией Ajax.

Вот еще один вариант метода `val()`, позволяющий отмечать флажки и переключатели или выбирать варианты в элементе `<select>`. Этот вариант команды `val()` имеет следующий синтаксис:

Синтаксис команды val

`val(values)`

Отмечает (или выбирает) все флажки, переключатели или варианты выбора в элементах `<select>`, входящих в обернутый набор, если их значения соответствуют одному из значений, переданных в виде массива в параметре `values`.

Параметры

`values` (массив) Массив значений, с помощью которых определяются отмечаемые элементы.

Возвращаемое значение

Обернутый набор.

Рассмотрим следующую инструкцию:

```
$('input,select').val(['one','two','three']);
```

Эта инструкция отыскивает на странице все элементы <input> и <select> со значениями, совпадающими с любой из строк *one*, *two* или *three*, и отмечает все найденные флажки или переключатели либо выбирает все найденные варианты.

Благодаря этому команда val() применима не только к простым текстовым элементам.

3.5. Итоги

В этой главе мы возвысились над искусством выбора элементов и начали манипулировать ими. Приемы, которые мы здесь обсудили, позволяют отбирать элементы с помощью мощных критериев и затем, путем хирургического вмешательства, перемещать их в любое место страницы.

Мы можем копировать элементы, перемещать их и даже создавать на пустом месте совершенно новые элементы. Мы можем добавлять их в конец, в начало, а также обертывать любой элемент или наборы элементов на странице. Мы узнали, что одни и те же приемы можно применять как к единственному элементу, так и к набору элементов, что позволяет писать краткие, но мощные инструкции.

Обладая всеми этими знаниями, мы готовы приступить к изучению более сложных концепций, начав с такой грязной работы, как обработка событий.

4

В этой главе:

- Как реализована модель событий в броузерах
- Как с помощью jQuery подключать обработчики событий к элементам
- Что такое экземпляр объекта Event
- Как вызывать обработчики событий из сценариев

События: где это происходит

Если вы знакомы с бродвейским шоу «Кабаре» или его голливудской экранизацией, то вспомните песню «Money makes the world go around» (деньги заставляют вращаться мир). Эта циничная точка зрения относится к миру физическому, а в виртуальном мире Всемирной паутины все происходящее обусловлено событиями!

Как и многие другие системы, управляющие графическим интерфейсом пользователя, интерфейсы, представленные веб-страницами HTML, являются *асинхронными* и *управляются событиями* (даже при том, что протокол HTTP, по которому осуществляется их доставка, по сути является синхронным). Независимо от того, как реализован графический интерфейс, – настольным приложением с помощью Java Swing, X11, .NET Framework или в виде страницы веб-приложения с применением HTML и JavaScript, – алгоритм остается неизменным:

1. Создать пользовательский интерфейс.
2. Ждать, пока что-то произойдет.
3. Соответственно отреагировать на событие.
4. Вернуться к началу.

На первом шаге пользовательский интерфейс *появляется* на экране. Остальные шаги определяют его *поведение*. В веб-страницах интерфейс выводится на экран броузером в ответ на получение разметки (HTML и CSS). А поведение интерфейса определяется сценарием, подключенным к странице.

Этот сценарий принимает форму *обработчиков событий*, также называемых *слушателями* (*listeners*), которые реагируют на различные события, возникающие в ходе отображения страницы. Эти события генерируются системой (таймерами или по завершении асинхронных

запросов), но чаще – в результате действий пользователя (например, перемещение указателя мыши, нажатие кнопки или ввод текста с клавиатуры). Если бы Всемирная паутина не умела реагировать на эти события, вся ее прелесть заключалась бы в показе картинок с котятами.

Сам язык разметки HTML *определяет* весьма небольшой набор встроенных семантических действий, не требующих сценария (таких, как перезагрузка страницы по щелчку на ссылке или отправка формы по щелчку на кнопке), но любое другое нужное поведение страницы требует от нас обработки различных событий, возникающих в результате взаимодействия пользователя со страницей.

В этой главе мы исследуем способы представления этих событий в броузерах, позволяющих нам обрабатывать их для контроля над происходящим, и те трудности, с которыми нам приходится сталкиваться из-за различий между разными типами броузеров. А затем вы увидите, как jQuery рассеет туман, нагоняемый броузерами, и освободит нас от этих трудностей.

JavaScript нужно знать!

Одно из основных преимуществ, привносимых библиотекой jQuery в веб-приложения, – возможность реализовать широкую функциональность без необходимости писать большие объемы программного кода. Все тонкости реализации берет на себя jQuery, позволяя нам сконцентрироваться на главной задаче – заставить наши приложения делать именно то, что они должны делать!

До этого момента мы двигались почти свободно. Чтобы создавать свой программный код и понимать примеры из предыдущих глав, вам было вполне достаточно азов языка JavaScript. В этой и следующих главах для эффективного использования библиотеки jQuery вам *понадобится* понимать фундаментальные концепции JavaScript.

Вполне возможно, что вы уже знакомы с этими концепциями, но некоторые авторы страниц могут много написать на JavaScript, при этом не понимая, что именно происходит за кулисами, – гибкость JavaScript вполне допускает это. Прежде чем продолжить, проверим, насколько точно вы понимаете эти базовые концепции.

Если вы без труда справляетесь с классами JavaScript, такими как `Object` и `Function` и хорошо понимаете такие концепции, как *контекст функции* и *замыкания* (*closures*), то можете продолжить чтение этой и последующих глав. Если эти концепции вам неизвестны или неясны, настоятельно рекомендуем обратиться к Приложению, которое поможет вам быстро изучить эти необходимые понятия.

Давайте начнем наше исследование с того, как броузеры реализуют свои модели событий.

4.1. Модель событий броузера

Задолго до первых попыток стандартизировать обработку событий в броузерах корпорация Netscape Communications представила модель обработки событий в броузере Netscape Navigator – эту модель, пожалуй, одну из самых понятных, используют многие авторы страниц и продолжают поддерживать все современные броузеры.

Эту модель называют по-разному. Вам могла встретиться «модель событий Netscape» (Netscape Event Model), «базовая модель событий» (Basic Event Model), а то и вовсе расплывчатая «модель событий броузера» (Browser Event Model). Однако для большинства это – модель событий DOM уровня 0 (DOM Level 0 Event Model).

Примечание

Уровень DOM (*DOM Level*) определяет уровень соответствия спецификации W3C DOM Specification. Нет никакого DOM нулевого уровня – здесь этот термин нужен лишь для неформального указания того, что реализация этой модели предшествовала DOM уровня 1 (DOM Level 1).

Модель обработки событий не была стандартизована консорциумом W3C до появления спецификации DOM уровня 2 (DOM Level 2) в ноябре 2000 года. Эту модель поддерживают все современные броузеры, совместимые со стандартами, такие как Firefox, Camino (как и все остальные броузеры, созданные на базе Mozilla), Safari и Opera. Internet Explorer продолжает идти своим путем, поддерживая модель событий DOM уровня 2 (DOM Level 2 Event Model) лишь частично и в основном используя собственные интерфейсы.

Прежде чем посмотреть, как jQuery ликвидирует раздражающий фактор, обусловленный различием броузеров, потратим некоторое время на изучение работы моделей событий.

4.1.1. Модель событий DOM уровня 0

Модель событий DOM уровня 0 (The DOM Level 0 Event Model) – пожалуй, самая популярная среди веб-разработчиков. Кроме того, что она достаточно независима от типа броузера и удобна.

Согласно этой модели, обработчики событий объявляются присваиванием ссылок на экземпляры функций свойствам элементов DOM. Это специальные свойства, предназначенные для организации обработки событий определенных типов, например, щелчок мышью обрабатывается функцией, связанной со свойством `onclick`, а перемещение указателя мыши на элемент – функцией, связанной со свойством `onmouseover` элемента, поддерживающего эти типы событий.

Броузеры позволяют определять функции в виде значений атрибутов в HTML-разметке элементов DOM, что обеспечивает краткую форму записи обработчиков событий. Пример такого определения обработчиков приведен в листинге 4.1. Эту страницу вы найдете в загружаемом пакете с примерами, в файле *chapter4/dom.0.events.html*.

Листинг 4.1. Объявление обработчиков событий DOM уровня 0

```
<html>
  <head>
    <title>DOM Level 0 Events Example</title>
    <script type="text/javascript"
            src="../scripts/jquery-1.2.1.js">
    </script>
    <script type="text/javascript">
      $(function(){
        $('#vstar')[0].onmouseover = function(event) {
          say('Whee!');
        }
      });

      function say(text) {
        $('#console').append('<div>'+new Date()+' '+text+'</div>');
      }
    </script>
  </head>

  <body>
    <img id="vstar" src="vstar.jpg"
         onclick="say('Vroom vroom!');"/>
    <div id="console"></div>
  </body>
</html>
```

❶ В обработчике готовности документа объявляется обработчик onmouseover

❷ Вспомогательная функция вывода текста в консоль

❸ Элемент <img>, где используется обработчик

❹ Элемент , играющий роль консоли

В этом примере используются два стиля объявления обработчиков событий: объявление в блоке сценария и объявление в HTML-разметке.

Сначала на странице объявляется обработчик события готовности документа ❶, который получает ссылку на элемент-изображение со значением атрибута `id`, равным `vstar` (средствами jQuery), а затем присваивает его свойству `onmouseover` ссылку на встроенную функцию. Эта функция становится обработчиком события перемещения указателя мыши на этот элемент. Обратите внимание: эта функция должна получить единственный входной параметр. Его мы подробно рассмотрим чуть ниже.

Здесь также объявляется вспомогательная функция `say()` ❷, которая служит для вывода текстовых сообщений в элемент `<div>` ❹ страницы, – это позволит нам не беспокоиться о выводе диалога, обозначающего факт события.

В теле страницы (помимо элемента-консоли) определяется элемент-изображение ❸, в котором задается обработчик события. Мы уже видели, как определяется обработчик в сценарии, в обработчике события

готовности документа ❶, а здесь объявляется обработчик события щелчка мышью – с помощью атрибута `onclick` элемента `<img>`.

Откройте эту страницу в броузере (она находится в файле *chapter4/dom.0.events.html*), проведите несколько раз указатель мыши над изображением и затем щелкните на нем. Результат должен получиться таким, как показано на рис. 4.1.

Мы объявили обработчик события щелчка мышью в разметке элемента `<img>` с помощью следующего атрибута:

```
onclick="say('Vroom vroom!');"
```

Можно было бы подумать, что обработчиком события щелчка мышью для этого элемента является функция `say()`, но это не так. Если обработчик события объявлен в атрибуте разметки, автоматически создается анонимная функция, телом которой является значение атрибута. Результат такого объявления эквивалентен следующему фрагменту (если предположить, что `imageElement` – это ссылка на элемент-изображение):

```
imageElement.onclick = function(event) {
  say('Vroom vroom!');
}
```

Обратите внимание: значение атрибута используется в качестве тела сгенерированной функции, кроме того, функция создана так, что внутри нее доступен параметр `event`.

Прежде чем перейти к исследованию этого параметра, следует отметить, что использование механизма объявления обработчиков событий в модели DOM уровня 0 нарушает принципы ненавязчивого JavaScript, рассмотренные в разд. 1.2. Применяя в своих страницах jQuery, мы должны стараться придерживаться принципов ненавязчивого JavaScript, не смешивая определение поведения элементов на языке JavaScript с отображаемой разметкой. Далее, ближе к концу главы,

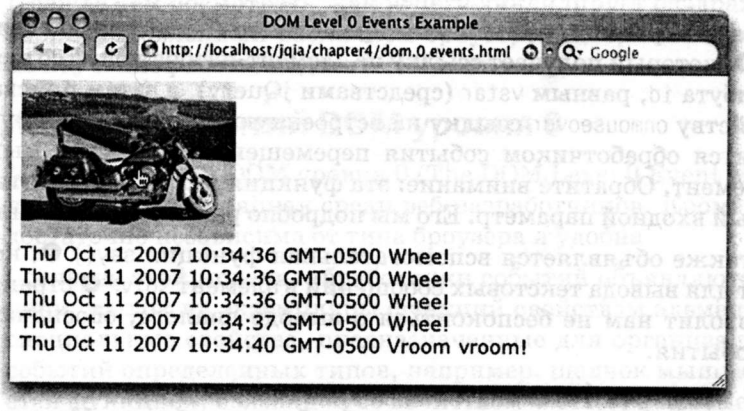


***Рис. 4.1.** Перемещение указателя мыши на изображение и щелчок приводят к запуску обработчиков событий и появлению сообщений в консоли*

мы увидим, что jQuery предоставляет другой, лучший способ объявления обработчиков событий.

Но сначала выясним все подробности о параметре event.

Экземпляр объекта Event

В большинстве броузеров при вызове обработчика события в первом входном параметре ему передается экземпляр объекта Event. Internet Explorer всегда шел другим путем, многое реализуя по-своему, поэтому в нем объект Event помещается в свойство event объекта window.

Из-за этого несоответствия нам часто будет встречаться в виде первой строки обработчика события такой фрагмент:

```
if (!event) event = window.event;
```

Определение наличия параметра event и присваивание ему значения свойства event объекта window нивелирует различия между броузерами. После этой строки параметр event может использоваться независимо от того, каким образом он стал доступен обработчику события.

Свойства экземпляра объекта Event содержат массу информации о событии, обрабатываемом в настоящий момент. Это такие сведения, как информация об элементе, вызвавшем событие, координаты указателя для событий от мыши и нажатые клавиши для событий от клавиатуры.

Но все не так просто. Internet Explorer не только по-своему определяет экземпляр Event в обработчиках событий, он еще использует собственное, а не определенное стандартом W3C определение класса Event – и мы все еще в путах различий между объектами.

Например, чтобы получить ссылку на тот элемент, где возникло событие, мы должны обращаться к свойству target в броузерах, соответствующих стандартам, и к свойству srcElement – в Internet Explorer. Обойти это несоответствие позволит следующая инструкция:

```
var target = (event.target) ? event.target : event.srcElement;
```

Эта инструкция проверяет наличие свойства event.target: если оно присутствует, то его значение присваивается локальной переменной target; в противном случае переменной присваивается значение свойства event.srcElement. Аналогично мы должны поступать и с другими свойствами объекта Event, представляющими для нас интерес.

Вплоть до этого момента мы действовали так, как если бы вызывались только обработчики событий элемента, вызывавшего событие (как элемент-изображение в листинге 4.1, например). Но события распространяются по всему дереву DOM. Давайте рассмотрим это.

Всплытие события

Когда в элементе дерева DOM возникает событие, механизм обработки событий броузера проверяет наличие обработчика данного события

в самом элементе, и если таковой определен, вызывает его. Но история на этом не заканчивается.

После того как элемент получает свой шанс обработать событие, модель управления событиями проверяет наличие обработчика событий данного типа в родительском элементе, и если в *этом* элементе присутствует обработчик событий данного типа, он также будет вызван; затем проверяется наличие обработчика событий у родителя уже *этого* элемента, затем у его родителя и т. д., пока не будет достигнута вершина дерева DOM. Поскольку события распространяются вверх (если считать, что корень дерева DOM находится наверху), как пузырьки в бокале шампанского, этот процесс назвали *всплытием события*.

Изменим пример из листинга 4.1, чтобы увидеть этот процесс в действии. Новый программный код примера приводится в листинге 4.2.

Листинг 4.2. Распространение событий от места их появления вверх по дереву DOM

```
<html id="greatgreatgrandpa">
  <head>
    <title>DOM Level 0 Events Example</title>
    <script type="text/javascript"
            src="../scripts/jquery-1.2.1.js">
    </script>
    <script type="text/javascript">
      $(function(){
        $('*').each(function(){
          var current = this;
          this.onclick = function(event) {
            if (!event) event = window.event;
            var target = (event.target) ?
                         event.target : event.srcElement;
            say('For ' + current.tagName + '#'+ current.id +
                ' target is ' + target.id);
          }
        });
      });

      function say(text) {
        $('#console').append('<div>'+text+'</div>');
      }
    </script>
  </head>

  <body id="greatgrandpa">
    <div id="grandpa">
      <div id="pops">
        <img id="vstar" src="vstar.jpg"/>
      </div>
    </div>
    <div id="console"></div>
  </body>
</html>
```

❶ Выбор всех элементов страницы

❷ Применение обработчика onclick к каждому выбранному элементу

В эту страницу мы внесли множество интересных изменений. Прежде всего, мы полностью удалили обработку события перемещения указателя мыши, чтобы все свое внимание сосредоточить на событии щелчка. Мы также включили элемент-изображение, который будет играть роль объекта исследований, в два элемента , чтобы глубже поместить изображение в иерархию DOM. Еще мы присвоили уникальные идентификаторы (атрибут id) почти всем элементам страницы – даже тегам <body> и <html>!

Мы сохранили консоль и вспомогательную функцию say() для вывода отчетов, как и в предыдущем примере.

Теперь рассмотрим более интересные изменения.

В обработчике события готовности документа для страницы мы используем jQuery, чтобы выбрать все элементы страницы, и обходим их с помощью метода each() ❶. Для каждого соответствующего элемента мы присваиваем ссылку на него в локальную переменную current и устанавливаем обработчик onclick ❷. Этот обработчик сначала выполняет трюки, нивелирующие различия между броузерами, обсуждаемые в предыдущем разделе, чтобы определить местонахождение объекта Event и элемент, для которого это событие предназначено, а затем выводит сообщение в консоль. Текст сообщения – пожалуй, самая интересная часть данного примера.

Пример выводит имя тега и значение атрибута id текущего элемента, создавая *замыкание* (если не очень хорошо понимаете этот термин, пожалуйста, прочтите разд. А.2.4 приложения), после этого выводится значение атрибута id элемента, для которого предназначено событие. При таком подходе каждое сообщение в консоли будет содержать информацию о текущем элементе, участвующем в процессе всплытия события, а также об элементе, с которого все началось.

Откройте страницу (файл *chapter4/dom.0.propagation.html*) и щелкните на изображении. Результат приведен на рис. 4.2.

Этот пример наглядно показывает, что когда возникает событие, оно сначала доставляется элементу, для которого это событие предназначено, а затем каждому родительскому элементу, вплоть до элемента <html>.

Это весьма мощная возможность, потому что она позволяет нам устанавливать обработчики событий в элементы на любом уровне вложенности и обрабатывать события, предназначенные для дочерних элементов. Примером могут служить обработчики событий в элементе <form>, которые реагируют на любые изменения в дочерних элементах, чтобы вызвать динамические изменения в отображении формы на основе новых значений элементов.

Но как быть, если дальнейшее распространение события становится *нежелательным*? Как нам его *пресечь*?

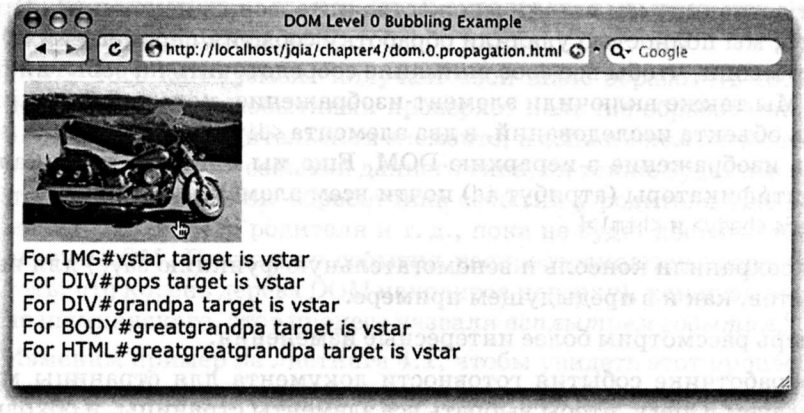


Рис. 4.2. *Сообщения в консоли наглядно демонстрируют, что события, как пузырьки, поднимаются вверх по дереву DOM от элемента, для которого предназначено событие, к корню дерева*

Влияние на распространение события и семантику

Могут возникать ситуации, когда требуется прекратить дальнейшее распространение события в дереве DOM. Например, если мы достаточно скрупулезны и точно знаем, что все необходимые действия по обработке события уже выполнены и нужно предотвратить возможную обработку этого события в дереве DOM выше.

Независимо от причин, побудивших нас к этому, мы можем остановить дальнейшее распространение события посредством механизмов, предоставляемых объектом `Event`. В броузерах, следующих стандартам, чтобы остановить дальнейшее распространение события вверх по иерархии родительских элементов, следует вызвать метод `stopPropagation()` экземпляра `Event`. В Internet Explorer нужно присвоить свойству `cancelBubble` экземпляра `Event` значение `true`. Самое интересное, что многие броузеры, следующие стандартам, поддерживают механизм `cancelBubble`, даже при том, что он не определен стандартом W3C.

Некоторые события обладают связанной с ними семантикой по умолчанию. Примером может служить событие щелчка на якорном элементе, которое вызывает переход броузера по ссылке, указанной в атрибуте `href`, или событие отправки формы в элементе `<form>`, вызывающее отправку формы. Если мы хотим отменить такую семантику событий, иногда называемую *действием по умолчанию*, нужно, чтобы обработчик события возвращал значение `false`.

Часто такая отмена происходит в ходе проверки на корректность заполнения полей формы. В обработчике события отправки формы мы можем реализовать проверку элементов формы `<input>` и возвращать значение `false`, если во введенных данных будут обнаружены какие-либо ошибки.

В элементах <form> можно также встретить следующий фрагмент:

```
<form name="myForm" onsubmit="return false;" ...
```

Этот способ эффективно препятствует отправке формы при любых обстоятельствах, возможна отправка только из сценария (при помощи метода form.submit(), который не вызывает появление события отправки) – этот прием получил наибольшее распространение в приложениях с поддержкой технологии Ajax, где вместо отправки формы выполняются асинхронные запросы.

При использовании модели событий DOM уровня 0 практически на каждом шаге, выполняемом в обработчике события, приходится учитывать тип броузера, чтобы определить, какое действие следует предпринять. Как же это утомительно! Но убирать аспирин еще рано – более продвинутая модель событий не проще.

4.1.2. Модель событий DOM уровня 2

Серьезный недостаток модели событий DOM уровня 0 состоит в том, что свойство, в котором хранится ссылка на функцию, играющую роль обработчика события, может хранить только одно значение, то есть для события каждого типа в элементе можно зарегистрировать только один обработчик. Если имеется два варианта действий, которые требуется выполнить по щелчку мыши на элементе, следующий фрагмент не позволит это реализовать:

```
someElement.onclick = doFirstThing;
someElement.onclick = doSecondThing;
```

Поскольку вторая операция присваивания заменит первое значение в свойстве onclick, при появлении события вызываться будет только функция doSecondThing(). Безусловно, мы могли обернуть эти две функции еще одной функцией, которая вызывала бы и ту и другую, но с увеличением сложности страниц, что более чем вероятно при создании полнофункциональных интернет-приложений, становится все сложнее и сложнее следить за такими ситуациями. К тому же используемые в страницах компоненты многократного использования или библиотеки могут ничего не знать о необходимости дополнительной обработки событий других компонентов.

Мы могли бы найти другие решения: реализовать шаблон проектирования Observable (наблюдатель), определяющий схему издания/подписки для обработчиков событий, или даже применить трюк с замыканиями. Но все это только усложнило бы и без того достаточно сложные страницы.

Помимо *стандартной* модели событий была разработана модель событий DOM уровня 2, которая решает проблемы такого рода. Давайте посмотрим, как можно установить обработчик события в элемент DOM – и даже несколько обработчиков одного и того же события, в более современной модели.

Установка обработчиков событий

Вместо того чтобы присваивать ссылку на функцию свойству элемента, в модели событий DOM уровня 2 обработчики, также называемые *слушателями* (*listeners*), устанавливаются с помощью *метода* элемента. В каждом элементе DOM объявляется метод с именем `addEventListener()`, предназначенный для подключения обработчиков событий к элементу. Формат этого метода:

```
addEventListener(eventType,listener,useCapture)
```

Параметр `eventType` – это строка, идентифицирующая тип события. В общем случае она совпадает с именами событий, которые используются в модели DOM уровня 0, но без префикса *on*: например, `click`, `mouseover`, `keydown` и т. д.

Параметр `listener` – это ссылка на функцию (или встроенная функция), которая подключается к элементу как обработчик события указанного типа. Как и в базовой модели событий, в эту функцию в первом параметре передается экземпляр `Event`.

Последний параметр `useCapture` – это логическое значение, которое мы рассмотрим немного ниже при обсуждении порядка распространения события в модели DOM уровня 2. А пока просто будем считать, что это значение `false`.

Давайте еще раз изменим пример из листинга 4.1, применив более совершенную модель событий. Мы сконцентрируем свое внимание только на одном типе событий – щелчок мышью. На этот раз мы установим в элемент-изображение три обработчика одного и того же события. Программный код примера можно найти в файле *chapter4/dom.2.events.html* и он же приводится в листинге 4.3.

Листинг 4.3. Установка обработчиков событий в модели DOM уровня 2

```
<html>
  <head>
    <title>DOM Level 2 Events Example</title>
    <script type="text/javascript"
            src="../scripts/jquery-1.2.1.js">
    </script>
    <script type="text/javascript">
      $(function(){
        var element = $('#vstar')[0];
        element.addEventListener('click',function(event) {
          say('Whee once!');
        },false);
        element.addEventListener('click',function(event) {
          say('Whee twice!');
        },false);
        element.addEventListener('click',function(event) {
          say('Whee three times!');
        },false);
```

❶ Устанавливает три обработчика событий!

```
    });
    function say(text) {
      $('#console').append('<div>'+text+'</div>');
    }
  </script>
</head>
<body>
  <img id="vstar" src="vstar.jpg"/>
  <div id="console"></div>
</body>
</html>
```

Этот программный код проще и он наглядно демонстрирует, как можно установить в один элемент сразу несколько обработчиков для одного и того же события, – чего не позволяет базовая модель событий. В обработчике события готовности документа ❶ мы получаем ссылку на элемент-изображение и затем устанавливаем сразу *три* обработчика для события щелчка мышью.

Откройте страницу в броузере, соответствующем стандартам (то есть *не* в Intenet Explorer), и щелкните на изображении. Результат приведен на рис. 4.3.

Обратите внимание: хотя обработчики и были вызваны в том же порядке, в каком мы их установили, *этот порядок не гарантирован стандартами*! Еще никто не видел, чтобы обработчики вызывались в порядке, отличном от того, в каком они были установлены, но было бы довольно глупо писать программный код, который полагался бы на порядок вызова обработчиков. Вы должны знать, что в случае подключения сразу нескольких обработчиков для обслуживания одного и того же события они могут вызываться в произвольном порядке.

А теперь поговорим о параметре `useCapture`.

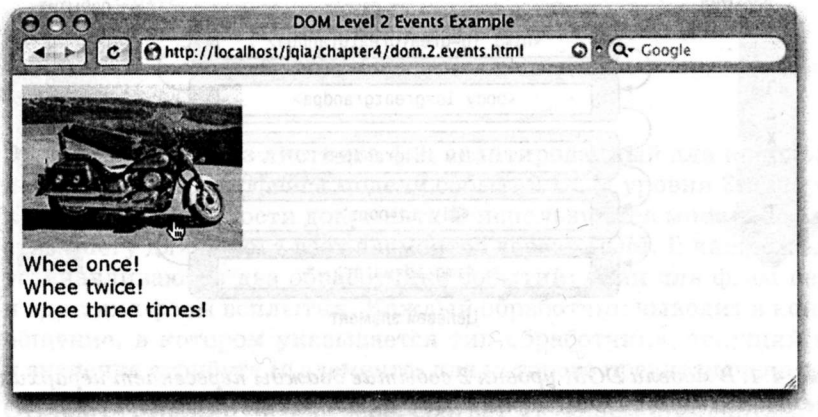


***Рис. 4.3.** Щелчок на изображении показывает, что были установлены все три обработчика события щелчка мышью*

Распространение события

Мы уже видели, что при использовании базовой модели событий сразу после возникновения события в элементе оно начинает распространяться вверх по дереву DOM и передается всем родительским элементам. Модель DOM уровня 2 также реализует фазу всплытия, но только после выполнения дополнительной фазы – *фазы захвата*.

В модели DOM уровня 2 при возникновении события оно сначала распространяется от корня дерева DOM вниз, к целевому элементу, а затем обратно вверх, к корню дерева DOM. Первая фаза (от корня дерева к целевому элементу) называется *фазой захвата*, а вторая (от целевого элемента к корню дерева) – *фазой всплытия*.

Функцию, установленную в качестве обработчика события, можно пометить как перехватывающий обработчик (вызывается в фазе захвата) или как обработчик всплывающего события (вызывается в фазе всплытия). Как вы уже наверняка поняли, параметр `useCapture` функции `addEventListener()` определяет тип устанавливаемого обработчика. Значение `false` этого параметра задает установку обработчика всплывающего события, а значение `true` – перехватывающего обработчика.

Вспомните пример из листинга 4.2, где мы исследовали распространение события по иерархии дерева DOM в базовой модели. В этом примере мы вложили элемент-изображение в два элемента `<div>`. В такой иерархии распространение события щелчка мышью на элементе `<img>` происходило бы, как показано на рис. 4.4.

Давайте проверим – так ли это? В листинге 4.4 приведен программный код страницы, содержащей иерархию элементов из рис. 4.4 (файл *chapter4/dom.2.propagation.html*).

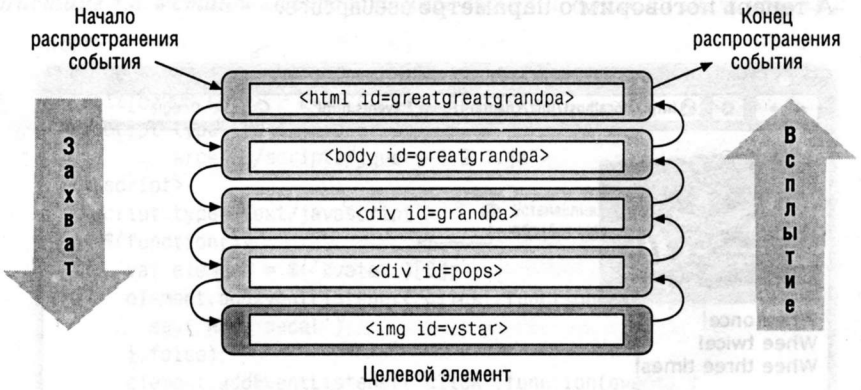


Рис. 4.4. В модели DOM уровня 2 событие дважды пересекает иерархию элементов дерева DOM: первый раз – в фазе захвата, от корня дерева к целевому элементу, и второй раз – в фазе всплытия, от целевого элемента к корню дерева

Листинг 4.4. Трассировка распространения события с помощью перехватывающих обработчиков и обработчиков всплывающих событий

```
<html id="greatgreatgrandpa">
  <head>
    <title>DOM Level 2 Propagation Example</title>
    <script type="text/javascript"
            src="../scripts/jquery-1.2.1.js">
    </script>
    <script type="text/javascript">
      $(function(){
        $('*').each(function(){
          var current = this;
          this.addEventListener('click',function(event) {
            say('Capture for ' + current.tagName + '#'+ current.id +
                ' target is ' + event.target.id);
            },true);
          this.addEventListener('click',function(event) {
            say('Bubble for ' + current.tagName + '#'+ current.id +
                ' target is ' + event.target.id);
          },false);
        });
      });

      function say(text) {
        $('#console').append('<div>'+text+'</div>');
      }
    </script>
  </head>

  <body id="greatgrandpa">
    <div id="grandpa">
      <div id="pops">
        <img id="vstar" src="vstar.jpg"/>
      </div>
    </div>
    <div id="console"></div>
  </body>
</html>
```

❶ Устанавливает обработчики события в элемент

Это код примера из листинга 4.2, адаптированный для использования прикладного интерфейса модели событий DOM уровня 2. В обработчике события готовности документа ❶ используются мощные возможности jQuery для обхода всех элементов дерева DOM. В каждом элементе устанавливаются два обработчика событий: один для фазы перехвата и один для фазы всплытия. Каждый обработчик выводит в консоль сообщение, в котором указывается тип обработчика, текущий элемент и значение атрибута `id` элемента, для которого предназначено событие.

Откройте страницу в броузере, соответствующем стандартам, и щелкните на изображении. По результату (рис. 4.5) видно, что событие преодолело две фазы распространения и дважды пересекло дерево DOM.

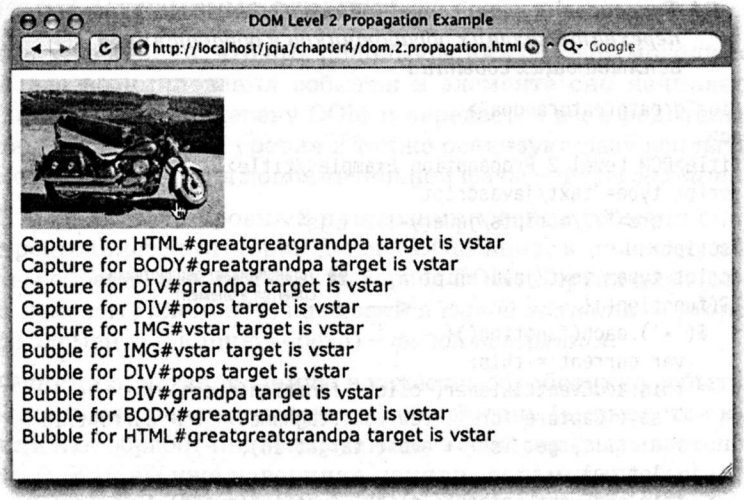


***Рис. 4.5.** После щелчка на изображении каждый обработчик этого события вывел в консоль сообщение, позволяющее отследить распространение события*

Обратите внимание: так как мы определили обработчики для двух фаз, для фазы перехвата и для фазы всплытия, было вызвано по два обработчика в целевом элементе и во всех его родительских узлах.

Теперь, когда мы преодолели все неприятности, чтобы разобраться в них, следует заметить, что перехватывающие обработчики событий практически никогда не используются в веб-страницах по той простой причине, что Internet Explorer (до сих пор неясно, как ему удается оставаться самым распространенным броузером) не поддерживает модель событий DOM уровня 2. Хотя у него *есть* собственная модель событий, соответствующая фазе всплытия модели DOM уровня 2, она не поддерживает фазу перехвата.

Прежде чем выяснить, какого рода помощь может оказать нам библиотека jQuery, давайте вкратце рассмотрим модель событий Internet Explorer.

4.1.3. Модель событий Internet Explorer

Броузер Internet Explorer (обе его версии – IE6 и, что самое неутешительное, IE7) не поддерживает модель событий DOM уровня 2. Обе версии броузера компании Microsoft реализуют собственный интерфейс, напоминающий фазу всплытия стандартной модели.

Вместо `addEventListener()` броузер Internet Explorer определяет для каждого элемента DOM метод с именем `attachEvent()`. Этот метод, как показано ниже, принимает два параметра, подобные первым двум параметрам метода стандартной модели:

```
attachEvent(eventName,handler)
```

Первый параметр – это строка с названием типа обрабатываемого события. Однако здесь используются не стандартные названия, а имена соответствующих свойств элемента из модели DOM уровня 0 – `onclick`, `onmouseover`, `onkeydown` и т. д.

Второй параметр – это функция, которая будет установлена как обработчик события. Как и в базовой модели, экземпляр `Event` придется извлекать из свойства `window.event`.

Какая каша! Даже при использовании модели событий DOM уровня 0, довольно независимой от типа броузера, мы сталкиваемся с массой особенностей, зависящих от типа броузера, на каждой стадии обработки событий. А при использовании более современной модели DOM уровня 2 или модели Internet Explorer учитывать различия между броузерами приходится уже на этапе установки обработчика события.

Итак, библиотека jQuery предлагает упростить нашу жизнь, скрыв различие между броузерами, насколько это возможно. Давайте посмотрим, как!

4.2. Модель событий jQuery

При создании полнофункциональных интернет-приложений, действительно, приходится во многом полагаться на обработку событий, однако мысль о разработке больших объемов программного кода, учитывающего различия между броузерами, способна оттолкнуть даже самых бесстрашных авторов страниц.

Мы могли бы скрыть имеющиеся различия за прикладным интерфейсом, абстрагирующим эти различия для программного кода наших страниц, но к чему нам эта морока, если все необходимое уже реализовано в библиотеке jQuery?

Реализация событий в jQuery, которую мы неофициально называем моделью событий jQuery, обладает следующими особенностями:

- поддерживает единый метод установки обработчиков событий;
- позволяет устанавливать в каждом элементе несколько обработчиков одного и того же события;
- использует стандартные названия типов событий, например: `click` или `mouseover`;
- организует доступ к экземпляру объекта `Event` в обработчике как к параметру;
- нормализует имена наиболее часто используемых свойств экземпляра `Event`;
- предоставляет единые методы отмены события и блокирования действия по умолчанию.

За исключением поддержки фазы захвата, по характеристикам модель событий jQuery очень близко напоминает модель событий DOM

уровня 2 и при этом поддерживает единый прикладной интерфейс для работы как с броузерами, отвечающими стандартам, так и с Internet Explorer. Отсутствие поддержки фазы захвата не станет большой проблемой для подавляющего большинства авторов страниц, которые никогда не использовали ее (а многие о ней и не догадывались) из-за отсутствия поддержки в IE.

Неужели все действительно просто? Давайте посмотрим!

4.2.1. Подключение обработчиков событий с помощью jQuery

В модели событий jQuery обработчики событий устанавливаются с помощью команды `bind()`. Рассмотрим простой пример:

```
$('img').bind('click',function(event){alert('Hi there!');});
```

Эта инструкция подключает встроенную функцию как обработчик события щелчка мышью к каждому элементу-изображению на странице. Полный синтаксис команды `bind()`:

Синтаксис команды bind

`bind(eventType, data, listener)`

Устанавливает функцию `listener` как обработчик события `eventType` во все элементы соответствующего набора.

Параметры

`eventType`	(строка) Название типа события, реакцию на которое реализует устанавливаемый обработчик. Это название может состоять из имени события и имени пространства имен, добавляемого в виде суффикса, который отделяется от имени события символом точки. Более подробные сведения приводятся в оставшейся части раздела.
`data`	(объект) Данные, поставляемые вызывающей программой функции-обработчику, присоединенные к экземпляру `Event` в виде свойства с именем `data`. Если данные отсутствуют, функция-обработчик может передаваться во втором параметре.
`listener`	(функция) Функция, которая устанавливается как обработчик события.

Возвращаемое значение

Обернутый набор.

Давайте посмотрим на команду `bind()` в действии. Возьмем за основу пример из листинга 4.3 и переведем его с использования модели событий DOM уровня 2 на модель событий jQuery. В результате мы получили программный код листинга 4.5, который вы найдете в файле *chapter4/jquery.events.html*.

Листинг 4.5. Установка обработчиков событий без кода, учитывающего особенности броузеров

```
<html>
  <head>
    <title>DOM Level 2 Events Example</title>
    <script type="text/javascript"
            src="../scripts/jquery-1.2.1.js">
    </script>
    <script type="text/javascript">
      $(function(){                                    ❶ Подключает три обработчика
        $('#vstar')                                      событий к элементу-изображению
          .bind('click',function(event) {
            say('Whee once!');
          })
          .bind('click',function(event) {
            say('Whee twice!');
          })
          .bind('click',function(event) {
            say('Whee three times!');
          });
      });

      function say(text) {
        $('#console').append('<div>'+text+'</div>');
      }
    </script>
  </head>

  <body>
    <img id="vstar" src="vstar.jpg"/>
    <div id="console"></div>
  </body>
</html>
```

Небольшие изменения в этом примере, затронувшие только тело обработчика события готовности документа, тем не менее существенны ❶. Сначала создается обернутый набор, содержащий целевой элемент <img>, и затем к нему применяются три команды bind(). Не забывайте, что способность jQuery создавать цепочки команд позволяет нам объединять в одной инструкции сразу несколько команд, каждая из которых устанавливает в элемент обработчик щелчка мышью.

Откройте страницу в броузере, соответствующем стандартам, и щелкните на изображении. Результат показан на рис. 4.6 и не содержит ничего удивительного, так как полностью (кроме URL в заголовке окна броузера) совпадает с результатом, показанным на рис. 4.3.

Самое важное в этом примере, пожалуй, то, что он работает и в Internet Explorer (рис. 4.7). В примере из листинга 4.3 было бы невозможно этого добиться, не добавив большой объем кода, проверяющего и учитывающего особенности текущего броузера.

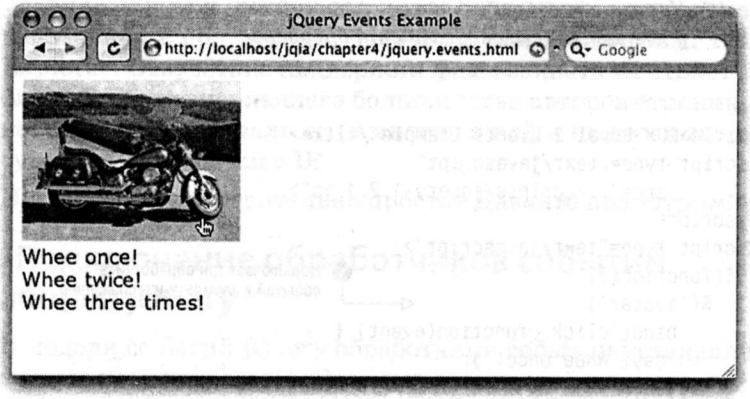


Рис. 4.6. *Модель событий jQuery позволяет определять несколько обработчиков событий, как в модели событий DOM уровня 2*

В этом месте авторы страниц, поседевшие в боях с горами кода для обработки различий между броузерами, без сомнения, запоют: «И вновь наступили счастливые дни» и прокатятся на своих офисных креслах. Но кто осудит их за это?

Еще одна интересная дополнительная особенность, которую предоставляет jQuery, – возможность группировать обработчики событий, определяя их в пространствах имен. В отличие от привычных пространств имен (когда пространство имен указывается в виде префикса), имя пространства имен обработчиков событий добавляется к имени события в виде суффикса, отделенного точкой.

Объединив обработчики событий таким способом, впоследствии мы легко сможем воздействовать на них как на единое целое.

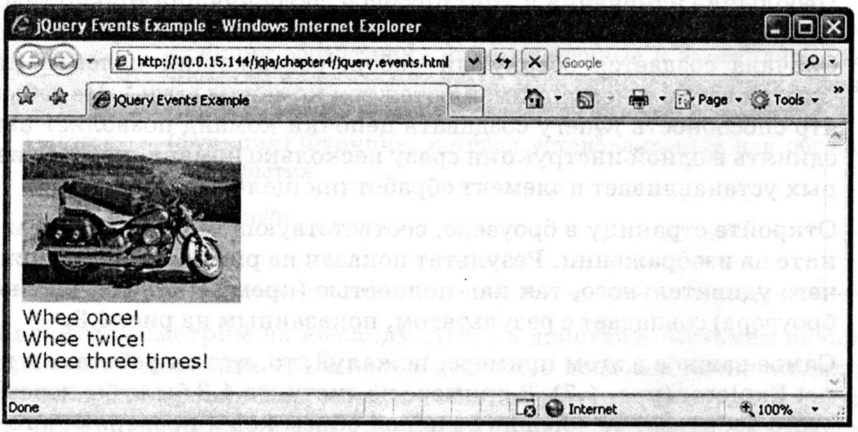


Рис. 4.7. *Модель событий jQuery предлагает универсальный код с поддержкой событий в Internet Explorer*

Возьмем в качестве примера страницу с двумя режимами работы – режимом отображения и режимом редактирования. В режиме редактирования к множеству элементов подключаются обработчики событий, неуместные в режиме отображения, и их надо удалять при выходе страницы из режима редактирования. Мы объединим обработчики событий режима редактирования в пространство имен:

```
$('#thing1').bind('click.editMode',someListener);
$('#thing2').bind('click.editMode',someOtherListener);
...
$('#thingN').bind('click.editMode',stillAnotherListener);
```

Сгруппировав все эти привязки обработчиков событий в пространство имен с именем `editMode`, позднее мы сможем оперировать с ними как с единым целым. Например, когда страница выходит из режима редактирования и наступает момент удаления всех привязок, мы легко сделаем это следующей инструкцией:

```
$('*').unbind('click.editMode');
```

Она удалит все привязки к событию `click` (описание метода `unbind()` приведено в следующем разделе) в пространстве имен `editMode` для всех элементов на странице. В дополнение к команде `bind()` библиотека jQuery предоставляет команды для установки обработчиков определенных событий. Так как синтаксис всех этих команд повторяется, за исключением имени команды, мы даем одно описание для всех команд:

Синтаксис команд, устанавливающих обработчики определенных событий

eventTypeName`(listener)`

Устанавливает функцию `listener` как обработчик события, имя которого совпадает с именем метода. Поддерживаются следующие команды:

- `blur`
- `change`
- `click`
- `dblclick`
- `error`
- `focus`
- `keydown`
- `keypress`
- `keyup`
- `load`
- `mousedown`
- `mousemove`
- `mouseout`
- `mouseover`
- `mouseup`
- `resize`
- `scroll`
- `select`
- `submit`
- `unload`

Обратите внимание: при использовании этих методов мы не можем передавать значения, находящиеся в свойстве `event.data`.

Параметры

`listener` (функция) Функция, которая устанавливается как обработчик события.

Возвращаемое значение

Обернутый набор.

Кроме того, jQuery предоставляет специализированную версию команды `bind()` с именем `one()`. Эта команда устанавливает обработчик

события однократного запуска. После первого вызова обработчика он автоматически отключается. Синтаксис этой команды похож на синтаксис команды `bind()`:

Синтаксис команды one

`one(eventType, data, listener)`

Устанавливает функцию `listener` как обработчик события `eventType` во все элементы соответствующего набора. После выполнения обработчик события автоматически отключается.

Параметры

`eventType`	(строка) Название типа события, реакцию на которое реализует устанавливаемый обработчик.
`data`	(объект) Данные, поставляемые вызывающей программой функции-обработчику, присоединенные к экземпляру `Event` в виде свойства с именем `data`. Если данные отсутствуют, функция-обработчик может передаваться во втором параметре.
`listener`	(функция) Функция, которая устанавливается как обработчик события.

Возвращаемое значение

Обернутый набор.

Эти команды предоставляют нам широкий выбор способов привязки обработчиков событий к соответствующим элементам. Но после того как обработчик будет подключен, рано или поздно нам может потребоваться отключить его.

4.2.2. Удаление обработчиков событий

Обычно после того, как обработчик события установлен, он остается задействованным на протяжении всего времени жизни страницы. Некоторые типы событий могут требовать удаления обработчика при выполнении определенных условий. Представьте себе в качестве примера страницу пошаговой настройки. Как только выполнен один шаг, относящиеся к нему элементы управления должны стать доступными только для чтения.

В таких случаях лучше удалить обработчик события из сценария. Мы уже видели, что команда `one()` может автоматически удалять обработчик после его первого вызова. Для более общего случая, когда обработчики событий удаляются под нашим управлением, библиотека jQuery предоставляет команду `unbind()`. Ее синтаксис показан на стр. 126.

С помощью этой команды можно удалять обработчики событий из элементов соответствующего набора по различным критериям. Опустив входные параметры, можно удалить все обработчики. Указав только тип события, можно удалить все обработчики событий указанного типа.

Синтаксис команды unbind

`unbind(eventType,listener)`

`unbind(event)`

Удаляет обработчики событий изо всех элементов обернутого набора согласно необязательным параметрам. Если параметры опущены, из элементов удаляются все обработчики.

Параметры

`eventType`	(строка) Если присутствует, указывает, что следует удалить только те обработчики, которые были установлены для обработки событий данного типа.
`listener`	(функция) Если присутствует, определяет функцию, которая должна быть удалена.
`event`	(Event) Удаляется обработчик, который вызван для обработки события, представленного этим экземпляром объекта `Event`.

Возвращаемое значение

Обернутый набор.

Можно удалять определенные обработчики, указывая ссылки на ранее установленные функции. Для этого нужно сохранить ссылку на функцию при ее установке в качестве обработчика события. Поэтому если функцию, в конечном счете, нужно будет удалить как обработчик события, ее либо следует определить как глобальную функцию (чтобы можно было получить ссылку на нее по имени), либо должна иметься возможность получить ссылку каким-то другим способом. Если использовать анонимные встроенные функций, то позже будет невозможно получить ссылки на них при вызове команды `unbind()`.

Мы увидели, что модель событий jQuery делает установку и удаление обработчиков событий достаточно простыми, устраняя необходимость учета различия между броузерами. А что она может дать нам непосредственно для создания обработчиков событий?

4.2.3. Исследование экземпляра Event

При вызове обработчика события, установленного командой `bind()`, в виде первого параметра ему передается экземпляр `Event`. При этом не надо беспокоиться о свойстве `window.event` в Internet Explorer – но как быть со свойствами экземпляра `Event`, которые в разных броузерах называются по-разному?

Даже при установке обработчиков событий средствами библиотеки jQuery экземпляр `Event`, передаваемый обработчику, является копией объекта, определяемого броузером. Это означает, что в броузерах, следующих стандартам, экземпляр `Event` будет иметь свойства, предусмотренные стандартами, а в Internet Explorer – свойства, присущие этому броузеру. Прежде чем передать обработчику события нестандартный

экземпляр, jQuery выполнит все необходимое, чтобы исправить объект и обеспечить доступ к свойствам и методам объекта в соответствии со стандартами. Благодаря этому мы можем писать программный код обработчиков событий без учета типа броузера, за исключением самых редко используемых свойств объекта Event.

В табл. 4.1 приведен перечень свойств объекта Event, к которым можно обращаться независимым от платформы способом.

Важно запомнить, что свойство keypress в случае неалфавитных символов совместимо не со всеми типами броузеров. Например, клавише с левой стрелкой соответствует код 37, получаемый в результате событий keyup и keypress. В броузере Safari для таких клавиш в случае события keypress возвращаются нестандартные результаты.

Более надежный способ получения кода символа с учетом регистра при событии keypress обеспечивает свойство which. Во время событий keyup и keydown мы можем получить только независимый от регистра код нажатой клавиши (так, при вводе любого из символов *a* или *A* возвращается код 65), но при этом имеется возможность определить регистр с помощью свойства shiftKey.

У экземпляра Event есть не только свойства, содержащие информацию о событии, но и ряд методов, позволяющих управлять распространением события. Давайте рассмотрим их.

4.2.4. Воздействие на распространение события

В дополнение к стандартизации наиболее часто используемых свойств экземпляра Event библиотека jQuery поддерживает аналогичную стандартизацию методов управления распространением события.

Метод stopPropagation() останавливает дальнейшее всплытие события в дереве DOM (если нужно, взгляните на рис. 4.4, чтобы вспомнить, как происходит распространение событий), а метод preventDefault() отменяет любое семантическое действие, подразумеваемое событием. В качестве примеров таких семантических действий можно привести переход по ссылке для элементов <a>, отправку формы и переключение состояния флажка по событию щелчка мышью.

Если нам потребуется остановить распространение события и одновременно отменить связанное с ним действие по умолчанию, мы можем вернуть значение false из функции-обработчика.

Кроме возможности устанавливать обработчики событий независимым от типа броузера способом, библиотека jQuery предоставляет набор команд, позволяющих запускать обработчики событий из сценариев. Давайте рассмотрим эти команды.

4.2.5. Запуск обработчиков событий

Обработчики событий запускаются, когда ассоциированные с ними события распространяются по иерархии дерева DOM. Но иногда требуется

Таблица 4.1. Платформонезависимые свойства экземпляра Event

Свойство	Описание
altKey	Имеет значение true, если в момент появления события была нажата клавиша Alt, в противном случае – false. На большинстве клавиатур для Маков клавиша Alt называется Option.
ctrlKey	Имеет значение true, если в момент появления события была нажата клавиша Ctrl, в противном случае – false.
data	Значение (если есть), передаваемое в виде второго параметра команды bind() при установке обработчика.
keyCode	Для событий keydown и keyup данное свойство содержит код нажатой клавиши. Обратите внимание: для алфавитных клавиш код соответствует символу в верхнем регистре независимо от того, в каком регистре был введен символ – в верхнем или в нижнем. Например, как для символа *a*, так и для символа *A* это свойство будет содержать код 65. Определить регистр введенного символа можно с помощью свойства shiftKey. В случае события keypress лучше использовать свойство which, подходящее для броузеров всех типов.
metaKey	Содержит значение true, если в момент появления события была нажата клавиша Meta, в противном случае – значение false. Клавиша Meta – это клавиша Ctrl на PC и клавиша Command на Маках.
pageX	Для событий от мыши: содержит горизонтальную координату точки события относительно начала координат страницы.
pageY	Для событий от мыши: содержит вертикальную координату точки события относительно начала координат страницы.
relativeTarget	Для некоторых событий от мыши: идентифицирует элемент, в границы которого пересек указатель мыши в момент появления события.
screenX	Для событий от мыши: определяет горизонтальную координату точки события относительно начала координат экрана.
screenY	Для событий от мыши определяет вертикальную координату точки события относительно начала координат экрана.
shiftKey	Имеет значение true, если в момент появления события была нажата клавиша Shift, в противном случае – false.
target	Идентифицирует целевой элемент события.
type	Для всех событий определяет тип события (например, click). Это свойство может пригодиться в случае, когда одна функция обрабатывает несколько событий.
which	Для событий от клавиатуры определяет числовой код клавиши, вызвавшей событие, а для событий от мыши определяет, какая кнопка была нажата (1 – левая, 2 – средняя, 3 – правая). Это свойство следует использовать вместо свойства button, совместимого не со всеми типами броузеров.

запустить обработчик программным путем. Для этого мы, конечно, могли бы объявить глобальную функцию, чтобы иметь возможность вызывать ее по имени, но, как уже говорилось, обычно удобнее объявлять обработчики как анонимные встроенные функции.

Библиотека jQuery позволяет уйти от необходимости объявления глобальных функций, определяя ряд методов, автоматически вызывающих обработчики событий программным способом. Самый универсальный из них – метод `trigger()`. Его синтаксис:

Синтаксис команды trigger

`trigger(eventType)`

Вызывает любые обработчики событий, установленные для обработки событий типа `eventType` во всех соответствующих элементах.

Параметры

`eventType`	(строка) Определяет имя типа событий для обработчика, который требуется вызвать.

Возвращаемое значение

Обернутый набор.

Обратите внимание: команда `trigger()` (как и другие команды, которые вскоре будут представлены) не приводит к появлению события и не вызывает его распространение по иерархии DOM. Так как нет достаточно надежного способа генерировать события независимым от типа броузера способом, jQuery просто вызывает обработчики как обычные функции.

Каждому обработчику передается экземпляр `Event`, в котором заполнен необходимый минимум свойств. Поскольку никакого события на самом деле нет, свойства, сообщающие такую информацию, как координаты события от мыши, не имеют значения. В свойство `target` записывается ссылка на элемент соответствующего набора, к которому подключен вызываемый обработчик.

Кроме того, поскольку нет никакого события, событие не распространяется в дереве DOM. Будут вызваны только обработчики, подключенные к соответствующим элементам, а обработчики родительских элементов вызываться не будут. Помните – эти команды обеспечивают удобный способ вызвать обработчик события, но не имитируют события.

В дополнение к команде `trigger()` jQuery предоставляет удобные команды для большинства типов событий. Все эти команды имеют аналогичный синтаксис, за исключением имени команды (см. с. 131).

В дополнение к возможности присоединять, удалять и запускать обработчики событий jQuery предлагает высокоуровневые функции, которые упрощают работу с событиями насколько это возможно.

Синтаксис команды *eventName*

`eventName()`

Вызывает любые обработчики событий, установленные для обработки события того типа, название которого совпадает с именем команды, во всех соответствующих элементах. Поддерживаются следующие команды:

- `blur`
- `click`
- `focus`
- `select`
- `submit`

Параметры

Нет

Возвращаемое значение

Обернутый набор.

4.2.6. Прочие команды для работы с событиями

Существуют такие разновидности взаимодействий, которые обычно используются в полнофункциональных интернет-приложениях и реализуются комбинированием различных режимов работы.

Библиотека jQuery для работы с событиями предоставляет несколько удобных команд, упрощающих применение взаимодействий такого типа в страницах. Давайте рассмотрим их.

Переключение обработчиков событий

Начнем с команды `toggle()`, которая устанавливает пару обработчиков события щелчка мышью, сменяющих друг друга по событию щелчка мышью. Вот синтаксис этой команды:

Синтаксис команды toggle

`toggle(listenerOdd,listenerEven)`

Для всех элементов обернутого набора устанавливает функции `listenerOdd` и `listenerEven` в качестве обработчиков события щелчка мышью, сменяющих друг друга по событию щелчка мышью.

Параметры

`listenerOdd` (функция) Функция, которая будет играть роль обработчика события щелчка мышью для всех нечетных щелчков (первый, третий, пятый и т. д.).

`listenerEven` (функция) Функция, которая будет играть роль обработчика события щелчка мышью для всех четных щелчков (второй, четвертый, шестой и т. д.).

Возвращаемое значение

Обернутый набор.

Обычно с помощью этой команды переключают элемент из активного состояния в неактивное и обратно, в зависимости от количества щелчков, выполненных на этом элементе. Мы можем имитировать это поведение в предыдущих примерах, изменяя степень прозрачности элементов-изображений, чтобы визуально их состояние воспринималось как активное (полностью непрозрачный элемент) или неактивное (полупрозрачный элемент). Можно было бы реализовать с помощью этой команды более действенный пример, включая и выключая режим «только для чтения» в текстовых элементах ввода, но это не так наглядно, поэтому давайте продемонстрируем работу команды на примере с изображениями.

Взгляните на рис. 4.8, где показана страница с изображением в трех состояниях в разные моменты времени:

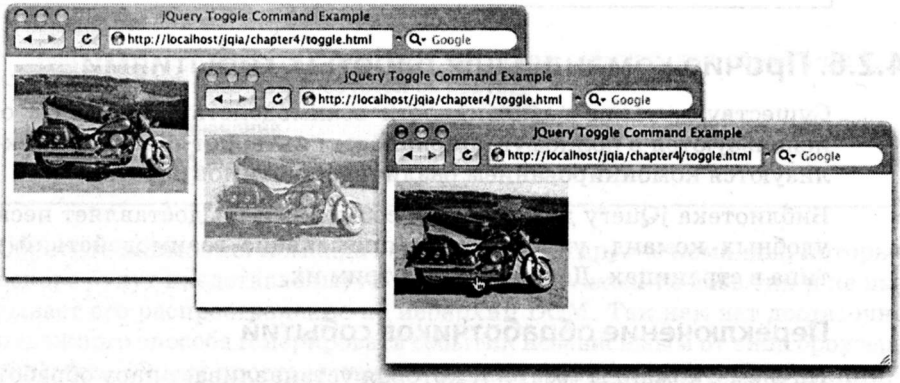


***Рис. 4.8.** Команда toggle( ) упрощает переключение визуального состояния изображения*

1. Первоначальное состояние.
2. После первого щелчка мышью.
3. После второго щелчка мышью.

Код этой страницы приведен в листинге 4.6 и содержится в файле *chapter4/toggle.html*.

Листинг 4.6. Поочередный вызов пары обработчиков событий по событию щелчка мышью

```
<html>
  <head>
    <title>jQuery Toggle Command Example</title>
    <script type="text/javascript"
            src="../scripts/jquery-1.2.1.js">
    </script>
    <script type="text/javascript">
      $(function(){
```

```
        $('#vstar').toggle(
          function(event) {
            $(event.target)
              .css('opacity',0.4);
          },
          function(event) {
            $(event.target)
              .css('opacity',1.0);
          }
        );
      });
    </script>
  </head>

  <body>
    <img id="vstar" src="vstar.jpg"/>
  </body>
</html>
```

❶ Команда toggle() применяется к изображению

❷ Обработчик нечетных щелчков, который делает изображение полупрозрачным

❸ Обработчик четных щелчков, который делает изображение непрозрачным

В этом примере мы применяем команду `toggle()` ❶ к изображению и подключаем к нему обработчик нечетных щелчков мышью ❷, который уменьшает значение непрозрачности (opacity) до 0,4 (*окрашивание в серые полутона* – обычный способ обозначить неактивное состояние элемента), и обработчик четных щелчков мышью, который восстанавливает полную непрозрачность, устанавливая значение 1,0 ❸. Поскольку команда `toggle()` сама выполняет все необходимые действия по замене обработчиков, нам не нужно беспокоиться о том, чтобы определять – четный был щелчок или нечетный. Очень удобно.

Все, чего мы достигли в этом примере, это переключение степени прозрачности изображения между полностью непрозрачным и полупрозрачным состояниями, но довольно просто представить обработчики, которые могли бы выполнять переключение между любыми двумя состояниями, например между активным и неактивным.

Другой пример ситуации с несколькими событиями, часто встречающейся в полнофункциональных интернет-приложениях, – пересечение указателем мыши границ элемента.

Перемещение указателя мыши над элементами

На основе событий, возникающих при входе указателя мыши в определенную область и при выходе из нее, строятся элементы пользовательского интерфейса, которые пользователи обычно видят на наших страницах. Наиболее типичные представители таких элементов – меню, обеспечивающие навигацию в веб-приложениях.

Досадная неприятность, связанная с событиями `mouseover` и `mouseout`, нередко усложняющая создание таких элементов, заключается в том, что при перемещении указателя мыши в область дочернего элемента возникает событие `mouseout`. Рассмотрим страницу на рис. 4.9 (файл *chapter4/hover.html*).

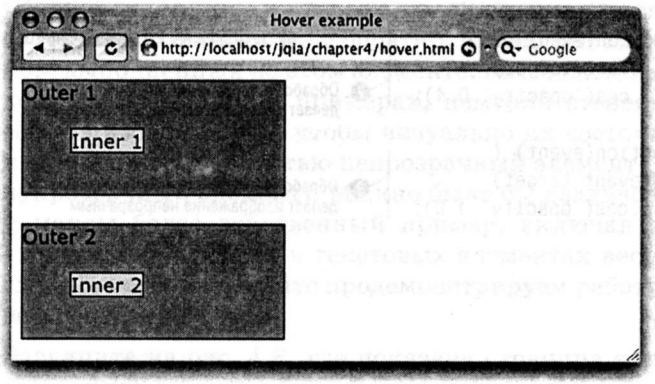


***Рис. 4.9.** Эта страница помогает понять, когда возникают события от мыши при перемещении указателя над основной и вложенной областями*

Эта страница содержит две идентичные (за исключением имен) группы областей, состоящих из внешней и внутренней области. Откройте эту страницу в броузере, так как дальнейшие пояснения в этом разделе относятся именно к ней.

Вот фрагмент сценария, устанавливающего обработчики событий от мыши в верхней группе:

```
$('#outer1')
  .bind('mouseover',report)
  .bind('mouseout',report);
```

Эта инструкция устанавливает функцию с именем `report` в качестве обработчика двух событий от мыши – `mouseover` и `mouseout`. Определение функции `report()`:

```
function report(event) {
  $('#console').append('<div>'+event.type+'</div>');
}
```

Этот обработчик просто добавляет элемент `<div>`, содержащий название события, в элемент `<div>` с именем `console`, который определяется в нижней части страницы, что позволяет нам увидеть все возникающие события.

Теперь попробуйте переместить указатель мыши внутрь области Outer 1 (внимательно следите, чтобы указатель мыши не попал в пределы области Inner 1). Вы увидите (в нижней части страницы), что было сгенерировано событие `mouseover`. Переместите указатель обратно за пределы области. Предположительно, вы должны увидеть, что было сгенерировано событие `mouseout`.

Обновите страницу, чтобы очистить консоль.

Теперь переместите указатель мыши в область Outer 1 (обратите внимание на появление события), но на этот раз продолжайте перемещать

указатель, пока он не окажется внутри области Inner 1. Как только указатель окажется в области Inner 1, для области Outer 1 будет сгенерировано событие mouseout. Если подвигать указателем над внутренней областью, можно заметить чередование событий mouseout и mouseover. Это вполне *ожидаемое* явление. Даже при том, что указатель остается в пределах области Outer 1, всякий раз, когда он будет входить в область вложенного элемента, модель событий посчитает, что при переходе из области Outer 1 в область вложенного элемента совершается выход за пределы внешней области.

Это поведение ожидаемое, но не всегда желательное. Чаще мы бы предпочли получать событие, только когда указатель мыши действительно покидает границы внешней области, не задумываясь о том, действительно ли указатель покинул область или был перемещен во вложенную область.

Мы могли бы написать свои обработчики так, чтобы они определяли, когда событие от мыши действительно происходит в результате перемещения указателя за пределы области, а когда просто произошло перемещение в область вложенного элемента, но, к счастью, этого делать не нужно. Библиотека jQuery предоставляет нам в помощь команду hover(). Вот ее синтаксис:

Синтаксис команды hover

`hover(overListener, outListener)`

Устанавливает обработчики событий mouseover и mouseout для элементов соответствующего набора. Эти обработчики вызываются только когда указатель мыши входит и выходит за пределы области, покрытой элементом, а переходы во вложенные элементы игнорируются.

Параметры

overListener	(функция) Функция, которая будет играть роль обработчика события mouseover.
outListener	(функция) Функция, которая будет играть роль обработчика события mouseout.

Возвращаемое значение

Обернутый набор.

Следующий код устанавливает обработчики событий от мыши во вторую группу областей (Outer 2 и вложенная в нее область Inner 2) на странице *hover.html*:

```
$('#outer2').hover(report, report);
```

Как и в случае с первой группой областей, функция report() устанавливается как обработчик событий mouseover и mouseout для области Outer 2. Но в отличие от первой группы областей, когда указатель мыши пересекает границу между областями Outer 2 и Inner 2, не вызывается ни один

из обработчиков. Это удобно в ситуациях, когда нам не требуется реагировать на перемещение указателя мыши в область дочернего элемента.

Теперь, имея в арсенале все эти инструменты для работы с событиями, попробуем применить полученные знания и рассмотрим страницу, в которой, кроме всего прочего, применяются некоторые другие приемы jQuery, изученные нами в предыдущих главах!

4.3. Запуск событий (и не только) в работу

Узнав, как jQuery вносит порядок в хаос моделей событий разных броузеров, давайте поработаем над созданием страницы, в которую вложим все полученные к настоящему моменту знания. В этой странице мы задействуем не только события, но и некоторые приемы jQuery из предыдущих глав, включая несколько громоздких селекторов jQuery.

Создаваемая в этом разделе страница является частью формы заказа для ресторана азиатской кухни под названием «Bamboo Asian Grille». Для краткости мы ограничим меню разделом закусок, но, чтобы попрактиковаться в использовании jQuery, вы можете применить полученные знания к оставшейся части формы меню и доделать его.

Цель этого примера достаточно проста: позволить пользователям выбрать закуски, которые они хотели бы добавить к заказу. Ничего сложного, да? Несколько флажков с текстовыми полями ввода прекрасно подойдут для создания элемента графического интерфейса, с помощью которого можно выбрать несколько блюд, указав для каждого количество порций.

Но здесь есть и сложность: для каждой из закусок необходимо предоставить возможность выбора дополнительных вариантов. Например, к «Рангунским крабам» посетитель может выбрать кисло-сладкий соус, острую горчицу или (вариант для нерешительных) обе приправы сразу. На первый взгляд, здесь нет никакой проблемы, потому что мы можем связать с каждой закуской свой набор переключателей, представляющих каждую приправу.

Но оказалось, что здесь все-таки *есть* небольшая проблема. Поколдовав немного с HTML-разметкой и стилями CSS, мы создали страницу, показанную на рис. 4.10.

Даже в случае всего пяти закусок с соответствующими приправами к ним элементов управления стало так невообразимо много, что здесь трудно увидеть выбор посетителя. Форма выполняет возложенные на нее функции, но удобство использования оставляет желать лучшего.

Мы можем решить дилемму с удобством за счет применения принципа *постепенного раскрытия*. Нам нет нужды отображать приправы для закусок, не выбранных посетителем, поэтому можно скрыть лишние переключатели.

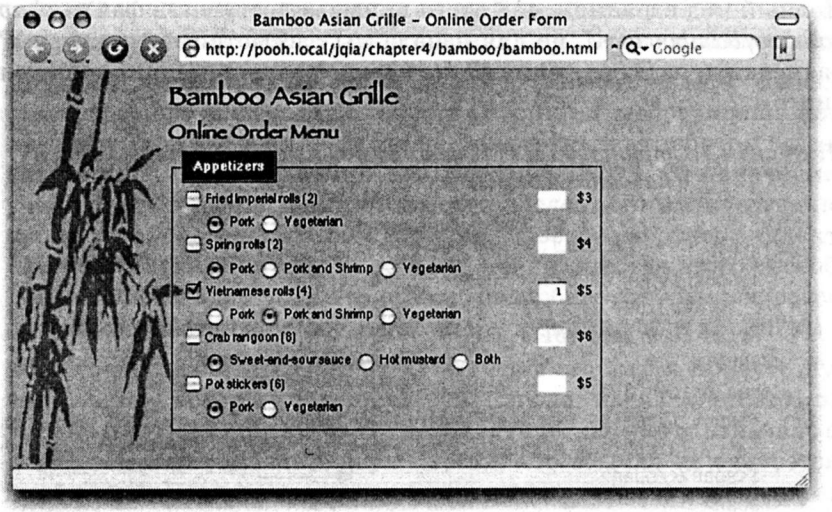


Рис. 4.10. *Все наши закуски и приправы к ним отображены на странице, но от всего этого рябит в глазах*

Применение принципа постепенного раскрытия информации, уменьшая количество элементов на странице, значительно повышает удобство пользования формой (рис. 4.11).

В качестве бонуса, когда голодный пользователь введет количество заказываемых порций, мы покажем ему соответствующую стоимость. Давайте посмотрим, что нужно сделать, чтобы реализовать все это.

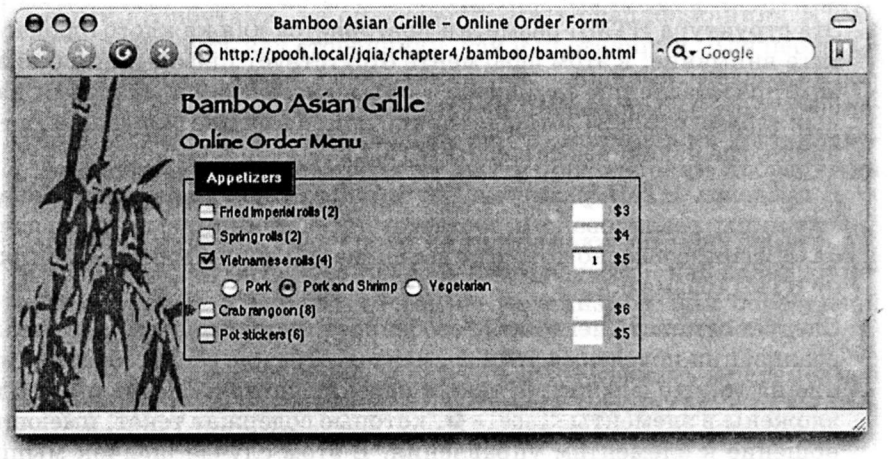


Рис. 4.11. *Скрыв ненужные варианты выбора, мы смогли уменьшить сумятицу и беспорядок – у посетителя больше не рябит в глазах от обилия элементов управления*

Полный код страницы для этого примера доступен в файле *chapter4/bamboo/bamboo.html*, но для начала мы рассмотрим структуру HTML-разметки для отображения *единственной* закуски, приведенную в листинге 4.7.

Листинг 4.7. Структура HTML-разметки для единственной записи с названием блюда

```
<div>
  <label>
    <input type="checkbox" name="appetizers"
           value="imperial"/>
    Fried Imperials rolls (2)
  </label>
  <span price="3">
    <input type="text" name="imperial.quantity"
           disabled="disabled" value="1"/>
    $<span></span>
  </span>
  <div>
    <label>
      <input type="radio" name="imperial.option"
             value="pork" checked="checked"/>
      Pork
    </label>
    <label>
      <input type="radio" name="imperial.option"
             value="vegetarian"/>
      Vegetarian
    </label>
  </div>
</div>
```

❶ Элемент метки содержит элемент управления

❷ Информация о цене хранится в нестандартном атрибуте

❸ Место для вывода вычисленной стоимости

❹ Перечень приправ, выводимый при определенных условиях

Эта структура HTML-разметки повторяется для каждого блюда. Обратите внимание: в этом фрагменте отсутствует информация о визуальном представлении элементов – она вынесена в определения каскадных таблиц стилей (которые можно найти в файле *bamboo.css*, в той же папке, где находится сам файл HTML).

В пределах HTML-разметки нет и каких-либо сценариев. Поведение страницы реализовано в соответствии с принципами ненавязчивого JavaScript, согласно которым все сценарии должны быть изолированы от HTML-разметки.

Следует отметить некоторые особенности этой структуры, важные для реализации поведения этих элементов. Прежде всего, обратите внимание на то, что флажки (а также переключатели, следующие за ними) вложены в элементы <label> ❶, которые содержат текст, имеющий отношение к элементам управления. В этом случае щелчок мышью на тексте метки изменит состояние вложенного элемента управления, как если бы пользователь щелкнул на нем. Это дополнительное удоб-

ство облегчает пользование страницей (особенно для так называемых *неаккуратных кликеров*, к которым принадлежит и один из авторов).

Другая примечательная особенность – элемент , содержащий текстовое поле для ввода количества порций и отображающий их стоимость ❷. Этот элемент снабжен нестандартным атрибутом price, предназначенным для хранения цены одного блюда. Это значение будет участвовать в вычислении стоимости введенного количества порций, кроме того, благодаря этому атрибуту удобнее работать с селекторами jQuery. (Многие считают спорными преимущества этого способа применения нестандартных атрибутов; подробнее этот вопрос рассмотрен во врезке «Нестандартные атрибуты: хорошо это или плохо?».)

Еще одна особенность состоит в том, что в структуре присутствует элемент , который содержит цену блюда; изначально этот элемент пуст ❸. Мы могли бы заполнять его статически, но это означало бы, что цена на блюдо указана в двух местах, что обычно не считается хорошим тоном. Позднее мы увидим, как заполняется эта часть страницы.

Последняя значительная особенность нашей разметки – это элемент ❹, который содержит переключатели, представляющие приправы к блюду. Это тот самый элемент, который не показывается на странице до выбора блюда.

Покончив с разметкой, давайте шаг за шагом разработаем поведение страницы, начав со скрытия контейнерного элемента, вмещающего переключатели.

Нестандартные атрибуты: хорошо это или плохо?

У подхода, при котором элементы DOM украшаются нестандартными атрибутами, не определенными в спецификациях HTML и XHTML, есть как сторонники, так и противники. Для первых – это удобный способ расширения возможностей, предоставляемых HTML-разметкой и броузерами. Для вторых – это осквернение всего самого лучшего, потому что применение нестандартных атрибутов может препятствовать прохождению проверки на правильность.

Авторы книги в этом споре не принимают ни одну из сторон, предоставляя читателю право самому решать – являются ли нестандартные атрибуты полезным механизмом или бородавкой на лице страницы.

Без такого атрибута цену можно было бы хранить в переменной JavaScript, содержащей ассоциативный массив, где с названиями блюд (например, imperial) сопоставлены их цены.

Повторимся: мы оставляем вам право решить, какой подход лучше удовлетворяет ваши потребности.

Исследование структуры HTML для одного блюда позволяет нам придумать селектор, которому будут соответствовать элементы , и применять к ним команду hide():

```
$('fieldset div div').hide();
```

Примечание

С помощью CSS мы могли бы изначально объявить эти элементы невидимыми, но выполнение этой операции в сценарии гарантирует, что пользователи, у которых поддержка JavaScript отключена (да, некоторые отключают эту поддержку), получат работоспособный интерфейс, хотя и ценой потери некоторых функциональных возможностей. Есть и другие причины скрывать элементы с помощью обработчика события готовности документа; мы рассмотрим их в главе 5, где будем подробно исследовать методы-обертки, такие как hide().

Скрыв варианты выбора приправ к блюдам, чтобы отобразить их позже, мы можем сконцентрироваться на реализации поведения, желаемого для элементов. Для начала свяжем появление переключателей с фактом выбора блюда.

Чтобы реагировать на изменение состояния флажка блюда, в соответствии с которым должен изменяться режим видимости элемента , содержащего варианты выбора приправы, мы устанавливаем обработчик события щелчка мышью в элементы-флажки, где мы сможем управлять видимостью вариантов выбора на основе состояния флажка. Рассмотрим инструкцию, которая устанавливает обработчик:

```
$(':checkbox').click(function(){
  var checked = this.checked;
  $('div',$(this).parents('div:first'))
    .css('display',checked ? 'block':'none');
  $('input[type=text]',$(this).parents('div:first'))
    .attr('disabled',!checked)
    .css('color',checked ? 'black' : '#f0f0f0')
    .val(1)
    .change()
    .each(function(){ if (checked) this.focus();});
});
```

И все это только для того, чтобы скрывать и показывать элемент ?

Не совсем так. Скрытие/отображение – лишь одно из действий, которые нужно выполнить при изменении состояния одного из флажков. Давайте рассмотрим каждый шаг этого фрагмента, чтобы разобраться в том, что при этом происходит.

Прежде всего, обработчик события сохраняет состояние флажка в переменной checked. Это упрощает програмный доступ к информации, кроме того, созданную локальную переменную мы можем использовать в любых замыканиях, которые создадим.

Затем обработчик отыскивает элемент , содержащий варианты приправ к блюду, и устанавливает CSS-свойство display этого элемента, чтобы скрыть варианты выбора, если флажок не отмечен, и показать их, если он отмечен. Выражение jQuery, применяемое для поиска элемента, видимостью которого мы управляем, имеет вид: $('div',$(this).parents('div:first')), что равносильно выражению: «отыскать элементы в первом родительском элементе this, каковым является ». Из имеющейся HTML-структуры мы знаем, что будет найдено единственное совпадение, поэтому дальнейшее деление нам не требуется.

Наиболее проницательные из вас могут заметить – раз мы знаем, что изначально флажок не отмечен и варианты выбора скрыты, мы могли бы уменьшить объем программного кода, применив команду toggle() без необходимости получать текущее состояние флажка. Программный код, основанный на таких допущениях, не обладает достаточной жесткостью и при изменении допущений легко может стать неработоспособным, поэтому всегда лучше явно определять видимость вариантов выбора в соответствии с состоянием флажка.

Наш обработчик пока не закончен – он еще должен привести в соответствие текстовые поля ввода. Эти поля ввода изначально недоступны для ввода и будут активизированы только в случае выбора флажка с названием блюда. Поиск текстового поля ввода производится с помощью селектора $('input[type=text]',$(this).parents('div:first')), похожего на тот, что мы только что использовали. Этот селектор говорит: «Отыскать элемент <input> текстового типа в моем первом родительском элементе ».

Над этим элементом выполняются следующие операции:

- С помощью команды attr() устанавливается свойство атрибута disabled в значение, противоположное состоянию флажка.
- Применяется значение CSS-свойства color, чтобы текст в поле ввода был невидимым, когда элемент находится в неактивном состоянии. (Примечательно, что этот прием срабатывает не во всех броузерах – некоторые броузеры, такие как Opera и Internet Explorer, не позволяют изменять цвет текста в неактивных полях ввода.)
- Значение элемента устанавливается равным 1. Когда элемент ввода активизируется – это то самое значение, которое должно использоваться по умолчанию; когда элемент ввода деактивизируется, следует вернуться к этому значению по умолчанию.
- Вызывается обработчик события изменения текстового поля ввода (который мы еще не определили, но не волнуйтесь, потому что вскоре мы рассмотрим его). Этот обработчик вычислит стоимость блюда и отобразит ее. Этот обработчик необходимо вызвать по той причине, что мы изменили значение (на 1), и нужно вычислить стоимость блюда.

- С помощью метода each() получается ссылка на элемент и фокус ввода переносится в него, если флажок находится в отмеченном состоянии. Вы все еще не любите замыкания, позволяющие обращаться к локальной переменной checked?

Примечание

Размышляя о том, какой тип события выбрать для обработки изменения состояния флажков, вы, возможно, первым делом подумали о перехвате события изменения, а не о событии щелчка. Чтобы наша схема работала, нужна немедленная реакция на изменение состояния флажка. Однако событие изменения состояния немедленно возникает лишь в Safari и в броузерах на базе Mozilla, а в Internet Explorer оно возникает только *после* того, как элемент потеряет фокус ввода, что делает событие изменения состояния неподходящим для нашего случая, поэтому мы выбрали событие щелчка мышью.

Теперь обратим внимание на обработчик изменения текстового поля ввода. После изменения значения текстового поля мы должны пересчитать и отобразить общую стоимость заказа. Стоимость вычисляется путем умножения цены блюда на количество порций.

Вот инструкция, которая устанавливает обработчик изменения текста в поле ввода:

```
$('span[price] input[type=text]').change(function(){
  $('~ span:first',this).text(
    $(this).val() *
    $(this).parents("span[price]:first").attr('price')
  );
});
```

После того как будут найдены текстовые поля ввода (с помощью селектора, который читается так: «найти все элементы <input> типа text, расположенные внутри элементов , имеющих атрибут price»), мы устанавливаем обработчик события изменения. Он отыскивает элемент , который следует обновить, и записывает в него вычисленное значение в текстовом виде; выражение $('~ span:first',this) отыскивает первый, соседний для this, элемент, то есть элемент . В ходе вычисления из текстового поля извлекается его значение и затем оно умножается на значение атрибута price родительского элемента .

Если какой-либо из этих селекторов вам непонятен, возможно, стоит освежить в памяти синтаксис селекторов, о котором рассказывалось в главе 2.

Прежде чем позволить пользователю взаимодействовать со страницей, нужно сделать еще кое-что. Помните, у нас еще оставались пустые элементы , которые должны содержать вычисленную стоимость блюда? Пора заполнить их.

Предварительно значения текстовых полей ввода были установлены равными 1, поэтому все, что от нас требуется, это те же вычисления,

которые выполняются при изменении значения. Для этого нам не нужно дублировать программный код, достаточно просто запустить обработчик события изменения в текстовых полях и позволить ему выполнить необходимые вычисления.

```
$('span[price] input[type=text]').change();
```

Реализация формы заказа закусок, сделанная на скорую руку, практически готова, по крайней мере, она соответствует изначально обозначенным целям. Этот пример преподал нам некоторые очень важные уроки.

- Пример показывает, как устанавливать обработчики событий щелчка мышью и изменения в элементах, позволяющие изменять пользовательский интерфейс тогда, когда нам это понадобится.
- Мы увидели, как можно запускать обработчики событий под управлением сценария, чтобы избежать дублирования программного кода и необходимости выносить программный код обработчиков в глобальные функции.
- Мы увидели несколько сложных селекторов, используемых для выбора элементов, над которыми нужно выполнить некоторые операции.

Полный исходный текст страницы приведен в листинге 4.8.

Листинг 4.8. Полный исходный код формы заказа закусок

```
<html>
  <head>
    <title>Bamboo Asian Grille - Online Order Form</title>
    <link rel="stylesheet" type="text/css" href="bamboo.css">
    <script type="text/javascript"
            src="../../scripts/jquery-1.2.1.js"></script>
    <script type="text/javascript">
      $(function(){
        $('fieldset div div').hide();
        $(':checkbox').click(function(){
          var checked = this.checked;
          $('div',$(this).parents('div:first'))
            .css('display',checked ? 'block':'none');
          $('input[type=text]',$(this).parents('div:first'))
            .attr('disabled',!checked)
            .css('color',checked ? 'black' : '#f0f0f0')
            .val(1)
            .change()
            .each(function(){ if (checked) this.focus();});
        });
        $('span[price] input[type=text]').change(function(){
          $('~ span:first',this).text(
            $(this).val() *
            $(this).parents("span[price]:first").attr('price')
          );
        });
```

```
      $('span[price] input[type=text]').change();
    });
  </script>
</head>

<body>
  <h1>Bamboo Asian Grille</h1>
  <h2>Online Order Menu</h2>
  <fieldset>
    <legend>Appetizers</legend>

    <div>
      <label>
        <input type="checkbox" name="appetizers"
               value="imperial"/>
        Fried Imperials rolls (2)
      </label>
      <span price="3">
        <input type="text" name="imperial.quantity"
               disabled="disabled" value="1"/>
        $<span></span>
      </span>
      <div>
        <label>
          <input type="radio" name="imperial.option"
                 value="pork" checked="checked"/>
          Pork
        </label>
        <label>
          <input type="radio" name="imperial.option"
                 value="vegetarian"/>
          Vegetarian
        </label>
      </div>
    </div>

    <div>
      <label>
        <input type="checkbox" name="appetizers" value="spring"/>
        Spring rolls (2)
      </label>
      <span price="4">
        <input type="text" name="spring.quantity"
               disabled="disabled" value="1"/>
        $<span></span>
      </span>
      <div>
        <label>
          <input type="radio" name="spring.option" value="pork"
                 checked="checked"/>
          Pork
        </label>
        <label>
```

```
      <input type="radio" name="spring.option"
             value="shrimp"/>
      Pork and Shrimp
    </label>
    <label>
      <input type="radio" name="spring.option"
             value="vegetarian"/>
      Vegetarian
    </label>
  </div>
</div>

<div>
  <label>
    <input type="checkbox" name="appetizers" value="vnrolls"/>
    Vietnamese rolls (4)
  </label>
  <span price="5">
    <input type="text" name="vnrolls.quantity"
           disabled="disabled" value="1"/>
    $<span></span>
  </span>
  <div>
    <label>
      <input type="radio" name="vnrolls.option" value="pork"
             checked="checked"/>
      Pork
    </label>
    <label>
      <input type="radio" name="vnrolls.option"
             value="shrimp"/>
      Pork and Shrimp
    </label>
    <input type="radio" name="vnrolls.option"
           value="vegetarian"/>
    <label>Vegetarian</label>
  </div>
</div>

<div>
  <label>
    <input type="checkbox" name="appetizers" value="rangoon"/>
    Crab rangoon (8)
  </label>
  <span price="6">
    <input type="text" name="rangoon.quantity"
           disabled="disabled" value="1"/>
    $<span></span>
  </span>
  <div>
    <label>
      <input type="radio" name="rangoon.option"
```

```
                value="sweet checked="checked"/>
              Sweet-and-sour sauce
            </label>
            <label>
              <input type="radio" name="rangoon.option" value="hot"/>
              Hot mustard
            </label>
            <label>
              <input type="radio" name="rangoon.option" value="both"/>
              Both
            </label>
          </div>
        </div>

        <div>
          <label>
            <input type="checkbox" name="appetizers"
                   value="stickers"/>
            Pot stickers (6)
          </label>
          <span price="5">
            <input type="text" name="stickers.quantity"
                   disabled="disabled" value="1"/>
            $<span></span>
          </span>
          <div>
            <label>
              <input type="radio" name="stickers.option"
                     value="pork" checked="checked"/>
              Pork
            </label>
            <label>
              <input type="radio" name="stickers.option"
                     value="vegetarian"/>
              Vegetarian
            </label>
          </div>
        </div>

        <div></div>

      </fieldset>
    </body>
  </html>
```

Этот код не зависит от числа закусок. Можно заметить, что программный код JavaScript нигде не требует, чтобы ему указывали, какие элементы соответствуют записям с закусками. Возможности селекторов jQuery позволяют нам отыскивать такие элементы автоматически. В разметку страницы можно добавлять новые записи с закусками, соблюдая предопределенный формат, а программный код будет автоматически обрабатывать их, как и уже имеющиеся записи.

Этот программный код можно несколько усовершенствовать. Чтобы уменьшить размер примера и сконцентрировать внимание на уроках, мы допустили некоторые сокращения, которые следует восполнить, прежде чем код пойдет в эксплуатацию. Ниже приведен список некоторых улучшений (или даже исправлений явных недостатков кода), которые мы предлагаем вам исследовать в качестве упражнения.

- Программный код написан в расчете на то, что пользователи будут вводить в поля ввода только допустимые числовые значения. Какая наивность! Добавьте проверку, гарантирующую ввод только допустимых числовых значений. Что вы стали бы делать в случае ввода неверного значения?
- Даже когда блюдо не выбрано и варианты приправ скрыты, они попрежнему остаются в активном состоянии и будут отправляться в составе формы заказа вместе с прочими, видимыми элементами. Это дополнительный трафик и лишний объем данных для программного кода на стороне сервера, которому придется их обработать. Как бы вы реализовали перевод переключателей из активного состояния в неактивное и обратно в соответствующие моменты времени?
- Форма в примере еще не завершена. В действительности, без элемента `<form>` это вообще не форма! Дополните HTML-разметку, чтобы форма приобрела законченный вид и могла быть отправлена на сервер.
- Человек не живет одними только закусками! Как бы вы реализовали новые разделы меню, например напитки и десерт? «Фламбированные бананы» – звучит заманчиво! Как появление этих разделов повлияет на программный код JavaScript?
- В ходе составления заказа посетителем по мере включения (и исключения) блюд можно подсчитывать общую стоимость заказа. Как бы вы реализовали вычисление общей стоимости?
- Если вы предпочитаете не использовать нестандартные атрибуты, преобразуйте страницу так, чтобы их не было. Но при этом информация о цене должна храниться в одном и только одном месте!
- Возможно, самый крупный недостаток этого примера состоит в том, что он в значительной степени зависит от взаимоотношений между элементами внутри записи с информацией о закуске. Это позволило упростить разметку, но создало тесную зависимость между структурой записи и программным кодом поддержки, а также усложнило используемые селекторы jQuery. Что бы вы предприняли, чтобы сделать программный код более устойчивым к изменениям в структуре записи? Добавление имен классов CSS в теги элементов (вместо использования информации о взаимоотношениях) – отличный способ достичь этого. Как бы вы использовали такой вариант? Нет ли у вас какой-нибудь другой идеи?

Если вас посетят идеи, которыми вы будете гордиться, обязательно посетите веб-страницу этой книги на сайте издательства Manning по адресу *http://www.manning.com/bibeault*, где вы найдете ссылку на дискуссионный форум. Вы можете представить свое решение для всеобщего обозрения и обсуждения!

4.4. Итоги

Опираясь на знания jQuery, полученные из предыдущих глав, эта глава ввела нас в мир обработки событий.

Мы познакомились с досадными проблемами, препятствующими реализации обработки событий в веб-страницах, но на этой обработке строятся полнофункциональные интернет-приложения. Из этих проблем особо выделяется факт существования трех разных моделей событий, каждая из которых реализована в наиболее популярных современных броузерах по-своему.

Устаревшая базовая модель событий, также называемая моделью событий DOM уровня 0, не так уж сильно зависит от типа броузера, в смысле объявления обработчиков событий, но зато в ее реализации функций-обработчиков следует учитывать имеющиеся различия между броузерами. Эта модель событий, вероятно, знакома авторам страниц больше других: в ней связывание обработчиков событий с элементами DOM реализовано в виде операции присваивания функций-обработчиков свойствам элементов; например, свойству `onclick`.

Несмотря на ее простоту, у этой модели есть недостаток, который выражается в том, что для каждого типа события в каждом конкретном элементе DOM можно определить только один обработчик.

Этой проблемы позволяет избежать модель событий DOM уровня 2, более совершенная и стандартизованная модель, в которой связывание обработчиков событий с типами событий и с элементами DOM производится с помощью методов прикладного программного интерфейса. Несмотря на свою универсальность, эта модель поддерживается только броузерами, соответствующими стандартам, такими как Firefox, Safari, Camino и Opera.

В Internet Explorer 6 и 7 реализована собственная модель событий на основе методов прикладного программного интерфейса, которая предоставляет подмножество функциональных возможностей модели событий DOM уровня 2.

Создание обработчиков событий сопряжено с необходимостью использования серии условных операторов `if` – один для стандартных броузеров и один для Internet Explorer – а это верный способ сойти с ума. К счастью, здесь нам на помощь приходит библиотека jQuery, которая спасает нас от столь печальной участи.

Библиотека jQuery предоставляет универсальную команду `bind()` для установки обработчиков событий любого типа в любой элемент, а также удобные команды для установки обработчиков событий определенного типа, такие как `change()` и `click()`. Эти методы работают независимым от типа броузера способом и нормализуют экземпляр `Event`, который передается обработчикам, придавая ему стандартные свойства и методы, наиболее часто используемые в обработчиках событий.

Кроме того, jQuery предоставляет возможность удалять обработчики событий, вызывать их под управлением сценария и даже определяет некоторые высокоуровневые команды, которые реализуют наиболее типичные задачи обработки событий, делая их простыми, насколько это возможно.

Мы исследовали несколько примеров использования событий в наших страницах. В следующей главе мы увидим, как на основе этих возможностей jQuery реализует анимационные эффекты.

5

В этой главе:

- Скрытие и отображение элементов без анимации
- Скрытие и отображение элементов с применением базовых анимационных эффектов jQuery
- Прочие встроенные эффекты отображения
- Создание собственной анимации

Наводим лоск: анимация и эффекты

Приходилось ли вам когда-нибудь разглядывать веб-сайты с Flash-эффектами, завидуя арсеналу их разработчиков? Черт побери, вы даже могли подумывать о том, не изучить ли Flash ради этих эффектов.

Еще совсем недавно JavaScript не позволял гладко воспроизводить эффекты и анимацию. Из-за проблем, связанных с различиями между броузерами и невысокой скоростью выполнения программного кода JavaScript, было чрезвычайно трудно плавно скрыть элемент, изменить его размеры или даже переместить из одной точки в другую. К счастью, те времена миновали, и для создания подобных эффектов jQuery предоставляет необычайно простой интерфейс.

Но прежде чем кинуться добавлять «бантики и рюшечки» к нашим страницам, мы должны ответить на вопрос: *а нужно ли нам это*? Как голливудский блокбастер, целиком состоящий из спецэффектов, не содержит никакой интриги, переполненная эффектами страница может вызвать у пользователя совсем не ту реакцию, на которую мы рассчитывали. Будьте внимательны при выборе визуальных эффектов: они должны *способствовать* пользованию страницей, а не препятствовать ему.

А теперь, помня об этом предостережении, давайте посмотрим, что может предложить jQuery.

5.1. Скрытие и отображение элементов

Пожалуй, наиболее полезный тип динамического эффекта, который нам может понадобиться применить к элементу или группе элементов,– это простое скрытие или отображение. Доступны и более причуд-

ливые эффекты (вроде плавного исчезновения или появления элемента), но иногда хочется, чтобы элемент просто появлялся или исчезал!

Названия команд, реализующих появление или исчезновение элементов, как и следовало ожидать, соответствуют выполняемым ими действиям: команда `show()` делает видимыми элементы в обернутом наборе, а команда `hide()` скрывает их. Отложим пока изучение формального синтаксиса этих команд (вскоре станет понятно, почему) и сконцентрируемся на их применении без параметров.

Несмотря на всю кажущуюся простоту этих методов, кое-что следует помнить. Во-первых, jQuery скрывает элементы, изменяя значение `display` свойства `style` на `none`. Если какой-либо элемент в обернутом наборе уже невидим, он останется невидимым, но, присутствуя в наборе, будет доступен последующим командам в цепочке. Предположим, имеется следующий фрагмент HTML:

```
<div style="display:none;">This will start hidden</div>
<div>This will start shown</div>
```

Применив к нему инструкцию `$("div").hide().addClass("fun")`, мы получим такой фрагмент:

```
<div style="display:none;" class="fun">This will start hidden</div>
<div style="display:none;" class="fun">This will start shown</div>
```

Во-вторых, если элемент изначально невидим и для его свойства стиля `display` явно задано значение `none`, то команда `show()` всегда будет устанавливать это свойство в значение `block`. Даже для элементов, у которых в видимом состоянии это свойство обычно имеет значение `inline`, например, для элементов `<span>`. Если изначально значение свойства `display` элемента не было задано явно, и мы скрыли элемент с помощью команды `hide()`, то команда `show()` вспомнит первоначальное значение свойства `display` и восстановит его.

Поэтому лучше не скрывать элементы предварительно, с помощью атрибутов `style`, а применять к ним команду `hide()` из обработчика события готовности документа. Это обеспечит их невидимость на стороне клиента с гарантией известного начального состояния и предсказуемого поведения при последующих вызовах команд `hide()` и `show()`.

Давайте посмотрим, как лучше использовать эти команды.

5.1.1. Реализация сворачиваемого списка

Перегруженность пользовательского интерфейса информацией – классический недостаток. Лучше позволить пользователям самим запрашивать информацию по частям и управлять ими. Это слабый отголосок емкого принципа, называемого *постепенным раскрытием* (см. предыдущую главу), согласно которому следует отображать для пользователя минимум данных, раскрывая их по мере необходимости.

Прекрасным примером реализации этого принципа может служить операция просмотра содержимого файловой системы компьютера. Эта информация часто выводится в виде иерархического списка, где содержимое папок скрыто на более глубоких уровнях вложенности, необходимых для представления всех файлов и папок в системе. Было бы нелепо пытаться вывести полный список файлов и папок сразу! Гораздо лучше реализовать отображение в виде иерархического списка, который предусматривает возможность сворачивания и разворачивания на каждом уровне вложенности. Разумеется, элементы управления, позволяющие просматривать содержимое файловой системы, встречались вам во многих приложениях.

В этом разделе мы увидим, как с помощью команд `hide()` и `show()` реализовать иерархический список, действующий подобным образом.

Примечание

Несколько интересных модулей расширения содержат элементы управления этого типа. Поняв принцип их действия, вы сможете применять готовые решения вместо создания собственного.

Для начала рассмотрим HTML-структуру списка, позволяющего тестировать наш программный код.

```
<body>
  <fieldset>
    <legend>Collapsible List &mdash; Take 1</legend>
    <ul>
      <li>Item 1</li>
      <li>Item 2</li>
      <li>
        Item 3
        <ul>
          <li>Item 3.1</li>
          <li>
            Item 3.2
            <ul>
              <li>Item 3.2.1</li>
              <li>Item 3.2.2</li>
              <li>Item 3.2.3</li>
            </ul>
          </li>
          <li>Item 3.3</li>
        </ul>
      </li>
      <li>
        Item 4
        <ul>
          <li>Item 4.1</li>
          <li>
            Item 4.2
            <ul>
```

```
                <li>Item 4.2.1</li>
                <li>Item 4.2.2</li>
              </ul>
            </li>
          </ul>
        </li>
        <li>Item 5</li>
      </ul>
    </fieldset>
  </body>
```

Примечание

Поскольку в этом разделе рассматриваются визуальные эффекты, будем исходить из предположения, что список, с которым нам предстоит работать, достаточно невелик, чтобы целиком уместиться на странице. Очевидно, что это предположение будет неверным для таких больших объемов данных, как список файлов и папок в файловой системе. В таких случаях вам придется повторно обращаться к серверу за новыми порциями данных, но это уже выходит за рамки данной главы. Прочитав главу, посвященную технологии Ajax, вы сможете вернуться к этим примерам и добавить элементам управления новые возможности, применив новые знания.

При отображении в броузере (до внесения наших изменений) этот список выглядит, как показано на рис. 5.1.

Список такого размера достаточно просто просмотреть, но легко вообразить список подлиннее и с более глубокой вложенностью, чреватый ужасным синдромом информационной перегрузки для пользователей.

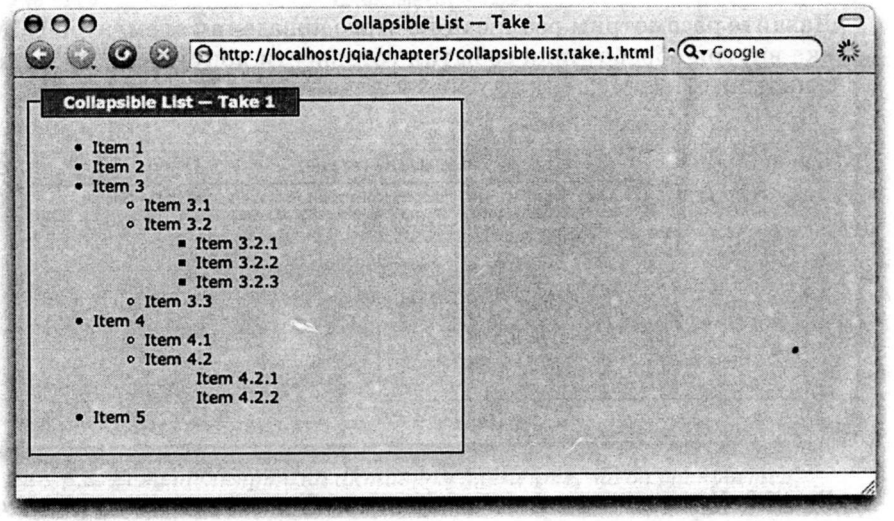


Рис. 5.1. Многоуровневый список до нашего вмешательства

Нужно так обработать все элементы списка, содержащие вложенные списки, чтобы вложенное содержимое было скрыто до тех пор, пока пользователь не щелкнет на выбранном элементе. После щелчка вложенное содержимое отображается, повторный щелчок снова скрывает содержимое.

Если просто скрыть содержимое элементов, содержащих вложенные списки (например, пункты 3 и 4 на самом верхнем уровне), пользователь может быть введен в заблуждение, так как не знает, что эти пункты активны и могут быть раскрыты, поэтому нужно визуально выделить эти пункты, чтобы они отличались от простых элементов списка (которые не могут быть раскрыты). Мы сделаем это, изменяя форму указателя мыши, когда он будет находиться над активным элементом, а также отметим графическими значками «плюс» и «минус» элементы списка, которые могут быть развернуты или свернуты.

Исходно список полностью свернут и пользователь может работать с ним. После реализации программного кода список в начальном свернутом состоянии будет выглядеть, как показано на рис. 5.2.

Здесь можно увидеть, что произошло с элементами списка, имеющими вложенное содержимое (пункты 3 и 4):

- вложенное содержимое элементов 3 и 4 стало невидимым;
- маркеры элементов списка заменены графическим значком «плюс», показывающим, что эти элементы можно раскрыть;
- указатель мыши при наведении на эти элементы списка приобретает форму «указующего перста».

Если щелкать на активных элементах, отобразятся их вложенные элементы (рис. 5.3).

Давайте рассмотрим реализацию этого поведения элементов DOM списка внутри обработчика события готовности документа, приведенную в листинге 5.1.

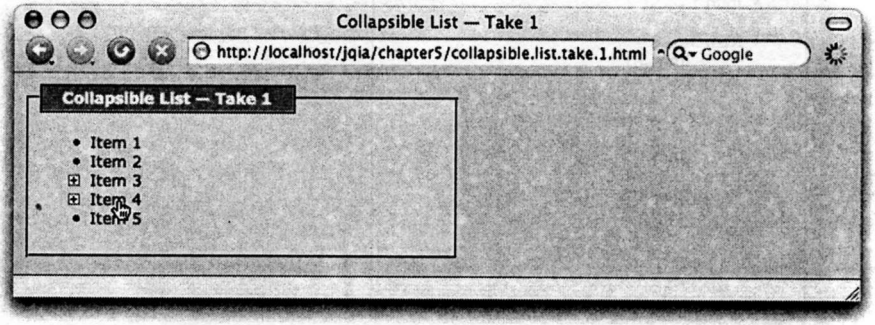


***Рис. 5.2.** После реализации программного кода список изначально свернут; элементы, допускающие раскрытие, визуально отличаются от простых элементов*

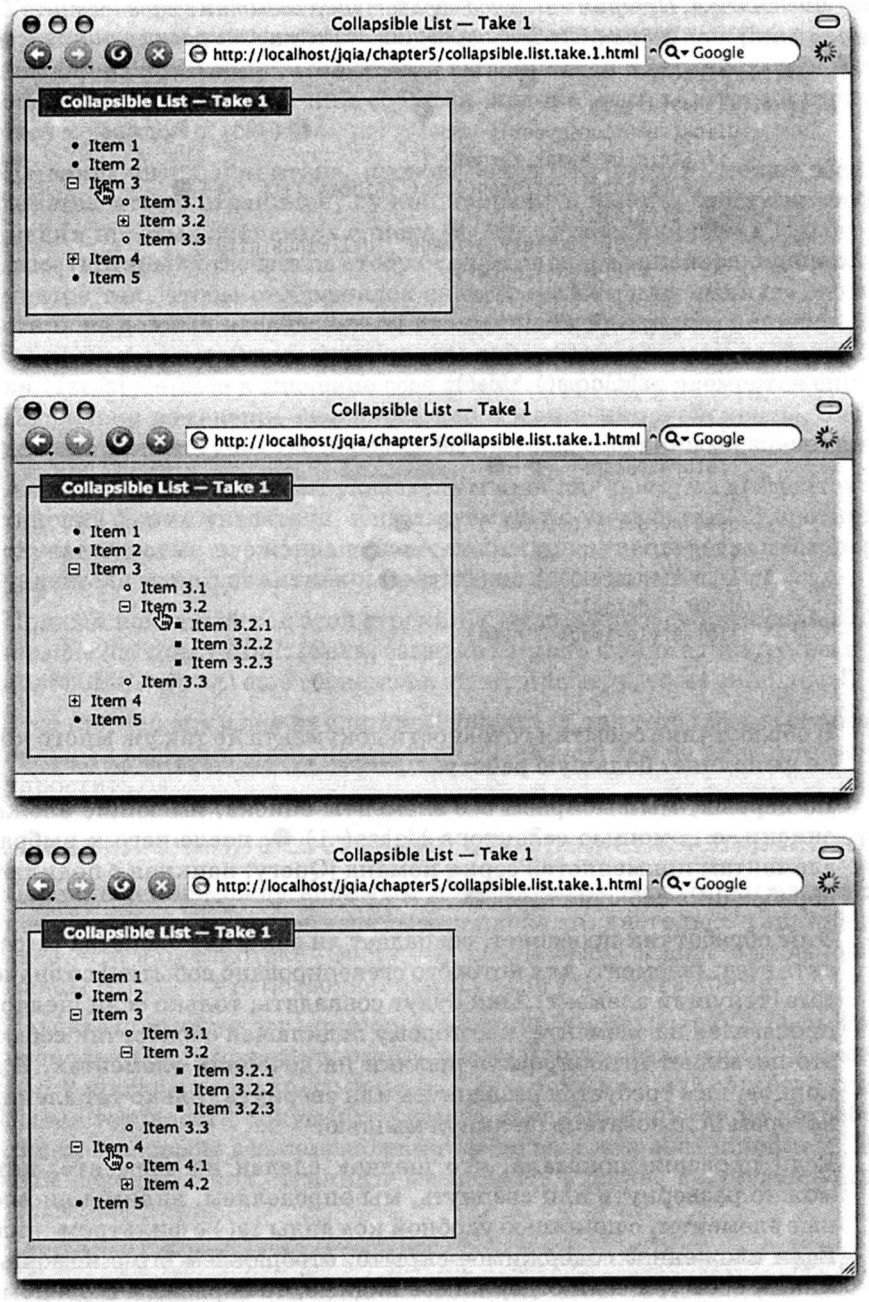


***Рис. 5.3.** Если щелкнуть на активном элементе, то он раскроется, и отобразятся его вложенные элементы*

Листинг 5.1. Программный код обработчика события готовности документа, добавляющий возможность раскрытия списка

```
$(function(){
  $('li:has(ul)')                                    ❶
    .click(function(event){                          ❷
      if (this == event.target) {
        if ($(this).children().is(':hidden')) {      ❸
          $(this)
            .css('list-style-image','url(minus.gif)')
            .children().show();
        } else {
          $(this)
            .css('list-style-image','url(plus.gif)')
            .children().hide();
        }
      }
      return false;                                  ❹
    })
    .css('cursor','pointer')                         ❺
    .click();                                        ❻
  $('li:not(:has(ul))').css({                        ❼
    cursor: 'default',
    'list-style-image':'none'
  });
});
```

В обработчике события готовности документа не так уж много кода, но он выполняет большую работу.

Во-первых, мы выбираем все элементы списка, имеющие вложенные списки, с помощью селектора `li:has(ul)` ❶, после чего к выбранным элементам применяется серия команд jQuery, начиная с подключения обработчика события `click` ❷.

Этот обработчик проверяет, совпадает ли значение свойства `target` объекта `event` (элемент, для которого сгенерировано событие) со значением `this` (текущий элемент). Они будут совпадать, только если щелчок был произведен на элементе, к которому подключен обработчик события, – это позволяет игнорировать щелчки на дочерних элементах. В конце концов, нам требуется развернуть или свернуть только тот элемент, на котором пользователь щелкнул мышью.

Если проверка показала, что щелчок сделан на элементе, который можно развернуть или свернуть, мы определяем, видимы ли вложенные элементы, с помощью удобной команды `is()` с фильтром `:hidden` ❸. Если вложенное содержимое скрыто, отображаем его с помощью команды `show()`, а если содержимое видимо, то скрываем его с помощью команды `hide()`. И в том и в другом случае мы изменяем маркер родительского элемента списка на значок «плюс» или «минус» и в качестве возвращаемого значения обработчика события используем значение `false` ❹, чтобы предотвратить бесполезное распространение события.

Установка формы указателя мыши для активного элемента списка производится с помощью команды css() ❺. Скрытие дочерних элементов для активных пунктов списка выполняется в ветке else условного оператора if, в обработчике события щелчка мыши, который вызывается командой click() ❻.

На заключительном этапе, прежде чем пользователь сможет взаимодействовать со страницей, нужно поправить некоторые стили отображения простых элементов списка ❼. Мы установили стиль list-style-image (который отвечает за отображение маркера списка) активных элементов так, чтобы отображался значок «плюс» или «минус», но этот стиль не должен наследоваться вложенными элементами списка. Чтобы предотвратить наследование, мы явно устанавливаем свойство стиля list-styleimage в значение none для всех простых элементов списка. Поскольку изменение вносилось непосредственно в элементы, оно будет иметь преимущество перед любыми унаследованными значениями.

Аналогично настраиваем указатель мыши для простых элементов, установив форму указателя, используемую по умолчанию. В противном случае простые вложенные элементы списка унаследовали бы форму указателя мыши от активного родительского элемента.

Полный исходный код этой страницы вы найдете в файле *chapter5/collapsible.list.take.1.html*. (Если, заметив в имени файла единицу, вы предположили, что мы еще вернемся к этому примеру, то не ошиблись!)

Этот пример оказался не слишком сложным для того объема функциональных возможностей, которые он реализует, тем не менее его можно упростить.

5.1.2. Переключение состояния отображения элементов

Переключение между видимым и невидимым состоянием элементов, как в примере со сворачиваемыми списками, настолько типичная задача, что в библиотеке jQuery она реализована в виде отдельной команды toggle(), что позволяет еще больше упростить пример.

Давайте применим эту команду к сворачиваемому списку и посмотрим, насколько она сможет упростить предыдущую реализацию. В листинге 5.2 приведен только измененный программный код обработчика события готовности документа (другие изменения просто не потребовались), в котором все исправления выделены жирным шрифтом. Полный исходный код страницы вы найдете в файле *chapter5/collapsible.list.take.2.html*.

Листинг 5.2. Код сворачиваемого списка с применением команды toggle( )

```
$(function(){
  $('li:has(ul)')
    .click(function(event){
      if (this == event.target) {
        $(this).children().toggle();
```

```
        $(this).css('list-style-image',
          ($(this).children().is(':hidden')) ?
            'url(plus.gif)' : 'url(minus.gif)');
      }
      return false;
    })
    .css('cursor','pointer')
    .click();
  $('li:not(:has(ul))').css({
    cursor: 'default',
    'list-style-image':'none'
  });
});
```

Обратите внимание: нам больше не требуется условный оператор для определения состояния видимости элементов – команда `toggle()` сама позаботится об изменении этого состояния. Мы по-прежнему используем команду `.is(':hidden')` как условную часть трехместного оператора, чтобы определить, каким значком будет маркер элемента списка.

Мгновенно отображать или скрывать элементы удобно, но иногда желателен более плавный переход между этими двумя состояниями элемента. Давайте посмотрим, можно ли это реализовать.

5.2. Анимационные эффекты при изменении визуального состояния элементов

Человек с трудом воспринимает быстрые изменения визуальной информации, мгновенное появление или исчезновение элементов списка может вызывать неприятные ощущения. Случайно моргнув, мы можем не заметить момент перехода элементов из одного состояния в другое и не понять – что, собственно, произошло.

Быстрый, но постепенный переход от одного состояния к другому поможет нам осознать, *что* и *как* изменилось. Это как раз тот случай, когда на помощь приходят базовые эффекты библиотеки jQuery, которые делятся на три категории:

- постепенное отображение и скрытие (команды `show()` и `hide()` могут немножко больше, чем было показано в разд. 5.1);
- плавное проявление и растворение;
- закатывание и выкатывание.

Давайте познакомимся поближе с этими категориями эффектов.

5.2.1. Постепенное отображение и скрытие элементов

Команды `show()`, `hide()` и `toggle()` значительно сложнее, чем можно было бы подумать, читая предыдущий раздел. При вызове без параметров эти команды просто изменяют визуальное состояние обернутых элементов,

заставляя их мгновенно появляться или исчезать. Однако при вызове с параметрами эти команды могут воспроизводить эффект постепенного изменения состояния на протяжении указанного периода времени.

Теперь мы готовы увидеть полный синтаксис этих команд.

Давайте сделаем третий подход к реализации сворачиваемого списка, добавив в него анимационные эффекты.

С учетом вышесказанного вы могли бы подумать, что для этого достаточно просто изменить во втором примере реализации сворачиваемого списка вызов команды `toggle()` на следующий:

```
toggle('slow')
```

Но все не так просто! Внеся это изменение и поэкспериментировав со страницей, мы заметим нечто странное. Во-первых, вспомните, что на этапе инициализации мы скрываем раскрывающиеся элементы, вызывая обработчик события `click` для активных элементов. Этот прием прекрасно подходит для случая мгновенного скрытия дочерних элементов. Но теперь мы воспроизводим анимационный эффект, и при открытии страницы наблюдаем плавное исчезновение вложенных элементов. Это совершенно лишнее!

Синтаксис команды hide

`hide(speed,callback)`

Скрывает элементы из обернутого набора. При вызове без параметров операция выполняется мгновенно, установкой свойства стиля `display` элементов в значение `none`. Если задан параметр `speed`, элементы будут постепенно исчезать в течение указанного интервала времени за счет плавного уменьшения их размеров и уровня непрозрачности до нуля. По истечении этого времени свойство стиля `display` устанавливается в значение `none`, чтобы полностью удалить элементы с экрана.

Можно указать необязательную функцию обратного вызова `callback`, которая будет вызвана по окончании воспроизведения эффекта.

Параметры

`speed` (число | строка) Необязательный параметр, определяющий продолжительность воспроизведения эффекта. Может быть числом миллисекунд или одной из предопределенных строк – *slow*, *normal* или *fast*. При отсутствии этого параметра эффект не воспроизводится и элементы будут исчезать с экрана мгновенно.

`callback` (функция) Необязательная функция, которая будет вызвана по окончании воспроизведения эффекта. Функция вызывается без параметров, однако в контексте функции (`this`) ей передается элемент, над которым выполняется операция.

Возвращаемое значение

Обернутый набор.

Синтаксис команды show

`show(speed, callback)`

Скрывает элементы из обернутого набора. При вызове без параметров операция выполняется мгновенно установкой свойства стиля `display` элементов в первоначальное значение (такое как `block` или `inline`), предшествовавшее операции скрытия, при условии, что элемент был сделан невидимым с помощью эффекта jQuery. Если элемент был скрыт не средствами jQuery, свойство стиля `display` по умолчанию будет установлено в значение `block`.

Если задан параметр `speed`, элементы будут постепенно появляться в течение указанного интервала времени за счет плавного увеличения их размеров и уровня непрозрачности.

Можно указать необязательную функцию обратного вызова `callback`, которая будет вызвана по окончании воспроизведения эффекта.

Параметры

`speed` (число | строка) Необязательный параметр, определяющий продолжительность воспроизведения эффекта. Может быть числом миллисекунд или одной из предопределенных строк – *slow*, *normal* или *fast*. При отсутствии этого параметра эффект не воспроизводится и элементы будут появляться на экране мгновенно.

`callback` (функция) Необязательная функция, которая будет вызвана по окончании воспроизведения эффекта. Функция вызывается без параметров, однако в контексте функции (`this`) ей передается элемент, над которым выполняется операция.

Возвращаемое значение

Обернутый набор.

Прежде чем пользователь получит шанс увидеть элементы, нужно скрыть их явным вызовом команды `hide()` без параметров и установить в качестве маркеров списка графические значки «плюс». В исходном примере мы не сделали это, потому что пришлось бы вводить повторяющийся программный код. А с внесенными изменениями эта проблема отпала.

Другая замеченная проблема – изображения маркеров теперь устанавливаются неправильно. При мгновенном переключении визуального состояния мы могли проверить результаты немедленно. Теперь, когда переключение происходит с анимационным эффектом, результат получается несинхронным, потому мгновенная проверка видимости дочерних элементов (позволяющая узнать, какой маркер следует отобразить) дает неверный результат.

Давайте инвертируем условие проверки, чтобы проверять состояние дочерних элементов *перед* тем, как запустить анимированное переключение.

Новый обработчик события готовности документа, изменения в котором выделены жирным шрифтом, приведен в листинге 5.3.

Синтаксис команды toggle

`toggle(speed,callback)`

Для скрытых элементов из обернутого набора выполняет команду `show()`, а для видимых – команду `hide()`. Соответствующая семантика этих команд приведена в описании их синтаксиса.

Параметры

`speed` (число | строка) Необязательный параметр, определяющий продолжительность воспроизведения эффекта. Может быть числом миллисекунд или одной из предопределенных строк – *slow*, *normal* или *fast*. При отсутствии этого параметра эффект не воспроизводится.

`callback` (функция) Необязательная функция, которая будет вызвана по окончании воспроизведения эффекта. Функция вызывается без параметров, однако в контексте функции (`this`) ей передается элемент, над которым выполняется операция.

Возвращаемое значение

Обернутый набор.

Листинг 5.3. Пример реализации сворачиваемого списка с анимационным эффектом

```
$(function(){
  $('li')
    .css('pointer','default')
    .css('list-style-image','none');
  $('li:has(ul)')
    .click(function(event){
      if (this == event.target) {
        $(this).css('list-style-image',
          (!$(this).children().is(':hidden')) ?
            'url(plus.gif)' : 'url(minus.gif)');
        $(this).children().toggle('slow');
      }
      return false;
    })
    .css({cursor:'pointer',
          'list-style-image':'url(plus.gif)'})
    .children().hide();
  $('li:not(:has(ul))').css({
    cursor: 'default',
      'list-style-image':'none'
  });
});
```

Страницу с этим изменениями вы найдете в файле *chapter5/collapsible.list.take.3.html*.

Для любителей экспериментов мы создали удобный инструмент, который далее будем применять для исследования этих команд.

Введение в лабораторную страницу jQuery Effects Lab Page

В главе 2 вы уже видели лабораторную страницу, помогавшую нам экспериментировать с селекторами jQuery. В этой главе мы представляем вам лабораторную страницу для исследования эффектов jQuery (файл *chapter5/lab.effects.html*).

Откройте этот файл в броузере. Вы должны увидеть страницу, показанную на рис. 5.4.

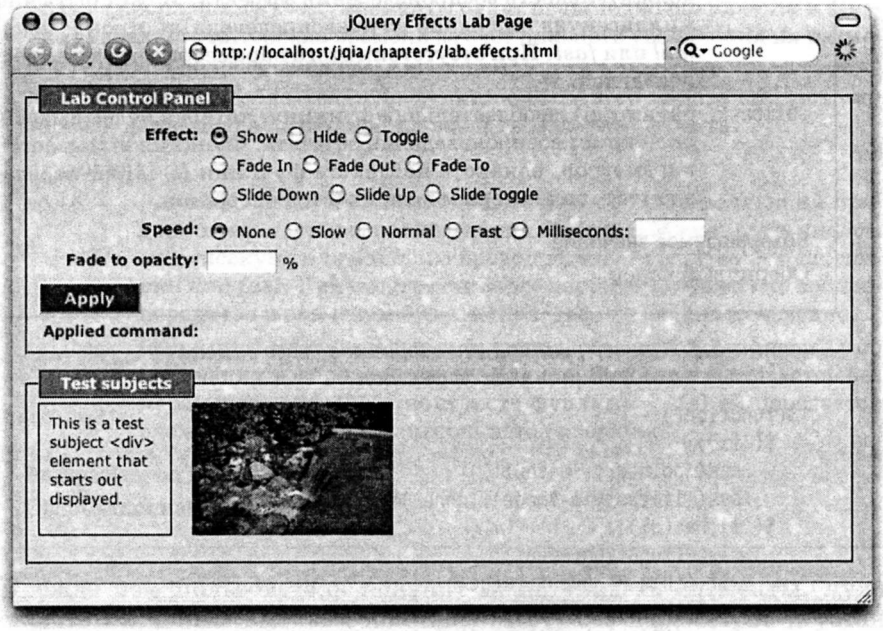


Рис. 5.4. *Начальное состояние лабораторной страницы jQuery Effects Lab Page, которая поможет нам исследовать действие команд jQuery, предназначенных для воспроизведения эффектов*

Эта лабораторная страница содержит две основные панели: Lab Control Panel (панель управления), где задаются применяемые эффекты, и Test subjects (объекты исследования) – четыре объекта экспериментов, к которым эффекты будут применяться.

«У них что-то с головой? – можете подумать вы. – Здесь только два объекта!»

Нет, авторы пока еще в своем уме. Здесь *действительно* четыре элемента, но два из них (второй элемент `<div>` с текстом и второе изображение) изначально невидимы.

Давайте с помощью этой страницы посмотрим действие команд, которые мы уже рассмотрели к этому моменту. Откройте страницу в броузере и выполните следующие упражнения:

- *Упражнение 1.* Оставив элементы управления в том же состоянии, как сразу после открытия страницы, щелкните на кнопке Apply (применить). В результате будет выполнена команда `show()` без параметров. Результат действия команды отображается ниже кнопки Apply. Обратите внимание: изначально скрытые элементы (объекты исследования) появляются немедленно. На вопрос, почему правое крайнее изображение немного блеклое, ответим, что значение непрозрачности (opacity) для него преднамеренно было установлено равным 50%.
- *Упражнение 2.* Выберите переключатель Hide (скрыть) и щелкните на кнопке Apply, чтобы выполнить команду `hide()` без параметров. В результате все объекты исследования 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 принимается в учет механизмом отображения броузера, точно так же, как если бы CSS-свойство `display` было установлено в значение `none`. Но это *не* означает, что элемент был удален из дерева DOM, – ни один из эффектов не удаляет элементы из дерева DOM.

- *Упражнение 3.* Выберите переключатель Toggle (переключение) и щелкните на кнопке Apply. Щелкните на кнопке Apply еще раз. Вы заметите, что каждое последующее выполнение команды `toggle()` переключает режим отображения объектов исследования.
- *Упражнение 4.* Перезагрузите страницу, чтобы вернуть ее в первоначальное состояние (в броузере Firefox перенесите фокус ввода в адресную строку и нажмите клавишу Enter). Выберите переключатель Toggle (переключение) и щелкните на кнопке Apply. Обратите внимание: два изначально видимых элемента исчезли, а два других, изначально скрытых, появились. Это доказывает, что команда `toggle()` применяется отдельно к каждому элементу обернутого набора, делая видимыми одни и скрывая другие.
- *Упражнение 5.* В этом упражнении переместим свое внимание в область анимационных эффектов. Обновите страницу и в группе Speed (скорость) выберите переключатель Slow (медленно). Щелкните на кнопке Apply и внимательно посмотрите, что происходит в 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, обновите страницу, выберите переключатель Milliseconds (миллисекунды) и введите в расположенном правее него поле число 10 000. Это увеличит продолжительность до 10 (мучительных) секунд и позволит вам подробно рассмотреть поведение эффекта.

- *Упражнение 6*. Выбирая переключатели Show, Hide и Toggle и задавая различные скорости, поэкспериментируйте с этими эффектами до тех пор, пока не почувствуете, что достаточно хорошо понимаете, как они действуют.

Вооружившись лабораторной страницей jQuery Effects Lab Page и пониманием действия эффектов из первого набора, перейдем к изучению следующего набора эффектов.

5.2.2. Плавное растворение и проявление элементов

Внимательно наблюдая работу команд `show()` и `hide()`, вы могли заметить, что элементы изменяются в размерах (увеличиваются либо уменьшаются), *и* одновременно, по мере увеличения или уменьшения, изменяется степень их прозрачности. Следующий набор эффектов, `fadeIn()` и `fadeOut()`, воздействует только на прозрачность элементов.

За исключением изменения размеров элементов, эти команды действуют точно так же, как команды `show()` и `hide()` соответственно. Синтаксис команд `fadeIn()` и `fadeOut()`:

Синтаксис команды fadeOut

`fadeOut(speed,callback)`

Скрывает элементы из обернутого набора, которые до этого не были скрыты, вплоть до их удаления из отображения страницы, постепенно уменьшая их непрозрачность до 0% и затем удаляет их из отображения страницы. Скорость изменения непрозрачности задается параметром `speed`. Уже скрытые элементы не подвергаются действию команды.

Параметры

`speed` (число | строка) Определяет продолжительность воспроизведения эффекта. Может быть числом миллисекунд или одной из предопределенных строк – *slow*, *normal* или *fast*. При отсутствии этого параметра эффект воспроизводится со скоростью, соответствующей значению *normal*.

`callback` (функция) Необязательная функция, которая будет вызвана по окончании воспроизведения эффекта. Функция вызывается без параметров, однако в контексте функции (`this`) ей передается элемент, над которым выполняется операция.

Возвращаемое значение

Обернутый набор.

Давайте еще немного поэкспериментируем с лабораторной страницей jQuery Effects Lab Page. Откройте страницу в броузере и, как в предыдущем разделе, выполните упражнения с переключателями Fade In (плавное проявление) и Fade Out (плавное растворение). (На переключатель Fade To (изменить прозрачность до значения) пока не смотрите, мы займемся им чуть позже.)

Синтаксис команды fadeIn

`fadeIn(speed,callback)`

Делает видимыми элементы из обернутого набора, которые до этого были скрыты, постепенно увеличивая их непрозрачность до 100%. Скорость изменения непрозрачности задается параметром `speed`. Уже видимые элементы не подвергаются действию команды.

Параметры

`speed` (число | строка) Определяет продолжительность воспроизведения эффекта. Может быть числом миллисекунд или одной из предопределенных строк – *slow*, *normal* или *fast*. При отсутствии этого параметра эффект воспроизводится со скоростью, соответствующей значению *normal*.

`callback` (функция) Необязательная функция, которая будет вызвана по окончании воспроизведения эффекта. Функция вызывается без параметров, однако в контексте функции (`this`) ей передается элемент, над которым выполняется операция.

Возвращаемое значение

Обернутый набор.

Важно отметить, что при изменении непрозрачности элемента команды `hide()`, `show()`, `fadeIn()` и `fadeOut()` запоминают первоначальное значение непрозрачности элемента. В лабораторной странице мы намеренно установили начальное значение непрозрачности правого изображения равным 50%, прежде чем скрыть его. Во всех эффектах jQuery, где изменяется степень непрозрачности, первоначальное значение никогда не теряется.

Выполните дополнительные упражнения с лабораторной страницей, пока не почувствуете, что достаточно хорошо понимаете действие эффектов проявления и растворения.

Еще один эффект jQuery предоставляет в виде команды `fadeTo()`. Этот эффект, как и два предыдущих, изменяет степень непрозрачности элемента, но никогда не удаляет элементы из отображения страницы. Прежде чем экспериментировать с командой `fadeTo()` на лабораторной странице, ознакомьтесь с ее синтаксисом.

В отличие от других эффектов, изменяющих непрозрачность до полного растворения или проявления элементов, команда `fadeTo()` не запоминает первоначальное значение непрозрачности элемента. В этом есть определенный смысл, так как цель этого эффекта – явно изменить непрозрачность до указанного значения.

Откройте лабораторную страницу и заставьте проявиться все элементы (теперь вы знаете, как это сделать). Затем выполните следующие упражнения:

Синтаксис команды fadeTo

`fadeTo(speed, opacity, callback)`

Изменяет степень непрозрачности элементов из обернутого набора от текущего значения до нового, указанного в параметре `opacity`.

Параметры

`speed` (число | строка) Определяет продолжительность воспроизведения эффекта. Может быть числом миллисекунд или одной из предопределенных строк – *slow*, *normal* или *fast*. При отсутствии этого параметра эффект воспроизводится со скоростью, соответствующей значению *normal*.

`opacity` (число) Конечное значение, до которого будет изменяться непрозрачность элемента. Задается числом от 0,0 до 1,0.

`callback` (функция) Необязательная функция, которая будет вызвана по окончании воспроизведения эффекта. Функция вызывается без параметров, однако в контексте функции (`this`) ей передается элемент, над которым выполняется операция.

Возвращаемое значение

Обернутый набор.

- *Упражнение 1* – Выберите переключатель Fade To (изменить прозрачность до значения) и задайте значение скорости достаточно большое, чтобы увидеть, как действует эффект, (4 000 миллисекунд будет вполне достаточно). Теперь введите в поле Fade to Opacity (изменить до уровня непрозрачности) число 10 – это поле предполагает ввод значения от 0 до 100 (процентов), которое для передачи в команду будет преобразовано в число от 0,0 до 1,0, – и щелкните на кнопке Apply (применить). В результате непрозрачность объектов исследования плавно изменится до 10% за 4 секунды.
- *Упражнение 2* – Введите в поле Fade to Opacity значение 100 и щелкните на кнопке Apply. Все элементы, включая изначально полупрозрачное изображение, станут полностью непрозрачными.
- *Упражнение 3* – Введите в поле Fade to Opacity значение 0 и щелкните на кнопке Apply. Все элементы растворятся до невидимого состояния, но, что самое примечательное, когда они полностью исчезнут, панель Test subjects (объекты исследования) *не* свернется. В отличие от `fadeOut()`, эффект `fadeTo()` никогда не удаляет элементы из отображения страницы, даже когда они полностью невидимы.

Продолжайте экспериментировать с эффектом Fade To, пока не поймете, как он работает. После этого перейдем к следующей группе эффектов.

5.2.3. Закатывание и выкатывание элементов

Эффекты следующей группы, скрывающие или отображающие элементы – `slideDown()` и `slideUp()`, – действуют примерно как эффекты `hide()`

и show(), но элемент отображается или скрывается, «выезжая» из-под своей верхней границы или «въезжая» под нее.

Как и в случае с командами hide() и show(), в эту группу эффектов входит эффект переключения состояния элементов между видимым и невидимым: slideToggle(). Ниже приведен теперь уже знакомый синтаксис этих команд.

Синтаксис команды slideDown

`slideDown(speed, callback)`

Делает видимыми элементы из обернутого набора, которые до этого были скрыты, постепенно увеличивая их вертикальный размер. Элементы, которые не были скрыты, не подвергаются действию команды.

Параметры

`speed` (число | строка) Определяет продолжительность воспроизведения эффекта. Может быть числом миллисекунд или одной из предопределенных строк – *slow*, *normal* или *fast*. При отсутствии этого параметра эффект воспроизводится со скоростью, соответствующей значению *normal*.

`callback` (функция) Необязательная функция, которая будет вызвана по окончании воспроизведения эффекта. Функция вызывается без параметров, однако в контексте функции (this) ей передается элемент, над которым выполняется операция.

Возвращаемое значение

Обернутый набор.

Синтаксис команды slideUp

`slideUp(speed, callback)`

Скрывает элементы из обернутого набора, которые до этого не были скрыты, постепенно уменьшая их вертикальный размер.

Параметры

`speed` (число | строка) Определяет продолжительность воспроизведения эффекта. Может быть числом миллисекунд или одной из предопределенных строк – *slow*, *normal* или *fast*. При отсутствии этого параметра эффект воспроизводится со скоростью, соответствующей значению *normal*.

`callback` (функция) Необязательная функция, которая будет вызвана по окончании воспроизведения эффекта. Функция вызывается без параметров, однако в контексте функции (this) ей передается элемент, над которым выполняется операция.

Возвращаемое значение

Обернутый набор.

Синтаксис команды slideToggle

`slideToggle(speed,callback)`

Выполняет команду `slideDown()` для скрытых элементов из обернутого набора и команду `slideUp()` для всех видимых обернутых элементов. Подробные сведения о семантике этих команд приведены в соответствующих описаниях синтаксиса.

Параметры

`speed` (число | строка) Определяет продолжительность воспроизведения эффекта. Может быть числом миллисекунд или одной из предопределенных строк – *slow*, *normal* или *fast*. При отсутствии этого параметра эффект не воспроизводится.

`callback` (функция) Необязательная функция, которая будет вызвана по окончании воспроизведения эффекта. Функция вызывается без параметров, однако в контексте функции (`this`) ей передается элемент, над которым выполняется операция.

Возвращаемое значение

Обернутый набор.

За исключением способа отображения и скрытия элементов, эти эффекты похожи по действию на эффекты `show/hide`. Можете убедиться в этом с помощью упражнений на лабораторной странице jQuery Effects Lab Page, как при изучении других эффектов.

5.2.4. Остановка анимационных эффектов

Иногда требуется остановить воспроизведение анимационного эффекта сразу после его запуска по некоторым причинам. Это может быть, например, событие от пользователя, которому нужно сделать что-то еще, или мы сами хотим запустить другой анимационный эффект. Сделать это позволит команда `stop()`.

Синтаксис команды stop

`stop()`

Останавливает воспроизведение всех анимационных эффектов для элементов из обернутого набора.

Параметры

Нет

Возвращаемое значение

Обернутый набор.

Обратите внимание: все изменения, уже произведенные в каждом элементе, останутся. Если требуется вернуть элементы в исходное состоя-

ние, то мы сами должны будем восстановить прежние значения CSS с помощью метода `css()` или похожих на него команд.

Познакомившись со встроенными базовыми эффектами jQuery, попробуем создать собственные!

5.3. Создание собственных анимационных эффектов

Набор базовых эффектов jQuery ограничен (чтобы не рос объем библиотеки) в предположении, что большее число эффектов может быть доступно авторам страниц в виде подключаемых модулей. Кроме того, дополнительные эффекты легко можно создавать собственными руками.

В библиотеке jQuery есть метод-обертка `animate()`, позволяющий применять к обернутому набору собственные анимационные эффекты. Давайте посмотрим на его синтаксис.

Мы создадим собственные анимационные эффекты с помощью CSS-свойств и конечных значений этих свойств, которые должны быть достигнуты к моменту завершения воспроизведения анимации. Анимаци-

Синтаксис команды animate

`animate(properties, duration, easing, callback)`

`animate(properties, options)`

Применяет анимационный эффект, заданный параметрами `properties` и `easing`, ко всем элементам из обернутого набора. В параметре `callback` можно указать необязательную функцию, которая будет вызываться по завершении воспроизведения анимационного эффекта. В альтернативном варианте в дополнение к аргументу `properties` задается набор параметров в аргументе `options`.

Параметры

`properties` (объект) Объект-хеш, определяющий конечные значения, которых должны достигнуть значения поддерживаемых CSS-стилей к окончанию воспроизведения эффекта. Эффект воспроизводится за счет изменения свойств стиля от текущих значений в элементе до значений, указанных в этом объекте.

`duration` (число | строка) Необязательный параметр. Определяет продолжительность эффекта либо в виде числа миллисекунд, либо в виде одной из строк – *slow*, *normal* или *fast*. Если параметр отсутствует, анимационный эффект не воспроизводится.

`easing` (строка) Необязательный параметр. Имя функции, выполняющей переходы в анимации. Эти функции должны регистрироваться по имени, их часто реализуют в виде подключаемых модулей. Библиотека jQuery предоставляет две такие функции, зарегистрированные как `linear` (линейный переход) и `swing` (колебательный переход).

`callback`	(функция) Необязательный параметр. Функция, которая будет вызвана по завершении воспроизведения эффекта анимации. Функция вызывается без параметров, однако в контексте функции (`this`) ей передается элемент, над которым выполняется операция.
`options`	(объект) Объект-хеш, который содержит значения параметров анимационного эффекта. Поддерживаются следующие свойства: • `duration` (см. описание параметра `duration` выше); • `easing` (см. описание параметра `easing` выше); • `complete` – функция, которая должна быть вызвана по окончании воспроизведения анимации; • `queue` – если имеет значение `false`, анимационный эффект не ставится в очередь, а начинает воспроизводиться немедленно.

Возвращаемое значение

Обернутый набор.

онные эффекты начинают с исходных значений свойств стиля элемента, постепенно изменяя их в направлении заданных конечных значений. Промежуточные значения свойств стиля (автоматически вычисляются механизмом анимации в ходе воспроизведения эффекта) определяются продолжительностью эффекта и функцией реализации перехода.

Конечные значения можно задавать абсолютной величиной или смещением относительно начальных значений. Относительные значения должны предваряться оператором `+=` (положительное смещение) или `-=` (отрицательное смещение).

Функция перехода (*easing*) описывает изменение темпа и порядка обработки кадров анимации. При сложном математическом преобразовании скорости воспроизведения и текущей отметки времени возможны весьма интересные анимационные эффекты. Тема написания функций перехода очень сложна и представляет интерес только для матерых авторов подключаемых модулей; мы в этой книге не будем погружаться в тему создания собственных функций перехода. Множество функций перехода, отличных от линейной (равномерный переход) или колебательной (переход с небольшим ускорением к концу анимации), вы можете увидеть на странице проекта Easing Plugin по адресу *http://gsgd.co.uk/sandbox/jquery.easing.php*.

По умолчанию анимационный эффект добавляется в очередь для исполнения – применение нескольких анимационных эффектов к объекту приведет к тому, что они будут выполнены последовательно. Если хотите запускать анимационные эффекты параллельно, установите параметр `queue` в значение `false`.

Перечень свойств CSS-стилей ограничен только принимающими числовые значения, для которых переход от начального значения к ко-

нечному логичен. Это ограничение совершенно понятно, например: как логически представить переход от начального значения к конечному для нечисловых значений, таких как image-background? Для значений, представленных в определенных единицах измерения, jQuery предполагает, что по умолчанию они измеряются в пикселах, но можно выбрать и такие единицы измерения, как em или проценты, добавив суффикс *em* или %.

Часто анимационный эффект применяют к таким свойствам стиля, как top, left, width, height и opacity. Однако допустимо использовать такие числовые свойства стиля, как размер шрифта и толщина рамки, если это имеет смысл для создания требуемого эффекта.

Примечание

Если вам понадобится эффект, воздействующий на цвет, вас наверняка заинтересует официальный модуль расширения Color Animation Plugin, который вы найдете по адресу *http://jquery.com/plugins/project/color*.

В дополнение к определенным значениям можно также указать одну из строк hide, show или toggle – jQuery вычислит соответствующее конечное значение, подразумеваемое строкой. Например, использование строки hide для описания воздействия на свойство opacity приведет к тому, что непрозрачность элемента будет уменьшена до 0. Использование любой из этих специальных строк автоматически добавляет операцию отображения или удаления элемента с экрана (подобно командам show() и hide()).

Когда мы рассматривали базовые анимационные эффекты, заметили ли вы, что для эффектов проявления/растворения (fadeIn/fadeOut) отсутствует команда переключения? Этот недостаток легко исправить с помощью animate() и значения toggle:

```
$('.animateMe').animate({opacity:'toggle'},'slow');
```

Попробуем создать несколько анимационных эффектов своими руками.

5.3.1. Эффект масштабирования

Рассмотрим простой анимационный эффект масштабирования – двукратное увеличение первоначальных размеров элементов. Можно реализовать этот эффект так:

```
$('.animateMe').each(function(){
  $(this).animate(
    {
      width: $(this).width() * 2,
      height: $(this).height() * 2
    },
    2000
  );
});
```

Для воспроизведения этого эффекта мы производим обход всех элементов в обернутом наборе с помощью команды `each()`, применяя эффект отдельно к каждому элементу. Это очень важно, поскольку свойства, которые мы должны указать для каждого элемента, имеют разные начальные значения. Если бы заранее было известно, что эффект применяется к единственному элементу (как в случае использования селектора `id`), или мы применяли бы один и тот же набор значений ко всем элементам, то можно было бы обойтись и без команды `each()`, воздействуя непосредственно на весь обернутый набор.

Внутри функции-итератора к элементу (идентифицируемому значением `this`) применяется команда `animate()` со значениями для свойств стилей `width` и `height`, увеличенными в два раза относительно их первоначальных значений. В результате в течение двух секунд (задаваемых значением `2000` параметра `duration`) размеры элементов вырастают в два раза.

Давайте попробуем создать что-нибудь более экстравагантное.

5.3.2. Эффект падения

Предположим, нам захотелось анимировать удаление элемента из отображения страницы, при этом наглядно показать пользователям, что удаляемый элемент *исчез*, и это не должно вызывать у них сомнений. Эффект, который мы имитируем: элемент выпадает из страницы и исчезает с экрана.

Тут не о чем долго думать: изменяя значение свойства `top` элемента, мы заставим его перемещаться вниз по странице, как бы в падении, а изменение непрозрачности усилит эффект исчезновения. Наконец, когда все необходимое будет сделано, элемент следует удалить с экрана (как в анимированной версии команды `hide()`).

Получить подобный эффект падения можно с помощью следующего программного кода:

```
$('.animateMe').each(function(){
  $(this)
    .css('position','relative')
    .animate(
      {
        opacity: 0,
        top: $(window).height() - $(this).height() -
             $(this).position().top
      },
      'slow',
      function(){ $(this).hide(); });
});
```

Здесь пояснений будет чуть больше, чем в предыдущем примере. Мы снова выполняем обход набора элементов, но на этот раз изменяем координату и степень непрозрачности элементов. Но чтобы изменить

значение свойства top элемента относительно его исходного значения, сначала мы должны изменить свойство стиля CSS position, установив его в значение relative.

Затем при вызове команды animate()указываем конечное значение 0 для свойства opacity и вычисленное значение координаты top. Не надо перемещать элемент за нижнюю границу окна – могут появиться нелюбимые пользователями полосы прокрутки, которых, возможно, до этого не было на странице. Нам нужно лишь привлечь внимание к анимационному эффекту – собственно, за этим он и создается! Поэтому мы вычисляем всю низость падения элемента с помощью текущей вертикальной координаты элемента, его высоты и высоты окна.

Примечание

В большинстве примеров этой книги мы постарались обойтись без модулей расширения, чтобы сконцентрировать все внимание на базовых возможностях jQuery. На самом деле, обычно наряду с базовыми возможностями jQuery используются модули расширения, необходимые авторам страниц для решения их задач. Простота создания собственных модулей и богатое разнообразие уже существующих – две самые сильные стороны jQuery. Начиная с этого примера анимационного эффекта, мы определяем начальное положение элемента относительно страницы с помощью команды position() из модуля расширения Dimensions Plugin. Мы достаточно подробно рассмотрим этот модуль в главе 9 (точнее, в разд. 9.2).

По завершении воспроизведения анимационного эффекта нужно удалить элемент из отображения страницы, поэтому мы указываем функцию обратного вызова, которая применит к элементу не анимированную версию команды hide() (доступную этой функции через ее контекст функции).

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 по окончании воспроизведения анимационного эффекта элемент был бы автоматически удален, и функция обратного вызова не понадобилась бы.

Теперь для полноты картины попробуем реализовать еще один эффект типа «было и прошло».

5.3.3. Эффект рассеивания

Предположим, что вместо эффекта падения требуется воспроизвести эффект рассеивания, как тают клубы дыма в воздухе. Мы воссоздадим этот эффект комбинацией эффектов масштабирования и изменения непрозрачности, увеличивая размеры элемента и одновременно растворяя

его до полного исчезновения. Одна из проблем, которую нам придется решить для пущей реалистичности, – элемент должен разрастаться 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 следует изменять и его координаты.

Вот программный код, реализующий эффект рассеивания:

```
$('.animateMe').each(function(){
  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)
      },
      'normal');
});
```

Здесь мы уменьшаем непрозрачность до 0, одновременно пятикратно увеличиваем первоначальные размеры элемента и смещаем начало координат элемента на половину нового размера. В результате центр элемента остается на месте. Растущий элемент не должен выталкивать окружающие элементы с их мест, поэтому мы изымаем его из потока документа, установив его свойство `position` в значение `absolute`, и задаем его координаты явно.

Поскольку для свойства `opacity` мы указали значение `hide`, элементы автоматически будут скрыты (удалены из отображения страницы) по завершении воспроизведения эффекта.

Действие каждого из трех эффектов можно понаблюдать, открыв страницу *chapter5/custom.effects.html* (рис. 5.5).

Перед тем как сделать снимок экрана, мы уменьшили окно броузера – вы же перед запуском эффектов можете развернуть окно, чтобы лучше видеть, как они действуют. Нам очень хотелось бы продемонстрировать эти эффекты здесь, но процедура создания скриншотов имеет свои очевидные ограничения. Тем не менее на рис. 5.6 показан эффект рассеивания в процессе воспроизведения.

5.4. Итоги

В этой главе мы познакомились с анимационными эффектами, входящими в состав jQuery, а также с методом-оберткой `animate()`, позволяющим создавать собственные анимационные эффекты.

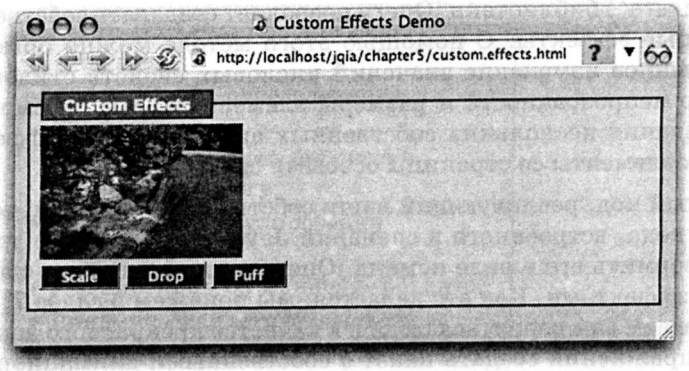


Рис. 5.5. *Первоначальный вид страницы, где реализованы три созданных нами эффекта: масштабирование, падение и рассеивание*

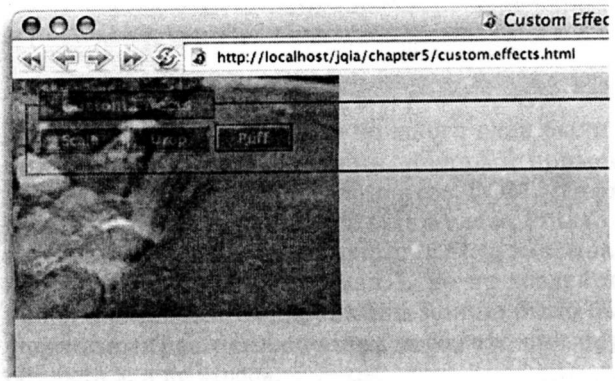


Рис. 5.6. *Эффект рассеивания: элемент увеличивается в размерах, одновременно перемещаясь, при этом степень его непрозрачности уменьшается*

Команды `show()` и `hide()` при вызове без параметров отображают и скрывают элементы немедленно, без анимационных эффектов. С помощью анимационных версий этих команд можно отображать и скрывать элементы, передавая им параметры, управляющие скоростью анимации, а также функцию обратного вызова, которая вызывается по завершении анимационного эффекта. Команда `toggle()` переключает визуальное состояние элемента между видимым и невидимым.

Другая группа методов, `fadeOut()` и `fadeIn()`, также скрывает и отображает элементы путем изменения степени непрозрачности элементов. Третий метод из этой группы `fadeTo()` изменяет непрозрачность обернутых элементов без удаления их из отображения страницы.

Последняя группа встроенных анимационных эффектов удаляет или отображает обернутые элементы, изменяя их высоту: `slideUp()`, `slideDown()` и `slideToggle()`.

Команда `animate()` библиотеки jQuery позволяет создавать собственные анимационные эффекты. С помощью этого метода можно организовать постепенное изменение значений числовых свойств стилей CSS (обычно это непрозрачность и размеры элементов). Мы рассмотрели процесс создания нескольких собственных анимационных эффектов, удаляющих элементы со страницы особыми способами.

Программный код, реализующий наши собственные эффекты, оформлен в виде кода, встроенного в сценарий JavaScript. Гораздо удобнее было бы оформить его в виде команд jQuery в составе пакета с анимационными эффектами. Как это делается, мы покажем в главе 7, после чего предлагаем вам вернуться сюда и в качестве прекрасного дополнительного упражнения создать пакет с собственными анимационными эффектами из этой главы и другими, которые вы сможете придумать.

В этой главе:

- Флаги jQuery, определяющие тип броузера
- Применение других библиотек совместно с jQuery
- Функции манипулирования массивами
- Расширение и объединение объектов
- Динамическая загрузка нового сценария

6

Вспомогательные функции jQuery

До этого момента достаточно много глав были посвящены исследованию *команд* jQuery – термин, который применяется к методам, воздействующим на набор элементов DOM, обернутый в функцию `$()`. Возможно, вы помните, что еще в главе 1 было введено такое понятие, как *вспомогательные функции* (*utility functions*), – функции, принадлежащие пространству имен `$`, но не воздействующие на обернутый набор элементов. Эти функции можно было бы считать глобальными функциями, за исключением того, что они определены в экземпляре объекта `$`, а не `window`.

Как правило, эти функции либо воздействуют на объекты JavaScript, которые *не являются* элементами DOM (в конце концов, это прерогатива команд), либо они выполняют некоторые другие операции, не имеющие отношения к объектам.

Вы можете удивиться, почему мы отложили знакомство с этими функциями до этой главы. Что ж, на то у нас были две основные причины.

- Мы хотели научить вас мыслить в терминах применения команд jQuery, вместо того чтобы показать, как пользоваться низкоуровневыми операциями, которые вам более знакомы, но не столь эффективны и просты, как команды jQuery.
- Поскольку команды в значительной степени учитывают, что мы предполагаем сделать, манипулируя элементами DOM на страницах, эти низкоуровневые функции, зачастую более полезны при написании самих команд (и других расширений), а не для создания программного кода уровня страницы. (О том, как писать собственные расширения, мы поговорим в следующей главе.)

В этой главе мы наконец подошли к официальному представлению большинства вспомогательных функций уровня $ и небольшой группы полезных флагов. Мы отложим разговор о вспомогательных функциях, обеспечивающих поддержку технологии Ajax, до главы, в которой будут рассматриваться только функциональные возможности Ajax, присутствующие в библиотеке jQuery.

Начнем с уже упомянутых флагов.

6.1. Флаги jQuery

Библиотека jQuery предоставляет нам как авторам страниц, и даже расширений, некоторую информацию – но не через методы или функции, а в виде переменных, определенных в пространстве имен $. Эти *флаги* помогают нам определить особенности текущего броузера, чтобы в случае необходимости мы могли принимать решения на основе этой информации.

Следующие флаги jQuery предназначены для общего использования:

- `$.browser`
- `$.boxModel`
- `$.styleFloat`

Для начала посмотрим, как jQuery сообщает нам тип активного броузера.

6.1.1. Определение типа броузера

К счастью, команды jQuery, которые мы применяли до сих пор, защищали нас от необходимости задумываться о различиях между разными типами броузеров, даже в таких традиционно проблемных областях, как обработка событий. Но если мы сами пишем эти команды (или другие расширения), то зачастую приходится учитывать различия между броузерами, чтобы избавить от проблем пользователей наших расширений.

Прежде чем углубиться в описание этих возможностей jQuery, поговорим о концепции определения типа броузера.

Почему определять тип броузера – так ужасно

Ладно, *ужасно* – пожалуй, слишком сильно сказано, но за исключением случаев абсолютной необходимости, определение типа броузера – это прием, который следует применять только если нет других вариантов.

Определение типа броузера, на первый взгляд, может показаться вполне логичным способом борьбы с различиями броузеров. Казалось бы, простой вопрос: «Почему бы не проверять тип броузера, если знаешь возможности каждого?» Но при определении типа броузера вы можете столкнуться с массой ловушек и проблем.

Один из основных аргументов против применения этого метода заключается в том, что постоянное увеличение числа броузеров, а также различные уровни поддержки в пределах версий одного и того же броузера делают такой подход к решению проблемы неприемлемым.

Можно подумать: «Да все, что нужно проверить, – это с каким типом броузера я работаю, с Internet Explorer или Firefox». А вы не забыли, что пользователей Safari все больше? И как быть с броузером Opera? Кроме того, есть определенные ниши, хоть и незначительные, занимаемые броузерами, которые используют возможности более популярных броузеров. Например, Camino, применяющий ту же технологию, что и Firefox, был разработан для обеспечения поддержки пользовательского интерфейса Mac OS. А в OmniWeb задействован тот же механизм отображения, что и в Safari.

Исключать поддержку этих типов броузеров не хочется, но необходимость их проверки чревата ужасной головной болью. И это еще без учета различий между версиями одного и того же броузера, например между IE6 и IE7.

Еще один аргумент против определения типа броузера – или *сниффинга* (*sniffing*), как его иногда называют: все труднее определить, «кто есть кто».

Броузеры идентифицируют себя с помощью *заголовка запроса*, называемого *user agent*. Разбор этой строки – занятие не для слабонервных. Кроме того, сегодня многие броузеры позволяют пользователям изменить этот заголовок, поэтому мы не можем доверять информации, полученной после *анализа* заголовка!

Объект JavaScript, который называется `navigator`, дает нам частичное представление об информации в заголовке user agent, но даже *этот объект* отличается для разных броузеров. Фактически нам придется определить тип броузера, чтобы потом определить тип броузера!

Определение типа броузера может быть:

- *неточным* – из-за некоторых особенностей броузеров, под управлением которых будет выполняться наш программный код;
- *неприемлемым* – из-за необходимости писать большое число вложенных условных инструкций `if` и `if-else`, чтобы решить проблему;
- *ошибочным* – из-за того, что пользователь может указать ложную информацию в заголовке user agent.

Очевидно, что мы хотим по возможности избежать этого.

Но что мы можем сделать?

Есть ли альтернатива?

Если хорошенько задуматься, то окажется, что *в действительности* нас не интересует то, какой броузер используется, не так ли? Единственное, ради чего хотелось бы определить тип броузера, – узнать, какие

возможности броузера мы можем использовать. Эти *возможности* мы и будем использовать в дальнейшем, а определение типа броузера – неуклюжая попытка их выяснить.

Почему бы не определять эти возможности напрямую, вместо того чтобы предполагать их наличие, исходя из строки идентификации броузера? Методика, широко известная как *обнаружение объекта* (*object detection*), позволяет программному коду определять доступные объекты, свойства и даже методы.

Давайте для примера вспомним, о чем говорилось в главе, посвященной обработке событий. Мы знаем две современные модели обработки событий – стандартную модель событий DOM уровня 2 и модель Internet Explorer. Обе модели определяют методы элементов DOM, позволяющие устанавливать обработчики событий, но имена методов в каждой модели разные. Стандартная модель определяет метод `addEventListener()`, тогда как модель IE определяет метод `attachEvent()`.

Смирившись с определением типа броузера и предполагая, что несмотря на все трудности нам удалось определить его тип (возможно, даже правильно), мы можем написать:

```
...
complex code to set flags: isIE, isFirefox and isSafari
...
if (isIE) {
  element.attachEvent('onclick',someHandler);
}
else if (isFirefox || isSafari) {
  element.addEventListener('click',someHandler);
}
else {
  throw new Error('event handling not supported');
}
```

Помимо того что в этом примере не показано, какие сложности пришлось преодолеть, чтобы установить флаги `isIE`, `isFirefox` и `isSafari`, нет уверенности, что эти флаги точно определяют используемый броузер. Более того, этот код будет выдавать ошибку в броузерах Opera, Camino, OmniWeb и во множестве других малоизвестных броузеров, прекрасно поддерживающих стандартную модель.

А вот другой вариант:

```
if (element.attachEvent) {
  element.attachEvent('onclick',someHandler);
}
else if (element.addEventListener) {
  element.addEventListener('click',someHandler);
}
else {
  throw new Error('event handling not supported');
}
```

Этот код не выполняет множество сложных и, в конечном счете, ненадежных операций для определения типа броузера. Он автоматически обеспечивает поддержку всех броузеров, реализующих одну из двух конкурирующих моделей обработки событий. Так гораздо лучше!

Прием, основанный на обнаружении объекта, обладает значительными преимуществами по сравнению с определением типа броузера. Он более надежен и не приводит к ошибке в броузерах, поддерживающих проверяемую функциональную возможность, только из-за того, что нам неизвестны возможности того или иного броузера.

Примечание

Даже прием обнаружения объекта желательно не использовать без абсолютной необходимости. Предпочтительным будет решение, приемлемое для всех типов броузеров, – если вам удастся его найти.

Но даже обнаружение объекта, которое лучше определения типа броузера, поможет не всегда. Порой требуются конкретные решения (вскоре мы увидим это на примере), достижимые только путем определения типа броузера.

Итак, без лишней суеты, вернемся, наконец, к ответу на вопрос...

Есть ли флаги, определяющие тип броузера?

Для случаев, когда обойтись без определения типа броузера невозможно, jQuery предоставляет набор флагов, позволяющих выполнить настройку при загрузке библиотеки и доступных еще до того, как будет запущен обработчик события готовности документа. Они определяются как свойства экземпляра объекта со ссылкой `$.browser`. Формальный синтаксис этого набора флагов:

Синтаксис набора флагов $.browser

`$.browser`

Определяет набор флагов, позволяющих определить, к какому семейству принадлежит текущий броузер. В этот набор входят следующие флаги:

- `msie` – содержит значение `true`, если заголовком user agent броузер идентифицирует себя как Internet Explorer.
- `mozilla` – содержит значение `true`, если заголовком user agent броузер идентифицирует себя как любой броузер, основанный на Mozilla, включая такие как Firefox, Camino и Netscape.
- `safari` – содержит значение `true`, если заголовком user agent броузер идентифицирует себя как Safari или любой броузер, основанный на Safari, например OmniWeb.
- `opera` – содержит значение `true`, если заголовком user agent броузер идентифицирует себя как Opera.
- `version` – содержит номер версии механизма отображения броузера.

Обратите внимание: эти флаги не пытаются точно идентифицировать используемый броузер; jQuery классифицирует броузер с помощью заголовка user agent, определяя, к какому *семейству* броузеров он принадлежит. Броузеры одного семейства обладают одинаковыми наборами характеристик; определение конкретного броузера не является необходимым.

Подавляющее большинство популярных современных броузеров попадут в одно из этих четырех семейств.

Свойство `version` заслуживает особого внимания, потому что оно не столь удобно, каким кажется. Значение, установленное в этом свойстве, не является версией броузера (как можно было бы предположить), – это номер версии механизма отображения броузера. Например, программный код, выполняемый под управлением Firefox 2.0.0.2, получает в этом флаге значение `1.8.1.6` – версию механизма отображения Gecko. Это значение *удобно* для различения IE6 и IE7, в которых этот флаг содержит значение `6.0` и `7.0` соответственно.

Ранее уже говорилось, что бывают моменты, когда прием обнаружения объекта неприменим и приходится прибегать к определению типа броузера. Один из примеров таких ситуаций – случай, когда проблема состоит не в том, что броузеры реализуют различные классы или различные методы объекта, а в том, что передаваемые методу параметры в разных броузерах интерпретируются по-разному. В этом случае нет объекта, который можно было бы обнаружить.

Давайте посмотрим на метод `add()` элемента `<select>`. Он определяется так:

```
selectElement.add(element,before)
```

Первый параметр этого метода определяет элемент `<option>` или `<optgroup>`, добавляемый в элемент `<select>`, а второй параметр определяет существующий элемент `<option>` (или `<optgroup>`), перед которым будет добавлен новый элемент. В броузерах, соответствующих стандарту, второй параметр является *ссылкой* на существующий элемент, а в Internet Explorer это *порядковый индекс* существующего элемента.

Поскольку нет никакого другого способа узнать, что следует передать методу – ссылку или целое число, мы вынуждены определять тип броузера, как показано в примере из листинга 6.1, который можно найти в файле *chapter6/$.browser.html*.

Листинг 6.1. Проверка типа броузера

```
<html>
  <head>
    <title>$.browser Example</title>
    <link rel="stylesheet" type="text/css" href="../common.css">
    <script type="text/javascript"
            src="../scripts/jquery-1.2.1.js"></script>
    <script type="text/javascript">
```

```
        $(function(){
          $('#testButton').click(function(event){
            var select = $('#testSubject')[0];
            select.add(
              new Option('Two and \u00BD','2.5'),
              $.browser.msie ? 2 : select.options
            );
          });
        });
      </script>
    </head>

    <body class="plain">
      <select id="testSubject" multiple="multiple" size="5">
        <option value="1">One</option>
        <option value="2">Two</option>
        <option value="3">Three</option>
        <option value="4">Four</option>
      </select>
      <div>
        <button type="button" id="testButton">Click me!</button>
      </div>
    </body>
  </html>
```

Используется определение типа броузера для второго параметра

В этом примере создаются элемент <select> с четырьмя записями и простая кнопка. Кнопка добавляет новый элемент <option> между изначально вторым и третьим элементами <option>. Поскольку Internet Explorer ожидает порядковый номер 2, а броузеры, совместимые со стандартами W3C, ожидают ссылку на третий существующий элемент <option>, мы определяем тип броузера, чтобы правильно передать второй параметр методу add().

Результаты до и после щелчка на кнопке для различных броузеров показаны на рис. 6.1

Этот пример был опробован в шести современных броузерах – Firefox, Safari, Opera, Camino, Internet Explorer 7 и OmniWeb, которые представляют все четыре семейства броузеров, поддерживающих набор флагов $.browser библиотеки jQuery. Как видите, добавление элемента <option> правильно выполняется в каждом из броузеров.

Важно еще раз подчеркнуть, что этот метод (определение типа броузера) далеко не оптимален. Наш программный код предполагает, что все броузеры, не являющиеся Internet Explorer, относятся к семейству, которое будет следовать стандартам W3C для метода add() элемента <select>. В результате испытаний мы установили, что он правильно работает в пяти броузерах, отличных от Internet Explorer, как показано на рис. 6.1. Но как быть с другими броузерами? Может быть, вы и без испытаний знаете, как работает Konqueror?

Суть в том, что из-за приблизительности определения типа броузера придется тщательно изучить все типы броузеров и платформ, которые

***Рис. 6.1.** Функция добавления элемента <option> безупречно работает в самых разных броузерах*

мы хотели бы поддерживать, чтобы узнать, как обслуживать каждый конкретный броузер.

Оставим вопрос определения типа броузера и рассмотрим другой флаг, позволяющий преодолеть различия между броузерами.

6.1.2. Определение блочной модели

Определить *блочную модель* (*box model*), применяемую текущей страницей, позволяет флаг `$.boxModel` логического типа, который имеет значение `true`, если это блочная модель стандарта W3C, и значение `false`,

если это блочная модель Internet Explorer (иногда ее называют *традиционной* блочной моделью).

«В чем разница?» – спросите вы.

Блочная модель устанавливает порядок определения размеров содержимого элемента с учетом отступов и рамки (полей, хотя эта часть блочной модели никак не участвует в определении размера содержимого). Большинство броузеров, кроме Internet Explorer, поддерживают только блочную модель W3C, а Internet Explorer может использовать одну из двух моделей, в зависимости от режима отображения страницы, – в *строгом режиме* или в *режиме обратной совместимости* (*quirks mode*). Выбор режима зависит от объявления DOCTYPE (или от его отсутствия) для исполняемой страницы.

Описание проблем, связанных с объявлением DOCTYPE, далеко выходит за рамки этой книги, и мы не будем вдаваться в подробный анализ различных DOCTYPE, приведя лишь некоторые правила, работающие в большинстве случаев:

- страницы с правильным и распознаваемым объявлением DOCTYPE отображаются в строгом режиме;
- страницы без объявления DOCTYPE или содержащие ошибочное определение отображаются в режиме обратной совместимости.

Более подробную информацию о проблемах, связанных с объявлением DOCTYPE, вы можете получить на сайте *http://www.quirksmode.org/css/quirksmode.html*.

В двух словах, разница между этими двумя блочными моделями состоит в том, как интерпретируются стили `width` и `height`. В блочной модели W3C эти значения определяют размеры содержимого элементов без учета отступов и ширины рамки, а в традиционной блочной модели эти значения *включают* отступы и ширину рамки.

Предположим, у нас есть элемент с таким описанием стилей:

```
{
  width: 180px;
  height: 72px;
  padding: 10px;
  border-width: 5px;
}
```

Разница между способами интерпретации размеров элемента в блочных моделях показана на рис. 6.2.

В блочной модели W3C размеры содержимого элемента составляют 180 на 72 пиксела – в точности, как указано в значениях `width` и `height`. Отступы и рамка находятся *за пределами* этой зоны 180 на 72 пиксела, в результате чего общие размеры для всего элемента составляют 210 на 102 пиксела.

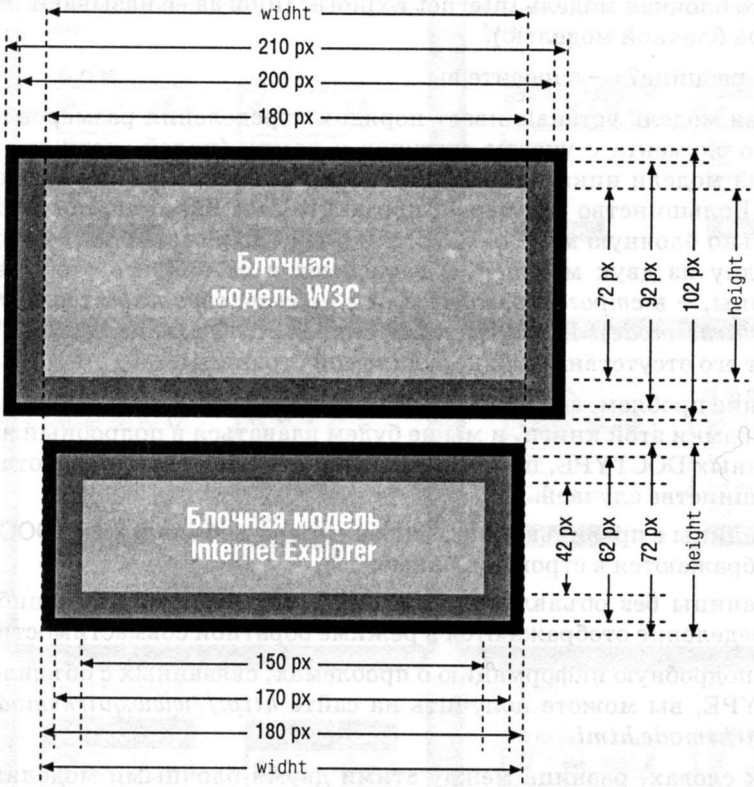


***Рис. 6.2.** В блочной модели W3C ширина элемента не включает отступы и ширину рамки, а в Internet Explorer учитываются ширина рамки и отступы*

В традиционной блочной модели весь элемент отображается в поле 180 на 72 пиксела, как определено атрибутами `width` и `height`, и размер содержимого элемента уменьшается до 150 на 42 пиксела.

Есть два противоположных мнения о том, какая из этих моделей более понятна и правильна, но факт остается фактом: нам придется мириться с этим различием.

Если для правильного отображения элементов приходится учитывать эти различия в программном коде, флаг `$.boxModel` поможет узнать, какая модель используется, и выполнить вычисления соответствующим образом.

6.1.3. Определение правильного имени для стиля float

Между броузерами есть еще одно различие, для обслуживания которого jQuery предоставляет флаг, содержащий имя для представления CSS-стиля `float` в свойстве `style` элемента. Это флаг `$.StyleFloat`, который описывает строку для замещения имени этого свойства.

Например, установить значение свойства `float` элемента можно так:

```
element.style[$.styleFloat] = 'left';
```

Это приводит в соответствие разные имена этого свойства в различных броузерах – этот флаг возвращает имя `styleFloat` для Internet Explorer и `cssFloat` для других броузеров.

Примечание

Этот флаг вообще не нужно использовать в страницах. Метод обертки `css()` автоматически выбирает правильное имя свойства (с помощью этого флага). Его используют авторы подключаемых модулей и других расширений, если требуется контроль на низком уровне.

Давайте покинем мир флагов и посмотрим на вспомогательные функции, которые предоставляет jQuery.

6.2. Применение других библиотек совместно с jQuery

В разделе 1.3.6 мы узнали, как команда разработчиков jQuery позаботилась о том, чтобы мы могли спокойно применять jQuery на одной странице с другими библиотеками. Обычно при совместном использовании jQuery и других библиотек на одной и той же странице камнем преткновения является определение переменной $. Как известно, jQuery использует идентификатор $ в качестве псевдонима имени `jQuery`, которое используется при каждом вызове функций jQuery. Но имя $ носят и другие библиотеки, прежде всего Prototype.

jQuery предоставляет вспомогательную функцию `$.noConflict()`, которая освобождает имя $, чтобы любая другая библиотека могла его использовать. Синтаксис этой функции:

Синтаксис функции $.noConflict

`$.noConflict()`

Передает контроль над именем $ другой библиотеке, что позволяет применять различные библиотеки на тех же страницах, что и jQuery.

После того как эта функция выполнится, все функции библиотеки jQuery придется вызывать с именем `jQuery`, а не $.

Параметры

Нет

Возвращаемое значение

Не определено.

Поскольку $ – это псевдоним для имени `jQuery`, после вызова функции `$.noConflict()` все функциональные возможности jQuery по-прежнему

доступны, но уже исключительно с применением идентификатора jQuery. Чтобы компенсировать потерю короткого и любимого идентификатора $, можно определить собственный короткий (но не конфликтный) псевдоним для jQuery, например:

```
var $j = jQuery;
```

Еще одна часто используемая идиома состоит в создании окружения, где имя $ будет ссылаться на объект jQuery. Этот прием широко используют при создании расширений для jQuery, особенно те авторы расширений, которые не знают и не могут знать, вызывали ли авторы страниц функцию $.noConflict(), и которые никак не могут противостоять желанию авторов страниц вызвать ее.

Эта идиома выглядит так:

```
(function($) { /* здесь располагается тело функции */ })(jQuery);
```

Если вас что-то смутило, не волнуйтесь! Тут нет ничего сложного, хотя эта форма записи странновато выглядит для тех, кому она в новинку.

Давайте проанализируем первую часть этой идиомы:

```
(function($) { /* здесь располагается тело функции */ })
```

Эта часть объявляет функцию – в круглых скобках, чтобы превратить ее в выражение с результатом в виде ссылки на анонимную функцию, возвращаемую как значение выражения. Функции передается единственный параметр с именем $; на все, что передается этой функции, внутри нее можно ссылаться по идентификатору $. И поскольку объявления параметров имеют приоритет над любыми идентификаторами глобальной области видимости, любое значение, определенное для идентификатора $ вне функции, в пределах функции будет заменено переданным аргументом.

Вторая часть идиомы

```
(jQuery);
```

выполняет вызов функции, передавая анонимной функции объект jQuery в качестве аргумента.

В результате в теле функции имя $ ссылается на объект jQuery независимо от того, определено ли оно библиотекой Prototype или какой-либо другой библиотекой за пределами функции. Неплохо придумано, а?

При данном подходе внешнее объявление $ недоступно в пределах тела функции.

Вариант этой идиомы также часто позволяет объявить обработчик события готовности документа – в виде третьего синтаксиса, в дополнение к средствам, которые мы уже исследовали в разделе 1.3.3. Например:

```
jQuery(function($) {
  alert("I'm ready!");
});
```

Передавая функцию как параметр функции jQuery, мы объявляем ее обработчиком события готовности документа, как было показано в разделе 1.3.3. Но на этот раз мы задаем единственный параметр, передаваемый этому обработчику, с помощью идентификатора $. Поскольку jQuery всегда передает ссылку на jQuery обработчику события готовности документа в виде первого и единственного параметра, это гарантирует, что имя $ будет ссылаться на jQuery независимо от определения $, которое может располагаться за пределами обработчика.

Давайте докажем это самим себе с помощью простого теста. В качестве первой части теста исследуем HTML-документ из листинга 6.2.

Листинг 6.2. Тест 1 обработчика события готовности документа

```
<html>
  <head>
    <title>Hi!</title>
    <script type="text/javascript"
            src="../scripts/jquery-1.2.1.js">
    </script>
    <script type="text/javascript">
      var $ = 'Hi!';
      jQuery(function(){
        alert('$ = '+ $);
      });
    </script>
  </head>
  <body>
  </body>
</html>
```

❶ Переопределяет имя $ собственным определением

❷ Объявляет обработчик события готовности документа

В этом документе мы импортируем библиотеку jQuery, которая (как мы знаем) определяет глобальное имя jQuery и его псевдоним $. Затем мы переопределяем глобальное имя $ присваиванием строковой переменной ❶, отменив определение jQuery. Мы заменяем $ простой строковой переменной исключительно для упрощения данного примера, но его можно переопределить путем включения другой библиотеки, такой как Prototype.

Затем мы определяем обработчик события готовности документа ❷, который выводит предупреждение, отображая значение $.

При открытии этой страницы мы увидим предупреждение (рис. 6.3).

Обратите внимание: в пределах обработчика события готовности документа *глобальное* значение $ для этой области видимости оказывается ожидаемым, полученным в результате операции присваивания строки. Неутешительно, так как мы хотели использовать в обработчике определение $, которое было дано библиотекой jQuery.

Давайте немного изменим этот пример документа. В листинге 6.3 приведен только отредактированный фрагмент документа (изменения выделены жирным шрифтом).

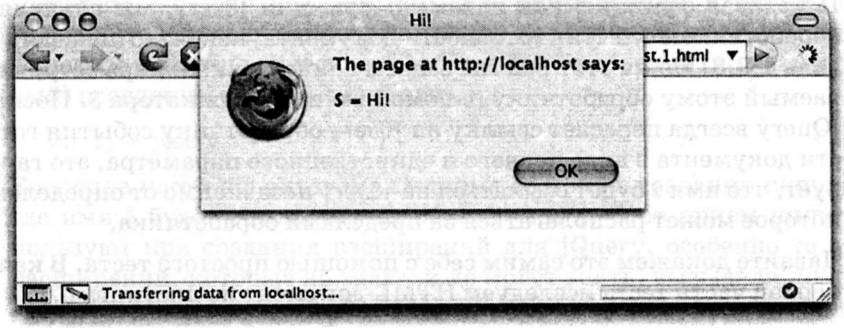


Рис. 6.3. Идентификатору $ соответствует значение «Hi!», так как внутри обработчика события готовности документа действует выполненное выше переопределение

Листинг 6.3. Тест 2 обработчика события готовности документа

```
<script type="text/javascript">
  var $ = 'Hi!';
  jQuery(function($){
    alert('$ = '+ $);
  });
</script>
```

Единственное изменение заключалось в том, что мы добавили передачу параметра с именем $ в функцию обработчика события готовности документа. Если открыть в броузере эту измененную версию страницы, мы увидим нечто совершенно иное (рис. 6.4).

Возможно, это не совсем то, что мы предполагали. Но быстрый взгляд на исходный код jQuery показывает, что благодаря объявлению первого параметра с именем $, внутри этой функции имя $ будет ссылаться на функцию `jQuery`, которая передается как единственный параметр

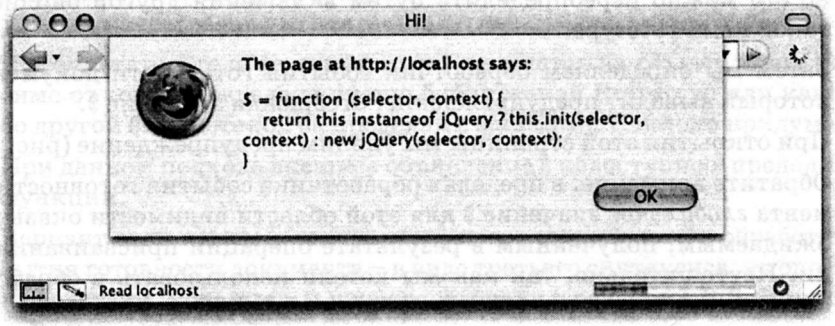


Рис. 6.4. Теперь в окне диалога отображается версия $ из библиотеки jQuery, потому что именно это определение действует в пределах функции

всем обработчикам события готовности документа (поэтому в диалоге выводится определение той функции).

При написании компонентов многократного использования, к которым страницы обращаются или нет при вызове функции `$.noConflict()`, лучше принять такие меры предосторожности в отношении определения `$`.

Большинство оставшихся вспомогательных функций jQuery позволяют манипулировать объектами JavaScript. Давайте посмотрим на эти функции.

6.3. Управление объектами и коллекциями JavaScript

Многие функции jQuery реализованы как вспомогательные функции для работы с объектами JavaScript, не являющимися элементами DOM. Обычно все, что предназначено для работы с DOM, реализуется как команда jQuery. Хотя некоторые из этих функций и позволяют работать с элементами DOM, которые сами *являются* объектами JavaScript, работа с DOM – не главная область применения вспомогательных функций.

Начнем с основ.

6.3.1. Усечение строк

Это трудно объяснить, но объект `String` в JavaScript не обладает методом удаления пробельных символов в начале и в конце строки. Такие базовые функциональные возможности обычно являются обязательной частью класса `String` в большинстве других языков, но по каким-то таинственным причинам в JavaScript нет этой полезной функции.

Несмотря на это, операция усечения строк требуется во многих приложениях JavaScript; один из ярких примеров – проверка данных формы. Поскольку пробельные символы невидимы на экране (отсюда и название), пользователь вполне может случайно ввести дополнительные пробельные символы после (а иногда и до) значимой записи в текстовом поле или текстовой области. При проверке можно просто удалить лишние пробельные символы, не сообщая пользователю о чем-то невидимом для него.

Чтобы помочь нам, библиотека jQuery определяет функцию `$.trim()`.

Небольшой пример усечения значения в текстовом поле с помощью этой функции:

```
$('#someField').val($.trim($('#someField').val()));
```

Имейте в виду, что эта функция не проверяет, является ли передаваемый ей параметр строковым значением, поэтому если передать этой функции значение любого другого типа, есть шанс получить неопределенный и неудачный результат (скорее всего – ошибку JavaScript).

Синтаксис функции $.trim

`$.trim(value)`

Удаляет все начальные или концевые пробельные символы из строки `value` и возвращает результат.

Пробельными символами эта функция считает обычные символы, соответствующие регулярному выражению JavaScript `\s`, то есть не только символ пробела, но и такие символы, как перевод страницы, новая строка, возврат каретки, табуляция и вертикальная табуляция, а также символы Unicode `\u00A0`, `\u2028` и `\u2029`.

Параметры

`value` (строка) Строковое значение для усечения. Это исходное значение не изменяется.

Возвращаемое значение

Усеченная строка.

Теперь рассмотрим функции, которые работают с массивами и другими объектами.

6.3.2. Итерации по свойствам и элементам коллекций

Часто при формировании нескалярных значений, состоящих из других компонентов, нам требуется выполнить обход этих элементов. Является ли контейнерный элемент 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 выполняется с помощью цикла `for-in`.

Приведем примеры для всех случаев:

```
var anArray = ['one','two','three'];
for (var n = 0; n < anArray.length; n++) {
  // Какие-то действия
}

var anObject = {one:1, two:2, three:3};
for (var p in anObject) {
  // Какие-то действия
}
```

Довольно просто, но кому-то этот синтаксис может показаться слишком многословным и сложным – циклы `for` часто критикуют за это. Мы знаем, что для обернутого набора элементов DOM jQuery предоставляет команду `each()`, которая позволяет легко обойти все элементы в наборе без необходимости использовать сложный синтаксис оператора `for`. Для обычных массивов и объектов jQuery предоставляет аналогичную вспомогательную функцию с именем `$.each()`.

Синтаксис итераций по элементам массивов или свойствам объектов не меняется.

Синтаксис функции $.each

`$.each(container,callback)`

Выполняет цикл по элементам переданного контейнера, вызывая функцию `callback` для каждого элемента.

Параметры

`container` (массив | объект) Массив или объект, по элементам или свойствам которого будут выполняться итерации.

`callback` (функция) Функция, которая будет вызываться для каждого элемента в контейнере. Если контейнер является массивом, то функция вызывается для каждого элемента массива, а если объектом, то для каждого свойства объекта. Первый параметр этой функции – индекс элемента массива или имя свойства объекта. Второй параметр – значение элемента массива или свойства объекта. Контекст функции (`this`) содержит значение, которое передается во втором параметре.

Возвращаемое значение

Объект-контейнер.

Этот унифицированный синтаксис подходит как для массивов, так и для объектов, с тем же форматом вызова функции. Перепишем предыдущий пример с применением этой функции:

```
var anArray = ['one','two','three'];
$.each(anArray,function(n,value) {
  // Какие-то действия
});

var anObject = {one:1, two:2, three:3};
$.each(anObject,function(name,value) {
  // Какие-то действия
});
```

Использование `$.each()` со встроенной функцией может показаться попыткой свалить все в одну кучу, тем не менее эта функция позволяет с легкостью написать отдельную функцию-итератор многократного использования или вынести ее за скобки тела цикла в другую функцию, чтобы сделать код более ясным, как в следующем примере:

```
$.each(anArray,someComplexFunction);
```

Примечательно, что при выполнении итераций в массивах можно выйти из цикла, возвратив из функции-итератора значение `false`. При итерациях по свойствам объектов этот прием не действует.

Иногда обход массива нужен, чтобы выбрать элементы, которые составят новый массив. Давайте посмотрим, насколько это просто с jQuery.

6.3.3. Фильтрация массивов

Приложениям, работающим с большими количествами данных, часто требуется обход массивов для того, чтобы найти элементы, соответствующие определенным критериям. Мы можем, к примеру, отфильтровать элементы, значения которых находятся выше или ниже определенного порога или, возможно, соответствуют определенному шаблону. Для выполнения любой подобной фильтрации jQuery предоставляет вспомогательную функцию `$.grep()`.

Имя функции `$.grep()` может навести на мысль, что она использует те же регулярные выражения, что и ее тезка – команда `grep` UNIX. Но критерий фильтрации, который использует вспомогательная функция `$.grep()`, не является регулярным выражением, фильтрацию выполняет функция обратного вызова, предоставляемая вызывающей программой, которая определяет критерии для выявления значений,

Синтаксис функции $.grep

`$.grep(array, callback, invert)`

Выполняет цикл по элементам переданного массива, вызывая функцию `callback` для каждого элемента. Возвращаемое значение функции `callback` определяет, входит ли это значение в новый массив, который возвращается как значение функции `$.grep()`. Если параметр `invert` опущен или имеет значение `false`, и возвращаемое значение функции `callback` равно `true`, то данные включаются. Если параметр `invert` имеет значение `true`, и возвращаемое значение функции `callback` равно `false`, данные включаются.

Исходный массив не изменяется.

Параметры

`array` (массив) Массив, данные которого проверяются на возможность включения в новый массив. Этот массив при выполнении функции не изменяется.

`callback` (функция | строка) Функция, возвращаемое значение которой определяет, должно ли текущее значение входить в новый массив. Возвращаемое значение `true` приводит к включению данных при условии, что значение параметра `invert` не равно `true`, в противном случае результат меняется на противоположный.

Этой функции передаются два параметра: текущее значение и индекс этого значения в исходном массиве.

Входным параметром также может быть строка, в этом случае она преобразуется в функцию обратного вызова. Более подробно об этом рассказывается ниже.

`invert` (логическое значение) Если определено как `true`, то инвертирует нормальное действие функции.

Возвращаемое значение

Массив отобранных значений.

включаемых или исключаемых из полученного набора значений. Ничто не мешает этой функции *использовать* регулярные выражения для выполнения своей задачи, но автоматически это не делается.

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, которую мы передаем в `$.grep()`, может выполнять любые определенные нами действия, чтобы выяснить, какие значения должны быть включены. Решение может быть легким, как в этом примере, или сложным, как создание синхронных обращений Ajax (с неизбежной потерей производительности) к серверу, чтобы определить, должно ли значение быть включено или исключено.

Для таких же простых решений, как в этом примере, jQuery предлагает использовать более краткую форму записи, чтобы инструкция была более компактной, – предоставить выражение в виде строки. Например, мы можем переписать наш фрагмент кода так:

```
var bigNumbers = $.grep(originalArray,'a>100');
```

Если функция обратного вызова определена в виде строки, jQuery автоматически создаст эту функцию с указанной строкой в качестве значения инструкции `return` и передаст ей два параметра: `a` – текущее значение и `i` – текущий индекс. Таким образом, в этом примере будет сгенерирована функция, эквивалентная следующей:

```
function(a,i){return a>100;}
```

Хотя функция `$.grep()` и не использует регулярные выражения непосредственно (несмотря 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: в массиве значений требуется идентифицировать некоторые значения, не соответствующие шаблону почтового индекса Соединенных Штатов.

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})?$/`, которое мы можем применить для фильтрации исходного массива и извлечь ошибочные записи:

```
var badZips = $.grep(
              originalArray,
              function(value) {
                return value.match(/^\d{5}(-\d{4})?$/) != null;
              },
              true);
```

В этом примере метод match() класса String позволяет определить, соответствует ли значение шаблону; в параметре invert функции $.grep() передается значение true, чтобы *исключить* все значения, соответствующие шаблону.

Выборка подмножеств данных из массивов – не единственное, что мы можем проделать с ними. Рассмотрим еще одну функцию для работы с массивами, которую предоставляет jQuery.

6.3.4. 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 массив значений индексов, отсчет которых начинается с нуля, в соответствующий массив индексов, отсчет которых начинается с единицы.

Как и в случае с функцией $.grep(), для таких простых примеров мы можем передать строку, которая представляет собой выражение, что

Синтаксис функции $.map

$.map(array, callback)

Выполняет цикл по элементам переданного массива, вызывая функцию обратного вызова для каждого элемента массива и собирая возвращаемые значения функции в новый массив.

Параметры

array	(массив) Массив, значения элементов которого будут преобразованы в значения элементов нового массива.
callback	(функция | строка) 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. Строка также может быть входным параметром, в этом случае она преобразуется в функцию обратного вызова. Более подробно об этом рассказывается ниже.

Возвращаемое значение

Обернутый набор.

позволит сделать инструкцию более компактной. В этом случае будет сгенерирована функция, которой будет передано значение для преобразования в виде параметра с именем a (в отличие от $.grep(), здесь индекс не передается в функцию, сгенерированную автоматически). Можно переписать наш пример так:

```
var oneBased = $.map([0,1,2,3,4],'a+1');
```

Еще один важный момент, который надо принять к сведению: если функция возвращает значение null или undefined, результат не включается в новый массив. В таких случаях в полученном массиве будет меньше элементов, чем в исходном (то есть у этих массивов будет разная *длина*), и однозначное соответствие между ними утрачивается.

Рассмотрим чуть более сложный пример. Допустим, имеется массив строк, возможно, собранных из полей формы, предположительно являющихся числовыми значениями, и требуется преобразовать этот массив строк в соответствующий ему массив чисел. Поскольку нет никакой гарантии, что строки не содержат нечисловые значения, нам понадобится принять некоторые меры предосторожности. Рассмотрим следующий фрагмент:

```
var strings = ['1','2','3','4','S','6'];

var values = $.map(strings,function(value){
  var result = new Number(value);
  return isNaN(result) ? null : result;
});
```

Изначально имеется массив строк, каждая из которых, как ожидается, представляет числовое значение. Но опечатка (или, возможно, ошибка пользователя) привела к тому, что вместо ожидаемого символа *5* у нас имеется символ *S*. Наш программный код обрабатывает этот случай, проверяя экземпляр Number, созданный конструктором, чтобы увидеть, было ли преобразование из строки в число успешным или нет. Если преобразование не удалось, возвращаемое значение будет константой Number.NaN. Но интереснее всего то, что по определению значение Number.NaN не равно никакому другому значению, даже самому себе! Поэтому логическое выражение Number.NaN == Number.NaN будет равно false!

Поскольку мы не можем использовать оператор сравнения для проверки на равенство NaN (что, кстати, означает *Not a Number* – не число), JavaScript предоставляет метод isNaN(), который и позволит нам проверить результат преобразования строки в число.

В этом примере в случае неудачи мы возвращаем null, гарантируя, что полученный в результате массив содержит только действительно числовые значения, а все ошибочные значения игнорируются. Если требуется собрать все значения, мы можем 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 функции преобразования возвра-

щает *массив*, объединяя возвращаемые значения в массив результата. Рассмотрим следующий пример:

```
var characters = $.map(
  ['this','that','other thing'],
  function(value){return value.split('');}
);
```

Эта инструкция преобразует массив строк в массив символов, из которых составлены строки. После выполнения инструкции значения переменной `characters` таковы:

```
['t','h','i','s','t','h','a','t','o','t','h','e','r',' ','t','h',
➥'i','n','g']
```

Этот результат получен при помощи метода `String.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 могут оказаться полезными для вас.

6.3.5. Другие полезные функции для работы с массивами JavaScript

Бывало ли так, что вам требовалось 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()`.

Несложный, но показательный пример применения этой функции:

```
var index = $.inArray(2,[1,2,3,4,5]);
```

В результате в переменную `index` записывается значение **1**.

Другая полезная функция для работы с массивами создает массив JavaScript из других объектов, подобных массиву.

Синтаксис функции $.inArray

`$.inArray(value,array)`

Возвращает индекс первого вхождения значения `value` в массив `array`.

Параметры

`value` (объект) Значение, которое требуется найти в массиве.

`array` (массив) Массив, в котором будет производиться поиск.

Возвращаемое значение

Индекс первого вхождения искомого значения в массиве или `-1`, если значение не найдено.

Вам интересно, что такое *объекты, подобные массиву*, и с чем их едят? jQuery считает, что *объектом, подобным массиву*, может быть любой объект, который имеет длину и поддерживает индексы. Эта возможность очень полезна для объектов NodeList. Рассмотрим следующий фрагмент:

```
var images = document.getElementsByTagName("img");
```

Здесь переменная images заполняется объектами NodeList, представляющими все изображения на странице.

Работать с NodeList достаточно сложно, поэтому преобразование его в массив JavaScript может существенно упростить работу. Функция $.makeArray() из библиотеки jQuery облегчает преобразование объектов NodeList.

Синтаксис функции $.makeArray

$.makeArray(object)

Преобразует объект, подобный массиву, в массив JavaScript.

Параметры

object (объект) Объект, подобный массиву (например NodeList), для преобразования.

Возвращаемое значение

Полученный в результате массив JavaScript.

Эта функция предназначена для использования в программном коде, почти не обращающегося к библиотеке jQuery, внутренние механизмы которой могут выполнять подобные преобразования незаметно для нас. Эта функция также полезна, когда приходится иметь дело с объектами NodeList при навигации по XML-документу без применения jQuery.

Другая редко используемая функция, которая может быть удобной при работе с массивами, созданными не средствами jQuery, – функция $.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.

Возвращаемое значение

Массив элементов DOM, состоящий из уникальных элементов массива, переданного функции.

Эта функция также используется внутренними механизмами jQuery, чтобы исключать неуникальные элементы из списков, с которыми мы работаем, и предназначена для работы с массивами элементов, созданных за пределами jQuery.

Увидев, как jQuery упрощает работу с массивами, рассмотрим способ, который поможет нам управлять простыми объектами JavaScript.

6.3.6. Расширение объектов

Несмотря на то что часть функций JavaScript реализованы, как в объектно-ориентированном языке, мы не можем считать JavaScript полноценным объектно-ориентированным языком из-за отсутствия поддержки некоторых возможностей. Одна из таких важных возможностей – *наследование*, когда новый класс определяется как расширение существующего класса.

Наследование в JavaScript можно имитировать, копируя свойства базового объекта в новый объект и затем расширяя новый объект с сохранением свойств базового объекта.

Примечание

Если вы поклонник объектно-ориентированного JavaScript, то, разумеется, знакомы с возможностью расширения не только экземпляров объектов, но и их механизмов копирования через свойство `prototype` конструктора объекта. Функция `$.extend()` позволяет организовать как наследование на основе конструктора, с расширением `prototype`, так и наследование на основе объекта, с расширением существующего экземпляра объекта. Для эффективного использования jQuery необязательно понимать такую сложную тему, поэтому в данной книге она не рассматривается, несмотря на всю ее важность.

Написать программный код, который выполнял бы такое расширение простым копированием, несложно, но jQuery уже имеет все необходимое для этого и предоставляет вам готовую вспомогательную функцию. Как мы увидим в следующей главе, эту функцию можно применять не только для расширения объектов, но даже в этом случае она оправдывает свое имя `$.extend()`. Ее синтаксис представлен на стр. 201.

Посмотрим на эту функцию в деле. Рассмотрим код листинга 6.4 (файл *chapter6/$.extend.html*).

Листинг 6.4. Тестирование функции $.extend

```
<html>
  <head>
    <title>$.extend Example</title>
    <link rel="stylesheet" type="text/css" href="../common.css">
    <script type="text/javascript"
            src="../scripts/jquery-1.2.1.js"></script>
    <script type="text/javascript"
            src="../scripts/support.labs.js"></script>
```

Синтаксис функции $.extend

`$.extend(target, source1, source2,...sourceN)`

Расширяет объект, полученный в параметре `target`, свойствами остальных объектов, передаваемых функции в виде дополнительных параметров.

Параметры

`target`	(объект) Объект, свойства которого дополняются свойствами объектов-источников. Этот объект полностью модифицируется новыми свойствами, прежде чем будет возвращен функцией. Все свойства, одноименные свойствам любого объекта-источника, переопределяются значениями свойств объектов-источников.
`source1 ... sourceN`	(объект) Один или больше объектов, свойства которых добавляются к объекту `target`. Если объектов-источников больше одного и у них есть одноименные свойства, объекты в конце списка имеют более высокий приоритет по сравнению с теми, что находятся в начале списка.

Возвращаемое значение

Расширенный объект `target`.

```
<script type="text/javascript">
  var target = { a: 1, b: 2, c: 3 };
  var source1 = { c: 4, d: 5, e: 6 };
  var source2 = { e: 7, f: 8, g: 9 };

  $(function(){
    $('#targetBeforeDisplay').html($.toSource(target));
    $('#source1Display').html($.toSource(source1));
    $('#source2Display').html($.toSource(source2));
    $.extend(target,source1,source2);
    $('#targetAfterDisplay').html($.toSource(target));
  });
</script>
<style type="text/css">
  label { float: left; width: 108px; text-align: right; }
  p { clear: both; }
  label + span { margin-left: 6px; }
</style>
</head>

<body>
  <fieldset>
    <legend>$.extend() Example</legend>
    <p>
      <label>target (before) =</label>
      <span id="targetBeforeDisplay"></span>
```

1 Определение объектов тестирования

2 Вывод начального состояния объектов

3 Расширение объекта

4 Вывод конечного состояния объекта

5 Определение элементов HTML для вывода информации

```
        </p>
        <p>
          <label>source1 =</label>
          <span id="source1Display"></span>
        </p>
        <p>
          <label>source2 =</label>
          <span id="source2Display"></span>
        </p>
        <p>
          <label>target (after) =</label>
          <span id="targetAfterDisplay"></span>
        </p>
      </fieldset>
    </body>
  </html>
```

В этом простом примере мы определили три объекта: целевой объект target и два объекта-источника ❶. Эти объекты позволят нам продемонстрировать, что функция $.extend() делает с целевым объектом для слияния этого объекта с двумя объектами-источниками.

После объявления объектов определяется обработчик события готовности документа, в котором будут выполняться операции над ними. Хотя объекты доступны непосредственно в программном коде, мы выводим результаты на страницу, поэтому нам придется подождать, пока HTML-элементы ❺ отобразятся в окне броузера.

Внутри обработчика события готовности документа мы выводим информацию о состоянии трех объектов в элементах , предназначенных для вывода результатов ❷. (Если вам интересно, как работает функция $.toSource(), ее определение можно найти в файле *support.labs.js*. О том, как добавить такие вспомогательные функции в свой арсенал, мы расскажем в следующей главе.)

Мы расширяем объект двумя объектами-источниками ❸ с помощью следующей инструкции:

```
$.extend(target,source1,source2);
```

Эта инструкция объединяет свойства объектов source1 и source2 в объекте target. Объект target возвращается как значение функции, и поскольку он изменяется на месте, мы можем не создавать переменную для хранения ссылки на него. Тот факт, что функция возвращает объект target, очень важен для построения цепочек инструкций с этой функцией.

Затем мы отображаем значения свойств модифицированного объекта target ❹. Результаты показаны на рис. 6.5.

Как видите, все свойства объектов-источников объединены в объекте target. Но следует отметить следующие важные нюансы:

- И target, и source1 содержат свойство с именем c. Свойство c объекта source1 замещает одноименное свойство у объекта target.

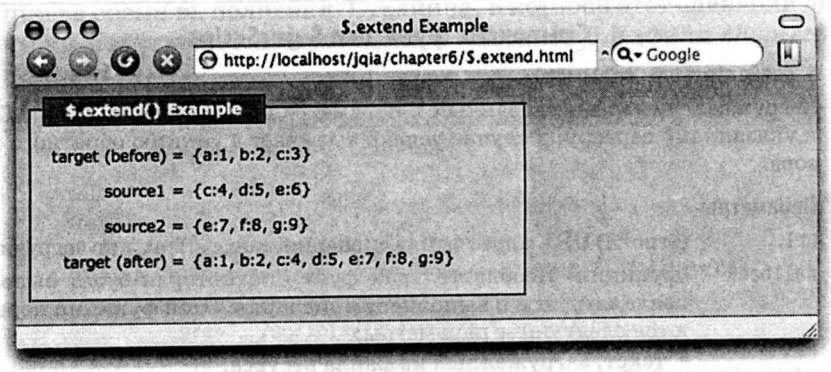


Рис. 6.5. Объединение объектов с помощью функции $.extend – в результате все свойства объектов-источников копируются в объект target

- И `source1`, и `source2` содержат свойство с именем `e`. Обратите внимание: свойство `e` объекта `source2` имеет преимущество перед одноименным свойством объекта `source1` при объединении их в объекте `target`, что наглядно демонстрирует приоритет объектов из конца списка аргументов перед теми, что стоят ближе к началу списка.

Вполне очевидно, что эта вспомогательная функция может быть полезна во многих случаях, когда один объект нужно расширить свойствами другого объекта (или набора объектов). Мы увидим примеры использования этой функции при изучении порядка определения самих вспомогательных функций в следующей главе.

Но прежде чем перейти к этому, завершим изучение вспомогательных функций еще одной, позволяющей динамически загружать новые сценарии в наши страницы.

6.4. Динамическая загрузка сценариев

В большинстве случаев – пожалуй, всегда, – внешние сценарии, необходимые для нашей страницы, загружаются из файлов сценариев с помощью тегов `<script>` в области заголовка (`<head>`) страницы. Но время от времени может понадобиться загрузить какой-то сценарий и во время выполнения другого сценария.

Например, так может получаться потому, что заранее, пока пользователь не выполнил некоторые действия, мы можем не знать, нужен ли этот сценарий, и хотим включать его только при абсолютной необходимости. Может быть и так, что для выбора из нескольких сценариев нам требуется информация, отсутствующая на этапе загрузки.

jQuery предоставляет вспомогательную функцию `$.getScript()`, позволяющую загрузить новый сценарий во время выполнения вне зависимости от причины, по которой нам это понадобилось.

Синтаксис функции $.getScript

`$.getScript(url,callback)`

Загружает сценарий, указанный в параметре `url`, выполняя запрос GET к указанному серверу; в случае успеха вызывает функцию обратного вызова.

Параметры

`url`	(строка) URL-адрес файла сценария, который нужно загрузить.
`callback`	(функция) Необязательная функция, которая будет вызвана после загрузки и выполнения сценария. Этой функции передаются следующие параметры: • текст, загруженный из файла ресурса; • строка *success*.

Возвращаемое значение

Экземпляр XHR, используемый для загрузки сценария.

В соответствии со своим именем, эта функция для загрузки файла сценария использует встроенные в jQuery механизмы Ajax. В главе 8 мы рассмотрим эти возможности механизмов Ajax достаточно подробно, но чтобы использовать эту функцию здесь, знание Ajax нам не требуется.

После загрузки сценарий из файла обрабатывается – выполняются все встроенные сценарии и становятся доступными все объявленные переменные и функции.

Предупреждение

В Safari определения из загруженного сценария не становятся доступными сразу же, они недоступны даже в функции обратного вызова. Элементы динамически загруженного сценария станут доступными лишь после завершения блока сценария, внутри которого выполняется загрузка, то есть после того, как управление будет передано обратно броузеру. Если ваши страницы должны поддерживать Safari, учитывайте это обстоятельство!

Поглядим на эту функцию в действии. Рассмотрим следующий файл сценария (файл *chapter6/new.stuff.js*):

```
alert("I'm inline!");

var someVariable = 'Value of someVariable';

function someFunction(value) {
  alert(value);
}
```

Этот простой сценарий содержит встроенную инструкцию (вывод предупреждения, подтверждающего, что выполняется именно эта инструкция), объявление переменной и объявление функции для вывода сообщения, содержащего все значения, которые были переданы функ-

ции во время выполнения. Страница, в которой этот сценарий динамически подключается, приведена в листинге 6.5 (файл *chapter6/$.getScript.html*).

Листинг 6.5. Динамическая загрузка файла сценария и проверка результатов

```
<html>
  <head>
    <title>$.getScript Example</title>
    <link rel="stylesheet" type="text/css" href="../common.css">
    <script type="text/javascript"
            src="../scripts/jquery-1.2.1.js"></script>
    <script type="text/javascript">
      $(function(){
        $('#loadButton').click(function(){                    ❶ Загружает сценарий по щелчку на кнопке Load
          $.getScript(
            'new.stuff.js'//,function(){$('#inspectButton').click()}
          );
        });
        $('#inspectButton').click(function(){                 ❷ Выводит результаты по щелчку на кнопке Inspect
          someFunction(someVariable);
        });
      });
    </script>
  </head>
  <body>                                                      ❸ Содержит определения кнопок
    <button type="button" id="loadButton">Load</button>
    <button type="button" id="inspectButton">Inspect</button>
  </body>
</html>
```

На этой странице определены две кнопки ❸, предназначенные для активизации действий в примере. По нажатию кнопки Load (загрузить) с помощью функции `$.getScript()` ❶ динамически загружается файл *new.stuff.js*. Обратите внимание: изначально второй параметр (функция обратного вызова) закомментирован – вскоре мы выясним, почему.

Щелчок на этой кнопке вызывает загрузку и выполнение файла *new.stuff.js*. Как и следовало ожидать, встроенная в файл инструкция выполняется и выводит предупредительное сообщение (рис. 6.6).

В результате щелчка на кнопке Inspect (проверить) вызывается ее обработчик события `click` ❷, который выполняет динамически загруженную функцию `someFunction()`, передавая ей значение динамически загруженной переменной `someVariable`. Если на экране появляется сообщение (рис. 6.7), это свидетельствует о том, что и переменная, и функция загружены без ошибок.

Если хотите понаблюдать за поведением Safari, о котором мы предупредили выше, сделайте копию HTML-файла листинга 6.5 и раскомментируйте второй параметр в вызове функции `$.getScript()`. Функция

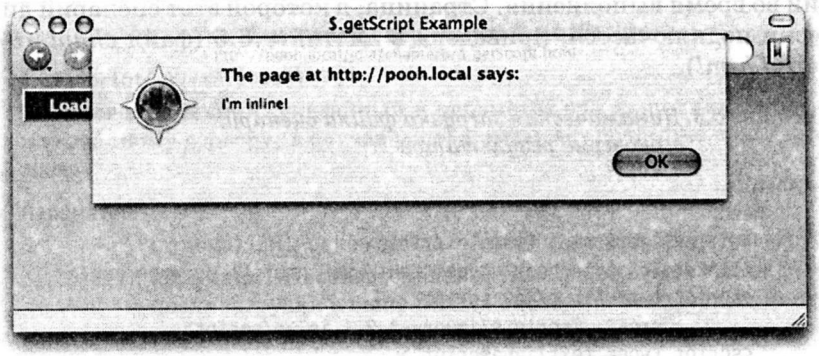


Рис. 6.6. Динамическая загрузка и выполнение сценария приводят к выполнению встроенной инструкции, которая выводит предупреждение

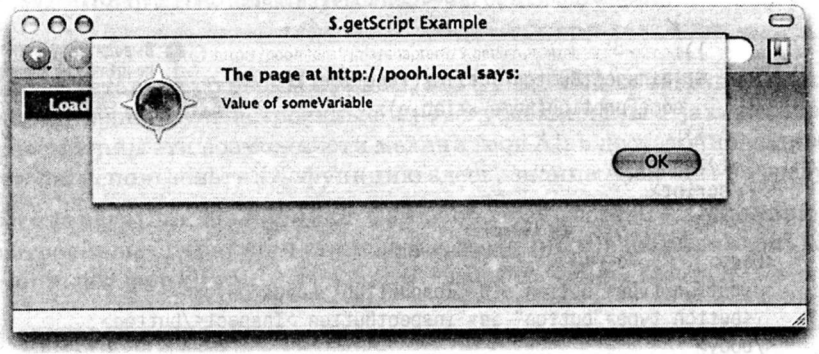


Рис. 6.7. Появление сообщения показывает, что функция динамически загружена правильно, а отображенное правильное значение показывает, что была динамически загружена и переменная

обратного вызова в этом параметре вызывает обработчик события `click` для кнопки Inspect, который в свою очередь вызывает динамически загруженную функцию с загруженной переменной в виде параметра.

В других броузерах, отличных от Safari, функция и переменная, загруженные динамически, становятся доступными уже внутри функции обратного вызова. Но когда она выполняется в Safari, ничего не происходит! Применяя функцию `$.getScript()`, следует учитывать это расхождение в функциональных возможностях.

6.5. Итоги

В этой главе мы рассмотрели функциональные возможности, которые библиотека jQuery предоставляет помимо методов, предназначенных для выполнения операций над обернутыми наборами элементов DOM.

Это разнообразные функции и ряд флагов, определенных непосредственно на уровне идентификатора `jQuery` (и его псевдонима $).

Если понадобилось определить тип броузера, чтобы учесть различие функциональных возможностей разных броузеров, то выяснить семейство, к которому принадлежит текущий броузер, можно с помощью набора флагов `$.browser`. К определению типа броузера следует прибегать только в крайнем случае – если реализовать независимый от типа броузера программный код невозможно, а более предпочтительный метод обнаружения объекта неприменим.

Флаг `$.boxModel` сообщает, какая из двух блочных моделей используется для отображения страницы, а флаг `$.styleFloat` информирует об имени свойства стиля `float`, что позволяет писать программный код, независимый от типа броузера.

Учитывая, что иногда авторы страниц наряду с jQuery могут обращаться к другим библиотекам, библиотека jQuery предоставляет вспомогательную функцию `$.noConflict()`, позволяющую другим библиотекам использовать свое определение псевдонима $. После вызова этой функции все операции jQuery должны вместо псевдонима $ использовать идентификатор `jQuery`.

В библиотеке есть функция `$.trim()`, которая восполняет отсутствие этой функции в классе `String` языка JavaScript и отсекает лишние пробелы в начале и в конце строковых значений.

Кроме того, jQuery предоставляет набор функций для работы с наборами данных в виде массивов. Функция `$.each()` упрощает обход всех элементов в массиве. Функция `$.grep()` позволяет создавать новые массивы, фильтруя данные исходного массива по какому-либо критерию. Функция `$.map()` позволяет легко преобразовать исходный массив, создавая соответствующий ему новый массив с преобразованными значениями.

Для объединения объектов, возможно даже, для имитации своего рода механизма наследования, jQuery предоставляет функцию `$.extend()`. Эта функция позволяет объединять произвольное число свойств исходных объектов в указанном целевом объекте.

Для случая, когда нам требуется динамически загрузить файл сценария, jQuery определяет функцию `$.getScript()`, способную загрузить и выполнить сценарий в любой точке выполнения другого сценария страницы.

Теперь, вооруженные дополнительными функциональными возможностями, попробуем добавить к jQuery собственные расширения. Это мы сделаем в следующей главе.

7

В этой главе:
- Зачем расширять jQuery собственным программным кодом
- Правила эффективного расширения возможностей jQuery
- Как писать собственные вспомогательные функции
- Как писать собственные методы обертки

Расширение jQuery с помощью собственных модулей

В предыдущих главах мы показали, что jQuery предлагает множество полезных команд и функций, комбинируя которые можно задавать поведение своих страниц. Иногда программный код реализует общий алгоритм, который бы хотелось применять неоднократно. Если появляются такие «шаблонные» операции, есть смысл оформить их в виде инструментов многократного использования и добавить в свой арсенал. В этой главе мы узнаем, как оформлять такие повторяющиеся фрагменты программного кода в виде расширений для jQuery.

Но давайте сначала посмотрим, *зачем* может потребоваться расширять jQuery собственным программным кодом.

7.1. Зачем нужны расширения?

Если вы прилежно читаете эту книгу, внимательно просматривая представленные в ней примеры программного кода, то наверняка заметили, что применение jQuery сильно влияет на оформление кода сценариев в страницах.

Применение библиотеки jQuery требует определенного стиля оформления исходного кода страницы, часто в виде набора обернутых элементов, к которым применяется команда или цепочка команд jQuery. В собственном программном коде можно придерживаться любого стиля по своему усмотрению, но наиболее опытные разработчики считают работу хорошей, только когда весь программный код сайта, или хотя бы его основная часть, выполняется в едином стиле.

Таким образом, выделить повторяющиеся участки программного кода и оформить их в виде расширений для jQuery – значит поддерживать единство стиля оформления программного кода на всем сайте.

Этого мало? Нужны еще обоснования? В общем и целом jQuery предоставляет нам набор инструментов многократного использования и прикладной программный интерфейс. Создатели jQuery тщательно подошли к планированию архитектуры библиотеки и философии внутреннего устройства инструментов, чтобы поддержать возможность многократного использования. Следуя архитектуре этих инструментов, мы автоматически получаем заложенные в ней преимущества – вот вторая веская причина для создания собственных расширений.

И это еще не убедило? Последняя причина, которую мы приведем (хотя можем вспомнить еще много), заключается в том, что, расширяя возможности jQuery, мы надстраиваем уже имеющийся программный код, который jQuery сделала доступным для нас. Например, создавая новые команды (методы обертки), мы автоматически наследуем богатые функциональные возможности механизма селекторов jQuery. Зачем писать все с нуля, если можно опираться на мощные инструменты библиотеки jQuery?

Из этих причин прямо следует, что создание собственных программных компонентов многократного использования в виде расширений для jQuery – это положительный опыт и грамотный подход к работе. На протяжении этой главы мы исследуем основные правила и шаблоны проектирования, позволяющие создавать *модули расширения*, или *подключаемые модули* (*plugins*) для jQuery, и на их основе создадим несколько собственных расширений. Следующая глава, которая охватывает совершенно другую тему (Ajax), добавит другие доказательства в пользу создания собственных компонентов многократного использования в виде модулей расширения для jQuery. Мы увидим, как модули расширения помогают сохранить единство стиля оформления программного кода и насколько просто создаются эти компоненты при таком подходе.

Начнем с правил.

7.2. Основные правила создания модулей расширения jQuery

Sign! Sign! Everywhere a sign! Blocking out the scenery, breaking my mind. Do this! Don't do that! Can't you read the sign?

(Правила! Правила! Всюду правила! Заслоняют пейзаж, сводят меня с ума! Делайте это! Не делайте то! Вы что, не знаете правил?)

«Five Man Electric Band» (1971)

В далеком 1971 году группа «Five Man Electric Band» выразила в своей песне протест против существовавших тогда социальных и этических

норм. Тем не менее иногда без правил не обойтись. Без правил мир погрузился бы в хаос.

Это справедливо и для правил (скорее, общих принципов), обеспечивающих успешное расширение jQuery собственными модулями. Эти правила не только помогут включить свой новый программный код в архитектуру jQuery, но и гарантируют, что он прекрасно будет работать с другими расширениями jQuery и даже с другими библиотеками JavaScript.

Расширение для jQuery может принимать одну из двух форм:

- вспомогательные функции, определяемые непосредственно в $ (псевдоним идентификатора `jQuery`);
- методы, оперирующие обернутым набором jQuery (так называемые *командами* jQuery).

В оставшейся части этого раздела мы рассмотрим некоторые правила создания расширений, общие для обоих типов. В последующих разделах мы займемся правилами и приемами, относящимися к определенным типам элементов расширений.

7.2.1. Именование функций и файлов

В 1950-х в США появилось телешоу «To Tell the Trust» («По правде говоря»), в котором сразу несколько человек выдавали себя за одного и того же человека, а звездное жюри должно было определить, кто из них на самом деле тот, за кого себя выдает. Это забава для телевизионной аудитории, но если *конфликт имен* случился в программе, то уже не до веселья.

Мы еще обсудим, как избежать конфликта имен *внутри* модулей, но в первую очередь нужно внимательно следить, чтобы имена файлов, в которые записываются наши модули, не конфликтовали с именами других файлов.

Согласно простым, но эффективным рекомендациям разработчиков jQuery, имя файла должно:

- начинаться с префикса *jquery.*;
- содержать имя модуля, идущее после префикса;
- завершаться расширением имени *.js*.

Если мы пишем модуль с именем Fred, то файл с программным кодом JavaScript для этого модуля будет называться так:

```
jquery.fred.js
```

Наличие префикса *jquery.* предотвратит возможный конфликт с именами файлов, предназначенных для использования с другими библиотеками. В конце концов, тому, кто пишет модули не для библиотеки jQuery, использовать префикс *jquery.* нет смысла.

Но при этом все еще возможен конфликт имен модулей расширения *внутри* сообщества jQuery.

Если мы пишем модули расширения для собственного пользования, то нам достаточно избежать конфликта имен только с теми модулями, с которыми мы планируем работать сами. Но если модуль создается для всех желающих, то необходимо избежать конфликта имен со всеми уже выпущенными модулями.

Лучший способ избежать конфликта имен – постоянно быть в курсе дел сообщества jQuery. Замечательная отправная точка для этого – страница *http://docs.jquery.com/Plugins*, но помимо информации на этой странице есть и другие меры предосторожности, которыми мы можем воспользоваться.

Один из способов обеспечить отсутствие конфликта имен файлов между нашим и другими модулями – использовать дополнительный префикс, уникальный для вас или вашей организации. Например, имена файлов всех модулей расширения, разработанных для этой книги, содержат префикс *jquery.jqia* (где *jqia* является сокращением от *jQuery in Action* – «jQuery в действии»), чтобы исключить вероятность конфликта с именами файлов любых других модулей расширения, которые кто-нибудь может использовать в своих веб-приложениях. Точно так же имена файлов для jQuery Form Plugin начинаются с префикса *jquery.form*. Не все модули следуют этому соглашению, но по мере роста их количества будет постоянно расти и важность следования ему.

Похожий принцип можно применить и к *именам*, которые мы даем своим новым функциям, – как вспомогательным функциям, так и методам в обертке jQuery.

Создавая модули расширения «для себя», мы обычно знаем о других модулях, которые будем использовать, – избежать конфликта имен в этом случае достаточно легко. Но как быть, если модуль создается для общего использования? Или как быть, если наши модули первоначально предназначались для личного пользования, но оказались настолько эффективными, что мы решили передать их всему сообществу?

Напомним еще раз, что ознакомление с уже существующими модулями для предотвращения конфликта имен в прикладных интерфейсах займет достаточно много времени. Кроме того, рекомендуем собирать коллекции взаимосвязанных функций под общим префиксом (точно так же, как и в случае с именами файлов), чтобы избежать беспорядка в пространстве имен.

7.2.2. Остерегайтесь $

«Удастся ли нам отстоять $?»

Написав довольно много программного кода для jQuery, мы убедились, насколько удобно использовать псевдоним $ вместо `jQuery`. Но разрабатывая модуль для других, нельзя быть бесцеремонным. Автор

модуля расширения не может заранее знать, будет ли автор страницы использовать функцию `$.noConflict()`, чтобы позволить другой библиотеке использовать псевдоним $.

Мы могли бы действовать «в лоб», применяя имя `jQuery` вместо псевдонима $. Но, черт возьми, нам *нравится* использовать $ и так легко мы с ним не расстанемся.

В разделе 6.2 обсуждалась идиома, часто используемая для того, чтобы обеспечить локальное соответствие псевдонима $ имени `jQuery`, не влияя на остальную часть страницы. Эту маленькую хитрость можно применить и при определении модулей расширения jQuery (что часто и делается):

```
(function($){
//
// Здесь располагается определение модуля расширения
//
})(jQuery);
```

Передавая имя `jQuery` функции, которая определяет параметр $, мы тем самым гарантируем, что в теле функции имя $ будет ссылаться на имя `jQuery`.

Теперь мы можем без особых хлопот использовать $ в тексте определения модуля расширения.

Прежде чем погрузиться в изучение добавления новых элементов в jQuery, рассмотрим еще один прием, рекомендуемый авторам модулей расширения.

7.2.3. Укрощение сложных списков параметров

Большинство модулей расширения требуют (если вообще требуют) небольшого числа параметров, стремясь к простоте. Мы уже видели достаточно доказательств тому – подавляющее большинство базовых методов и функций библиотеки jQuery либо требуют небольшого числа параметров, либо вообще в них не нуждаются. Если необязательные параметры опущены, для них всегда предусмотрены достаточно разумные значения по умолчанию. Кроме того, когда опускаются некоторые необязательные параметры, можно даже изменять порядок следования остальных параметров.

Отличный пример – метод `bind()`: если его необязательный параметр `data` опущен, ссылка на функцию-обработчик, обычно передаваемая в третьем параметре, может быть передана во втором параметре. Благодаря динамической и интерпретирующей природе JavaScript мы можем писать такой гибкий программный код, но с ростом числа параметров реализация подобных возможностей скоро может невероятно усложниться (как для авторов страниц, так и для авторов модулей расширения). Дело осложняется еще больше, если необязательных параметров много.

Рассмотрим некоторую сложную функцию, сигнатура которой выглядит так:

```
function complex(p1,p2,p3,p4,p5,p6,p7) {
```

Эта функция принимает семь аргументов. Допустим, что все они, кроме первого, являются необязательными. Здесь слишком много необязательных аргументов, чтобы делать какие-либо обоснованные предположения о намерениях вызывающего программного кода, если какие-то из необязательных параметров опущены (как это было с методом `bind()`). Если при вызове этой функции опущены последние параметры в списке, то проблема невелика, так как отсутствующие заключительные параметры будут определяться как пустые значения. Но что если вызывающему программному коду при обращении к функции потребовалось определить параметр `p7`, а для параметров с `p2` по `p6` позволить взять значения по умолчанию? Тогда при вызове функции для всех опущенных параметров пришлось бы использовать заполнители, и обращение к ней выглядело бы так:

```
complex(valueA,null,null,null,null,null,valueB);
```

Фу! Хуже всех из вариантов, допустимых природой JavaScript, смотрится такой вызов:

```
complex(valueA,null,valueC,valueD,null,null,valueB);
```

Обращаясь к этой функции, авторы страниц будут вынуждены внимательно следить за количеством пустых ссылок (null) и порядком следования параметров. Кроме того, такой программный код сложнее читается и воспринимается.

Но что здесь можно предпринять, кроме как запретить использовать много параметров?

И опять помогает гибкая природа JavaScript: среди сообществ создателей страниц появился шаблон, позволяющий укротить этот хаос, – *хеш параметров* (*options hash*).

Этот шаблон позволяет собрать необязательные параметры в *единый* параметр – экземпляр объекта `Object`, свойства которого, в виде пар имя/значение, играют роль необязательных параметров

Применив этот прием, мы можем записать наш первый пример так:

```
complex(valueA, {p7: valueB});
```

А второй так:

```
complex(valueA, {
  p3: valueC,
  p4: valueD,
  p7: valueB
});
```

Гораздо лучше!

Нам не требуется подставлять пустые ссылки (null) на место пропущенных параметров значения, нам также не требуется следить за количеством параметров – каждый необязательный параметр помечается, поэтому достаточно четко видно, что это за параметры (особенно когда вместо имен от p1 до p7 применяются более осмысленные имена).

Это огромное преимущество для тех, кто вызывает наши сложные функции, но что несет этот прием *нам* как авторам функций? Но ведь мы уже видели механизм jQuery, облегчающий сборку этих необязательных параметров и подстановку значений по умолчанию. Вернемся еще раз к нашей сложной функции с одним обязательным и шестью необязательными параметрами. Новая упрощенная сигнатура этой функции теперь выглядит так:

```
complex(p1,options)
```

Внутри этой функции мы можем подставить в необязательные параметры значения по умолчанию с помощью удобной вспомогательной функции `$.extend()`. Например:

```
function complex(p1,options) {
  var settings = $.extend({
    option1: defaultValue1,
    option2: defaultValue2,
    option3: defaultValue3,
    option4: defaultValue4,
    option5: defaultValue5,
    option6: defaultValue6
  },options||{});
  // Остальная часть функции
}
```

Объединив значения, полученные от автора страницы в параметре `options`, с объектом, содержащим все доступные параметры с их значениями по умолчанию, мы получаем переменную `settings` со всеми допустимыми параметрами по умолчанию, которые заменяются значениями, явно переданными функции автором страницы.

Обратите внимание на то, как мы защищаемся от случая, когда в параметре `options` передается пустая ссылка или неопределенное значение, с помощью конструкции `||{}`, создающей пустой объект, если выражение `options` возвращает результат `false` (которое, как мы знаем, дают значения `null` и `undefined`).

Легко, универсально и удобно для вызывающей программы!

Нам будут встречаться примеры использования этого шаблона ниже в этой главе и в функциях jQuery, которые будут рассматриваться в главе 8. А теперь давайте, наконец, перейдем к изучению расширения jQuery собственными вспомогательными функциями.

7.3. Создание собственных вспомогательных функций

В этой книге термином *вспомогательная функция* мы обозначаем функции, определяемые как свойства jQuery (и отсюда – $). Такие функции не работают с элементами DOM – этим занимаются методы, специально предназначенные для работы с обернутыми наборами элементов jQuery: они предназначены либо для работы с объектами JavaScript, которые не являются элементами, либо для других действий, не связанных с объектами. Примерами таких функций являются $.each() и $.noConflict(), с которыми мы уже встречались.

В этом разделе мы узнаем, как добавлять собственные подобные функции.

Добавить функцию в виде свойства экземпляра Object достаточно просто: сначала нужно объявить функцию, а затем присвоить ее свойству объекта. (Если для вас это звучит странно, и вы еще не прочли Приложение, то сейчас самое время сделать это.) Простейшая функция создается просто:

```
$.say = function(what) { alert('I say '+what); }
```

По правде говоря, это *действительно* легко. Но в таком подходе к созданию вспомогательных функций кроются свои опасности – помните обсуждение в разделе 7.2.2 касательно псевдонима $? Что если какой-нибудь автор страницы подключит эту функцию к странице, которая уже использует библиотеку Prototype и вызывает $.noConflict()? Вместо того чтобы добавить расширение jQuery, будет создан метод функции $() библиотеки Prototype. (Обратитесь к Приложению, если понятие *метода* функции для вас неясно.)

Для наших личных функций, которые мы не собираемся передавать кому-то еще, здесь нет никакой проблемы, но вдруг впоследствии что-то изменится, и идентификатор $ окажется занят какой-то другой библиотекой? Поэтому лучше предотвратить появление подобных ошибок.

Один из способов обеспечить корректную работу функции, если кто-то вдруг подменит значение идентификатора $, состоит в том, чтобы вообще не использовать его. Мы могли бы определить нашу простейшую функцию как:

```
jQuery.say = function(what) { alert('I say '+what); }
```

Похоже, выход найден – но для более сложных функций он оказывается менее оптимальным. Что если в теле функции для внутренней работы применяется много методов и функций jQuery? Повсюду внутри функции вместо псевдонима $ нам придется использовать идентификатор jQuery. Это слишком неудобно и некрасиво, кроме того, привыкнув к псевдониму $, мы не хотим отказываться от него!

Поэтому, вернувшись к идиоме, введенной в разделе 7.2.2, мы можем безопасно определить нашу функцию так:

```
(function($){
  $.say = function(what) { alert('I say '+what); }
})(jQuery);
```

Настоятельно рекомендуем использовать этот шаблон (даже при том, что он кажется несколько громоздким для такой простой функции), потому что он защищает работу с псевдонимом $ при объявлении и определении функции. Если функцию когда-либо придется дорабатывать, мы сможем расширить и дополнить ее, не волнуясь о безопасности применения псевдонима $.

Вооружившись этим новым шаблоном, давайте реализуем собственную нетривиальную вспомогательную функцию.

7.3.1. Создание вспомогательной функции для манипулирования данными

При выводе в поля фиксированной ширины часто требуется отформатировать числовое значение, чтобы уместить его в это поле (где *ширина* определяется числом символов). Обычно такие операции выполняются при выводе значений с выравниванием по правому краю, когда к выводимому числу нужно добавить слева *символы-заполнители*, чтобы приравнять длину выводимого значения к ширине поля.

Давайте создадим такую вспомогательную функцию, определяемую следующим синтаксисом:

Синтаксис команды $.toFixedWidth

`$.toFixedWidth(value,length,fill)`

Форматирует значение как строку фиксированной длины. Позволяет указать необязательный символ-заполнитель. Если длина числового значения превышает заданную, цифры старших разрядов отсекаются до заданной длины.

Параметры

`value`	(число) Форматируемое значение.
`length`	(число) Длина результирующей строки.
`fill`	(строка) Символ-заполнитель, служит для дополнения возвращаемой строки слева. Если отсутствует, используется символ `0`.

Возвращаемое значение

Строка фиксированной длины.

Реализация этой функции приведена в листинге 7.1.

Листинг 7.1. Реализация вспомогательной функции $.toFixedWidth()

```
(function($){
  $.toFixedWidth = function(value,length,fill) {
    var result = value.toString();          ❶
    if (!fill) fill = '0';
    var padding = length - result.length;   ❷
    if (padding < 0) {
      result = result.substr(-padding);     ❸
    }
    else {
      for (var n = 0; n < padding; n++)     ❹
        result = fill + result;
    }
    return result;                          ❺
  };
})(jQuery);
```

В этой функции нет ничего сложного. Переданное значение преобразуется в строковый эквивалент, в качестве символа-заполнителя принимается значение входного параметра либо значение по умолчанию 0 ❶. Затем вычисляется, сколько символов следует добавить слева к строке со значением ❷.

Если это число получается отрицательным (результат длиннее, чем заданная длина строки), из результата исключаются лишние символы слева, чтобы уместить его в поле заданной ширины ❸, в противном случае результат дополняется слева соответствующим количеством символов-заполнителей ❹. Полученная строка возвращается как результат функции ❺.

Ничуть не сложно – и это лишний раз доказывает, насколько просто мы можем добавлять вспомогательные функции. Однако здесь есть что улучшить. Попробуйте выполнить следующие упражнения:

- Как и в большинстве примеров этой книги, чтобы сконцентрировать внимание на изучении основного предмета, здесь почти отсутствуют проверки на ошибки. Как бы вы доработали функцию, чтобы учесть ошибки вызывающей программы, такие как передача нечисловых значений в параметрах `value` и `length` или полное отсутствие входных параметров?
- Мы старательно усекаем слишком длинные числовые значения, чтобы результат всегда имел заданную длину. Но если вызывающая программа передаст в параметре `fill` строку, содержащую больше одного символа, все наши труды пойдут прахом. Как бы вы исправили этот недостаток?
- Как бы вы поступили, если бы усекать слишком длинные значения было нежелательно?

Теперь напишем более сложную функцию, в которой применим только что созданную функцию `$.toFixedWidth()`.

7.3.2. Создание функции форматирования даты

Если в мир программирования на стороне клиента вы пришли со стороны сервера, то одной из функций, которых вам, возможно, не хватало, является простая функция форматирования дат, отсутствующая в типе данных `Date` языка JavaScript. Поскольку такая функция должна оперировать экземпляром объекта типа `Date`, а не элементом DOM, ее можно рассматривать как потенциального кандидата на вспомогательную функцию. Давайте напишем функцию, которая использует следующий синтаксис:

Синтаксис команды $.formatDate

```
$.formatDate(date,pattern)
```

Форматирует значение даты в соответствии с заданным шаблоном. В шаблоне допустимы следующие спецификаторы формата:

yyyy – год в четырехзначном формате;
yy – две последние цифры года;
MMMM – полное название месяца;
MMM – сокращенное название месяца;
MM – двухсимвольный номер месяца с предшествующим символом 0;
M – номер месяца;
dd – двухсимвольный номер дня месяца с предшествующим символом 0;
d – номер дня месяца;
EEEE – полное название дня недели;
EEE – сокращенное название дня недели;
a – утро или вечер (AM или PM);
HH – 2 символа часа дня в 24-часовом формате с предшествующим символом 0;
H – час дня в 24-часовом формате;
hh – 2 символа часа дня в 12-часовом формате с предшествующим символом 0;
h – час дня в 12-часовом формате;
mm – 2 символа минут с предшествующим символом 0;
m – минуты;
ss – 2 символа секунд с предшествующим символом 0;
s – секунды;
S – 3 символа миллисекунд с предшествующим символом 0.

Параметры

`date` (дата) Форматируемое значение даты.

`pattern` (строка) Шаблон формата представления даты. Любые символы, не являющиеся допустимыми спецификаторами, просто копируются в результат.

Возвращаемое значение

Форматированная дата.

Реализация этой функции представлена в листинге 7.2. Мы не будем углубляться в описание алгоритма форматирования (в конечном счете, алгоритмы не являются темой этой книги), но на основе этой реализации мы продемонстрируем ряд интересных приемов, которые можно применять при разработке сложных вспомогательных функций.

Листинг 7.2. Реализация вспомогательной функции $.formatDate()

```
(function($){
  $.formatDate = function(date,pattern) {
    var result = [];
    while (pattern.length > 0) {
      $.formatDate.patternParts.lastIndex = 0;
      var matched = $.formatDate.patternParts.exec(pattern);
      if (matched) {
        result.push(
          $.formatDate.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.patternParts =
    /^(yy(yy)?|M(M(M(M)?)?)?|d(d)?|EEE(E)?|a|H(H)?|h(h)?|m(m)?|s(s)?|S)/;

  $.formatDate.monthNames = [
    'January','February','March','April','May','June','July',
    'August','September','October','November','December'
  ];

  $.formatDate.dayNames = [
    'Sunday','Monday','Tuesday','Wednesday','Thursday','Friday',
    'Saturday'
  ];

  $.formatDate.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);
    },
```

❶ Реализация основного тела функции

❷ Определение регулярного выражения

❸ Названия месяцев

❹ Названия дней недели

❺ Коллекция функций преобразования спецификаторов в значения

```
    MM: function(date) {
      return $.toFixedWidth(date.getMonth() + 1,2);
    },
    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';
    }
  };
})(jQuery);
```

Самое интересное в этой реализации помимо некоторых уловок, позволяющих сократить размер программного кода, заключается в том, что функции ❶ требуются некоторые дополнительные данные, в частности:

- регулярное выражение для поиска спецификаторов формата в шаблоне ❷;
- список названий месяцев на английском языке ❸;
- список названий дней недели на английском языке ❹;
- набор дополнительных функций, возвращающих значение каждого спецификатора формата для заданной даты ❺.

Мы могли бы включить все эти данные в тело функции в виде определений `var`, что только загромоздило бы алгоритм, но поскольку эти данные являются константами, есть смысл отделить их от переменных.

Не желая загрязнять глобальное пространство имен (и пространство имен $) идентификаторами, необходимыми только для этой функции, мы поместили объявления этих свойств непосредственно в нашу новую функцию. Не забывайте, что функции JavaScript – обычные объекты и могут обладать собственными свойствами, как и любые другие объекты JavaScript.

Что можно сказать по поводу самого алгоритма? В двух словах он работает так:

- Создается массив, где будут сохраняться части, составляющие результат.
- Выполняется обход строки шаблона, из нее извлекаются спецификаторы формата и обычные символы, пока не будет достигнут конец строки.
- В начале каждой итерации регулярное выражение (хранящееся в `$.formatDate.patternParts`) инициализируется установкой свойства `lastIndex` в значение 0.
- С помощью регулярного выражения выполняется поиск спецификатора формата, соответствующего текущему началу строки шаблона.
- Если совпадение было найдено, вызывается функция из коллекции функций преобразования `$.formatDate.patternValue`, чтобы получить соответствующее значение из экземпляра `Date`. Полученное значение добавляется в конец массива результата, а найденный спецификатор удаляется из начала строки шаблона.
- Если совпадение с текущим началом строки шаблона не найдено, первый символ из строки шаблона удаляется и добавляется в конец массива результата.
- Когда достигнут конец строки шаблона, массив результата преобразуется в строку и возвращается как значение функции.

Обратите внимание: функции преобразования в коллекции `$.formatDate.patternValue` используют функцию `$.toFixedWidth()`, созданную нами в предыдущем разделе.

Обе эти функции вы найдете в файле *chapter7/jquery.jqia.dateFormat.js*, а простенькую страницу для проверки работы этих функций – в файле *chapter7/test.dateFormat.html*.

Манипулирование «родными» объектами JavaScript – дело нужное и хорошее, но главная сила jQuery заключается в методах обертки, которые манипулируют наборами элементов DOM, собранными с помощью селекторов jQuery. Давайте посмотрим, как добавлять собственные методы обертки.

7.4. Добавление новых методов обертки

Истинная мощь jQuery заключается в возможности легко и быстро отобрать и обработать элементы DOM. А мы можем наращивать эту мощь, добавляя собственные методы, которые выполняют операции над выбранными элементами DOM. Добавляя методы обертки, мы автоматически получаем возможность использовать мощные селекторы jQuery для выбора элементов, без необходимости выполнять эту сложную работу самостоятельно.

Наши знания о JavaScript позволяют самостоятельно выяснить, как добавить в пространство имен $ вспомогательные функции – но не функции обертки. Для этого нужно обладать сведениями, характерными для jQuery: чтобы добавить метод обертки в jQuery, необходимо присвоить их соответствующим свойствам объекта `fn` в пространстве имен $.

Типичный способ создания функций обертки:

```
$.fn.wrapperFunctionName = function(params){тело функции};
```

Давайте создадим простейший метод обертки, который будет устанавливать синий цвет для соответствующих элементов DOM.

```
(function($){
  $.fn.makeItBlue = function() {
    return this.css('color','blue');
  }
})(jQuery);
```

Как и в случае со вспомогательными функциями, мы заключаем объявление во внешнюю функцию, которая обеспечивает соответствие псевдонима $ идентификатору `jQuery`. Но в отличие от вспомогательной функции, метод обертки здесь создается как свойство объекта `$.fn`, а не $.

Примечание

Если вы знакомы с объектно-ориентированными возможностями JavaScript и объявлениями классов на основе прототипов, вам будет интересно узнать, что идентификатор `$.fn` – просто псевдоним свойства `prototype` функции-конструктора объекта `jQuery`.

В теле этого метода контекст функции (this) ссылается на обернутый набор. Для работы с ним можно использовать любые предопределенные команды jQuery. В этом примере мы применяем к обернутому набору команду css(), чтобы установить свойство color всех элементов DOM из обернутого набора в значение blue.

Предупреждение

Контекст функции (this) внутри метода обертки ссылается на обернутый набор, но встроенные функции, объявленные в теле этой функции, имеют собственный контекст. При использовании ссылки this в таких обстоятельствах вы должны позаботиться о том, чтобы она указывала именно туда, куда вам требуется! Например, если вы используете each() в паре с функцией-итератором, ссылка this внутри функции-итератора будет ссылаться на элемент DOM текущей итерации.

Над элементами DOM в обернутом наборе можно выполнять практически любые операции, но есть одно *очень* важное правило, которое применяется при создании новых методов обертки: если создаваемая функция не возвращает какое-то определенное значение, она всегда должна возвращать обернутый набор. Это правило позволит нашей новой команде входить в состав любых цепочек команд jQuery. В нашем примере благодаря тому, что команда css() возвращает обернутый набор, мы просто будем возвращать результат, полученный от css().

В данном примере команда css() библиотеки jQuery относится ко всем элементам в обернутом наборе, поэтому мы применяем ее к ссылке this. Если по каким-то причинам требуется обработать каждый обернутый элемент отдельно (например, с учетом каких-либо условий), можно использовать следующий шаблон:

```
(function($){
  $.fn.someNewMethod = function() {
    return this.each(function(){
      //
      // Здесь располагается тело функции - контекст this ссылается
      // на отдельные элементы
      //
    });
  }
})(jQuery);
```

В этом шаблоне команда each() выполняет обход всех элементов в обернутом наборе. Примечательно, что внутри функции-итератора ссылка this указывает на текущий элемент DOM, а не на весь обернутый набор. Новый метод возвращает обернутый набор, полученный как результат метода each(), поэтому данный метод может входить в состав цепочек команд.

Это все, что можно сказать, но (опять это пресловутое *но*?) есть несколько приемов, которые необходимо знать и использовать при создании

более сложных методов обертки jQuery. Давайте определим еще пару методов расширения, чтобы исследовать эти приемы на практике.

7.4.1. Применение нескольких операций в методах обертки

Давайте создадим другой метод расширения, который выполняет над обернутым набором несколько операций. Предположим, требуется реализовать перевод всех текстовых полей формы в состояние «только для чтения» и при этом соответственно изменить их внешний вид. Эту задачу легко решить с помощью цепочки команд jQuery, но хочется, чтобы в нашем пакете эти операции выполнялись в одном методе.

Назовем новую команду `setReadOnly()`. Ее синтаксис:

Синтаксис команды setReadOnly

`setReadOnly(state)`

Устанавливает обернутые текстовые поля в состояние, определяемое параметром `state`. Непрозрачность текстовых полей устанавливается так: 100% – если состояние не «только для чтения»; 50% – если состояние «только для чтения». Любые элементы в обернутом наборе, не являющиеся текстовыми полями, игнорируются.

Параметры

`state` (логическое значение) При значении `true` устанавливается состояние «только для чтения», в противном случае состояние «только для чтения» сбрасывается.

Возвращаемое значение

Обернутый набор.

Реализация расширения приведена в листинге 7.3, кроме того, вы найдете ее в файле *chapter7/jquery.jqia.setreadonly.js*.

Листинг 7.3. Реализация метода расширения setReadOnly( )

```
(function($){
  $.fn.setReadOnly = function(readonly) {
    return this.filter('input:text')
      .attr('readonly',readonly)
      .css('opacity', readonly ? 0.5 : 1.0);
  }
})(jQuery);
```

Этот пример немногим сложнее предыдущего, но в нем продемонстрированы следующие ключевые концепции:

- методу передается параметр, определяющий его поведение;
- к обернутому набору применяются три команды jQuery, объединенные в цепочку;

- новая команда может входить в состав цепочек jQuery, потому что она возвращает обернутый набор;
- команда `filter()` гарантирует воздействие только на текстовые поля независимо от того, какие элементы включены в обернутый набор автором страницы.

Как бы мы могли использовать этот метод?

Часто при определении электронной формы заказа нам может понадобиться, чтобы пользователь мог ввести два адреса: первый – для доставки заказа, второй – для доставки счета. Чаще всего эти адреса совпадают, поэтому невежливо заставлять его вводить одну и ту же информацию дважды.

Можно было бы написать программный код на стороне сервера так, чтобы он принимал адрес, указанный для доставки заказа, в качестве адреса доставки счета, если это поле формы осталось незаполненным. Но будем считать, что приемщик заказов предельно педантичен и предпочитает получать от пользователя всю информацию.

Мы удовлетворим его потребности, добавив рядом с полем адреса доставки счета флажок, указывающий, что этот адрес совпадает с адресом доставки заказа. Если этот флажок установлен, адрес доставки счета копируется из поля адреса доставки заказа, которое затем переводится в режим «только для чтения». При снятии флажка поле ввода очищается и его состояние «только для чтения» отменяется.

На рис. 7.1 показана тестовая форма до и после изменения состояния.

Сама тестовая форма находится в файле *chapter7/test.setreadonly.html*, а ее исходный код разметки приведен в листинге 7.4.

Листинг 7.4. Реализация формы для тестирования новой команды setReadOnly()

```
<html>
  <head>
    <title>setReadOnly() Test</title>
    <link rel="stylesheet" type="text/css" href="../common.css">
    <link rel="stylesheet" type="text/css"
          href="test.setreadonly.css">
    <script type="text/javascript"
            src="../scripts/jquery-1.2.1.js"></script>
    <script type="text/javascript"
            src="jquery.jqia.setreadonly.js"></script>
    <script type="text/javascript">
      $(function(){
        $('#sameAddressControl').click(function(){
          var same = this.checked;
          $('#billAddress').val(same ? $('#shipAddress').val():'');
          $('#billCity').val(same ? $('#shipCity').val():'');
          $('#billState').val(same ? $('#shipState').val():'');
          $('#billZip').val(same ? $('#shipZip').val():'');
```

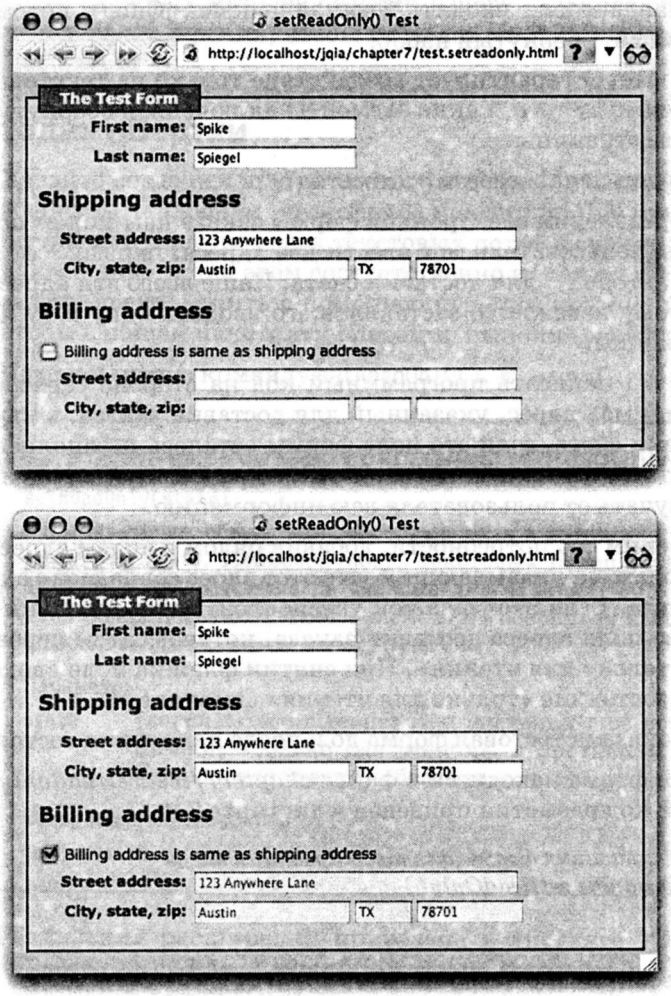


***Рис. 7.1.** Тестовая форма до и после щелчка на флажке*

```
        $('#billingAddress input')
          .setReadOnly(same);
      });
    });
  </script>
</head>

<body>
  <fieldset>
    <legend>The Test Form</legend>
    <div>
      <form name="testForm">
        <div>
```

❶ Применяет новый модуль расширения

```
        <label>First name:</label>
        <input type="text" name="firstName" id="firstName"/>
      </div>
      <div>
        <label>Last name:</label>
        <input type="text" name="lastName" id="lastName"/>
      </div>
      <div id="shippingAddress">
        <h2>Shipping address</h2>
        <div>
          <label>Street address:</label>
          <input type="text" name="shipAddress"
                 id="shipAddress"/>
        </div>
        <div>
          <label>City, state, zip:</label>
          <input type="text" name="shipCity" id="shipCity"/>
          <input type="text" name="shipState" id="shipState"/>
          <input type="text" name="shipZip" id="shipZip"/>
        </div>
      </div>
      <div id="billingAddress">
        <h2>Billing address</h2>
        <div>
          <input type="checkbox" id="sameAddressControl"/>
          Billing address is same as shipping address
        </div>
        <div>
          <label>Street address:</label>
          <input type="text" name="billAddress"
                 id="billAddress"/>
        </div>
        <div>
          <label>City, state, zip:</label>
          <input type="text" name="billCity" id="billCity"/>
          <input type="text" name="billState" id="billState"/>
          <input type="text" name="billZip" id="billZip"/>
        </div>
      </div>
    </form>
  </div>
 </fieldset>
</body>
</html>
```

Не будем вдаваться в подробности, так как в этой странице нет ничего сложного. Единственное, что действительно интересно, – обработчик события щелчка мыши, подключенный к флажку в обработчике события готовности документа. При щелчке, изменяющем состояние флажка:

1. **Состояние флажка копируется в переменную `same`, чтобы упростить работу с этим значением в оставшейся части обработчика.**

2. Определяются значения полей адреса доставки счета. Если это тот же адрес, куда доставляется заказ, копируется содержимое соответствующих полей адреса доставки заказа. Если нет – поля очищаются.
3. Для всех полей ввода в блоке адреса доставки счета вызывается новая команда `setReadOnly()` ❶.

Минуточку! На этом последнем шаге мы поступаем не совсем аккуратно. Обернутый набор, созданный с помощью `$('#billingAddress input')`, содержит не только текстовые поля в блоке адреса доставки счета, но и сам флажок. Элемент флажка не имеет семантики «только для чтения», но его непрозрачность может изменяться – это мы определенно не предусмотрели!

К счастью, от этой неаккуратности спасает наша *предусмотрительность* при определении модуля расширения. Вспомните: прежде чем применять остальные команды в этом методе, мы отфильтровываем все элементы, которые не являются текстовыми полями ввода. Настоятельно рекомендуем проявлять такого рода внимание к деталям, особенно в модулях расширения, предназначенных для общего пользования.

Можно ли как-то улучшить эту команду? Попробуйте выполнить следующие упражнения.

- Мы забыли о текстовых областях ввода! Как бы вы изменили программный код, чтобы добавить обработку текстовых областей наряду с текстовыми полями?
- Степень непрозрачности полей для разных состояний в функции жестко определенна. Такая жесткость может быть неудобной для вызывающей программы. Измените метод так, чтобы позволить вызывающей программе самой определять степень непрозрачности.
- Почему мы разрешаем автору страницы влиять только на непрозрачность? Как бы вы изменили метод, чтобы позволить автору страницы определять параметры отображения полей в любом из двух состояний?

7.4.2. Сохранение состояния внутри метода обертки

Многие любят смотреть фотографии!

По крайней мере, в Сети. В отличие от несчастных гостей, вынужденных после ужина сидеть перед экраном, до умопомрачения глядя на бесконечную череду нечетких снимков, посетители веб-сайтов могут покинуть их в любой момент, никого не обидев!

В более сложном примере модуля расширения мы создадим новую команду jQuery, которая позволит быстро создавать страницы для просмотра фотографий. Созданный модуль расширения для jQuery мы назовем *Photomatic*, а затем создадим тестовую страницу, чтобы проверить модуль на практике. По завершении работы тестовая страница будет выглядеть, как показано на рис. 7.2.

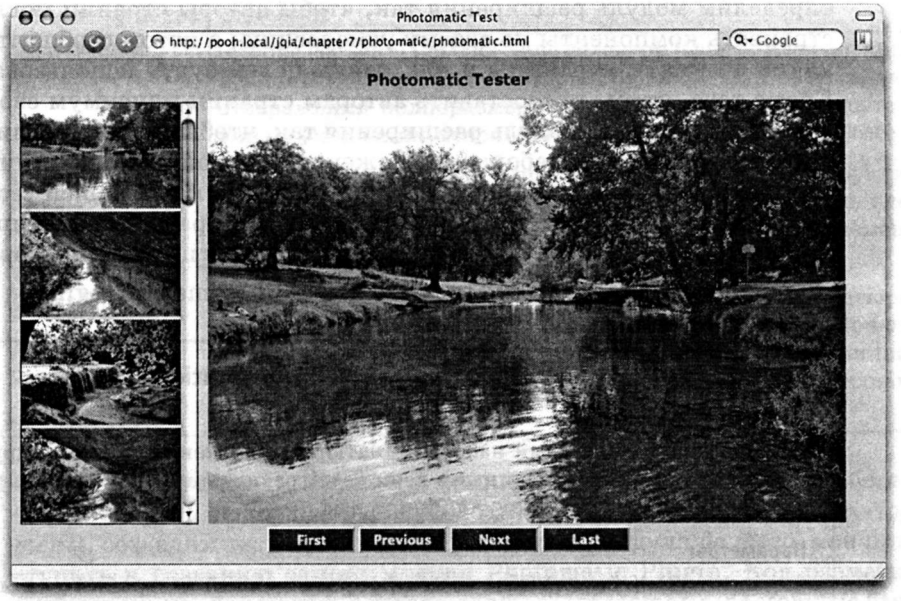


***Рис. 7.2.** Страница Photomatic Tester, предназначенная для проверки нашего модуля на практике*

Эта страница состоит из следующих компонентов:

- набор миниатюр изображений;
- полноразмерное изображение, одно из списка миниатюр;
- набор кнопок для перемещения по изображениям.

Страница будет обладать следующим поведением:

- по щелчку мыши на любой миниатюре изображения отображается его полноразмерная версия;
- по щелчку мыши на полноразмерном изображении выполняется переход к следующему изображению;
- по щелчку на кнопках выполняются следующие операции:
  - First (в начало) – переход к первому изображению;
  - Previous (назад) – переход к предыдущему изображению;
  - Next (вперед) – переход к следующему изображению;
  - Last (в конец) – переход к последнему изображению;
- любая операция, вызвавшая переход за конец (или начало) списка изображений, будет выполнять переход к началу (или в конец) списка. Например, при щелчке на кнопке Next, когда отображается последнее изображение, выполняется переход к первому изображению в списке, и наоборот.

Мы также предоставим авторам страниц такую свободу в выборе размещения и оформления компонентов, какая только возможна. Мы

определим модуль расширения так, чтобы авторы страниц могли настраивать компоненты по своему усмотрению, а потом сообщать нам, какие элементы страницы и для каких целей будут использоваться. Кроме того, чтобы предоставить авторам страниц максимум свободы, мы определим наш модуль расширения так, чтобы авторы могли сами создавать обернутые наборы изображений, играющих роль миниатюр. Обычно миниатюры собираются в одном вместе, как в нашей тестовой странице, но авторы страниц смогут сами выбирать на странице любые изображения, которые будут представлены в виде миниатюр.

Для начала рассмотрим синтаксис модуля Photomatic Plugin.

Синтаксис команды photomatic

`photomatic(settings)`

Обрабатывает обернутый набор миниатюр изображений, а также другие элементы страницы, заданные в хеше `settings`, которые будут играть роль элементов управления модуля Photomatic.

Параметры

`settings` (объект) Объект-хеш, определяющий настройки для модуля Photomatic. За дополнительной информацией обращайтесь к табл. 7.1.

Возвращаемое значение

Обернутый набор.

Поскольку параметров, определяющих порядок работы модуля Photomatic (многие из которых могут иметь значения по умолчанию), достаточно много, мы передаем эти параметры методу с помощью объекта-хеша, описанного в разделе 7.2.3. Допустимые для передачи параметры перечислены в табл. 7.1.

Таблица 7.1. Параметры команды Photomatic( )

Имя параметра	Описание
`firstControl`	(строка | объект) Ссылка либо селектор jQuery, идентифицирующий элемент(ы) DOM, который будет играть роль элемента управления First (в начало). При отсутствии элемент управления не создается.
`lastControl`	(строка | объект) Ссылка либо селектор jQuery, идентифицирующий элемент(ы) DOM, который будет играть роль элемента управления Last (в конец). При отсутствии элемент управления не создается.
`nextControl`	(строка | объект) Ссылка либо селектор jQuery, идентифицирующий элемент(ы) DOM, который будет играть роль элемента управления Next (вперед). При отсутствии элемент управления не создается.

Имя параметра	Описание
photoElement	(строка | объект) Ссылка либо селектор jQuery, идентифицирующий элемент <img>, который будет использоваться для отображения полноразмерного изображения. При отсутствии по умолчанию используется селектор '#photomaticPhoto'.
previousControl	(строка | объект) Ссылка либо селектор jQuery, идентифицирующий элемент(ы) DOM, который будет играть роль элемента управления Previous (назад). При отсутствии элемент управления не создается.
transformer	(функция) Функция для преобразования URL миниатюры изображения в URL полноразмерного изображения. При отсутствии по умолчанию будет использоваться URL миниатюры, где подстрока *thumbnail* будет замещена подстрокой *photo*.

Проявляя должное уважение к методике *создания программного обеспечения с предшествующей разработкой тестов* (*test-driven development*), создадим тестовую страницу для этого модуля *до того*, как приступить к созданию самого модуля Photomatic Plugin. Код разметки этой страницы находится в файле *chapter7/photomatic/photomatic.html* и приведен в листинге 7.5.

Листинг 7.5. Тестовая страница, которая создает отображение (рис. 7.2)

```
<html>
  <head>
    <title>Photomatic Test</title>
    <link rel="stylesheet" type="text/css" href="../../common.css">
    <link rel="stylesheet" type="text/css" href="photomatic.css">
    <script type="text/javascript"
            src="../../scripts/jquery-1.2.1.js"></script>
    <script type="text/javascript"
            src="jquery.jqia.photomatic.js"></script>
    <script type="text/javascript">
      $(function(){
        $('#thumbnails img').photomatic({
          photoElement: '#photo',
          previousControl: '#previousButton',
          nextControl: '#nextButton',
          firstControl: '#firstButton',
          lastControl: '#lastButton'
        });
      });
    </script>
  </head>

  <body>
    <h1>Photomatic Tester</h1>
    <div id="thumbnails">
      <img src="thumbnails/IMG_0340.jpg"/>
```

❶ Вызывает Photomatic Plugin

❷ Содержит миниатюры изображений

```
      <img src="thumbnails/IMG_0939.jpg"/>
      <img src="thumbnails/IMG_1520.jpg"/>
      <img src="thumbnails/IMG_1904.jpg"/>
      <img src="thumbnails/IMG_2063.jpg"/>
      <img src="thumbnails/IMG_2186.jpg"/>
    </div>
    <div id="photoContainer">
      <img id="photo" src=""/>
    </div>
    <div id="buttonBar">
      <button type="button" id="firstButton">First</button>
      <button type="button" id="previousButton">Previous</button>
      <button type="button" id="nextButton">Next</button>
      <button type="button" id="lastButton">Last</button>
    </div>
  </body>
</html>
```

❸ Элемент для отображения полноразмерного изображения

❹ Содержит элементы, играющие роль элементов управления

Если страница выглядит проще, чем вы предполагали, это уже не должно вас удивлять. Применяя принцип ненавязчивого JavaScript и сохраняя всю информацию о стилях во внешних таблицах стилей, наша разметка становится простой и аккуратной. В действительности, даже сценарий, занимающий незначительную часть страницы, состоит из единственной инструкции, которая вызывает наш модуль расширения ❶.

HTML-разметка состоит из контейнера с миниатюрами изображений ❷, элемента <img> (для которого изначально URL-адрес изображения не указан) для отображения полноразмерной фотографии ❸ и набора кнопок для управления просмотром фотографий ❹. Все остальное будет делать модуль расширения.

Приступим к его разработке.

Для начала создадим заготовку (мы будем наполнять ее новыми возможностями по ходу обсуждения). Заготовка должна быть вам уже знакома, потому что создана на основе тех же шаблонов, о которых говорилось выше.

```
(function($){
  $.fn.photomatic = function(callerSettings) {
  };
})(jQuery);
```

Это определение нашей, пока еще пустой, функции обертки, которая (как ожидается, если исходить из описания синтаксиса) принимает единственный параметр-хеш с именем callerSettings. Внутри функции мы сначала объединим параметры, переданные вызывающей программой, со значениями по умолчанию, в соответствии с описанием в табл. 7.1. В результате получим единый объект с параметрами настройки, которые мы сможем использовать в оставшейся части метода.

Операцию объединения выполним с применением следующей идиомы (встречавшейся нам уже не раз):

```
var settings = $.extend({
  photoElement: '#photomaticPhoto',
  transformer: function(name) {
                 return name.replace(/thumbnail/,'photo');
               },
  nextControl: null,
  previousControl: null,
  firstControl: null,
  lastControl: null
}, callerSettings||{});
```

После выполнения этой инструкции переменная `settings` будет содержать значения, полученные от вызывающей программы, со значениями по умолчанию, полученными из встроенного объекта. Но на этом работа с переменной `settings` еще не закончена. Взгляните на свойство `photoElement`: оно может содержать строку, определяющую селектор jQuery (либо по умолчанию, либо полученный от автора страницы) или ссылку на объект. Нужно определить, с чем мы имеем дело. Добавив инструкцию

```
settings.photoElement = $(settings.photoElement);
```

мы создаем обернутый набор, содержащий элемент с фотографией (или несколько элементов, если так пожелал автор страницы). Теперь мы знаем, с чем работать.

Кроме того, нужно отслеживать некоторые параметры. Чтобы понять, что означают такие термины, как *вперед* и *назад*, нам потребуется не только список миниатюр изображений, но и *индикатор*, указывающий на текущее *изображение*.

Список миниатюр изображений – это обернутый набор, управляемый методом, или, по крайней мере, так должно быть. Мы не знаем, какие элементы отобрал автор в обернутый набор, поэтому отфильтруем его, чтобы остались только изображения. Это несложно сделать с помощью селектора jQuery. Но где сохранить полученный список?

Мы легко могли бы создать для этого еще одну переменную, но лучше все-таки хранить все настройки в одном месте. Давайте сохраним список в другом свойстве объекта `settings`:

```
settings.thumbnails = this.filter('img');
```

После фильтрации обернутого набора (доступного методу через ссылку `this`) в новом обернутом наборе остаются только изображения (то есть он содержит только элементы `<img>`). Мы сохраним его в свойстве `thumbnails` объекта `settings`.

Еще один параметр, который нужно отслеживать, – это *текущее* изображение. Мы будем следить за ним, сохраняя его индекс в списке миниатюр и добавив к объекту `settings` еще одно свойство с именем `current`:

```
settings.current = 0;
```

Есть еще один шаг, который мы должны сделать. Чтобы следить за *текущим* изображением по его индексу, нам потребуется, как минимум, один раз определить индекс изображения по ссылке на него. Проще всего это сделать, добавив *дополнительное* (нестандартное) свойство к каждому элементу с миниатюрой изображения и записав в него индекс, соответствующий элементу. Мы делаем это с помощью следующей инструкции:

```
settings.thumbnails.each(function(n){this.index = n;});
```

Эта инструкция выполняет обход всех миниатюр изображений, добавляет к каждой из них свойство `index` и записывает в него порядковый номер изображения. Теперь, когда начальное состояние зафиксировано, мы готовы перейти к основной части модуля.

Минутку! *Состояние*? Как мы можем следить за состоянием, хранящимся в локальной переменной функции, которая собирается завершить свою работу? После выхода из функции локальная переменная исчезнет, и вместе с ней исчезнут все наши настройки!

В общем случае это действительно так, но есть частный случай, когда такие переменные продолжают существовать после выхода из их обычной области видимости, – этот случай называется *замыканием*. Прежде нам уже встречались замыкания, но если вам не все ясно, пожалуйста, ознакомьтесь с Приложением. Вы должны разобраться с замыканиями не только для того, чтобы закончить реализацию модуля Photomatic Plugin, – они также потребуются вам для создания чего-то посущественнее простеньких модулей.

Задумавшись о том, что осталось сделать, мы понимаем, что нам потребуется подключить несколько обработчиков событий к элементам управления и другим элементам, которые мы уже получили такими большими усилиями. Каждый определяемый обработчик должен создавать замыкание, чтобы получить доступ к переменной `settings`, поэтому можно не сомневаться, что даже при кажущейся неустойчивости переменной `settings` состояние, которое она хранит, будет доступно всем обработчикам событий, которые мы определим.

Заговорив об обработчиках, приведем список обработчиков событий `click`, которые мы должны подключить к различным элементам:

- По щелчку на миниатюре изображения отображается его полноразмерная версия.
- По щелчку на полноразмерном изображении выполняется переход к следующему изображению.
- По щелчку на элементе управления, определенном для перехода назад, выполняется переход к предыдущему изображению.
- По щелчку на элементе управления, определенном для перехода вперед, выполняется переход к следующему изображению.

- По щелчку на элементе управления, определенном для перехода в начало, выполняется переход к первому изображению.
- По щелчку на элементе управления, определенном для перехода в конец, выполняется переход к последнему изображению.

Просмотрев этот список, сразу же замечаем, что у всех обработчиков есть нечто общее – все они отображают полноразмерное изображение, соответствующее одной из миниатюр. Как грамотные и умные программисты, вынесем эту общую часть в отдельную функцию, чтобы не дублировать фрагмент программного кода многократно.

Но как это сделать?

Если бы мы писали обычный сценарий JavaScript для страницы, то могли бы определить глобальную функцию. Если бы мы использовали объектно-ориентированные особенности, то могли бы определить метод в объекте JavaScript. Но мы создаем модуль расширения jQuery – так где нам определить реализацию функции?

Для нас нежелательно посягать на глобальное пространство имен или на пространство имен $, чтобы определить функцию, которая нужна только нашему модулю, – так что же нам делать? Мы *могли бы* определить функцию как метод самой функции модуля (точно так же, как мы делали это, создавая функцию форматирования даты в листинге 7.2). В JavaScript даже функции могут иметь методы, поскольку являются самыми обычными объектами. Но есть и более простой способ.

Не забывайте, что наша функция модуля определена в теле другой функции – функции, с помощью которой гарантируется соответствие идентификаторов $ и `jQuery`. Таким образом, локальные переменные, определенные внутри этой внешней функции, являются частью замыкания, созданного нашей функцией модуля. Что если мы определим реализацию функции, которую назовем `showPhoto()`, как локальную переменную внешней функции?

Это решит нашу проблему! Функция `showPhoto()` будет доступна функции модуля как часть этого замыкания, но будучи объявленной локально во внешней функции, она не будет доступна за пределами этой функции и не загрязнит какое-либо другое пространство имен.

За пределами функции модуля, но внутри внешней функции, мы определяем функцию `showPhoto()`:

```
var showPhoto = function(index) {
  settings.photoElement
    .attr('src',
          settings.transformer(settings.thumbnails[index].src));
  settings.current = index;
};
```

Эта функция, получив индекс миниатюры изображения, полноразмерную версию которого нужно отобразить, на основе значений в объекте `settings` выполняет следующее:

1. Отыскивает атрибут src миниатюры по ее индексу.
2. Передает его значение функции transformer для получения URL полноразмерной фотографии из URL миниатюры.
3. Записывает результат в атрибут src элемента полноразмерного изображения.
4. Запоминает индекс отображаемой фотографии как новый текущий индекс.

Но прежде чем хвалить себя за сообразительность, вспомним, что осталась еще одна проблема. Функции showPhoto() требуется доступ к параметрам в переменной settings, но она определена *внутри* функции модуля и недоступна за ее пределами.

«Ох уж эти замыкания, Бэтмен! Как выйти из сложившегося положения?»

Можно просто передать переменную settings в функцию в виде параметра, но мы пойдем другим путем. Так как проблема в том, что переменная settings определена не во внешнем замыкании, содержащем реализацию и функции модуля, самое простое решение – просто *переместить определение туда*. Такой подход никак не повлияет на все внутренние замыкания, но обеспечит доступность переменной как в функции модуля, так и в функции showPhoto() или в любых других функциях, которые нам потребуется определить. (Этот прием предполагает, что у нас на странице используется только один экземпляр Photomatic – это ограничение не является существенным в данном случае. В более общем случае передача переменной settings в виде параметра не накладывает такого ограничения.)

Таким образом, нужно добавить во внешнюю функцию следующее объявление:

```
var settings;
```

И удалить ключевое слово var из инструкции в функции модуля:

```
settings = $.extend({
```

Теперь переменная settings доступна обеим функциям, и мы по-настоящему готовы к тому, чтобы закончить реализацию функции модуля. Далее мы определяем обработчики событий, перечисленные выше:

```
settings.thumbnails.click(function(){ showPhoto(this.index); });
settings.photoElement.click(function(){
  showPhoto((settings.current + 1) % settings.thumbnails.length);
});
$(settings.nextControl).click(function(){
  showPhoto((settings.current + 1) % settings.thumbnails.length);
});
$(settings.previousControl).click(function(){
  showPhoto((settings.thumbnails.length + settings.current - 1) %
            settings.thumbnails.length);
```

```
});
$(settings.firstControl).click(function(){
  showPhoto(0);
});
$(settings.lastControl).click(function(){
  showPhoto(settings.thumbnails.length - 1);
});
```

Каждый из этих обработчиков вызывает функцию showPhoto() и передает ей индекс миниатюры изображения, определяемый либо индексом миниатюры, на которой был сделан щелчок, либо длиной списка, либо вычисляется относительно текущего индекса. (Обратите внимание на то, как используется оператор деления по модулю, чтобы выполнить переход к началу списка, если вычисленный индекс выходит за любую из границ списка.)

Осталось решить еще две задачи, прежде чем мы сможем возвестить об успехе, – нужно отобразить первую фотографию до того, как пользователь получит возможность взаимодействовать со страницей, и вернуть первоначальный обернутый набор, чтобы наш модуль мог участвовать в цепочках команд jQuery. В этом нам поможет следующий фрагмент:

```
showPhoto(0);
return this;
```

Теперь можно праздновать победу – мы наконец-то закончили!

Полный исходный код модуля расширения, который вы найдете в файле *chapter7/photomatic/jquery.jqia.photomatic.js*, приведен в листинге 7.6.

Листинг 7.6. Законченная реализация Photomatic Plugin

```
(function($){
  var settings;

  $.fn.photomatic = function(callerSettings) {
    settings = $.extend({
      photoElement: '#photomaticPhoto',
      transformer: function(name) {
                     return name.replace(/thumbnail/,'photo');
                   },
      nextControl: null,
      previousControl: null,
      firstControl: null,
      lastControl: null
    }, callerSettings || {});
    settings.photoElement = $(settings.photoElement);
    settings.thumbnails = this.filter('img');
    settings.thumbnails.each(function(n){this.index = n;});
    settings.current = 0;
    settings.thumbnails.click(function(){ showPhoto(this.index); });
    settings.photoElement.click(function(){
      showPhoto((settings.current + 1) % settings.thumbnails.length);
    });
```

```
    $(settings.nextControl).click(function(){
      showPhoto((settings.current + 1) % settings.thumbnails.length);
    });
    $(settings.previousControl).click(function(){
      showPhoto((settings.thumbnails.length + settings.current - 1) %
                settings.thumbnails.length);
    });
    $(settings.firstControl).click(function(){
      showPhoto(0);
    });
    $(settings.lastControl).click(function(){
      showPhoto(settings.thumbnails.length - 1);
    });
    showPhoto(0);
    return this;
  };

  var showPhoto = function(index) {
    settings.photoElement
      .attr('src',
            settings.transformer(settings.thumbnails[index].src));
    settings.current = index;
  };

})(jQuery);
```

Пока мы продирались через него, казалось, он будет длиннее 45 строк, правда?

Этот модуль расширения – типичный программный код в стиле jQuery, реализующий множество особенностей за счет компактного программного кода. А еще это яркий пример набора приемов – замыкания позволяют сохранять информацию о состоянии, доступную во всем модуле, и обеспечивают возможность создания частных функций, которые модуль может определять и использовать, не загрязняя ни одно из пространств имен.

Теперь вы готовы создавать собственные модули расширения для jQuery. Придумав нечто полезное, рассмотрите возможность поделиться этим с остальным сообществом пользователей jQuery. За дополнительной информацией обращайтесь по адресу *http://jquery.com/plugins/*.

7.5. Итоги

В этой главе даны начальные сведения о том, как создавать свои расширения для jQuery.

Создание собственных расширений для jQuery несет множество преимуществ. Это не только помогает обеспечить непротиворечивость программного кода по всему веб-приложению, вне зависимости от того, используем ли мы функции jQuery или собственные, но также привносит мощь jQuery в наш собственный программный код.

Следование некоторым правилам именования поможет избежать конфликтов между именами файлов, а также проблем, с которыми можно было бы столкнуться, когда идентификатор $ переназначается страницей, использующей наш модуль.

Создание вспомогательных функций выполняется очень просто и заключается в создании свойств функции $, а создание новых методов – в создании свойств объекта $.fn.

Если при разработке модуля у вас возникают трудности, настоятельно рекомендуем вам загрузить и ознакомиться с программным кодом готовых модулей, чтобы увидеть, как их авторы выполнили реализацию. Вы сможете посмотреть, насколько широко используются приемы, представленные в этой главе, а также познакомиться с новыми приемами, описание которых далеко выходит за рамки этой книги.

Узнав о jQuery много нового, давайте посмотрим, насколько просто jQuery позволяет интегрировать технологию Ajax в наши полнофункциональные интернет-приложения.

В этой главе:

- Краткий обзор технологии Ajax
- Загрузка готовой HTML-разметки с сервера
- Выполнение общих запросов GET и POST
- Полное управление запросами Ajax
- Установка значений по умолчанию для свойств Ajax
- Всеобъемлющий пример

8

Взаимодействие с сервером по технологии Ajax

Можно смело утверждать, что за последние несколько лет ни одна технология так не повлияла на развитие Сети, как технология Ajax. Возможность осуществлять асинхронные запросы к серверу без необходимости полной перезагрузки страниц способствовала появлению целого набора новых парадигм взаимодействия с пользователем и обеспечила возможность создания полнофункциональных интернет-приложений.

Технология Ajax – всовсе не новейшее дополнение к инструментам Сети, как многие могли бы подумать. В 1998 году компания Microsoft реализовала возможность выполнять асинхронные запросы под управлением сценариев (избавив от необходимости использовать для этого `<iframe>`) в виде элемента управления ActiveX, что требовалось для создания Outlook Web Access (OWA)[1], и хотя OWA пользовался определенным успехом, похоже, очень немногие заметили его базовую технологию.

Спустя несколько лет, благодаря некоторым событиям технология Ajax проникла в коллективное сознание сообщества веб-разработчиков. В броузерах, созданных не корпорацией Microsoft, была реализована стандартизованная версия технологии в виде объекта XMLHttpRequest (XHR). Компания Google начала применять XHR, и в 2005 году Джесси Джеймс Гарретт (Jesse James Garrett) из компании Adaptive Path придумал термин *Ajax* (сокращение от *Asynchronous JavaScript and XML – асинхронный JavaScript и XML*).

---

[1] Веб-клиент для доступа к серверу совместной работы Microsoft Exchange. – *Прим. перев.*

Все как будто только и ждали, когда технология получит броское название. Широкие массы веб-разработчиков тут же заметили *разрекламированную* технологию Ajax, и она превратилась в один из основных инструментов, с помощью которых мы можем создавать полноценные интернет-приложения.

В этой главе мы вкратце ознакомимся с технологией Ajax (если вы уже достаточно близко знакомы с Ajax, можете сразу перейти к разделу 8.2) и с тем, как применять ее при помощи jQuery.

Для начала узнаем, что представляет из себя технология Ajax.

8.1. Знакомство с Ajax

Несмотря на то что этот раздел знакомит нас с технологией Ajax, он не является полным учебным пособием по Ajax. Тем, кто ничего не знает об этой технологии (или, хуже того, пребывает в уверенности, что речь идет о средстве для мытья посуды или о герое древнегреческих мифов), рекомендуем обратиться к источникам информации, которые расскажут об Ajax *все*. Прекрасный пример таких источников – книги издательства Manning, такие как «Ajax in Action»[1] и «Ajax in Practice»[2].

Некоторые утверждают, что термин «Ajax» применим к любым технологиям, позволяющим выполнять запросы к серверу без необходимости полностью обновлять веб-страницы (например, передача запроса с помощью скрытого элемента `<iframe>`), но большинство связывают этот термин с использованием объекта XHR или элемента управления ActiveX – XMLHTTP, разработанного компанией Microsoft.

Давайте посмотрим, как с помощью этих объектов генерируются запросы к серверу. Начнем с создания такого объекта.

8.1.1. Создание экземпляра XHR

В идеальном мире программный код, написанный для одного броузера, работал бы в любом другом броузере. Но мы уже знаем, что наш мир не идеален, и технология Ajax не является исключением. Есть стандартная методика выполнения асинхронных запросов – с помощью объекта JavaScript XHR, и собственная методика Internet Explorer – с помощью элемента управления ActiveX. В IE7 появилась обертка, имитирующая стандартный интерфейс, но для IE6 приходится писать собственный программный код.

После создания экземпляра объекта остальной программный код (к счастью для нас), выполняющий настройку объекта, инициализацию запроса и получение ответа на него, относительно слабо зависит от типа броузера, а сам процесс создания экземпляра XHR выполняется

---

1 «Ajax в действии». – (К.: Вильямс, 2006.). – *Прим. перев.*

2 «Ajax на практике». – (К.: Вильямс, 2007.). – *Прим. перев.*

достаточно просто в любом броузере. Проблема в том, что в различных броузерах объект XHR реализован по-разному, и мы вынуждены создавать его способом, характерным для используемого броузера.

Вместо того чтобы полагаться на идентификационную информацию о броузере клиента с целью определить, каким путем пойти, мы применим более предпочтительную методику, называемую *обнаружением объекта*. Эта методика предполагает определение функциональных возможностей броузера, а не его типа. Программный код, применяющий методику обнаружения объекта, более устойчив, потому что способен выполняться в любом броузере, поддерживающем возможность выполнения проверки.

В листинге 8.1 приведен программный код, демонстрирующий типичный способ создания экземпляра XHR с применением этой методики.

Листинг 8.1. С методикой обнаружения объекта программный код способен выполняться в большинстве броузеров

```
var xhr;
if (window.XMLHttpRequest) {
  xhr = new XMLHttpRequest();
}
else if (window.ActiveXObject) {
  xhr = new ActiveXObject("Msxml2.XMLHTTP");
}
else {
  throw new Error("Ajax is not supported by this browser");
}
```

Проверка наличия XHR

Проверка наличия объекта ActiveX

Возбуждает исключение, если XHR не поддерживается

После создания объект XHR предоставляет множество удобных свойств и методов, присутствующих во всех поддерживающих его броузерах. Список этих свойств и методов приведен в табл. 8.1, а наиболее часто используемые из них описаны в последующих разделах.

Таблица 8.1. Методы и свойства XHR

Метод	Описание
`abort()`	Отменяет выполнение текущего запроса.
`getAllResponseHeaders()`	Возвращает единую строку, содержащую имена и значения всех заголовков ответа.
`getResponseHeader(name)`	Возвращает значение заголовка с указанным именем.
`open(method,url,async, username,password)`	Определяет метод отправки запроса и URL-адрес. Дополнительно запрос может быть объявлен как синхронный, и к нему могут прилагаться имя пользователя и пароль для прохождения процедуры аутентификации на сервере.
`send(content)`	Инициализирует запрос заданным содержимым (необязательный параметр).
`setRequestHeader(name,value)`	Устанавливает заголовок запроса с заданным именем и содержимым.

Свойства	Описание
onreadystatechange	Обработчик события, вызываемый при изменении состояния запроса.
readyState	Целочисленное значение, определяющее состояние запроса: • 0 – не инициализирован; • 1 – ввод; • 2 – отправлен; • 3 – идет обмен; • 4 – завершен.
responseText	Содержимое ответа.
responseXML	Если содержимое ответа представляет собой документ XML, 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.[a]
statusText	Текстовая строка сообщения о состоянии (статуса), полученная в ответе.

[a] *http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html#sec10*

Создав объект, давайте посмотрим, какие настройки нужно выполнить и как отправить запрос серверу.

8.1.2. Инициализация запроса

Прежде чем отправить запрос серверу, надо выполнить следующие действия:

1. Указать HTTP-метод выполнения запроса (POST или GET).
2. Указать URL ресурса на стороне сервера, которому будет направляться запрос.
3. Указать объекту XHR, как он должен информировать нас о ходе выполнения запроса.
4. Предоставить информацию, которая будет передаваться в теле запроса POST.

Первые два пункта этого списка выполняются вызовом метода `open()`:

```
xhr.open('GET','/some/resource/url');
```

Обратите внимание: этот метод *не* отправляет запрос серверу. Он просто настраивает URL и метод HTTP, которые должны использоваться для выполнения запроса. Кроме того, этому методу можно передать третий, необязательный параметр (логическое значение), который определяет, будет ли запрос выполняться асинхронно (значение `true`, по умолчанию) или синхронно (`false`). Редко когда возникают серьезные причины,

чтобы отказаться от асинхронных запросов (даже если это означает, что нам не придется возиться с функциями обратного вызова). В конце концов, асинхронная природа запроса – это неоспоримое преимущество.

Третий пункт списка означает, что мы 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*, объект XHR будет вызывать на разных стадиях обработки запроса. Просматривая значения различных свойств XHR, мы сможем точно определить, как протекает процесс выполнения запроса. Работу типичного обработчика события изменения состояния мы рассмотрим в следующем разделе.

Последний шаг инициализации запроса заключается в том, чтобы предоставить содержимое для запроса POST и отправить запрос серверу. Оба эти действия выполняются с помощью метода `send()`. Для запросов GET, обычно не имеющих содержимого, параметр с содержимым передается так:

```
xhr.send(null);
```

Когда отправляется запрос POST, передаваемая методу `send()` строка 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). Принципы такого преобразования выходят далеко за рамки этого раздела (с другой стороны, jQuery делает все необходимое без нашего участия), но если вам интересно, что это такое, выполните поиск в Сети по термину `encodeURIComponent`, и ваши усилия будут вознаграждены.

Пример такого вызова метода:

```
xhr.send('a=1&b=2&c=3');
```

Теперь познакомимся поближе с обработчиком события изменения состояния.

8.1.3. Слежение за ходом выполнения запроса

Объект XHR может информировать нас о ходе выполнения запроса с помощью *обработчика события изменения состояния*. Этот обработчик задается присваиванием свойству `onreadystatechange` объекта XHR ссылки на функцию, играющую роль обработчика.

После инициализации запроса вызовом метода `send()` эта функция вызывается несколько раз в процессе прохождения различных этапов выполнения запроса. Текущее состояние (статус) запроса доступно в виде числового кода в свойстве `readyState` (см. табл. 8.1).

Это, конечно, хорошо, но в большинстве случаев нам достаточно знать, выполнен ли запрос и успешно ли он выполнен. Поэтому чаще всего нам будет встречаться реализация обработчика, приведенная в листинге 8.2.

Листинг 8.2. Обработчик события, игнорирующий все промежуточные состояния, кроме полного завершения запроса

```
xhr.onreadystatechange = function() {
  if (xhr.readyState == 4) {
    if (xhr.status >= 200 &&
        xhr.status < 300) {
      // success
    }
    else {
      // error
    }
  }
}
```

Игнорировать все промежуточные состояния

Ветвление по статусу ответа

Успешное выполнение

Выполнение с ошибкой

Этот шаблон игнорирует все промежуточные состояния, кроме состояния завершения выполнения запроса, и по его наступлении проверяет значение свойства `status`, чтобы определить, был ли запрос выполнен успешно или нет. В спецификации протокола HTTP определено, что коды статуса в диапазоне от 200 до 299 означают успех, а значения от 300 и выше свидетельствуют о различных ошибках.

Следует заметить, что этот обработчик обращается к объекту XHR с помощью глобальной переменной. Но разве ссылка на объект должна передаваться обработчику в виде параметра?

Мы, конечно, могли бы этого *ожидать*, но так не происходит. Объект должен быть размещен каким-то другим способом, обычно для этого используется глобальная переменная. Это могло бы стать проблемой в случае, если бы нам понадобилось выполнить одновременно больше одного запроса. К счастью, поддержка Ajax в jQuery ловко справляется с этой проблемой.

Давайте посмотрим, как выполняется обработка ответа по завершении запроса.

8.1.4. Получение ответа

Как только обработчик события изменения состояния определит, что свойство `readyState` содержит признак успешного завершения запроса, мы можем извлечь содержимое ответа из объекта XHR.

Несмотря на название Ajax (где символ X происходит от XML), содержимое ответа может иметь произвольный текстовый формат – оно не ограничивается форматом XML. На самом деле, в большинстве случаев ответы на Ajax-запросы имеют формат, *отличный* от XML. Это может быть и обычный текст, и фрагмент HTML-разметки, это может быть даже текстовое представление объекта JavaScript или массива в формате JavaScript Object Notation (JSON).

Независимо от формата, содержимое ответа доступно через свойство `responseText` объекта XHR (при условии, что запрос завершился без ошибок). Если в ответе указано, что он имеет формат XML, за счет

включения заголовка типа содержимого, указывающего тип MIME *text/xml* (или любой другой тип MIME для формата XML), содержимое ответа будет интерпретироваться как документ XML. Доступ к полученному дереву элементов можно получить через свойство `responseXML`. После этого XML DOM можно обработать с помощью средств JavaScript (и jQuery, используя интерфейс селекторов).

Обработка документов XML на стороне клиента – не бог весть какая сложная задача, но даже с jQuery она может быть источником неудобств. Порой передать данные со сложной иерархической структурой позволяет только формат XML, тем не менее когда его возможности (и сопутствующая головная боль) не являются абсолютно необходимыми, авторы страниц зачастую стремятся применять другие форматы.

Однако и у других форматов есть свои недостатки. Если ответ приходит в формате JSON, его нужно преобразовать в эквивалентную конструкцию времени выполнения. Если ответ приходит в формате HTML, его нужно записать в соответствующий элемент. А что если полученная HTML-разметка содержит блоки `<script>`, которые надо выполнить? Мы не будем заниматься этими проблемами в данном разделе, потому что он не является полным руководством по технологии Ajax и, что важнее, потому что большинство этих проблем за нас решает библиотека jQuery, о чем мы вскоре и узнаем.

Диаграмма всего процесса выполнения запроса показана на рис. 8.1.

В этом кратком обзоре технологии Ajax мы выявили следующие неприятности, с которыми приходится сталкиваться авторам страниц:

- при создании объекта XHR необходимо учитывать характерные особенности броузера;

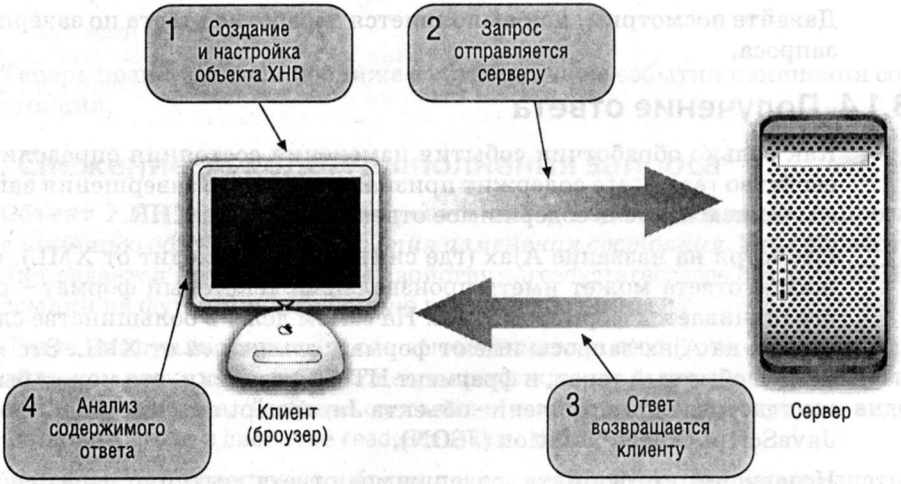


Рис. 8.1. Жизненный цикл запроса Ajax от его создания на стороне клиента до получения ответа от сервера

- обработчики события изменения состояния вынуждены отсеивать многие изменения состояния, не представляющие для нас никакого интереса;
- обработчики события изменения состояния не получают ссылку на вызывающие их объекты XHR;
- содержимое ответа приходится обрабатывать самыми разными способами, в зависимости от используемого формата.

В оставшейся части этой главы описаны команды и вспомогательные функции jQuery, которые упрощают (и делают более понятным) применение технологии Ajax в наших страницах. Прикладной интерфейс jQuery Ajax насчитывает много функций, и мы начнем знакомство с самых простых и наиболее часто используемых инструментов.

8.2. Загрузка содержимого в элемент

Пожалуй, чаще всего технология Ajax используется для получения от сервера фрагментов содержимого и добавления этих фрагментов в некоторые важные для нас места дерева DOM. Содержимое может быть фрагментом HTML-разметки или простым текстом, который затем станет содержимым требуемого элемента.

Предположим, нам требуется получить от сервера HTML-фрагмент, используя ресурс с именем `/serverResource`, и превратить его в содержимое элемента `<div>` со значением атрибута `id`, равным `someContainer`. Последний раз в этой главе мы посмотрим, как это можно сделать без помощи jQuery. В листинге 8.3 представлена реализация обработчика

Подготовка к опробованию примеров

В отличие от прочих приведенных выше примеров, программный код примеров этой главы требует наличия веб-сервера, который будет принимать запросы к ресурсам на стороне сервера. Поскольку обсуждение работы серверных механизмов выходит далеко за рамки этой книги, мы настроим лишь самые необходимые серверные ресурсы, которые будут отправлять данные обратно клиенту, не выполняя никаких настоящих действий. Мы будем воспринимать сервер как некий «черный ящик» – нам не надо знать, как он работает.

Чтобы обеспечить работу этих ресурсов, вам потребуется настроить веб-сервер любого типа. Для удобства серверные ресурсы были разработаны в двух форматах: в виде Java Server Pages (JSP) и некоторые из них – в виде сценариев на языке PHP. Ресурсы JSP можно использовать при условии, что у вас настроен (или вы предполагаете настроить) механизм сервлетов/JSP. Если ваш веб-сервер поддерживает PHP, можете использовать ресурсы PHP.

Если вы хотите работать с ресурсами JSP, но у вас нет подходящего настроенного сервера, в состав примеров к этой главе включены инструкции по настройке свободного веб-сервера Tomcat. Находятся они в файле *chapter8/tomcat.pdf*. И не волнуйтесь, это гораздо проще, чем вы думаете!

Загружаемые примеры программного кода к этой главе предполагают использование ресурсов JSP. Если вы пожелаете использовать ресурсы PHP, выполните поиск в примерах всех вхождений строки *.jsp* и замените их строкой *.php*. Обратите внимание: не все серверные ресурсы были переведены с JSP на PHP, но существующих ресурсов PHP должно быть достаточно, чтобы, зная PHP, можно было восполнить недостаток.

Как только сервер будет настроен, вы можете посетить страницу *http://localhost:8080/chapter8/test.jsp* (чтобы проверить корректность установки Tomcat) или *http://localhost/chapter8/test.php* (чтобы проверить корректность установки PHP). В данном случае предполагается, что в качестве корневого каталога документов веб-сервера Apache (или любого другого, по вашему выбору) настроен базовый каталог с примерами.

Увидев соответствующую тестовую страницу, вы будете готовы опробовать примеры этой главы.

события `onload` с применением шаблонов, представленных выше в этой главе. Полный код HTML-разметки этого примера вы найдете в файле *chapter8/listing.8.3.html*.

Листинг 8.3. Использование объекта XHR для включения фрагмента HTML в страницу

```
var xhr;

if (window.XMLHttpRequest) {
  xhr = new XMLHttpRequest();
}
else if (window.ActiveXObject) {
  xhr = new ActiveXObject("Msxml2.XMLHTTP");
}
else {
  throw new Error("Ajax is not supported by this browser");
}

xhr.onreadystatechange = function() {
  if (xhr.readyState == 4) {
    if (xhr.status >= 200 && xhr.status < 300) {
      document.getElementById('someContainer')
        .innerHTML = xhr.responseText;
    }
```

```
    }
}

xhr.open('GET','/serverResource');
xhr.send();
```

Здесь нет ничего сложного, но программный код получился немалым – приблизительно 19 строк, даже с учетом пустых строк, добавленных для удобочитаемости, и строки, которую пришлось принудительно разбить, чтобы уместить ее по ширине книжной страницы.

Эквивалентный программный код обработчика, который мы написали бы с использованием jQuery, выглядит так:

```
$('#someContainer').load('/serverResource');
```

Можем поспорить, что знаем, какой код написали бы вы! Давайте познакомимся с командами jQuery, которые мы использовали в этой инструкции.

8.2.1. Загрузка содержимого с помощью jQuery

Простая инструкция jQuery из предыдущего раздела легко справляется с загрузкой содержимого с сервера с помощью одной основной команды jQuery Ajax `load()`. Вот полное описание синтаксиса этой команды:

Синтаксис команды load

`load(url,parameters,callback)`

Инициирует запрос Ajax по заданному URL-адресу, возможно, с дополнительными параметрами. Можно указать функцию обратного вызова, обращение к которой произойдет по завершении запроса. Содержимое всех элементов в обернутом наборе будет замещено текстом ответа.

Параметры

`url`	(строка) URL-адрес ресурса на стороне сервера, которому отправляется запрос.
`parameters`	(объект) Объект, свойства которого преобразуются соответствующим образом в строку параметров, которая затем передается запросу. Если этот параметр определен, запрос выполняется методом POST, если отсутствует – методом GET.
`callback`	(функция) Функция обратного вызова, которая вызывается после того, как данные, полученные в ответе, будут загружены в элементы обернутого набора. В качестве параметра этой функции передается текст ответа, код статуса и объект XHR.

Возвращаемое значение

Обернутый набор.

Несмотря на простоту вызова этой команды, она обладает некоторыми важными особенностями. Например, если для определения парамет-

ров запроса используется параметр parameters, запрос выполняется методом POST HTTP, в противном случае инициируется запрос GET. Если потребуется выполнить запрос GET с параметрами, мы должны будем включить их в URL в виде строки запроса. Но при этом на наши плечи ложится вся ответственность за правильность оформления строки запроса и кодирование имен и значений параметров.

В большинстве случаев загрузку полного текста ответа в элементы обернутого набора мы будем выполнять с помощью команды load(), но иногда нам может потребоваться отфильтровать некоторые элементы, полученные в ответе. Для решения этой задачи jQuery позволяет определять селектор в строке URL, позволяющий определить элементы, которые должны быть загружены в обернутые элементы. Для этого к строке URL нужно добавить пробел и символ решетки (#), за которым должен следовать селектор.

Например, отфильтровать из ответа все элементы, не являющиеся экземплярами , позволит следующая инструкция:

```
$('.injectMe').load('/someResource #div');
```

Если параметры запроса определяются содержимым элементов формы, при сборке строки запроса можно воспользоваться удобной командой serialize(). Ее синтаксис:

Синтаксис команды serialize

`serialize()`

Создает правильно отформатированную и закодированную строку запроса из всех успешных элементов управления формы в обернутом наборе.

Параметры

Нет

Возвращаемое значение

Отформатированная строка запроса.

Команда serialize() выбирает информацию только из элементов формы, входящих в обернутый набор, и только из тех, которые считаются *успешными*. Успешный элемент – это элемент, который был бы отправлен серверу в процессе отправки формы согласно правилам, изложенным в спецификации HTML.[1] Такие элементы, как неотмеченные флажки и переключатели, списки, в которых не был выбран ни один из вариантов, и любые неактивные элементы не считаются успешными и не передаются серверу вместе с формой. Команда serialize() также проигнорирует их.

[1] *http://www.w3.org/TR/html401/interact/forms.html#h-17.13.2*

Если потребуется, собрать данные формы в виде массива JavaScript (в противоположность строке запроса) поможет метод `serializeArray()`.

Синтаксис команды serializeArray

`serializeArray()`

Собирает значения из всех успешных элементов формы в обернутом наборе в массив объектов, содержащих имена и значения элементов управления.

Параметры

Нет

Возвращаемое значение

Массив с данными формы.

Массив, возвращаемый командой `serializeArray()`, состоит из анонимных экземпляров объектов, каждый из которых содержит свойства `name` и `value` с именем и значением успешного элемента формы.

Вооружившись командой `load()`, давайте попробуем приспособить ее для решения задачи, с которой часто приходится сталкиваться веб-разработчикам.

8.2.2. Загрузка динамических данных

Нередко в бизнес-приложениях, особенно на сайтах интернет-магазинов, требуется получать данные с сервера в реальном масштабе времени, чтобы представить пользователям самую свежую информацию. В конце концов, никто не хочет вводить пользователей в заблуждение, предлагая им купить что-то, чего на самом деле нет.

В этом разделе мы начнем разработку страницы, которую будем расширять на протяжении всей главы. Эта страница – часть веб-сайта вымышленной фирмы The Boot Closet, которая занимается розничной продажей уцененных излишков спортивной 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 в этот день и что уже удалось продать. Поэтому для нас важно обеспечивать пользователей самой свежей информацией!

Для начала на нашей странице (без элементов навигации по сайту и прочих типичных элементов, чтобы все внимание сосредоточилось на предмете изучения) мы предоставим клиентам раскрывающийся список, содержащий доступные в настоящий момент модели обуви, а при выборе модели клиентом отображается подробная информация о ней. Сразу после открытия страница будет выглядеть, как показано на рис. 8.2.

После первой загрузки страницы на ней будут присутствовать раскрывающийся список с предварительно загруженным перечнем моделей

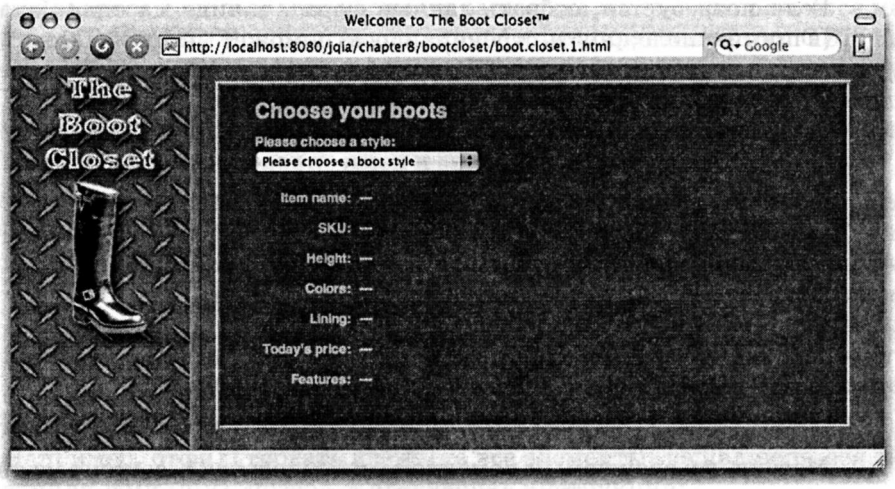


Рис. 8.2. Начальное состояние нашей страницы с информацией о модели

и поля с метками для отображения информации о выбранной модели. Пока модель не выбрана, в полях вместо данных будут выводиться прочерки.

Для начала определим HTML-разметку, которая создаст структуру страницы:

```
<body id="bootCloset1">
  <img id="banner" src="boot.closet.branding.png"/>
  <form action="" id="orderForm">
    <div id="detailFormContainer">
      <h1>Choose your boots</h1>
      <div>
        <label>Please choose a style:</label><br/>
        <select id="styleDropdown">
          <option value="">Please choose a boot style</option>
          <option value="7177382">Caterpillar Tradesman Work Boot</option>
          <option value="7269643">Caterpillar Logger 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>
        </select>
      </div>
      <div id="detailsDisplay"></div>
    </div>
  </form>
</body>
```

Совсем немного, правда?

Всю информацию о визуальном представлении мы определили во внешней таблице стилей, не включив в HTML-разметку никакие аспекты поведения с целью соответствовать принципам ненавязчивого JavaScript.

Раскрывающийся список с перечнем моделей был предварительно заполнен. Во всех примерах этой главы мы будем полагать, что приводим наше веб-приложение в движение с помощью ресурсов на стороне сервера. В конечном счете, взаимодействие с этими ресурсами – основная цель технологии Ajax. Поэтому даже при том, что в примере используется достаточно простой файл HTML, мы будем полагать, что он сгенерирован некоторым серверным ресурсом шаблонов, таким как JSP или PHP, и что данные о продукте динамически включаются в страницу из базы данных (или из другого хранилища данных).

Кроме того, в странице определен совершенно пустой контейнер `<div>` (со значением атрибута `id`, равным `detailsDisplay`), где будет размещаться подробная информация о модели. Мы полностью полагаемся на серверный ресурс шаблонов, который будет генерировать динамическое содержимое, поэтому нам не требуется определять его здесь *и* в странице JSP (или PHP) – определение структуры в двух местах потребовало бы следить за их соответствием друг другу, а это плохо!

Мы будем получать незаполненную версию содержимого с сервера во время загрузки страницы, поэтому данная структура должна быть определена всего в одном месте. Теперь рассмотрим наш обработчик события готовности документа.

```
$(function(){
  $('#styleDropdown')                    ❶ Обертывает элемент раскрывающегося списка
    .change(function(){                    и подключает обработчик события изменения
      var styleValue = $(this).val();
      $('#detailsDisplay').load(           ❷ Загружает данные
        'getDetails.jsp',                    для выбранной модели
        { style: styleValue }
      );
    })
    .change();                             ❸ Вызывает обработчик
                                             события изменения
});
```

В этом обработчике события готовности документа мы обертываем раскрывающийся список с перечнем моделей и подключаем к нему обработчик события `change` ❶. В функции-обработчике, вызываемой всякий раз, когда клиент изменит выбор, мы получаем значение выбора, применяя команду `val()` к ссылке `this`, которая в обработчике события представляет элемент `<select>`, вызвавший появление события. Далее мы применяем команду `load()` ❷ к элементу `detailsDisplay`, чтобы инициировать Ajax-запрос к серверному ресурсу `getDerails.jsp`, передавая команде значение, соответствующее выбранной модели, в качестве параметра `style`.

В конце обработчика события готовности документа мы вызываем команду `change()` ❸, чтобы запустить обработчик события изменения раскрывающегося списка. Благодаря этому выполняется запрос информации для модели по умолчанию "" (пустая строка), что заставляет серверный ресурс вернуть конструкцию, которая в окне броузера отобразится так, как показано на рис. 8.2.

После того как клиент выберет одну из доступных моделей обуви, страница будет выглядеть, как показано на рис. 8.3.

Самая примечательная операция, выполняемая в обработчике события готовности документа, – это применение команды `load()` для быстрой загрузки с сервера фрагмента HTML-разметки и размещение его в дереве DOM в качестве вложенного содержимого существующего элемента. Эта команда чрезвычайно удобна и прекрасно подходит для веб-приложений, приводимых в движение серверными механизмами, такими как JSP и PHP.

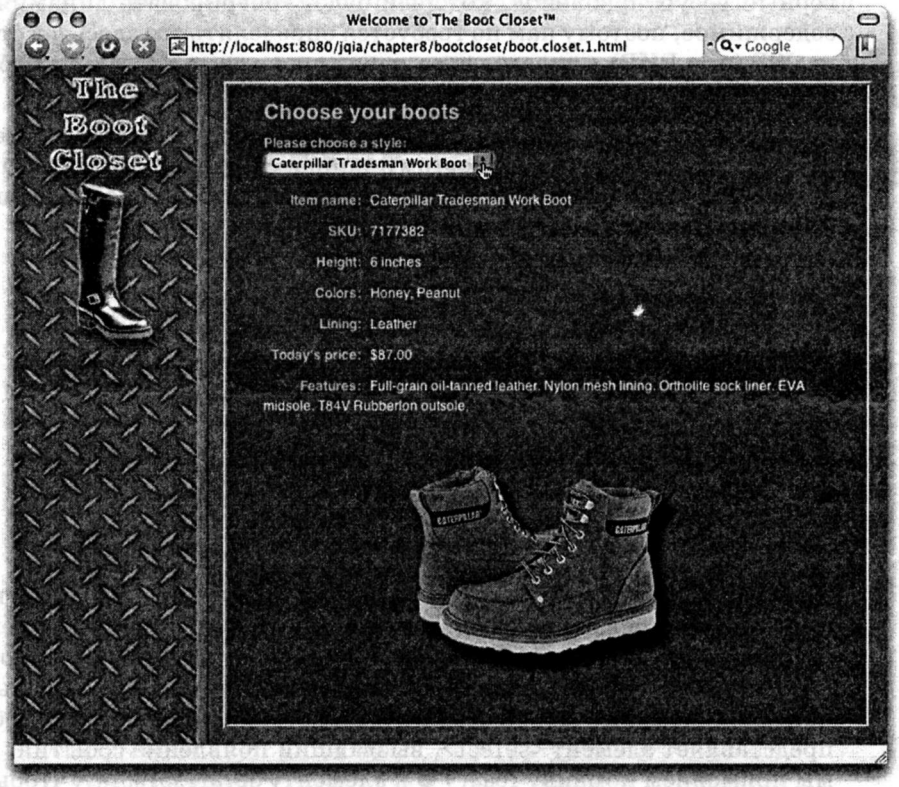


***Рис. 8.3.** Серверный ресурс возвращает предварительно сформированный фрагмент HTML-разметки для отображения информации о выбранной модели обуви*

В листинге 8.4 приведен полный код нашей страницы The Boot Closet, который находится в файле *chapter8/bootcloset/boot.closet.1.html*. На протяжении всей главы мы еще не раз вернемся к этой странице, чтобы дополнить ее новыми возможностями.

Листинг 8.4. Первая версия нашей страницы интернет-магазина The Boot Closet

```
<html>
  <head>
    <title>Welcome to The Boot Closet&trade;</title>
    <link rel="stylesheet" type="text/css" href="boot.closet.css">
    <script type="text/javascript"
            src="../../scripts/jquery-1.2.1.js"></script>
    <script type="text/javascript">
      $(function(){
        $('#styleDropdown')
          .change(function(){
            var styleValue = $(this).val();
            $('#detailsDisplay').load(
              'getDetails.jsp',
              { style: styleValue }
            );
          })
          .change();
      });
    </script>
  </head>

  <body id="bootCloset1">
    <img id="banner" src="boot.closet.branding.png"/>
    <form action="" id="orderForm">
      <div id="detailFormContainer">
        <h1>Choose your boots</h1>
        <div>
          <label>Please choose a style:</label><br/>
          <select id="styleDropdown">
            <option value="">Please choose a boot style</option>
            <option value="7177382">
              Caterpillar Tradesman Work Boot</option>
            <option value="7269643">Caterpillar Logger 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>
          </select>
        </div>
        <div id="detailsDisplay"></div>
      </div>
    </form>
```

```
    </body>
</html>
```

Команда `load()` очень удобна, когда требуется получить фрагмент HTML-разметки, чтобы наполнить элемент (или набор элементов). Но иногда нужно получить более полный контроль над тем, как выполняется запрос Ajax, или выполнить некоторые магические действия над полученными в теле ответа данными.

Продолжим наши исследования и посмотрим, что может предложить jQuery для этих более сложных ситуаций.

8.3. Выполнение запросов GET и POST

Команда `load()` выполняет запрос GET либо POST в зависимости от того, вызывается ли она с дополнительными параметрами запроса, но иногда бывает необходим чуть больший контроль над тем, каким методом HTTP выполняется запрос. Зачем это нам? Возможно, для того, чтобы соответствовать требованиям *сервера*.

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. Назначение каждого из методов состоит в следующем.

- *Запросы GET* соблюдают *идемпотентность* (тождественность). Состояние сервера и данные для приложения не должны изменяться под воздействием запросов GET. Один и тот же запрос GET, выполняясь снова и снова, должен возвращать в точности те же самые результаты (предполагается, что в это время не происходит ничего другого, что привело бы к изменению состояния сервера).
- *Запросы POST* могут быть *неидемпотентыми* (нетождественными). 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 использоваться 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, чтобы вызвать изменения, следует использовать метод POST.

Предупреждение

Это не просто теория. Броузеры 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 броузера, касающимися различных типов запросов.

Добавим только, что библиотека jQuery предоставляет в наше распоряжение несколько способов выполнения запросов GET, которые, в отличие от `load()`, реализованы не как команды jQuery, предназначенные для манипулирования обернутыми наборами. Различные виды запросов GET позволяют выполнять *вспомогательные функции*. Как уже отмечалось в главе 1, вспомогательные функции jQuery – это глобальные функции, определенные в пространстве имен `jQuery`, или `$`.

Давайте познакомимся с каждой из этих функций.

8.3.1. Получение данных с помощью jQuery

Если требуется получить некоторые данные с сервера и решить, что с ними делать (вместо того, чтобы позволить команде `load()` загрузить их в качестве содержимого элемента HTML), можно использовать вспомогательную функцию `$.get()`. Ее синтаксис:

Синтаксис команды $.get

`$.get(url, parameters, callback)`

Инициирует запрос GET к серверу, используя заданный URL-адрес и все параметры, передаваемые в виде строки запроса.

Параметры

`url`	(строка) URL-адрес ресурса на стороне сервера, которому отправляется запрос GET.
`parameters`	(объект | строка) Объект, из свойств которого (пар имя/значения) конструируется строка запроса, которая затем добавляется в конец URL, или предварительно сформированная и закодированная строка запроса.
`callback`	(функция) Функция, вызываемая после завершения запроса. Первым параметром этой функции передается текст ответа, вторым – код статуса.

Возвращаемое значение

Экземпляр XHR.

Рассмотрим простой пример использования этой функции, приведенный в листинге 8.5 (файл *chapter8/$.get.html*).

Листинг 8.5. Вспомогательная функция $.get( ) позволяет получить данные с сервера

```
<html>
  <head>
    <title>$.get() Example</title>
    <link rel="stylesheet" type="text/css" href="../common.css">
    <script type="text/javascript"
            src="../scripts/jquery-1.2.1.js"></script>
    <script type="text/javascript">
```

```
        $(function(){
          $('#testButton').click(function(){
            $.get(
              'reflectData.jsp',
              {a:1, b:2, c:3},
              function(data) { alert(data); }
            );
          });
        });
      </script>
    </head>

    <body>
      <button type="button" id="testButton">Click me!</button>
    </body>
  </html>
```

❶ Получение данных с сервера

В этом примере страницы присутствует кнопка, с помощью которой инициируется вызов функции `$.get()` ❶. Запрос GET отправляется серверному ресурсу `reflectData.jsp` (этот ресурс возвращает отрывок текста, который содержит значения параметров, переданных запросу) и содержит значения параметров запроса `a`, `b` и `c`. Функция обратного вызова извлекает эти данные и может делать с ними все, что угодно. В данном случае она просто вызывает диалог, в котором отображает эти данные.

Если эту страницу открыть в броузере и щелкнуть на кнопке, можно увидеть изображение (рис. 8.4).

Если ответ сервера будет содержать документ XML, он будет проанализирован, и в первом параметре функции будет передано дерево DOM.

Формат XML прекрасно подходит, если требуется высокая гибкость и данные имеют иерархическую структуру, но работать с ним очень

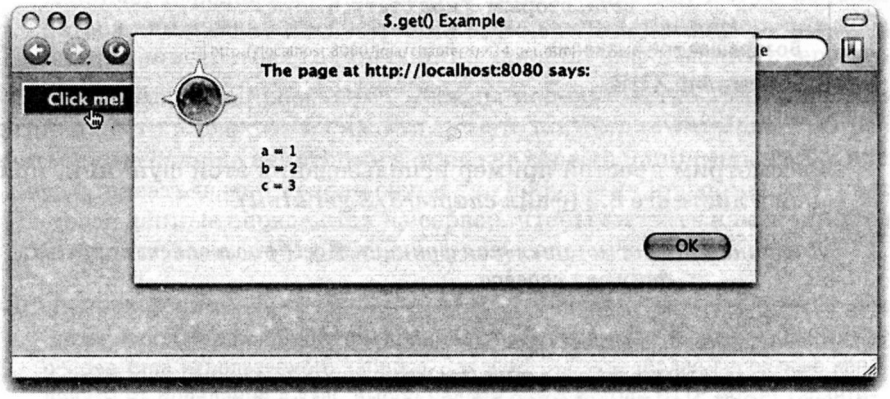


Рис. 8.4. Вспомогательная функция $.get( ) получает с сервера данные, которыми мы можем манипулировать по своему усмотрению, включая их отображение в диалоге alert

непросто. Давайте познакомимся с еще одной вспомогательной функцией, которая очень полезна, когда наши данные не предъявляют высоких требований.

8.3.2. Получение данных в формате JSON

Как отмечалось в предыдущем разделе, возвращаемый сервером документ XML автоматически анализируется и полученное дерево DOM становится доступно функции обратного вызова. Если применение формата XML не является насущной необходимостью или по каким-то причинам не подходит для передачи данных, вместо него часто используется формат JSON. Одна из причин – с форматом JSON очень просто работать в клиентских сценариях, а jQuery еще больше упрощает дело.

В случаях, когда заранее известно, что ответ будет иметь формат JSON, можно использовать вспомогательную функцию `$.getJSON()`, которая автоматически проанализирует полученную строку JSON, а полученные данные передаст функции обратного вызова. Эта вспомогательная функция имеет следующий синтаксис:

Синтаксис команды $.getJSON

`$.getJSON(url, parameters, callback)`

Инициирует запрос GET к серверу, используя заданный URL-адрес 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.

Параметры

`url`	(строка) URL-адрес ресурса на стороне сервера, которому отправляется запрос GET.
`parameters`	(объект | строка) Объект, из свойств которого (пар имя/значения) конструируется строка запроса, которая затем добавляется в конец URL, или предварительно сформированная и закодированная строка запроса.
`callback`	(функция) Функция, вызываемая после завершения запроса. Первым параметром этой функции передаются данные, полученные в результате синтаксического разбора строки JSON, вторым – код статуса.

Возвращаемое значение

Экземпляр XHR.

Эта функция прекрасно подходит для случаев, когда нужно получить данные от сервера без лишних сложностей, присущих формату XML. Давайте рассмотрим пример, который демонстрирует применение этой функции на практике.

Загрузка каскадов раскрывающихся списков

При создании полнофункциональных интернет-приложений нередко требуется установить для выбора в раскрывающемся списке варианты, значения которых зависят от параметров некоторого другого элемента управления, зачастую такого же раскрывающегося списка. Типичный пример – когда выбор некоторого штата или области в одном списке вызывает загрузку перечня городов, расположенных в этом штате или области, в другой список.

Такой набор элементов управления называется *зависимыми списками*, иногда *каскадными списками*, и давно уже стал второй эмблемой Ajax – он используется в качестве примера практически в любой книге об Ajax, а также во многих учебных материалах в Сети. В этом разделе мы посмотрим, как решается эта классическая проблема, создав набор элементов, усиленных вспомогательной функцией jQuery, – `$.getJSON()`.

Для демонстрации этого примера вернемся к странице The Boot Closet, созданной в разделе 8.2.2, и дополним ее новыми возможностями. Первоначально эта страница позволяла клиентам определять, какие модели обуви имеются в наличии, но у них не было возможности выбрать пару для покупки. Нам нравится делать покупки, поэтому на следующем шаге мы добавим элементы управления, позволяющие выбрать цвет и размер.

Не забывайте, что наш бизнес – распродажа, а это означает, что в наличии не всегда есть вся линейка каждой модели обуви. Каждый день у нас бывает обувь только определенных цветов и размеров. Поэтому мы не можем жестко определить списки цветов и размеров – нам требуется получать эти списки динамически, из базы данных.

Чтобы позволить клиентам выбирать цвет и размер, добавим в форму еще два раскрывающихся списка: один – для выбора цвета, а другой – размера. Начальное состояние улучшенной формы показано на рис. 8.5, ее исходный код находится в файле *chapter8/bootcloset/boot.closet.2.html*.

Раскрывающийся список выбора модели обуви находится в активном состоянии (и содержит предварительно загруженный перечень доступных моделей, как было сказано выше), но элементы управления для выбора цвета и размера не содержат ничего и находятся в неактивном состоянии. Мы не можем заранее заполнить эти списки, потому что не знаем, какие цвета отображать в списке, пока клиент не выберет модель. Мы также не знаем, какие размеры отображать, пока не будут выбраны модель и цвет.

С учетом всего этого нам нужно, чтобы:

- после выбора модели раскрывающийся список выбора цвета переводился в активное состояние и заполнялся цветами, доступными для выбранной модели;

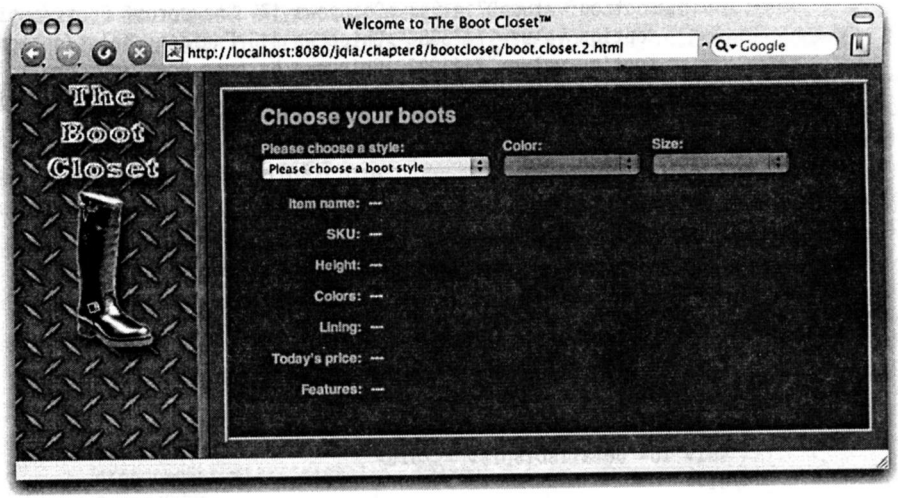


Рис. 8.5. Начальное состояние формы заказа с взаимозависимыми раскрывающимися списками, пустыми и неактивными

- после выбора модели и цвета раскрывающийся список выбора размера переводился в активное состояние и заполнялся размерами, доступными для выбранной модели указанного цвета;
- раскрывающиеся списки всегда находились в непротиворечивом состоянии друг относительно друга: мы обязаны гарантировать, что независимо от действий пользователя, оформляющего заказ, отсутствующая на складе комбинация модели, цвета и размера никогда не будет показана в форме.

Итак, засучим рукава и примемся за работу.

Для начала добавим дополнительные элементы в HTML-разметку тела страницы. Ниже приведен код HTML новой страницы, изменения выделены жирным шрифтом:

```
<body id="bootCloset2">
  <img id="banner" src="boot.closet.branding.png"/>
  <form action="" id="orderForm">
    <div id="detailFormContainer">
      <h1>Choose your boots</h1>
      <div id="cascadingDropdowns">
        <div>
          <label>Please choose a style:</label><br/>
          <select id="styleDropdown">
            <option value="">Please choose a boot style</option>
            <option value="7177382">
              Caterpillar Tradesman Work Boot</option>
            <option value="7269643">Caterpillar Logger 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>
          </select>
        </div>
        <div>
          <label>Color:</label><br/>
          <select id="colorDropdown" disabled="disabled"></select>
        </div>
        <div>
          <label>Size:</label><br/>
          <select id="sizeDropdown" disabled="disabled"></select>
        </div>
      </div>
      <div id="detailsDisplay"></div>
    </div>
  </form>
</body>
```

Мы изменили значение атрибута id в теге <body> (в первую очередь для того, чтобы мы могли использовать его как ключ внутри таблицы стилей CSS, применяемой к различным версиям страницы) и добавили два пустых неактивных раскрывающихся списка.

Чтобы добавить новое поведение, связанное с этими элементами управления, мы также должны расширить обработчик события готовности документа. Вот код нового обработчика (изменения также выделены жирным шрифтом):

```
$(function(){
  $('#styleDropdown')
    .change(function(){
      var styleValue = $(this).val();
      $('#detailsDisplay').load(
        'getDetails.jsp',
        { style: styleValue }
      );
      adjustColorDropdown();
    })
    .change();
  $('#colorDropdown')
    .change(adjustSizeDropdown);
});
```

❶ Выполнить коррекцию состояния списка цветов

❷ Подключить обработчик события изменения списка цветов

Изменения небольшие, но важные. Во-первых, внутри обработчика изменения раскрывающегося списка моделей вызывается функция adjustColorDropdown() ❶. Она скорректирует содержимое и состояние раскрывающегося списка цветов в соответствии с выбранной моделью.

Затем к новому раскрывающемуся списку цветов ❷ подключается обработчик события, который скорректирует содержимое и состояние

раскрывающегося списка размеров, когда будет выбран цвет. В качестве обработчика будет использована функция `adjustSizeDropdown()`.

Пока все достаточно просто, но нам еще предстоит написать функции, изменяющие состояние и содержимое зависимых раскрывающихся списков. Для начала займемся раскрывающимся списком цветов и определим функцию `adjustColorDropdown()`.

```
function adjustColorDropdown() {
  var styleValue = $('#styleDropdown').val();
  var dropdownSet = $('#colorDropdown');
  if (styleValue.length == 0) {
    dropdownSet.attr("disabled",true);
    dropdownSet.emptySelect();
    adjustSizeDropdown();
  }
  else {
    dropdownSet.attr("disabled",false);
    $.getJSON(
      'getColors.jsp',
      {style:styleValue},
      function(data){
        $(dropdownSet).loadSelect(data);
        adjustSizeDropdown();
      }
    );
  }
}
```

❶ Активизирует или деактивизирует список цветов

❷ Очищает список в неактивном состоянии и очищает зависимый список

❸ Получает список цветов на основе выбранной модели

❹ Корректирует зависимый раскрывающийся список

После получения значения раскрывающегося списка с перечнем моделей и его сохранения в переменной `styleValue` для последующего использования формируется обернутый набор, содержащий раскрывающийся список цветов, и сохраняется в переменной `dropdownSet`. Мы будем неоднократно ссылаться на этот набор, и было бы нежелательно всякий раз, когда он понадобится, тратить время на его повторное создание.

Затем, в зависимости от значения раскрывающегося списка с перечнем моделей, принимается решение об активизации или деактивизации списка цветов ❶ – если значение пустое, список деактивизируется, в противном случае – активизируется. Если список цветов оказался в неактивном состоянии, он еще и очищается командой `emptySelect()` ❷.

Минуточку! Что еще за команда `emptySelect()`?

Не волнуйтесь, не нужно листать предыдущие главы в поисках команды! Такой команды нет, по крайней мере, пока. Эта команда – из тех, что мы реализуем в следующем разделе. А пока заметим, что эта команда удаляет из раскрывающегося списка цветов все варианты выбора.

После деактивизации и очистки списка цветов вызывается функция `adjustSizeDropdown()` ❷, корректирующая состояние и содержимое зависимого списка размеров.

Если раскрывающийся список цветов находится в активном состоянии, его нужно заполнить значениями, которые соответствуют значению, выбранному в раскрывающемся списке моделей. Для получения этих значений воспользуемся услугами функции `$.getJSON()` ❸, указав ей имя серверного ресурса `getColors.jsp` и передав информацию о выбранной модели в виде параметра запроса с именем `style`.

При обращении к функции обратного вызова (после того, как в ответ на запрос будет получен ответ) первым параметром этой функции будет передано значение JavaScript, вычисленное в результате интерпретации ответа в формате JSON. Вот типичная конструкция JSON, возвращаемая ресурсом `getColors.jsp`:

```
[
  {value:'',caption:'choose color'},
  {value:'bk',caption:'Black Oil-tanned'},
  {value:'br',caption:'Black Polishable'}
]
```

Таким способом определяется массив объектов, каждый из которых обладает двумя свойствами – `value` и `caption`. Эти объекты определяют значение и отображаемый текст вариантов выбора, которые должны быть добавлены в раскрывающийся список цветов. Список заполняется путем передачи вычисленного значения JSON команде `loadSelect()`, применяемой к раскрывающемуся списку. Да, `loadSelect()` – это еще одна команда, которую мы напишем самостоятельно.

Наконец, всякий раз, когда изменяется значение в раскрывающемся списке цветов, список размеров должен быть приведен в соответствие этому новому значению, поэтому вызывается функция `adjustSizeDropdown()` ❹, которая выполнит серию операций над раскрывающимся списком размеров, как в случае раскрывающегося списка цветов.

Определение функции `adjustSizeDropdown()`:

```
function adjustSizeDropdown() {
  var styleValue = $('#styleDropdown').val();
  var colorValue = $('#colorDropdown').val();
  var dropdownSet = $('#sizeDropdown');
  if ((styleValue.length == 0)||(colorValue.length == 0) ) {
    dropdownSet.attr("disabled",true);
    dropdownSet.emptySelect();
  }else {
    dropdownSet.attr("disabled",false);
    $.getJSON(
      'getSizes.jsp',
      {style:styleValue,color:colorValue},
      function(data){dropdownSet.loadSelect(data)}
    );
  }
}
```

Ничего удивительного в том, что по структуре эта функция поразительно похожа на функцию `adjustColorDropdown()`. Помимо воздействия на раскрывающийся список размеров, а не на список цветов, *обе* проверяют значение раскрывающихся списков моделей и цветов, чтобы решить – активизировать или деактивизировать раскрывающийся список размеров. В данном случае раскрывающийся список размеров активизируется, только если оба эти списка активны.

Если список активизируется, то загружаются доступные размеры – опять же, с помощью функции `$.getJSON()` – из серверного ресурса `getSizes.jsp`, которому передаются номер модели и цвет.

Все эти функции обеспечивают оперативное взаимодействие наших каскадных раскрывающихся списков. На рис. 8.5 мы видели страницу сразу после ее открытия, а на рис. 8.6 отображена соответствующая часть страницы после выбора модели (вверху) и после выбора цвета (внизу).

Полный исходный код страницы приведен в листинге 8.6 и в файле *chapter8/bootcloset/boot.closet.2.html*.

Листинг 8.6. Страница The Boot Closet, дополненная каскадными раскрывающимися списками

```
<html>
  <head>
    <title>Welcome to The Boot Closet&trade;</title>
    <link rel="stylesheet" type="text/css" href="boot.closet.css">
    <script type="text/javascript"
            src="../../scripts/jquery-1.2.1.js"></script>
```

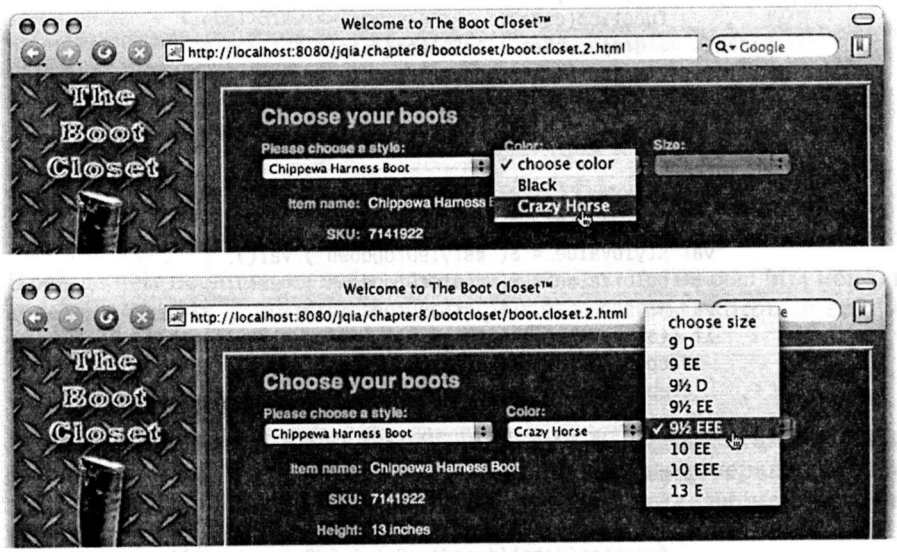


***Рис. 8.6.** Выбор модели активизирует список выбора цвета (вверху), а выбор цвета активизирует список выбора размера (внизу)*

```
<script type="text/javascript"
        src="jquery.jqia.selects.js"></script>
<script type="text/javascript">
  $(function(){
    $('#styleDropdown')
      .change(function(){
        var styleValue = $(this).val();
        $('#detailsDisplay').load(
          'getDetails.jsp',
          { style: styleValue }
        );
        adjustColorDropdown();
      })
      .change();
    $('#colorDropdown')
      .change(adjustSizeDropdown);
  });

  function adjustColorDropdown() {
    var styleValue = $('#styleDropdown').val();
    var dropdownSet = $('#colorDropdown');
    if (styleValue.length == 0) {
      dropdownSet.attr("disabled",true);
      dropdownSet.emptySelect();
      adjustSizeDropdown();
    } else {
      dropdownSet.attr("disabled",false);
      $.getJSON(
        'getColors.jsp',
        {style:styleValue},
        function(data){
          dropdownSet.loadSelect(data);
          adjustSizeDropdown();
        }
      );
    }
  }

  function adjustSizeDropdown() {
    var styleValue = $('#styleDropdown').val();
    var colorValue = $('#colorDropdown').val();
    var dropdownSet = $('#sizeDropdown');
    if ((styleValue.length == 0)||(colorValue.length == 0) ) {
      dropdownSet.attr("disabled",true);
      dropdownSet.emptySelect();
    }else {
      dropdownSet.attr("disabled",false);
      $.getJSON(
        'getSizes.jsp',
        {style:styleValue,color:colorValue},
        function(data){dropdownSet.loadSelect(data)}
      );
    }
```

```
    }
  </script>
</head>
<body id="bootCloset2">
  <img id="banner" src="boot.closet.branding.png"/>
  <form action="" id="orderForm">
    <div id="detailFormContainer">
      <h1>Choose your boots</h1>
      <div id="cascadingDropdowns">
        <div>
          <label>Please choose a style:</label><br/>
          <select id="styleDropdown">
            <option value="">Please choose a boot style</option>
            <option value="7177382">
              Caterpillar Tradesman Work Boot</option>
            <option value="7269643">Caterpillar Logger 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>
          </select>
        </div>
        <div>
          <label>Color:</label><br/>
          <select id="colorDropdown" disabled="disabled"></select>
        </div>
        <div>
          <label>Size:</label><br/>
          <select id="sizeDropdown" disabled="disabled"></select>
        </div>
      </div>
      <div id="detailsDisplay"></div>
    </div>
  </form>
</body>
</html>
```

Прежде чем поздравить себя, заметим, что это еще не все! Мы использовали в наших функциях нестандартные команды, которые еще не написаны! Так что давайте еще поработаем.

Создание нестандартных команд

В нашем примере с каскадными списками над элементами <select> (раскрывающиеся списки) должны выполняться две операции: удаление из списка всех вариантов выбора и его заполнение вариантами, определенными в конструкции JavaScript.

Мы решили реализовать эти операции в виде команд jQuery по двум основным причинам.

- Эти операции должны выполняться в нескольких местах страницы, поэтому нужно, как минимум, выделить их в отдельные функции, чтобы избежать дублирования программного кода.
- Эти достаточно общие операции могут быть полезными в других страницах сайта и даже в других веб-приложениях, поэтому есть смысл определить их как команды jQuery().

Реализуем сначала команду `emptySelect()`.

```
$.fn.emptySelect = function() {
  return this.each(function(){
    if (this.tagName=='SELECT') this.options.length = 0;
  });
}
```

Как добавлять новые команды jQuery, мы узнали в главе 7 и теперь применяем эти знания, чтобы дополнить `$.fn` новой функцией с именем `emptySelect()`. Вспомните: когда вызывается такая функция, в контексте функции (`this`) она получает обернутый набор. С помощью команды `each()` мы выполняем обход всех элементов набора, вызывая функцию-итератор, заданную как параметр команды `each()`.

Внутри *этой* функции контекст функции представлен отдельным элементом для текущей итерации. Мы проверяем этот элемент, чтобы убедиться в том, что имеем дело с элементом `<select>`, игнорируя элементы всех других типов, и устанавливаем размер массива `options` в значение 0. Такой способ удаления вариантов выбора из раскрывающихся списков поддерживается всеми типами броузеров.

Обратите внимание: мы возвращаем обернутый набор в качестве значения функции, гарантируя тем самым возможность использования этой команды в составе любых цепочек команд jQuery.

Достаточно просто! А теперь перейдем к команде `loadSelect()`.

Мы добавим в пространство имен `$.fn` следующую функцию:

```
$.fn.loadSelect = function(optionsDataArray) {
  return this.emptySelect().each(function(){
    if (this.tagName=='SELECT') {
      var selectElement = this;
      $.each(optionsDataArray,function(index,optionData){
        var option = new Option(optionData.caption,
                                optionData.value);
        if ($.browser.msie) {
          selectElement.add(option);
        }
        else {
          selectElement.add(option,null);
        }
      });
    }
  });
}
```

Эта команда немного сложнее.

Она ожидает получить единственный параметр – конструкцию JavaScript, как было определено в предыдущем разделе – массив объектов, каждый из которых обладает свойствами `value` и `name`, определяющими варианты выбора для добавления в элемент `<select>`.

Здесь мы снова выполняем обход всех элементов в обернутом наборе, но перед этим очищаем все элементы `<select>` в наборе вызовом команды `emptySelect()`, определенной выше. Тем самым перед добавлением новых вариантов выбора мы удалим все имеющиеся.

Внутри функции-итератора мы проверяем имя тега и игнорируем все элементы, не являющиеся элементами `<select>`. Для элементов, переживших эту проверку, выполняем обход переданного команде массива с данными, создаем новый экземпляр `Option` для каждого элемента массива и добавляем его в элемент `<select>`.

Операция добавления имеет один недостаток – она должна выполняться тем или иным способом, который поддерживается броузером. Стандарты W3C определяют метод `add()`, таким образом, чтобы добавить вариант выбора в конец содержимого элемента `<select>`, вторым параметром методу `add()` следует передать значение `null`. Мы полагали, что это легко сделать, просто опустив второй параметр:

```
selectElement.add(option);
```

Но в жизни все не так просто. Броузеры Opera и Safari прекрасно работают независимо от того, был ли передан этот параметр явно или неявно, но в броузерах на базе Mozilla, если второй параметр опущен, то возникает исключение, сообщающее о нехватке входных параметров. Однако добавление явного значения `null` в качестве второго параметра здесь не помогает, потому что в этом случае Internet Explorer перестает добавлять новый вариант в список.

Ловушка 22!

Поскольку нет никакого способа выполнить эту операцию, одинаково хорошо работающего во всех броузерах, а также из-за невозможности применить методику *обнаружения объекта* мы были вынуждены прибегнуть к методике определения типа броузера. Хорошо хоть, что у нас есть вспомогательные флаги `$.browser`, упрощающие это определение.

Еще раз обратите внимание: в качестве результата функция возвращает обернутый набор.

Полная реализация этих команд находится в файле *chapter8/bootcloset/jquery.jqia.selects.js* и приведена в листинге 8.7.

Листинг 8.7. Реализация нестандартных команд для работы с элементами <select>

```
(function($) {
  $.fn.emptySelect = function() {
```

```
    return this.each(function(){
      if (this.tagName=='SELECT') this.options.length = 0;
    });
  }

  $.fn.loadSelect = function(optionsDataArray) {
    return this.emptySelect().each(function(){
      if (this.tagName=='SELECT') {
        var selectElement = this;
        $.each(optionsDataArray,function(index,optionData){
          var option = new Option(optionData.caption,
                                  optionData.value);
          if ($.browser.msie) {
            selectElement.add(option);
          }
          else {
            selectElement.add(option,null);
          }
        });
      }
    });
  }
})(jQuery);
```

Функции `$.get()` и `$.getJSON()` библиотеки jQuery предоставляют нам мощные инструменты для работы с запросами GET. Но на запросах GET свет клином не сошелся!

8.3.3. Выполнение запросов POST

«Иногда вам хочется орешков, а иногда – нет». То, что помогает сделать выбор между батончиками «Натс» и «Твикс», имеет значение и при выполнении запросов к серверу. В одном случае мы хотим выполнить запрос GET, в другом хотим (или должны) выполнить запрос POST.

Можно привести массу причин предпочесть метод POST методу GET. Во-первых, согласно спецификации протокола http, метод POST следует использовать для выполнения всех неидемпотентных запросов. То есть если ваш запрос может вызвать изменения состояния на стороне сервера, следует выбрать метод POST (по крайней мере, тем, кто стремится следовать положениям спецификации). С другой стороны, согласно общепринятой практике и соглашениям, допускается использовать метод POST в случае, когда объем передаваемых данных слишком велик и его нельзя передать серверу в строке запроса URL – это ограничение зависит от используемого броузера. Иногда серверный ресурс, с которым мы взаимодействуем, поддерживает исключительно запросы POST или даже может выполнять различные операции в зависимости от того, какой запрос использован, – POST или GET.

Для случаев, когда желательно или необходимо выполнять запросы методом POST, jQuery предоставляет вспомогательную функцию `$.post()`,

которая действует точно так же, как функция `$.get()`, за исключением используемого метода протокола HTTP. Ее синтаксис:

Синтаксис команды $.post

`$.post(url, parameters, callback)`

Инициирует запрос POST к серверу, используя заданный URL-адрес и все параметры, передаваемые в теле запроса.

Параметры

`url`	(строка) URL-адрес ресурса на стороне сервера, которому отправляется запрос POST.
`parameters`	(объект | строка) Объект, из свойств которого (пар имя/значения) конструируется тело запроса, или предварительно сформированная и закодированная строка запроса.
`callback`	(функция) Функция, вызываемая после завершения запроса. Первым параметром этой функции передается текст ответа, вторым – код статуса.

Возвращаемое значение

Экземпляр XHR.

Между вызовами команды `load()` и различных функций jQuery Ajax, выполняющих запросы методами GET и POST, мы можем в некоторой степени влиять на то, как инициируется запрос, и на способ получения информации о результатах выполнения запроса. Но иногда нам необходим *полный* контроль на запросами Ajax. В библиотеке jQuery есть средства, позволяющие нам быть настолько требовательными, насколько захотим сами.

8.4. Полное управление запросами Ajax

Функции и команды, с которыми мы уже познакомились, удобны в большинстве случаев, но в некоторые моменты нам нужна полная власть над всем происходящим.

В этом разделе мы посмотрим, как jQuery обеспечивает нам это преимущество.

8.4.1. Выполнение запросов Ajax со всеми настройками

Для случаев, когда нам требуется полностью управлять выполнением запросов Ajax, jQuery предоставляет универсальную вспомогательную функцию `$.ajax()` для выполнения Ajax-запросов. Прочие команды и функции библиотеки jQuery, основанные на технологии Ajax, в конечном счете для инициации запроса используют именно эту функцию. Ее синтаксис:

Синтаксис команды $.ajax

`$.ajax(options)`

Инициирует Ajax-запрос, используя переданные ей параметры для управления передачей запроса и обращения к функции обратного вызова.

Параметры

`options` (объект) Объект, свойства которого определяют параметры выполнения операции. Подробности приведены в табл. 8.2.

Возвращаемое значение

Экземпляр XHR.

Казалось бы, все просто? Но не спешите с выводами. Аргумент `options` может определять широкий диапазон значений для настройки действий, выполняемых функцией. Эти параметры (в порядке убывания частоты их использования) приведены в табл. 8.2.

Таблица 8.2. Параметры вспомогательной функции $.ajax( )

Имя	Тип	Описание
`url`	строка	URL-адрес запроса.
`type`	строка	Применяемый метод HTTP, обычно POST или GET. Если отсутствует, по умолчанию используется метод GET.
`data`	объект	Объект, свойства которого служат параметрами запроса и передаются вместе с запросом. Если запрос выполняется методом GET, эти данные передаются в виде строки запроса. Если запрос выполняется методом POST, данные передаются в теле запроса. В любом случае функция `$.ajax` кодирует значения.
`dataType`	строка	Ключевое слово, идентифицирующее тип данных, которые, как ожидается, будут получены в ответе. Это значение определяет, каким образом должны быть обработаны данные, если это потребуется, прежде чем они 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. • `json` – текст ответа анализируется как строка JSON, функции обратного вызова передается полученный объект. • `jsonp` – напоминает формат `json`, за исключением того, что допускает удаленное создание сценариев (предполагается, что сервер поддерживает данную возможность).

Имя	Тип	Описание
dataType	строка	• 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. Если этот параметр опущен, текст ответа передается функции обратного вызова без какой-либо предварительной обработки.
timeout	число	Устанавливает предельное время ожидания ответа на запрос (в миллисекундах). Если запрос не завершен в течение указанного времени, его выполнение прерывается с вызовом функции обработки ошибок error (если определена).
global	логическое значение	Разрешает (true) или запрещает (false) вызов так называемых глобальных функций. Это функции, которые могут быть присоединены к элементам и вызываться на различных этапах или при различных условиях выполнения запроса Ajax. Мы подробно рассмотрим их в разделе 8.4.3. Если опущен, по умолчанию вызов глобальных функций разрешен.
contentType	строка	Тип содержимого в запросе. Если опущен, по умолчанию предполагается тип *application/x-www-form-urlencoded*; этот же тип используется по умолчанию при отправке форм.
success	функция	Функция, вызываемая в случае, если код статуса в ответе сообщает об успехе. Тело ответа передается этой функции в виде первого параметра, предварительно пройдя обработку в соответствии со значением параметра dataType. Вторым параметром функции передается код статуса, который (в данном случае всегда) сообщает об успехе.
error	функция	Функция, вызываемая в случае, если код статуса в ответе сообщает об ошибке. Функции передаются три аргумента: экземпляр XHR, строка сообщения о состоянии (в данном случае всегда *error*) и необязательный объект-исключение, возвращаемый экземпляром XHR.
complete	функция	Функция, вызываемая по завершении запроса. Получает два аргумента: экземпляр XHR и строку сообщения о состоянии – *success* либо *error*. Если также заданы функции success и error, данная функция будет вызвана после них.
beforeSend	функция	Функция, вызываемая перед инициацией запроса. Ей передается экземпляр XMLHttpRequest и может использоваться для установки дополнительных заголовков или выполнения других предварительных операций.

Таблица 8.2 (продолжение)

Имя	Тип	Описание
async	логическое значение	Если задано значение false, запрос выполняется как синхронный. По умолчанию выполняется асинхронный запрос.
processData	логическое значение	Если задано значение false, 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*.
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, заголовок не проверяется.

Достаточно много параметров, чтобы запомнить их все, однако, скорее всего, в отдельном запросе их будет всего несколько. Но даже в этом случае – разве не удобнее было бы установить свои значения по умолчанию для тех параметров, которые предполагается задействовать для выполнения многих запросов?

8.4.2. Настройка запросов, используемых по умолчанию

Очевидно, что последний вопрос в предыдущем разделе был руководством к действию. Как вы уже наверняка догадались, библиотека jQuery предоставляет способ определить значения свойств Ajax по умолчанию, которые будут использоваться, если их не переопределить явно. Это поможет упростить страницы, выполняющие множество однотипных запросов Ajax.

Функция для установки значений по умолчанию называется `$.ajaxSetup()` и имеет следующий синтаксис:

Синтаксис команды $.ajaxSetup

`$.ajaxSetup(properties)`

Устанавливает переданный набор свойств как значения по умолчанию для всех последующих вызовов функции `$.ajax()`.

Параметры

`properties` (объект) Объект, свойства которого определяют значения свойств Ajax по умолчанию. Это те же самые свойства, которые используются функцией `$.ajax()` и описаны в табл. 8.2.

Возвращаемое значение

Не определено.

Эту функцию можно использовать в любом месте сценария, обычно при загрузке страницы (но по желанию автора это может быть любое

другое место) для установки значений по умолчанию, применяемых для всех последующих вызовов функции `$.ajax()`.

Примечание

Набор значений по умолчанию, задаваемый этой функцией, не применяется к команде `load()`. Для вспомогательных функций `$.get()` и `$.post()` нельзя переопределить используемый ими метод HTTP. Установка параметра `type` в значение `GET` не приведет к тому, что функция `$.post()` будет использовать метод GET.

Предположим, мы настраиваем страницу, где для большей части запросов (выполняемых функциями, отличными от команды `load()`) требуется определить некоторые значения по умолчанию, чтобы не определять их при каждом вызове. Тогда первой инструкцией в блоке `<script>` можно записать, например, следующую:

```
$.ajaxSetup({
  type: 'POST',
  timeout: 5000,
  dataType: 'html',
  error: function(xhr) {
           $('#errorDisplay)
           .html('Error: ' + xhr.status + ' ' + xhr.statusText);
         }
})
```

Эта инструкция гарантировала бы, что каждый последующий запрос (опять же, это не относится к команде `load()`) по умолчанию будет использовать эти значения, если не переопределить их явно в параметрах, передаваемых используемым вспомогательным функциям Ajax. Обратите внимание на функцию обратного вызова, которая по умолчанию используется для обработки ошибок. Такое определение вполне приемлемо, поскольку функции обратного вызова `error`, `complete` и даже `beforeSend` применяются ко всем вызовам Ajax.

А теперь поговорим о *глобальных функциях*, которыми управляет параметр `global`.

8.4.3. Глобальные функции

В дополнение к возможности указать функции по умолчанию, вызываемые для всех запросов Ajax, установив их с помощью `$.ajaxSetup()`, jQuery позволяет нам подключать функции к конкретным элементам DOM. Эти функции будут вызываться на различных этапах обработки запроса или когда он полностью завершился, успешно или с ошибкой.

Например, чтобы подключить функцию к элементу со значением атрибута `id`, равным `errorConsole`, предназначенному для вывода сообщений об ошибках, можно написать

```
$('#errorConsole').ajaxError(reportError);
```

Функция `reportError` будет вызываться по событию, когда какой-либо запрос Ajax будет завершаться с ошибкой.

При вызове этой или любой другой подобной функции первым параметром ей передается экземпляр объекта JavaScript – `Object`, со следующими двумя свойствами:

- `type` – строка, содержащая тип глобальной функции, например *ajaxError*;
- `target` – ссылка на элемент DOM, к которому подключена глобальная функция. Для предыдущего примера это будет элемент со значением атрибута `id`, равным `errorConsole`.

Мы будем называть такую конструкцию объектом с *глобальной информацией обратного вызова* (*Global Callback Info*). Некоторым типам глобальных функций передаются дополнительные параметры (как мы вскоре увидим), но этот первый универсальный параметр позволит выяснить тип вызванной функции и элемент, к которому она была подключена.

Подключать глобальные функции можно посредством следующих команд: `ajaxStart()`, `ajaxSend()`, `ajaxSuccess()`, `ajaxError()`, `ajaxComplete()` и `ajaxStop()`.

Поскольку синтаксис команд, выполняющих подключение функций всех этих типов, ничем не отличается, все они представлены единым описанием синтаксиса:

Синтаксис команды: глобальные функции Ajax

```
ajaxStart(callback)
ajaxSend(callback)
ajaxSuccess(callback)
ajaxError(callback)
ajaxComplete(callback)
ajaxStop(callback)
```

Подключает переданную функцию ко всем обернутым элементам при вызове на определенном этапе обработки запроса Ajax.

Параметры

`callback` (функция) Подключаемая функция обратного вызова. Подробные сведения о том, когда вызываются функции обратного вызова и какие параметры они получают, приведены в табл. 8.3.

Возвращаемое значение

Обернутый набор.

Каждая из этих глобальных функций обратного вызова вызывается на определенном этапе обработки запроса Ajax или при определенном условии в зависимости от кода статуса ответа; предполагается, что вызов

глобальных функций разрешен для запроса Ajax. В табл. 8.3 описан порядок вызова каждого типа глобальных функций обратного вызова и параметры, которые им передаются.

Таблица 8.3. Глобальные функции обратного вызова, перечисленные в порядке запуска

Тип функции	Когда вызывается	Параметры
`ajaxStart`	После запуска любой из функций jQuery Ajax, но перед созданием экземпляра XHR	• Объект с глобальной информацией обратного вызова, со значением свойства `type`, равным `ajaxStart`
`ajaxSend`	После создания экземпляра XHR, но перед тем, как запрос будет отправлен на сервер	• Объект с глобальной информацией обратного вызова, со значением свойства `type`, равным `ajaxSend` • Экземпляр XHR • Свойства, использованные функцией `$.ajax()`
`ajaxSuccess`	После того как запрос вернется от сервера, и ответ будет содержать код статуса, свидетельствующий об успехе	• Объект с глобальной информацией обратного вызова, со значением свойства `type`, равным `ajaxSuccess` • Экземпляр XHR • Свойства, использованные функцией `$.ajax()`
`ajaxError`	После того как запрос вернется от сервера, и ответ будет содержать код статуса, свидетельствующий об ошибке	• Объект с глобальной информацией обратного вызова, со значением свойства `type`, равным `ajaxError` • Экземпляр XHR • Свойства, использованные функцией `$.ajax()` • Объект-исключение полученный от XHR, если имеется
`ajaxComplete`	После того как запрос вернется от сервера, и после вызова любой функции обратного вызова, объявленной с помощью `ajaxSuccess` или `ajaxError`	• Объект с глобальной информацией обратного вызова, со значением свойства `type`, равным `ajaxComplete` • Экземпляр XHR • Свойства, использованные функцией `$.ajax()`
`ajaxStop`	После того как завершатся все этапы обработки запроса и будут вызваны все другие допустимые функции обратного вызова	• Объект с глобальной информацией обратного вызова, со значением свойства `type`, равным `ajaxStop`

Рассмотрим на примере, насколько просто некоторые из этих команд позволяют выводить сообщения об успешном или неудачном выполнении запроса. Тестовая страница (слишком простая, чтобы назвать ее

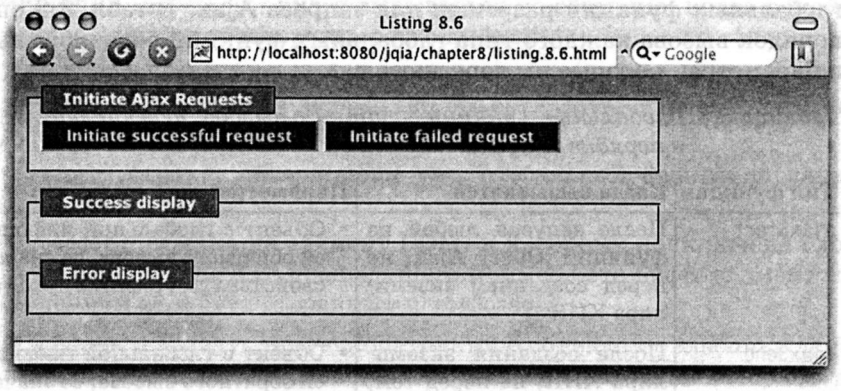


***Рис. 8.7.** Начальное состояние страницы, демонстрирующей применение глобальных функций обратного вызова Ajax*

лабораторной) приведена на рис 8.7, а найти ее можно в файле *chapter8/listing.8.6.html*.

На этой странице мы определили три панели: одна содержит кнопки, которые инициируют запросы Ajax, другая представляет собой область вывода сообщений об успешном выполнении запросов, а третья – область вывода сообщений об ошибках. HTML-разметка для этой страницы достаточно проста.

```
<body>
  <fieldset>
    <legend>Initiate Ajax Requests</legend>
    <div>
      <button type="button" id="goodButton">
        Initiate successful request
      </button>
      <button type="button" id="badButton">
        Initiate failed request
      </button>
    </div>
  </fieldset>

  <fieldset>
    <legend>Success display</legend>
    <div id="successDisplay"></div>
  </fieldset>

  <fieldset>
    <legend>Error display</legend>
    <div id="errorDisplay"></div>
  </fieldset>
</body>
```

Обработчик события готовности документа для этой страницы решает следующие три задачи:

1. Устанавливает обработчики события щелчка мыши для кнопок.
2. Устанавливает глобальную функцию как обработчик события успешного завершения запроса, подключенную к панели Success display (поле успеха).
3. Устанавливает глобальную функцию как обработчик события неудачного завершения запроса, подключенную к панели Error display (поле неудачи).

Настройка обработчиков события щелчка мыши выполняется просто.

```
$('#goodButton').click(function(){
  $.get('reflectData.jsp');
});
$('#badButton').click(function(){
  $.get('returnError.jsp');
});
```

«Хорошая кнопка» (*goodButton*) инициирует запрос к ресурсу, который вернет код состояния успеха, а «плохая кнопка» (*badButton*) инициирует запрос к ресурсу, всегда возвращающий код ошибки.

Теперь с помощью команды `ajaxSuccess()` установим обработчик события успешного завершения запроса, подключенный к элементу `<div>` со значением атрибута `id`, равным `successDisplay`:

```
$('#successDisplay').ajaxSuccess(function(info){
  $(info.target)
    .append('<div>Success at '+new Date()+'</div>');
});
```

Она установит функцию, вызываемую при успешном завершении запроса. Функции обратного вызова передается объект с глобальной информацией обратного вызова, чье свойство `target` идентифицирует элемент привязки – в данном случае, `successDisplay`. Эта ссылка позволит нам создать сообщение и вывести его в панель Success display.

Функция обратного вызова для вывода сообщений об ошибках похожа на нее.

```
$('#errorDisplay').ajaxError(function(info,xhr){
  $(info.target)
    .append('<div>Failed at '+new Date()+'</div>')
    .append('<div>Status: ' + xhr.status + ' ' +
            xhr.statusText+'</div>');
});
```

С помощью команды `ajaxError()` мы подключаем функцию обратного вызова, запускаемую в случае неудачного завершения запроса. Так как функции передается экземпляр XHR, мы используем его, чтобы дать пользователю информацию о природе ошибки.

Поскольку каждая функция связана только с одним типом глобальных функций, нам не нужно анализировать свойство `type` объекта с глобальной информацией обратного вызова. Заметили, что эти две функции

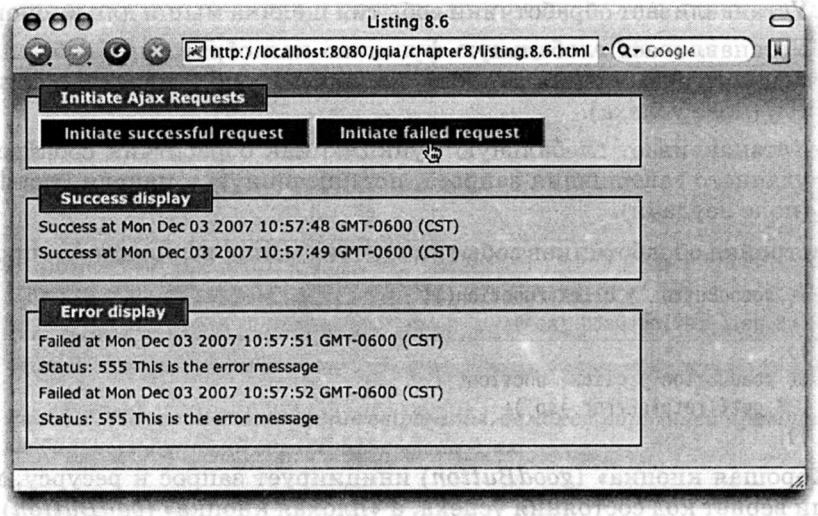


Рис. 8.8. Щелчки на кнопках показывают, как функции получают информацию и как они определяют элемент, к которому подключены

выполняют похожие действия? Как бы вы для повышения эффективности объединили эти функции в одну с помощью свойства `type`?

Откройте страницу в броузере, щелкните на каждой кнопке несколько раз (в текст сообщений добавляется информация о времени, поэтому легко определить порядок, в котором они появлялись). В результате вы получите примерно такую страницу, как на рис. 8.8.

Прежде чем перейти к другой главе, давайте соединим все полученные знания для реализации примера.

8.5. Соединяем все вместе

Пришло время еще одного всеобъемлющего примера. Давайте применим все полученные знания – селекторы, приемы манипулирования деревом DOM, улучшенные приемы программирования на JavaScript, события, эффекты и технологию Ajax – и взяв все это за основу, реализуем еще одну команду jQuery!

Для этого опять вернемся к странице The Boot Closet. Просмотрите еще раз на рис. 8.2, 8.3, 8.5 и 8.6, потому что мы будем расширять возможности этой страницы.

В области вывода подробной информации о продаваемых моделях обуви (см. рис. 8.3) применяются специальные обувные термины, которые могут быть незнакомы нашим клиентам. Мы бы хотели дать клиенту возможность понять, что они означают, потому что информированный клиент – это счастливый клиент. А счастливый клиент более склонен что-нибудь купить!

Можно составить предметный указатель и поместить в него описания всех 1998 терминов, но это отвлекло бы клиента от страницы, на которой мы хотим его удержать, – от той страницы, где покупаются вещи! *Чуть* более современный подход – открывать диалоговое окно с предметным указателем или даже давать определение термина по запросу. Но даже этот подход слишком устарел.

Если призадуматься, можно прийти к мысли – а нельзя ли с помощью атрибута `title` элемента DOM выводить всплывающую подсказку с определением термина, когда клиент наводит на него указатель мыши? Отличная мысль! Это позволит показывать определение прямо в странице и не отвлекать от нее внимание клиента.

Но работа с атрибутом `title` представляет для нас некоторую проблему. Во-первых, всплывающая подсказка появляется только при наведении указателя мыши на элемент и всего на несколько секунд, а нам бы хотелось быть более открытыми и отображать информацию сразу после щелчка на термине. Но что еще важнее: некоторые броузеры ограничивают во всплывающей подсказке объем текста из атрибута `title`, и это слишком мало для нас.

А раз так, создадим собственные всплывающие подсказки!

Итак, нам требуется каким-либо образом идентифицировать термины, для которых будут даны определения, изменить их внешний вид, чтобы пользователи могли отличать их, и выбрать инструмент, который обеспечивал бы вывод всплывающей подсказки с описанием термина после щелчка на этом термине. При следующем щелчке мыши всплывающая подсказка должна исчезнуть с экрана.

На рис. 8.9 показана часть страницы, демонстрирующая поведение, которое нам требуется добавить.

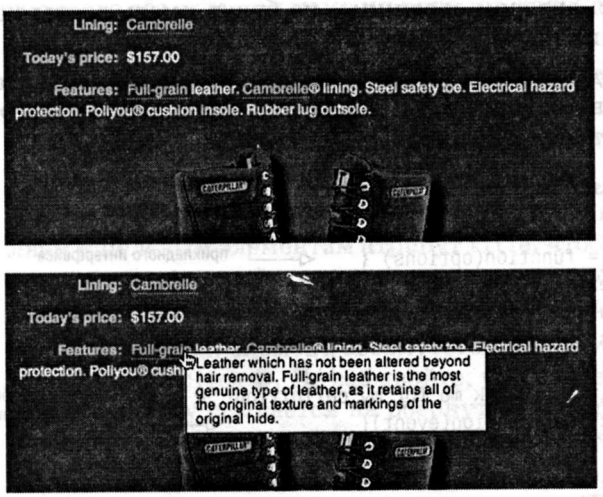


Рис. 8.9. *Страница до и после вывода всплывающей подсказки*

В верхней части страницы мы видим в поле Features (характеристики) выделенные термины Full-grain и Cambrelle. В нижней части рисунка видна всплывающая подсказка с описанием, появившаяся после щелчка на термине Full-grain.

Можно было бы жестко встроить все определения непосредственно в страницу, но мы выберем более грамотный подход. Как и в случае с расширениями для манипулирования отобранными элементами, мы должны создать компонент многократного использования для повсеместного применения на этом и на любом другом сайте. А поскольку мы хотим следовать стилю jQuery, то и реализуем команду jQuery.

8.5.1. Реализация всплывающей подсказки

Как вы помните, команда jQuery добавляется с помощью свойства `$.fn`. Наше новое расширение, отыскивающее описания терминов, мы назовем *The Termifier*, а сама команда будет называться `termifier()`.

Команда `termifier()` будет обрабатывать все элементы в обернутом наборе так, чтобы достичь следующих целей:

- В каждый обернутый элемент необходимо установить обработчик события `click`, который будет выводить на экран всплывающую подсказку The Termifier.
- После щелчка мыши описание термина должно быть получено из серверного ресурса.
- Полученное описание термина будет отображаться во всплывающей подсказке с применением эффекта проявления (fade-in).
- После щелчка на всплывающей подсказке она исчезает с применением эффекта растворения (fade-out).
- URL-адрес серверного ресурса и CSS-класс элемента-подсказки могут выбираться автором страницы, но будут иметь достаточно разумные значения по умолчанию.

Программный код команды jQuery, соответствующей поставленным целям, приведен в листинге 8.8 и находится в файле *chapter8/bootcloset/jquery.jqia.termifier.js*.

Листинг 8.8. Реализация команды termifier()

```
(function($) {
  $.fn.termifier = function(options) {
    options = $.extend({
      lookupResource: 'getTerm',
      flyoutClass: 'lookerUpperFlyout'
    },options||{});
    this.attr('title','Click me for my definition!');
    return this.click(function(event){
      $.ajax({
        url: options.lookupResource,
        type: 'GET',
```

❶ Определение команды прикладного интерфейса

❷ Объединение входных параметров со значениями по умолчанию

❸ Установка обработчика события click для термина

❹ Запуск запроса на получение описания термина

```
        data: {term: this.innerHTML},
        dataType: 'html',                                    ❺ Обработка
        success: function(data) {                              полученных данных
          $('<div></div>')
            .css({
              position: 'absolute',
              left: event.pageX,
              top: event.pageY,
              cursor: 'pointer',
              display: 'none'
            })
            .html(data)
            .addClass(options.flyoutClass)                   ❻ Установка обработчика
            .click(function(){                                 события click для подсказки
              $(this).fadeOut(1500,function(){$(this).remove();});
            })
            .appendTo('body')                                ❼ Включение подсказки в дерево DOM
            .fadeIn();                                         страницы и вызов эффекта проявления
        }
      });
      return false;
    });
  }
})(jQuery);
```

Не такой уж длинный программный код, как можно было ожидать! Но это еще не все. Давайте за один раз будем делать только один шаг.

Во-первых, для создания команды `termifier()` ❶ мы использовали шаблон, описанный в главе 7. Единственный ожидаемый параметр этой команды является объектом, свойства которого играют роль входных аргументов. Для большего удобства мы предусмотрели значения по умолчанию, которые объединяются со значениями свойств входного объекта при помощи вспомогательной функции `$.extend()` ❷. Были определены следующие параметры:

- `lookupResource` – определяет URL используемого ресурса на стороне сервера;
- `flyoutClass` – класс CSS, применяемый к вновь создаваемым элементам всплывающих подсказок.

В качестве дополнительной подсказки для наших клиентов мы добавляем к целевым элементам атрибут `title`, чтобы, наведя указатель мыши на выделенный термин, они видели сообщение, предлагающее щелкнуть на термине для вывода полезной информации.

Затем в каждый элемент обернутого набора устанавливается обработчик события `click` ❸. Не забывайте, что контекст функции (`this`) в командах jQuery ссылается на обернутый набор, поэтому чтобы применить другие команды к этому набору, достаточно вызвать их как методы объекта `this`.

В обработчике щелчка (события click) инициируется Ajax-запрос, в ответ на который мы получаем определение термина. Максимальное управление запросом обеспечивает функция `$.ajax()` ❹, и мы передаем ей объект, в котором определяются следующие параметры:

- URL-адрес, определяемый параметром команды (либо по умолчанию, либо предоставленный автором страницы);
- метод HTTP GET (потому что запрос является исключительно идемпотентным);
- параметр запроса с именем `term`, значением которого является содержимое целевого элемента (контекст функции внутри обработчика события);
- признак того, что ожидается ответ в формате HTML;
- функция обратного вызова `success` ❺, которая на основе данных, полученных в ответе, создаст элемент – всплывающую подсказку.

Гораздо более интересные события происходят в функции обратного вызова `success`. Она сначала создает новый пустой элемент `<div>`, а затем выполняет над ним следующие операции:

- В элемент добавляются стили CSS, которые определяют для него режим абсолютного позиционирования с координатами, соответствующими координатам события `click`, придают указателю мыши для этого элемента форму «указующего перста» и делают элемент невидимым.
- Данные ответа, полученные функцией обратного вызова `success` в первом параметре (который, как мы знаем, содержит описание термина), вставляются в элемент `<div>` в качестве содержимого.
- В элемент `<div>` добавляется класс CSS, идентифицируемый параметром `flyoutClass`.
- К элементу `<div>` подключается обработчик события `click` ❻, который вызывает эффект медленного растворения и по завершении воспроизведения эффекта удаляет элемент из дерева DOM.
- Вновь созданный элемент подсказки `<div>` добавляется в конец дерева DOM, в элемент `<body>`.
- В заключение, элемент `<div>` отображается на экране с воспроизведением эффекта проявления со скоростью по умолчанию ❼.

Команда `termifier()` возвращает обернутый набор (полученный как результат работы команды `click()`), благодаря чему наша новая команда может входить в состав любых цепочек команд jQuery.

Теперь давайте посмотрим, как использовать эту команду в нашей странице The Boot Closet.

8.5.2. Применение расширения The Termifier

Поскольку всю сложную логику создания и манипулирования всплывающей подсказкой мы поместили в команду `termifier()`, применять

эту новую команду jQuery на странице The Boot Closet совсем несложно. Но сначала нужно принять некоторые решения.

Мы должны решить, как идентифицировать термины на странице. Не забывайте, что нам нужно создать обернутый набор элементов, содержимое которых включает элементы-термины, над которыми будут выполняться операции. Мы могли бы использовать элемент с определенным именем класса, например так:

```
<span class="term">Goodyear welt</span>
```

Создать обернутый набор из таких элементов можно с помощью инструкции $('span.term').

Кому-то разметка покажется несколько многословной. Вместо нее мы применим малоиспользуемый тег <abbr>. Тег <abbr> был добавлен в HTML 4, чтобы помочь идентифицировать применяемые в документе сокращения. Поскольку тег предназначен исключительно для идентификации элементов документа, ни один из броузеров почти ничего не делает с этим тегом в смысле семантики или визуального представления, поэтому он прекрасно подходит для наших целей.

Примечание

Спецификация HTML 4[1] определяет ряд подобных тегов, предназначенных для использования в документах, например <cite>, <dfn> и <acronym>. Предварительная спецификация HTML 5[2] предлагает добавить еще больше таких тегов, призванных обеспечить семантику, а не директивы схемы размещения или визуального представления. Среди них – такие теги, как <section>, <article> и <aside>.

Поэтому первое, что необходимо сделать, – это модифицировать серверный ресурс, чтобы он возвращал описание характеристик модели обуви, в котором термины, имеющие отдельные описания, должны быть заключены в теги <abbr>. Оказывается, ресурс getDetails.jsp уже предоставляет все необходимое. Но поскольку броузеры ничего не делают с тегом <abbr>, вы, возможно, даже не заметили его, если, конечно, не заглянули в файл JSP или PHP. Этот ресурс возвращает данные в формате JSON:

```
{
  name: 'Chippewa Harness Boot',
  sku: '7141922',
  height: '13"',
  lining: 'leather',
  colors: 'Black, Crazy Horse',
  price: '$188.00',
  features: '<abbr>Full-grain</abbr> leather uppers. Leather
  ➥lining. <abbr>Vibram</abbr> sole. <abbr>Goodyear welt</abbr>.'
}
```

[1] *http://www.w3.org/TR/html4/*

[2] *http://www.w3.org/html/wg/html5/*

Обратите внимание: термины *Full-grain*, *Vibram* и *Goodyear welt* идентифицируются с помощью тега <abbr>.

Вернемся непосредственно к странице. В качестве отправной точки примем исходный код страницы в листинге 8.6 и посмотрим, что нужно добавить в нее, чтобы задействовать The Termifier. Мы должны добавить в страницу новую команду, поэтому в раздел <head> вставляем следующую инструкцию (после загрузки самой библиотеки jQuery):

```
<script type="text/javascript"
        src="jquery.jqia.termifier.js"></script>
```

Команду `termifier()` следует применять ко всем тегам <abbr>, добавляемым в страницу после загрузки информации о модели обуви, поэтому мы добавили функцию обратного вызова в команду `load()`, которая получает эту информацию. Данная функция обратного вызова с помощью расширения The Termifier обрабатывает теги <abbr>. Подправленная команда `load()` (с изменениями в теле) выглядит так:

```
$('#detailsDisplay').load(
  'getDetails.jsp',
  { style: styleValue },
  function(){
    $('abbr').termifier({
      lookupResource: 'getTerm.jsp'
    });
  }
);
```

Добавленная функция создает обернутый набор всех элементов <abbr> и применяет к нему команду `termifier()`, указывая серверный ресурс `getTerm.jsp` и переопределяя тем самым значение по умолчанию.

Вот и все.

Мы поступили очень мудро, выполнив сложную реализацию в виде команды jQuery, благодаря чему использовать эту команду оказалось проще простого. Так же легко мы сможем использовать ее в любой другой странице или любом другом сайте. Именно в этом и заключается *разработка*!

Единственное, что осталось, – это изменить внешний вид текстовых элементов, чтобы пользователь мог понять, на каких элементах можно щелкнуть мышью. В файл CSS мы добавили следующие свойства CSS для тега <abbr>:

```
color: aqua;
cursor: pointer;
border-bottom: 1px aqua dotted;
```

Эти стили придают терминам внешний вид ссылок, с тем лишь различием, что они подчеркиваются пунктиром. Такой внешний вид приглашает пользователей щелкнуть на термине, но при этом сохраняет отличие от настоящих ссылок, присутствующих на странице.

Новая страница находится в файле *chapter8/bootcloset/boot.closet.3.html*. Поскольку изменения, коснувшиеся программного кода в листинге 8.6, минимальны (о чем уже говорилось), мы сэкономим немного бумаги, не приводя здесь полный листинг страницы.

Обновленная страница с новыми функциональными возможностями представлена на рис. 8.10.

Наша новая команда удобна и имеет широкие возможности, но всегда есть...

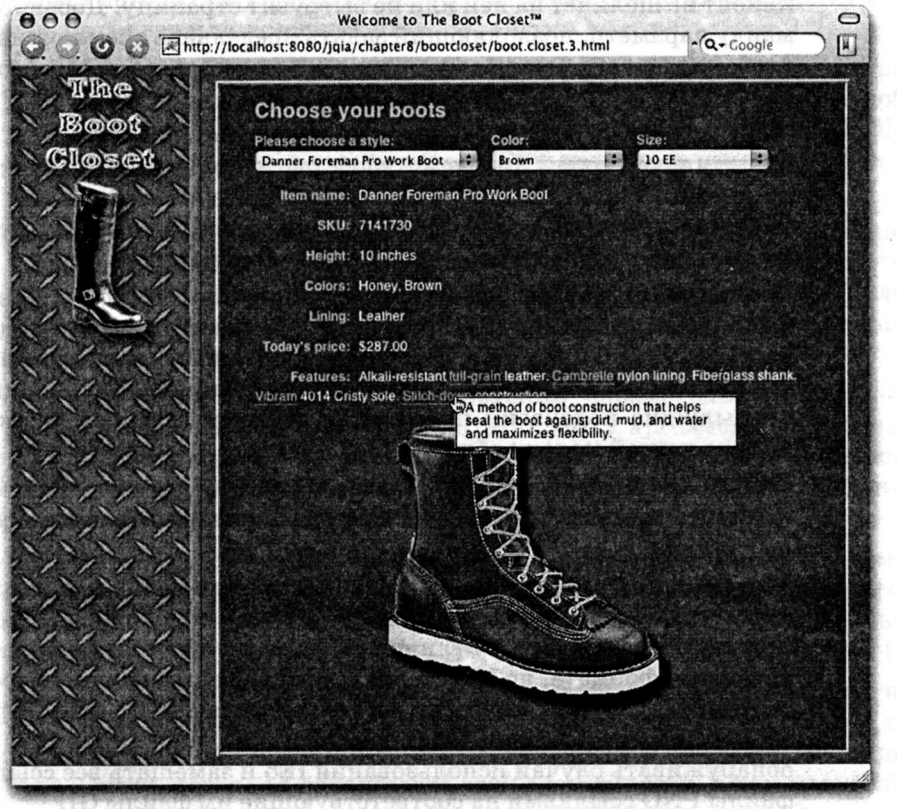


***Рис. 8.10.** Теперь наши клиенты знают, что такое Stitch-down*

8.5.3. Место для усовершенствований

Наша новая команда jQuery действительно очень полезна, но в ней есть несколько незначительных проблем, и ее можно существенно улучшить. Чтобы отточить ваши навыки, предлагаем вашему вниманию перечень изменений, которые вы могли бы внести в эту команду или в страницу The Boot Closet.

- Серверный ресурс получает термин в виде параметра запроса с именем `term`. Добавьте в команду параметр, позволяющий автору страницы определять название параметра запроса. Наша клиентская команда не должна диктовать, как писать серверный программный код.
- Добавьте параметр (или параметры), позволяющий автору страницы управлять продолжительностью эффектов проявления и растворения или даже применять другие эффекты.
- Всплывающая подсказка The Termifier остается на экране, пока клиент не щелкнет на ней или не выгрузит страницу. Добавьте в команду параметр, позволяющий задать предельное время ожидания, чтобы подсказка могла автоматически исчезать по истечении указанного времени.
- Если щелкнуть на всплывающей подсказке, она исчезает, а это представляет собой проблему удобства использования, поскольку из-за этого текст подсказки нельзя выделить и скопировать. Измените программный код так, чтобы он закрывал подсказку в случае щелчка где-нибудь на странице, за пределами подсказки.
- Вполне возможен вывод на экран одновременно нескольких всплывающих подсказок – если пользователь, не закрыв предыдущую подсказку, щелкнет на другом термине (причем даже при выборе другой модели обуви). Добавьте программный код, удаляющий все предыдущие подсказки перед отображением новой, а также при выборе другой модели обуви.
- Наша команда не предусматривает обработку ошибок. Как можно изменить команду, чтобы она предусматривала обработку серверных ошибок?
- Мы добились отображения привлекательных теней в наших изображениях посредством файлов PNG с частичной прозрачностью. Большинство броузеров отлично справляются с отображением файлов в этом формате, но IE6 отображает их с белым фоном. Чтобы решить эту проблему, мы могли бы также поставлять файлы в формате GIF с изображениями без теней. Как дополнить страницу, чтобы обнаруживать случаи использования IE6 и замещать все ссылки на файлы PNG ссылками на соответствующие им файлы GIF?
- И раз уж зашла речь об изображениях. У нас предусмотрена всего одна фотография для каждой модели обуви, даже когда на выбор предлагается несколько цветов. Предположим, у нас есть несколько фотографий каждой модели – по одной для каждого возможного цвета. Как расширить возможности страницы, чтобы выводить соответствующую фотографию при изменении цвета?

А может, вы сами придумали другие возможные усовершенствования этой страницы или команды `termifier()`? Тогда поделитесь идеями и решениями на форуме книги по адресу *http://www.manning.com/bibeault*.

8.6. Итоги

Неудивительно, что эта глава – одна из самых длинных в книге. Технология Ajax – это ключевая составляющая полнофункциональных интернет-приложений, и jQuery не остается в стороне, предлагая нам богатый набор функциональных возможностей для работы с ней.

Для загрузки фрагментов HTML в элементы DOM команда `load()` предоставляет простой способ получить содержимое со стороны сервера и превратить его в содержимое любого обернутого набора элементов. Выбор метода POST или GET зависит от того, нужно ли отправить данные на сервер или получить их.

Для случая, когда требуется применить метод GET, jQuery предоставляет вспомогательные функции `$.get()` и `$.getJSON()`. Последнюю удобно использовать, когда сервер возвращает данные в формате JSON. Принудительно использовать метод POST позволяет вспомогательная функция `$.post()`.

Если требуется максимальная гибкость, вспомогательная функция `$.ajax()`, обладая богатым набором параметров, позволит управлять большинством аспектов Ajax-запросов. Все остальные функциональные возможности Ajax, реализованные в библиотеке 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 нам отслеживать ход выполнения запросов и связывать эти события с элементами DOM с помощью команд `ajaxStart()`, `ajaxSend()`, `ajaxSuccess()`, `ajaxError()`, `ajaxComplete()` и `ajaxStop()`.

Столь внушительная коллекция инструментов поддержки технологии Ajax упрощает для нас реализацию возможностей полнофункциональных интернет-приложений. Не забывайте, насколько просто расширяется jQuery, позволяя наращивать свои возможности, в чем мы убедились, разобрав отсутствие готовой поддержки в библиотеке jQuery для некоторых случаев. Не исключено, что найдется и готовый модуль расширения, официальный или нет, который реализует именно те возможности, в которых вы нуждаетесь.

Этим модулям посвящена следующая глава.

9

В этой главе:

- Обзор модулей расширения jQuery
- Официальный модуль расширения Form Plugin
- Официальный модуль расширения Dimensions Plugin
- Модуль расширения Live Query Plugin
- Модуль расширения UI Plugin

Замечательные, мощные и практичные расширения

В первых восьми главах этой книги все свое внимание мы сосредоточили на изучении базовых функциональных возможностей, предоставляемых библиотекой jQuery нам как авторам страниц. Но это лишь вершина айсберга! Нам доступна поистине огромная коллекция модулей расширения jQuery, которая содержит внушительный набор инструментов, основанных на ядре jQuery.

Создатели ядра библиотеки jQuery очень тщательно отбирали функции, необходимые большинству авторов страниц, и в результате создали платформу, на базе которой с успехом могут создаваться модули расширения. Такая архитектура позволяет минимизировать размер базовой части библиотеки и дает возможность нам, авторам страниц, самим решать, как израсходовать остальную часть полосы пропускания, выбирая только те дополнительные функциональные возможности, которые действительно необходимы.

В одной главе невозможно охватить все модули расширения jQuery – для этого не хватило бы и целой книги, – поэтому нам пришлось внимательно отнестись к выбору модулей, которые здесь описаны. Это была невероятно сложная задача, и мы выбрали модули, на наш взгляд, наиболее важные или необходимые большинству разработчиков веб-приложений.

Отсутствие описания какого-либо модуля в этой главе вовсе не говорит о его бесполезности или непригодности! Мы пошли на это скрепя сердце.

Отыскать дополнительную информацию обо всех доступных модулях расширения можно по адресу *http://docs.jquery.com/Plugins* или *http://jquery.com/plugins/most_popular*.

Заметим, что мы не смогли полностью охватить даже те модули, которые *будут* обсуждаться, но эта глава должна дать вам неплохое понимание основ модулей и принципов их использования. За необходимой информацией обращайтесь к официальной документации для каждого модуля.

Начнем обзор модулей с расширения, неоднократно упомянутого на страницах книги.

9.1. Form Plugin

Работа с формами может оказаться непростым делом. У каждого типа элементов управления есть свои особенности, а сама форма часто может отправляться непредусмотренными способами. Ядро библиотеки jQuery включает несколько методов, облегчающих работу с формами, но их не так много, чтобы оказать нам существенную помощь. Официальный модуль расширения Form Plugin призван восполнить этот недостаток и помочь нам управлять формами.

Этот модуль можно найти по адресу *http://jquery.com/plugins/project/form*, в файле *jquery.form.js*.

Он расширяет возможности библиотеки в трех направлениях:

- получение значений из элементов формы;
- очистка и установка значений по умолчанию в элементах формы;
- отправка форм (включая выгрузку файлов) с применением механизмов Ajax.

Начнем с возможности получения значений из элементов формы.

9.1.1. Получение значений элементов формы

Модуль Form Plugin предоставляет два способа получения значений элементов формы: в виде массива значений или в виде сериализованной строки. Для извлечения значений из элементов управления модуль Form Plugin предоставляет три метода: `fieldValue()`, `formSerialize()` и `fieldSerialize()`.

Начнем с извлечения значений из полей ввода.

Получение значений элементов управления

Получить значения элементов формы можно с помощью команды `fieldValue()`. На первый взгляд, команда `fieldValue()` повторяет функциональность команды `val()`. Однако до версии 1.2 библиотеки jQuery команда `val()` обладала значительно меньшими возможностями, и метод `fieldValue()` призван восполнить этот недостаток.

Первое главное различие между этими двумя методами состоит в том, что метод `fieldValue()` возвращает массив всех значений элементов формы из обернутого набора, тогда как метод `val()` возвращает значение

первого элемента (и только в том случае, если этот элемент является элементом формы). На самом деле `fieldValue()` всегда возвращает массив, даже если возвращается всего одно значение или отсутствуют значения, которые можно было бы вернуть.

Другое различие: `fieldValue()` игнорирует все элементы в обернутом наборе, не являющиеся элементами формы. Если создать обернутый набор всех элементов страницы, то возвращаемый массив будет содержать столько значений, сколько элементов формы удастся найти методу `fieldValue()`. Но в возвращаемом массиве не все элементы управления будут представлены своими значениями: подобно команде `val()`, метод `fieldValue()` по умолчанию возвращает значения только для тех элементов, которые считаются *успешными*.

Что такое *успешный* элемент управления? Это не тот элемент, который может позволить себе коллекционировать спортивные автомобили, а формальное определение спецификации HTML[1], определяющей значения элементов, которые могут или не могут отправляться в составе формы.

Мы не будем здесь углубляться в подробности, лишь вкратце отметим, что успешными элементами считаются такие элементы, которые имеют атрибут `name`, находятся в активном состоянии и помечены (последнее касается элементов управления, которые могут быть отмечены, таких как флажки и переключатели). Некоторые элементы управления, такие как кнопки, всегда рассматриваются как неуспешные и никогда не отправляются серверу в составе формы. Другие элементы, такие как `<select>`, чтобы считаться успешными, должны иметь выбранное значение.

Команда `fieldValue()` предоставляет возможность выбора – включать или нет неуспешные элементы в массив. Ее синтаксис:

Синтаксис функции $.fieldValue

`$.fieldValue(excludeUnsuccessful)`

Выбирает значения всех успешных элементов формы в обернутом наборе и возвращает их в виде массива строк. Если не найдено ни одно значение, возвращается пустой массив.

Параметры

`excludeUnsuccessful`	(логическое значение) Если имеет значение `true` или опущен, все неуспешные элементы форм в обернутом наборе игнорируются.

Возвращаемое значение

Строковый массив собранных значений.

[1] *http://www.w3.org/TR/REC-html40/*

Чтобы показать работу этой команды, мы создали еще одну лабораторную страницу. Вы найдете ее в файле *chapter9/form/lab.get.values.html*. В окне броузера она выглядит, как показано на рис. 9.1.

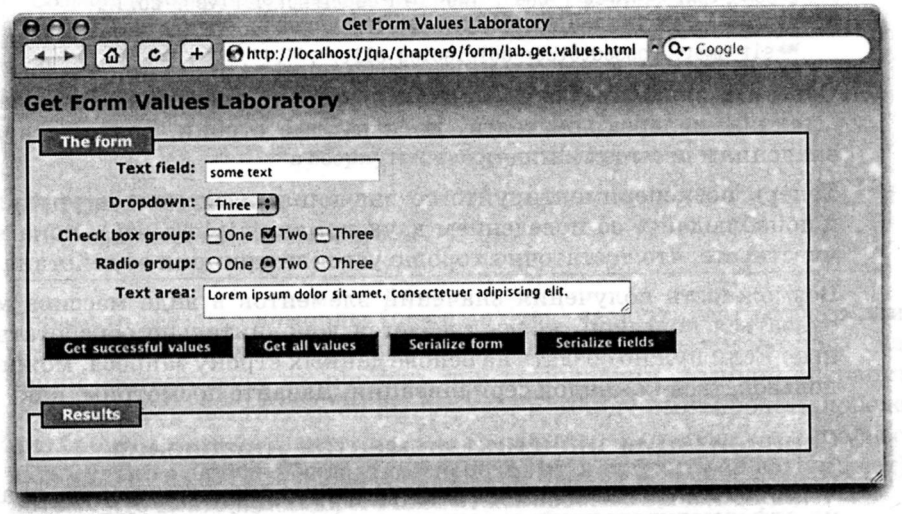


Рис. 9.1. Страница Get Form Values Laboratory поможет нам понять принцип действия команд fieldValue( ) и serialize( )

Откройте эту страницу и, оставив поля ввода в исходном состоянии, щелкните на кнопке Get Successful Values (извлечь значения успешных элементов). В результате будет выполнена следующая команда:

```
$('#testForm *').fieldValue()
```

В ходе ее выполнения будет создан обернутый набор всех дочерних элементов тестовой формы, включая элементы <label> и , и вызвана команда `fieldValue()` для этого набора. Поскольку эта команда при вызове без параметра игнорирует все элементы, не являющиеся элементами формы, она отберет только успешные элементы управления, а результаты ее работы будут отображены на экране:

```
['some text','Three','cb.2','radio.2','Lorem ipsum dolor sit amet,
➥consectetuer adipiscing elit.']
```

Как и следовало ожидать, в массив попали значения текстового поля ввода, раскрывающегося списка, *отмеченного* флажка, *отмеченного* переключателя и текстовой области.

Теперь щелкните на кнопке Get All Values (извлечь все значения), которая выполняет команду:

```
$('#testForm *').fieldValue(false)
```

Значение false, передаваемое этой команде, предписывает ей не исключать неуспешные элементы управления. В результате мы увидим более обширный набор данных:

```
['some text','Three','One','Two','Three','Four','Five','cb.1',
➥'cb.2','cb.3','radio.1','radio.2','radio.3','Lorem ipsum dolor
➥sit amet, consectetuer adipiscing elit.','','','','']
```

Обратите внимание: сюда попали не только значения неотмеченных флажков и переключателей, но и пустые строки, соответствующие значениям четырех кнопок.

Теперь поэкспериментируйте со значениями элементов управления и понаблюдайте за поведением двух команд fieldValue(), пока не почувствуете, что достаточно хорошо усвоили принцип ее действия.

Возможность получения значений элементов в виде массива может оказаться полезной, когда требуется дополнительно обработать данные. Если нужно создать на основе данных строку запроса, можно воспользоваться командой сериализации. Давайте посмотрим, как.

Сериализация значений элементов управления

Если из значений элементов формы нужно сконструировать правильно отформатированную и закодированную в формат URL строку запроса, можно воспользоваться командой formSerialize() или fieldSerialize(). Оба эти метода обертки выбирают значения из обернутого набора и возвращают отформатированную строку запроса с именами и значениями, закодированными в формат URL. Метод formSerialize() принимает форму в обернутом наборе и сериализует все ее *успешные* дочерние элементы. Команда fieldSerialize() сериализует все элементы управления в обернутом наборе, поэтому ее удобно применять для сериализации части формы.

Эти команды имеют следующий синтаксис:

Синтаксис функции formSerialize

`formSerialize(semantic)`

Создает и возвращает отформатированную и закодированную строку запроса, состоящую из значений всех успешных элементов управления в обернутой форме.

Параметры

semantic	(логическое значение) Указывает, должны ли значения в строке запроса следовать в семантическом порядке, то есть в порядке объявления элементов формы. В этом случае скорость работы команды может существенно снижаться по сравнению со случайным порядком следования значений.

Возвращаемое значение

Сгенерированная строка запроса.

Синтаксис функции fieldSerialize

`fieldSerialize(excludeUnsuccessful)`

Создает и возвращает отформатированную и закодированную строку запроса, состоящую из значений элементов управления в обернутом наборе.

Параметры

`excludeUnsuccessful`	(логическое значение) Если имеет значение `true` или опущен, все неуспешные элементы формы в обернутом наборе игнорируются.

Возвращаемое значение

Сгенерированная строка запроса.

Параметр `semantic` команды `formSerialize()` заслуживает особого внимания. Если он имеет значение `true`, то порядок сериализованных значений соответствует их порядку при выполнении обычной операции отправки формы, то есть он вынуждает команду имитировать поведение броузера при отправке формы. Эту возможность следует использовать только в случае абсолютной необходимости (которой обычно нет), потому что за это приходится платить скоростью выполнения.

Предупреждение

Параметр `semantic` вынуждает упорядочивать данные в соответствии с семантическим порядком, выбираемым при отправке формы, но будет ли серверный программный код учитывать этот порядок, – от клиентского сценария уже не зависит. Например, при использовании сервлетов вызов метода `getParameterMap()` экземпляра запроса не сохраняет порядок следования значений.

Можно исследовать эти команды с помощью лабораторной страницы Get Form Values Laboratory. Откройте страницу и, оставив элементы формы в исходном состоянии, щелкните на кнопке Serialize Form (сериализовать форму). В результате выполнится команда `formSerialize()`:

```
$('#testForm').formSerialize()
```

Будет получена строка:

```
text=some%20text&dropdown=Three&cb=cb.2&radio=radio.2&
➥textarea=Lorem%20ipsum%20dolor%20sit%20amet%2C%20conse
➥ctetuer%20adipiscing%20elit.
```

Заметим, что были выбраны все успешные элементы с их именами и значениями, и на их основе была создана строка запроса, закодированная в формат URL.

Щелчок на кнопке Serialize Fields (сериализовать поля) приведет к выполнению команды:

```
$('#testForm input').fieldSerialize()
```

Обернутый набор, созданный этим селектором, будет включать в себя только подмножество элементов формы: только элементы типа `input`. В результате будет создана строка запроса, полученная только из успешных элементов управления в обернутом наборе:

```
text=some%20text&cb=cb.2&radio=radio.2
```

Одна из причин, по которым нам может потребоваться сериализовать элементы формы в строку запроса, состоит в том, чтобы использовать эту возможность для представления данных при передаче данных через механизмы Ajax. Постойте! Если потребуется отправить форму посредством механизмов Ajax, мы сможем использовать другие возможности модуля Form Plugin. Но прежде чем подойти к этой теме вплотную, давайте исследуем некоторые команды, позволяющие манипулировать значениями элементов формы.

9.1.2. Очистка и сброс значений в элементах формы

Модуль Form Plugin предоставляет две команды, воздействующие на значения элементов формы. Команда `clearForm()` очищает все поля в обернутой форме, а команда `resetForm()` сбрасывает значения.

«Хм, а в чем разница?» – спросите вы.

Когда вызывается команда `clearForm()`, она *очищает* элементы управления так:

- В текстовые поля, поля ввода пароля и текстовые области записываются пустые значения.
- Элементы `<select>` устанавливаются в состояние отсутствия выбранных вариантов.
- Флажки и переключатели переводятся в неотмеченное состояние.

Когда вызывается команда `resetForm()`, она *сбрасывает* значения элементов управления, вызывая метод формы `reset()`. В результате элементы управления приобретают значения, указанные в HTML-разметке. Текстовые поля возвращаются к значениям атрибута `value`, другие элементы управления – к значениям атрибута `checked` или `selected`.

Для демонстрации этих отличий мы создали еще одну лабораторную страницу. Откройте файл *chapter9/form/lab.reset.and.clear.html* в своем броузере. Вы увидите страницу, показанную на рис. 9.2.

Обратите внимание: эта уже знакомая форма была инициализирована посредством HTML-разметки. Текстовое поле и текстовая область инициализируются значениями своих атрибутов `value`, в раскрывающемся списке один из вариантов выбора отмечен как `selected`, один флажок и один переключатель имеют значение `checked`.

Щелкните на кнопке Clear Form (очистить форму) и посмотрите, что произойдет. Текстовое поле и текстовая область будут очищены, раскрывающийся список перейдет в состояние, когда ни один вариант не выбран, а все флажки и переключатели окажутся непомеченными.

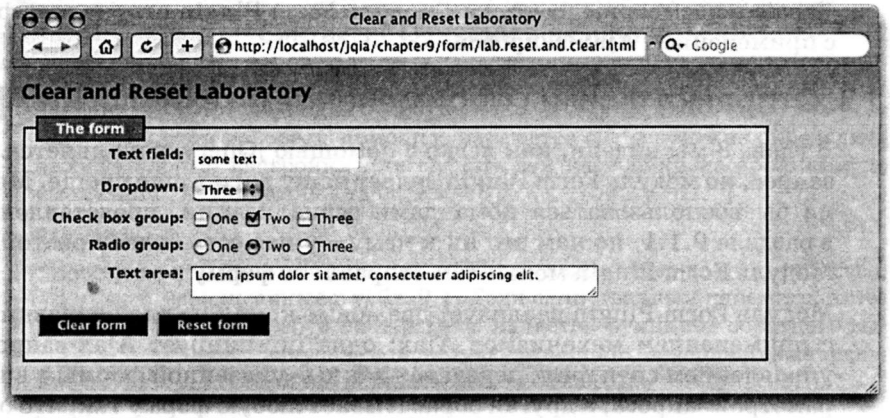


***Рис. 9.2.** Лабораторная страница Clear and Reset Laboratory демонстрирует отличия между понятиями «сбросить» и «очистить»*

Теперь щелкните на кнопке Reset Form (сбросить форму) и посмотрите, как элементы управления вернутся к своим первоначальным значениям. Измените значения в каждом из элементов управления и щелкните на кнопке Reset Form. Обратите внимание: они снова вернулись к первоначальным значениям.

Эти команды имеют следующий синтаксис:

Синтаксис функции clearForm

`clearForm()`

Очищает значения всех элементов управления в обернутом наборе или тех, что являются дочерними по отношению к элементам в обернутом наборе.

Параметры

Нет

Возвращаемое значение

Обернутый набор.

Синтаксис функции resetForm

`resetForm()`

Вызывает «родной» метод `reset()` форм в обернутом наборе.

Параметры

Нет

Возвращаемое значение

Обернутый набор.

Давайте посмотрим на то, как модуль Form Plugin отправляет формы с применением Ajax-запросов.

9.1.3. Отправка формы с применением технологии Ajax

В главе 8 мы видели, как легко с помощью jQuery выполняется Ajax-запрос, но модуль Form Plugin делает нашу жизнь еще проще. Мы могли бы воспользоваться командами сериализации, представленными в разделе 9.1.1, но нам это ни к чему – у нас есть более простой путь! Модуль Form Plugin может сам отправлять форму в запросе.

Модуль Form Plugin реализует две новые команды для отправки форм с применением механизмов Ajax: одна инициирует Ajax-запрос под управлением сценария, передавая данные указанной формы в виде параметров запроса, а другая обрабатывает любую форму так, что она будет отправляться в виде Ajax-запроса.

В обоих случаях для выполнения запросов используются базовые функции Ajax, реализованные в jQuery, поэтому все глобальные функции по-прежнему будут действовать даже при использовании этих методов.

Начнем с исследования первого подхода.

Получение данных для запроса Ajax

Разрабатывая пример сайта электронной коммерции в главе 8, мы столкнулись с рядом ситуаций, когда нам требовалось извлечь значения из элементов формы, чтобы отправить их на сервер в виде запроса Ajax – вполне обычное требование. Мы видели, что для базовой функции поддержки технологии Ajax это простое дело, особенно если нужно отправить небольшое число значений.

Комбинация метода `serializeForm()` модуля Form Plugin и базовых функций поддержки технологии Ajax делает операцию отправки *всех* значений формы еще проще. Но есть вариант еще *проще* – метод `ajaxSubmit()` модуля Form Plugin делает отправку всей формы с применением технологии Ajax почти тривиальной задачей.

Эта команда, применяемая к обернутому набору, содержащему форму, извлекает имена и значения успешных элементов формы и посылает их серверу в виде запроса Ajax. Мы можем передать методу параметры, определяющие, как должен выполняться запрос, или разрешить использовать параметры по умолчанию.

Эта команда имеет синтаксис, представленный на стр. 299.

Параметр `options` позволяет точно определить, как должен выполняться запрос. Необязательные свойства описаны в табл. 9.1, все свойства имеют значения по умолчанию, чтобы обеспечить возможность создавать запросы спокойно, без суеты. Обычно этот метод вызывается без параметра, чтобы задействовать значения по умолчанию.

Синтаксис функции ajaxSubmit

`AjaxSubmit(options)`

Генерирует запрос Ajax для отправки значений *успешных* элементов формы в обернутом наборе. С помощью параметра `options` можно определить необязательные настройки или сделать эти настройки настройками по умолчанию, как описано в табл. 9.1.

Параметры

`options` (объект | функция) Объект-хеш, содержащий свойства, описанные в табл. 9.1. Если единственный параметр, который требуется передать, является функцией обратного вызова, запускаемой в случае успешного завершения запроса, то ее можно передать вместо объекта.

Возвращаемое значение

Обернутый набор.

Таблица 9.1. Необязательные свойства для команды ajaxSubmit( ), перечисленные в порядке убывания частоты использования

Имя	Описание
`url`	(строка) URL-адрес, по которому будет отправлен Ajax-запрос. Если опущен, URL извлекается из атрибута `action` формы.
`type`	(строка) Метод HTTP для отправки запроса, такой как GET или POST. Если опущен, используется метод, указанный в атрибуте `method` формы. Если это свойство опущено и в форме отсутствует атрибут `method`, используется метод GET.
`dataType`	(строка) Ожидаемый тип данных в ответе. Это свойство определяет, какая предварительная обработка тела ответа будет произведена. Если указывается, должно иметь одно из следующих значений: • `xml` – ответ интерпретируется как данные в формате XML. Любой функции обратного вызова в случае успешного завершения запроса будет передано значение свойства `responseXML`. • `json` – ответ интерпретируется как данные в формате JSON. Любой функции обратного вызова в случае успешного завершения запроса будет передан результат интерпретации JSON. • `script` – ответ интерпретируется как сценарий JavaScript. Этот сценарий будет выполнен в глобальном контексте. Если это свойство опущено, предварительная обработка данных не производится (кроме случаев, когда таковая обработка подразумевается другими свойствами, например `target`).
`target`	(строка | объект | элемент) Определяет элемент или элементы DOM, куда должно помещаться тело ответа. Это может быть строка-селектор jQuery, обертка jQuery, содержащая элементы, или прямая ссылка на элемент. Отсутствие свойства означает отсутствие элементов, принимающих данные.

Таблица 9.1 (продолжение)

Имя	Описание
beforeSubmit	(функция) Определяет функцию обратного вызова, вызываемую перед инициацией Ajax-запроса. Это удобно, если требуется организовать выполнение каких-либо предварительных операций, включая проверку данных формы. Если эта функция возвращает значение false, отправка формы отменяется. Функции обратного вызова передаются три параметра: • Массив значений, передаваемых запросу в качестве параметров. Это массив объектов: каждый объект содержит два свойства, name и value, в которых содержатся имя и значение параметра запроса. • Обернутый набор jQuery, к которому применяется команда. • Объект options, переданный команде. Если это свойство опущено, то функция предварительной обработки не вызывается.
success	(функция) Определяет функцию обратного вызова, вызываемую после успешного завершения запроса. Этой функции передаются три параметра: • Тело ответа, прошедшее предварительную обработку в соответствии со значением свойства dataType. • Строка *success*. • Обернутый набор jQuery, к которому применяется команда. Если это свойство опущено, то эта функция не вызывается. Если это свойство – единственное, указанное при обращении к команде, то эту функцию можно передать непосредственно команде вместо объекта options. Обратите внимание: не делается никаких предположений о возвращаемом значении функции.
clearForm	(логическое значение) Если имеет значение true, то после благополучной отправки форма очищается. Семантика очистки объясняется в описании команды clearForm().
resetForm	(логическое значение) Если имеет значение true, то после благополучной отправки форма сбрасывается. Семантика отпускания значений объясняется в описании команды resetForm().
semantic	(логическое значение) Если имеет значение true, то значения упорядочиваются в соответствии с семантикой формы. Порядок следования параметров имеет значение, только если отправляются параметры элементов ввода типа *image*, когда форма отправляется щелчком на таком элементе. Поскольку эта обработка может быть долгой, данный параметр следует использовать только если порядок следования параметров имеет значение для обработки на стороне сервера и в форме используются графические элементы ввода.
прочие параметры	Здесь можно указать любые другие параметры, доступные в базовой функции $.ajax() библиотеки jQuery и описанные в табл. 8.2, и они будут передаваться в вызовы функций низкого уровня.

Несмотря на большое число параметров, в большинстве случаев вызовы команды ajaxSubmit() выглядят очень просто. Если нужно только отправить форму на сервер (и не выполнять никаких действий после отправки), то вызов команды выглядит по-спартански:

```
$('#targetForm').ajaxSubmit();
```

Если текст ответа нужно передать функции обратного вызова:

```
$('#targetForm').ajaxSubmit(function(response){
  /* Выполнить обработку ответа */
});
```

И так далее. Поскольку все параметры имеют достаточно разумные значения по умолчанию, достаточно определить лишь желаемые для нас параметры.

Предупреждение

Поскольку объект-хеш options передается функции обратного вызова beforeSubmit(), у вас может появиться желание изменить его. Будьте осторожны! На этом этапе уже слишком поздно изменять значение свойства beforeSubmit, так как эта функция уже была вызвана, но вы можете изменить другие, более простые настройки, такие как resetForm или clearForm. Будьте внимательны к другим изменениям, они могут привести в ошибочному выполнению операций. Ни в коем случае не изменяйте значение свойства semantic, потому что к моменту вызова beforeSubmit его действие уже закончилось.

Мы не удивимся, если вы спросите – нет ли лабораторной страницы для этой команды? Откройте файл *chapter9/form/lab.ajaxSubmit.html* в броузере и вы увидите страницу, показанную на рис. 9.3.

Примечание

Обратите внимание: поскольку мы собираемся направлять запросы серверу, вам следует запускать эту страницу под управлением веб-сервера, как описывалось в примерах главы 8 (раздел 8.2).

Эта лабораторная страница содержит уже знакомую форму, на которую мы можем воздействовать командой ajaxSubmit(). В самой верхней панели находится сама форма. В середине находится панель управления, позволяющая добавлять и убирать параметры resetForm и clearForm из вызова. В панели результатов будут отображаться: данные, отправленные в запросе, свойства объекта options, переданного команде, и тело ответа.

Если вы решите ознакомиться с исходным кодом лабораторной страницы, то без труда обнаружите, что первые два блока результатов отображаются функцией обратного вызова beforeSubmit, а третий блок представляет собой элемент, определяемый свойством target. (Для большей ясности, функция beforeSubmit не перечисляется в блоке параметров запроса.)

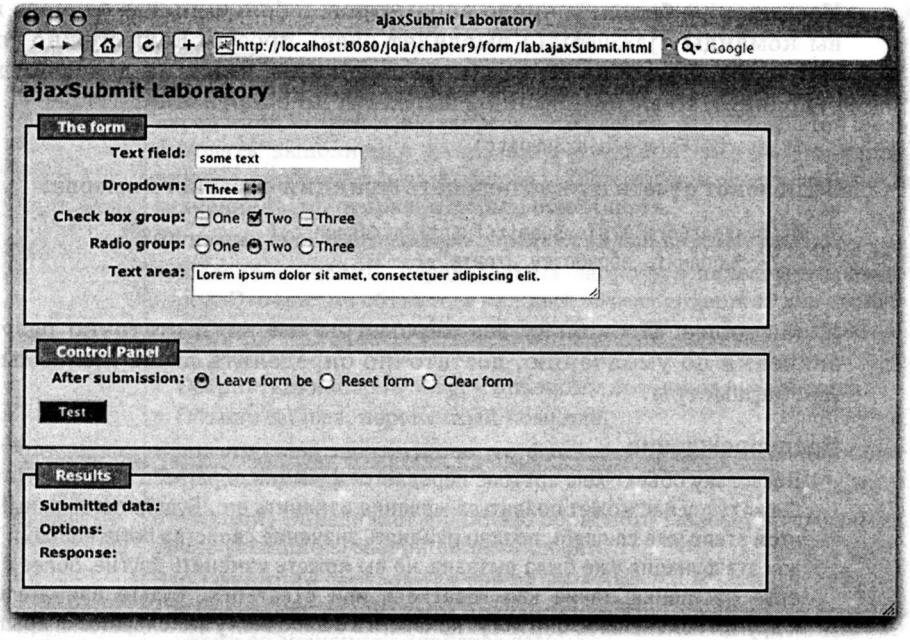


***Рис. 9.3.** Лабораторная страница ajaxSubmit Laboratory позволяет экспериментировать с методом ajaxSubmit( )*

После щелчка на кнопке Test (тест) инициируется запрос за счет применения команды `ajaxSubmit()` к обернутому набору, содержащему форму в верхней части страницы. URL-адрес запроса по умолчанию принимается равным значению свойства `action` формы: `reflectData.jsp`, этот ресурс формирует ответ в формате HTML, отображающий данные, которые получены сервером с запросом.

Оставьте все элементы управления в исходном состоянии и щелкните на кнопке Test (тест) – и вы увидите результаты, как показано на рис. 9.4.

В разделе Submitted data (отправленные данные), как и следовало ожидать, появились имена и значения всех успешных элементов управления. Обратите внимание на отсутствие неотмеченных флажков и переключателей. Это в точности имитирует обычную операцию отправки формы.

В разделе Options (параметры) показаны параметры, с которыми был выполнен запрос, что позволяет нам увидеть изменения при изменении параметров в разделе Control Panel (панель управления). Например, если отметить флажок Reset Form (сбросить значения элементов формы) и щелкнуть на кнопке Test (тест), то мы увидим, как параметр `resetForm` добавится к вызову метода.

Параметры, обнаруженные серверным ресурсом (по умолчанию – JSP), отображаются последними. Мы можем сравнить ответ с данными

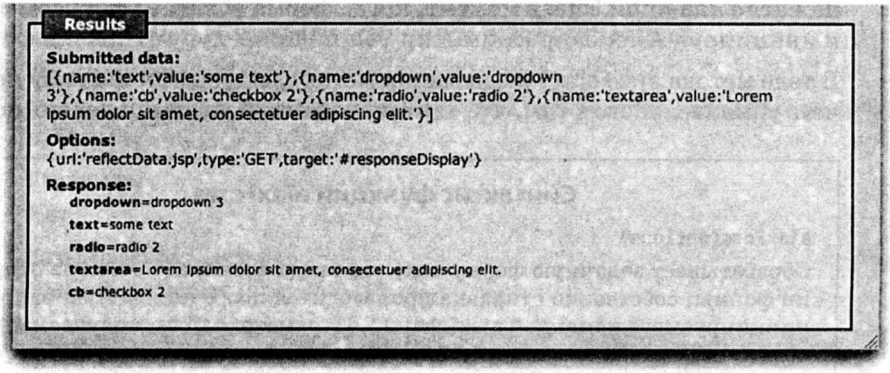


***Рис. 9.4.** В панели Results (результаты) мы можем наблюдать данные, отправляемые запросом, параметры вызова команды и тело ответа, которое содержит данные, полученные серверным ресурсом*

в блоке Submitted data (отправленные данные), лишний раз убедившись, что все в порядке.

Поэкспериментируйте с этой страницей, пробуя различные комбинации параметров и изменяя данные в форме. Это поможет понять работу метода `ajaxSubmit()`.

В этом разделе мы исходили из предположения, что отправка формы инициируется под управлением сценария. Это может потребоваться, когда обрабатываемое событие отличается от обычного события отправки формы, например, в результате щелчка на кнопке, отличной от кнопки `submit` (как в этой лабораторной странице), или по такому событию от мыши, которое использовалось для вызова модуля The Termifier в примерах главы 8. Но иногда (или даже часто) запрос отправки формы будет результатом обычного события отправки формы.

Давайте посмотрим, сможет ли модуль Form Plugin помочь нам в этом.

Перехват события отправки формы

Метод `ajaxSubmit()` прекрасно подходит для случаев, когда отправкой формы должен управлять сценарий на основе события, отличающегося от события отправки формы. Однако зачастую нам требуется возможность инициировать отправку формы обычным образом, при этом перехватывая выполнение операции и отправляя форму на сервер в виде запроса Ajax, а не методом, вызывающим полную перезагрузку страницы.

Мы могли бы самостоятельно выполнить отправку формы, применив знания в области обработки событий и метод `ajaxSubmit()`. Но оказывается, что это не требуется: все нужное сделает метод `ajaxForm()` модуля Form Plugin.

Этот метод обрабатывает форму, блокируя попытку ее отправки по одному из обычных событий (такому как щелчок на кнопке `submit` или

нажатие клавиши Enter в момент, когда форма обладает фокусом ввода) и инициируя Ajax-запрос, имитирующий процедуру отправки формы.

В теле метода `ajaxForm()` вызывается метод `ajaxSubmit()`, поэтому нет ничего удивительного в том, что синтаксис этих двух команд очень похож:

Синтаксис функции ajaxForm

`ajaxForm(options)`

Обрабатывает заданную форму так, что когда возникает событие отправки формы, собственно отправка производится посредством Ajax-запроса, инициируемого командой `ajaxSubmit()`. Параметр `options`, получаемый методом, передается команде `ajaxSubmit()`.

Параметры

`options` (объект | функция) Объект-хеш, содержащий свойства, описанные в табл. 9.1. Если единственный параметр, который требуется передать, является функцией обратного вызова, запускаемой в случае успешного завершения запроса, ее можно передать вместо объекта.

Возвращаемое значение

Обернутый набор.

Как правило, команда `ajaxForm()` применяется к форме в обработчике события готовности документа, где можно позволить ей подготовить форму к отправке предусмотренным нами способом.

Вполне возможно, даже общепринято, объявить разметку формы HTML так, как будто она будет передаваться на сервер обычным способом, и позволить команде `ajaxForm()` самой выполнить настройки, исходя из объявления формы. В случае если пользователь запретит использование JavaScript, функциональные возможности такой формы несколько ухудшатся, но она все же будет отправляться на сервер обычным способом, пусть и минуя всю нашу специальную обработку. Довольно удобно!

Если в какой-то момент, после того как форма была обработана командой `ajaxForm()`, нам потребуется убрать выполненные настройки, чтобы обеспечить передачу формы обычным способом, можно воспользоваться командой `ajaxFormUnbind()`.

Для любителей лабораторных страниц мы создали страницу ajaxForm Laboratory, которая находится в файле *chapter9/form/lab.ajaxForm.html*. В окне броузера эта страница выглядит, как показано на рис. 9.5.

Эта лабораторная страница выглядит и работает практически так же, как страница ajaxSubmit Laboratory, но имеет следующие важные отличия:

- Кнопка Test (тест) отсутствует, вместо нее добавлена кнопка Submit me! (отправь меня).

Синтаксис функции ajaxFormUnbind

`ajaxFormUnbind()`

Удаляет изменения из формы в обернутом наборе, чтобы отправка ее производилась обычным образом.

Параметры

Нет

Возвращаемое значение

Обернутый набор.

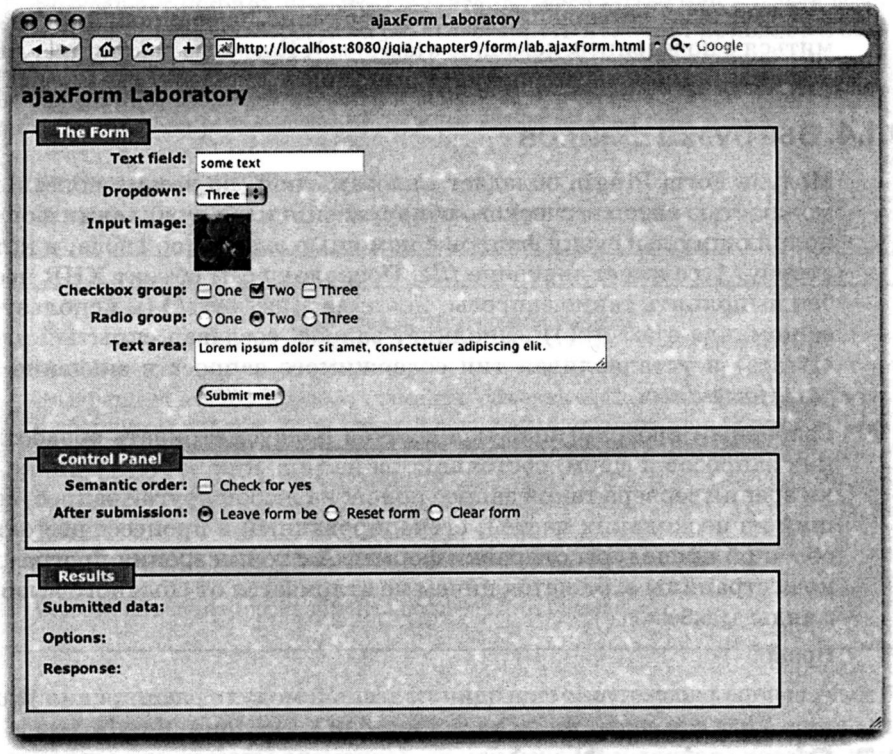


Рис. 9.5. Лабораторная страница ajaxForm Laboratory позволяет исследовать перехват события отправки формы и передачу данных в Ajax-запросе

- В разделе Control Panel (панель управления) появился флажок, позволяющий указать, следует ли добавлять свойство `semantic` в параметры запроса.

- Был добавлен элемент ввода со значением атрибута `type`, равным *image*, чтобы мы могли наблюдать различия, которые возникают, когда свойство `semantic` устанавливается равным значению `true`.

Эту форму можно отправить следующими тремя способами:

- щелчком на кнопке Submit me! (отправь меня);
- нажатием клавиши Enter, когда фокус ввода находится в одном из элементов формы, способных принимать его;
- щелчком на элементе управления Input image (цветок).

В любом из этих случаев вы увидите, что страница не обновляется, то есть форма отправляется с помощью Ajax-запроса, информация о котором отображается в нижней части страницы. Поэкспериментируйте с элементами управления на этой странице, чтобы поближе познакомиться с поведением команды `ajaxForm()`. Когда вы закончите, мы продолжим знакомство с модулем Form Plugin.

9.1.4. Выгрузка файлов

Модуль Form Plugin обладает малозаметной, но весьма полезной возможностью автоматического обнаружения и обслуживания форм, выполняющих выгрузку файлов с помощью элементов ввода, в которых атрибут `type` имеет значение *file*. Поскольку сам объект XHR неспособен выполнять такие запросы, команда `ajaxSubmit()` (и использующая ее команда `ajaxForm()`) выполняет запрос, создавая скрытый элемент `<iframe>` и устанавливая тип содержимого запроса в значение *multipart/form-data*.

Серверный программный код должен предусматривать обработку таких запросов и форм, состоящих из нескольких частей. Однако с точки зрения сервера такой запрос похож на любой другой запрос, состоящий из нескольких частей, сгенерированный в процессе выполнения обычной процедуры отправки формы. А с точки зрения программного кода страницы этот метод ничем не отличается от обычного вызова команды `ajaxSubmit()`.

Браво!

А теперь рассмотрим еще один полезный модуль расширения jQuery.

9.2. Dimensions Plugin

При создании полнофункциональных интернет-приложений порой совершенно необходимо знать точное местоположение и размеры элемента. Например, если нужно реализовать раскрывающееся меню, которое должно появляться в точной позиции вызвавшего его элемента.

В библиотеке jQuery имеются команды `width()`, `height()` и `offset()`, но отсутствует возможность определения точного местоположения элемента. Здесь нам на выручку приходит модуль расширения Dismensions Plugin.

Давайте рассмотрим его прикладной интерфейс.

9.2.1. Улучшенные методы width и height

Модуль Dimensions Plugin расширяет действие команд `width()` и `height()`, чтобы с их помощью можно было получать ширину и высоту объектов `window` и `document`, поскольку это не всегда доступно базовым командам библиотеки. Синтаксис улучшенных команд:

Синтаксис функции width

`width()`

Возвращает ширину первого элемента, объекта `window` или объекта `document` в обернутом наборе. Если первый обернутый элемент не является объектом `window` или объектом `document`, вызывается базовая команда jQuery.

Параметры

Нет

Возвращаемое значение

Ширина объекта `window`, объекта `document` или элемента.

Синтаксис функции height

`height()`

Возвращает высоту первого элемента, объекта `window` или объекта `document` в обернутом наборе. Если первый обернутый элемент не является объектом `window` или объектом `document`, вызывается базовая команда jQuery.

Параметры

Нет

Возвращаемое значение

Высота объекта `window`, объекта `document` или элемента.

Эти улучшенные команды не противоречат семантике одноименных базовых команд, когда им передается входной параметр (для изменения ширины или высоты элемента), *за исключением* случая, когда первым элементом обернутого набора является объект `window` или объект `document`. В таких случаях команда действует так, как если бы входной параметр вообще отсутствовал, и возвращается ширина или высота объекта `window` или объекта `document`. Помните об этом и не совершайте ошибок.

Команды `width()` и `height()` возвращают размеры *содержимого* элемента, но иногда требуется учитывать и другие аспекты блочной модели, такие как ширина отступов и толщина рамок, окружающих элемент. Для таких случаев модуль Dimensions Plugin предоставляет набор команд, возвращающих размеры с учетом этих аспектов.

Первые из этих команд, `innerWidth()` и `innerHeight()`, возвращают размеры не только содержимого элемента, но и всех отступов, окружающих его. Команды второй группы, `outerWidth()` и `outerHeight()`, включают не только отступы, но также рамку и, при желании, поля.

Синтаксис функций innerWidth и innerHeight

`innerWidth()`

`innerHeight()`

Возвращает *внутреннюю* ширину или высоту первого элемента. Под внутренними размерами подразумеваются размеры содержимого элемента и все отступы вокруг него.

Параметры

Нет

Возвращаемое значение

Внутренняя ширина или высота первого элемента в обернутом наборе.

Синтаксис функций outerWidth и outerHeight

`outerWidth(options)`

`outerHeight(options)`

Возвращает *внешнюю* ширину или высоту первого элемента. Под внешними размерами подразумеваются размеры содержимого элемента, включая все отступы и рамки вокруг него.

Параметры

`options` (объект) Объект-хеш, который содержит единственное свойство `margin`. Значение этого свойства определяет, следует ли при расчетах учитывать поля. Значение по умолчанию – `false`.

Возвращаемое значение

Внешняя ширина или высота первого элемента в обернутом наборе.

Примечательно, что все эти методы определения внутренних и внешних размеров действуют одинаково в случае их применения к объектам `window` и `document`.

9.2.2. Определение размеров прокручиваемых областей

Когда-то давно для пользовательских интерфейсов всех типов настоящей проблемой было несоответствие размеров содержимого и области, выделенной для его размещения. Эта проблема была решена с помощью полос прокрутки, которые позволяют пользователю прокручивать содержимое, не умещающееся в ограниченной области просмотра. Веб-интерфейс не исключение – содержимое часто не умещается в отведенную для него область.

Иногда мы хотим узнать, как далеко было прокручено содержимое окна или элемента, чтобы добавить новое содержимое (или переместить имеющееся) с учетом текущей позиции прокрутки в окне или в элементе. Кроме того, нам может потребоваться изменить позицию прокрутки окна или элемента.

Модуль Dimensions Plugin позволяет получить или изменить позицию прокрутки таких элементов с помощью методов `scrollTop()` и `scrollLeft()`.

Синтаксис функций scrollTop и scrollLeft

`scrollTop(value)`

`scrollLeft(value)`

Получает или устанавливает позицию прокрутки объекта `window`, объекта `document` или прокручиваемого содержимого элемента. Элементы, допускающие возможность прокрутки содержимого, – это элементы со стилями CSS `overflow`, `overflow-x` или `overflow-y`, имеющими значение `scroll` или `auto`.

Параметры

`value` (число) Значение (в пикселах), определяющее позицию, в которую следует установить вертикальную или горизонтальную полосу прокрутки. В случае передачи ошибочного значения, по умолчанию принимается значение `0`. Если этот параметр опущен, возвращаются текущие значения вертикальной и горизонтальной позицию прокрутки.

Возвращаемое значение

Если входной параметр определен, возвращается обернутый набор. В противном случае возвращается запрошенная величина.

Для объектов `window` и `document` возвращается одно и то же значение.

Хотите исследовать эти методы с помощью лабораторной страницы? Если да, откройте в броузере файл *chapter9/dimensions/lab.scroll.html* и вы увидите страницу Scrolling Lab (лабораторная страница прокрутки), показанную на рис. 9.6.

Эта лабораторная страница позволяет изменять значения вертикальной и горизонтальной прокрутки как для объекта исследований, так и для самого окна. Для отображения величины прокрутки в ней используются версии команд `scrollTop()` и `scrollLeft()`, которые *возвращают* значения.

Панель Lab Control Panel (панель управления) содержит два текстовых поля для ввода числовых значений, которые будут передаваться командам `scrollTop()` и `scrollLeft()`, а также три переключателя, позволяющих выбрать объект применения команд. Можно выбрать вариант window (объект `window`), document (объект `document`) или test subject (объект исследований – элемент `<div>`). Обратите внимание: мы устанавливаем

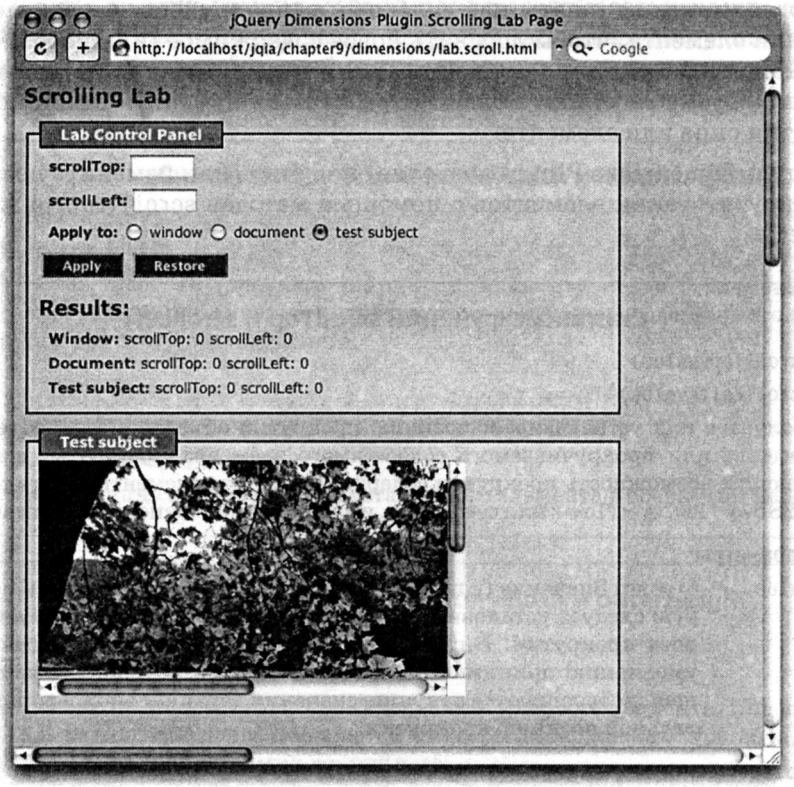


***Рис. 9.6.** Лабораторная страница Scrolling Lab позволяет исследовать методы scrollTop( ) и scrollLeft( )*

ширину и высоту элемента `<body>` равными 2000 пикселам, чтобы заставить объект `window` отобразить полосы прокрутки.

Щелчок на кнопке Apply (применить) применяет заданные значения к указанному объекту, а кнопка Restore (восстановить) устанавливает значения позиций прокрутки равными `0`. Под кнопками находится раздел, где в режиме реального времени отображаются текущие значения для всех трех объектов.

В панели Test subject находится объект исследований – элемент `<div>` размером 360×200 пикселов, содержащий изображение, размеры которого намного больше размеров области отображения.

CSS-свойство `overflow` этого элемента имеет значение `scroll`, что вызывает появление полос прокрутки, благодаря которым мы можем перемещать изображение в области просмотра.

С помощью этой страницы выполните следующие упражнения:

- *Упражнение 1.* Попробуйте просмотреть изображение с помощью полос прокрутки в панели Test subject (объект исследований). Обра-

тите внимание на изменение значений в разделе Results (результаты). Эти значения извлекаются с помощью команд scrollTop() и scrollLeft() в обработчике события scroll элемента.

- *Упражнение 2.* Повторите упражнение 1, но на этот раз попробуйте перемещаться по странице с помощью полос прокрутки окна. Обратите внимание на изменение значений в разделе Results (результаты) (разумеется, если он в пределах видимой области), а также на то, что значения прокрутки в строке Document (объект document) жестко связаны со значениями в строке Window (объект window), – это еще раз доказывает, что для этих двух объектов методы возвращают одинаковые значения. Мы нарочно включили в исследования объекты window и document, чтобы убедить вас в этом.
- *Упражнение 3.* Щелкните на кнопке Restore (восстановить), чтобы вернуть все полосы прокрутки в *нормальное* положение. Выберите переключатель test subject (объект исследований) и введите в текстовые поля ввода числовые значения, например 100 и 100. Щелкните на кнопке Apply (применить). Ого! Какой красивый водопад! Попробуйте ввести другие значения и посмотрите, как они воздействуют на объект исследований, когда в результате щелчка на кнопке Apply вызываются методы scrollTop() и scrollLeft().
- *Упражнение 4.* Повторите упражнение 3, выбрав в качестве объекта исследований объект window.
- *Упражнение 5.* Повторите упражнение 3, выбрав в качестве объекта исследований объект document. Убедитесь, что результаты исследований для объектов window и document совпадают.

9.2.3. Смещение и позиция

На первый взгляд, определить местоположение элемента просто. Все, что нужно сделать, – это узнать положение элемента относительно начала координат окна, так? А вот и нет.

Когда к элементу применяются значения свойств CSS top и left, эти величины измеряются относительно *вмещающего элемента*. В простейших случаях вмещающим элементом является окно (точнее – элемент <body> страницы, загруженный в окно). Но если какой-либо из вмещающих элементов имеет свойство CSS position со значением relative или absolute, ближайший такой вмещающий элемент и будет служить началом координат для элемента. Это понятие также называют *контекстом позиционирования* элемента.

При определении местоположения элемента важно знать, какие координаты запрашивать – относительно начала координат окна или относительно вмещающего элемента? Нужно помнить и о других факторах. Например, учитываются ли рамки.

Модуль Dimensions Plugin позволяет выяснить все это, начиная с ближайшего вмещающего элемента, играющего роль начала координат.

Синтаксис функции offsetParent

`offsetParent`

Возвращает вмещающий элемент, который играет роль начала координат (контекст позиционирования), для первого элемента в обернутом наборе. Это ближайший вмещающий элемент, свойство `position` которого имеет значение `relative` или `absolute`, или элемент `<body>`, если такие вмещающие элементы отсутствуют. Этот метод должен применяться только к видимым элементам.

Параметры

Нет

Возвращаемое значение

Вмещающий элемент, который играет роль начала координат.

Получить позицию элемента относительно вмещающего элемента, который играет роль начала координат, можно с помощью команды `position()`.

Синтаксис функции position

`position()`

Возвращает координаты (`top` и `left`) первого элемента в обернутом наборе относительно вмещающего элемента, который играет для него роль начала координат.

Параметры

Нет

Возвращаемое значение

Объект с двумя свойствами, `top` и `left`, содержащими значения координат элемента относительно контекста позиционирования (вмещающего элемента, который играет роль начала координат).

Эта команда удобна, если нужно переместить элемент относительно его текущего местоположения, но иногда требуется определить позицию элемента относительно элемента `<body>` (без учета местоположения вмещающего элемента, который играет роль начала координат) и получить чуть больший контроль над тем, как производятся вычисления. Для таких случаев модуль Dimensions Plugin предоставляет команду `offset()`.

Чтобы гарантировать точность значений, возвращаемых методом, размеры и координаты элементов на странице должны быть выражены в пикселах. С параметрами по умолчанию этот метод обычно дает точные результаты, но если нас больше интересует скорость выполнения, чем точность (например, когда этот метод используется в тяжелом

Синтаксис функции offset

`offset(options, result)`

Возвращает информацию о местоположении первого элемента в обернутом наборе. По умолчанию местоположение определяется относительно элемента `<body>`, а порядок вычислений определяется свойствами объекта `options`.

Параметры

`options` (объект) Объект-хеш, параметры которого управляют тем, как метод производит вычисления. Допустимы следующие значения:

- `relativeTo` (элемент) Указывает вмещающий элемент, относительно которого будут вычисляться координаты обернутого элемента. Этот элемент должен иметь свойство `position` со значением `relative` или `absolute`. Если опущен, то в качестве начала координат по умолчанию используется элемент `<body>`.
- `lite` (логическое значение) Указывает, следует ли пропустить некоторые способы оптимизации, характерные для определенных типов броузеров. Это увеличит скорость вычислений за счет снижения точности. Значение по умолчанию – `false`.
- `scroll` (логическое значение) Указывает, следует ли учитывать величину прокрутки. Значение по умолчанию – `true`.
- `padding` (логическое значение) Указывает, должны ли включаться в расчеты величины отступов. Значение по умолчанию – `false`.
- `border` (логическое значение) Указывает, должна ли включаться в расчеты толщина рамок. Значение по умолчанию – `false`.
- `margin` (логическое значение) Указывает, должна ли включаться в расчеты ширина полей. Значение по умолчанию – `true`.

`results` (объект) Необязательный объект, в котором должны сохраняться результаты вычислений. Если опущен, то создается новый объект, заполняется результатами и возвращается методом. Этот параметр удобен, если нужно включить вызов метода в цепочку команд jQuery.

Возвращаемое значение

Обернутый набор, если был указан объект `results`, в противном случае – вновь созданный объект `results`. Объект `results` содержит свойства `top` и `left`, а также `scrollTop` и `scrollLeft` при условии, что параметр `scroll` не был явно установлен в значение `false`.

цикле для вычисления координат множества элементов), в такой ситуации можно попробовать посмотреть, как повлияют на производительность параметр `lite` и другие настройки.

А теперь покинем царство координат и размеров и познакомимся еще с одним модулем расширения, способным повысить динамику страницы и улучшить обработку событий.

9.3. Live Query Plugin

Вникая в высокоуровневые концепции, представленные в этой книге, вы могли заметить, что jQuery сильно влияет на структуру страниц, написанных с прицелом на эту библиотеку. Следуя положениям концепции ненавязчивого JavaScript, наши страницы обычно содержат HTML-разметку и обработчик события готовности документа, определяющий поведение страницы, включая установку обработчиков событий в элементы, определенные в теле страницы.

Библиотека jQuery существенно упрощает не только оформление страниц, но и изменение этих страниц на протяжении всей их жизни. В течение жизненного цикла страницы многие элементы DOM просто отсутствуют на момент запуска обработчика события готовности документа и добавляются в дерево DOM позже. Добавляя новые элементы, нередко приходится подключать к ним обработчики событий, чтобы определить их поведение, как это делается для элементов, изначально присутствующих в странице. Начинающий автор веб-страниц мог бы добавить значительный объем дублированного программного кода методом «копирования и вставки», но более опытные разработчики выносят общие элементы программы в функции или в классы JavaScript.

Но разве не лучше было бы заранее объявить в обработчике события готовности документа поведение всех появляющихся на странице элементов независимо от того, присутствуют ли они в момент ее загрузки?

Похоже на несбыточные мечты? А вот и нет – с модулем Live Query Plugin это возможно!

Модуль Live Query Plugin позволяет связать следующие варианты поведения с элементами DOM, основываясь на их соответствии определяемому нами селектору jQuery:

- подключать обработчики событий к элементам, соответствующим селектору;
- вызывать функцию, когда будет обнаружен элемент, соответствующий селектору;
- вызывать функцию, когда какой-либо элемент перестанет соответствовать селектору.

Модуль Live Query Plugin также позволяет нам отключить любой из этих вариантов поведения в любой момент времени.

Знакомство с модулем Live Query Plugin мы начнем с того, что выясним, как с его помощью подключить обработчик события к элементу, которого еще нет.

9.3.1. Упреждающая установка обработчиков событий

Модуль Live Query позволяет устанавливать обработчики событий заранее – обработчики событий подключаются к элементам, которые соответствуют селектору jQuery в данный момент или будут соответство-

вать впоследствии. Обработчики подключаются не только к имеющимся элементам, соответствующим селектору на момент установки, но и к любым другим элементам, которые будут соответствовать селектору позднее, в течение жизненного цикла страницы, в том числе к имеющимся элементам, которые могут измениться так, что будут соответствовать селектору и вновь созданным элементам.

Когда эти элементы изменятся так, что перестанут соответствовать селектору, обработчики событий, подключенные модулем Live Query, автоматически отключаются от этих элементов.

Изменения, влияющие на соответствие элемента селектору, могут не зависеть от применения методов jQuery. При небрежном обращении к элементам из-за пределов jQuery модуль Live Query, безусловно, утратит способность следить за ними. Если нам абсолютно необходимо выполнить изменения не под управлением jQuery, модуль Live Query даст нам такую возможность, но об этом позже.

Все механизмы поведения Live Query, в том числе обработчики событий, подключаются к элементам с помощью метода `livequery()`. Формат вызова метода для выполнения упреждающей установки обработчиков событий:

Синтаксис функции livequery

`livequery(event,listener)`

Подключает функцию `listener` как обработчик события типа `event` ко всем элементам в обернутом наборе и к любым другим элементам, которые будут соответствовать селектору обернутого набора в будущем.

Параметры

`event`	(строка) Тип события, которое будет обрабатываться подключаемым обработчиком. Тот же набор типов событий используется командой `bind()` jQuery.
`listener`	(функция) Функция, которая будет установлена как обработчик события. Контекст функции (`this`) при каждом вызове будет ссылаться на соответствующий элемент.

Возвращаемое значение

Обернутый набор.

При такой форме вызова метод `livequery()` в точности соответствует команде `bind()`. Как и `bind()`, он подключает указанный обработчик ко всем элементам в обернутом наборе. Но он также автоматически подключит обработчик ко всем элементам, которые будут соответствовать селектору в течение жизненного цикла страницы. Кроме того, он автоматически отключит обработчик от любых элементов, переставших соответствовать селектору, включая и те, что изначально входили в состав обернутого набора.

Это чрезвычайно мощная возможность. Она позволяет определять поведение элементов, соответствующих селектору, всего *однажды*, в обработчике события готовности документа, не заботясь об отслеживании изменений в элементах или добавлений новых элементов в страницу. Разве не круто?

Подключение обработчиков событий – это лишь частный случай, а общий случай, заслуживающий особого внимания, – это выполнение действий при таком изменении (или при добавлении) элементов, когда они начинают или перестают соответствовать исходному селектору. В эти моменты мы могли бы делать много интересного, и модуль Live Query не разочарует нас.

9.3.2. Определение обработчиков событий начала и конца периода соответствия

Выполнить некоторые действия (помимо подключения или отключения обработчиков событий) в те моменты, когда элементы начинают или перестают соответствовать некоторому селектору, поможет другая форма вызова команды `livequery()`.

Синтаксис функции livequery

`livequery(onmatch, onmismatch)`

Подключает функции обратного вызова, вызываемые в момент прихода элемента в состояние или выхода элемента из состояния соответствия селектору обернутого набора.

Параметры

`onmatch` (функция) Определяет функцию-*обработчик события прихода в состояние соответствия* (*match listener*). Эта функция будет вызываться для всех элементов (передаваемых обработчику как контекст функции), пришедших в состояние соответствия селектору. Если во время вызова этого метода отыщутся какие-либо имеющиеся элементы, соответствующие селектору, функция немедленно будет вызвана для каждого из них.

`onmismatch` (функция) Определяет необязательную функцию-*обработчик события выхода из состояния соответствия* (*mismatch listener*). Эта функция будет вызываться для всех элементов (передаваемых обработчику как контекст функции), вышедших из состояния соответствия селектору. Если этот параметр опущен, *обработчик события выхода из состояния соответствия* не устанавливается.

Возвращаемое значение

Обернутый набор.

Если требуется установить только необязательный обработчик события выхода из состояния соответствия, мы не сможем сделать это, передав методу значение null в первом параметре, потому что в такой ситуации второй параметр будет установлен как обработчик события перехода в состояние соответствия, как если бы мы передали его в первом параметре. Вместо этого в первом параметре следует передать пустую функцию:

```
$('div.whatever').livequery(
  function(){},
  function(){ /* Обработка события выхода из состояния соответствия */ }
);
```

Как и в случае обработчиков событий, подключаемых с помощью Live Query, изменения, вызывающие переход в состояние соответствия или выход из состояния соответствия, автоматически приведут к запуску этих функций, при условии, что эти изменения были выполнены средствами библиотеки jQuery. Но как быть, если выполнить эти изменения с помощью jQuery невозможно?

9.3.3. Принудительный запуск обработчиков Live Query

Если с помощью сторонних средств, не имеющих отношения к библиотеке jQuery, вызвать изменения в элементах, в результате которых они перешли в состояние соответствия или вышли из состояния соответствия, то можно заставить Live Query вызвать соответствующие обработчики, подключенные к таким элементам, с помощью вспомогательной функции.

Синтаксис функции $.livequery.run

`$.livequery.run()`

Заставляет Live Query выполнить глобальную обработку элементов и вызвать соответствующие обработчики. Это удобно, когда изменение элементов происходит не под управлением методов библиотеки jQuery.

Параметры

Нет

Возвращаемое значение

Не определено.

Зная, что Live Query автоматически отключает обработчики событий, когда элемент выходит из состояния соответствия, мы можем установить обработчик этого события, чтобы выполнить дополнительные действия при выходе из состояния соответствия. Но как быть, если нам потребуется совсем убрать обработчики?

9.3.4. Удаление обработчиков Live Query

Как jQuery предоставляет команду `unbind()`, отменяющую действие команды `bind()`, так и модуль Live Query предоставляет собственную команду, отменяющую операцию подключения обработчиков событий Live Query и обработчиков входа/выхода из состояния соответствия – команду `expire()`, которая поддерживает несколько форматов параметров.

Синтаксис функции expire

```
expire()
expire(event,listener)
expire(onmatch,onmismatch)
```

Удаляет обработчики, связанные с селектором обернутого набора. При вызове без параметров удаляются все обработчики, связанные с селектором. Действие команды при вызове с параметрами приведено в описании параметров.

Параметры

`event`	(строка) Указывает тип события, обработчики которого должны быть удалены. Если параметр `listener` не указан, удаляются все обработчики этого типа события.
`listener`	(функция) Если этот параметр указан, то из селектора удаляется только данный конкретный обработчик (для указанного типа события).
`onmatch`	(функция) Указывает обработчик события перехода в состояние соответствия, который должен быть удален из селектора.
`mismatch`	(функция) Если присутствует, то указывает обработчик события выхода из состояния соответствия, который должен быть удален из селектора.

Возвращаемое значение

Обернутый набор.

Чтобы помочь понять работу этих методов, мы создали лабораторную страницу. Откройте в броузере файл *chapter9/livequery/lab.livequery.html*, чтобы увидеть страницу, показанную на рис. 9.7.

На этой лабораторной странице есть три панели: Control Panel (панель управления) с кнопками, которые выполняют некоторые интересные действия, контейнер Test Subjects (объекты исследований) и Console (консоль), где отображаются сообщения, информирующие нас о происходящих событиях.

Нам хотелось бы дать некоторые пояснения к внутреннему устройству этой страницы. В обработчике события готовности документа выполняется настройка кнопок панели Control Panel для выполнения соответствующих им действий по щелчку (за более подробной информацией

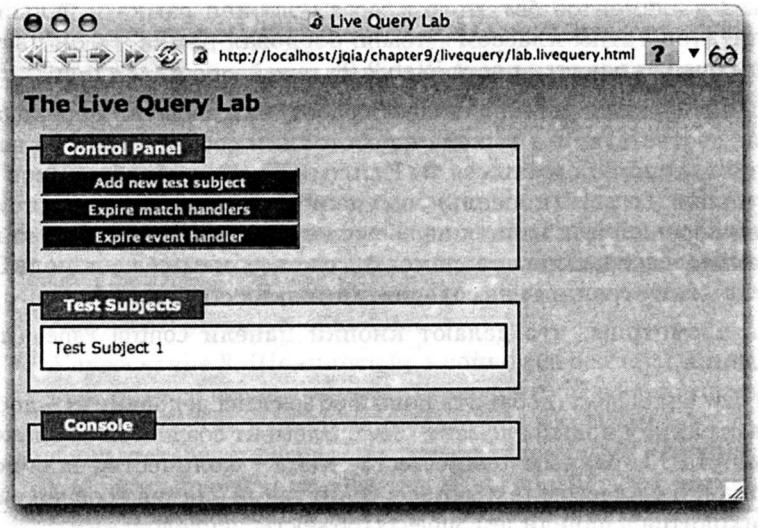


Рис. 9.7. Лабораторная страница Live Query Lab позволяет исследовать модуль Live Query в действии

обращайтесь к файлу страницы) посредством двух инструкций Live Query:

```
$('div.testSubject').livequery('click',
  function(){
    $(this).toggleClass('matched');
  }
);
$('div.matched').livequery(
  function(){ reportMatch(this,true); },
  function(){ reportMatch(this,false); }
);
```

❶ Упреждающая установка обработчика события click

❷ Установка обработчиков событий перехода в состояние соответствия и выхода из состояния соответствия

Первая из этих инструкций подключает обработчик события Live Query ко всем элементам `<div>` с классом `testSubject` ❶. Таким элементом является единственный элемент-объект исследований, который уже присутствует в странице в момент ее загрузки (рис. 9.7), а также все элементы-объекты исследований, которые будут создаваться щелчком на кнопке Add new test subject (добавить новый объект исследований). Обработчик события `click`, который мы вскоре обсудим, устанавливается не только в имеющийся элемент-объект исследований – он также будет *автоматически* подключен ко всем элементам, добавляемым в страницу в будущем (то есть ко всем создаваемым элементам `<div>` с классом `testSubject`).

Обработчик события `click` переключает класс `matched` в целевом элементе события. Чтобы проще было увидеть, какие объекты исследований имеют этот класс, а какие нет, мы определили правила CSS, согласно

которым у элементов *без* этого класса – черная рамка и белый фон, а у элементов *с* этим классом – темно-бордовая рамка и фон цвета хаки. Нам очень нравятся цвета Web 2.0!

Вторая инструкция устанавливает обработчики событий перехода в состояние соответствия и выхода из состояния соответствия во все элементы `<div>` с классом `matched` ❷. Каждый из этих обработчиков выводит в панель Console (консоль) сообщение, свидетельствуя о том, что элемент перешел или вышел из состояния соответствия. Так как первоначально на странице нет элементов с классом `matched`, панель Console при открытии страницы не содержит никаких сообщений.

Теперь посмотрим, что делают кнопки панели Control Panel (панель управления):

- Add New Test Subject (добавить новый объект исследований) – добавляет в страницу новый элемент `<div>`. Элемент создается со значением атрибута `id`, равным `testSubject#`, где `#` – количество щелчков на кнопке, и с классом `testSubject`. Один такой элемент создан заранее и находится в панели Test Subjects (объекты исследований).
- Expire Match Handlers (удалить обработчики событий изменения состояния соответствия) – запускает инструкцию `$('div.matched').expire();`, которая удаляет обработчики событий перехода в состояние соответствия и выхода из состояния соответствия, установленные в обработчике события готовности документа.
- Expire Event Handlers (удалить обработчики событий) – запускает инструкцию `$('div.testSubject').expire();`, которая удаляет упреждающие обработчики событий, установленные в обработчике события готовности документа.

Теперь, когда мы разобрались с тем, как работает лабораторная страница, попробуйте выполнить несколько упражнений:

- *Упражнение 1.* Откройте страницу и щелкните в пределах элемента Test Subject 1. Так как установленный нами обработчик события `click` приводит к переключению класса `matched` (в данном случае класс добавляется) в элементе, цвет элемента изменится (рис. 9.8).

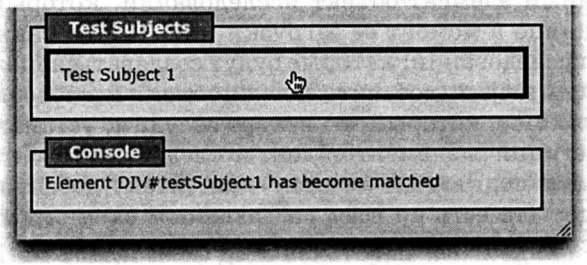


***Рис. 9.8.** Добавление класса matched к объекту исследований приводит к вызову обработчика события перехода в состояние соответствия и изменению его внешнего вида*

Кроме того, вследствие добавления класса `matched` элемент перейдет в состояние соответствия селектору, с помощью которого мы устанавливали обработчики событий перехода/выхода из состояния соответствия, и как результат запуска обработчика события перехода в состояние соответствия в панели Console появится сообщение.

- *Упражнение 2.* Щелкните на элементе Test Subject 1 еще раз. Это приведет к удалению класса `matched` из элемента. Элемент вернется к своему прежнему виду и выполнится обработчик события выхода из состояния соответствия, так как элемент перестанет соответствовать селектору. В результате этого вызова в панели Console появится соответствующее сообщение.
- *Упражнение 3.* Щелкните на кнопке Add New Test Subject (добавить новый объект исследований). В страницу будет добавлен новый элемент `<div>`. Так как элемент создается с классом `testSubject`, он будет соответствовать селектору, предназначенному для упреждающей установки обработчика события `click`, обработчик которого в свою очередь переключает класс `matched`. В результате Live Query *автоматически* подключит обработчик события `click` к этому новому элементу (программный код, добавивший элемент, не подключает к нему никаких обработчиков событий). Чтобы проверить это утверждение, щелкните на недавно созданном элементе Test Subject 2. Вы увидите, что его внешний вид изменится и будет вызван обработчик перехода в состояние соответствия, в подтверждение того, что для переключения класса `matched` к элементу автоматически был подключен обработчик события `click`. Доказательства приведены на рис. 9.9.
- *Упражнение 4.* Поэкспериментируйте с кнопкой Add New Test Subject (добавить новый объект исследований) и с объектами исследований,

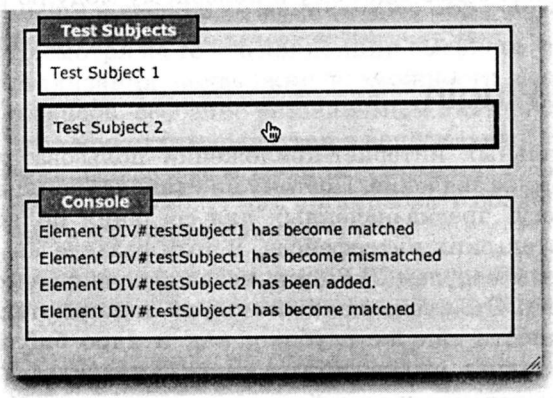


***Рис. 9.9.** Тот факт, что вновь созданный элемент Test Subject 2 реагирует на щелчки мыши так же, как элемент Test Subject 1, доказывает, что Live Query автоматически подключил к вновь созданному элементу обработчик события click, а также обработчики событий перехода в состояние/выхода из состояния соответствия*

пока не убедитесь, что обработчики событий щелчка мыши и перехода в состояние/выхода из состояния соответствия всегда автоматически добавляются к элементам-объектам исследований.

- *Упражнение 5.* Поэкспериментируйте с удалением обработчиков Live Query. Перезагрузите страницу, чтобы восстановить первоначальные настройки, и добавьте в нее один или два объекта исследований с помощью кнопки Add New Test Subject. Затем щелкните на кнопке Expire Match Handlers (удалить обработчики событий изменения состояния соответствия) – она удалит обработчики событий перехода в состояние/выхода из состояния соответствия, которые были подключены к объектам исследований. Обратите внимание: после этого щелчки на объектах исследований не приводят к появлению сообщений в панели Console, доказывая тем самым, что обработчики удалены. Событие `click` по-прежнему приводит к переключению класса `matched`, в чем можно убедиться по изменению внешнего вида элементов.
- *Упражнение 6.* Щелкните на кнопке Expire Event Handlers. В результате из объектов исследований будет удален обработчик события `click` Live Query. Обратите внимание: после этого объекты исследований перестанут реагировать на щелчки, оставаясь в том же состоянии, в каком они находились до щелчка на кнопке.

И не обладая богатым воображением, можно представить, какую мощь придает этот модуль расширения страницам полнофункциональных интернет-приложений, где элементы постоянно изменяются, появляются и исчезают. Позволяя выполнять упреждающую установку обработчиков событий, модуль Live Query помогает минимизировать объем программного кода, который потребовалось бы написать для добавления и изменения элементов страницы.

А теперь перейдем к еще более важному и полезному модулю расширения.

9.4. Введение в UI Plugin

Для полнофункциональных интернет-приложений пользовательский интерфейс имеет огромное значение. Поэтому не удивляет обилие модулей расширения jQuery, предназначенных для создания полнофункциональных пользовательских интерфейсов. В этом разделе мы познакомимся с официальным модулем UI Plugin, недавно появившемся в семействе модулей jQuery. Это очень важный модуль, и нам хотелось бы охватить его возможности так же глубоко, как и ядро библиотеки jQuery, но реальность такова, что для этого в книге не хватило бы места.

Мы достаточно подробно рассмотрим только два основных метода этого модуля расширения, реализующих технологию перетаскивания объектов страницы мышью, чтобы вы могли представить работу составляющих этого модуля. Затем мы опишем оставшуюся часть модуля, показав возможности, которые он может привнести в наши полнофункцио-

нальные интернет-приложения. Дополнительную информацию об этих возможностях вы найдете на странице *http://docs.jquery.com/ui*.

Модуль UI Plugin обеспечивает дополнительную функциональность в трех основных направлениях: взаимодействие с мышью, графические компоненты и визуальные эффекты. Операция перетаскивания объектов мышью относится к категории взаимодействия с мышью, с которой мы и начнем наше знакомство.

9.4.1. Взаимодействия с мышью

Взаимодействие с указателем мыши – неотъемлемая и главная часть любого графического интерфейса. Несмотря на то что в веб-интерфейсы уже встроено множество простых взаимодействий с указателем мыши (например, щелчок), у веб-интерфейса нет встроенной поддержки некоторых улучшенных способов взаимодействия, доступных обычным настольным приложениям. Ярким примером может служить отсутствие поддержки операции перетаскивания объектов мышью (drag-and-drop).

Операция перетаскивания широко используется в пользовательских интерфейсах настольных приложений. Например, в любой настольной системе есть менеджер файлов с графическим интерфейсом, позволяющим нам легко копировать и перемещать файлы, перетаскивая их мышью из папки в папку, а также удалять, перетаскивая их на пиктограмму Trash или Wasterbasket (Корзина). Но насколько этот тип взаимодействия распространен в настольных приложениях, настолько же редко он встречается в веб-приложениях, главным образом потому, что современные броузеры изначально не поддерживают возможность перетаскивания. Его правильная реализация – задача не из легких.

«Не из легких? – усмехнетесь вы. – Пара перехваченных событий от мыши и несколько махинаций с CSS – неужели это так сложно?»

Несмотря на то что концепции высокого уровня легки для понимания, реализация поддержки возможности перетаскивания с учетом всех нюансов, особенно независимым от типа броузера способом, очень быстро может превратиться в непосильный труд. Но как раньше библиотека jQuery и ее модули расширения облегчали нашу работу, так и теперь они помогут нам организовать поддержку операции перетаскивания в веб-приложениях.

Но прежде чем мы сможем перетаскивать объекты, *оставляя* их на новом месте, следует научиться просто перетаскивать их.

Перетаскивание объектов

В большинстве словарей нет слова *перетаскиваемый* (*draggable*). Этим термином обычно обозначают элементы, которые можно перемещать путем перетаскивания. Так и в модуле UI Plugin этот термин обозначает подобные элементы и служит именем метода, придающего эту способность элементам в обернутом наборе.

Синтаксис функции draggable

`draggable(options)`

Делает элементы в обернутом наборе перетаскиваемыми в соответствии с указанными параметрами.

Параметры

`options` (объект) Объект-хеш параметров, применяемых к перетаскиваемым элементам, как описано в табл. 9.2. Если параметры не указаны, элемент можно просто перетаскивать по всей поверхности страницы в пределах окна.

Возвращаемое значение

Обернутый набор.

Для применения команды `draggable()` необходимо подключить к странице как минимум два файла (помимо файла самой библиотеки jQuery):

```
ui.mouse.js
ui.draggable.js
```

Чтобы получить доступ к дополнительным возможностям, мы также подключим файл:

```
ui.draggable.ext.js
```

Параметры, как базовые, так и расширенные, поддерживаемые этим методом, перечислены в табл. 9.2.

Таблица 9.2. Базовые и расширенные параметры команды draggable( )

Имя	Описание
Базовые параметры	
`Helper`	(строка | функция) Точно определяет, что нужно перетаскивать. При заданном значении *original* (по умолчанию) перетаскивается исходный элемент. При заданном значении *clone* для перетаскивания создается копия элемента. В этом параметре можно также определить функцию, которая принимает в качестве параметра исходный элемент DOM и возвращает элемент для перетаскивания. Чаще всего это копия исходного элемента, видоизмененная некоторым способом, например: `function(e){return $(e).clone().css('color','green')}`.
`Ghosting`	(логическое значение) Значение `true` является синонимом `helper:'clone'`.
`Handle`	(обернутый набор | элемент) Альтернативный элемент или обернутый набор jQuery, содержащий элемент, который играет роль ручки для перетаскивания, – элемент, на котором выполняется щелчок, инициирующий операцию перетаскивания. Этот элемент обычно является дочерним по отношению к перетаскиваемому элементу, но им может быть и любой другой элемент страницы.

Имя	Описание
preventionDistance	(число) Число пикселов, на которое должен сместиться указатель мыши после щелчка, чтобы началось перетаскивание. Этот параметр позволяет предотвратить случайное перемещение элементов. Если опущен, то по умолчанию используется значение 0.
dragPrevention	(массив) Массив селекторов дочерних элементов, которые не должны инициировать операцию перетаскивания в результате щелчка на них. Значение по умолчанию: ['input','textarea','button','select','option']. Этот параметр позволяет предотвратить запуск перетаскивания, например, по щелчку на встроенном элементе управления.
cursorAt	(объект) Определяет пространственные отношения между указателем мыши и перемещаемым объектом во время перетаскивания. Указанный объект может определять свойства top, left, bottom и right. Например, объект {top:5,left:5} определяет, что указатель мыши должен находиться в пяти пикселах от левого верхнего угла перетаскиваемого элемента. Если опущен, то используется первоначальная позиция указателя мыши относительно начала координат элемента, в которой произошел щелчок, инициировавший перетаскивание.
appendTo	(строка | элемент) Указывает элемент, к которому по окончании перетаскивания должен быть добавлен элемент helper. Если указана строка *parent* (по умолчанию), то элемент helper остается на своем исходном месте в иерархии.
start	(функция) Функция обратного вызова, вызываемая в начале перетаскивания. В качестве контекста функции ей передается перетаскиваемый элемент. Она принимает два параметра: экземпляр объекта event, свойство target которого ссылается на перетаскиваемый элемент, и объект, содержащий следующие свойства: • helper – текущий элемент helper; • position – объект со свойствами top и left, определяющими координаты начала перетаскивания; • offset – объект, определяемый параметром cursorAt; • draggable – внутренний перетаскиваемый объект JavaScript (не очень полезен); • options – хеш параметров, использованных при создании перемещаемого объекта.
stop	(функция) Функция обратного вызова, вызываемая по окончании перетаскивания. Ей передаются те же самые два параметра, что и функции start. В качестве контекста функции ей передается перетаскиваемый элемент. Свойство position во втором параметре определяет координаты левого верхнего угла перетаскиваемого элемента.

Таблица 9.2 (продолжение)

Имя	Описание
drag	(функция) Функция обратного вызова, постоянно вызываемая в процессе перетаскивания. Ей передаются те же два параметра, что и функции start. В качестве контекста функции ей передается перетаскиваемый элемент. Свойство position во втором параметре определяет координаты левого верхнего угла перетаскиваемого элемента.
Расширенные параметры	
axis	(строка) Ограничивает ось координат, вдоль которой можно перетаскивать объект: *x* – по горизонтали, *y* – по вертикали. Если опущен, никакое ограничение не накладывается.
containment	(строка | объект | элемент) Определяет границы, в которых допускается перетаскивать объект. Если опущен, никакие ограничения не накладываются. Этот параметр может иметь следующие значения: • *parent* – удерживает перетаскиваемый элемент в пределах его родительского элемента, не допуская появления полос прокрутки окна; • *document* – удерживает перетаскиваемый элемент в пределах документа, не допуская появления полос прокрутки родительского элемента; • *селектор* – идентифицирует содержимое элемента; • *объект* – определяет границы прямоугольной области относительно родительского элемента с помощью свойств left, right, top и bottom.
effect	(массив) Массив из двух строк, применяющих эффект растворения к копиям перетаскиваемых объектов. Может иметь значения ['fade','fade'], ['fade',''] или ['','fade']. На первый взгляд, это не совсем правильный путь применения эффекта, но он позволяет реализовать поддержку дополнительных эффектов в дальнейшем. В настоящее время поддерживается только эффект растворения.
grid	(массив) Массив из двух чисел, определяющих прямоугольную сетку дискретных позиций, в которые можно переместить перетаскиваемый объект, например [100,100]. Начало координат сетки определяется относительно исходной позиции перетаскиваемого объекта. Если опущен, никакие ограничения не накладываются.
opacity	(число) Указывает уровень непрозрачности перетаскиваемого объекта во время перетаскивания в виде значения в диапазоне от 0.0 до 1.0 (включительно). Если опущен, непрозрачность перетаскиваемого объекта не изменяется.
revert	(логическое значение) Если содержит значение true, то по окончании перетаскивания объект возвращается в исходную позицию. Если опущен или содержит значение false, объект остается в новой позиции.

Подумав о том, что мы предложим вам поэкспериментировать с лабораторной страницей, демонстрирующей применение этих параметров, вы не ошиблись! Но прежде чем перейти к экспериментам, давайте рассмотрим еще три метода, связанные с перетаскиванием элементов.

Сделать перетаскиваемый элемент снова *неперетаскиваемым* позволяет команда `draggableDestroy()`, которая лишает элементы способности к перетаскиванию.

Синтаксис функции draggableDestroy

`draggableDestroy()`

Лишает элементы в обернутом наборе способности к перетаскиванию.

Параметры

Нет

Возвращаемое значение

Обернутый набор.

Временно запретить элементу быть перетаскиваемым, с тем чтобы позже восстановить эту его способность, позволяет команда `draggableDisable()`. Отменяет этот запрет команда `draggableEnable()`.

Синтаксис функции draggableDisable

`draggableDisable()`

Приостанавливает у перетаскиваемых элементов в обернутом наборе способность к перетаскиванию, не удаляя информацию и параметры перетаскивания.

Параметры

Нет

Возвращаемое значение

Обернутый набор.

Синтаксис функции draggableEnable

`draggableEnable()`

Восстанавливает у перетаскиваемых элементов в обернутом наборе способность к перетаскиванию, заблокированную командой `draggableEnable()`.

Параметры

Нет

Возвращаемое значение

Обернутый набор.

Давайте исследуем параметры операции перетаскивания с помощью лабораторной страницы UI Draggables Lab (лабораторная страница перетаскиваемых элементов пользовательского интерфейса). Откройте в браузере файл *chapter9/ui/lab.draggables.html*, чтобы увидеть страницу, показанную на рис. 9.10.

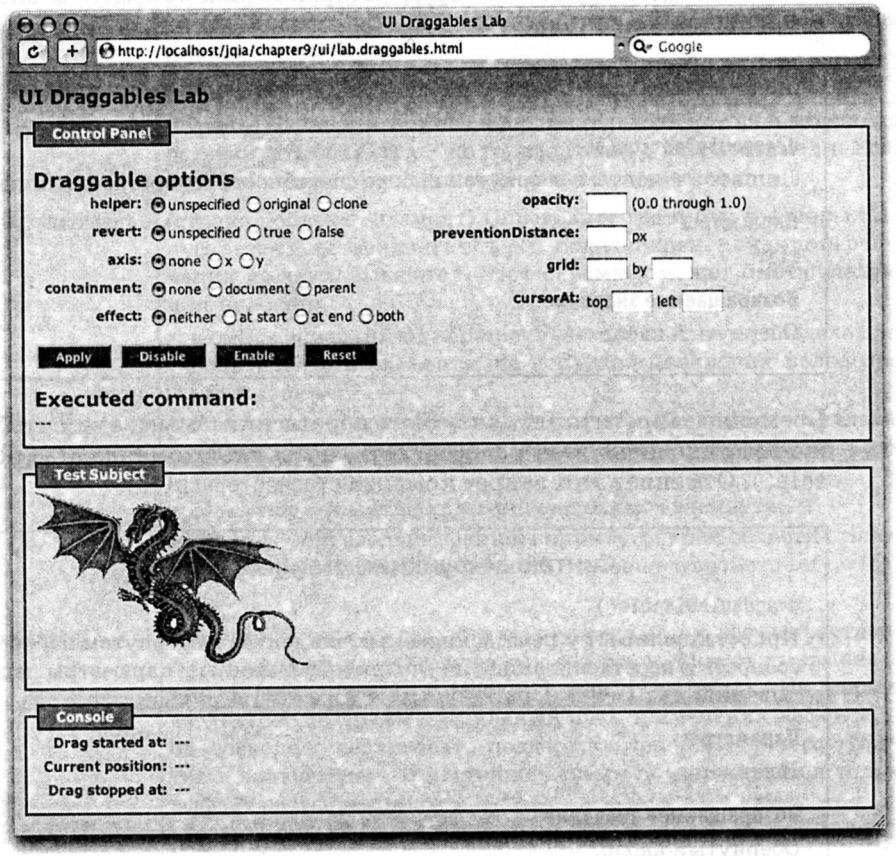


***Рис. 9.10.** Лабораторная страница UI Draggables Lab позволяет экспериментировать с большинством параметров настройки перетаскиваемых объектов*

Структура этой лабораторной страницы вам уже знакома: панель Control Panel (панель управления) позволяет указывать параметры для метода `draggable()`, панель Test Subject (объект исследований) содержит элемент-изображение, играющий роль объекта исследований, а в панель Console (консоль) будут выводиться сообщения о ходе выполнения операции перетаскивания.

По щелчку на кнопке Apply (применить) в панели Control Panel заданные параметры собираются в объект и вызывается команда `draggable()`.

Формат команды отображается ниже кнопки Apply. (Для большей наглядности отображаются только параметры, выбранные в панели Control Panel. Функции обратного вызова, добавляемые в параметры для вывода сообщений в панель Console, не отображаются, хотя и включаются в вызов команды.) Действие кнопок Disable (запретить) и Enable (разрешить) мы понаблюдаем в упражнении 3. Кнопка Reset (сбросить настройки) возвращает параметры в исходное состояние и лишает объект исследований способности к перетаскиванию.

Исследования мы начнем с нескольких упражнений!

Упражнение 1. Для первого упражнения мы создадим простой перетаскиваемый объект со значениями всех параметров по умолчанию. Откройте лабораторную страницу UI Draggables Lab в броузере и оставьте все параметры в исходном состоянии – изначально никакие значения не определены.

Нажмите левую кнопку мыши на объекте исследований (изображение дракона) и, не отпуская ее, переместите рисунок. Ничего особенного не произойдет – если, конечно, у вас не операционная система Mac OS X, в которой вы *можете* заметить кое-что интересное. В Mac OS X можно перетаскивать изображения из веб-страниц в локальную систему.[1] Не путайте эту возможность, поддерживаемую системой, с той, поддержку которой мы собираемся обеспечивать в своих страницах.

Теперь щелкните на кнопке Apply (применить) и обратите внимание на команду, которая будет выполнена:

```
$('#testSubject').draggable({});
```

Попробуйте переместить изображение дракона еще раз. Вы можете заставить дракона летать по всему окну броузера. (Правда, крыльями он не машет, но и это можно реализовать – с помощью анимации!) Кроме того, обратите внимание: если переместить изображение за границу окна, например за правую, то появляется горизонтальная полоса прокрутки, которой здесь раньше не было (в большинстве броузеров).

Отпустите дракона и подберите его опять. Перемещайте его по всему окну. Посмотрите, как изменяются значения в панели Console в процессе перетаскивания. Обновлением значений занимаются функции обратного вызова, указанные в параметрах `start`, `stop` и `drag`.

Упражнение 2. Перезагрузите страницу, чтобы вернуть ее в исходное состояние. У броузеров Firefox или Camino есть одна раздражающая особенность: при щелчке на кнопке Reload (обновить) в панели инструментов броузера элементы формы не возвращаются к своим *начальным* значениям. Чтобы перезагрузить страницу в исходное состояние, перенесите фокус ввода в поле ввода URL-адреса и нажмите клавишу Enter.

---

[1] Такая возможность есть и в ОС Windows, и в Linux (по крайней мере, в окружении рабочего стола KDE). – *Прим. перев.*

Теперь установите параметр helper в значение Original (исходный элемент), щелкните на кнопке Apply (применить) и попробуйте переместить объект исследований в пределах окна. Все должно выглядеть точно так же, как в упражнении 1, потому что когда параметр helper опущен, по умолчанию используется значение Original. Установите значение Clone (копия) и щелкните на кнопке Apply.

Теперь, попытавшись переместить изображение, вы увидите, что перемещается не исходный элемент, а его *копия*. Как только вы закончите перетаскивание, эта копия исчезнет.

Упражнение 3. Оставив параметры в том состоянии, в каком они находились к концу упражнения 2, щелкните на кнопке Disable (запретить) и обратите внимание на команду, которая была выполнена:

```
$('#testSubject').draggableDisable();
```

Попробуйте переместить дракона. Ничего не происходит. Теперь щелкните на кнопке Enable (разрешить), которая выполнит команду:

```
$('#testSubject').draggableEnable();
```

Обратите внимание: теперь вы снова можете перетаскивать дракона *и* все прежние значения параметров (в данном случае, значение clone в параметре helper) остаются в силе, подтверждая различие между командой draggableDestroy(), отнимающей способность к перетаскиванию, и командой draggableDisable(), которая временно блокирует эту способность до вызова команды draggableEnable().

Упражнение 4. Приведите лабораторную страницу в исходное состояние, выберите значение True для параметра revert и щелкните на кнопке Apply. Попробуйте перемещать объект исследований, наблюдая за тем, как по завершении операции он возвращается в исходную позицию. Теперь установите значение Clone для параметра helper, щелкните на кнопке Apply и повторите упражнение. Обратите внимание: теперь параметр revert применяется к копии.

Упражнение 5. Приведите лабораторную страницу в исходное состояние и поэкспериментируйте с параметром axis, позволяющим ограничить перемещение объекта по горизонтали или по вертикали.

Упражнение 6. В этом упражнении мы займемся параметром containment. Увеличьте высоту окна броузера, насколько это возможно. Надеемся, разрешение вашего экрана таково, что позволит открыть достаточное пространство ниже панели Console.

До сих пор мы оставляли параметр containment не заданным. Помните, что дракон мог летать по всему окну броузера? Теперь выберите значение Document (документ) и щелкните на кнопке Apply. Перетаскивая изображение, обратите внимание на два момента:

- Теперь нельзя переместить изображение за границы окна, поэтому полосы прокрутки уже не появляются.

- Невозможно переместить изображение ниже панели Console, где заканчивается область действия объекта *document*, но не окна.

Теперь выберите значение Parent и щелкните на кнопке Apply. Начав перетаскивание, обратите внимание на то, что изображение можно перетаскивать только внутри панели Test Subject (объект исследований – элемент `<fieldset>`), для которой объект исследований является дочерним элементом. Заметьте и то, что так происходит, даже если начать перетаскивание за пределами элемента-родителя (где объект исследований оказался в результате предыдущего перетаскивания, не ограниченного рамками элемента-родителя).

Упражнение 7. Выберите значение Clone для параметра `helper` и исследуйте эффект, воспроизводимый параметром `effect` с различными значениями.

Упражнение 8. Приведите лабораторную страницу в исходное состояние и установите для параметра `opacity` значение 0,5. Посмотрите, как изменяется непрозрачность элемента при перетаскивании исходного объекта и его копии.

Упражнение 9. Приведите лабораторную страницу в исходное состояние и установите для параметра `preventionDistance` достаточно большое значение, например 200. После щелчка на кнопке Apply начните перетаскивать дракона за край левого крыла, перемещая указатель мыши вправо. Прежде чем начнется собственно перетаскивание, вам придется пересечь дракона по ширине (250 пикселов) почти полностью. Вы сами вряд ли когда-нибудь установите такое большое значение этого параметра – мы сделали это лишь для наглядности влияния параметра. В большинстве случаев этому параметру задают гораздо меньшее значение, чтобы предотвратить инициализацию непреднамеренного перемещения при случайном движении на несколько пикселов.

Упражнение 10. Приведите лабораторную страницу в исходное состояние и установите для параметра `grid` значения 100 и 100. После щелчка на кнопке Apply вы заметите, что теперь изображение перемещается только с шагом в 100 пикселов в любом направлении. Поэкспериментируйте с разными значениями, чтобы исследовать влияние этого параметра на перемещение.

Упражнение 11. Приведите лабораторную страницу в исходное состояние и установите для параметра `cursorAt` значения 10 и 10. Щелкните на кнопке Apply. С началом перетаскивания указатель мыши займет положение в 10 пикселах от левого верхнего угла (возле кончика левого крыла) независимо от того, в какой точке изображения находился указатель мыши перед началом перетаскивания.

Упражнение 12. Экспериментируйте с параметрами настройки, как по отдельности, так и совместно, пока не убедитесь в том, что правильно понимаете, как каждый из них влияет на перетаскивание элемента.

С перетаскиванием объектов по экрану все в порядке, но какой в этом прок? Немного забавно, но быстро приедается, как йо-йо (если, конечно, вы не фанат этой игрушки). На практике мы могли бы организовать для пользователей возможность перемещать по экрану модульные элементы (заботливо фиксируя их положение с помощью cookies или других механизмов сохранения информации), например, в такой игре, как пазл.

Перетаскивать что-то по-настоящему интересно, только если это что-то можно где-то отпустить. Давайте посмотрим, как создаются элементы – *пункты отпускания*, применяемые совместно с перетаскиваемыми элементами.

Отпускание перетаскиваемого элемента

Перетаскиваемым элементам необходимы *элементы отпускания* (*droppables*), способные принять перетаскиваемый элемент, выполнив при этом событии некоторые действия. Создание элементов отпускания похоже на создание перетаскиваемых элементов, фактически даже проще, потому что в этом случае значительно меньше параметров, о которых нужно заботиться.

Как и в случае с перетаскиваемыми элементами, реализация элементов отпускания разбита на два файла сценариев: в основном файле определяется команда `droppable()` с базовыми параметрами, а дополнительный файл содержит расширенные параметры. Вот эти файлы:

```
ui.droppable.js
ui.droppable.ext.js
```

Команда `droppable()` имеет следующий синтаксис:

Синтаксис функции droppable

```
droppable(options)
```

Делает элементы в обернутом наборе элементами отпускания, то есть элементами, на которых можно отпустить перетаскиваемый элемент.

Параметры

`options` (объект) Параметры, применяемые к элементам отпускания. Подробная информация о параметрах приведена в табл. 9.3.

Возвращаемое значение

Обернутый набор.

Став элементом отпускания, элемент может быть в одном из трех состояний – не активен, активен или готов.

Неактивное состояние – обычное состояние элементов отпускания, в котором они пребывают большую часть времени, ожидая начала операции перетаскивания. С началом перетаскивания элемент отпуска-

ния выясняет, насколько пригоден для него перетаскиваемый элемент (понятие *пригодности* мы рассмотрим чуть ниже), и если пригоден (и *только* если пригоден), он переходит в *активное состояние*. В активном состоянии элемент отпускания следит за перетаскиванием, ожидая либо завершения перетаскивания (в этом случае элемент отпускания возвращается в неактивное состояние), либо момента, когда пригодный перетаскиваемый элемент окажется над элементом отпускания, – в этом случае он переходит в *состояние готовности*.

Если перетаскивание завершилось, когда элемент отпускания был в состоянии готовности, перетаскиваемый элемент считается *отпущенным* на элемент отпускания. Если перетаскиваемый элемент, продолжая перемещаться, выйдет за пределы элемента отпускания, тот вернется в активное состояние.

Ого, как меняются эти состояния – не уследишь! Диаграмма на рис. 9.11 поможет вам наглядно представить их.

Как и в случае с перетаскиваемыми элементами, базовые параметры настройки определены в основном файле сценария, а набор расширенных параметров настроек – в дополнительном. Оба набора параметров описаны в табл. 9.3.

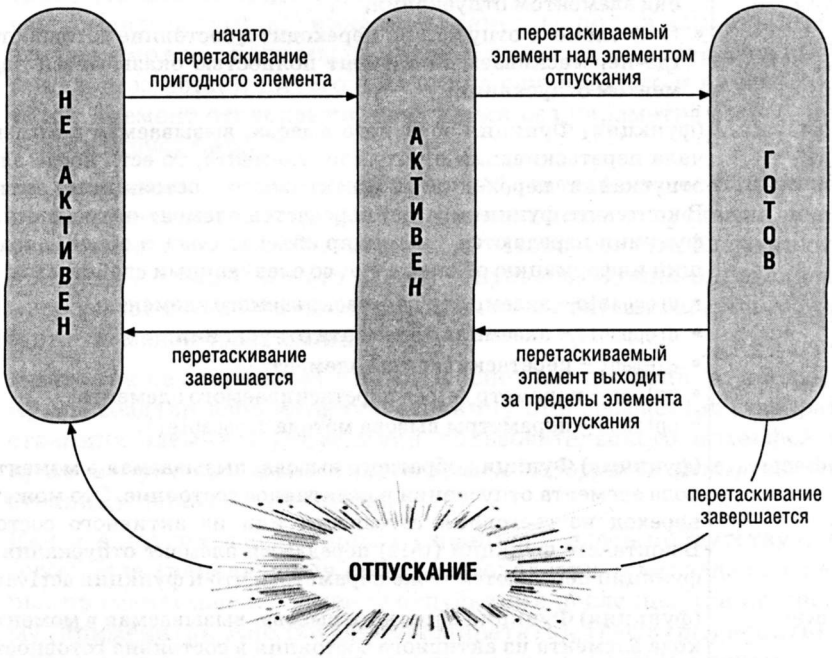


***Рис. 9.11.** По мере перемещения перетаскиваемого элемента изменяется состояние элемента отпускания*

Таблица 9.3. Базовые и расширенные параметры команды droppable( )

Имя	Описание
Базовые параметры	
Accept	(строка | функция) Определяет перетаскиваемые элементы, пригодные для элемента отпускания. Это может быть строка, содержащая селектор jQuery, или функция, которая возвращает true для пригодных перетаскиваемых элементов. Если указана функция, в виде единственного параметра ей будет передан рассматриваемый элемент-кандидат.
tolerance	(строка) Строковое значение, которое определяет, как перетаскиваемый элемент должен быть позиционирован относительно элемента отпускания, чтобы привести последний в состояние готовности. Допустимые значения: • touch – элемент отпускания переходит в состояние готовности, когда перетаскиваемый элемент прикасается к элементу отпускания или перекрывает какую-либо его часть; • pointer – элемент отпускания переходит в состояние готовности, когда в ходе перетаскивания указатель мыши оказывается над ним; • intersect – элемент отпускания переходит в состояние готовности, когда перетаскиваемый элемент наполовину оказывается над элементом отпускания; • fit – элемент отпускания переходит в состояние готовности, когда перетаскиваемый элемент полностью оказывается над элементом отпускания.
activate	(функция) Функция обратного вызова, вызываемая в момент начала перетаскивания пригодного элемента, то есть когда элемент отпускания переходит из неактивного состояния в активное. В контексте функции (this) передается элемент отпускания. Этой функции передаются экземпляр объекта event и объект, содержащий информацию об операции, со следующими свойствами: • draggable – экземпляр перетаскиваемого элемента; • droppable – экземпляр элемента отпускания; • element – перетаскиваемый элемент; • helper – параметр helper перетаскиваемого элемента; • options – параметры вызова метода droppable().
deactivate	(функция) Функция обратного вызова, вызываемая в момент перехода элемента отпускания в неактивное состояние. Это может быть переход из состояния готовности или из активного состояния. В контексте функции (this) передается элемент отпускания. Этой функции передаются те же параметры, что и функции activate.
over	(функция) Функция обратного вызова, вызываемая в момент перехода элемента из активного состояния в состояние готовности как результат соответствия критериям, заданным параметром tolerance. В контексте функции (this) передается элемент отпускания. Этой функции передаются те же параметры, что и функции activate.

Имя	Описание
out	(функция) Функция обратного вызова, вызываемая в момент перехода элемента отпускания из состояния готовности в активное состояние, когда перетаскиваемый элемент покидает пределы элемента отпускания согласно критериям, заданным в параметре tolerance. В контексте функции (this) передается элемент отпускания. Этой функции передаются те же параметры, что и функции activate.
drop	(функция) Функция обратного вызова, вызываемая в момент отпускания перетаскиваемого элемента над элементом отпускания, когда последний находился в состоянии готовности. В контексте функции (this) передается элемент отпускания. Этой функции передаются те же параметры, что и функции activate.
Расширенные параметры	
activeClass	(строка) Имя класса CSS, применяемого к элементу отпускания, когда тот находится в активном состоянии.
hoverClass	(строка) Имя класса CSS, применяемого к элементу отпускания, когда пригодный перетаскиваемый элемент располагается над ним, то есть когда элемент отпускания находится состоянии готовности.

Рассматривая порядок создания перетаскиваемых элементов, мы заметили, что можно создать перетаскиваемый элемент, вообще не передавая команде draggable() никакие параметры, но для команды droppable() это невозможно. Если мы не укажем никаких параметров в вызове команды droppable(), ничего *плохого* не случится, но и ничего хорошего тоже. Элемент отпускания, созданный без параметра accept, прекрасно имитирует кирпич.

По умолчанию элемент отпускания не считает пригодным *ни один* перетаскиваемый элемент. А если над элементом отпускания нельзя ничего отпустить – какой в нем толк? Чтобы создать элемент отпускания, над которым можно что-то отпустить, нужно определить параметр accept с указанием перетаскиваемых элементов, *пригодных* для этого элемента отпускания.

Ничто так не закрепляет в памяти основные понятия, как игра, поэтому мы создали лабораторную страницу UI Droppables Lab (лабораторная страница элемнтов отпускания пользовательского интерфейса). Открыв в броузере файл *chapter9/ui/lab.droppables.html*, вы увидите страницу, показанную на рис. 9.12.

Как и в других лабораторных страницах, здесь присутствует панель Control Panel (панель управления), которая позволяет задавать параметры, применяемые к элементу отпускания после щелчка на кнопке Apply (применить). Кнопки Disable (запретить), Enable (разрешить) и Reset (сбросить в исходное состояние) по своему поведению похожи на одноименные кнопки лабораторной страницы UI Draggables Lab.

В панели Test Subjects (объекты исследований) имеется шесть перетаскиваемых объектов и элемент Drop Zone (зона отпускания), который после

Рис. 9.12. *Лабораторная страница UI Droppables Lab позволяет исследовать воздействие различных параметров на перетаскивание и отпускание*

щелчка на кнопке Apply (применить) превращается в элемент отпускания. Под элементом Drop Zone расположены текстовые элементы серого цвета с надписями ACTIVATE (активизация), OVER (над), OUT (выход за пределы), DROP (отпускание) и DEACTIVATE (деактивизация). При вызове той или иной функции обратного вызова (эти функции добавляются к элементу отпускания внутри реализации лабораторной страницы) мгновенно подсвечивается соответствующий ей текстовый элемент (эти элементы мы называем *индикаторами вызова*), показывая, что была запущена функция обратного вызова.

Эта лабораторная страница поможет нам глубже изучить принцип действия элемента отпускания. Вперед!

Упражнение 1. В этом упражнении мы познакомимся с параметром `accept`, который позволяет элементу отпускания определить *пригодность* перетаскиваемого элемента. В этом параметре можно указать любой селектор jQuery (и даже функцию, определяющую пригодность программно), однако мы сосредоточимся на элементах, обладающих

определенными именами классов. В частности, можно определить селектор, отбирающий элементы с любым из имен классов – flower (цветок), dog (собака), motorcycle (мотоцикл) и water (вода), пометив соответствующий флажок для параметра accept.

В левой части панели Test Subjects (объекты исследований) расположены шесть элементов-изображений, каждому из которых присвоено одно или два имени класса в соответствии с рисунком. Например, левый верхний перетаскиваемый элемент определен с именами классов CSS dog и flower (потому что на нем есть собака и цветок), а нижний средний элемент – с именами классов motorcycle и water (мотоцикл Yamaha V-Star и река Колорадо).

Прежде чем щелкнуть на кнопке Apply (применить), попробуйте перетащить любой из этих элементов на элемент Drop Zone (зона отпускания). Кроме перетаскивания, ничего не произойдет. Обратите внимание на индикаторы вызова – они не изменяются. Это совершенно не удивительно, потому что изначально на странице нет элемента отпускания.

Теперь, оставив все элементы управления в исходном состоянии (в том числе все отмеченные флажки параметра accept), щелкните на кнопке Apply (применить). Выполняемой в этот момент команде будет передан параметр accept, определяющий селектор, которому соответствуют имена всех четырех классов.

Попробуйте еще раз перетащить любое из изображений на элемент Drop Zone, наблюдая при этом за индикаторами вызова. На этот раз вы увидите, как с началом перемещения любого изображения индикатор ACTIVATE мигнет, свидетельствуя о том, что элемент отпускания заметил начало перетаскивания, и перетаскиваемый элемент пригоден для отпускания.

Проведите изображение над элементом Drop Zone несколько раз. Индикаторы вызова в соответствующие моменты времени покажут, что были вызваны функции over и out. Теперь отпустите изображение за пределами зоны отпускания и посмотрите, как будет подсвечен индикатор DEACTIVATE.

Наконец, повторите перетаскивание, но на этот раз отпустите изображение над зоной отпускания. Будет подсвечен индикатор DROP (свидетельствуя о вызове функции drop). Обратите внимание: в зоне отпускания осталось изображение, отпущенное над ней.

Упражнение 2. Снимите все флажки в параметре accept и щелкните на кнопке Apply. Не важно, какое изображение вы попробуете перетащить, – ни один из индикаторов не будет подсвечен и ничего не произойдет при попытке отпустить изображение в зоне отпускания. Без параметра accept наш элемент отпускания превратился в кирпич.

Упражнение 3. Установите в параметре accept хотя бы один флажок, например flower (цветок), и щелкните на кнопке Apply. Обратите внимание: пригодными элементами теперь считаются только изображения

с цветами (страница тоже их узнает, потому что для таких изображений определено имя класса flower).

Упражнение 4. Верните элементы управления в исходное состояние, для параметра activeClass выберите переключатель greenBorder и щелкните на кнопке Apply. В результате элемент отпускания будет создан с параметром activeClass, то есть с именем класса, который (как вы уже догадались) определяет зеленую рамку.

Теперь с началом перетаскивания изображения, пригодного с точки зрения элемента отпускания (в соответствии со значением параметра accept), черная рамка вокруг элемента Drop Zone превратится в зеленую. Если у вас возникнут трудности с реализацией подобного поведения в собственных страницах, вспомните правила определения приоритетов CSS. Класс, указанный в параметре activeClass, должен отменять правило, определяющее визуальное представление по умолчанию, которое вы хотите переопределить. Это справедливо и для параметра hoverClass.

Упражнение 5. Выберите для параметра hoverClass переключатель redBorder и щелкните на кнопке Apply. Теперь, когда пригодное изображение будет перемещаться над зоной отпускания, рамка сменяет свой цвет с зеленого (результат действия класса из параметра activeClass) на красный.

Поэкспериментируйте с этими двумя параметрами, пока не почувствуете, что достаточно ясно представляете, какие изменения вызывает каждый из параметров.

Упражнение 6. Чтобы выполнить это упражнение, попробуйте выбрать по очереди все переключатели параметра tolerance и посмотрите, как они влияют на переход элемента отпускания из активного состояния в состояние готовности (в соответствии с описанием этого параметра в табл. 9.3). Этот переход легко заметить, когда установлен параметр hoverClass или по подсветке индикатора вызова OVER.

Подобно перетаскиваемому элементу, элемент отпускания можно лишить способности быть элементом отпускания, временно запретить и затем вновь разрешить. Для этого предназначены следующие методы:

Синтаксис функции droppableDestroy

```
droppableDestroy()
```

Лишает элементы в обернутом наборе способности играть роль элементов отпускания.

Параметры

Нет

Возвращаемое значение

Обернутый набор.

Синтаксис функции droppableDisable

`droppableDisable()`

Временно блокирует у элементов отпускания в обернутом наборе способность играть роль элементов отпускания, не очищая информацию и параметры такой способности.

Параметры

Нет

Возвращаемое значение

Обернутый набор.

Синтаксис функции droppableEnable

`droppableEnable()`

Восстанавливает у элементов отпускания в обернутом наборе способность играть роль элементов отпускания, заблокированную командой `droppableEnable()`.

Параметры

Нет

Возвращаемое значение

Обернутый набор.

Перетаскивание элементов – полезный прием организации взаимодействий в самых разных случаях. Она часто применяется для того, чтобы показать связь, но может также использоваться для переупорядочения элементов. Этот последний случай встречается настолько часто, что получил непосредственную поддержку в модуле UI Plugin. Давайте рассмотрим его подробнее.

Прочие методы взаимодействий с мышью

Последняя группа команд в категории методов взаимодействия с мышью модуля UI Plugin – это методы *переупорядочения*, *выделения* и *изменения размеров*. Методы в этих трех категориях дополняют возможность перетаскивания элементов, позволяя переупорядочивать их в пределах контейнера и изменять их размеры, соответственно.

Как и в предыдущих командах модуля UI Plugin, каждый из этих методов применяется к обернутому набору и получает в качестве параметра объект-хеш, определяющий значения настроек. Более полную информацию по этим методам вы сможете получить на сайте *http://docs.jquery.com/UI*.

В дополнение к возможности перетаскивания элементов и другим способам взаимодействия с мышью, модуль UI Plugin предоставляет мно-

жество визуальных компонентов для расширения набора основных элементов пользовательского интерфейса, предоставляемых языком разметки HTML.

9.4.2. Визуальные компоненты и эффекты

Мы бы и рады более подробно осветить обширный набор компонентов пользовательского интерфейса, предоставляемый модулем UI Plugin, но реальность неумолима. Вместо глубокого изучения мы хотя бы покажем вам возможности модуля, чтобы вы знали, что искать в документации на сайте.

В следующем списке перечислены визуальные компоненты с краткими описаниями. За дополнительной информацией по этим элементам обращайтесь по адресу *http://docs.jquery.com/ui*.

Accordion – простой визуальный компонент, позволяющий создавать разворачивающиеся и сворачивающиеся элементы интерфейса, такие как списки или вложенные элементы `<div>`.

Tabs – компонент, предназначенный для создания страниц с вкладками, обладает довольно интересными и разнообразными параметрами. Для создания *вкладок* применяются якорные элементы, а значения их атрибутов `href` позволяют идентифицировать разделы вкладок (для чего используется хеш ссылок внутри страницы). Этот компонент учитывает наличие кнопки броузера Назад и может быть настроен на открытие определенной вкладки в момент загрузки страницы.

Calendar – этот компонент представляет собой элемент выбора даты, ассоциированный с элементом ввода. У этого компонента много параметров настройки, он может отображаться как внутри страницы, так и в виде отдельного окна диалога.

Dialog – компонент, который реализует модальный диалог, обладающий возможностью перетаскивания и изменения размеров.

Slider – компонент, создающий *бегунок* (похожий на те, что есть в обычных настольных приложениях), который может быть интегрирован в форму через скрытые элементы. Компонент позволяет настраивать его ориентацию, а также минимальное и максимальное значения.

Table – компонент таблицы, поддерживающей быструю сортировку.

В дополнение к этим компонентам предоставляются следующие визуальные эффекты:

Shadow – создает эффект тени вокруг указанных элементов.

Magnifier – увеличивает размеры содержимого элементов при приближении указателя мыши.

Эти элементы пользовательского интерфейса расширяют наши возможности при создании прекрасных полнофункциональных интернет-приложений. Но, как говорится, «Минутку! Есть еще кое-что!».

Сообщество пользователей jQuery стремится поделиться своими расширениями для библиотеки jQuery. Посетите страницу *http://jquery.com/plugins*, где вы найдете много (действительно *много!*) модулей расширения, переданных другими пользователями jQuery.

9.5. Итоги

Создатели jQuery изначально разрабатывали библиотеку так, чтобы она имела простую, но надежную расширяемую архитектуру. Основная идея состояла в том, чтобы сохранить размер загружаемого файла с ядром библиотеки как можно меньшим и обеспечить только самые необходимые функциональные возможности, оставляя возможность реализовать недостающие особенности в виде модулей расширения, которые можно подключать по мере необходимости. Такая стратегия оправдала себя, поскольку сообществом пользователей jQuery было создано и передано в общее пользование великое множество расширений, которые может загрузить и использовать любой желающий.

Рассмотрев некоторые из наиболее часто используемых модулей расширения, мы увидели широчайший диапазон функциональных возможностей, расширяющих базовый набор функций jQuery.

Модуль Form Plugin предоставляет методы обертки, позволяющие обрабатывать элементы форм и даже обеспечивающие возможность легко и просто отправлять формы на сервер с применением механизмов технологии Ajax, не прибегая к традиционным методам, при которых требуется полная перезагрузка страницы.

Возможность получения точных (или не очень точных, но зато быстро) координат и размеров элементов DOM обеспечивается модулем расширения Dimensions Plugin, незаменимым при необходимости размещать элементы на странице друг относительно друга или относительно начала координат страницы с высокой точностью.

Другой, не менее востребованный модуль расширения Live Query Plugin позволяет устанавливать обработчики событий в элементы, которых даже еще нет. Очевидно, что это неоценимая возможность для страниц, в которых, как ожидается, будут часто создаваться и уничтожаться элементы DOM.

Модуль расширения UI Plugin, который, к нашему сожалению, не удалось исследовать достаточно полно, предоставляет такие возможности пользовательского интерфейса, как перетаскивание и сортировка, а также ряд полезных визуальных компонентов пользовательского интерфейса.

И это только начало. Посетите страницу *http://jquery.com/plugins*, где вы найдете полный список всех доступных модулей расширения, который постоянно пополняется!

9.6. Конец?

Едва ли!

Даже при том, что мы представили весь прикладной интерфейс jQuery, в ограниченных рамках книги было бы невозможно показать все многообразие способов использования этого интерфейса в страницах. Примеры, которые мы приводили здесь, были созданы с расчетом показать вам, как с помощью jQuery можно решать свои проблемы, с которыми ежедневно приходится сталкиваться при разработке страниц веб-приложений.

Библиотека jQuery – это развивающийся проект. Для авторов в ходе работы над этой книгой оказалось сложным делом уследить за быстрым развитием библиотеки. Ядро библиотеки становится все более полезным ресурсом, и с каждым днем все больше модулей расширения.

Мы настоятельно рекомендуем вам следить за развитием библиотеки и искренне надеемся, что эта книга оказала вам немалую помощь в том, чтобы начать писать полнофункциональные интернет-приложения с меньшим объемом программного кода, чем можно было бы себе представить.

Мы желаем вам счастья и здоровья, и пусть все ваши проблемы будут легко разрешимыми!

В этом приложении:

- Понятия JavaScript, важные для эффективного использования jQuery
- Основы объекта `Object` в языке JavaScript
- Почему функции – это обычные объекты
- Определение (и управление) понятия `this`
- Что такое замыкание?

A

JavaScript: что вам нужно знать, а может, и нет!

Одно из главных преимуществ применения jQuery в веб-приложениях – возможность выполнять много сценариев без необходимости писать эти сценарии. jQuery позволяет нам сконцентрироваться на создании своих приложений и заставить их делать то, что они должны делать!

Чтобы писать и понимать примеры первых нескольких глав этой книги, достаточно было лишь самых элементарных навыков работы с JavaScript. В главах, посвященных более сложным темам, таким как обработка событий, анимация и Ajax, требовалось понимать основные концепции JavaScript, направленные на эффективное использование библиотеки jQuery. Вы могли обнаружить, что многое в JavaScript, поначалу принимаемое за нечто само собой разумеющееся (или на веру), стало приобретать особый смысл.

Мы не собираемся освещать все концепции JavaScript полностью – это не для данной книги. Цель книги – эффективно овладеть jQuery, чем быстрее, тем лучше. Поэтому мы сосредоточимся на основных концепциях, необходимых для наиболее эффективного применения jQuery в веб-приложениях.

Наиболее важные из этих концепций касаются того, как JavaScript определяет и работает с функциями, а именно – подхода, при котором функции являются обычными объектами JavaScript. Что мы имеем в виду? Чтобы понять, как функция может быть объектом, мы должны прежде всего убедиться в понимании того, что представляет из себя собственно объект JavaScript. Итак, приступим.

A.1. Основные сведения об объекте Object языка JavaScript

Большинство объектно-ориентированных (ОО) языков определяют базовый тип Object, как порождающий все другие объекты. В языке JavaScript тоже есть базовый объект Object, который служит основой для всех других объектов, но этим сходство и ограничивается. По сути, у объекта Object мало общего с базовыми объектами, определяемыми в родственных объектно-ориентированных языках.

На первый взгляд, Object – скучный и обыденный элемент. После создания он не содержит никаких данных и предоставляет минимум семантических действий. Но такая ограниченная семантика в действительности дает ему большой потенциал.

Давайте посмотрим, каким образом.

A.1.1. Как создаются объекты

Новый объект создается с помощью оператора new в паре с конструктором Object. Объект создается просто:

```
var shinyAndNew = new Object();
```

Можно сделать это еще проще (как мы вскоре увидим), но пока будем поступать именно так.

Но что можно *сделать* с этим новым объектом? У него, казалось бы, ничего нет: ни информации, ни сложной семантики, ничего. Наш совершенно новый, чистый объект не представляет интереса до тех пор, пока мы не начнем добавлять к нему некоторые атрибуты, называемые *свойствами*.

A.1.2. Свойства объектов

Подобно своим серверным аналогам, объекты JavaScript могут содержать данные и обладать методами (например, сортировки, но не будем спешить). В отличие от своих серверных собратьев, эти элементы в объекте заранее не объявлены, мы создаем их динамически, по мере необходимости.

Взгляните на следующий фрагмент:

```
var ride = new Object();
ride.make = 'Yamaha';
ride.model = 'V-Star Silverado 1100';
ride.year = 2005;
ride.purchased = new Date(2005,3,12);
```

Мы создаем новый экземпляр Object и присваиваем его переменной ride. Затем наполняем эту переменную рядом *свойств* различных типов: два строковых свойства, числовое свойство и свойство типа Date.

Нам не надо объявлять эти свойства до операции присваивания, они появляются автоматически, в момент присваивания им значений. Это чрезвычайно мощная особенность, дающая нам большую гибкость. Но не будем слишком легкомысленными, помня о том, что за гибкость придется платить!

Предположим, нам требуется изменить значение даты покупки:

```
ride.purchased = new Date(2005,2,1);
```

Ничего страшного... если мы случайно не сделаем опечатку, например:

```
ride.purcahsed = new Date(2005,2,1);
```

Здесь нет компилятора, который предупредил бы вас о сделанной ошибке; новое свойство с именем `purcahsed`, с готовностью созданное по вашему требованию, позже заставит задуматься о том, почему вы *не получили* новую дату, обратившись к свойству с правильно указанным именем.

С большими возможностями приходит большая ответственность (вы уже слышали об этом?), так что печатайте внимательно!

Примечание

Отладчики JavaScript, такие как Firebug для Firefox, могут облегчить жизнь при решении таких проблем. Поскольку опечатки подобного типа зачастую не приводят к ошибкам во время выполнения, консоль JavaScript и диалоговые окна с сообщениями об ошибках обычно менее эффективны.

Из этого примера мы узнали, что экземпляр `Object` в языке JavaScript, который мы далее будем называть просто *объект*, – это набор свойств, каждое из которых состоит из *имени* и *значения*. Имя свойства – это строка, а значение может быть любым объектом JavaScript, таким как `Number`, `String`, `Date`, `Array`, `Object` или любой другой тип JavaScript (в том числе и функция, как мы увидим).

Таким образом, основная цель экземпляра `Object` заключается в том, чтобы служить контейнером для именованных наборов других объектов. Это может напомнить концепции других языков, например отображение (map) в языке Java или словари и хеши в других языках.

Обращаясь к свойствам, мы можем по цепочке ссылаться на свойства объектов, выступающих в качестве свойств родительского объекта. Предположим, мы добавили к нашему объекту `ride` новое свойство, описывающее владельца транспортного средства. Это свойство – еще один объект JavaScript, который в свою очередь содержит такие свойства, как имя и профессия владельца:

```
var owner = new Object();
owner.name = 'Spike Spiegel';
owner.occupation = 'bounty hunter';
ride.owner = owner;
```

Обращение к вложенным свойствам можно описать так:

```
var ownerName = ride.owner.name;
```

Здесь нет никаких ограничений на глубину вложенности (за исключением здравого смысла). Теперь иерархия наших объектов выглядит, как показано на рис. А.1.

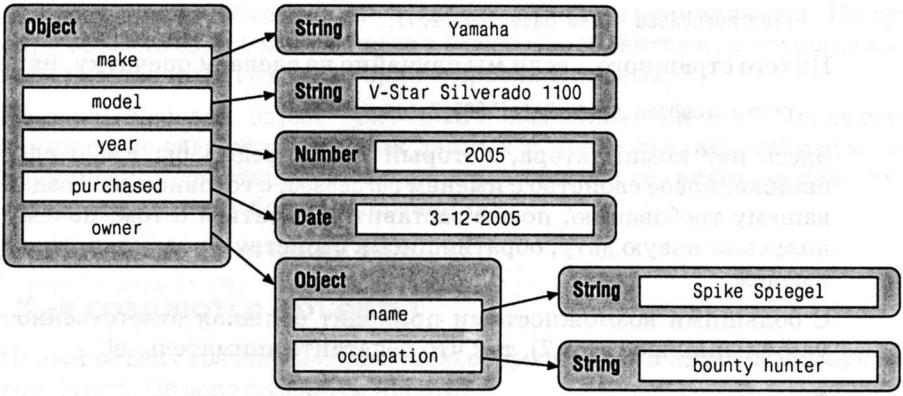


Рис. А.1. Иерархия наших объектов показывает, что объекты являются контейнерами для именованных ссылок на другие объекты JavaScript или на встроенные объекты JavaScript

Между прочим, нет никакой необходимости создавать промежуточные переменные (такие как `owner`) – мы создали их в этих фрагментах программного кода исключительно для наглядности. Далее вы увидите более эффективные и компактные способы объявления объектов и их свойств.

До этого момента мы ссылались на свойства объекта с помощью оператора «точка» (символ точки), но, как выясняется, это синоним более общего оператора для выполнения ссылок на свойства.

Что если, к примеру, у нас есть свойство с именем `color.scheme`? Заметили точку в середине имени? Это портит все дело, потому что интерпретатор JavaScript будет пытаться найти `scheme` как вложенное свойство `color`.

«Тогда просто не делайте так!» – скажете вы. Но как быть с пробельными символами? Как быть с другими символами, которые могут восприниматься как разделители, а не как часть имени? И, что самое главное, мы даже не знаем, чем является это свойство – значением другой переменной или результатом выражения.

Для всех этих случаев оператор «точка» не подходит, и для доступа к свойствам мы должны выбрать более общее обозначение. Вот более общий формат обращения к свойствам:

```
object[propertyNameExpression]
```

propertyNameExpression – выражение JavaScript, которое определяется как строка, формирующая имя свойства, к которому происходит обращение. Например, все три следующие ссылки эквивалентны:

```
ride.make
ride['make']
ride['m'+'a'+'k'+'e']
```

Также как:

```
var p = 'make';
ride[p];
```

Применение общего оператора ссылки – единственный способ обратиться к свойствам, имена которых не являются допустимыми идентификаторами JavaScript, например:

```
ride["a property name that's rather odd!"]
```

Такие имена содержат символы, недопустимые для идентификаторов JavaScript, или являются значениями других переменных.

Построение объектов путем создания новых экземпляров с помощью оператора new и присваивание значений каждому свойству с помощью отдельных операторов – утомительное занятие. В следующем разделе мы рассмотрим более компактные и удобочитаемые способы объявления объектов и их свойств.

A.1.3. Литералы объектов

В предыдущем разделе мы создали объект, моделирующий некоторые свойства мотоцикла, и присвоили его переменной с именем ride. Мы сделали это с помощью двух операторов new, промежуточной переменной с именем owner и тучи операторов присваивания. Это утомительно, кроме того, приходится вводить с клавиатуры много текста, что способствует появлению ошибок и делает затруднительным визуальное восприятие структуры объекта при просмотре кода.

К счастью, мы можем использовать более компактную и удобную для визуального восприятия запись.

Рассмотрим инструкцию:

```
var ride = {
  make: 'Yamaha',
  model: 'V-Star Silverado 1100',
  year: 2005,
  purchased: new Date(2005,3,12),
  owner: {
    name: 'Spike Spiegel',
    occupation: 'bounty hunter'
  }
};
```

В этом фрагменте с помощью *литерала объекта* создается тот же самый объект ride, который в предыдущем разделе мы создали с помощью операторов присваивания.

Большинство авторов страниц отдают предпочтение такой форме записи, называемой *JSON* (JavaScript Object Notation[1]), когда при построении объекта приходится использовать множество операций присваивания. Структура этой формы записи очень проста – объект обозначается парой фигурных скобок, внутри которых через запятую перечислены свойства. Каждое свойство обозначается путем записи его имени и значения, разделенных символом двоеточия.

Примечание

С технической точки зрения, формат JSON не обладает возможностью определять значения даты, в первую очередь потому, что в JavaScript отсутствует литерал для определения даты. При использовании в сценарии, как правило, применяется конструктор Date, как показано в предыдущем примере. При использовании в качестве формата обмена данными даты часто выражаются строкой в формате ISO 8601 либо числом, определяющим дату как количество миллисекунд, возвращаемое функцией Date.getTime().

Как видно из объявления свойства owner, объявления объектов могут быть вложенными.

Кстати, точно так же в формате JSON можно описывать массивы, поместив список элементов, разделенных запятыми, в квадратные скобки:

```
var someValues = [2,3,5,7,11,13,17,19,23,29,31,37];
```

Как показано в примерах этого раздела, ссылки на объекты часто хранятся в переменных или в свойствах других объектов. Рассмотрим особый случай последнего сценария.

A.1.4. Объекты как свойства объекта window

До этого момента мы видели два способа хранения ссылок на объекты JavaScript – в переменных и в свойствах. Эти два способа хранения ссылок записываются по-разному, как показано в следующем фрагменте:

```
var aVariable =
  'Before I teamed up with you, I led quite a normal life.';

someObject.aProperty =
  'You move that line as you see fit for yourself.';
```

В этих двух инструкциях два экземпляра String (созданные с помощью литералов) присваиваются переменной и свойству объекта соответственно, с помощью операторов присваивания. (Честь и хвала тому, кто сможет определить источник этих цитат, не пользуясь Google! Выше была подсказка.)

[1] Дополнительная информация доступна по адресу *http://www.json.org/*.

Но *действительно* ли эти инструкции выполняют разные операции? Как выясняется – нет!

Когда ключевое слово `var` используется на глобальном уровне, за пределами какой-либо функции, эта удобная для программиста форма записи является всего лишь ссылкой на свойство предопределенного объекта JavaScript `window`. Любая глобальная ссылка неявно превращается в свойство объекта `window`.

Это означает, что эквивалентны все следующие инструкции:

```
var foo = bar;
```

и

```
window.foo = bar;
```

и

```
foo = bar;
```

Независимо от формы записи во всех трех случаях создается свойство объекта `window` с именем `foo` (если оно еще не существовало), и ему присваивается значение `bar`. Кроме того, поскольку идентификатор `bar` никак не квалифицирован, предполагается, что он представляет собой имя свойства объекта `window`.

Надеемся, вы не подумали, что глобальная область видимости – это область видимости объекта `window`, потому что любые неквалифицированные *глобальные* ссылки подразумевают ссылки на свойства объекта `window`. Правила области видимости еще усложнятся, когда мы углубимся в изучение тел функций, причем сильно усложнятся, и случится это достаточно скоро.

На этом обзор объекта `Object` в языке JavaScript можно считать практически завершенным. Здесь были рассмотрены следующие важные понятия:

- Объект в языке JavaScript – это неупорядоченный набор свойств.
- Свойство состоит из имени и значения.
- Можно объявлять объекты посредством литералов объектов.
- Глобальные переменные являются свойствами объекта `window`.

Теперь посмотрим, что имелось в виду под словами «функции в JavaScript – это обычные объекты».

A.2. Функции как обычные объекты

Во многих традиционных объектно-ориентированных языках объекты могут содержать данные и обладать методами. В этих языках данные и методы, как правило, – не одно и то же, но язык JavaScript пошел другим путем.

В языке JavaScript функции считаются объектами, подобно объектам любого другого типа, определенного в JavaScript, например `String`, `Number` или `Date`. Как и другие объекты, функции определяются конструктором JavaScript, в данном случае – `Function`, и могут:

- присваиваться переменным;
- присваиваться свойствам объектов;
- передаваться в виде параметров;
- возвращаться как результат других функций;
- создаваться с использованием литералов.

Поскольку в некоторых случаях функции рассматриваются как объекты, мы говорим, что функции – это *обычные объекты*.

Вы могли бы подумать, что функции кардинально отличаются от объектов других типов, таких как `String` или `Number`, потому что они обладают не только значением (тело функции, если речь идет об экземпляре `Function`), но еще и *именем*.

Не спешите с выводами!

A.2.1. Что есть имя?

Большинство программистов JavaScript напрасно полагают, что функции являются именованными сущностями. Это не так. Если вы один из таких программистов, то вас сбили с толку искусно замаскированные уловки. Как и в случае с экземплярами других объектов, будь то `String`, `Date` или `Number`, – на функции ссылаются, только когда они присваиваются переменным, свойствам или параметрам.

Рассмотрим объекты типа `Number`. Мы часто описываем экземпляр `Number` буквенным обозначением, таким как `213`. Инструкция

```
213;
```

вполне правильна, но совершенно бесполезна. Экземпляр `Number` не всегда полезен, если не присваивается свойству или переменной или не связан с именем параметра. В противном случае, мы не сможем обратиться к такому экземпляру.

Это же относится и к экземплярам объектов `Function`.

«Стоп-стоп-стоп, – скажете вы. – А как насчет следующего фрагмента программного кода?»

```
function doSomethingWonderful() {
  alert('does something wonderful');
}
```

«Разве такое объявление не создает функцию с *именем* `doSomethingWonderful`?»

Нет, не создает. Несмотря на то что запись может показаться знакомой и часто используется для создания глобальных функций, это тот же са-

мый синтаксический подсластитель, который использует ключевое var для создания свойств window. Ключевое слово function автоматически создает экземпляр Function и присваивает его свойству window, имя которого совпадает с «именем» функции (то, что мы ранее назвали искусно замаскированными уловками), как показано в следующем примере:

```
doSomethingWonderful = function() {
  alert('does something wonderful');
}
```

Если это объявление выглядит странным, взгляните на другую инструкцию, где используется точно такой же формат, за исключением того, что здесь участвует литерал Number:

```
aWonderfulNumber = 213;
```

Здесь нет ничего странного, и инструкция присваивания функции глобальной переменной (свойству объекта window) ничем не отличается – литерал функции используется для создания экземпляра Function, а затем присваивается переменной doSomethingWonderful так же, как литерал 213 объекта Number был использован для присваивания экземпляра Number переменной aWonderfulNumber.

Если вы никогда раньше не встречались с синтаксисом *литерала функции*, это может показаться странным. Он состоит из ключевого слова function, за которым идет список параметров, заключенный в круглые скобки, и далее следует тело функции.

Когда мы объявляем глобальную именованную функцию, создается экземпляр Function и *присваивается* свойству объекта window, которое создается автоматически, на основе так называемого имени функции. Сам по себе экземпляр Function не имеет имени, как литерал Number или String. Это понятие иллюстрирует рис. А.2.

Помните, что, когда в HTML-странице создается глобальная переменная, она создается как свойство объекта window. Поэтому все следующие инструкции эквивалентны:

```
function hello(){ alert('Hi there!'); }
```

Броузеры Gecko и имена функций

Броузеры на основе механизма отображения Gecko, например Firefox и Camino, хранят имена функций, определенных с применением глобального синтаксиса, в нестандартном свойстве функции с именем name. Хотя эта особенность не очень полезна для широкого круга разработчиков – особенно если учесть, что она присутствует только в броузерах, созданных на базе Gecko, – тем не менее она важна для авторов расширений и отладчиков.

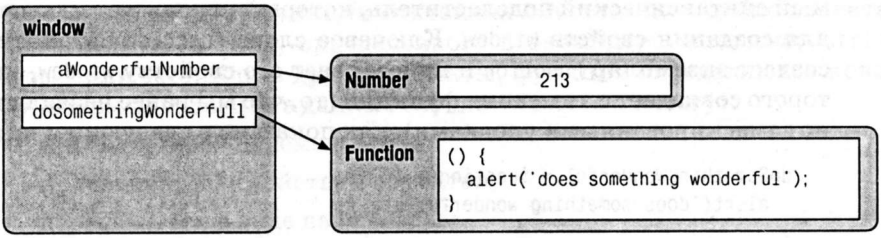


Рис. А.2. Экземпляр Function является неименованным объектом, таким же, как значение 213 типа Number или любое другое значение JavaScript. Он именуется только ссылками, которые созданы для него

```
hello = function(){ alert('Hi there!'); }

window.hello = function(){ alert('Hi there!'); }
```

Хотя все это может показаться синтаксическими манипуляциями, очень важно понять, что экземпляры Function являются *значениями*, которые можно присвоить переменным, свойствам или параметрам, а также экземплярам объектов других типов. И подобно этим объектам других типов безымянные экземпляры нельзя использовать, если они не связаны с переменными, свойствами или параметрами, через которые на них можно сослаться.

Мы посмотрели примеры присваивания функций переменным и свойствам – а как насчет передачи функции в качестве параметра? Давайте посмотрим, зачем и как можно это сделать.

А.2.2. Функции обратного вызова

Глобальные функции хороши, когда наш программный код следует красивым и упорядоченным синхронным потоком, но характерной чертой HTML-страниц сразу после загрузки является асинхронность. Будь то обработка событий, установка таймеров или выполнение запросов Ajax, программный код веб-страницы по своей природе является асинхронным. И одно из самых распространенных понятий в асинхронном программировании – понятие *функции обратного вызова*.

Возьмем в качестве примера таймер. Мы можем заставить таймер сработать, например, через пять секунд, передав соответствующее значение длительности методу window.setTimeout(). Но каким образом этот метод сообщит нам о том, что время таймера истекло, чтобы мы могли выполнить необходимые действия по истечении времени ожидания? Делается это путем вызова функции, которую *мы* предоставим.

Рассмотрим следующий программный код:

```
function hello() { alert('Hi there!'); }

setTimeout(hello,5000);
```

Мы объявляем функцию с именем hello и устанавливаем таймер на 5 секунд, заданных во втором параметре как 5000 миллисекунд. В первом параметре мы передаем методу setTimeout() ссылку на функцию. Передача функции в виде параметра ничем не отличается от передачи любого другого значения – точно так же мы передали во втором параметре значение типа Number.

Когда время таймера истечет, будет вызвана функция hello. Поскольку метод setTimeout() делает обратный вызов функции в нашем собственном программном коде, эту функцию называют *функцией обратного вызова*.

Этот пример программного кода покажется наивным большинству опытных программистов JavaScript, поскольку создание имени hello не является необходимостью. Если функцию не требуется вызвать где-нибудь в другом месте страницы, нет никакой необходимости создавать свойство hello в объекте window, чтобы на мгновение сохранить экземпляр Function и потом передать его в качестве параметра.

Вот более изящный способ записи этого фрагмента:

```
setTimeout(function() { alert('Hi there!'); },5000);
```

Здесь мы определяем функцию непосредственно в списке параметров в виде литерала, и нет никакой нужды создавать имя. Это – идиома, которая часто будет нам встречаться в программном коде jQuery, когда нет необходимости присваивать экземпляр функции глобальному свойству.

Функции, которые мы до сих пор создавали в примерах, – это либо глобальные функции (которые, как мы знаем, являются свойствами объекта window), либо параметры других функций. Мы также можем присваивать экземпляры Function свойствам объектов, и эта возможность действительно представляет интерес. Читайте дальше...

A.2.3. К чему это все?

Объектно-ориентированные языки программирования автоматически предоставляют средства для получения ссылки на текущий экземпляр объекта внутри метода. В таких языках, как Java и C++, на текущий экземпляр указывает переменная с именем this. В JavaScript тоже есть подобное понятие и даже используется то же самое ключевое слово this, которое обеспечивает доступ к объекту, связанному с функцией. Но будьте внимательны! Реализация this в JavaScript отличается от аналогов других объектно-ориентированных языков едва различимым, но существенным образом.

В объектно-ориентированных языках, основанных на классах, указатель this, как правило, ссылается на экземпляр класса, в пределах которого был объявлен метод. В JavaScript, где функции являются обычными объектами, они не объявляются как часть чего-либо. Объект, на который ссылается this, называется *контекстом функции* и определяется не тем, как функция объявляется, а тем, как она *вызывается*.

Это означает, что одна и та же функция может иметь *различный* контекст в зависимости от того, как она вызывается. На первый взгляд, это кажется странным, но может оказаться весьма полезным.

По умолчанию контекст (this) функции – это объект, свойство которого содержит ссылку для вызова функции. Давайте вернемся к нашему примеру с мотоциклом и изменим создание объекта (дополнения выделены жирным шрифтом):

```
var ride = {
  make: 'Yamaha',
  model: 'V-Star Silverado 1100',
  year: 2005,
  purchased: new Date(2005,3,12),
  owner: {name: 'Spike Spiegel',occupation: 'bounty hunter'},
  whatAmI: function() {
    return this.year+' '+this.make+' '+this.model;
  }
};
```

К первоначальному коду примера мы добавили свойство с именем whatAmI, которое ссылается на экземпляр Function. Новая иерархия объектов, включающая экземпляр Function в свойстве с именем whatAmI, показана на рис. А.3.

Если функция вызывается через свойство

```
var bike = ride.whatAmI();
```

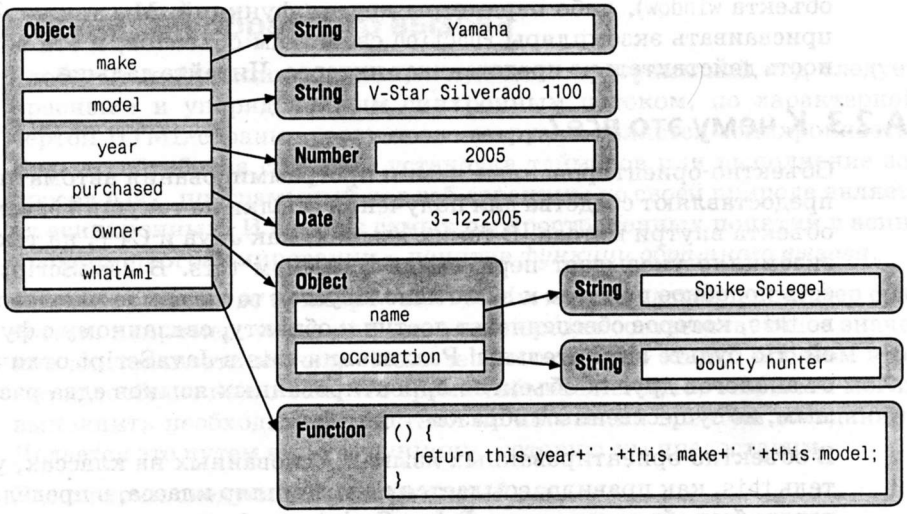


***Рис. А.3.** Эта модель ясно показывает, что функция не является частью объекта Object, она лишь доступна через свойство объекта, которое называется whatAmI*

то в качестве контекста функции (ссылка this) устанавливается экземпляр объекта, на который указывает ride. В результате в переменную bike записывается строка *2005 Yamaha V-Star Silverado 1100*, потому что функция выбирает с помощью this значения свойств объекта, посредством которого она была вызвана.

То же справедливо и для глобальных функций. Помните, что глобальные функции являются свойствами объекта window, поэтому контекстом таких функций при их вызове *как* глобальных функций является объект window.

Хотя такое неявное поведение вполне обычно, JavaScript позволяет нам четко установить, что будет использоваться в качестве контекста функции. Мы можем передать в контексте функции все, что угодно, вызвав функцию с помощью метода call() или apply() экземпляра Function.

Кроме того, будучи обычными объектами, даже функции имеют методы, определяемые конструктором Function.

Метод call() вызывает функцию, передавая в качестве первого параметра объект, который является контекстом функции, а в качестве остальных параметров передаются параметры вызываемой функции – вторым параметром метода call() становится первый аргумент вызываемой функции и т. д. Метод apply() работает аналогично, за исключением того, что вторым параметром он ожидает получить массив объектов, которые становятся аргументами вызываемой функции.

Запутались? Пришло время для более обстоятельного примера. Рассмотрим листинг А.1 (файл *appendixA/function.context.html*)

Листинг А.1. Покажем, что значение контекста функции зависит от способа вызова функции

```
<html>
  <head>
    <title>Function Context Example</title>
    <script>
      var o1 = {handle:'o1'};
      var o2 = {handle:'o2'};
      var o3 = {handle:'o3'};
      window.handle = 'window';

      function whoAmI() {
        return this.handle;
      }

      o1.identifyMe = whoAmI;

      alert(whoAmI());
      alert(o1.identifyMe());
      alert(whoAmI.call(o2));
      alert(whoAmI.apply(o3));
    </script>
  </head>
```

❶ ❷ ❸ ❹ ❺ ❻ ❼

```
    <body>
    </body>
</html>
```

В данном примере мы определяем три простых объекта, у каждого из которых есть свойство `handle`, позволяющее легко идентифицировать объект по ссылке на него. Мы также добавляем свойство `handle` в экземпляр объекта `window`, чтобы его тоже можно было легко идентифицировать.

Затем мы определяем глобальную функцию, которая возвращает значение свойства `handle` для любого объекта, используемого в качестве контекста функции ❷, и присваиваем ту же самую функцию свойству `identifyMe` ❸ объекта `o1`. Можно сказать, что тем самым был создан метод объекта `o1` с именем `identifyMe`, хотя важно отметить, что функция объявляется независимо от объекта.

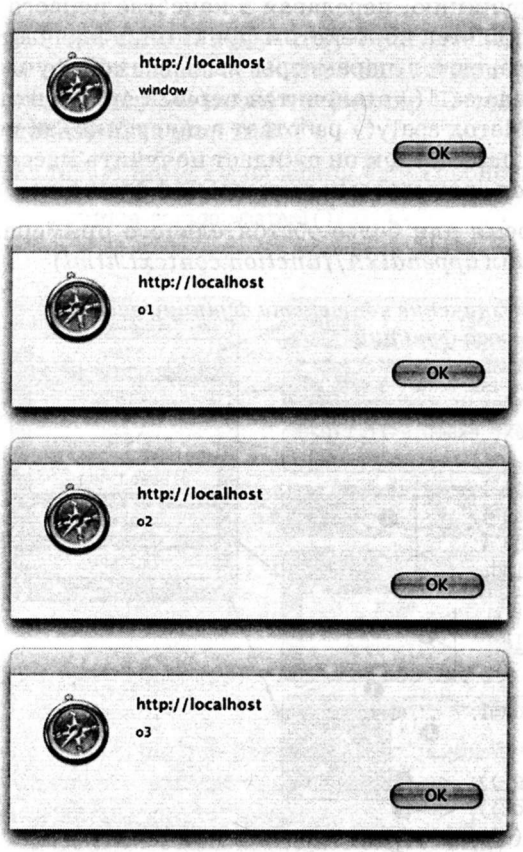


Рис. А.4. В зависимости от того, как вызывалась функция, изменяется объект, выступающий в качестве контекста функции

Наконец, мы выводим четыре предупреждения, каждый раз вызывая один и тот же экземпляр функции другим способом. Последовательность из четырех предупреждений, выведенных после открытия страницы в броузере, показана на рис. А.4.

Эта последовательность сообщений иллюстрирует следующее:

- Если функция вызывается как глобальная функция, контекстом функции является экземпляр объекта `window` ❹.
- Если функция вызывается как свойство объекта (`o1` в данном случае), контекстом функции становится этот объект ❺. Мы могли бы сказать, что функция действует как метод этого объекта – аналогично объектно-ориентированным языкам. Но не слишком радуйтесь этой аналогии. Вы можете сбиться с пути, если не будете осторожны, что показывает оставшаяся часть этого примера.
- Использование метода `call()` объекта `Function` приводит к тому, что контекстом функции становится любой объект, полученный методом `call()` в качестве первого параметра, – в данном случае, `o2` ❻. В этом примере функция действует как метод объекта `o2` при том, что она никак не связана с объектом `o2`, даже как свойство.
- Как и в случае с методом `call()`, при использовании метода `apply()` контекстом функции становится любой объект, переданный в качестве первого параметра ❼. Разница между этими двумя методами становится существенной, только когда параметры передаются функции (чего мы в этом примере не делали для простоты).

Этот пример страницы явно свидетельствует о том, что контекст функции определяется способом вызова и что одна и та же функция может быть вызвана с любым объектом, выступающим в качестве ее контекста. Поэтому, скорее всего, будет ошибкой сказать, что функция является методом объекта. Гораздо правильнее сказать так:

> *Функция `f` действует как метод объекта `o`, когда объект `o` выступает в качестве контекста функции при вызове функции `f`.*

В качестве дополнительной иллюстрации данной концепции рассмотрим результат добавления к нашему примеру следующей инструкции:

```
alert(o1.identifyMe.call(o3));
```

Даже при том, что мы ссылаемся на функцию как на свойство объекта `o1`, роль контекста функции в этом вызове играет объект `o3`. Далее подчеркнем еще раз, что дело не в том, как функция объявляется, а в том, как она вызывается, что и определяет контекст функции.

При использовании команд и функций jQuery применяются функции обратного вызова, что доказывает важность этой концепции. Мы видели эту концепцию в действии и ранее (даже если тогда вы этого не понимали), в разделе 2.3.3, где мы передавали функцию обратного вызова методу `filter()` объекта `$`, и эта функция последовательно вызыва-

лась для всех элементов обернутого набора, которые в свою очередь выступали в качестве контекста функции.

Теперь, когда мы понимаем, каким образом функции могут действовать в качестве методов объектов, перейдем к другой достаточно сложной теме, важной для эффективного использовании jQuery, – к замыканиям.

A.2.4. Замыкания

Для авторов страниц, пришедших из традиционного объектно-ориентированного или процедурного программирования, *замыкания* часто являются странным понятием, тогда как для тех, кто знаком с функциональным программированием, замыкания являются знакомым и удобным понятием. Что же такое замыкания?

Сформулируем просто, насколько это возможно: *замыкание* (*closure*) – это экземпляр `Function` вместе с локальными переменными из его окружения, необходимыми для выполнения.

При объявлении функция может ссылаться на любые переменные, находящиеся в ее области видимости на момент объявления. Эти переменные достижимы для функции *даже после* того, как текущее положение в объявлении выйдет из области видимости, замыкая объявление.

Возможность для функций обратного вызова ссылаться на локальные переменные, действующие на момент объявления, – важный инструмент создания эффективного программного кода JavaScript. Воспользовавшись таймером еще раз, посмотрим наглядный пример в листинге А.2 (файл *appendixA/closure.html*).

Листинг А.2. Получение доступа к окружению функции, существовавшему на момент объявления, через замыкания

```
<html>
  <head>
    <title>Closure Example</title>
    <script type="text/javascript"
            src="../scripts/jquery-1.2.js"></script>
    <script>
      $(function(){
        var local = 1;                          ❶
        window.setInterval(function(){          ❷
          $('#display')
            .append('<div>At '+new Date()+' local='+local+'</div>');
          local++;                              ❸
        },3000);
      });
    </script>
  </head>

  <body>
    <div id="display"></div>                    ❹
  </body>
```

```
</html>
```

В данном примере мы определяем обработчик события готовности документа, который запускается после загрузки дерева DOM. В этом обработчике мы объявляем локальную переменную с именем `local` ❶ и присваиваем ей числовое значение 1. Затем с помощью метода `window.setInterval()` взводим таймер, который будет срабатывать каждые 3 секунды ❷. В качестве функции обратного вызова для таймера мы определяем встроенную функцию, которая ссылается на переменную `local` и показывает текущее время и значение переменной `local`, добавляя элемент `<div>` в элемент с именем `display`, определенный в теле страницы ❹. Кроме того, внутри функции обратного вызова значение переменной `local` увеличивается ❸.

Если бы мы были незнакомы с замыканиями, то до запуска этого примера могли бы заметить в его коде некоторые проблемы. Мы могли бы предположить, что поскольку функция обратного вызова запустится через три секунды после загрузки страницы (что произойдет далеко не сразу после того, как обработчик события готовности документа закончит выполняться), то во время выполнения функции обратного вызова значение переменной `local` окажется неопределенным. В конце концов, блок, в котором объявляется переменная `local`, выходит из области видимости, когда обработчик события готовности документа заканчивает работу, правильно?

Но после загрузки страницы, позволив ей отработать, через короткий промежуток времени мы увидим изображение, как на рис. А.5.

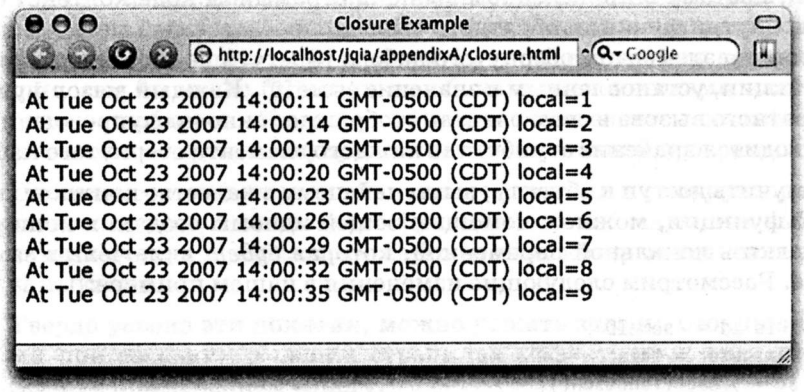


***Рис. А.5.** Замыкания позволяют функциям обратного вызова получать доступ к их окружению, даже если это окружение выходит из области видимости*

Она работает! Но каким образом?

Несмотря на то что блок, в котором объявляется переменная `local`, *действительно* выходит из области видимости, когда обработчик события

готовности документа завершит работу, замыкание, созданное в объявлении функции, включающее в себя переменную local, остается в области видимости функции на протяжении всего жизненного цикла.

Примечание

Возможно, вы заметили, что это замыкание, как и все замыкания в JavaScript, было создано неявно, поскольку нет необходимости явного синтаксиса, как это требуется в некоторых других языках, поддерживающих замыкания. Это обоюдоострый меч – с его помощью легко создать замыкание (предполагаете вы это или нет!), но он может и усложнить их реализацию в программном коде.

Непредусмотренные замыкания могут вызвать непредвиденные последствия. Например, циклические ссылки могут привести к утечкам памяти. Классический пример этого – создание элементов DOM, которые ссылаются на переменные в замыканиях, препятствуя удалению этих переменных.

Еще одна важная особенность замыканий заключается в том, что контекст функции никогда не является частью замыкания. Например, следующий фрагмент кода не будет выполняться, как можно было бы ожидать:

```
...
this.id = 'someID';
$('*').each(function(){
  alert(this.id);
});
```

Помните, что у каждого вызова функции собственный контекст функции, поэтому в приведенном выше программном коде контекст функции внутри функции обратного вызова, передаваемый методом each(), является элементом обернутого набора jQuery, а не свойством внешней функции, установленным в значение 'someID'. Каждый вызов функции обратного вызова в свою очередь отображает окно предупреждения, где выводится значение атрибута id каждого элемента обернутого набора.

Получить доступ к объекту, выступающему в качестве контекста внешней функции, можно с помощью общей идиомы создания копии этой ссылки в локальной переменной, которая *будет* включена в замыкание. Рассмотрим следующие изменения в нашем примере:

```
this.id = 'someID';
var outer = this;
$('*').each(function(){
  alert(outer.id);
});
```

Локальная переменная outer, которой присваивается ссылка на контекст внешней функции, становится частью замыкания и будет доступна внутри функции обратного вызова. Теперь измененный программный код будет выводить предупреждение со строкой 'someID' столько раз, сколько элементов в обернутом в наборе.

Замыкания действительно незаменимы для создания элегантного программного кода с применением команд jQuery, использующих асинхронные обратные вызовы, что особенно актуально в случае применения Ajax-запросов и обработки событий.

А.3. Итоги

JavaScript – это язык, широко распространенный в Сети, но многие авторы страниц, создающие веб-приложения, зачастую используют *далеко не все* его возможности. В этом Приложении мы представили часть глубинных аспектов языка, которые необходимо понимать, чтобы эффективно применять jQuery на своих страницах.

Мы увидели, что объект `Object` в языке JavaScript в первую очередь нужен затем, чтобы служить *контейнером* для других объектов. Если вы знакомы с объектно-ориентированным программированием, понимание экземпляра объекта как неупорядоченного набора пар имя/значение может оказаться далеким от того, что вы представляете себе под термином *объект*, но эта концепция важна при создании программного кода JavaScript, даже умеренной сложности.

Функции в JavaScript являются обычными объектами, которые могут быть объявлены и описаны таким же образом, как и объекты других типов. Мы можем объявлять их как литералы, сохранять в переменных и в свойствах объектов и даже передавать их другим функциям в виде параметров для использования в качестве функций обратного вызова.

Термин *контекст функции* описывает объект, на который указывает ссылка `this` внутри функции. Хотя функция может играть роль метода объекта, для чего объект должен быть определен как контекст функции, тем не менее функции не объявляются методами какого-то отдельного объекта. Контекст функции определяется способом вызова (возможно, под явным управлением вызывающей программы).

Наконец, мы видели, как объявление функции и ее окружение образуют *замыкание*, позволяющее функции при последующих вызовах получить доступ к локальным переменным, которые становятся частью замыкания.

Твердо усвоив эти понятия, можно решать задачи, стоящие перед нами при создании в наших страницах эффективных сценариев JavaScript с применением jQuery.

Алфавитный указатель

Специальные символы

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

A

Б

В

Г

Д

З

И

К

Л

М

Н

О

П

Р

С

Т

У

Ф

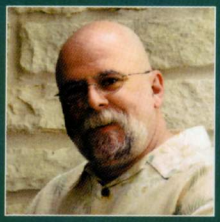
Х

Ц

Ч

Ш

Э



Бер Бибо занимается программированием уже тридцать лет. Ныне он разработчик архитектур программных комплексов и технический менеджер в компании, специализирующейся на создании и сопровождении крупных финансовых веб-приложений.

Иегуда Кац участвовал в разработке нескольких проектов с открытым исходным кодом. Он не только член основной команды проекта jQuery, но также участник проекта Merb, альтернативы Ruby on Rails (реализованной тоже на языке Ruby).

Категория: веб-программирование
Уровень подготовки читателей: средний

MANNING

Символ®
www.symbol.ru

Издательство «Символ-Плюс»
(812) 324-5353, (495) 945-8100

«jQuery. Подробное руководство по продвинутому JavaScript» – это краткое введение и справочное руководство по jQuery, мощной платформе для разработки веб-приложений. Разработчики сразу же оценят эту библиотеку JavaScripr, увидев, как с ее помощью 20 строк программного кода можно уменьшить втрое. Уникальная способность jQuery составлять «цепочки» из команд позволяет выполнять несколько последовательных операций над элементами страницы.

В книге подробно рассматривается, как:

- Выполнять обход документов HTML
- Обрабатывать события
- Манипулировать элементами DOM
- Воспроизводить анимацию и визуальные эффекты
- Добавлять поддержку технологии Ajax в веб-страницы

Описаны вопросы взаимодействия jQuery с другими инструментами и платформами, а также методы создания модулей расширения для этой библиотеки. Уникальные «лабораторные страницы» помогут закрепить изучение каждой новой концепции на практических примерах.

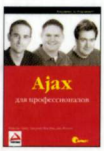
Книга предназначена для разработчиков, знакомых с языком JavaScript и технологией Ajax и стремящихся создавать краткий и понятный программный код – сократить его в несколько раз позволит грамотное использование библиотеки jQuery.

Спрашивайте наши книги:

Дари и др. AJAX и PHP. Разработка динамических веб-приложений

Флэнаган JavaScript. Подробное руководство, 5-е издание

Венц Программирование в ASP.NET AJAX



Закас и др. Ajax для профессионалов

ISBN-13 978-5-93286-135-6
ISBN-10 5-93286-135-5