Оглавление

Оглавление

Содержание

Введение

Случается, что проходит много лет, прежде чем человек поймет свое предназначение. Среди технологий, с которыми мне пришлось работать в начале 1990-х, был неприметный язык сценариев под названием JavaScript. Вскоре я понял, что, несмотря на название, он не имеет ничего общего с языком Java, с которым я предпочитал работать в то время. Однако судьба снова и снова сводила меня с JavaScript.

В конце 1990-х я решил остепениться и заняться работой, которая соответствовала бы моему возрасту. И как-то само собой оказалось, что я стал руководителем небольшой группы, занимающейся написанием программ для управления окнами и планировщиков, а также решением некоторых других задач. Надо ли говорить, что основная часть программного обеспечения полностью разрабатывалась на JavaScript. "Любопытно, — сказал я. — Ведь я никогда не стремился к этому".

Со временем я перешел на более ответственную работу. Моей задачей было написание базовых средств для службы сообщений корпоративной системы. При поступлении на работу было оговорено, что основным языком будет Java, но прошло немного времени, и незаметно моей главной задачей стала разработка пользовательских интерфейсов на JavaScript. Как ни странно, я все чаще стал слышать мнения специалистов о том, что JavaScript — достаточно серьезный язык и создание библиотек для него — перспективная работа. Вскоре мне пришлось ознакомиться с библиотекой x, разработанной Майком Фостером (Mike Foster); о ней вы узнаете, прочитав данную книгу. Однажды при разборе почты мне пришла в голову идея принимать новые сообщения в скрытом фрейме и включать их в состав пользовательского интерфейса *без обновления содержимого экрана*. Несколько часов вдохновенной работы — и в моих руках оказался действующий макет системы. Более того, я придумал выделять новые сообщения цветом, чтобы привлечь к ним внимание пользователя. Придя в хорошее расположение духа от создания интересной игрушки, я вернулся к серьезной работе. Я не знал, что приблизительно в то же время Эрик Костелло (Eric Costello), Эрик Хетчер (Erik Hatcher), Брент Эшли (Brent Ashley) и многие другие воплощали подобные идеи, а в компании Microsoft шла работа над объектом XMLHttpRequest для Outlook Web Access.

Судьба продолжала направлять меня по одному ей известному пути. Моя следующая работа была связана с разработкой программного обеспечения

щя крупного банка. Мы использовали Java и JavaScript и применили на фактике подход с использованием скрытых фреймов. Группа, которой я ру-ювводил, поддерживала более 1,5 Мбайт JavaScript-кода, как расположенного t статических документах, так и генерируемого JSP. Он используется при вы-юлнении банковских операций, суммы которых составляют миллионы долла-)ов. Среди читателей этой книги, быть может, есть те, счета которых управ-[яются данной программой.

Тем временем JavaScript продолжал развиваться. В феврале 2005 года Джеймс Гаррет (James Garrett) нашел "недостающее звено". Он предложил :ороткое и запоминающееся имя Ajax для инфраструктуры, объединяющей огатых клиентов, которые взаимодействуют с серверами в асинхронном ре-жиме, и DHTML. Сочетание нескольких известных технологий создало усло-ия для программных решений, которые не были возможны ранее.

Инфраструктура Ajax привлекла к себе внимание многих специалистов; ыли созданы Prototype, Rico, Dojo, qooxdoo, Sarissa и многие другие библио-еки. Мы попытаемся кратко проанализировать их в приложении В. Работая ними, я получил массу удовольствия.

Ajax нельзя рассматривать как нечто завершенное. Данное направление родолжает развиваться. Через четыре месяца после написания первой главы астоящей книги мне пришлось существенно изменить ее. Несомненно, сле-ующие несколько лет порадуют нас новыми разработками в этой области. [очень рад, что Эрик и Даррен разделили со мной труд и радость написания гой книги.

Надеюсь, что вы, читатель, присоединитесь к нам в путешествии по увле-ательной стране, название которой — Ajax.

Дейв Крейн

Несколько слов о книге

Сейчас, на момент написания данной книги, наблюдается бурный рост популярности Ajax, и мы рады, что можем рассказать вам подробнее об этой инфраструктуре. Однако, попытавшись сделать это, мы столкнулись с интересной проблемой. Несмотря на свою популярность, ни одна из технологий, составляющих Ajax, не нова. Да и Ajax в целом — не технология, а скорее сумма технологий.

Действительно, это так. В рамках Ajax собраны давно известные Web-технологии, и их совместное использование позволило получить новые результаты. Изучать совершенно новую технологию в некотором смысле проще, поскольку мы начинаем с чистого листа. С Ajax дело обстоит по-другому: нам надо научиться видеть давно известное с иной точки зрения. Специфика Ajax обусловила структуру данной книги. Несмотря на то что технологии, составляющие Ajax, относятся к клиентской части, различия между Ajax и классическим Web-приложением затрагивают также серверные программы.

Данная книга посвящена в основном созданию кода, выполняющегося на стороне клиента, поэтому бльшая часть примеров написана на языке JavaScript. Применение принципов Ajax позволяет разделить клиентскую и серверную часть приложения, поэтому для написания программ, выполняющихся на стороне сервера, может использоваться любой язык. Таким образом, данная книга будет полезна разработчикам независимо от того, применяют ли они для серверных программ PHP, Java, C# или Visual Basic. Кроме того, разрабатывая примеры, мы старались добиться относительной простоты серверного кода, поэтому вы можете без труда переносить их в требуемую

среду. Если же в примере встречаются решения, зависящие от конкретного языка, мы подробно объясняем их, чтобы разработчик, незнакомый с данной средой, мог понять их.

На кого рассчитана книга

В рамках Ajax объединено несколько дисциплин, поэтому читатели могут прийти к пониманию данной инфраструктуры различными путями. Часть предполагаемой аудитории — это профессиональные разработчики корпоративных систем, имеющие ученые степени в области компьютерных наук, за плечами которых годы продуктивной работы над большими программными проектами. Их кругозор позволяет выйти за рамки уровня представления классического Web-приложения. Другую группу читателей составляют Web-дизайнеры, которые освоили "новую среду", изучив такие языки, как PHP, Visual Basic, JavaScript и ActionScript. Нельзя также забывать разработчиков приложений, перешедших от настольных систем к Web, и системных администраторов, которых в основном интересуют инструментальные средства управления на базе Web.

Все эти категории читателей объединяет неподдельный интерес к Ajax. При написании книги мы попытались в той или иной мере учесть интересы каждой из этих категорий. Для тех, кто рассматривает Web-браузер как низкоуровневый терминал, мы предоставили информацию об основных технологиях Web. Для тех, кто готов следовать сложившемуся стилю программирования, мы предложили основные сведения о разработке и организации программного кода. Чем бы вы ни занимались ранее, вы должны знать, что Ajax — это объединение различных технологий и, изучая данную инфраструктуру, вам неизбежно придется столкнуться с новыми для себя вопросами. Мы призываем вас расширить свой кругозор и повысить квалификацию. Мы сами делаем это постоянно; занимались этим мы и при написании данной книги. Затраченные усилия окупают себя, а результаты проявляются в различных областях нашей профессиональной деятельности.

Структура книги

Книга разделена на четыре части. В части I вы узнаете, что такое Ajax и почему вам стоит взять эту инфраструктуру в свой арсенал средств и приемов разработки. Здесь же мы расскажем об инструментах, которые упростят работу и повысят вероятность успеха. Часть II посвящена базовым технологиям, составляющим Ajax, а в части III вы узнаете, как перейти от формулировки основных понятий к созданию реального программного обеспечения. В части IV мы перейдем к конкретным примерам: поэтапно рассмотрим особенности работы над пятью проектами Ajax. Затем мы реструктуризируем код и выделим компоненты, которые вы сможете использовать в собственных Web-приложениях.

Еще раз подчеркнем, что Ajax — это не технология, а скорее процесс. Поэтому в главе 1 мы постарались рассказать разработчикам, привыкшим к традиционным подходам к созданию Web-программ, о том, что им при-

дется столкнуться с совершенно новой методологией написания приложений. Мы рассмотрели основные различия между Ajax и классическими Web-приложениями, уделили внимание практичности программ и обсудили ряд других понятий. Если вы хотите с самого начала составить общее представление о том, что же такое Ajax, мы советуем вам начать с этой главы. Если же вы выступаете в роли "потребителя кода", вам стоит сразу перейти к главе 2.

Технологии, составляющие Ajax, давно известны, поэтому они хорошо документированы. Краткий их обзор, сопровождаемый примерами, приведен в главе 2. Мы излагаем лишь общие сведения о данных технологиях и отнюдь не претендуем на полноту изложения. Мы лишь отмечаем особенности их использования в рамках Ajax и специфику получаемых результатов.

Глава 3 посвящена одной из основных тем данной книги — управлению кодом Ajax. Принимая во внимание тот факт, что размер JavaScript-кода может превышать 1,5 Мбайт, трудно не согласиться с тем, что создание Ajax-приложения принципиально отличается от написания сценария для обычной Web-страницы. В этой главе мы обсудим образы разработки и реструктуризацию. Эти вопросы освещаются не столько потому, что они важны сами по себе, сколько потому, что они применяются при работе практически над любым приложением Ajax. Мы уверены, что вы также возьмете их на вооружение.

В главах 4 и 5 мы обратим ваше внимание на базовые компоненты Ajax и на особенности практического применения некоторых образов разработки. В главе 4 речь пойдет о создании простого для восприятия клиентского кода. Основным инструментом будет выбрана известная архитектура "модель-представление-контроллер".

В главе 5 мы рассмотрим различные способы взаимодействия клиента и сервера и их соответствие требованиям инфраструктуры Ajax. В результате вы получите полное представление о работе всех составных частей Ajax-приложения.

В главах 6-8 мы поговорим о том, что принято называть "доводкой" приложения. Вы узнаете, как обеспечить безопасность и практичность вашего приложения, работающего в реальных условиях. В главе б речь пойдет о взаимодействии с пользователем, в частности, о том, каким образом следует информировать его о задачах, выполняющихся асинхронно. Необходимо решить, следует ли предоставлять пользователю полную информацию о происходящем или держать его в неведении. В большинстве случаев приходится искать компромиссное решение, и мы покажем вам, как найти его.

В главе 7 с разных точек зрения рассматривается вопрос безопасности Ajax-приложений. Технологии, составляющие Ajax, имеют непосредственное отношение к Web, и многие из проблем, связанных с Ajax, типичны для любых других Web-приложений. Мы вкратце рассмотрим основные вопросы защиты и сосредоточим внимание на проблемах, специфических для Ajax. В частности, мы обсудим обеспечение безопасности при получении с сервера JavaScript-сценария и его выполнении, а также защиту точек входа Web-служб от несанкционированного обращения. В серьезных приложениях вопросы безопасности крайне важны, и мы предпримем основные меры для контроля над ними.

В главе 8 обсуждается еще одна важная характеристика приложения — фоизводительность. Недостаточное быстродействие или неоправданно большое потребление памяти может стать причиной того, что пользователь отсажется работать с данным приложением и предпочтет продукт, предлагаемый конкурентом. Мы покажем вам, как контролировать производительюсть программы и как анализировать код, улучшить его и обеспечить согласованность вносимых изменений в рамках приложения.

В часть IV входят главы 9-13. В них мы рассмотрим несколько проектов Yjax. В каждом случае мы сначала обеспечим функционирование програм-.IBI, а затем реструктуризируем ее так, чтобы вы могли включить ее элемен-:ы в свой проект, написав лишь несколько строк кода. Вы ознакомитесь на фактике с основными принципами реструктуризации и оцените преимущества кода, пригодного для повторного использования.

В главе 9 мы обсудим способы расширения возможностей HTML-форм средствами Ajax. В частности, мы используем данные, введенные в поле, для 5аполнения раскрывающегося списка. Чтобы это стало возможным, серверу тередается асинхронный запрос.

Обсуждение вопросов модификации форм мы продолжим в главе 10. 3 ней мы рассмотрим опережающий ввод, т.е. загрузку информации с сервера з ответ на ввод пользователем частичных данных.

В главе 11 речь снова пойдет о расширенных возможностях пользовательских интерфейсов Ajax. Мы разработаем портал, который, с точки зрения юльзователя, больше напоминает интерфейс рабочей станции, чем обычную vVeb-страницу. В частности, пользователю будут предоставлены окна, котозые можно не только перемещать, но и изменять их размеры. В асинхронном эежиме выполняются действия по поддержке перемещения окон, в результате, завершив сеанс взаимодействия и зарегистрировавшись с другой маши-1ы, вы увидите, что внешний вид интерфейса остался таким же, как в конце 1редыдущего сеанса.

В главе 12 будет разработана поисковая система на базе Ajax, которая демонстрирует возможности XSLT по преобразованию XML-информации з форматированные данные.

В главе 13 мы представим Ajax-клиент без компонентов, выполняющих-:я в фоновом режиме. Он по-прежнему взаимодействует с программами на стороне сервера, но в данном случае делает это непосредственно, используя стандартный протокол RSS.

Заканчивается книга тремя приложениями, которые, как мы надеемся, принесут вам пользу. В основном тексте книги рассматривались лишь технологии. Разработчику, обратившемуся к инфраструктуре, подобной Ajax, гораздо труднее подобрать подходящий набор инструментов, чем специалисту, использующему одну законченную технологию, например J2EE или .NET. Инструментальные средства Ajax еще не появились, но мы уверены, что производители скоро предложат их. Пока это не случилось, при работе над проектами Ajax можно применять инструменты и программные решения, описание которых мы приводим в приложении А.

Приложение Б ориентировано на разработчиков корпоративных систем, которые разбираются в программах и принципах их разработки, но не очень уверенно чувствуют себя, применяя стандартные подходы при работе с гибким и неструктурированным и, скажем прямо, странным языком JavaScript. Мы рассмотрим возможности языка и покажем, в чем состоят его основные отличия от Java и C#.

Похоже, что с Ajax еще не совсем освоились не только поставщики инструментальных средств, но и разработчики библиотек. При их создании надо реализовать (а в некоторых случаях и изобрести) совершенно новые решения. В приложении В описаны известные на сегодняшний день библиотеки, приведены краткие описания и ссылки на каждую из них.

Соглашения о представлении кода

Исходный код программ представлен моноширинным шрифтом. Это позволяет выделить его на фоне обычного текста. При написании примеров для данной книги применялись JavaScript, HTML, CSS, XML, Java, C#, Visual Basic .KET и PHP, однако для всех языков используется одинаковый подход. Имена методов и функций, свойства объектов, XML-элементы и атрибуты набраны одним и тем же шрифтом.

Во многих случаях мы изменяли формат исходного кода: добавляли пустые строки и отступы. В ряде случаев некоторые позиции кода пронумерованы. Это делалось для того, чтобы конкретизировать описание кода в тексте.

Коды примеров

Исходные коды примеров, приведенных в данной книге, можно скопировать, обратившись по адресу http://www.manning.com/crane. Их также можно найти на сервере http://www.williamspublishing.com.

Создавая примеры, мы отдавали себе отчет в том, что далеко не у всех читателей установлены сервер .NET, сервер приложений J2EE. Linux, Apache, MySQL, PHP/Python/Perl (LAMP), и в том, что большинство читателей интересуются только клиентскими технологиями. Поэтому мы старались создавать примитивные варианты кода, которые могли бы работать с фиктивными данными, размещенными на сервере любого типа: Apache, Tomcat или IIS. Наряду с упрощенными приводятся также реальные рабочие примеры, поэтому при желании вы можете "померяться силами" с базой данных или сервером приложений. Некоторые документы, содержащие сведения об установке основных продуктов, предлагаются для копирования вместе с кодом.

От издательства

Вы, читатель этой книги, и есть главный ее критик и комментатор. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно эыло сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересно услышать и любые другие замечания, которые вам хотелось бы высказать в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо либо просто посетить наш Web-сервер и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится или нет вам эта книга, а также выскажите свое мнение о том. как сделать наши книги более интересными для вас.

Посылая письмо или сообщение, не забудьте указать название книги и ее авторов, а также ваш обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию последующих книг. Наши координаты:

E-mail: info@williamspublishing.com
WWW: http: //www.williamspublishing.com

Информация для писем:

из России: 115419, Москва, а/я 783
• из Украины: 03150, Киев, а/я 152

Часть I

Новый взгляд на Web-приложение

В этой части описаны основные понятия, лежащие в основе Ajax, В главе 1 рассказано о недостатках классических Web-приложений и новом подходе к их разработке. Глава 2 посвящена технологиям, составляющим Ajax, и взаимосвязи между ними. Эта информация поможет вам лучше усвоить дальнейший материал и научиться писать приложения, более сложные, чем привычная всем программа Hello, World. В главе 3 представлены инструменты разработки программ и управления проектами и рассказывается, как применить их при работе над Ajax-приложениями.

1 Каким должен быть Web-интерфейс

В этой главе...

- Асинхронное сетевое взаимодействие и образы использования
- Основные различия между Ajax и классическими Web-приложениями
- Основные принципы Ajax
- Ajax в реальном мире

Слово "богатый" означает обилие возможностей, заложенных в модел взаимодействия. Эта модель предполагает поддержку разнообразных спосс бов ввода данных и обеспечение своевременного и интуитивно понятного от вета. Выяснить, является ли взаимодействие богатым, можно лишь путе! сравнения интерфейсов. В качестве эталона можно выбрать некоторые хоре шо зарекомендовавшие себя приложения для настольных систем, наприме] текстовые процессоры или электронные таблицы. Рассмотрим, что делае пользователь, работая с приложением.

1.1.1. Действия пользователя при работе с приложением

Оторвитесь на минутку от книги, выберите любое приложение (только н Web-браузер) и попытайтесь оценить, какие варианты взаимодействия с поль зователем оно обеспечивает. Мы выбрали для этой цели электронные таб лицы, но вы можете использовать другое приложение, например текстовьи процессор.

Готово? Тогда давайте обсудим вновь приобретенный опыт. Попытавшие] ввести несколько простых формул, мы обнаружили, что сделать это можн< различными способами, например, редактировать текст в строке, активизиро вать пункты меню либо изменять данные, перетаскивая их с помощью мыши

В процессе работы осуществляется обратная связь, т.е. приложение обес печивает отклик на действия пользователя. Вид курсора мыши изменяется кнопки подсвечиваются, цвет выбранного текста изменяется, внешний *вщ* активизированных окон отличается от неактивных и т.д. (рис. 1.1). Именш такое поведение характерно для богатого взаимодействия. Так что же, про грамма поддержки электронных таблиц является богатым клиентом? Позво лим себе не согласиться с этим утверждением.

В электронных таблицах и других подобных приложениях модель дан ных и логика их обработки функционируют в закрытой среде. Они могут бе: ограничений обращаться друг к другу, но внешний мир недоступен для них (рис. 1.2). Клиент же — это программа, которая взаимодействует с различ ными не зависящими один от другого процессами, выполняемыми обычно на стороне сервера. Традиционно сервер — это более масштабное приложение чем клиент. Обычно он обеспечивает хранение больших объемов информации и управление ими. Клиент позволяет конечному пользователю просматриват! и модифицировать данные. Если с сервером одновременно взаимодействуют несколько клиентов, он обеспечивает разделение данных. На рис. 1.3 условие показана архитектура клиент/сервер.

В современной n-связной архитектуре сервер взаимодействует с другими серверными программами, которые ориентированы, например, на поддержку базы данных или на выполнение специальных вычислений. Таким образом можно выделить средства промежуточного уровня, которые могут работать и как клиент, и как сервер. Наши Ajax-приложения занимают в этой цепочке крайнюю позицию, т.е. являются клиентами. Поэтому в ходе дальнейшего обсуждения мы будем рассматривать все остальные компоненты системы как единый "черный ящик", выполняющий функции сервера.

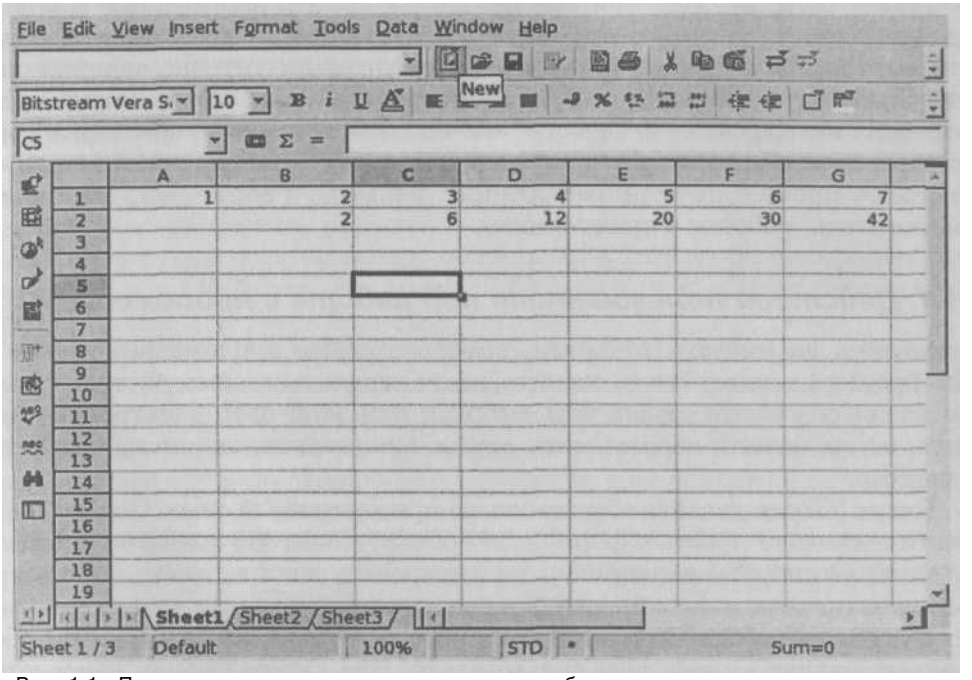


Рис. 1.1. Приложение, управляющее электронными таблицами, иллюстрирует различные варианты взаимодействия с пользователем. Заголовки строк и столбцов подсвечиваются, внешний вид кнопки изменяется при помещении на нее курсора мыши, на панели инструментов отображаются различные пиктограммы, а ячейки таблицы позволяют редактировать содержащиеся в них данные

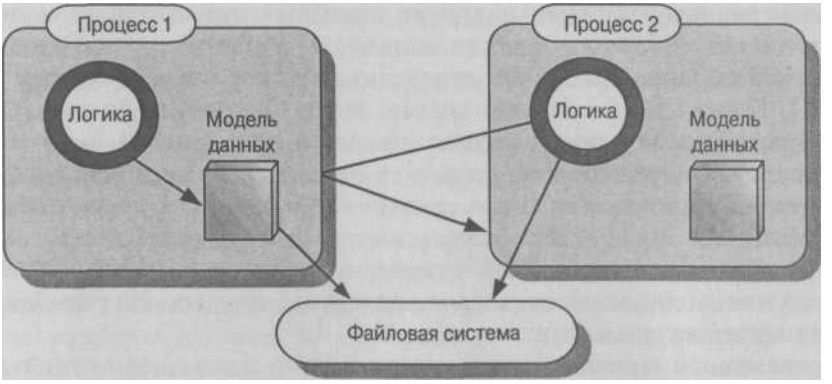


Рис. 1.2. Архитектура приложения, предназначенного для выполнения в рамках настольной системы. Прикладная программа представляет собой отдельный процесс, в пределах которого модель данных и логика их обработки могут взаимодействовать друг с другом. Если на компьютере одновременно запущены два приложения, то ни одно из них не имеет доступа к модели данных другого. Взаимодействовать друг с другом они могут только посредством файловой системы. Обычно состояние программы определяется содержимым некоторого файла. На время работы этот файл блокируется, не позволяя приложениям обмениваться информацией о состоянии

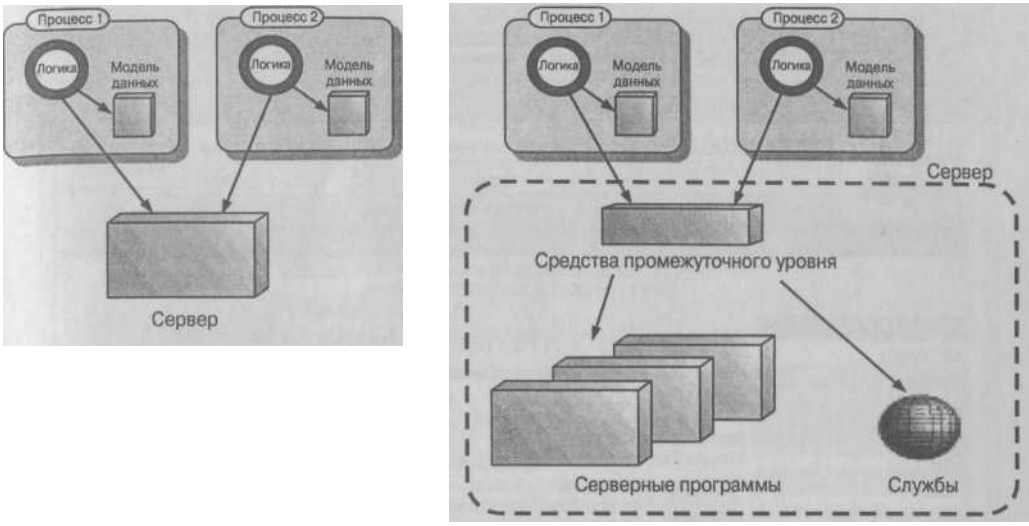


Рис. 1.3. Системы клиент/сервер в составе n-связной архитектуры. Сервер предоставляет разделяемую модель данных, с которой могут взаимодействовать клиенты. Для быстрого доступа каждый клиент поддерживает собственную частичную модель данных, которая синхронизирована с моделью, находящейся на сервере. Модель на стороне клиента по сути является альтернативным представлением бизнес-объектов. С сервером могут одновременно взаимодействовать несколько клиентов, при этом осуществляется избирательная блокировка. На время работы сданными одного клиента другим запрещается доступ к некоторым объектам или записям базы данных. Сервер может быть реализован как единый процесс (такой подход преимущественно применялся до середины 1990-х годов) или включать несколько промежуточных уровней, поддерживать различные Web-службы и т.д. В любом случае с точки зрения клиента сервер предоставляет единую точку входа и может рассматриваться как "черный ящик"

Программа, поддерживающая электронные таблицы, работает с данными, которые расположены в памяти локальной машины и в локальной файловой системе. Если система грамотно разработана, взаимосвязь между данными и средствами их отображения минимальна, но, несмотря на это, их нельзя разнести на разные узлы сети, а также невозможно организовать совместное использование информации. Таким образом, подобные программы нельзя считать клиентами.

Web-браузер несомненно является клиентом. Обращаясь к Web-серверу, браузер запрашивает у него тот или иной документ. Как правило, браузеры реализуют обширный набор функций, обеспечивающих просмотр Web-страниц. Например, пользователю доступны кнопка для возврата к предыдущему документу, список предыстории и фреймы, позволяющие отображать несколько документов. Однако мы считаем приложением набор страниц, расположенных на конкретном Web-узле, поэтому универсальные средства, предоставляемые браузером, связаны с приложением не больше, чем кнопка Start (пуск) в системе Windows, посредством которой мы раскрываем меню и запускаем программу поддержки электронных таблиц.

Рассмотрим современное Web-приложение. В качестве примера выбран узел Amazon (рис. 1.4), так как он знаком практически всем. Мы обратились к узлу Amazon посредством браузера. Поскольку сервер имеет информацию

Рис. 1.4. Исходная страница Amazon.com. Система имеет информацию о предыдущем визите, поэтому для навигации предложены различные ссылки: как общие для всех, так и ориентированные на конкретного пользователя

о нашем прошлом визите, мы видим приветствие, список рекомендованных книг и сведения о предыдущих покупках.

По щелчку на одном из заголовков в списке рекомендаций осуществляется переход на другую страницу (при этом мы теряем из виду все пункты списка, которые отображались на экране несколькими секундами раньше). На ней также представлена разнообразная информация: обзоры, ссылки, посредством которых можно получить информацию об авторах, и названия книг, выбранных ранее (рис. 1.5).

Короче говоря, нам предоставлена разнообразная информация, элементы которой тесно связаны между собой. Чтобы получить доступ к этой информации, следует щелкать на гипертекстовых ссылках и заполнять формы. Если в процессе просмотра узла Amazon вы вздремнете за клавиатурой, то не узнаете, вышла ли новая книга о Гарри Поттере, до тех пор, пока не обновите Web-страницу в окне браузера. К сожалению, мы не можем перенести список рекомендованных книг с одной страницы на другую и изменить порядок следования текстовых фрагментов.

В этом нельзя обвинять специалистов Amazon, так как они проделали большую работу. Но по сравнению с электронными таблицами модель взаимодействия, на которой основаны данные документы, безусловно ограничена.

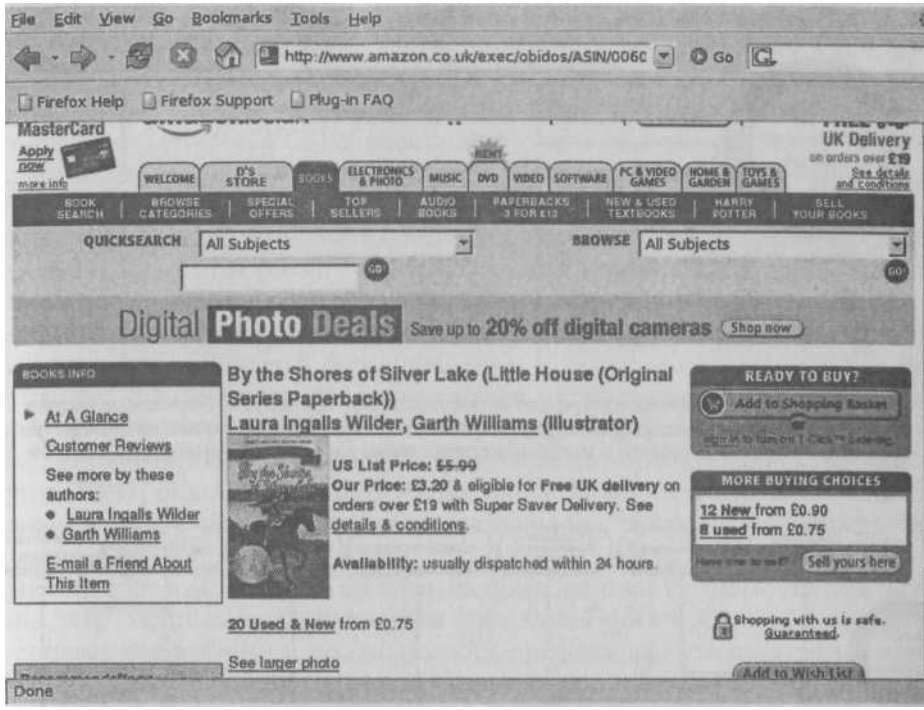


Рис. 1.5. Подробная информация о книге, предоставляемая Amazon.com. На странице имеются как универсальные ссылки, так и ссылки, предназначенные для конкретного пользователя. Здесь можно найти многие элементы, которые были показаны на рис. 1.4. Для того чтобы пользователь мог с помощью Web-браузера выполнять необходимые операции с документом, эти элементы должны отображаться на каждой странице

Почему же подобные ограничения присущи всем современным Web-приложениям? Они обусловлены техническими причинами, которые мы сейчас рассмотрим.

1.1.2. Накладные расходы при работе в сети

С наступлением эпохи Интернета появилась возможность связать все компьютеры в мире так, чтобы они формировали единый ресурс чрезвычайно большого объема. Обращения к удаленным процедурам теперь производятся так же, как и к локальным. Программа, вызывающая функцию, может даже не знать, на каком из компьютеров находится реализующий ее код.

Однако вызов удаленных и локальных процедур — это не одно и то же. Взаимодействие по сети связано с накладными расходами, а порой оно оказывается медленным и ненадежным. При выполнении скомпилированной программы или при интерпретации исходного текста без обращения к сети коды методов и функций располагаются в той же локальной памяти, что и данные, которые они обрабатывают (рис. 1.6). При передаче данных методу и получении результатов трудности не возникают.

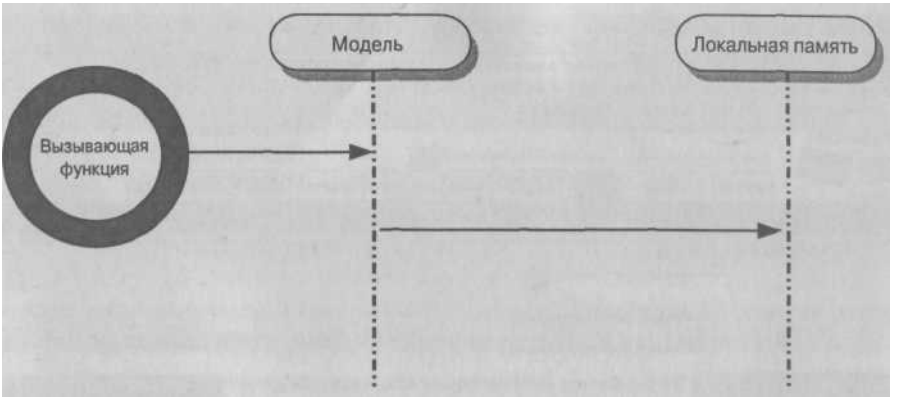


Рис. 1.6. Диаграмма, иллюстрирующая вызов локальной процедуры. В процессе вызова участвует небольшое число элементов, поскольку модель данных и логика их обработки хранятся в локальной памяти и могут непосредственно взаимодействовать друг с другом

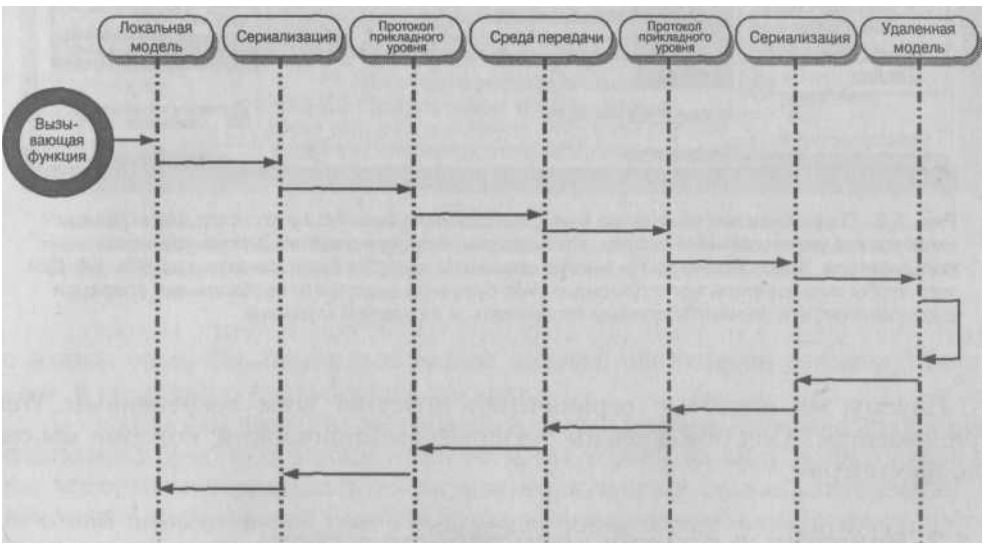


Рис. 1.7. Диаграмма, иллюстрирующая вызов удаленной процедуры. Программная логика на одном компьютере обращается к модели данных на другом компьютере

При работе в сети на обоих концах соединения выполняется большой бъем работы, скрытой от пользователя (рис. 1.7). Порой эти вычисления амедляют работу программы даже больше, чем пересылка данных по лини-м связи. Процесс сетевого взаимодействия включает в себя решение самых азнообразных задач: передачу электрических сигналов по проводам (или пе-есылку радиосигналов через спутник), представление этих сигналов в виде .воичных данных, проверку на наличие ошибок и организацию повторной ередачи, восстановление битовой последовательности и, наконец, интерпре-ацию двоичной информации.

Запрос, который исходит от вызывающей функции, должен быть представлен в виде объекта, который затем подвергается сериализации (т.е. преобразуется в последовательный набор байтов). Сериализованные данные затем передаются реализации протокола прикладного уровня (в современных условиях это чаще всего HTTP) и пересылаются по линиям связи (по проводам, оптоволоконным каналам или с помощью беспроводных соединений).

На удаленном компьютере данные, полученные посредством протокола прикладного уровня, десериализуются, в результате чего формируется копия объекта запроса. Этот объект передается модели данных, которая генерирует объект ответа. Для передачи ответа вызывающей функции снова задействуются механизм сериализации и транспортные средства; в результате объект доставляется вызывающей функции.

Процесс такого взаимодействия сложен, но его можно автоматизировать. Современные инструменты программирования, например Java или Microsoft .NET Framework, обеспечивают это. Тем не менее при удаленном вызове процедуры (RPC) реально выполняется большой объем вычислений, и если такие обращения будут производиться слишком часто, производительность системы снизится.

Таким образом, вызовы по сети никогда не смогут быть так же эффективны, как обращения к локальным методам. Более того, из-за ненадежности сетевых соединений и необходимости организовывать повторную передачу информационных пакетов трудно предсказать, насколько медленнее будет обработан удаленный запрос по сравнению с локальным обращением. Шина локального компьютера не только более надежна, но и позволяет легко определить время выполнения функции.

Но какое отношение это имеет к работе пользователя, которому предоставляется лишь интерфейс программы? Оказывается, очень большое.

Хороший интерфейс должен в чем-то имитировать реальный мир. А в большинстве случаев на действия в реальном мире следует немедленный отклик. Даже небольшая задержка между действием и реакцией на него сбивает с толку пользователя, отвлекает его от решаемой задачи и заставляет переключить внимание на особенности самого интерфейса.

Дополнительные вычисления, связанные с передачей по сети, часто так замедляют работу, что задержка становится заметной. Для обычных приложений, предназначенных для выполнения на настольных системах, малая практичность является следствием неправильных решений, принятых при разработке. В случае сетевого соединения мы получаем тот же результат автоматически.

Из-за того, что величина задержки, связанная с передачей по сети, непредсказуема, время отклика может быть то больше, то меньше, и, следовательно, становится труднее оценить качество приложения. Итак, задержка при передаче по сети является основной причиной плохой интерактивности современных приложений.

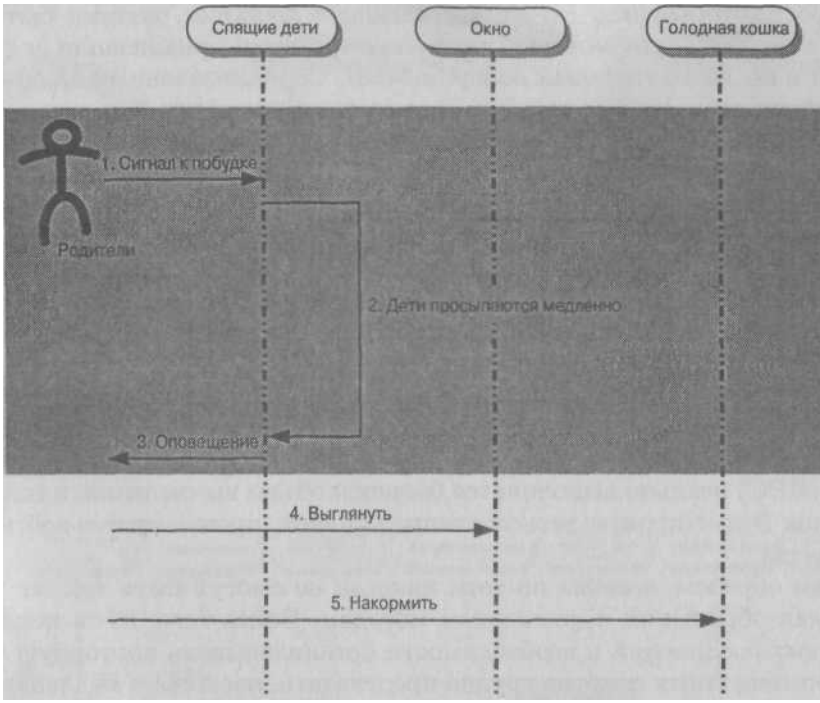


Рис. 1.8. Диаграмма, поясняющая синхронный ответ на действия пользователя. В качестве примера выбрана утренняя побудка детей. Время на диаграмме откладывается по вертикальной оси. Серым цветом выделен период, в течение которого деятельность родителей приостанавливается

1.1.3. Асинхронное взаимодействие

Решить проблему задержки, связанной с передачей данных по сети, разработчик может единственным образом — предположить наихудшее развитие событий. На практике это означает, что надо создавать пользовательский интерфейс так, чтобы он не зависел от сетевого взаимодействия. К счастью, в некоторых случаях проблемы, вызванные задержкой отклика, можно сделать менее острыми. Рассмотрим пример из реального мира. Всем родителям приходится отправлять по утрам детей в школу. Можно стать над ними и дожидаться, пока они не встанут из постелей и не оденутся. На это уходит много времени (рис. 1.8).

Необходимо будить детей, подавив желание выглянуть в окно и игнорируя мяуканье кошки. Если дети попросили есть — это сигнал о том, что они окончательно проснулись. Подобно процессу, выполняемому на сервере, дети пробуждаются медленно. Если родитель будет следовать синхронной модели взаимодействия, он затратит много времени на ожидание. Но в ожидании заветных слов "Все, мы проснулись, где завтрак?" можно заняться другими делами.

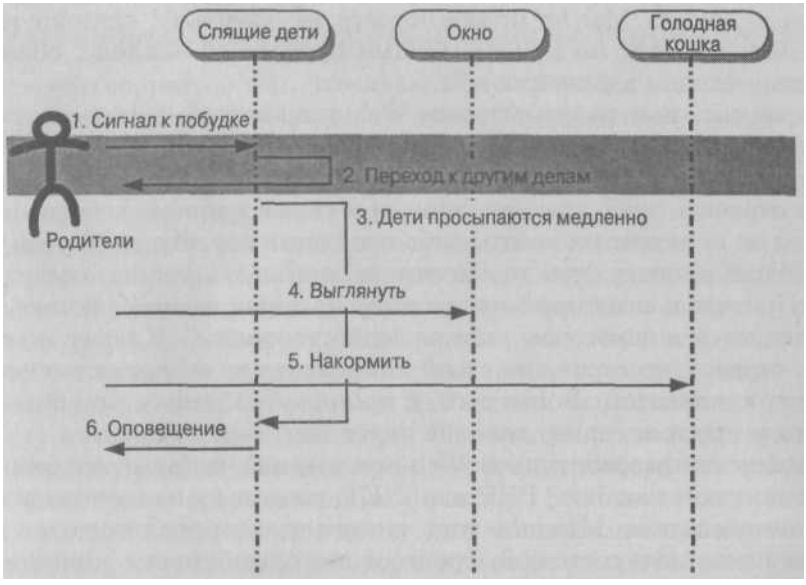


Рис. 1.9. Диаграмма, иллюстрирующая асинхронный ответ на действия пользователя. Следуя асинхронной модели ввода, родитель, подав сигнал побудки, предоставляет детям просыпаться самостоятельно и оповестить его об окончании этого процесса. В это время он продолжает заниматься другими делами; при этом время, непосредственно потраченное на побудку детей, существенно сокращается

Возвращаясь к компьютерной тематике, сказанное означает выполнение действий в асинхронном режиме. Для этой цели используется отдельный поток. Снова пользуясь примером из реального мира, можно сказать, что дети будут просыпаться сами "в своем потоке", а родителю не придется синхронизировать с ними свои действия до тех пор, пока он не узнает, что дети проснулись (т.е. пока они на захотят завтракать). После этого родителю придется взаимодействовать с детьми, так как, получив уведомление, он узнает, что дети проснулись и оделись (рис. 1.9).

Любой достаточно квалифицированный разработчик пользовательского интерфейса знает, что длительные вычисления лучше производить в отдельном потоке в фоновом режиме; в это время пользователь может выполнять другие действия. Конечно, на определенное время, необходимое для запуска потока, пользовательский интерфейс будет заблокирован, но это время очень мало. Учитывая неизбежную задержку при сетевом взаимодействии, имеет смысл априори считать любой вызов удаленной процедуры длительной операцией и выполнять ее асинхронно.

Проблема четко сформулирована, и решение ее известно. Задержки, связанные с обменом по сети, играли роль в ранних приложениях клиент/сервер. Созданные неопытными разработчиками, клиенты "зависали" на неопределенное время при попытке доступа к серверу, занимавшемуся обработкой большого числа запросов. Теперь, в эпоху Интернета, те же причины заставляют браузер временно останавливаться при переходе от одной Web-

страницы к другой. Мы не можем влиять на задержку, связанную с взаимодействием по сети, но можем минимизировать ее влияние, обращаясь к удаленным методам в асинхронном режиме.

К несчастью, нам, разработчикам Web-приложений, не так-то просто следовать этим давно известным рекомендациям. Протокол HTTP действует по принципу "запрос-ответ". Клиент передает запрос на получение документа, а сервер отвечает, либо предоставляя требуемые данные, либо информируя о том, что он не может их найти, либо предлагая перейти по другому адресу, либо сообщая клиенту, что тот может использовать копию, содержащуюся в кэше. Протокол, в котором предусмотрено лишь взаимодействие "запрос-ответ", является в некотором смысле "односторонним". Клиент может обратиться к серверу, но сервер по своей инициативе не может установить взаимодействие с клиентом. Более того, в промежутке между двумя последовательными запросами сервер даже не имеет сведений о клиенте.

Большинство разработчиков Web-приложений используют современные технологии, такие как Java, PHP или .NET, поскольку в них поддерживается сеанс взаимодействия. Мысль о том, что неплохо бы реализовать в серверах поддержку текущего состояния процесса взаимодействия с клиентами, пришла слишком поздно. Протокол HTTP очень хорошо выполняет те задачи, для которых он был изначально разработан, но адаптировать его для решения вопросов, которые не были предусмотрены создателями данного протокола, крайне сложно. Однако при асинхронном обращении к удаленному методу клиент должен быть оповещен дважды: при порождении потока и при его завершении. Для протокола HTTP и классических Web-приложений эта задача неразрешима.

Основой классической модели Web-приложения, подобной той, которая используется на сервере Amazon, является Web-страница. Документ, отображаемый в окне браузера, содержит набор гипертекстовых ссылок и формы, позволяющие переходить к другим документам и посылать серверу данные. Такой подход позволяет взаимодействовать с базами данных большого объема. Специалисты Amazon и других компаний накопили немалый опыт создания таких приложений и показали, что их вполне можно применять в серьезных проектах.

В процессе многолетней работы с Интернетом большинство из нас привыкли к описанной модели взаимодействия и не представляют себе альтернативы. Инструменты разработки с интерфейсом WYSIWYG формируют Web-узел в виде набора Web-страниц. Переход от одной страницы, расположенной на сервере, к другой можно представить в виде диаграммы изменения состояния. Классическим Web-приложениям изначально присущи недостатки, связанные с получением отклика, и прибегнуть к асинхронной обработке не представляется возможным.

Однако компания Amazon, и не она одна, используя свой узел, успешно строит бизнес. Поэтому классические Web-приложения нельзя однозначно считать непрактичными. Чтобы понять, почему Web-страницы выполняют свои функции на сервере Amazon, а для других целей могут оказаться неприменимыми, рассмотрим различные образы использования.

1.1.4. Независимый и переходный образы использования

Бессмысленно спорить о том, что лучше: велосипед или спортивный автомобиль. Каждый из них имеет свои преимущества: уровень комфорта, скорость, потребление горючего. Немаловажен и тот факт, насколько конкретный вид транспорта соответствует вашему имиджу. Рассматривая конкретные образы использования, например, перемещение в час пик по центру города, организацию летнего отдыха большой семьи или поиск зонтика в дождь, можно сказать, что в каждом из них возможен безусловно успешный исход. То же справедливо и для пользовательских интерфейсов компьютерных программ.

Алан Купер (Alan Cooper), общепризнанный специалист в области обеспечения практичности программ, в своих работах подчеркивает важность образов использования. Он выделил две модели использования: *переходную* (transient) и *независимую* (sovereign). Приложения, соответствующие переходной модели, или *переходное приложение,* используются ежедневно, но обычно выполняют второстепенные функции. В отличие от него приложение, отвечающее независимой модели, или *независимое приложение,* в течение нескольких часов подряд завладевает всем вниманием пользователя.

Многие приложения по своей сути являются переходными или независимыми. Текстовый процессор, которым пользуется писатель, является независимым приложением. В его работу вовлечено большое количество переходных процедур, например средства управления файлами (обычно они встроены в текстовый процессор, а доступ к ним осуществляется посредством диалогового окна), словари или программа проверки орфографии (они также чаще всего встроены), а также почтовая программа, используемая для обмена сообщениями с коллегами. Для разработчика программного обеспечения текстовый редактор и отладчик, входящие в состав интегрированной системы разработки, являются независимыми программами. Независимые приложения обычно используются более интенсивно, чем переходные.

Как вы помните, хороший интерфейс должен быть незаметен для пользователя. Оценить качество работы можно следующим образом. Надо определить, насколько часто последовательность действий, направленных на решение задачи, прерывается из-за того, что пользователю приходится обращать внимание на интерфейс программы. Если при копировании файлов из одной папки в другую возникает задержка в одну-две секунды, пользователь будет чувствовать себя вполне комфортно. Если с такой же двухсекундной задержкой программа рисования будет отображать проведенные линии или отладчик будет выводить текущее значение переменной, пользователь обязательно начнет нервничать.

Web-узел Amazon представляет собой переходное приложение. К этому же типу относятся eBay, Google и большинство масштабных Web-приложений. Когда всемирная сеть стала доступна широкой публике, некоторые предрекали, что вскоре традиционные офисные пакеты уступят место решениям на базе Web. С тех пор прошло больше десяти лет, но предсказание не сбылось. Web-страницы хороши как переходные приложения, но не как независимые.

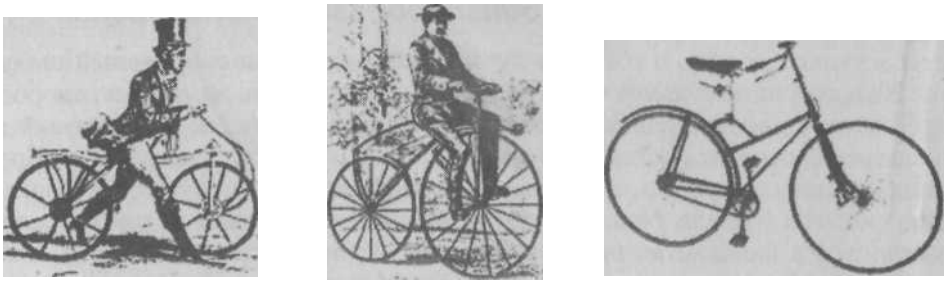


Рис. 1.10. Эволюция велосипеда

К счастью, современные Web-браузеры так же напоминают первоначальную идею клиента для работы с удаленными документами, как швейцарский армейский ноле — каменный топор первобытного человека. Стремление улучшить программы просмотра информации из Web привели к созданию средств интерактивного взаимодействия, языков сценариев и встраиваемых модулей. (Получить представление о развитии Web можно, ознакомившись с документом www.webhistory.org/www.lists/www-talk.1993ql/0182.html. Сейчас же достаточно сказать, что в 1993 году Марку Андрессену пришлось убеждать Тима Бернерса-Ли и других специалистов в том, что язык HTML лишь выиграет, если ввести в него дескриптор для поддержки изображений.)

В течение нескольких лет лишь немногие отваживались признать JavaScript серьезным языком программирования. Большинство считали его лишь средством для отображения окон с сообщениями и создания интерактивных рекламных вставок.

Ajax можно рассматривать как "реабилитационный центр" для жертв "войны браузеров" — средств, в свое время не понятых и отвергнутых. Предоставив среду для работы, мы можем вернуть JavaScript статус "полноправного члена" Интернет, способного обеспечить практичность Web-приложений, не доставляя беспокойства пользователям и не требуя замены браузеров. Добиться этого нам помогут тщательно продуманные прбстые в использовании инструменты. В качестве примера таких инструментов можно привести образы разработки, которые мы часто используем в своей работе. Мы будем периодически ссылаться на них в тексте книги.

Процесс введения новой технологии — не только технический, но и социальный. Получив в свое распоряжение технологию, пользователи должны хорошо представлять, какие преимущества она может обеспечить, и применять ее как нечто давно знакомое. Так, например, первые велосипеды назывались "лошадь для денди". Их приводили в движение, отталкиваясь ногами от земли. По мере того как эти устройства становились доступными различным слоям населения, конструкция усовершенствовалась: появились педали, тормоза, цепная передача и надувные шины. С каждым нововведением велосипед все меньше напоминал лошадь (рис. 1.10).

Нечто похожее происходит сегодня в Web. Технологии, которые лежат в основе Ajax, дают возможность превратить Web-страницы во что-то совершенно новое. В результате первых попыток применения Ajax Web-страницы были изменены так, что их уместно сравнить с переходной моделью по пути от "лошади для денди" к современному велосипеду. Но чтобы понять потенциальные возможности Ajax, нам надо отказаться от некоторых принципов создания Web-страниц, а следовательно, и от правил, которым мы следовали в течение последних лет. Лишь несколько месяцев прошло с момента появления Ajax, а процесс преобразования Web уже начался.

1.2. Четыре основных принципа Ajax

Классическая модель приложения на основе Web-страниц связана не только с используемыми базовыми средствами, но и с нашим образом мышления. Потратим несколько минут на то, чтобы выявить основные предпосылки и определить, что надо делать, чтобы получить наибольшую выгоду от использования Ajax.

1.2.1. Браузер имеет дело с приложением, а не с содержимым

Для классического приложения на базе Web-страниц браузер представляет собой лишь низкоуровневый терминал. Он не имеет информации о том, какой этап работы выполняется пользователем. На сервере содержатся минимальные сведения об этом, которые, по сути, сводятся к поддержке сеанса. Если вы работаете с Java или .NET, средства поддержки сеанса на сервере доступны, подобно запросам ответам и MIME-типам, посредством стандартного API. На рис. 1.11 показан типичный жизненный цикл классического Web-приложения.

Когда пользователь регистрируется или другим способом инициализирует сеанс, создается несколько объектов на стороне сервера. Они представляют, например, "корзинку" покупателя или платежную карточку пользователя. Одновременно браузер получает исходную страницу. Она доставляется в виде потока HTML-данных, которые представляют собой сочетание стандартных элементов и данных, специфических для конкретного пользователя.

При каждом обращении к серверу браузер получает очередную страницу, содержащую данные тех же типов, что в предыдущих документах. Браузер исправно убирает с экрана старый документ и отображает новый. Других Действий от него и не следует ожидать: это низкоуровневая программа, которая делает только то, что было предусмотрено разработчиками.

Когда пользователь активизирует ссылку, соответствующую окончанию сеанса, или закрывает браузер, выполнение приложения завершается и сеанс разрушается. Информация, которую пользователь должен увидеть при следующей регистрации, заносится в долговременное хранилище. В Ajax-приложении часть прикладной логики переносится на браузер (рис. 1.12).

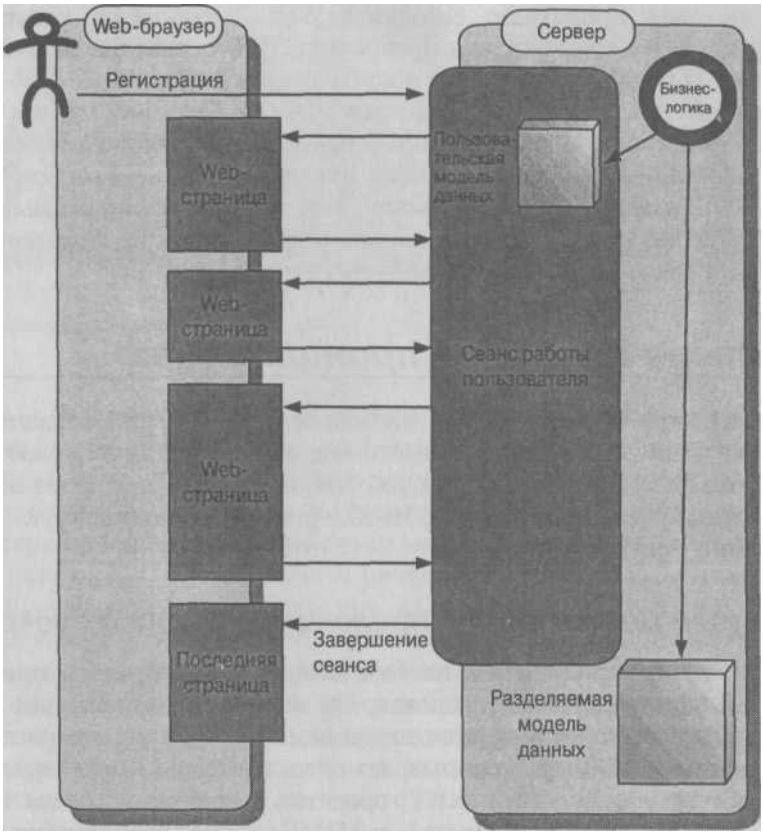


Рис. 1.11. Жизненный цикл классического Web-приложения. Сведения о текущем состоянии "диалога" пользователя с приложением хранятся на Web-сервере, пользователь же видит лишь последовательность страниц. Ни одна из них не обеспечивает продолжения диалога без обращения к серверу

После регистрации пользователя клиентское приложение доставляется браузеру. На многие действия пользователя это приложение способно реагировать самостоятельно. Если имеющихся в наличии возможностей недостает, оно передает запросы серверу, не прерывая последовательность действий пользователя.

При регистрации пользователя браузеру предоставляется более сложный документ, существенную часть которого составляет код JavaScript. Этот документ остается доступным пользователю в течение всего сеанса; при этом, в зависимости от действий пользователя, он изменяет свой внешний вид. Клиентская программа знает, как реагировать на вводимые данные, и способна решать, обрабатывать ли их самостоятельно, посылать ли запрос серверу (который в свою очередь обратится к базе данных или к другому ресурсу) или сделать и то и другое.

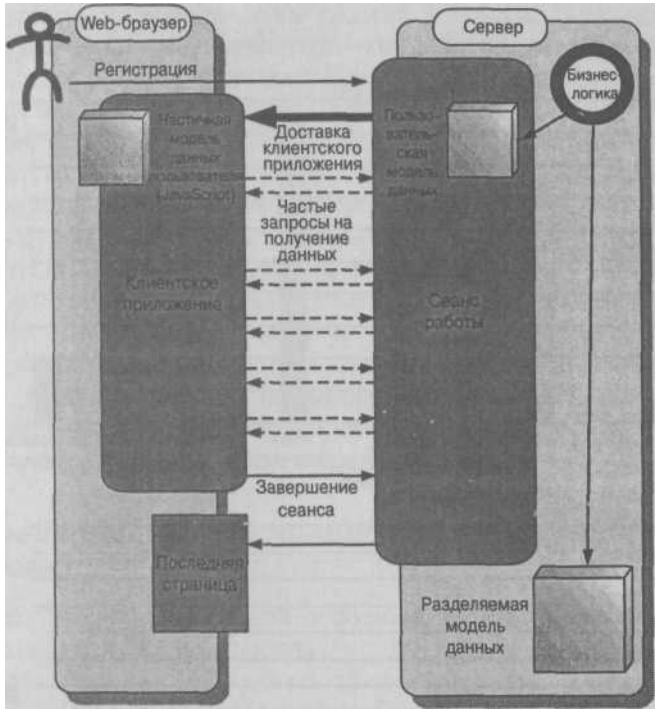


Рис. 1.12. Жизненный цикл Ajax-приложения

Поскольку документ присутствует на стороне клиента в течение всего сеанса, он способен хранить информацию о его состоянии. Например, сведения о состоянии "корзинки" покупателя могут храниться не на сервере, а в клиентской программе.

1.2.2. Сервер доставляет данные, а не содержимое

Как было сказано ранее, на каждом этапе работы классическое Web-приложение предоставляет сочетание стандартных элементов и специальных данных. Когда пользователь добавляет товар в "корзинку", приложение в ответ должно лишь изменить общую цену либо, при наличии ошибки, сообщить о ней. Как видно на рис. 1.13, код, отвечающий за эти действия, составляет незначительную часть документа.

В Ajax-приложении "корзинка" может обладать более высоким "интеллектом" и передавать серверу асинхронные запросы. Шаблон, элементы навигации и другие компоненты страницы уже присутствуют на стороне клиента, поэтому сервер должен передавать только данные, полученные в результате обработки запроса.

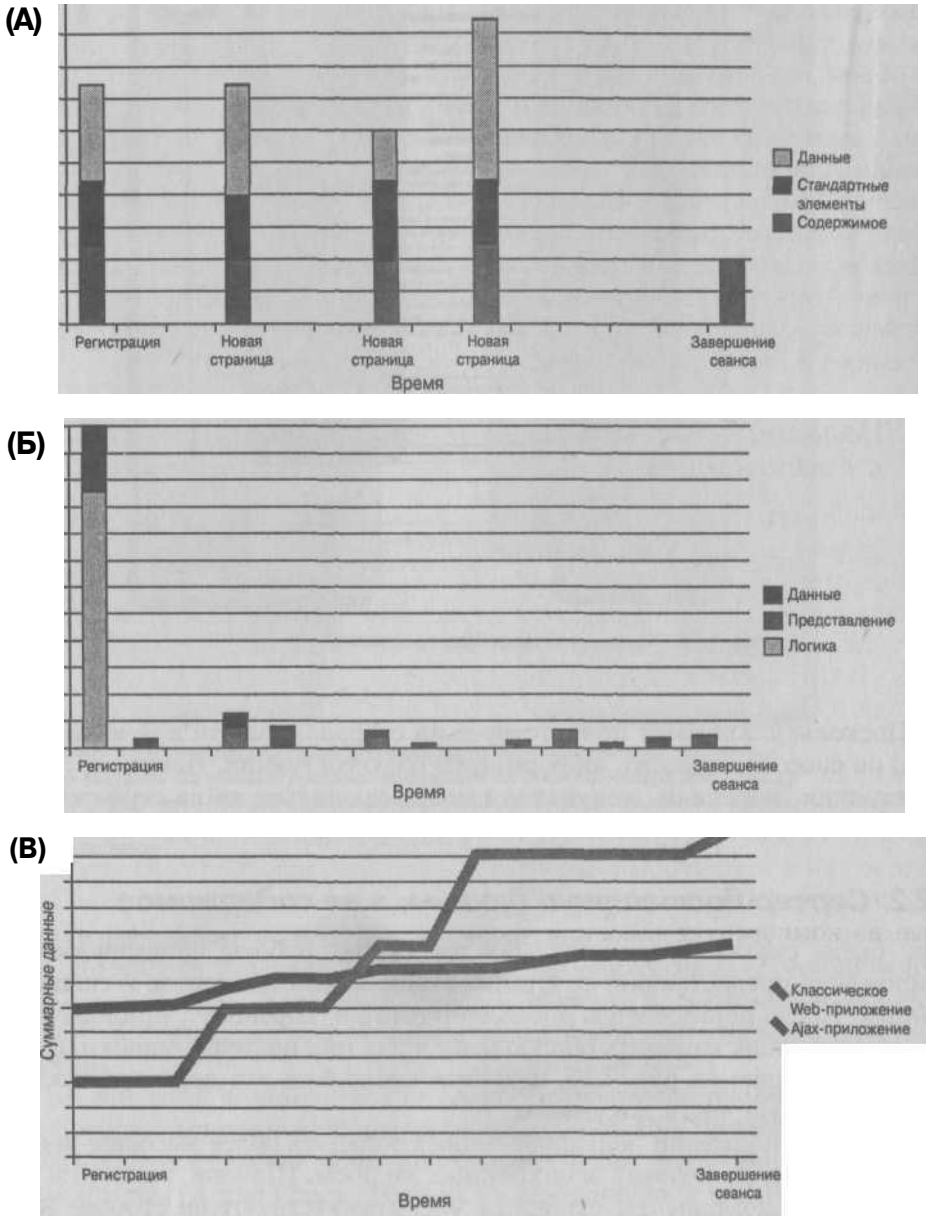


Рис. 1.13. Информация, доставляемая классическим Web-приложением (а) и Ajax-приложением (б). С увеличением длительности работы (в) суммарный трафик классического Web-приложения возрастает быстрее, чем трафик Ajax-приложения

Ajax-приложение может достичь данной цели различными способами, например, вернуть фрагмент JavaScript-кода, поток, содержащий обычный текст или небольшой XML-документ. Преимущества и недостатки каждого из этих решений мы подробно обсудим в главе 5. Сейчас же достаточно заметить, что данные в любом из этих форматов будут иметь значительно меньший объем, чем страница, возвращаемая классическим Web-приложением.

График, представляющий возрастание трафика при работе Ajax-приложения, имеет крутой фронт, объясняемый тем, что при регистрации пользователя клиентской программе доставляется сложное приложение. Дальнейшее взаимодействие с сервером становится гораздо более эффективным. Для переходного приложения, созданного на базе Web-страниц, суммарный трафик может оказаться меньше, чем для Ajax-приложения. Однако по мере увеличения времени работы пользователя ситуация меняется и Ajax-приложение становится гораздо более экономичнее своего конкурента, выполненного в рамках классического подхода.

12.3. *Пользователь может непрерывно взаимодействовать с приложением*

В Web-браузере предусмотрены два основных механизма ввода данных: гипертекстовые ссылки и HTML-формы.

Гипертекстовые ссылки могут быть сформированы на сервере и снабжены параметрами CGI (Common Gateway Interface — интерфейс общего шлюза). Их можно оформить как изображения и средствами CSS (Cascading Style Sheets — каскадные таблицы стилей) организовать обратную связь с пользователями, например, обеспечить изменение внешнего вида при наведении на них курсора мыши. Хороший Web-дизайнер при желании добьется того, что ссылки будут выглядеть как полноправные компоненты пользовательского интерфейса.

Формы могут содержать многие компоненты, типичные для пользовательского интерфейса обычных приложений, а именно: поля редактирования, флажки и переключатели опций, раскрывающиеся списки и пр. Однако некоторые из компонентов в составе форм не поддерживаются. Так, например, в формах не предусмотрены деревья и таблицы. Формы, как и гипертекстовые ссылки, содержат URL, указывающие на ресурсы сервера.

Гипертекстовые ссылки и формы могут также указывать на функции JavaScript. В традиционных Web-документах часто можно встретить JavaScript-сценарии, проверяющие корректность заполнения форм. Они следят за незаполненными полями, значениями, выходящими за пределы допустимого диапазона, и другими подобными ошибками. Передача данных на сервер происходит лишь в том случае, если форма заполнена корректно. JavaScript-функции присутствуют на стороне клиента в течение того же времени, что и содержащая их Web-страница.

При обращении к очередной странице пользователю приходится на время прервать работу. Предыдущий документ еще некоторое время отображается на экране; некоторые браузеры даже позволяют активизировать какую-либо

из видимых ссылок, но результаты предсказать невозможно. Скорее всего, пользователь, поступивший подобным образом, разрушит информацию о сеансе, поддерживаемую на сервере. После получения новой страницы пользователю предоставляются приблизительно те же возможности выбора, которые он имел до этого. Например, добавление в "корзинку" товара, скажем, брюк, вряд ли приведет к переходу от категории "мужская одежда" к категории "женская одежда" или "детская одежда".

Представим себе теперь, как ведет себя "корзинка" покупателя, реализованная в составе Ajax-приложения. Поскольку Ajax позволяет передавать данные в асинхронном режиме, пользователь может добавлять товар в "корзинку" с той же скоростью, с которой он щелкает мышью. Если клиентская часть приложения выполнена профессионально, она без труда справится с данной задачей, и пользователю не придется в своей работе учитывать специфику взаимодействия программы и сервера.

Очевидно, что реальной "корзинки" не существует; она выполнена в виде объекта поддержки сеанса на сервере. Однако пользователь, который делает покупки, не должен ничего знать об объекте сеанса. "Корзинка" — это понятие, позволяющее упрощенно представить выполняемые действия. Необходимость переходить от понятия "корзинка" к объектам, содержащимся в памяти компьютера, создает дискомфорт для пользователя. Необходимость ожидать обновления страницы принуждает вернуться из виртуального мира в реальный (рис. 1.14). Приложение, реализованное средствами Ajax, свободно от этого недостатка. Приобретать товары в сетевом магазине пользователям приходится лишь время от времени, однако в других областях деятельности, например, при решении сложных инженерных задач, поминутное прерывание работы, связанное с необходимостью ожидать появления очередной страницы, недопустимо.

Еще одно преимущество Ajax состоит в том, что данная технология позволяет обрабатывать более обширный набор событий, соответствующих действиям пользователя. Становится возможным реализовать перетаскивание объектов и другие сложные функции интерфейса, в результате чего Web-приложение становится похожим на обычную прикладную программу, выполняемую на настольной системе. С точки зрения практичности эта свобода действий важна не потому, что она больше соответствует представлениям пользователя о виртуальном мире, а потому, что позволяет лучше сочетать взаимодействие с пользователем и запросы к серверу.

Для того чтобы обратиться к серверу в классическом Web-приложении, мы должны щелкнуть на гипертекстовой ссылке либо активизировать элемент формы, а затем ожидать результата. Это неизбежно отвлекает от основной работы. Если же обращение к серверу происходит в ответ на перемещение мыши, перетаскивание объекта или нажатие клавиши, сервер работает параллельно с пользователем. Сказанное выше иллюстрирует Google Suggest (http://www.google.com/webhp?complete=1). В данном примере приложение реагирует на нажатие клавиш по мере того, как пользователь вводит информацию в поле. Клиент взаимодействует с сервером, извлекает и отображает наиболее вероятные завершения фраз. Исходной информацией при этом

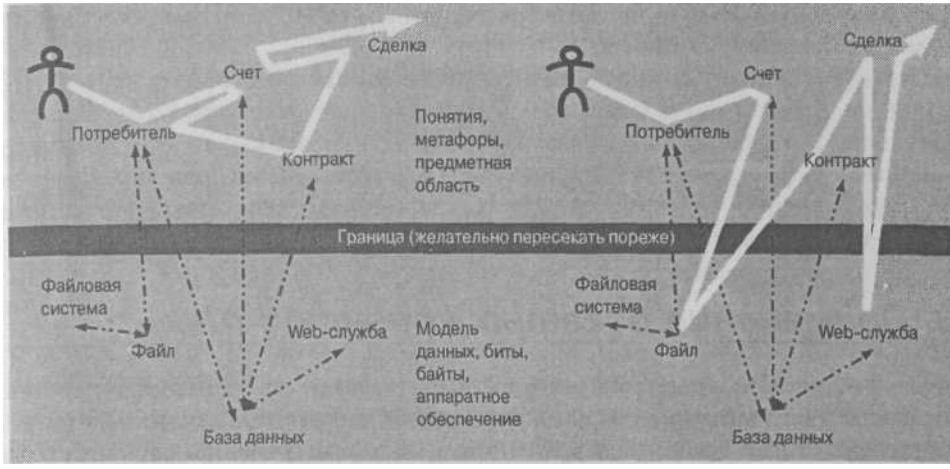


Рис. 1.14. Прерывание последовательности действий пользователя для обработки событий. Пользователь работает с двумя типами объектов: один тип непосредственно относится к выполняемой работе, а другой связан с используемой вычислительной системой. Когда пользователю приходится часто переключаться от одного типа к другому, это рассеивает его внимание, в результате производительность труда снижается

являются запросы, выполненные другими пользователями или поисковыми серверами. Упрощенный вариант подобной службы мы рассмотрим в главе 8.

1.2.4. Реальное кодирование требует порядка

В классических Web-приложениях для реализации дополнительных возможностей в документы часто включают фрагменты JavaScript-кода. Страничная модель не позволяет этим фрагментам присутствовать на стороне клиента дольше самой страницы, а это ограничивает их применимость. В результате JavaScript приобрел репутацию языка, пригодного лишь для поделок и не заслуживающего внимания серьезных разработчиков.

Создание кода для Ajax-приложений существенно отличается от того, к чему успели привыкнуть программисты. Код, доставляемый клиенту в начале работы с приложением, доступен до завершения сеанса, причем он не прерывает и не замедляет работу пользователя и не приводит к напрасному расходованию памяти. Если разработчик стремится выйти на рынок независимых приложений, то он должен учитывать, что его программа будет использоваться по нескольку часов подряд. Для этого нужен высокопроизводительный код, допускающий сопровождение. При его создании необходима Дисциплина программирования и понимание задачи, как и при написании серверных программ.

Объем кода, вероятно, будет большим, чем в случае классического Web-приложения. В результате большее значение получает сама структура кода. Создание приложения уже становится не под силу одиночке — этим должен заниматься коллектив разработчиков. В процессе работы необходимо обеспечить выделение подзадач, учитывать дальнейшую поддержку, одним словом,

работа будет происходить по тому же сценарию, что и создание обычных программ для настольных систем.

Ajax-приложение должно эффективно взаимодействовать с сервером в процессе работы пользователя. Очевидно, что они являются непосредственными "потомками" классических Web-приложений, но похожи на них не больше, чем современный спортивный велосипед на "лошадь для денди". Лишь помня об этих отличиях, можно создать конкурентоспособное Web-приложение.

1.3. Применение богатых клиентов Ajax

Однако достаточно теории. Ajax уже используется для создания реальных приложений, и преимущества данного подхода видны на практике. В те далекие времена, когда появились велосипеды, дальновидные конструкторы уже пытались снабдить их педалями и резиновыми шинами, а также подумывали о дисковых тормозах и цепной передаче. В этом разделе мы обсудим текущее положение дел с использованием Ajax, а также подробно поговорим об одном из первых применений данной технологии.

1.3.1. Системы, созданные с использованием Ajax

Наибольший вклад в формирование современного представления об Ajax-приложениях внесла компания Google. (Она использовала данную технологию еще до того, как та получила имя Ajax.) Бета-версия службы GMail стала доступна в начале 2004 года. Эта служба привлекла внимание пользователей не только размерами предоставляемого им почтового ящика, но и интерфейсом, который позволял одновременно открывать несколько сообщений и автоматически обновлял список корреспонденции, даже если пользователь подготавливал в это время новое сообщение. Это был существенный шаг вперед по сравнению с обычными почтовыми системами, предлагаемыми большинством провайдеров. Сравнивая GMail с Web-интерфейсами корпоративных почтовых серверов, например Microsoft Outlook и Lotus Notes, нетрудно заметить, что GMail обеспечивает большинство функций, не прибегая к помощи тяжеловесных и ненадежных элементов ActiveX или Java-аплетов. В результате служба доступна не только для корпоративных пользователей, вооруженных специально настроенными машинами, но и для большинства обычных систем.

За GMail последовали другие интерактивные службы, например, Google Suggest, поисковый сервер которой автоматически предлагает завершение фразы, указываемой в составе запроса, и Google Maps — интерактивная масштабируемая карта, посредством которой определяется расположение ресурса. Начали эксперименты с данной технологией и другие компании. В качестве примера можно привести интерактивную систему Flickr, которая в настоящее время является составной частью Yahoo!.

Упомянутые выше системы — лишь проверка возможностей нового подхода, они представляют собой переходные приложения, применяемые лишь эпизодически. За ними последовали независимые Ajax-приложения, о кото-

рых мы поговорим в главе 3, а в приложении B обсудим текущее состояние дел в этой области.

Наличие приложений, созданных на базе Ajax, свидетельствует о том, что новый подход получает признание разработчиков. Если отдельные программисты иногда используют новую технологию лишь для того, чтобы ознакомиться с ней, то такие компании, как Google и Yahoo!, прибегают к ней только в том случае, если она сулит выигрыш в конкурентной борьбе. Мы уже обсуждали преимущества Ajax, которые следуют из теоретических рассуждений. В следующем разделе мы разберем Google Maps и выясним, как теоретические предпосылки реализуются на практике.

1.3.2. Google Maps

Google Maps объединяет в себе черты средств просмотра и поискового сервера. Первоначально данная служба поддерживала только карту США (рис. 1.15), но впоследствии набор доступных регионов был расширен[1]. Запрос к карте формируется в текстовом формате; допускается детализация до конкретной улицы и даже до таких заведений, как гостиницы или ресторана (рис. 1.16).

Поисковые средства функционируют как классическое Web-приложение, обновляя всю страницу, но сама карта поддерживается с использованием Ajax. После щелчка на ссылке, соответствующей гостинице, отображается подсказка; не исключено, что карта немного сдвинется в окне, чтобы наилучшим образом разместить отображаемый текст. Прокрутка карты — одна из самых интересных особенностей Google Maps. Пользователь может перетаскивать всю карту с помощью мыши. Сама карта представляет собой мозаику, составленную из маленьких изображений, и если при прокрутке должен отобразиться новый фрагмент, он подгружается в асинхронном режиме. В связи с этим при прокрутке заметна задержка: сначала отображается белая пустая область, которая постепенно заполняется по мере копирования очередного фрагмента. Пользователь может продолжать прокрутку, загружая новые изображения. Элементы карты кэшируются браузером и сохраняются в течение сеанса, поэтому возврат к той части карты, которая уже была просмотрена ранее, происходит значительно быстрее.

Возвращаясь к вопросу о практичности, нельзя не отметить две особенности. Во-первых, действия, вызывающие загрузку новых фрагментов карты, существенно отличаются от щелчка на ссылке. Эти действия пользователь выполняет, не отвлекаясь от основной работы; он делал бы то же самое и просматривая карту, реализованную с помощью локального приложения. Другими словами, действия пользователя не прерываются из-за обращения к серверу. Во-вторых, запросы передаются в асинхронном режиме, а это означает, что ссылки, элементы управления масштабированием и другие компоненты страницы остаются доступными, несмотря на загрузку новых данных.

[1] В настоящее время уже существует приложение Google Earth, которое охватывает весь земной шар. — *Примеч. ред.*

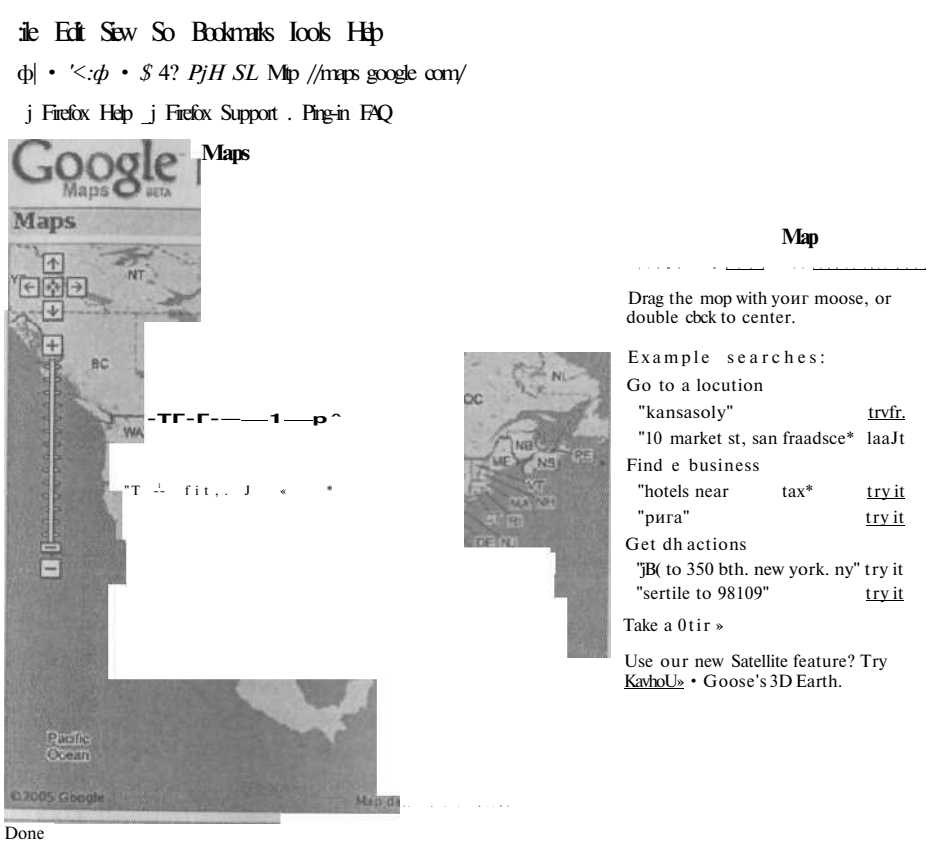


Рис. 1.15. На исходной странице Google Maps отображаются масштабируемая карта США и поле ввода ключевых слов, знакомое пользователям поискового сервера Google. Заметьте, что средства масштабирования располагаются в верхней части карты, что позволяет пользователю изменять масштаб, когда карта находится в поле зрения

Карты в Интернете — не новость. Но если мы обратимся к узлу подобного назначения, созданному задолго до появления Ajax, то увидим, что взаимодействие с пользователями реализовано совершенно по другим принципам Карта обычно разделена на фрагменты. Ссылки, управление масштабированием, как правило, расположены за пределами карты либо на обрамлении. После щелчка на одной из таких ссылок обновляется весь экран, в результате чего на той же Web-странице отображаются другие фрагменты карты. Пользователю приходится постоянно отвлекаться от основной задачи. Если он уже имел опыт работы с Google Maps, то, испытав на себе особенности взаимодействия с таким сервером, вряд ли захочет обращаться к нему снова.

Рассмотрим те же задачи с точки зрения программ, расположенных на сервере. Как службы, созданные на базе Ajax, так и службы, созданные оез использования этой инфраструктуры, применяют одинаковые решения. Ь обоих случаях карта представляется как набор простых изображений.

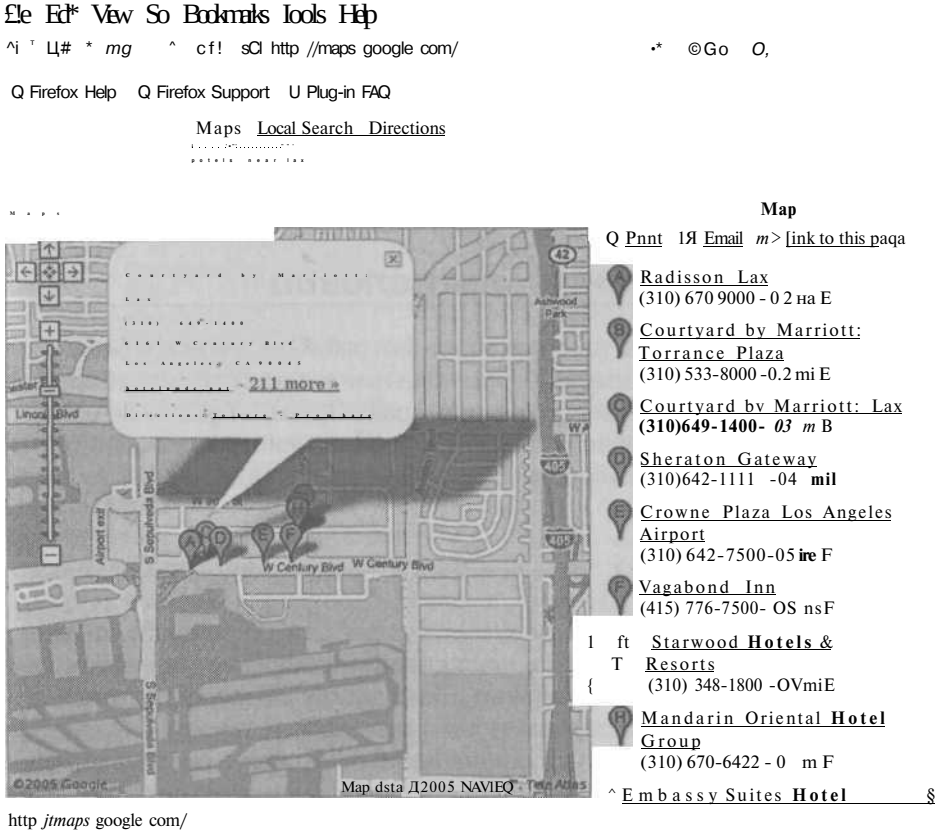


Рис. 1.16. Поиск гостиницы с помощью Google Maps. Обратите внимание на традиционное использование технологий DHTML для создания визуальных эффектов и подсказок. Запросы Ajax позволяют сделать процесс поиска более динамичным

Классический Web-сервер по мере прокрутки пользователем карты постоянно обновляет шаблон, в то время как после запуска приложения Google Maps передаются лишь необходимые данные, в частности, изображения, отсутствующие в кэше. (В обоих случаях браузер кэширует изображения, но в классических приложениях содержимое кэша используется лишь для снижения нагрузки на сеть.) Для интерактивных служб, таких как Google, простота использования является основной характеристикой, определяющей, захочет ли пользователь повторно обратиться к ней или предпочтет другой сервер. Другими словами, одним из ключевых показателей является впечатление пользователя от документа. Улучшая интерфейс и придавая ему гибкость, обеспечиваемую Ajax, компания Google заставила своих конкурентов задуматься о качестве их служб. Конечно же, нельзя упускать из виду другие факторы, например качество услуг, но при прочих равных возможностях Ajax может обеспечить существенное преимущество компании, использующей эту технологию.

Можно ожидать, что с ростом потребности в высококачественном прикладном интерфейсе традиция использования Ajax получит дальнейшее развитие. Вполне вероятно, что в течение ближайших нескольких лет позиции Ajax-приложений на рынке будут укрепляться. Однако есть и другие технологии, пригодные для создания богатых клиентов. Несмотря на то что рассмотрение этих технологий не входит в круг задач, стоящих перед нами, все же необходимо уделить им хотя бы немного внимания.

1.4. Альтернативные технологии

Технология Ajax призвана удовлетворить потребность рынка в богатых клиентах, обладающих высокими интерактивными возможностями и не требующих инсталляции на локальных компьютерах. Однако Ajax — не единственная инфраструктура подобного назначения. Для некоторых задач она даже не является наилучшим выбором. В последующих разделах мы рассмотрим альтернативные решения.

1.4.1. Macromedia Flash

Macromedia Flash — система, предназначенная для поддержки интерактивных движущихся изображений. Она использует сжатые данные в формате векторной графики. Изображения Flash могут воспроизводиться в процессе загрузки, что позволяет пользователю просматривать первые фрагменты еще до окончания копирования данных. Данная система предоставляет интерактивные возможности. Для их программирования используется ActionScript — язык, напоминающий JavaScript. Поддерживаются также компоненты, обеспечивающие ввод данных. Технология Flash подходит для самых разных применений — от компьютерных игр до сложных интерфейсов бизнес-приложений. В рамках данной технологии реализованы мощные средства поддержки графики, чем, к сожалению, не могут похвастаться базовые средства Ajax.

Технология Flash известна уже давно и поддерживается посредством встраиваемых модулей. В принципе, полагаться на модули, встраиваемые в клиентскую программу, не следует, однако модули, поддерживающие Flash, входят в комплект поставки большинства браузеров. Данная технология может использоваться на платформах Windows, Mac OS X и Linux, но средства, инсталлируемые в системе Linux, несколько уступают двум другим платформам.

Для создания богатых клиентов на базе Flash могут также использоваться две дополнительные технологии: Macromedia Flex и пакет Laszlo. Обе они реализуют на стороне сервера базовый набор средств, предназначенный для генерации интерфейсов бизнес-приложений, и используют средства J2EE (Java 2 Enterprise Edition). Для динамического управления изображениями Flash на низком уровне предоставляются специальные инструменты, например PHP-модуль libswf.

1.4.2. Java Web Start

Java Web Start — это спецификация, определяющая способ связывания серверных Web-приложений, созданных на базе Java. В результате программа, выполняемая на настольной системе, может находить, копировать и запускать их. Допускается создавать гипертекстовые ссылки, указывающие на эти приложения; такой подход обеспечивает обращение к ним из браузера, поддерживающего Web Start. Средства Web Start включены в состав последних версий Java. В процессе инсталляции продуктов Internet Explorer и Mozilla по умолчанию разрешено использование данной технологии.

Единожды скопированное приложение Web Start хранится в составе файловой системы в так называемой "песочнице" и автоматически обновляется при получении новой версии. Такой подход допускает работу при отсутствии сетевого соединения и снижает объем трафика при повторной загрузке документов. В результате становится возможной работа с приложениями объемом в несколько мегабайт. В приложениях используется цифровая подпись, и пользователь может решать, предоставлять ли им доступ к файловой системе, сетевым портам или к другим ресурсам.

Традиционно для создания интерфейса приложений Web Start используются средства Java Swing. Средствами Web Start могут доставляться компоненты SWT (Standard Widget Toolkit), используемые в составе Eclipse, но чтобы добиться этого на практике, надо затратить дополнительные усилия.

На платформе .NET реализованы средства аналогичного назначения — No Touch Deployment. Они также обеспечивают простоту доставки, богатый пользовательский интерфейс и защиту.

Основной недостаток обеих технологий — потребность в заранее установленной исполняющей системе. Очевидно, что исполняющая система необходима для любого богатого клиента, но Flash и Ajax используют для этой цели общепринятые программные средства. (Для Ajax исполняющей системой является сам браузер.) Системы Java и .NET имеются не на всех машинах, поэтому в настоящее время при реализации Web-служб, ориентированных на массовое применение, на них нельзя полагаться.

1.5. Резюме

Мы рассмотрели различия между переходными и независимыми приложениями и требования к каждому из них. Переходные приложения, конечно же, должны выполнять поставленные перед ними задачи, но считается, что, обратившись к ним, пользователь уже отвлекся от основной работы, поэтому допускается, чтобы пользователь уделил определенное внимание не только результатам работы, но и самой программе. Независимые приложения, напротив, предназначены для длительной и интенсивной работы, и хороший интерфейс должен выполнять свои функции незаметно для пользователя и не отвлекать его от решения основной задачи.

Системы клиент/сервер и n-связные архитектуры необходимы для взаимодействующих программ или для приложений, координируемых из единого центра, но для них неизбежна задержка, связанная с передачей по сети, что снижает производительность работы пользователя. Предпринимаются попытки решить проблему задержки путем асинхронной удаленной обработки событий, но, несмотря на это, необходимо признать, что модель "запрос-ответ", типичная для традиционных Web-приложений, в принципе не может обеспечить высокую производительность.

В этой главе мы поставили цель для Ajax и для нас, как пользователей этой инфраструктуры: реализовать посредством Web-браузера доставку, независимых приложений, обеспечивающих условия для высокой производительности пользователя и работу в сети. Кроме того, должна быть возможна одновременная централизованная поддержка всех приложений. Для того чтобы добиться поставленной цели, мы должны пересмотреть свой взгляд на Web-страницы и приложения. Мы определили основные правила, которыми следует руководствоваться, и привычные представления, от которых следует отказаться.

- Браузер имеет дело с приложением, а не с содержимым.
- Сервер доставляет данные, а не содержимое.
- Пользователь постоянно взаимодействует с приложением, и большинство запросов к серверу формируется без его участия.
- Код имеет большой объем, кроме того, он сложен и структурирован. Код играет главную роль в нашей архитектуре, поэтому ему надо уделять основное внимание.

В следующей главе мы обсудим основные технологии Ajax и попробуем написать небольшие фрагменты кода. Остальная часть книги будет посвящена рассмотрению важных принципов разработки, позволяющих достичь поставленных перед нами целей.

1.6. Ресурсы

Для того чтобы лучше понять вопросы, затронутые в данной главе, надо обратиться к перечисленным ниже источникам.

- Впервые инфраструктура Ajax была упомянута 18 февраля 2005 г. в статье Джесса Джеймса Гарретта, которая доступна по адресу http://www.adaptivepath.com/publications/essays/archives/000385.php.
- Рассуждения Алане. Купера о независимых и переходных приложениях можно найти в следующем документе: http://www.cooper.com/articles/art_your_programs_posture.htm.

Служба Google Maps доступна по следующим адресам.

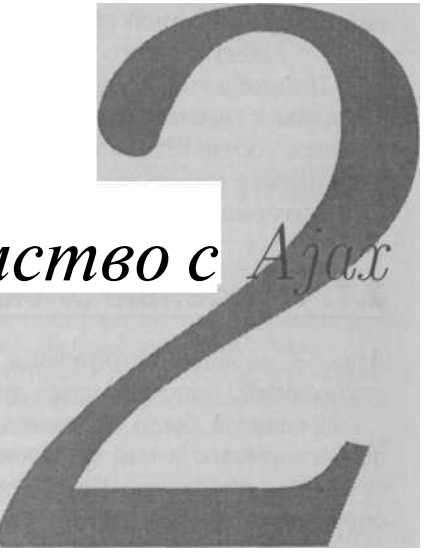
Для жителей США: http://maps.google.com

Для жителей Соединенного Королевства: http://maps.google.co.uk

Для тех, кто живет на Луне: http://moon.google.com[2]

Изображения велосипеда получены с Web-узла Pedaling History, расположенного по адресу http://www.pedalinghistory.com.

Приложение Google Earth можно найти на сайте http://earth.google.com/. — *Примеч. ред.*



Знакомство с Ajax

В этой главе...

- Технологии, лежащие в основе Ajax
- Использование каскадных таблиц стилей для формирования внешнего вида документа
- Использование Document Object Model для определения структуры пользовательского интерфейса
- Асинхронное взаимодействие с сервером посредством XMLHttpRequest
- Совместное использование базовых технологий в рамках инфраструктуры Ajax

В главе 1 мы говорили о работе пользователей и о том, как Ajax может помочь им в повседневной деятельности. Большинство из нас занимаются разработкой программ, и, убедившись в том, что инфраструктура Ajax достойна внимания, надо выяснить, как работать с ней. Несмотря на то что данный подход совершенно нов, ряд действий по созданию приложений уже знаком вам, особенно если вы имеете опыт работы с Интернетом.

В данной главе мы обсудим основы Ajax, поговорим о четырех главных принципах Ajax и их взаимосвязи, а также рассмотрим примеры, иллюстрирующие данный материал.

Эту главу можно рассматривать как вводную, подобно разделам других книг, в который обсуждается приложение "Hello, World!". Основная наша цель — добиться того, чтобы рассматриваемые примеры были работоспособны. Подробное обсуждение материала мы начнем в главе 3. Если вы уже знакомы с некоторыми технологиями, составляющими Ajax, то можете пропустить соответствующие разделы. Если же вы еще ничего не знаете об Ajax и не имеете опыта в программировании клиентских Web-программ, то вводный материал позволит вам лучше ориентироваться в остальной части книги.

2.1. Основные элементы Ajax

Ajax — не одна конкретная технология, скорее это совокупность четырех технологий, дополняющих друг друга (табл. 2.1).

В главе 1 было показано, как сложные Ajax-приложения доставляются пользователям и как пользователи взаимодействуют с ними. Это достигается за счет использования JavaScript-кода, который "объединяет" приложение, определяет последовательность действий пользователя и бизнес-логику программы. Действия с интерфейсом преобразуются в операции с элементами DOM (Document Object Model), с помощью которых обрабатываются данные, доступные пользователю, в результате чего представление их изменяется. Здесь же производится обработка перемещений и щелчков мышью, а также нажатий клавиш. Каскадные таблицы стилей, или CSS (Cascading Style Sheets), обеспечивают согласованный внешний вид элементов приложения и упрощают обращение к DOM-объектам. Объект XMLHttpRequest (или подобные механизмы) используется для асинхронного взаимодействия с сервером, обработки запросов пользователя и загрузки в процессе работы необходимых данных. Эти технологии и их взаимосвязь в рамках Ajax показаны на рис. 2.1.

Три из этих четырех технологий — CSS, DOM и JavaScript — составляют DHTML (Dynamic HTML). Следует заметить, что средства DHTML, появившиеся в 1997 году, подавали большие надежды, но так и не оправдали их. DHTML позволяет создавать на базе Web-страниц интерфейсы с достаточно большими интерактивными возможностями, но любые изменения внешнего вида страницы реализуются лишь путем повторной загрузки всего документа. Набор действий, которые можно выполнить без обращения к серверу, весьма ограничен. Средства DHTML интенсивно используются в Ajax, но благодаря асинхронным запросам можно получить результаты, которые невозможно получить с помощью обычных Web-страниц.

Таблица 2.1. Базовые технологии Ajax

JavaScript	JavaScript — это язык сценариев общего назначения, предназначенный для включения кода в Web-приложение. Интерпретатор JavaScript в составе Web-браузера обеспечивает взаимодействие со встроенными средствами браузера. Данный язык используется для создания Ajax-приложений
CSS (Cascading Style Sheets)	CSS предоставляет возможность определять стили элементов Web-страницы. С помощью данной технологии можно без труда обеспечить согласованность внешнего вида компонентов приложения. В Ajax CSS используется для изменения представления интерфейса в процессе интерактивного взаимодействия
DOM (Document Object Model)	DOM представляет структуру Web-страницы в виде набора объектов, которые можно обрабатывать средствами JavaScript. Это дает возможность изменять внешний вид интерфейса Ajax-приложения в процессе работы
Объект XMLHttpRequest	Объект XMLHttpRequest позволяет программисту получать данные с Web-сервера в фоновом режиме. Как правило, возвращаемая информация предоставляется в формате XML, но данный объект позволяет также работать с любыми текстовыми данными. Несмотря на то что XMLHttpRequest является наиболее гибким из всех инструментов общего назначения, позволяющих решать подобные задачи, существуют и другие способы получения данных с сервера. Мы обсудим их в этой главе

Очень валено, что средства поддержки всех рассматриваемых здесь технологий уже присутствуют в большинстве современных браузеров, включая Microsoft Internet Explorer, семейство Mozilla/Gecko, Firefox, Mozilla Suite, Netscape Navigator, Camino, Opera, Apple Safari и Konqueror (который ориентирован на выполнение в среде Unix KDE). К сожалению, конкретные реализации этих технологий в разных браузерах различаются рядом важных деталей, более того, различия встречаются даже в разных версиях одного продукта. Правда, за последние пять лет положение дел несколько улучшилось, и разработаны способы, позволяющие справиться с несовместимостью браузеров.

В составе каждой современной операционной системы имеется браузер. Таким образом, подавляющее большинство настольных и портативных компьютеров уже готово для запуска Ajax-приложений. Разработчики программ на базе Java или .NET могут лишь мечтать об этом. Браузеры для КПК и мобильных телефонов обычно имеют усеченный набор возможностей и не поддерживают полный набор Ajax-технологий. Следует заметить, что если бы средства поддержки Ajax и имелись в наличии, все равно размеры экрана и особенности ввода данных создавали бы существенные проблемы при работе с Ajax-приложениями. На сегодняшний день инфраструктура Ajax ориентирована лишь на настольные и портативные компьютеры.

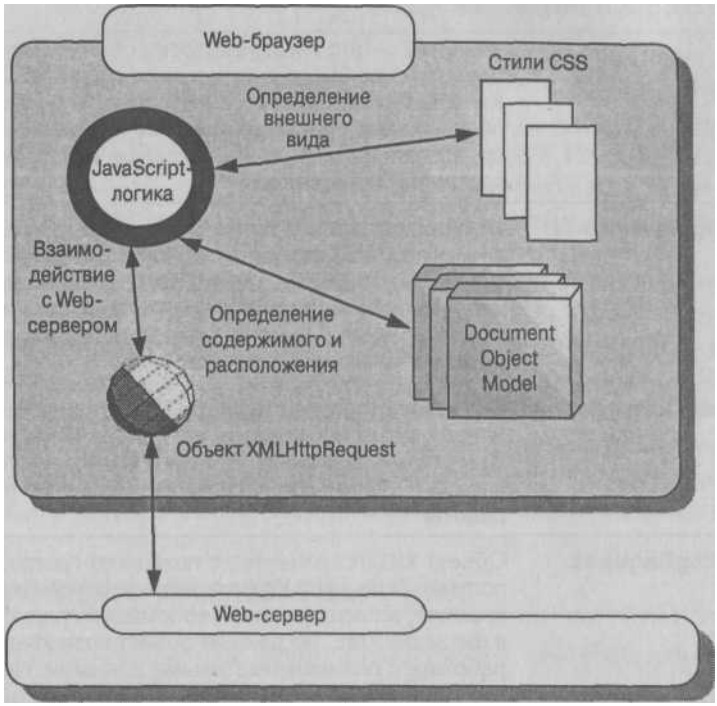


Рис. 2.1. Основные компоненты Ajax. JavaScript определяет бизнес-правила и поток выполнения. Document Object Model и каскадные таблицы стилей позволяют изменять внешний вид приложения в соответствии с данными, полученными с сервера в асинхронном режиме. Получение данных обеспечивается объектом XMLHttpRequest или другими подобными средствами

Сначала мы рассмотрим технологии Ajax, независимо друг от друга, а затем поговорим об их взаимодействии. Если вы имеете опыт разработки Web-приложений, то материал, изложенный в данной главе, уже знаком вам. В этом случае вы можете перейти к главе 3, где мы обсудим вопросы управления данными технологиями с использованием образов разработки.

Разговор о технологиях, составляющих Ajax, начнем с JavaScript.

2.2. JavaScript изучался не зря

Основным элементом Ajax несомненно является JavaScript. При работе Ajax-приложения клиентская часть, включающая данные, средства их представления и программную логику, полностью копируется в память. Инструментом для реализации программной логики является язык программирования JavaScript. Он несколько напоминает языки семейства C.

JavaScript можно охарактеризовать как язык сценариев общего назначения без поддержки типов. Отсутствие типов означает, что переменные не объявляются явно как строковые, целочисленные или объектные. Одной и той

переменной можно присваивать значения различных типов. Например, следующие строки кода допустимы.

```
var x=3.1415926;
x='pi-';
```

Сначала при определении переменной x она инициализируется числом, а впоследствии ей же присваивается строковое значение.

Термин "интерпретируемый" означает, что программа не компилируется в машинный код. Исходный код непосредственно выполняется без предварительного преобразования. Доставка JavaScript-приложения сводится к размещению исходного текста на Web-сервере и передаче этого кода по Интернету браузеру. Допустимо даже выполнение фрагментов кода в процессе копирования.

```
var x=eval('7*5');
```

Здесь вычисляемое выражение представляет собой не два числа и арифметический оператор, а строку текста. Если мы вызовем функцию eval() и передадим ей в качестве параметра строку, то JavaScript-выражение, содержащееся в строке, будет интерпретировано и функция возвратит результат вычислений. Чаще всего это лишь замедляет работу, но бывают случаи, когда такой подход позволяет увеличить гибкость программы.

Термин "общего назначения" говорит о том, что язык пригоден для решения большинства задач программирования. Базовые средства JavaScript обеспечивают поддержку чисел, строк, даты, времени, массивов, регулярных выражений, применяемых для обработки текста, и математических функций (в частности, тригонометрических). Кроме того, они обеспечивают генерацию псевдослучайных чисел. С помощью JavaScript можно определять структурированные объекты, что позволяет упорядочить сложный код и применять современные принципы разработки программ.

В среде, предоставляемой Web-браузером, процессору JavaScript доступны средства CSS, DOM и объекты XMLHttpRequest. В результате автор Web-страницы может контролировать ее поведение программными средствами. Если не учитывать объекты, специфические для браузера, то JavaScript можно рассматривать как обычный язык программирования.

Данная книга не является руководством по JavaScript. В приложении Б мы вкратце рассмотрим выразительные средства языка и обратим ваше внимание на его отличие от языков семейства C, к которому, несмотря на имя, можно отнести и Java. Фрагменты JavaScript-программ будут неоднократно встречаться в примерах, приводимых в данной книге. Существует немало книг, непосредственно посвященных основам JavaScript (ссылки на них приводятся в конце этой главы).

В наборе технологий, составляющих Ajax, язык JavaScript выполняет роль объединяющего элемента, обеспечивающего совместную работу других компонентов. Таким образом, чтобы написать Ajax-приложение, необходимо знать JavaScript. Не имея опыта программирования на данном языке, невозможно достичь успеха в использовании Ajax.

Теперь перейдем к рассмотрению каскадных таблиц стилей, которые позволяют контролировать визуальное представление элементов Web-страницы.

2.3. Определение внешнего вида с помощью CSS

Каскадные таблицы стилей, или CSS, часто используются не только в Ajax, но и в классических Web-приложениях. CSS позволяют централизованно определять стили и применять их к элементам Web-страницы. В дополнение к таким понятиям, как цвет, обрамления, фоновые изображения, прозрачность и размер, таблицы стилей определяют способ взаимодействия элементов, в результате чего удается обеспечить достаточно сложные визуальные эффекты.

В классических Web-приложениях таблицы стилей позволяют единожды определять стиль и применять его к различным Web-страницам. Говоря об Ajax, мы уже не имеем в виду последовательное отображение различных страниц. В этом случае таблицы стилей позволяют создавать набор предопределенных представлений элементов и динамически изменять внешний вид интерфейса, затрачивая минимальное время на кодирование. В данном разделе мы рассмотрим основные примеры использования CSS, но прежде надо рассказать о том, как определяются правила CSS.

CSS задают стили документа, определяя *правила*. Обычно правила помещаются в отдельный файл, на который ссылается Web-страница. Правила стилей могут быть также определены в составе самой страницы, но такой подход не приветствуется.

Правило стиля состоит из двух частей: *селектора* и *объявления стиля*. Селектор определяет, к каким элементам относятся стили, а объявление задает свойства стилей, применяемых к этим элементам. Предположим, что мы хотим, чтобы заголовки первого уровня в составе документа (определенные посредством дескриптора <H1>) выводились красным цветом. Для этого надо создать следующее правило:

```
hi { color: red }
```

Здесь селектор имеет очень простой вид. Он сообщает о том, что данный стиль применим ко всем элементам <H1>. Объявление стиля также просто; оно модифицирует одно свойство. На практике и селекторы, и объявления стилей обычно бывают более сложными. Рассмотрим их возможные разновидности. Начнем с селекторов.

2.3.1. Селекторы CSS

Помимо указания типов HTML-дескрипторов, к которым применяются стили, мы можем ограничить правило определенным контекстом. Задать контекст молено различными способами: указывая тип HTML-дескриптора, тип класса или уникальный идентификатор элемента.

Рассмотрим сначала селекторы, которые задают типы дескрипторов. Например, если вам надо применить рассмотренный выше стиль только к элементам <H1>, которые содержатся внутри элементов <DIV>, приведенное выше правило примет следующий вид:

```
div hi { color: red; }
```

Такие селекторы называют также селекторами на базе элементов, так как при определении, должен ли стиль применяться к элементу DOM, учитыва-

ется тип элемента. Мы можем также определять классы, которые не связаны с типами HTML-дескрипторов. Например, чтобы, определить класс с именем callout, который присутствует в цветном блоке, надо написать следующее выражение:

```
.callout { border: solid blue lpx; background-color: cyan }
```

Для того чтобы связать класс с элементом, надо указать в составе HTML-дескриптора атрибут class:

```
<div>I'll appear as a normal bit of text</div>
<div class='callout'>And I'll appear as a callout!</div>
```

С элементом можно связать несколько классов. Предположим, что мы определили класс loud следующим образом:

```
.loud { color: orange }
```

Ниже показано, как можно применить стили, определенные посредством классов loud и callout, к элементам документа.

```
<div class='loud'>I'll be bright orange</div>
<div class='callout'>I'll appear as a callout</div>
<div class='callout loud'>
And I'll appear as an unappealing mixture of both!
```

Текст, соответствующий третьему элементу , будет отображаться оранжевым цветом в бирюзовом блоке с синим обрамлением. Для того чтобы улучшить внешний вид интерфейса, можно также объединять стили CSS.

Сочетая классы и правила на основе элементов, есть возможность определять классы, которые будут воздействовать только на конкретные типы элементов. Например:

```
span.highlight { background-color: yellow }
```

Этот стиль будет применим только к элементам , для которых указан атрибут highlight. К элементам без данного атрибута и к элементам других типов, содержащим атрибут class='highlight', правило применяться не будет.

Классы можно сочетать с указанием родительских и дочерних элементов.

```
div.prose span.highlight { background-color: yellow }
```

Это правило применимо только к элементам класса highlight, вложенным в элементы класса prose.

В случае необходимости мы можем определять правила, которые относятся только к элементам, имеющим уникальный идентификатор. Этот идентификатор задается с помощью атрибута id. Конкретный идентификатор может быть указан в составе не более чем одного элемента, следовательно, подобные селекторы выбирают в документе один элемент. Для того чтобы выделить специальным образом кнопку, посредством которой закрывается документ, можно использовать правило, подобное следующему:

```
#close { color: red }
```

CSS также позволяет определять стили на базе псевдоселекторов. В браузере определен ограниченный набор псевдоселекторов. Например, в результате обработки представленного ниже выражения первая буква элемента будет иметь больший размер и отображаться полужирным шрифтом красного цвета.

```
*:first-letter {
 font-size: 500%;
 color: red;
 float: left;
}
```

Ограничить область действия этого правила можно следующим образом:

```
p.illuminated:first-letter {
 font-size: 500%;
 color: red;
 float: left;
}
```

Теперь оно применяется только к элементам <p> класса illuminated. Часто используются псевдоселекторы first-line и hover. Последний изменяет внешний вид гипертекстовой ссылки, на которой располагается курсор мыши. Например, чтобы при наведении курсора на ссылку она выделялась желтым цветом, надо создать следующее правило:

```
a:hover{ color:yellow; }
```

Мы рассмотрели основные принципы формирования селекторов. В примерах, иллюстрирующих сказанное, присутствовали несложные декларации стилей. Рассмотрим декларации подробнее.

2.3.2. Свойства стилей

Стиль элемента HTML-страницы может быть задан различными способами. Для универсальных элементов, например <DIV>, существуют десятки способов указания стилей. Рассмотрим некоторые из них.

Для текста в составе элемента можно задать цвет, размер шрифта, его "вес" и начертание. Если требуемый шрифт не установлен на клиентской машине, есть возможность заменить его одним из существующих. Для того чтобы абзац отображался символами серого цвета, начертание которых напоминало бы текст на экране алфавитно-цифрового терминала, можно задать следующее правило:

```
.robotic{
 font-size: 14pt;
 font-family: courier new, courier, monospace;
 font-weight: bold;
 color: gray;
```

Можно сократить запись, объединив элементы шрифта.

```
.robotic{
 font: bold 14pt courier new, courier, monospace;
 color: gray;
}
```

В любом случае свойства стилей записываются в виде пар "ключ-значение", которые отделяются друг от друга точкой с запятой.

Средствами CSS можно задать расположение и размер элемента, определить границы и область заполнения для каждой стороны или для всех четырех сторон.

```
.padded{ padding: 4px; }
.eccentricPadded {
 padding-bottom: 8px;
 padding-top: 2px;
 padding-left: 2px;
 padding-right: 16px;
 margin: 1px;
}
```

Размеры элемента задаются свойствами width и height. Позиция элемента может быть абсолютной или относительной. Абсолютная позиция указывается с помощью свойств top и left и отсчитывается в пределах всей страницы. Относительная позиция вычисляется относительно других элементов.

Для указания цвета фона предусмотрено свойство background-color. Кроме того, можно также определить фоновое изображение, указав свойство background-image.

```
.titlebar{ background-image: url(images/topbar.png); }
```

Элементы можно скрыть с помощью свойства visibility:hidden или display:none. Если задано выражение visibility:hidden, элемент не отображается, но по-прежнему занимает место на странице, а display:none полностью удаляет элемент.

Мы обсудили основные свойства стилей, необходимые для создания интерфейсов Ajax-приложений, а в следующем разделе рассмотрим, как молено применить средства CSS на практике.

2.3.3. Простой пример использования CSS

Средства CSS можно применить для создания высококачественных Web-страниц, однако нас, разработчиков Ajax-приложений, больше интересует имитация тех компонентов пользовательского интерфейса, которые пользователи привыкли видеть, работая с настольными системами. На рис. 2.2 показаны пиктограммы в виде папок, поддерживаемые с помощью CSS.

CSS выполняют две основные функции, связанные с созданием интерфейсных компонентов, подобных тем, которые показаны в правом окне на рис. 2.2. Рассмотрим каждую из этих функций.

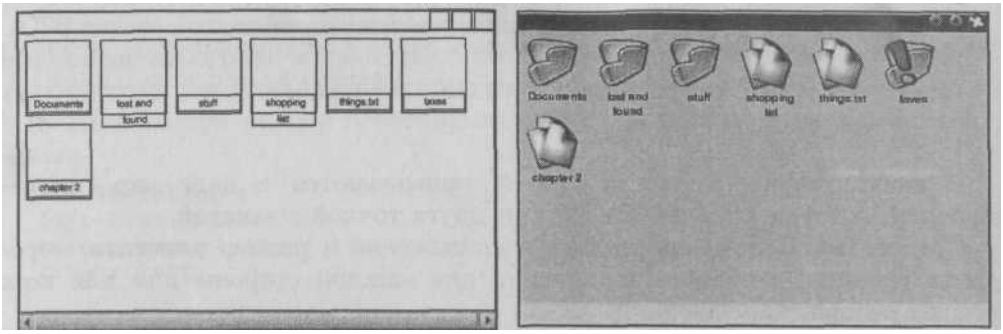


Рис. 2.2. Применение CSS для поддержки компонентов пользовательского интерфейса. Оба окна сгенерированы на основе одного и того же HTML-документа; различаются только таблицы стилей. В левом окне таблицы стилей используются лишь для позиционирования компонентов, а в правом окне с их помощью были выбраны цвет и изображения

Использование CSS для размещения компонентов

Первая из упомянутых выше задач — это позиционирование элементов. Для внешнего элемента, представляющего само окно, задается абсолютная позиция.

```
div.window{
 position: absolute;
 overflow: auto;
 margin: 8px;
 padding: Opx;
 width: 42Opx;
 height: 28Opx;
}
```

Для пиктограмм, находящихся в области содержимого, задается свойство float, в результате чего они располагаются в границах родительского элемента один за другим. После заполнения очередного ряда элементов формируется следующий ряд.

```
div.item{
 position: relative;
 height: 64px;
 width: 5 6px;
 float: left;
 padding: Opx;
 margin: 8px;
}
```

Элемент itemName, находящийся в составе элемента item, содержит текст, который располагается ниже пиктограммы. Это достигается установкой верхней границы, равной высоте графического изображения.

```
div.item div.itemName{
 margin-top: 48px;
 font: lOpx verdana, arial, helvetica;
 text-align: center;
```

Использование CSS для управления внешним видом компонентов

Вторая задача, выполняемая средствами CSS, — это формирование внешнего вида элементов. Графическое представление элементов определяется именем класса, например:

```
div.folder{
 background:
 transparent  url(images/folder.png)
 top  left  no-repeat;
}
div.filet
 background:
 transparent url(images/file.png)
 top  left  no-repeat;
}
div.special{
 background:
 transparent  url(images/folder_important.png)
 top  left  no-repeat;
}
```

Для свойства background стиля пиктограммы задаются значения, запрещающие повторение. Изображение размещается в верхнем левом углу элемента, и для него установлено значение прозрачности. (Окна, представленные на рис. 2.2, воспроизведены с помощью Firefox. Internet Explorer некорректно поддерживает прозрачность изображений .png; для устранения этого недостатка используются специальные приемы. В следующей версии этот недостаток, наверное, будет устранен. Если вам надо реализовать прозрачные изображения, одинаково отображающиеся во всех браузерах, надо использовать формат .gif.)

Для конкретных элементов указаны два класса. Один из них определяет положение в контейнере, а второй — используемую пиктограмму. Например:

```
<div  class='item  folder'>
<div   class='itemName'>stuff</div>

<div class='item
<div class='itemName'>shopping list</div>
```

Все изображения являются фоновыми. При определении стиля заголовка используется изображение, высота которого равна высоте строки, а ширина — одному пикселю. Для этого изображения задано повторение по горизонтали.

```
div.titlebar{
 background-color: #0066aa;
 background-image: url(images/titlebar_bg.png);
 background-repeat: repeat-x;
```

Полностью HTML-код компонента показан в листинге 2.1.

Листинг 2.1. Содержимое файла window.html

```
<htmixhead>
<!— Ссылка на таблицы стилей —>
<link rel='stylesheet' type='text/ess'
 href='window.ess' />
</head>
<body>
<!— Элемент окна верхнего уровня —>
<div class='window'>
 <div class='titlebar'>
 <!— Кнопки —>
 <span class='titleButton' id='close'X/span>
 <span class='titleButton' id='max'X/span>
 <span class='titleButton' id='min'></span>

 <div class='contents'>
 <div class='item folder'>
 <div class='itemName'>Documents</div>

 <div class='item folder'>
 <div class='itemName'>lost and found</div>

 <!— Пиктограмма в окне —>
 <div class='item folder'>
 <div class='itemName'>stuff</div>

 <div class='item
 <div class='itemName'>shopping list</div>

 <div class='item
 <div class='itemName'>things.txt</div>

 <div class='item special'>
 <div class='itemName'>faves</div>

 <div class='item
 <div class='itemName'>chapter 2</div>


</body>
</html>
```

HTML-разметка определяет не внешний вид, а лишь структуру документа. Она также указывает, к каким частям документа должно быть применено форматирование. Для этой цели используются имена классов, уникальные идентификаторы и типы самих дескрипторов. Просматривая HTML-код, мы видим, например, что одни элементы содержатся в составе других, но не можем сказать, как они будут выглядеть на экране. Редактируя таблицы стилей, можно изменить внешний вид документа, сохранив его структуру. Это видно на рис. 2.2. Таблицы стилей для компонента показаны в листинге 2.2.

Листинг 2.2. Содержимое файла window.css

```
div.window{
 position: absolute;
 overflow: auto;
 background-color: feeefff;
 border: solid #0066aa 2px;
 margin: 8px;
 padding: Opx;
 /* 1 Размеры элемента */
 width: 42Opx;
 height: 280px;
}
div.titlebar{
 /* 2 Текстура фона */
 background-color: #0066aa;
 background-image:
 url(images/titlebar_bg.png);
 background-repeat: repeat-x;
 color:white;
 border-bottom: solid black lpx;
 width: 100%;
 height: 16px;
 overflow:hidden;
}
span.titleButton{
 position: relative;
 height: 16px;
 width: 16px;
 padding: Opx;
 margin: Opx lpx; Opx lpx;
 /* 3 Выравнивание */
 float:right;
}
span.titleButtonfmin{
 background: transparent
 url(images/min.png) top left no-repeat;
}
span.titleButton#max{
 background: transparent
 url(images/max.png) top left no-repeat;
}
span.titleButton#close{
 background: transparent
 url(images/close.png) top left no-repeat;
}
div.contents {
 background-color: #e0e4e8;
 overflow: auto;
 padding: 2px;
 height:240px;
}
div.item{
 position : relative;
 height : 64px;
 width: 56px;
 float: left;
```

```
 color : #004488;
 font-size: 18;
 padding: Opx;
 margin: 4px;
}
div.item div.itemName {
 /* 4 Позиционирование текста */
 margin-top: 48px;
 font: 10px verdana, arial, helvetica;
 text-align: center;
}
div.folder{
 background: transparent
 url(images/folder.png) top left no-repeat;
}
div.file{
 background: transparent
 url(images/file.png) top left no-repeat;
}
div.special{
 background: transparent
 url(images/folder_important.png)
 top left no-repeat;
```

Мы уже рассмотрели ряд приемов, примененных в составе данных таблиц гилей и предназначенных для форматирования внешнего вида конкретных лементов. В листинге мы также отметили фрагменты, демонстрирующие озможности CSS: размещение на экране О, текстура ©, взаимное располоение элементов © и позиционирование текста относительно графики О.

CSS — важный элемент набора инструментальных средств для разработи Web-приложений. Как было показано выше, каскадные таблицы стилей южно применить не только для настройки внешнего вида отдельных Webграниц, но и для формирования пользовательского интерфейса, что имеет олыное значение при разработке Ajax-приложений.

\Л. Организация просмотра с помощью POM

ЮМ (Document Object Model) представляет документ (Web-страницу) роцессору JavaScript. Благодаря DOM появляется возможность управять структурой документа (рис. 2.3) из программы. Такая возможность райне важна для разработчиков Ajax-приложений. В классическом Webриложении все содержимое окна браузера регулярно обновляется при поучении с сервера потока, содержащего HTML-данные. Подготавливая ноый вариант документа, можно существенно изменить интерфейс. В Ajaxриложении большая часть изменений интерфейса осуществляется с помошно DOM. HTML-элементы в составе Web-страницы составляют древовидую структуру. Корнем данного дерева является дескриптор HTML, который редставляет весь документ. В нем содержится элемент BODY, соответству-

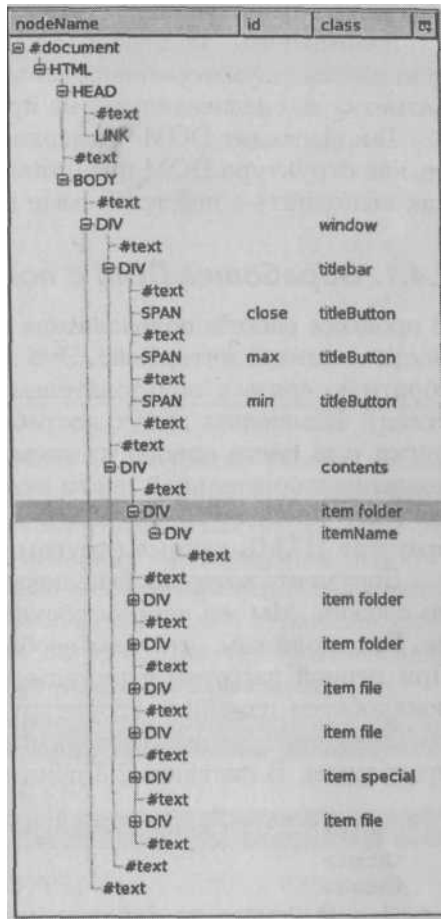


Рис. 2.3. HTML-документ представляется средствами DOM как древовидная структура, каждый элемент которой соответствует HTML-дескриптору

ющий телу документа. Он, в свою очередь, является корнем отображаемой структуры. В теле документа находятся таблицы, абзацы, списки и прочие элементы, которые могут содержать другие дескрипторы.

DOM-представление Web-страницы также имеет древовидную структуру, составленную из элементов, или узлов, которые содержат дочерние узлы, и т.д. Процессор JavaScript представляет корневой узел текущей Web-страницы посредством глобальной переменной document, которая выполняет роль стартовой точки для всех действий с DOM-данными. Структуру DOM определяет спецификация W3C. Для каждого элемента древовидной структуры существует один родительский элемент, любое, в том числе нулевое количество дочерних элементов, и любое количество атрибутов, хранящихся в ассоциативном массиве (т.е. массиве, для которого ключевым значением является не числовой индекс, а текст). На рис. 2.3 показана структура документа, код которого был представлен в листинге 2.2. Данная структура отображается с помощью инструмента Mozilla DOM Inspector (подробно он

рассмотрен в приложении А).

Взаимосвязь элементов DOM отражает структуру HTML-документа и является двухсторонней. Изменение структуры DOM влияет на HTML-разметку и, следовательно, на представление страницы.

Так выглядит DOM "на верхнем уровне". В следующем разделе вы увидите, как структура DOM представляется интерпретатору JavaScript, и узнаете, как выполнять с ней требуемые действия.

2.4.1 *Обработка DOM с помощью JavaScript*

В процессе работы пользователя с любым приложением возникает необходимость изменять интерфейс. Это надо, например, для того, чтобы обеспечить обратную связь с пользователем (он должен видеть реакцию на свои действия). Изменения могут потребоваться любые: замена текста статической метки или цвета одного из элементов, вывод диалогового окна и даже обновление значительной части окна и вывод нового набора компонентов. Чаще всего приходится создавать древовидную структуру DOM, предоставляя браузеру HTML-данные (другими словами, формировать Web-страницу).

Документ, который был показан в листинге 2.2 и на рис. 2.3, достаточно сложен. Мы же начнем обсуждение действий с DOM на простом примере. Предположим, что нам необходимо вывести приветствие пользователю. При первой загрузке страницы мы еще не знаем его имени, поэтому нам понадобится изменить структуру документа несколько позже, добавив имя пользователя. Воздействовать на узлы DOM, вероятнее всего, придется из программы. В листинге 2.3 показан HTML-код простой Web-страницы.

Листинг 2.3. Страница Ajax-приложения hello

```
<html>
<head>
<!— О Ссылка на таблицы стилей —>
<link rel='stylesheet' type='text/ess'
 href«•hello.ess' />
<!— © Ссылка на JavaScript-сценарий —>
<script type='text/javascript'
 src='hello.j s'></script>
</head>
<body>
<p id='hello'>hello</p>
<!— © Пустой элемент —>
<div id='empty'x/div>
</body>
</html>
```

В документе содержатся ссылки на два внешних ресурса: каскадные таблицы стилей О и файл, содержащий JavaScript-код ©. Мы также определили пустой элемент div с идентификатором ©. С помощью программы мы можем 5ключить в его состав другие элементы.

Рассмотрим ресурсы, на которые ссылается документ. В CSS-файле определены простые стили, позволяющие выделять пункты списка и изменять

шрифт и цвет. Содержимое CSS-файла приведено в листинге 2.4.

Листинг 2.4. Содержимое файла hello.css

```
.declared{
 color: red;
 font-family: arial;
 font-weight: normal;
 font-size: 16px;
}
.programmed!
 color: blue;
 font-family: helvetica;
 font-weight: bold;
 font-size: lOpx;
```

Мы определили два стиля для исходных DOM-узлов. (Имена стилей могут быть произвольными. В данном случае мы выбрали их так, чтобы упростить восприятие примера, но с таким же успехом мы могли использовать имена fred и jim.) Ни один из этих стилей не используется в HTML-доукменте, но они могут быть применены к элементам в результате выполнения программы. В листинге 2.5 показан JavaScript-код для страницы, представленной в листинге 2.4. Когда документ загружается, программа связывает стиль с существующим узлом и создает новые DOM-элементы.

Листинг 2.5. Содержимое файла hello.js

```
window.onload=function(){
 // Получение элемента по идентификатору
 var hello=document.getElementById('hello');
 hello.className='declared';
 var empty=document.getElementById('empty');
 addNode(empty,"reader of");
 addNode(empty, "Ajax in Action!");
 var children=empty.childNodes;
 for (var i=0;i<children.length;i++){
 children[i].className='programmed';
 }
 // Непосредственная установка стилей
 empty.style.border='solid green 2px';
 empty.style.width="200px";
}
function addNode(el,text){
 // Создание нового элемента
 var childEl=document.createElement("div");
 el.appendChild(childEl);
 // Формирование текста
 var txtNode=document.createTextNode(text);
 childEl.appendChild(txtNode);
```

Код JavaScript сложнее для восприятия, чем HTML-документ или стили. Точкой входа в данном случае является функция window.onload(), которая вызывается после загрузки всей страницы. К этому моменту древовидная структура DOM уже сформирована, и мы можем начинать работу с ней. В листинге 2.5 имеется несколько методов, предназначенных для модификации структуры DOM. В частности, с их помощью можно изменять атрибуты, скрывать и вновь отображать DOM-узлы и даже создавать в процессе выполнения программы новые узлы. Мы не будем останавливаться на каждом методе (подробную информацию о них вы сможете найти в источниках, ссылки на которые приведены в конце главы), а ограничимся рассмотрением в последующих разделах некоторых, используемых наиболее часто.

2.4.2. Поиск узла DOM

Первое, что надо сделать для того, чтобы изменить структуру DOM из JavaScript-программы, — найти элементы, которые следует модифицировать. Как было сказано ранее, в начале работы мы имеем лишь ссылку на корневой узел в глобальной переменной document. Все остальные узлы DOM являются дочерними (или более отдаленными потомками) document. Организовать пошаговый просмотр древовидной структуры большого документа — достаточно сложная задача. К счастью, ее можно упростить. Один из самых распространенных приемов — пометить элемент уникальным идентификатором. В функции onloadO, код которой показан в листинге 2.5, мы организуем поиск двух элементов: абзаца, стиль которого изменяем, и пустого элемента , в состав которого включаем новые элементы. В открывающем дескрипторе каждого из этих элементов мы с помощью атрибута id задаем идентификатор.

```
<p id='hello'>
```

и

```
<div id='empty'></div>
```

С каждым DOM-узлом можно связать идентификатор и использовать его при вызове функции, как показано ниже.

```
var hello=document.getElementById('hello') ;
```

Заметьте, что данный метод принадлежит объекту Document. В простых случаях, подобных рассматриваемому, молено ссылаться на текущий объект Document посредством переменной document. При использовании IFrame появляется несколько объектов Document, и приходится внимательно следить за тем, к какому из них осуществляется обращение.

В некоторых ситуациях необходимо организовать пошаговый обход дерева DOM. Поскольку узлы DOM организованы в виде древовидной структуры, каждый узел может иметь один родительский узел и любое число дочерних узлов. Доступ к ним осуществляется с помощью свойств parentNode и childNodes. Свойство parentNode ссылается на другой объект DOM, а childNodes — на JavaScript-массив узлов, которые приходится просматривать по очереди.

```
var children=empty.childNodes;
for (var i=0;Kchildren.length;i++){
```

Если требуемый узел не помечен уникальным идентификатором, поиск его осуществляется по-другому: с помощью функции getElementsByTagName. Например, в результате вызова document.getElementsByTagName("UL") будет возвращен массив элементов UL, присутствующих в документе.

Последний способ применяется при работе с документом, структуру которого разработчик не может контролировать. Считается, что функция getElementByld дает более надежные результаты, чем getElementsByTagName (), поскольку при ее использовании не приходится учитывать структуру документа, которая может измениться.

2.4.3. Создание узла DOM

Помимо модификации существующих узлов, может возникнуть необходимость создавать новые узлы и включать их в текущий документ (например, в процессе работы программы может потребоваться сформировать сообщение). JavaScript-реализация DOM предоставляет методы для решения этой задачи.

Вернемся к листингу 2.5. В начале работы узел с идентификатором empty пуст. Когда документ загружается, мы динамически формируем содержимое этого узла. В функции addNode () вызываются стандартные методы document .createElement() и document.createTextNode(). Метод createElement (), которому в качестве параметра передается тип дескриптора, создает HTML-элемент.

```
var childEl=document.createElement("div") ;
```

Метод createTextNode () формирует DOM-узел, представляющий фрагмент текста. Этот узел обычно является дочерним по отношению к заголовку, элементу div, абзацу или пункту списка.

```
var txtNode=document.createTextNode("some text");
```

Согласно стандарту DOM текстовые узлы отличаются от узлов, представляющих HTML-элементы. К ним нельзя непосредственно применять стили, поэтому они занимают меньше памяти, чем обычные узлы. Стиль текста, соответствующего текстовому узлу, определяется стилем родительского DOM-элемента.

Чтобы созданный узел мог отображаться в окне браузера, его надо включить в состав документа. Для этого предусмотрен метод appendChild().

```
el.appendChild(childEl) ;
```

Методов createElement(), createTextNode() и appendChild() достаточно для добавления новых узлов в документу. Однако новые узлы нуждаются в стилях. Рассмотрим, какими способами можно добавить их.

2.4.4. Добавление стилей к документу

До сих пор мы говорили о средствах, влияющих на структуру документа, и научились изменять статические HTML-данные. Кроме того, разработчику доступны методы, позволяющие изменять структуру таблиц стилей и, следовательно, модифицировать из программы стили элементов.

Выполняя действия с DOM, можно реализовать ряд визуальных эффектов для каждого элемента, например, задать его позицию, размеры, цвет, границы и обрамление. Модификация отдельных атрибутов позволяет управлять деталями отображения, однако эта процедура достаточно утомительна. Web-браузеры предоставляют средства, обеспечивающие необходимую точность при управлении интерфейсом из JavaScript-программы. Рассмотрим их более подробно.

Свойство className

Если элементы стилей созданы в результате выполнения кода программы, мы можем воспользоваться свойством className узла DOM. Ниже приведена строка кода, в результате выполнения которой к узлу применяются стили, определенные посредством класса declared.

```
hello.className='declared' ;
```

Здесь hello — это ссылка на узел DOM. Таким способом можно связывать с узлом любое количество правил CSS и управлять сложными стилями.

Свойство style

В некоторых случаях бывает необходимо изменить стиль конкретного элемента, т.е. модифицировать стиль, уже заданный средствами CSS.

С каждым узлом DOM связан ассоциативный массив style, содержащий информацию о стилях, установленных для данного узла. Как видно на рис. 2.4, в массиве стилей DOM-узла обычно содержится большое число элементов. Заметьте, что при присвоении значения свойству className узла автоматически модифицируются значения в массиве style.

Значениями массива style можно управлять непосредственно. Ниже показано, как отобразить рамку вокруг узла empty, задавая его стили.

```
empty.style.border="solid green 2px"; empty.style.width="200px";
```

Мы можем определить класс box и применить его посредством свойства className, но в ряде случаев описанный здесь подход проще и позволяет получить результаты быстрее. Кроме того, он позволяет формировать необходимые строки в программе. Если нам надо изменять размеры элементов с точностью до пикселя, то определять стили для значений ширины в интервале от 1 до 800 слишком сложно и неэффективно.

Используя рассмотренные выше способы, мы можем создавать новые элементы DOM и определять их стили. Существует еще один инструмент, позволяющий управлять содержимым документа. Он реализует альтернативный подход к решению задачи формирования Web-страницы с помощью программы. Рассмотрим свойство innerHTML.

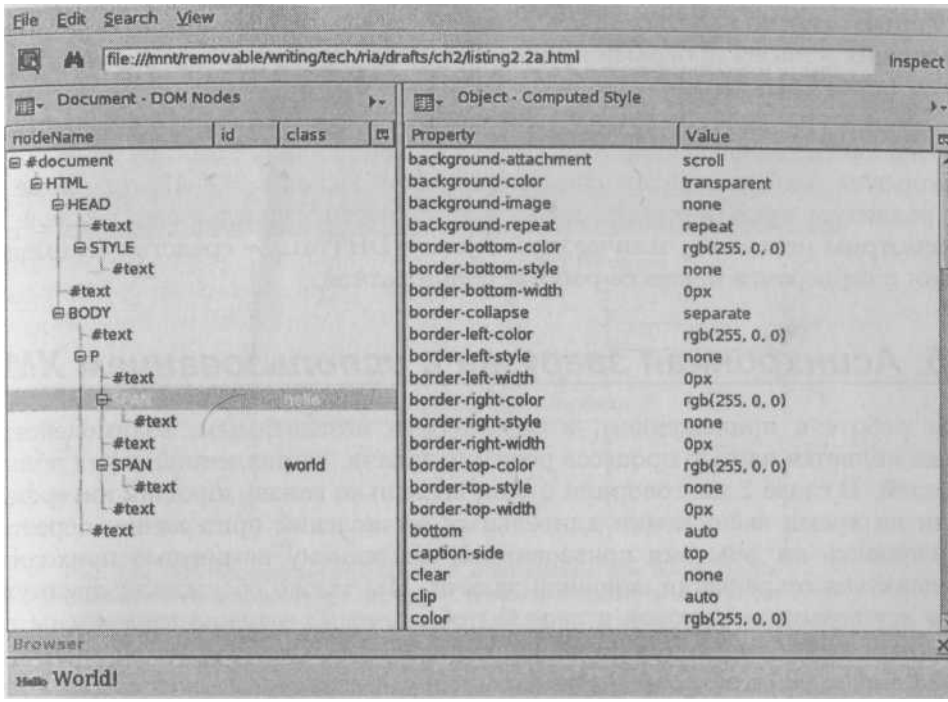


Рис. 2.4. Данные, соответствующие атрибуту s t y l e DOM-узла и отображаемые с помощью DOM Inspector. Большинство значений было установлено по умолчанию. Обратите внимание на полосу прокрутки: на экране отображается приблизительно четверть всех поддерживаемых стилей

2.4.5. Свойство innerHTML

Описанные выше методы обеспечивают управление структурой D0M на низком уровне. Методы createElement () и appendChild() лучше всего подходят тогда, когда документ имеет регулярную структуру и процедура создания документа легко формализуема. Все популярные браузеры поддерживают также innerHTML — свойство DOM-элементов. Это свойство позволяет без труда включить в состав элемента произвольное содержимое. Значением свойства innerHTML является строка, представляющая дочерние узлы в формате HTML. Используя данное свойство, мы можем переписать функцию addNode следующим образом:

```
function addListItemUsingInnerHTML(el,text){
 el.innerHTML+="<div class='programmed'>"+text+"</div>";
}
```

Элемент div и содержащийся в нем текстовый узел можно добавить с помощью одного выражения. Заметьте также, что строка не присваивается свойству непосредственно, а добавляется к нему с помощью оператора +=. Чтобы удалить узел путем редактирования свойства innerHTML, надо извлечь значение и организовать его разбор. Свойство innerHTML применимо

для сравнительно простых задач. Если необходимо обеспечить многократные изменения узла из программы, целесообразнее воспользоваться рассмотренными ранее средствами.

Мы рассмотрели технологии JavaScript, CSS и DOM, которые давно известны как Dynamic HTML. Как было замечено в начале данной главы, Ajax использует многие средства, характерные для Dynamic HTML, но, кроме того, реализует принципиально новые возможности. В следующем разделе мы рассмотрим механизм, отличающий Ajax от DHTML, — средства взаимодействия с сервером в процессе работы пользователя.

2.5. Асинхронная загрузка с использованием XML

При работе с приложением, в особенности независимым, взаимодействие с ним является частью процесса решения задачи, поставленной перед пользователем. В главе 1 мы говорили о том, насколько важен хороший интерфейс. Если на время выполнения длительных вычислений приложение перестает реагировать на действия пользователя, последнему неминуемо приходится отвлекаться от решения основной задачи. Мы также обсуждали преимущества асинхронных вызовов и способы обеспечения отклика программы при сложных вычислениях и выяснили, что вследствие неустранимой задержки, связанной с работой в сети, все обращения к серверу следует считать длительными операциями. Мы также обратили ваше внимание на то, что модель, предполагающая обмен в виде HTTP-запросов и ответов на них, не позволяет создавать интерфейсы высокого качества. В классических Web-приложениях каждое обращение к серверу сопровождается полным обновлением текущей Web-страницы, в результате чего нормальная последовательность действий пользователя часто прерывается.

Несмотря на то что мы не можем сократить время, которое проходит от передачи запроса серверу до получения ответа от него, существуют способы передачи запросов в асинхронном режиме; при этом пользователь может продолжать работу с приложением. Первой попыткой решить подобную задачу было взаимодействие в фоновом режиме с использованием элементов I Frame. Впоследствии было найдено лучшее решение — использование объекта XMLHttpRequest. Ниже мы рассмотрим обе технологии.

2.5.1. Элементы IFrame

С появлением четвертой версии браузеров Netscape Navigator и Microsoft Internet Explorer стали доступны средства DHTML, обеспечивающие возможность компоновки Web-страницы из программы. Естественным расширением системы фреймов HTML стал элемент I Frame. В его названии первая буква условно означает *inline* (встроенный), т.е. фрейм является элементом другого документа, а не его полноправным "соседом". IFrame представляется как элемент древовидной структуры DOM, а это значит, что его можно перемещать, изменять размеры и даже скрыть, причем остальная часть страницы останется видимой. Тот факт, что с элементом I Frame можно связывать стили

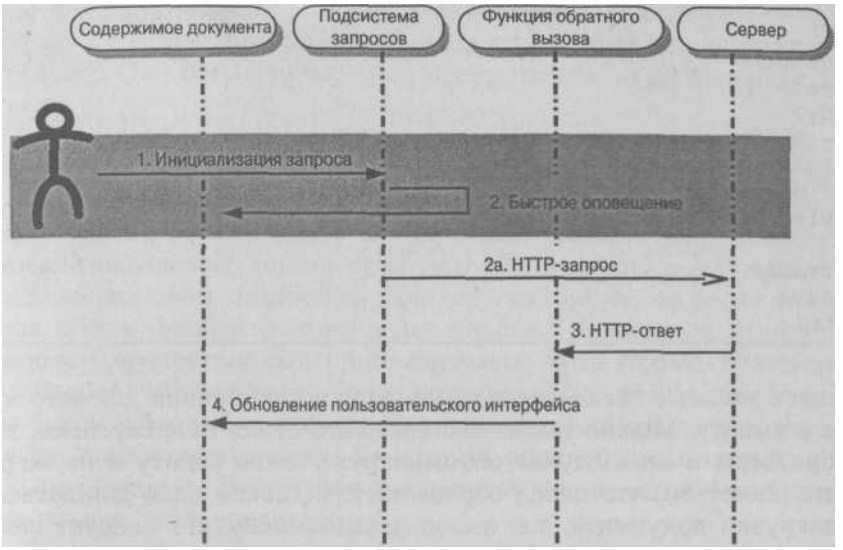


Рис. 2.5. Последовательность событий, возникающих при асинхронном взаимодействии документа с сервером. В результате действий пользователя формируется асинхронный запрос серверу от скрытого объекта (IFrame или XMLHttpRequest). Метод очень быстро возвращает управление, блокируя интерфейс лишь на короткий период времени (этот интервал выделен серым цветом). Функция обратного вызова осуществляет разбор ответа сервера и затем обновляет пользовательский интерфейс

и делать его невидимым, открывает большие возможности в создании пользовательских интерфейсов. Так, например, данные можно загружать в фоновом режиме, не затрагивая остальных элементов Web-страницы. По сути, это способ асинхронного взаимодействия с сервером, правда, пользоваться им несколько неудобно. На рис. 2.5 показана последовательность событий, типичная для данного подхода.

Подобно другим элементам DOM, IFrame можно объявить в составе HTML-кода страницы либо сгенерировать программно, вызвав document.createElement(). В простых ситуациях, когда нам нужен один невидимый элемент IFrame для загрузки данных, мы можем объявить его как часть документа и получить доступ к нему из программы, вызвав document.getElementByld(), как показано в листинге 2.6.

Листинг 2.6. Использование элемента IFrame

```
<html>
<head>
<script type='text/javascript'>
window.onload=function(){
 var iframe=document.getElementById('dataFeed');
 var src='datafeeds/mydata.xml';
 loadDataAsynchronously(iframe,src)   ;

function  loadDataAsynchronously(iframe,src){
```

```
 // Выполнение требуемых действий
}
</script>
</head>
<body>
<!— Отображаемые данные —>
<iframe
 id='dataFeed'
 style='height:Opx;width:Opx;'
>
</iframe>
</body>
</html>
```

Сделать элемент IFrame невидимым можно, установив для него нулевую ширину и высоту. Можно также использовать стиль display:none, но некоторые браузеры в этом случае оптимизируют свою работу и не загружают документ. Заметьте, что перед обращением к IFrame надо дождаться окончания загрузки документа, т.е. вызов getElementByldO следует поместить в обработчик window, onload. Можно также генерировать элементы IFrame из программы, как показано в листинге 2.7. В этом случае весь код, связанный с передачей запроса, находится в одном месте, и нет необходимости указывать уникальный идентификатор узла DOM и согласовывать его в сценарии и HTML-файле.

Листинг 2.7. Создание элемента IFrame

```
function fetchData(){
 var iframe=document.createElement('iframe') ;
 iframe.className='hiddenDataFeed' ;
 document.body.appendChild(iframe);
 var src='datafeeds/mydata.xml';
 loadDataAsynchronously(iframe,src);
```

Методы createElement и appendChild, используемые для модификации структуры DOM, уже знакомы вам по предыдущим примерам. Если мы будем не задумываясь применять данный подход в процессе работы приложения, то получим большое число элементов IFrame. Необходимо удалять их по завершении работы или формировать пул элементов.

Образы разработки, которые мы рассмотрим в главе 3, помогут вам в реализации пулов, очередей и других структур, которые могут быть использованы для масштабных приложений. Эти вопросы мы рассмотрим несколько позже, а сейчас перейдем к следующему набору технологий, используемых для формирования асинхронных запросов к серверу.

2.5.2. Объекты XmlDocument и XMLHttpRequest

Элементы IFrame можно использовать для получения данных в асинхронном режиме, но сделать это достаточно сложно, так как изначально они были

созданы для другой цели. В последних версиях популярных Web-браузеров реализованы специальные объекты, предназначенные для асинхронной передачи данных. Они обеспечивают ряд преимуществ по сравнению с элементами IFrame.

Объекты XmlDocument и XMLHttpRequest, хотя и представляют собой нестандартные расширения DOM, поддерживаются большинством браузеров. Поскольку они специально разработаны для фоновой загрузки данных, с их помощью удобно осуществлять асинхронные вызовы. Оба объекта сначала были реализованы как компоненты ActiveX, доступные в браузере Internet Explorer посредством JavaScript. Для других браузеров были реализованы объекты, обеспечивающие те же возможности. Объекты XmlDocument и XMLHttpRequest предоставляют приблизительно одинаковые возможности, но XMLHttpRequest обеспечивает более полный контроль над запросом. Именно его мы будем в основном использовать в данной книге, но уделим внимание и объекту XmlDocument, чтобы вы смогли оценить его и сравнить с объектом XMLHttpRequest. В листинге 2.8 показан код простой функции, которая создает объект XmlDocument.

Листинг 2.8. Функция getXMLDocument ()

```
function getXMLDocument(){
 var xDoc=null;
 if (document.implementation
 S&document.implementation.createDocument){
 xDoc=document.implementation
 // Mozilla/Safari
 .createDocument("","",null);
 }else if (typeof ActiveXObject != "undefined"){
 var msXmlAx==null;
 try{
 msXmlAx=new ActiveXObject
 // Последние версии Internet Explorer
 ("Msxml2.DOMDocument");
 }catch (e){
 msXmlAx=new ActiveXObject
 // Старые версии Internet Explorer
 ("Msxml.DOMDocument");


 xDoc=msXmlAx;
 }
 if (xDoc==null M typeof xDoc.load=="undefined"){
 xDoc=null;
 }
 return xDoc;
}
```

Данная функция возвращает объект XmlDocument с API, идентичным для большинства современных браузеров. Однако способы создания документа могут существенно различаться.

Функция проверяет, предоставляет ли объект документа возможности, необходимые для создания объекта XmlDocument (подобные средства имеются в последних версиях браузеров Mozilla и Safari). Если необходимые средства не предоставляются, функция пытается обратиться к объектам ActiveX и проверяет, поддерживаются ли они (такая поддержка реализована только в браузерах производства Microsoft). Если проверка дает положительный результат, функция пытается извлечь требуемый объект. Предпочтение отдается библиотекам MSXML версии 2.

На заметку Из программы можно запросить информацию о производителе браузера и номер версии продукта. Очень часто таким образом производится настройка кода на конкретный тип браузера. На наш взгляд, подобный подход чреват ошибками. Кроме того, таким образом невозможно предусмотреть работу с более новыми версиями браузеров. В нашей функции getXmlDocument () мы не пытаемся узнать версию клиентской программы, а непосредственно выясняем, доступен ли конкретный объект. Такой подход, называемый *обнаружением объектов* (object detection), увеличивает вероятность того, что программа будет нормально работать с последующими версиями браузеров и теми браузерами, наличие которых мы не учли при написании программы.

В листинге 2.9 показана функция, подобная предыдущей, но ориентированная на объект XMLHttpRequest.

ЛИСТИНГ 2.9. Функция getXMLHTTPRequest ()

```
function getXMLHTTPRequest() {
 var xRequest=null;
 if (window.XMLHttpRequest) {
 // Mozilla/Safari
 xRequest=new XMLHttpRequest();
 }else if (typeof ActiveXObject != "undefined"){
 xRequest=new ActiveXObject
 // Internet Explorer
 ("Microsoft.XMLHTTP");

 return xRequest;
```

Здесь мы также используем принцип обнаружения объектов, чтобы проверить, поддерживается ли XMLHttpRequest. Если проверка дает отрицательный результат, мы пытаемся обратиться к ActiveX. Если браузер не предоставляет ни одну из проверяемых возможностей, функция возвращает значение null. О более детальной обработке отрицательного результата проверки речь пойдет в главе 6.

Итак, мы можем создать объект, который передает запросы серверу. Попробуем использовать его.

2.5.3. Передача запроса серверу

Объект XMLHttpRequest позволяет без труда передать запрос серверу. Все, что надо для этого, — это указать URL серверного ресурса, который должен сгенерировать необходимые данные. Ниже представлена функция, которая решает эту задачу.

```
function sendRequest(url,params,HttpMethod){
 if (IHttpMethod){
   HttpMethod="POST";}
   var req=getXMLHTTPRequest();
   if (req){
   req.open(HttpMethod,url,true);
   req.setRequestHeader
     ("Content-Type", "application/x-www-form-urlencoded");
   req.send(params);
 }
}
```

Объект XMLHttpRequest предоставляет широкий спектр возможностей для работы с протоколом HTTP, в том числе позволяет указывать строку параметров для динамически генерируемых документов. (Эти параметры обрабатываются CGI-сценарием, объектом ServletRequest или другой серверной программой.) Рассмотрим основы протокола HTTP, а затем поговорим о том, как объект, передающий запросы серверу, работает с ним.

Общие сведения о протоколе HTTP

Протокол HTTP настолько привычен, что мы часто забываем о его существовании. При написании классических Web-приложений наши действия с данным протоколом ограничиваются определением гипертекстовых ссылок и, возможно, установкой атрибута method формы. Ajax, напротив, предоставляет разработчикам доступ к низкоуровневым средствам работы с протоколом HTTP, использование которых открывает удивительные возможности.

HTTP-транзакция представляет собой запрос браузера, за которым следует ответ сервера. (Конечно, в процессе обработки запроса, помимо стандартных серверных средств, могут участвовать сложные программы, написанные нами, Web-разработчиками.) И запрос, и ответ передаются в текстовом виде; каждый из них состоит из заголовка, за которым следует тело запроса или ответа. Заголовок можно условно представить себе как адрес, написанный на конверте, а тело запроса или ответа — как содержащееся внутри письмо. Заголовки указывают принимающей стороне на то, что надо сделать с остальными данными.

HTTP-запрос чаще всего состоит из одного заголовка, однако в некоторых случаях за ним могут следовать данные. В составе ответа обычно содержится HTML-код страницы, указанной в запросе. С браузерами Mozilla поставляется полезная утилита под названием LiveHTTPHeaders (см. ссылки в конце данной главы и приложение А). Она позволяет просматривать заголовки запросов и ответов в процессе работы браузера. Загрузим исходную страницу Google и посмотрим, что произойдет при этом.

Переданный нами запрос содержит следующий заголовок:

```
GET / HTTP/1.1
Host: www.google.com
User-Agent: Mozilla/5.0
 (Windows; U; Windows NT 5.0; en-US; rv:1.7)
 Gecko/20040803 Firefox/0.9.3
Accept: text/xml,application/xml,
 application/xhtml+xml,text/html;q=0.9,
 text/plain;q=O.8,image/png,*/*;q=0.5
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Cookie: PREF=ID=cabd38877dc0b6al:TM=1116601572
 :LM=1116601572:S=GD3SsQk3v0adtSBP
```

Первая строка заголовка сообщает о том, какой HTTP-метод должен быть использован. Большинство разработчиков Web-приложений знакомы с запросами GET и POST. Запрос GET используется для загрузки документов, а POST чаще всего применяется для передачи на сервер данных, введенных посредством HTML-форм. В спецификации, разработанной W3C (World Wide Web Consortium), предусмотрены и другие методы, в частности, HEAD, который загружает только заголовок; PUT, предназначенный для передачи документов на сервер; и DELETE, посредством которого можно удалять документы. За первой строкой следуют поля заголовка, с помощью которых клиент сообщает серверу тип данных, находящихся в теле запроса, используемые наборы символов и другие сведения. Поскольку мы не впервые посещаем узел Google, в составе запроса передается также запись cookie — короткое сообщение с информацией о клиенте.

Заголовок ответа выглядит подобно приведенному ниже.

```
HTTP/1.x 302 Found
Location: http://www.google.co.uk/cxfer?c=PREF%3D:
 TM%3D1116601572:S%3DzFxPsBpXhZzknVMF&prev=/
Set-Cookie: PREF=ID=cabd38877dc0b6al:CR=1:TM=1116601572:
 LM=1116943140:S=fRfhD-u49xp9UE18;
 expires=Sun, 17-Jan-2038 19:14:07 GMT;
 path=/; domain=.google.com
Content-Type: text/html
Server: GWS/2.1
Transfer-Encoding: chunked
Content-Encoding: gzip
Date: Tue, 24 May 2005 17:59:00 GMT
Cache-Control: private, x-gzip-ok=""
```

Первая строка сообщает статус ответа. Код 302 означает перенаправление на другую страницу. В составе ответа мы получаем запись cookie для текущего сеанса, а также тип содержимого запроса (MIME-тип). Следующий запрос формируется потому, что в ответе содержится код перенаправления. На него будет получен ответ со следующим заголовком:

```
HTTP/1.x 200 OK
Cache-Control: private
```

```
Content-Type: text/html
Content-Encoding: gzip
Server: GWS/2.1
Content-Length: 1196
Date: Tue, 24 May 2005 17:59:00 GMT
```

Код состояния 200 означает успешную обработку, а в тело ответа включается код исходной страницы Google, предназначенной для отображения в окне браузера. Поле заголовка content-type сообщает браузеру о том, что данные представлены в формате HTML.

В рассмотренной выше функции sendRequest второй и третий параметры в большинстве случаев можно не указывать. По умолчанию для получения ресурса используется метод POST, причем параметры в теле запроса отсутствуют.

После передачи запроса функция сразу возвращает управление, не дожидаясь, пока запрос будет доставлен по сети серверу, а тот передаст ответ. Это хорошо с точки зрения интерактивного взаимодействия с пользователем, но как мы узнаем, что запрос уже обработан?

2.5.4. Использование фуниции обратного вызова для контроля запроса

Вторая часть решения задачи асинхронного взаимодействия — формирование в составе кода точки повторного входа, которая нужна для получения результатов обработки запроса. Обычно для этой цели используется функция обратного вызова — фрагмент кода, получающий управление тогда, когда результаты уже готовы. Так, функцией обратного вызова является уже знакомая нам window.onload.

Функции обратного вызова применяются в программах, управляемых событиями (именно по такому принципу и созданы пользовательские интерфейсы большинства приложений). Пользователь может нажать клавишу, щелкнуть мышью и выполнить другие действия в произвольный момент времени. Программист обеспечивает реакцию программы на эти действия путем написания соответствующих функций обратного вызова. Создавая JavaScript-код обработки событий, мы используем onkeypress, onmouseover и другие подобные свойства объекта. При написании кода функции обратного вызова мы применяем свойства onload и onreadystatechange.

Функцию обратного вызова onreadystatechange поддерживают браузеры Internet Explorer и Mozilla, поэтому мы будем использовать именно ее. (Браузер Mozilla также поддерживает функцию onload, которая более проста в использовании.) Несложный код обработчика, действующего по принципу обратного вызова, показан в листинге 2.10.

Листинг 2.10. Обработчик, работающий по принципу обратного вызова

```
var READY_STATE_UNINITIALIZED=0;
var READY_STATE_LOADING=1;
var READY_STATE_LOADED=2;
var READY_STATE_INTERACTIVE=3;
var READY_STATE_COMPLETE=4;
```

```
var req;
function sendRequest(url,params,HttpMethod){
 if (!HttpMethod){
 HttpMethod="GET";
 }
 req=getXMLHTTPRequest();
 if (req){
 req.onreadystatechange=onReadyStateChange;
 req.open(HttpMethod,url,true);
 req.setRequestHeader
 ("Content-Type", "application/x-www-form-urlencoded");
 req.send(params);
 }
}
function onReadyStateChange(){
 var ready=req.readyState;
 var data=null;
 if (ready==READY_STATE_COMPLETE){
 data=req.responseText;
 }else{
 data="loading...["+ready+"]";
 }
 II... выполнение действий с данными ...
```

Здесь мы изменили код функции sendRequest(), а именно сообщили объекту запроса о наличии обработчика. Кроме того, мы определили функцию обработчика. Чтобы не тратить время на придумывание имени, мы назвали ее onReadyStateChange().

Свойство readyState получает числовые значения. Чтобы упростить восприятие программы, мы определили для каждого из этих значений символьное имя. В данный момент нас интересует только значение 4, соответствующее завершению обработки запроса.

Заметьте, что объект запроса объявлен как глобальная переменная. Сейчас это упрощает работу с объектом XMLHttpRequest, но впоследствии, когда нам надо будет передавать несколько запросов одновременно, такое решение может стать источником проблем. Выход из этой ситуации мы покажем в разделе 3.1. А сейчас пришло время собрать воедино все фрагменты программы поддержки запроса к серверу.

2.5.5. Жизненный цикл процедуры поддержки запроса

Теперь мы имеем достаточно информации для того, чтобы полностью описать жизненный цикл процедуры загрузки документа. Полный код HTML-страницы, содержащей JavaScript-программу, приведен в листинге 2.11. Мы создаем экземпляр объекта XMLHttpRequest, сообщаем ему о том, что необходимо загрузить документ, а затем управляем процессом загрузки, используя функцию обратного вызова. В данном простом примере определяется узел DOM с идентификатором console, в который помещается информация о состоянии запроса.

Листинг 2.11. Загрузка документа с использованием объекта XMLHTTPRequest

```
<html>
<head>
<script type='text/javascript'>
var req=null;
var console=null;
var READY_STATE_UNINITIALIZED=O;
var READY_STATE_LOADING=1 ;
var READY_STATE_L0ADED=2 ;
var READY_STATE_INTERACTIVE=3;
var READY_STATE_COMPLETE=4;
function sendRequest(url,params,HttpMethod){
 if (!HttpMethod){
 HttpMethod="GET";
 }
 req=initXMLHTTPRequest() ;
 if (req){
 req.onreadystatechange=onReadyState;
 req.open(HttpMethod,url,true);
 req.setRequestHeader
 ("Content-Type", "application/x-www-form-urlencoded");
 req.send(params);
 }
}
function initXMLHTTPRequest(){
 var xRequest=null;
 // Инициализация объекта запроса
 if (window.XMLHttpRequest){
 xRequest=new XMLHttpRequest();
 } else if (window.ActiveXObject){
 xRequest=new ActiveXObject
 ("Microsoft.XMLHTTP");
 }
 return xRequest;
}
// Определение обработчика обратного вызова
function onReadyState(){
 var ready=req.readyState;
 var data=null;
 // Проверка readyState
 i f (ready==READY_STATE_COMPLETE){
 // Чтение данных ответа
 data=req.responseText;
 }else{
 data="loading...["+ready+"]";
 }
 toConsole(data);
}
function toConsole(data){
 if (console!=null){
 var newline=document.createElement("div");
 console.appendChild(newline) ;
 var txt=document.createTextNode(data) ;
 newline.appendChild(txt) ;
 }
```

```
window.onload=function(){
 console=document.getElementById('console')
 sendRequest("data.txt");
}
</script>
</head>
<body>
<div id='console'X/div>
</body>
</html>
```

Рассмотрим выходные данные программы, полученные с помощью Microsoft Internet Explorer и Mozilla Firefox. Заметьте, что последовательности состояний readyState отличаются, но конечные результаты совпадают. Важно помнить, что промежуточные состояния readyState различаются в разных браузерах (и даже в различных версиях одного и того же браузера). Ниже показаны выходные данные, полученные с помощью Microsoft Internet Explorer.

```
loading...[1]
loading...[1]
loading...[3]
Here is some text from the server!
```

Каждая строка соответствует активизации функции обратного вызова. В процессе загрузки документа она вызывается дважды для различных фрагментов данных, затем еще раз в состоянии READY_STATE_INTERACTIVE (в случае синхронного запроса на этом этапе управление можно вернуть пользовательскому интерфейсу). Последний вызов соответствует завершению загрузки; текст ответа сервера можно отображать в окне браузера.

Теперь рассмотрим данные, полученные посредством Mozilla Firefox 1.0.

```
loading...[1]
loading...[1]
loading...[2]
loading...[3]
Here is some text from the server!
```

Последовательность вызовов похожа на предыдущую, но имеет место дополнительный вызов, при котором значение readyState равно 2.

В данном примере мы получаем ответ в текстовом виде, используя для этой цели свойство aresponseText объекта XMLHttpRequest. Такой подход хорошо подходит для простых данных, но если сервер должен возвратить большой объем структурированной информации, лучше использовать свойство responseXML. Если ответ соответствует MIME-типу text/xml, будет получена структура DOM, которую можно опрашивать посредством свойств и функций, таких как getElementByld и childNodes, которые упоминались в разделе 2.4.1.

Мы рассмотрели основные "строительные блоки" Ajax, каждый из которых предоставляет полезные возможности, но реально мощность Ajax проявляется лишь при их объединении в единое целое.

2.6. Отличия Ajax от классических технологий

Несмотря на то что CSS, DOM, асинхронные запросы и JavaScript являются неотъемлемыми компонентами Ajax, эти технологии можно использовать независимо.

В главе 1 мы уже обсуждали различия между классическими Web-приложениями и приложениями, выполняющими те же функции, но созданными средствами Ajax. Сейчас необходимо снова затронуть эту тему. В классических Web-приложениях возможности, предоставляемые пользователю, определяются кодом, расположенным на стороне сервера. При переходе от одного документа к другому происходит загрузка целой страницы. Во время загрузки очередной страницы пользователь не может продолжать работу. В приложениях Ajax последовательность действий пользователя по крайней мере частично определяется программой, выполняющейся на стороне клиента. Взаимодействие клиента с сервером происходит в фоновом режиме, незаметно для пользователя.

Однако, помимо этих крайних случаев, существует множество промежуточных вариантов. Web-приложение может предоставлять ряд отдельных страниц в рамках классического подхода, причем на каждой из этих страниц допустимо использование CSS, DOM, JavaScript и асинхронных запросов. JavaScript-приложение может отображать окна, похожие на классические Web-страницы. Браузер реализует гибкую среду, допускающую использование в рамках одного приложение классического подхода и средств Ajax.

Ajax отличается от других подходов к созданию Web-приложений не используемыми технологиями, а моделью взаимодействия, которая становится возможной благодаря применению этих технологий. Модель взаимодействия Web плохо подходит для создания независимых приложений, и как только появилась возможность обойти ограничения этой модели, появились и новые перспективы в разработке интерфейсов.

Очевидны два уровня, на которых целесообразно применять Ajax и ряд ситуаций, в которых есть смысл обращаться к классическому подходу, основанному на использовании Web-страниц. Проще всего создавать на базе Ajax отдельные компоненты и включать их в состав Web-страницы посредством фрагментов сценариев. В качестве примеров подобных компонентов можно привести обновляемые данные о котировках акций, интерактивный календарь, окно для обмена сообщениями в реальном времени. Таким образом, на классических Web-страницах могут располагаться "островки", реализующие поведение, подобное интерфейсу обычных приложений для настольных систем (рис. 2.6). Большинство попыток обратиться к Ajax при разработке средств Google вполне укладываются в эту модель. Заполнение поля в Google Suggest и карта в Google Maps — хорошие примеры интерактивных элементов в составе статического документа.

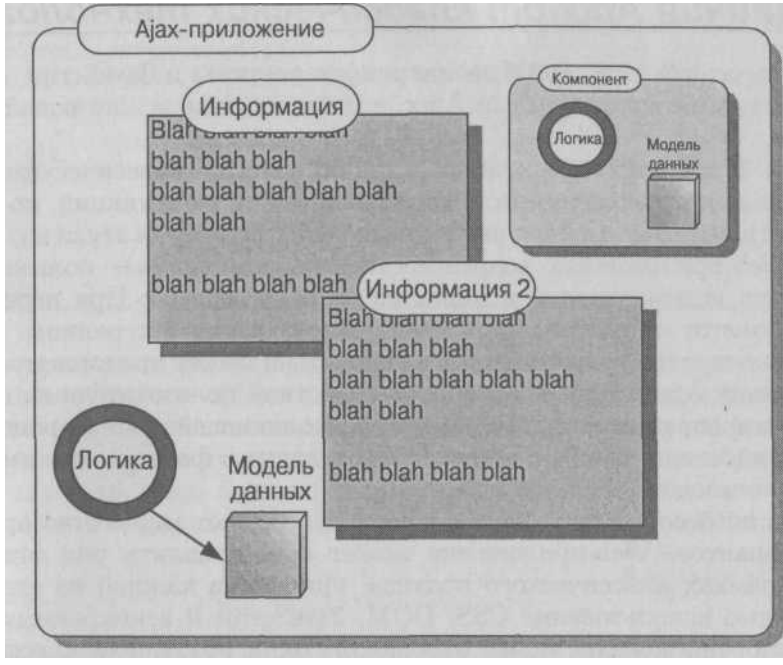


Рис. 2.6. Простое Ajax-приложение, которое представляет собой Web-страницу с "островками", реализующими интерактивные функции

Дальнейшим продвижением применения Ajax может стать создание системы, в которой фрагменты, ориентированные на приложение и документ, присутствуют на равных правах (рис. 2.7). Продукт, полученный в результате следования такому подходу, можно сравнить с приложением для настольной системы или даже с диспетчером окон. Этой модели соответствует Google GMail, где отдельные сообщения воспроизводятся как документы в составе интерактивной структуры.

Изучение новых технологий — приятное занятие. Несмотря на то что составные части Ajax уже известны разработчикам, интересно попробовать, как они будут работать совместно. Мы привыкли считать Web-приложение чем-то вроде повествования, в ходе которого мы ведем пользователя от одной странице к другой по предопределенному сценарию. Следуя новому подходу, мы предоставляем пользователю возможность лучше контролировать те вопросы, для решения которых и создавалось приложение.

Чтобы получить выгоду от дополнительной степени гибкости, которую обеспечивает новый инструмент, мы должны пересмотреть многие из стереотипов мышления, сложившихся за многие годы. Действительно ли HTML — единственное средство для ввода информации пользователем? Следует ли реализовывать все части пользовательского интерфейса в виде HTML-документов? Можно ли обращаться к серверу в ответ на такие действия пользователя, как нажатие клавиши и перемещение мыши, или надо по-прежнему щелкать мышью? Мир информационных технологий изменяет-

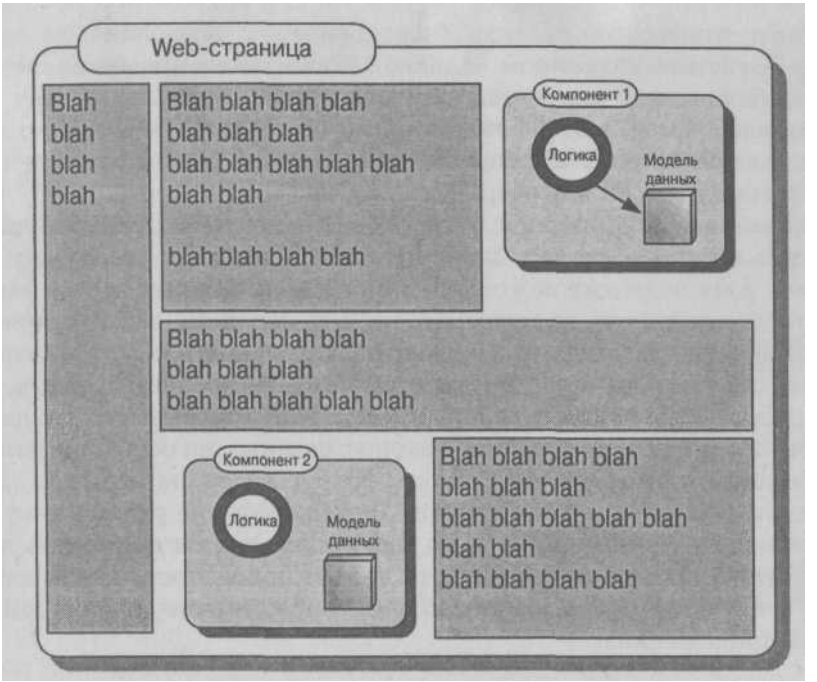


Рис. 2.7. Более сложное Ajax-приложение, представляющее собой интерактивную систему, в которой присутствуют элементы содержимого, ориентированного на документ. Эти элементы могут быть загружены или определены из программы

ся очень быстро, и все мы придаем большое значение приобретению новых навыков. Однако избавление от старых привычек не менее важно.

2.7. Резюме

В данной главе мы рассмотрели четыре составные части Ajax.

JavaScript — это мощный язык программирования общего назначения, незаслуженно получивший плохую репутацию инструмента, пригодного лишь для формирования окон с сообщениями, блокировки кнопок и загрузки изображений. В приложении Б содержится более подробное описание некоторых возможностей этого языка, но из примеров, приведенных в данной главе, можно получить представление о том, как можно использовать JavaScript для повышения практичности приложений.

CSS и DOM дополняют друг друга при формировании пользовательского интерфейса, причем позволяют разделять структуру и визуальное представление. Строгая структура документа упрощает работу программы с ним, а разделение функций важно для создания больших Ajax-приложений (этот вопрос будет подробнее рассмотрен в главах 3 и 4).

Вы узнали, как работать с объектом XMLHttpRequest и более старыми средствами: XmlDocument и IFrame. В настоящее время можно услышать мно-

го хвалебных отзывов об объекте XMLHttpRequest, позволяющем организовать взаимодействие с сервером. Однако I Frame также предоставляет подобные возможности, и в некоторых случаях удобно воспользоваться именно этим элементом. Зная, как работать с ними обоими, вы несомненно повысите свою квалификацию Web-разработчика. В главе 5 мы подробно обсудим взаимодействие клиента и сервера.

И наконец, мы рассмотрели объединение основных составляющих Ajax, в результате которого удалось получить нечто большее, чем сумму составных частей. Ajax подходит не только для решения частных задач, например, реализации компонентов, включаемых на Web-страницу, которая при отсутствии их была бы статической. Данный подход можно смело применять как основу для создания интерактивного пользовательского интерфейса, в котором содержались бы лишь отдельные "островки" статических данных. Для того чтобы средства Ajax перешли с второстепенных на основные роли, надо создать большой объем JavaScript-кода, который мог бы надежно работать в течение длительного периода времени. Чтобы успешно решить такую задачу, надо изменить привычный подход к созданию кода и применять приемы, обеспечивающие надежность, гибкость кода и пригодность его к сопровождению. В следующей главе мы рассмотрим способы упорядочения кода большого Ajax-приложения.

2.8. Ресурсы

Для тех, кто стремится глубже изучить каскадные таблицы стилей, мы рекомендуем узел CSS Zen Garden (http://www.csszengarden.com/), стили которого могут изменяться множеством способов, причем для получения результата не используются никакие другие средства, кроме CSS.

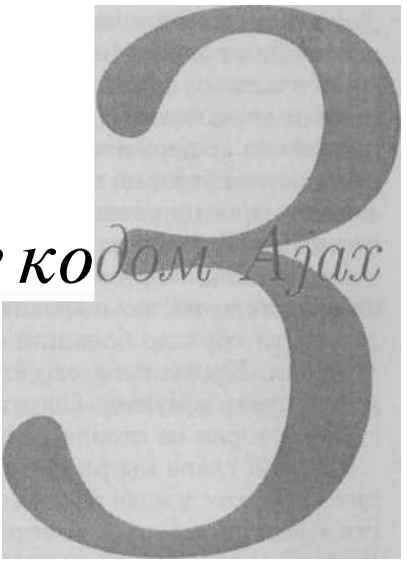
Эрик Мейер (Eric Meyer) интенсивно использует CSS. Посетите его Web-узел по адресу http://www.meyerweb.com/eric/css/. Blooberry (http://www.blooberry.com) — еще один превосходный ресурс для тех, кому необходима информация по CSS.

С первыми решениями Ajax, использующими элементы IFrame, можно ознакомиться по адресу http://developer.apple.com/internet/webcontent/iframe.html.

Расширение Mozilla LiveHttpHeaders можно найти по адресу http://livehttpheaders.mozdev.org/.

В книгах Денни Гудмена (Danny Goodman) *Dynamic HTML: The Definitive Reference* (O'Reilly, 2002) и *JavaScript Bible* (John Wiley, 2004) содержится обширный материал по работе с D0M. Кроме того, в них подробно описывается среда браузера.

На Web-узле W3Schools (http://www.w3schools.com/js/js_examples_3.asp) представлены интерактивные руководства по JavaScript.



Управление кодом Ajax

В этой главе...

- Разработка и сопровождение сложных клиентов Ajax
- Реструктуризация JavaScript-кода Ajax-приложения
- Использование образов разработки при создании Ajax-приложений
- Использование архитектуры "модель-представление-контроллер" при создании серверных программ в составе Ajax-приложений
- Общие сведения о библиотеках Ajax независимых производителей

В главе 2 мы обсудили основные технологии, составляющие инфраструктуру Ajax. Оказывается, что, пользуясь лишь давно известными нам средствами, можно написать превосходное Ajax-приложение, о котором мы давно мечтали. К сожалению, можно создать себе и массу проблем и получить в результате длительной работы запутанный клубок кода, HTML-элементов и стилей, непригодных для сопровождения. Не исключено, что однажды по непонятным причинам результат вашего труда вдруг окажется полностью неработоспособным. А может быть еще хуже. Приложение будет работать, пока вы стоите рядом, затаив дыхание. Если подобная ситуация возникнет с программой, написанной для себя лично, — это неприятно. А если таким результатом завершилась работа по заданию руководства и к тому же возникла необходимость изменить некоторые детали — это просто ужасно!

К счастью, подобные проблемы возникали уже на заре компьютерной эры и даже до ее начала. Разработчики нашли способ справляться со сложными проектами и держать в порядке коды большого объема. В этой главе мы рассмотрим базовые инструменты для работы с кодом, которые помогут вам создать Ajax-приложение, изменить его по требованию заказчика и при этом не задерживаться на работе после окончания рабочего дня.

Ajax отличается от DHTML не только способом применения уже известных технологий, но и масштабом их использования. В Ajax-приложении содержится гораздо больший объем JavaScript-кода, чем в классическом приложении. Кроме того, код присутствует в браузере в течение гораздо более длительного времени. Следовательно, возникает задача управления сложностью, которая не стояла перед специалистами, использовавшими DHTML.

В этой главе мы рассмотрим инструменты и приемы, позволяющие наводить порядок в коде программы. Важность этих средств зависит от сложности Ajax-приложения, которое вы создаете. Если вы собираетесь ограничиться созданием лишь простых продуктов, мы советуем вам пропустить материал книги вплоть до главы 9, в которой мы начнем рассмотрение программ, управляемых примерами. Если вы уже знакомы с принципами реструктуризации и образами разработки, перейдите к главам 4-6, в которых будет рассматриваться применение этих средств. В любом случае базовые понятия, рассматриваемые здесь, важны для управления JavaScript-кодом, и мы надеемся, что рано или поздно вы вернетесь к данной главе. В конце главы приведен обзор библиотек независимых производителей, ориентированных на Ajax, поэтому если вы ищете набор инструментов для своего проекта, обратитесь к разделу 3.5.

3.1. Порядок из хаоса

Основной прием, который мы будем применять для упорядочения кода приложения, — это *реструктуризация* (refactoring), т.е. процесс изменения кода, при котором выполняемые им функции не изменяются, но сам код становится более понятным. Наведение порядка имеет смысл уже само по себе, но, кроме того, выполнить подобную работу необходимо из соображений целесообразности.

Если код имеет ясную и понятную структуру, гораздо проще реализовывать новые функции, а также модифицировать и удалять уже существующие. Если код выполнен неаккуратно, он, может быть, и соответствует текущим требованиям, но никто из команды разработчиков не представляет, почему же он все-таки работает.

Практически каждому разработчику приходится реагировать на изменяющиеся условия. Как правило, приводить программу в соответствие новым требованиям приходится в сжатые сроки. В процессе реструктуризации код упорядочивается и становится пригодным к сопровождению. Реструктуризировав код, вы можете без страха изучать новые требования и выполнять их.

Элементарные действия по реструктуризации уже описывались в главе 2, когда речь шла о размещении JavaScript, HTML-кода и таблиц стилей в отдельных файлах. Однако JavaScript-код считается длинным, если он превышает 120 строк и объединяет в себе низкоуровневые функции (например, поддержку запроса серверу) с фрагментами кода, управляющими объектами документа. При работе над большим проектом иметь один JavaScript-файл (равно как и один файл со стилями) крайне неудобно. Гораздо лучше создавать небольшие фрагменты кода, удобные для чтения и модификации, каждый из которых выполняет конкретную задачу. Такой подход принято называть *распределением функций,* или *распределением ответственности* (separation of responsibilities).

Реструктуризация преследует и другую цель — идентификацию общих решений и приведение кода в соответствие конкретному образу разработки. Этот вопрос желательно обсудить подробнее.

3.1.1. Образы разработки

Если код соответствует некоторому хорошо зарекомендовавшему себя шаблону, есть все основания предполагать, что он будет удовлетворительно работать, хотя бы потому, что подобная задача уже была решена ранее. Разработчики обдумали все основные вопросы и решили большинство проблем. Если повезет, может оказаться, что существуют даже базовые программные средства, пригодные для повторного использования.

Такие шаблоны, отражающие опыт предшественников, принято называть *образами разработки.* Это понятие было введено в 1970-х годах в архитектуре, а в последние годы его начали применять и разработчики программного обеспечения. Образы разработки широко применяются при создании Java-программ, предназначенных для выполнения на сервере, а корпорация Microsoft активно внедряет его в .NET Framework. Данный термин звучит внушительно, напоминает нечто, относящееся к "чистой науке", и бывает, что кто-то, не потрудившись разобраться в данном вопросе, применяет его лишь для того, чтобы произвести впечатление на собеседников. Многие неправильно истолковывают это понятие. На самом деле образ разработки — это описание способа решения задачи, связанной с разработкой программы. Благодаря применению образов разработки ряд абстрактных решений получили имена, а это упрощает их обсуждение и понимание.

Образы разработки важны при реструктуризации, поскольку позволяют кратко описать основную цель этого процесса. Инструкция типа "оформить этот фрагмент кода в виде объектов, инкапсулирующих выполнения действия пользователем и позволяющих отменить это действие" слишком многословна, поэтому ее трудно удержать в голове, выполняя реструктуризацию кода. Если же вы скажете, что необходимо привести код в соответствие с образом разработки Command, то задача будет поставлена гораздо точнее и обсуждать ее станет намного удобнее.

Если вы давно занимаетесь разработкой серверных программ на Java, то, вероятно, скажете, что мы не сообщили вам ничего нового. Другие, возможно, подумают, что мы агитируем их применять нечто вроде блок-схем. В любом случае мы готовы услышать вопрос: какое отношение имеет вышесказанное к Ajax? Ответ прост: большое! Давайте выясним, какие выгоды может получить от реструктуризации программист, использующий Ajax.

3.1.2. Реструктуризация и Ajax

Мы уже говорили о том, что Ajax-приложения, как правило, содержат большой объем JavaScript-кода, который длительное время сохраняется в браузере.

В классическом Web-приложении сложный код находится на сервере и образы разработки применяются к серверным программам PHP, Java или .NET. Используя Ajax, мы должны применить те же средства к клиентскому коду.

Можно даже утверждать, что JavaScript-код больше нуждается в реструктуризации, чем программы, написанные на Java и C#. Несмотря на синтаксис, JavaScript больше напоминает такие языки, как Ruby, Python и даже Common Lisp, чем Java или C#. Он обеспечивает необычайную гибкость и богатые выразительные средства. Квалифицированный разработчик несомненно воплотит эти возможности в программу высокого качества, однако средний программист может не справиться с ними, в результате пострадает надежность программы. Языки, ориентированные на создание корпоративных приложений, такие как Java и C#, напротив, ориентированы на команду специалистов средней квалификации, состав которой может изменяться.

Опасность создать посредством JavaScript запутанный и сложный для восприятия код достаточно высока, и если этот код ориентирован не на одну Web-страницу, а на целое приложение, необходимо принимать меры для исключения такой возможности либо для уменьшения ее вероятности. Поэтому мы настаиваем на реструктуризации Ajax-программ и считаем, что она должна производиться более тщательно, чем это делается при работе с такими относительно "безопасными" языками, как Java и C#.

3.1.3. Во всем надо знать меру

Приступая к реструктуризации, необходимо помнить, что она, как и образы разработки, является средством, а не целью. Выполнять ее надо лишь До тех пор, пока это оправдано. Если вовремя не остановиться, произойдет

"блокировка разработки из-за анализа" — ситуация, при которой реализация приложения откладывается на неопределенное время из-за бесконечного поиска наилучшего решения. Разработчик пытается еще больше повысить гибкость системы, довести ее структуру до идеального состояния, чтобы будущие изменения не вызывали никаких затруднений. Следует ли говорить, что до изменений дело может вовсе не дойти, так как под угрозу ставится существование самой программы.

Подобную ситуацию описал Эрик Гамма (Erich Gamma), признанный специалист по образам разработки, в недавнем интервью (ссылка на него приведена в конце главы). Подобно тому как разработчик включает в свою программу целочисленные переменные, строки и массивы лишь там, где они необходимы, применять образы разработки надо только тогда, когда они нужны.

Э. Гамма считает, что наилучшим способом применения образов разработки является реструктуризация. Следует написать код так, чтобы он лишь выполнял поставленную задачу, а затем, приводя его в соответствие с образом разработки, можно решить типичные проблемы. Если у вас уже написан большой объем кода и вы собираетесь привести его в порядок или сделать то же самое с программой, написанной другим программистом, то в обычных условиях вам придется приложить все силы и все равно успех не будет гарантирован. К счастью, применять шаблоны разработки можно к коду любой сложности и любого качества. В следующем разделе мы рассмотрим фрагменты, приведенные в качестве примеров в главе 2, и выясним, как можно преобразовать их путем реструктуризации.

3.1.4. Реструктуризация в действии

На первый взгляд, преимущества реструктуризации кажутся очевидными, но программисты-практики, наверное, захотят увидеть их на конкретном примере. Потратим немного времени, чтобы применить процедуру реструктуризации к Ajax-коду, который был представлен в листинге 2.11. Как вы помните, мы определили функцию sendRequest(), предназначенную для передачи запросов серверу. Для извлечения объекта XMLHttpRequest эта функция обращается к initHttpRequest(), а обработка ответа производится посредством функции обратного вызова onReadyState(). Объект XMLHttpRequest был помещен в глобальную переменную, что позволило функции обратного вызова использовать его. Обработчик ответа, действующий по принципу обратного вызова, анализировал состояние объекта запроса и генерировал информацию, предназначенную для отладки.

Код, представленный в листинге 2.11, выполняет необходимые действия, но повторно использовать его будет непросто. Обычно,-передавая запрос серверу, мы собираемся анализировать ответ и, в зависимости от его содержания, выполнять действия, необходимые для решения текущей задачи. Для того чтобы включить в имеющийся код бизнес-логику, нам пришлось бы модифицировать функцию onReadyState().

Наличие глобальной переменной также может привести к возникновению проблем. Если надо будет одновременно передать несколько запросов серверу, нам придется указывать для каждого из них отдельный обработчик. Кроме того, получив набор ресурсов, необходимо будет следить, какие из них должны отображаться взамен существующих, а какие не надо учитывать.

В объектно-ориентированном программировании стандартным решением задач подобного рода является инкапсуляция требуемых функциональных возможностей в составе объекта. Объектные средства JavaScript позволяют нам сделать это. Мы назовем наш объект ContentLoader, поскольку он загружает данные с сервера. Каким должен быть этот объект? Наверное, он должен создаваться на основе URL ресурса, которому передается запрос. Нам также надо указать ссылку на произвольный обработчик обратного вызова, который будет выполнен при успешной загрузке документа. Необходима еще одна ссылка на обработчик, который получит управление при наличии ошибки. Обращение к объекту может выглядеть следующим образом:

```
var loader=new net.ContentLoader('mydata.xml', parseMyData);
```

где parseMyData — функция обратного вызова, соответствующая успешной загрузке документа. Код объекта ContentLoader показан в листинге 3.1. В нем были применены новые решения, которые мы обсудим ниже.

Листинг 3.1. Объект ContentLoader

```
//1  Объект пространства имен
var net=new Object();
net.READY_STATE_UNINITIALIZED=0;
net.READY_STATE_LOADING=1;
net.READY_STATE_LOADED=2;
net.READY_STATE_INTERACTIVE=3;
net.READY_STATE_COMPLETE=4;
// 2 Конструктор
net.ContentLoader=function(url,onload,onerror){
 this.url=url;
 this.req=null;
 this.onload=onload;
 this.onerror=(onerror) ? onerror : this.defaultError;
 this.loadXMLDoc(url);
}
net.ContentLoader.prototype={
 // 3 Переименованная функция initXMLHttpRequest
 loadXMLDoc:function(url){
 // 4 Реструктуризированная функция loadXML
 if (window.XMLHttpRequest){
 this.req=new XMLHttpRequest();
 } else if (window.ActiveXObject){
 this.req=new ActiveXObject("Microsoft.XMLHTTP");
 }
 if (this.reqH
 try{
 var loader=this;
 this.req.onreadystatechange=function(){
 loader.onReadyState.call(loader);
 }
```

```
// 5 Реструктуризированная функция sendRequest
this.req.open('GET•,url,true);
this.req.send(null);
}catch (err){
this.onerror.call(this);
}
}
},
// 6 Реструктуризированная функция обратного вызова
onReadyState:function(){
var req=this.req;
var ready=req.readyState;
if (ready==net.READY_STATE_COMPLETE){
var httpStatus=req.status;
if (httpStatus==200 || httpStatus==O){
this.onload.call(this);
}else{
this.onerror.call(this);
}
}
},
defaultError:function(){
alert("error fetching data!"
+"\n\nreadyState:"+this.req.readyState
+"\nstatus: "+this.req.status
+"\nheaders: "+this.req.getAHResponseHeaders());
}
}
```

Первое новшество — это единственная глобальная переменная net 1, которая содержит все необходимые ссылки. При этом вероятность конфликта имен уменьшается, а код, имеющий отношение к сетевым запросам, локализуется в одном месте.

Для нашего объекта предусмотрен конструктор 2, которому передаются три параметра, но только два из них являются обязательными. При установке обработчика ошибок мы проверяем, содержит ли параметр значение null, и в случае положительного результата проверки предпринимаем действия по умолчанию. Возможность передавать функции переменное число параметров и передавать функцию при первом обращении к классу может показаться странной для специалистов, привыкших создавать программы в рамках объектного подхода. Однако JavaScript допускает подобные решения. Свойства данного языка рассмотрены в приложении Б.

Значительную часть кода функций initXMLHttpRequest() 4 и sendRequest() 5 мы поместили в состав объекта и переименовали измененный код. Новое имя призвано отражать более широкую область применения. Функция, объединяющая возможности initXMLHttpRequest() и sendRequest(), теперь называется loadXMLDoc 3. Для нахождения объекта XMLHttpRequest и инициализации запроса мы используем те же подходы, что и ранее, но это никак не влияет на действия программиста, использующего Данный объект. Функция обратного вызова, onReadyState() 6, в основном

осталась такой же, как и в листинге 2.11. Мы лишь заменили обращения к консоли на вызовы функций onload и onerror. Эти действия также могут показаться несколько непривычными для многих разработчиков, поэтому мы рассмотрим их подробнее. Функции onload и onerror представляют собой объекты Function, а Function.call() — это метод данного объекта. Первый параметр, Function.call(), представляет собой контекст функции, и в теле функции на него можно ссылаться с помощью ключевого слова this.

Написать обработчик обратного вызова, передаваемый объекту ContentLoader, очень легко. Если нам надо обратиться к любому свойству ContentLoader, например XMLHttpRequest или url, это можно также сделать с помощью ключевого слова this. Например:

```
function myCallBacM ) {
 alert(
 this.url
 +" loaded! Here's the content:\n\n"
 +this.req.responseText
 );}
```

Очевидно, что, для того, чтобы написать код объекта, подобного рассмотренному выше, необходимо знать особенности языка JavaScript, но после создания объекта его можно использовать, не задумываясь о внутренней структуре.

Данный пример демонстрирует преимущества реструктуризированной программы. Мы "спрятали" внутри объекта сложные фрагменты кода, представив потенциальному пользователю внешние элементы, при работе с которыми не возникает трудностей. Фрагменты, разобраться с которыми может только квалифицированный специалист, теперь изолированы от остальной части программы. Если возникнет необходимость модификации кода, она будет выполнена локально, и не придется анализировать всю программу в поисках фрагментов, в которые надо вносить изменения.

Мы рассмотрели лишь основные принципы реструктуризации и их использование на практике. В следующем разделе мы обсудим более сложные проблемы, возникающие при создании Ajax-программ, и выясним, как можно решить их, используя ту же реструктуризацию. Попутно мы опишем ряд полезных приемов, которые будем применять в следующих главах и которые смогут помочь вам при работе над собственными проектами.

3.2. Варианты применения реструктуризации

В следующих разделах будут рассмотрены вопросы, часто возникающие при разработке приложений Ajax, и предложены ответы на них. В каждом из этих случаев найти решение задачи помогает реструктуризация. В процессе обсуждения вопросов мы выделим элементы, пригодные для повторного использования.

В литературе, посвященной образам разработки, принято описывать проблему, рассматривать ее решение, а затем обсуждать сопутствующие вопросы. Не будем отступать от этих традиций.

3.2.1- Несоответствие браузеров: образы разработки Fagade и Adapter

Спросите любого разработчика Web-приложений, какие проблемы он считает главными, и каждый, будь он программистом, дизайнером, художником или специалистом другого профиля, ответит, что в первую очередь необходимо обеспечить корректное отображение данных различными браузерами. Большинство вопросов функционирования Web регулируется стандартами, и производители браузеров в основном пытаются следовать им. Однако иногда стандарты допускают разночтение, что отражается в их реализации; в некоторых случаях производители расширяют стандарты в удобном для себя направлении, кроме того, в браузерах, как и в любых сложных программах, всегда есть ошибки.

Разработчикам JavaScript-программ очень часто приходится выяснять тип браузера, в котором отображается документ, и наличие того или иного объекта. Рассмотрим очень простой пример.

Обработка элементов DOM

Как было сказано в главе 2, Web-страница представляется JavaScript-программе в виде древовидной структуры DOM, элементы которой соответствуют элементам HTML-документа. При обработке дерева DOM из программы часто возникает необходимость определять позицию элемента в окне браузера. К сожалению, производители браузеров уже давно предоставляют для этой цели нестандартные методы, затрудняя перенос кода из одной клиентской программы в другую. В листинге 3.2 показан упрощенный вариант одной из функций библиотеки Майка Фостера (подробнее о ней — ниже, в разделе 3.5). Эта функция выполняет все необходимые действия для определения позиции левого края компонента страницы, соответствующей DOM-элементу e, который передается в качестве параметра.

Листинг 3.2. Функция getLeft (}

```
function getLeft(e){
 if(!(e=xGetElementById(e))){
 return 0;
 }
 var css=xDef(e.style);
 if (ess && xStr(e.style.left)) {
 iX=parselnt(e.style.left);
 if(isNaN(iX)) iX=0;
 }else if(ess && xDef(e.style.pixelLeft)) {
 iX=e.style.pixelLeft;
 }
 return iX;
 }
```

В различных браузерах предусмотрены разные способы определения позиции, соответствующей узлу DOM. В стандарте CSS2, разработанном W3C, описано свойство style.left. Его значением является строка, содержащая

числовую величину и единицы измерения, например ЮОрх. Поддерживаются не только пиксели, но и другие единицы. Свойство style.pixelLeft, напротив, допускает указание только числа; при этом предполагается, что значение задано в пикселях. Данное свойство поддерживается только Microsoft Internet Explorer. Метод getLef t (), рассматриваемый здесь, определяет, поддерживаются ли CSS, а затем проверяет оба значения, начиная с того, которое предусмотрено стандартом W3C. Если ни одно из значений не найдено, метод возвращает нулевую величину, предусмотренную по умолчанию. Заметьте, что здесь не определяется имя и версия браузера, а применяется более надежный способ обнаружения объектов, который рассматривался в главе 2.

Написание подобных функций, учитывающих особенности каждого браузера, — рутинная и утомительная работа, но после ее окончания программист может продолжать написание приложения, не уделяя внимания конкретной проблеме. Существуют библиотеки, авторы которых уже выполнили необходимую работу. Готовые надежные функции для определения позиции элемента DOM и решения других подобных задач могут существенно ускорить создание пользовательского интерфейса для Ajax-приложения.

Обращение к серверу

О проблеме несовместимости браузеров мы уже говорили в главе 2. Производители клиентских Web-программ предоставляют нестандартные средства получения объекта XMLHttpRequest, используемого для организации асинхронного взаимодействия с сервером. Если мы хотим скопировать XML-документ с сервера, необходимо решить, какое средство использовать для этого.

Internet Explorer выполняет необходимые действия только в том случае, если мы обратимся к компоненту ActiveX, а в браузерах Mozilla и Safari для этой цели используется встроенный объект. Об этих различиях "знает" только фрагмент кода, отвечающий за загрузку XML-документа. Как только программа получит объект XMLHttpRequest, она будет работать независимо от того, какое именно решение реализовано в конкретном браузере. Код, обращающийся к объекту, не должен зависеть от того, используется ли ActiveX или подсистема браузера. Все, что ему надо, — это конструктор net.ContentLoader().

Образ разработки Facade

Как для getLeft(), так и для new net.ContentLoader() код, предназначенный для обнаружения объектов, сложен, и создавать его утомительно. Определяя функцию, скрывающую его, мы делаем остальную часть программы эолее простой для восприятия. В этом состоит базовый принцип реструктуризации — *не повторяться* (don't repeat yourself — DRY). Если окажется, что код, предназначенный для обнаружения объектов, работает некорректно, ошибку надо будет исправить в одном месте, а не искать, где в программе эпределяются координаты элемента DOM или формируется запрос.

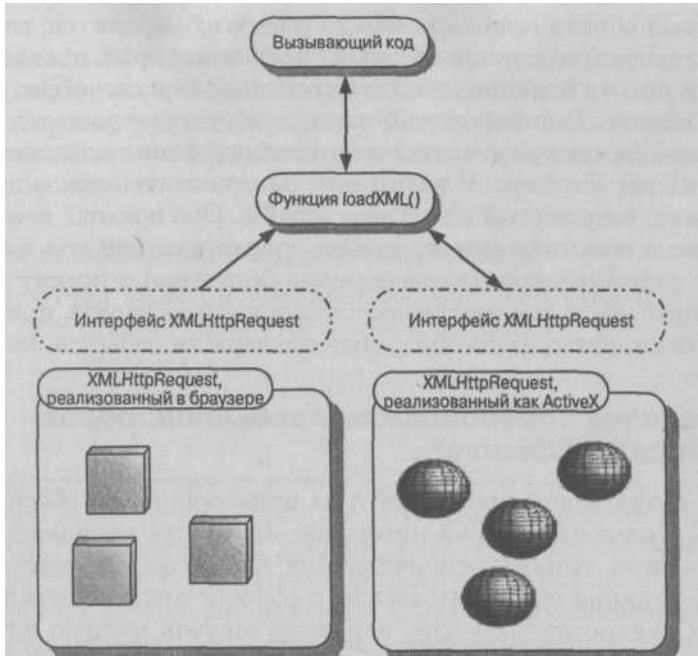


Рис. 3.1. Условное представление образа разработки Facade, используемого для работы с объектом XMLHttpRequest на различных браузерах. Функция loadXML() лишь использует объект XMLHttpRequest и не "заботится" о его реализации. Сама реализация может быть очень сложной, но вызывающей функции предоставляются только базовые элементы

Другими словами, поступив подобным образом, мы используем образ разработки Facade, который предоставляет одну точку доступа к функциональным средствам, реализованным различными способами. Например, объект XMLHttpRequest обеспечивает полезные возможности, а использующее его приложение не должно "заботиться" о том, как они реализованы (рис. 3.1).

Во многих случаях бывает необходимо упростить доступ к подсистеме. Например, при получении координаты левого края элемента надо учитывать, что CSS позволяет задавать значения в пикселях, пунктах, дюймах и других единицах измерения. В нашем случае такое разнообразие представлений лишь затрудняет работу. Функция getLeft(), приведенная в листинге 3.2, применима лишь до тех пор, пока в качестве единиц измерения используются пиксели. Упрощение подсистемы — это еще один результат, достигаемый посредством образа разработки Fagade.

Образ разработки Adapter

С образом разработки Fagade непосредственно связан образ Adapter. В этом случае мы также работаем с двумя подсистемами, выполняющими аналогичные функции. В качестве примера можно привести варианты получения объекта XMLHttpRequest в продуктах Microsoft и Mozilla. Вместо того чтобы

создавать новый образ Fagade для каждого из этих вариантов, мы реализуем над одной из подсистем дополнительный уровень, который предоставляет тот же API, что и другая подсистема. Этот уровень, как и сам образ разработки, [называется Adapter. В библиотеке Sarissa, которую мы рассмотрим в разделе 3-5.1, образ Adapter применяется для преобразования компонента ActiveX в браузере Internet Explorer. В результате он становится похожим на объект xMLHttpRequest, встроенный в браузере Mozilla. Оба подхода имеют право на существование и помогают интегрировать существующий код или продукты независимых разработчиков (включая сами браузеры) в проект Ajax.

Теперь перейдем к рассмотрению следующего варианта применения образов разработки, связанного с моделью поддержки событий JavaScript.

3.2.2. Управление обработчиками событий: образ разработки Observer

Невозможно создать нетривиальное Ajax-приложение, не обеспечив работу с событиями. Пользовательский интерфейс JavaScript управляется событиями, и асинхронные запросы, типичные для Ajax, требуют интенсивного использования функций обратного вызова и обработчиков событий. В простых приложениях конкретное событие, например щелчок мышью или получение данных с сервера, может быть обработано с помощью одной функции. По мере усложнения программы и увеличения ее размера необходимо выделять несколько подсистем и событий, причем в ряде случаев заинтересованные подсистемы могут оповещать сами себя о необходимости обработки. Рассмотрим эти вопросы подробнее.

Использование нескольких обработчиков событий

Очень часто фрагмент JavaScript-кода определяют как функцию window, onload, чтобы он получал управление тогда, когда документ полностью загружен (и, следовательно, древовидная структура DOM сформирована). Предположим, что нам необходим элемент DOM, который после загрузки страницы через определенные интервалы времени отображал бы данные, полученные с сервера. JavaScript-код, координирующий загрузку информации и ее отображение, должен иметь ссылку на узел DOM. Получить ее можно в обработчике window.onload.

```
window.onload=function(){
 displayDiv=document.getElementById('display');
)
```

Пока все хорошо. Предположим теперь, что нам также необходимо отображать новости (соответствующая программная реализация будет рассмотрена в главе 13). Код, контролирующий отображение новостей, также должен получить ссылки на некоторые элементы DOM. Для этого надо определить обработчик window.onload.

```
window.onload=function(){
 feedDiv=document.getEleraentById('feeds');
}
```

Проверим оба фрагмента кода на отдельных страницах и убедимся, что они работают корректно. Однако при их объединении вторая функция window, onload переопределяет первую, в результате чего возникают ошибки. Проблема в том, что с объектом окна можно связать только одну функцию onload.

Ограничения объединенного обработчика событий

Второй обработчик событий замещает первый. Этого можно избежать, реализовав все действия по обработке в одной функции.

```
window.onload=function()  {
 displayDiv=document.getElementById('display');
 feedDiv=document.getElementById('feeds');
}
```

Такой подход допустим в нашем конкретном примере, но он приводит к усложнению кода; теперь процессы отображения данных и просмотра новостей становятся зависимыми друг от друга. Если же мы будем иметь дело не с двумя системами, а с десятью или даже с двадцатью, и каждому потребуется ссылка на несколько элементов DOM, то объединенный обработчик станет неуправляемым. Добавление и удаление отдельных компонентов будет сопровождаться появлением ошибок, т.е. возникнет та ситуация, о которой мы говорили в начале главы: все будут бояться внести хотя бы минимальные изменения в состав кода, чтобы не разрушить его окончательно. Давайте прибегнем к реструктуризации и определим отдельную функцию для каждой подсистемы.

```
window.onload=function(){
 getDisplayElements();
 getFeedElements();
}
function  getDisplayElements(){
 displayDiv=document.getElementById('display' ) ;
}
function  getFeedElements(){
 feedDiv=document.getElementByld('feeds' ) ;
}
```

Теперь код стал более понятным: в обработчике window.onload() каждой подсистеме соответствует одна строка, однако объединенная функция все еще остается слабым местом и способна доставить неприятности. В следующем разделе мы рассмотрим альтернативное решение. Оно более сложное, но обеспечивает большую надежность.

Образ разработки Observer

Иногда бывает полезно задать себе вопрос: какой компонент отвечает за то или иное действие? В нашем случае составная функция возлагает на объект window ответственность за получение ссылок на элементы DOM. Теперь объект окна должен знать, какая из подсистем присутствует на странице. В идеале каждая подсистема должна сама отвечать за получение ссылок.

В этом случае, если подсистема присутствует, она получит ссылки, в противном случае действия по определению ссылок выполняться не будут.

Для того чтобы обеспечить разделение ответственности, мы дадим возможность подсистемам регистрироваться для оповещения. Для этого каждая подсистема должна указать функцию, которая будет вызвана при наступлении события onload. Реализуется такое решение следующим образом:

```
window.onloadListeners=new  Array();
window.addOnLoadListener(listener){
 window.onloadListeners[window.onloadListeners.length]=listener;
}
```

После загрузки окна объект window должен лишь осуществить перебор элементов массива и вызывать по очереди каждую функцию.

```
window.onload=function(){
 for(var  i=0;i<window.onloadListeners.length;i++){
 var  func=window.onlloadListeners[i] ;
 func.callO ;
  }
}
```

Если каждая подсистема будет создана с использованием данного подхода, мы обеспечим работу с ними, не объединяя фрагменты кода. Конечно, при этом существует опасность некорректно переопределить window. onload, что приведет к разрушению системы. Однако, поскольку "опасный" фрагмент лишь один, мы можем позаботиться, чтобы это не произошло.

Следует заметить, что новая модель событий W3C допускает существование нескольких обработчиков событий. Однако эта модель по-разному реализована в различных браузерах, поэтому мы предпочли сформировать собственный обработчик на базе знакомой модели событий JavaScript. Подробнее данный вопрос будет рассмотрен в главе 4.

В данном случае мы реструктуризируем код для того, чтобы он соответствовал образу разработки Observer. Согласно данному образу определяется объект Observable (в данном случае это встроенный объект окна) и набор обработчиков Observer или Listener, которые могут регистрироваться в объекте Observable (рис. 3.2).

При использовании образа Observer ответственность разделяется между источником события и обработчиком. Обработчики отвечают за свою регистрацию и ее отмену. На источник события возлагается ответственность за поддержку списка зарегистрированных обработчиков и оповещение о наступлении события. Данный образ давно используется при программировании пользовательских интерфейсов, управляемых событиями. Мы вернемся к нему в главе 4, когда будем подробно обсуждать события JavaScript. Как вы увидите, механизм событий может быть использован независимо от обработки браузером действий пользователя с мышью и клавиатурой.

Теперь перейдем к рассмотрению следующей проблемы, которая также может быть решена путем реструктуризации.

Рис. 3.2. Разделение ответственности согласно образу разработки Observer. Объекты Observer, которые должны оповещаться о событии, могут регистрироваться в объекте Observable или отменять регистрацию. Все зарегистрированные обработчики получают информацию о наступлении события

3.2.3. Повторное использование обработчиков событий: образ разработки Command

Очевидно, что при работе с большинством приложений пользователь сообщает им, какие действия надо выполнить (обычно он щелкает для этого мышью или нажимает клавиши на клавиатуре), а приложения следуют полученным инструкциям. В простых программах может существовать только один способ выполнения требуемых операций, но сложные программы обычно разрабатываются так, чтобы пользователь смог получить требуемый результат различными способами.

Компонент, реализующий кнопку

Предположим, что в нашем приложении предусмотрен элемент DOM, который выглядит на экране как кнопка. После щелчка на этой кнопке производятся некоторые вычисления и результаты отображаются в HTML-таблице. Нам необходимо определить функцию обработки щелчка мышью на кнопке. Эта функция имеет приблизительно следующий вид:

```
function buttonOnclickHandler(event){
 var data=new Array();
 data[0]=6;
 data[1]=data[0]/3;
 data[2]=data[0]*data[1]+7;
 var newRow=createTableRow(dataTable);
 for (var i=0;Kdata.length;i++){
 createTableCell(newRow,data[i]);
 }
}
```

Мы предполагаем, что переменная dataTable ссылается на существующую таблицу и что в функциях createTableRow() и createTableCell() корректно реализованы действия со структурой DOM. Необходимо заметить, что в реальном приложении вычисления могут занимать длительное время, а про-

грамма, выполняющая их, может содержать сотни строк кода. Обработчик связывается с кнопкой следующим образом:

```
buttonDiv.onclick=buttonOnclickHandler;
```

Поддержка различных типов событий

Теперь предположим, что приложение реализуется средствами Ajax. Мы обращаемся к серверу за данными и, получив их, выполняем вычисления и заполняем таблицу. Сейчас не будем говорить о том, как реализовать повторяющиеся обращения к серверу, предположим лишь, что в нашем распоряжении есть объект poller. В нем предусмотрены действия с объектом XMLHttpRequest, а обработчик onreadystatechange по окончании загрузки данных с сервера вызывает функцию onload. Чтобы не углубляться в детали вычислений и отображения данных, будем считать, что все необходимые операции реализованы в составе вспомогательных функций.

```
function buttonOnclickHandler(event){
 var data=calculate();
 showData(dataTable,data);

function ajaxOnloadHandler(){
 var data=calculate();
 showData(otherDataTable,data);

function calculate(){
 var data=new Array();
 data[0]=6;
 data[1]=data[0]/3;
 data[2]=data[0]*data[1]+7;
 return data;

function showData(table,data){
 var newRow=createTableRow(table);
 for (var i=0;i<data.length;i++){
 createTableCell(newRow,data[i]) ;


buttonDiv.onclick=buttonOnclickHandler;
poller.onload=ajaxOnloadHandler;
```

Многие действия скрыты в функциях calculate() и showData(), а принцип установки обработчиков onclick и onload уже знаком вам.

Мы получили более полное разделение бизнес-логики и кода, ответственного за обновление пользовательского интерфейса. Как и ранее, мы воспользовались уже готовым решением. На этот раз нам помог образ разработки Command. В объекте Command определяются действия любой сложности, после чего сам объект легко передается различным элементам пользовательского интерфейса. В классических объектно-ориентированных языках объект Command обычно создается на основе некоторого базового класса или интерфейса. Мы поступим несколько другим способом — будем рассматривать функции JavaScript как объекты Command, обеспечивающие требуемую степень абстрагирования.

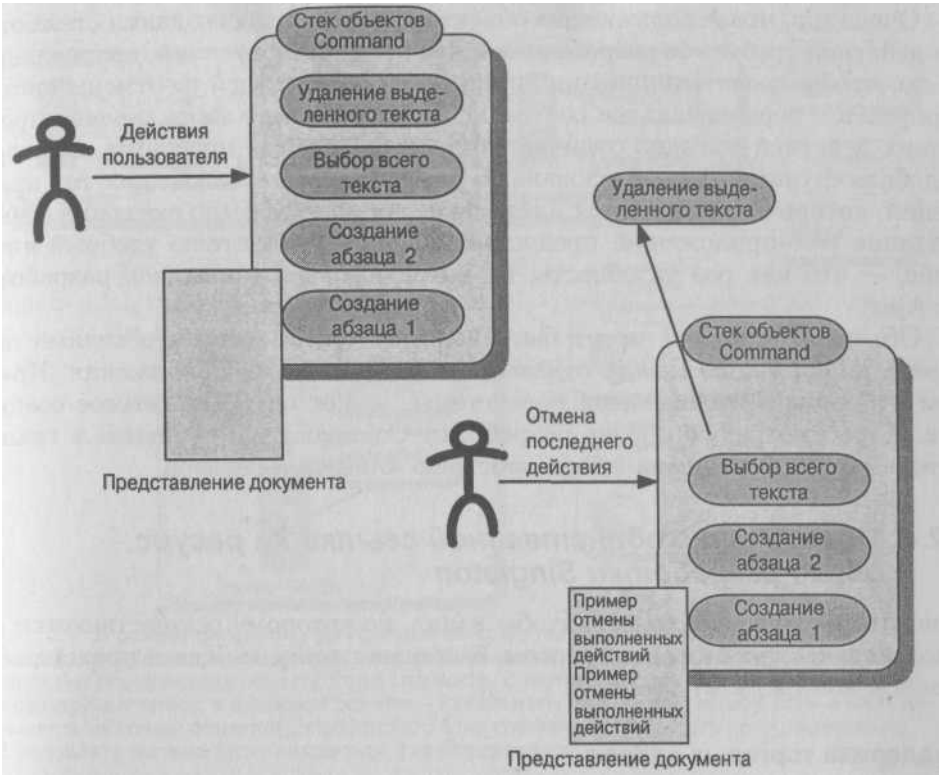


Рис. 3.3. Использование образа разработки Command для реализации универсального стека отмены действий в текстовом процессоре. Все действия пользователя представляются как объекты Command и могут быть легко отменены

Оформление всех действий пользователя в виде объекта Command может показаться сложным решением, однако затрата усилий окупается. Если все действия пользователя объединены в объекте Command, мы можем без труда связать с ними другие стандартные функции. При обсуждении образа разработки Command в качестве примера предоставляемых им возможностей чаще всего рассматривается добавление метода undo (), предполагающего отмену результатов действий в пределах всего приложения. При использовании образа разработки Command объекты, инкапсулирующие действия пользователя, записываются в стек, а при активизации кнопки, предназначенной для отмены действий, извлекаются из стека (рис. 3.3).

Каждый новый объект помещается в вершину стека, в результате появляется возможность пошаговой отмены действий. Процесс создания документа представляет собой не что иное, как последовательность действий. В какой-то момент пользователь может выделить весь текст документа и случайно нажать клавишу <Delete>. При вызове функции undo() извлекается верхний элемент стека и вызывается его метод undo (), который возвращает удаленный текст. Следующий вызов данной функции отменит выбор текста и т.д.

Очевидно, что использование объекта Command для создания стека отмены действий требует от разработчика дополнительных усилий, направленных на то, чтобы после выполнения команды и последующей ее отмены система вернулась в первоначальное состояние. Однако возможность отмены предыдущих действий выгодно отличает приложение от тех программ, в которых подобная функция не реализована. В особенности это валено для тех приложений, которые используются длительное время. Как было сказано в главе 1,1 создание Web-приложений, предоставляющих пользователю удобный интерфейс, — это как раз та область, на которую ориентировались разработчики Ajax.

Объекты Command могут быть полезны тогда, когда необходимо передавать информацию между отдельными подсистемами приложения. Примером "границы", разделяющей подсистемы, может служить сетевое соединение. К рассмотрению образа разработки Command мы вернемся в главе 5, которая будет посвящена взаимодействию клиента и сервера.

3.2.4. Обеспечение единственной ссылки на ресурс: образ разработки Singleton

В некоторых случаях валено, чтобы канал, по которому осуществляется взаимодействие с некоторым ресурсом, был единственным. Как и прежде, обратимся к конкретному примеру.

Поддержка торговых сделок

Предположим, что наше Ajax-приложение работает с биржевой информацией: позволяет пользователю делать реальные покупки и продажи, выполняет необходимые вычисления, а также обеспечивает режим имитации биржевой деятельности. Мы определили три режима работы приложения и дали им название цветов, которыми подаются известные всем сигналы светофора. В режиме реальной работы (зеленом режиме) пользователь может покупать и продавать ценные бумаги и выполнять вычисления на основе имеющихся данных. Когда биржа закрыта, доступен только режим анализа (красный режим), в котором доступны любые действия, кроме реальной покупки и продажи. В режиме имитации (желтый режим) эмулируются все действия, доступные в зеленом режиме, но реальное взаимодействие с рынком не происходит.

Наша клиентская программа поддерживает режимы с помощью объекта JavaScript.

```
var MODE_RED=1;
var MODE_AMBER=2;
var MODE_GREEN=2;
function TradingMode(){
  this.mode=MODE_RED;
}
```

Мы можем запрашивать и устанавливать режим для данного объекта, причем не исключено, что это будет сделано в разных местах программы. Мы также могли бы определить функции getMode () и setMode () и реализовать

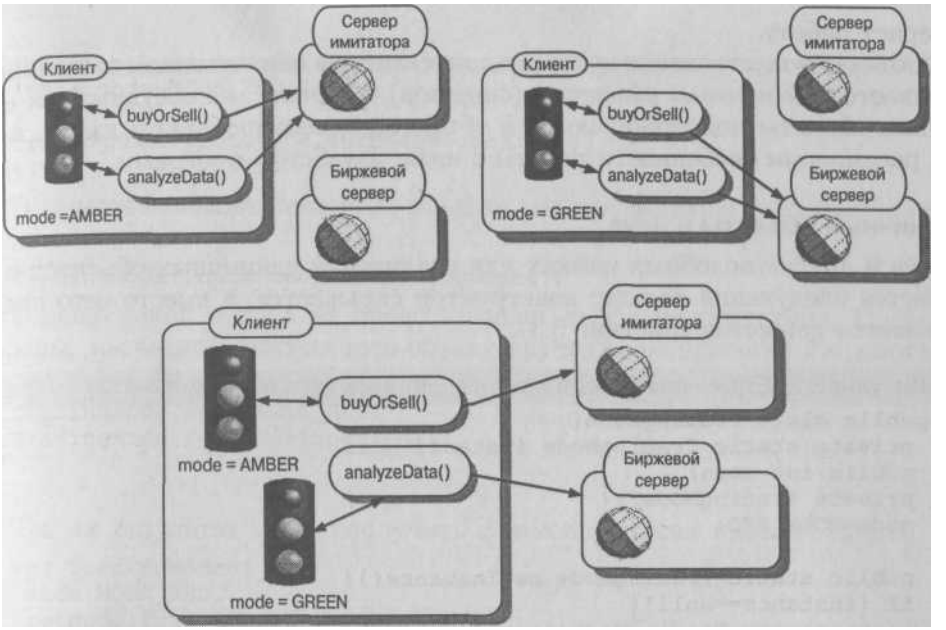


Рис. 3.4. В рассматриваемом Ajax-приложении средства покупки-продажи и функции анализа определяют, какие данные надо использовать: реальные или имитированные. Для этого они обращаются к объекту `TradingMode`. В желтом режиме происходит обращение к серверу-имитатору, а в зеленом режиме — к реальному биржевому серверу. Если в системе имеется несколько объектов `TradingMode`, их состояние может стать несогласованным, в результате система будет находиться в неопределенном режиме

в них проверку условий, допускающих установку режимов или препятствующих этому (например, эти функции могли бы выяснять, работает ли биржа), однако на данном этапе работы ограничимся простым решением.

Предположим, что пользователю доступны средства, позволяющие покупать и продавать ценные бумаги, а также вычислять выигрыш или потери от сделки, прежде чем совершать ее. В зависимости от режима взаимодействие, связанное с покупкой и продажей, может осуществляться с одной из двух служб: имитатором в желтом режиме и сервером брокера в зеленом режиме. В красном режиме средства взаимодействия заблокированы. Анализ производится на основе текущих и предыдущих цен. Источником данных являются имитатор в желтом режиме и биржевые сводки в зеленом режиме. Для того чтобы определить, откуда следует получать данные и с какой службой должно производиться взаимодействие при покупке и продаже, надо обратиться к объекту TradingMode (рис. 3.4).

Важно, чтобы средства покупки-продажи и средства анализа обращались к одному и тому же объекту TradingMode. Если пользователь будет осуществлять покупки и продажи в режиме имитации, но основываться при этом на реальных данных, в худшем случае он получит сообщение о том, что проиг-Рал условные деньги. Если же при реальных покупках и продажах пользователь будет исходить из данных, полученных от имитатора, он наверняка

потеряет работу.

Объект, существование которого допускается в единственном экземпляре, называется *единичным объектом* (singleton). Сначала мы обсудим, как единичные объекты поддерживаются в объектно-ориентированном языке, а затем рассмотрим особенности работы с ними JavaScript-программ.

Единичные объекты в Java

В Java и других подобных языках для реализации единичных объектов применяется следующий подход: конструктор скрывается, а вместо него предоставляется get-метод (листинг 3.3).

Листинг 3.3. Единичный объект TradingMode в языке Java

```
public class TradingMode{
 private static TradingMode instance=null;
 public int mode;
 private TradingMode(){
 mode=MODE_RED;
 }
 public static TradingMode getlnstance(){
 if (instance==null){
 instance=new TradingMode();
 }
 return instance;
 }
 public void setMode(int mode){

 }
```

В языке Java для определения поведения единичных объектов используются модификаторы доступа private и public. Приведенный ниже код не будет компилироваться, поскольку конструктор недоступен из других классов.

```
new TradingMode().setMode(MODE_AMBER);
```

Для того чтобы компиляция была выполнена без ошибок, выражение должно иметь следующий вид:

```
TradingMode.getlnstance().setMode(MODE_AMBER);
```

Такое решение гарантирует, что при каждом обращении к TradingMode реальный доступ будет предоставлен к одному и тому же объекту. Здесь были использованы языковые средства, отсутствующие в JavaScript, поэтому в следующем разделе мы рассмотрим, как добиться того же результата, пользуясь лишь имеющимися в наличии возможностями.

Единичные объекты в JavaScript

В JavaScript отсутствуют модификаторы доступа, поэтому, для того, чтобы "скрыть" конструктор, мы не будем создавать его. Язык JavaScript основан на прототипах, а конструкторы представляют собой объекты Function (подробно об этом речь пойдет в приложении Б). Мы можем создать объект TradingMode обычным способом.

```
function TradingMode(){
 this.mode=MODE_RED;
}
TradingMode.prototype.setMode=function(){
}
```

В качестве псевдоединичного объекта мы предоставим глобальную переменную.

TradingMode.instance=new TradingMode();

Однако такой подход не предотвращает вызов конструктора. С другой стороны, мы можем создать весь объект вручную, не прибегая к прототипу.

```
var TradingMode=new Object();
TradingMode.mode=MODE_RED;
TradingMode.setMode=function(){

}
```

Тот же результат можно получить с помощью более краткой записи.

```
var TradingMode={
 mode:MODE_RED,
 setMode:function() {

 }
};
```

Оба приведенных выше примера генерируют идентичные объекты. Первый способ более привычен программистам, использующим Java или C#. Второй способ мы показали здесь потому, что он часто используется в библиотеке Prototype и в базовых наборах средств, созданных на ее основе.

Данное решение работает в пределах одного контекста. Если сценарий будет загружен в отдельный элемент iFrame, он создаст собственную копию единичного объекта. Избежать такого эффекта можно, указав, что единичный объект доступен из документа верхнего уровня (в JavaScript свойство top всегда ссылается на текущий документ). Данный подход иллюстрируется кодом, представленным в листинге 3.4.

Листинг 3.4. Единичный объект TradingMode в JavaScript

```
Function getTradingMode(){
 if (!top.TradingMode){
 top.TradingMode=new Object();
 top.TradingMode.mode=MODE_RED;
 top.TradingMode.setMode=function(){

 }
 }
 return top.TradingMode;
}
```

Теперь сценарий можно включать в различные элементы I Frame, при этом обеспечивается присутствие объекта в одном экземпляре. (Если вы планируете поддержку единичного объекта в нескольких окнах верхнего уровня,

вам надо обратиться к top.opener. Из-за ограниченного объема книги мы предлагаем читателям сделать это самостоятельно.)

При написании кода пользовательского интерфейса обычно не возникает потребность в единичных объектах. В основном они бывают необходимы при моделировании бизнес-логики на JavaScript. В традиционных Web-приложениях код, реализующий бизнес-логику, обычно располагается на сервере, однако появление Ajax может изменить сложившееся положение и единичные объекты станут необходимыми и при разработке клиентских программ.

Итак, вы получили первое представление о том, каких результатов можно добиться с помощью реструктуризации. Примеры, рассмотренные в этой главе, чрезвычайно просты, но даже они показывают, что реструктуризация позволяет упростить код, исключить его слабые места, которые при увеличении размеров приложения могли бы привести к некорректной работе и даже полной его остановке.

В процессе обсуждения мы рассмотрели несколько шаблонов разработки. В следующем разделе речь пойдет о масштабных программах, работающих на стороне сервера, и вы увидите, как можно упорядочить запутанный код и обеспечить для него дополнительную степень гибкости.

3.3. *"Модель-представление-контроллер "*

Образы разработки, которые мы рассмотрели в предыдущих разделах, могут быть успешно применены для решения частных задач. Существуют также образы, предназначенные для организации всего приложения, — их принято называть *архитектурными шаблонами,* или *архитектурами.* В данном разделе мы рассмотрим архитектурный шаблон, который поможет в организации Ajax-проектов, упрощая его разработку и сопровождение.

Архитектура "модель-представление-контроллер" (Model-View-Controller — MVC) позволяет выделить фрагменты кода, отвечающие за взаимодействие с пользователем, и отделить их от тех частей программы, которые предназначены для выполнения сложных расчетов или реализации бизнес-логики. В этой главе мы покажем, как применить ее к разработке серверных компонентов, управляющих данными для Ajax-приложения. В главе 4 мы пойдем еще дальше и применим данную архитектуру для клиентской JavaScript-программы.

Архитектура "модель-представление-контроллер" определяет три роли, в которых выступают различные компоненты системы. Модель описывает проблемную область — те задачи, для решения которых и создано приложение. Так, текстовый процессор моделирует документ, а приложение для работы с картами — координаты на местности и контуры ландшафта.

Представление (его иногда также называют просмотром) — это программные средства, которые представляют информацию пользователю: реализуют формы ввода, изображения, текст, компоненты. Представление не обязательно должно работать с графикой. В программе, предоставляющей голосовой интерфейс, представление может быть реализовано на базе синтезатора речи.

Рис. 3.5. Основные компоненты архитектуры "модель—представление-контроллер". Представление и модель не могут обращаться друг к другу — они взаимодействуют только через контроллер. Контроллер рассматривается как промежуточный уровень между моделью и представлением. Благодаря такому подходу разграничиваются функции различных частей программы, обеспечивается дополнительная степень гибкости и упрощается сопровождение приложения

Основное правило архитектуры MVC гласит, что представление и модель не должны иметь доступа друг к другу. На первый взгляд такая программа может показаться неработоспособной, но здесь на помощь приходит контроллер. Когда пользователь нажимает клавишу или заполняет форму, представление сообщает об этом контроллеру. Контроллер выполняет действия с моделью и решает, должно ли измениться представление в соответствии с новым состоянием модели. Если изменения необходимы, контроллер сообщает о них представлению (рис. 3.5).

Преимущество подобного подхода заключается в том, что модель и представление практически не связаны между собой; они обладают лишь общими сведениями друг о друге. Конечно, информация должна быть достаточна, чтобы выполнять задачу, поставленную перед приложением, но совсем не обязательно, чтобы она была исчерпывающей.

Представьте себе программу для работы с перечнем товаров. Контроллер предоставляет представлению функцию, которая возвращает список продуктов, соответствующих определенной категории, но представление не знает ничего о том, как этот список был сформирован. Возможно, что первая версия программы содержала в памяти массив строк и заполняла его, читая данные из текстового файла. Вторая версия программы могла разрабатываться на основе новых требований, и в ней пришлось использовать реляционную базу данных. В результате код модели подвергся существенным изменениям. Поскольку контроллер по-прежнему доставляет представлению список продуктов, принадлежащих определенной категории, код представления остается неизменным.

Аналогично разработчики могут модифицировать представление, не опасаясь повредить модель. Понятно, что подобная ситуация сохраняется лишь До тех пор, пока будет соблюдаться соглашение об использовании интерфей-

сов, предоставляемых контроллером. Разделяя систему на подсистемы, MVC гарантирует, что изменения в одной части приложения не повлекут за собой изменений в другой части, и разработчики, ответственные за поддержку каждого компонента, могут работать практически независимо друг от друга.

У разработчиков классических Web-приложений уже накоплен опыт создания их в рамках архитектуры MVC. Эта архитектура помогает управлять последовательностью статических документов, составляющих интерфейс приложения. Когда приложение Ajax запускается и запрашивает информацию с сервера, используется тот лее механизм доставки данных, что и в классическом приложении. Таким образом, серверная часть Ajax-приложения может создаваться в рамках архитектуры "модель-представление-контроллер". Поскольку вопросы использования этой архитектуры для создания серверных программ уже хорошо отработаны и понять основные принципы несложно, мы начнем обсуждение MVC с серверной части, а затем перейдем к применениям, специфическим для Ajax.

Если вам еще не знакомы базовые средства разработки Web-программ, в следующем разделе вы найдете общие сведения о них и узнаете, как повысить масштабируемость и надежность Ajax-приложения. Если вы имеете опыт работы с инструментами Web-программирования, например, с процессорами шаблонов и ORM (Object-Relational Mapping), либо с такими средствами как Struts, Spring или Tapestry, то, наверное, уже в основном владеете сведениями, которые будут излагаться здесь. В этом случае мы советуем вам сразу перейти к главе 4, в которой будут обсуждаться другие способы использования этих инструментов.

3.4. Применение MVC для серверных программ

Архитектура "модель-представление-контроллер" применима к Web-приложениям, даже к их классическому варианту, который мы столь интенсивно критикуем в этой книге. Сама суть Web-приложения предписывает разделение представления и модели, поскольку поддерживающие их программы выполняются на различных машинах. Значит ли это, что любое Web-приложение автоматически попадает в категорию MVC и можно ли создать такую Web-программу, в которой средства представления и модели были объединены?

К сожалению, это возможно. Более того, сделать это достаточно просто, и многие разработчики, в том числе и авторы данной книги, попадали в эту ловушку.

Большинство сторонников применения архитектуры "модель-представление-контроллер" считают представлением не средства воспроизведения HTML-документа, а сам документ и генерирующий его код. Если применить такую точку зрения к приложению Ajax, в котором данные передаются JavaScript-клиенту, окажется, что представление — это XML-документ, передаваемый клиенту в составе HTTP-ответа. Для того чтобы отделить документ от бизнес-логики, требуются определенные навыки.

Рис; 3.6. Основные компоненты программы, используемой для генерации списка продукции интерактивного магазина. Список представляется в формате XML. При генерации представления мы извлекаем данные из базы, заполняем ими .структуры, соответствующие изделиям, а затем передаем информацию клиенту в виде потока XML-данных

3.4.1 *Серверная программа Ajax, созданная без применения образов разработки*

В качестве примера, иллюстрирующего обсуждаемый материал, создадим серверную программу для приложения Ajax. Основы создания клиентского кода Ajax мы уже рассматривали в главе 2 и в разделе 3.1.4, кроме того, мы вернемся к этому вопросу в главе 4. Сейчас же мы сосредоточим внимание на том, что происходит на Web-сервере. Начнем с программы, написанной самым простым способом, постепенно реструктуризируем ее в соответствии с архитектурой MVC и рассмотрим, выиграет ли от этого приложение. Сначала определим назначение самого приложения.

В базе данных хранится информация об одежде, имеющейся в наличии в магазине. Нам надо организовать обращение к базе и представить список товаров пользователю, причем представляемые данные должны содержать изображение, заголовок, краткое описание и цену. Если существуют различные варианты одного и того же изделия, отличающиеся, например, цветом или размером, это также надо учесть. На рис. 3.6 показаны основные компоненты системы, а именно: база данных, структура данных, представляющая конкретный продукт, и XML-доку мент, передаваемый Ajax-клиенту с перечнем всех продуктов (документ должен соответствовать запросу).

Предположим, что пользователь только что приступил к работе и ему предлагается выбрать категорию мужской, женской или детской одежды. Каждое изделие принадлежит одной из этих категорий; соответствующая информация хранится в базе данных, в столбце Category таблицы Garments. SQL-запрос, предназначенный для получения данных категории Menswear, имеет следующий вид:

```
SELECT * FROM garments WHERE CATEGORY = 'Menswear';
```

Нам надо получить результаты этого запроса, а затем передать их Ajax-приложению в формате XML. Рассмотрим, как можно сделать это.

Генерация XML-данных для передачи клиенту

В листинге 3.5 представлен код, решающий поставленную задачу. Пока этот код далек от совершенства. В данном примере используется технология PHP совместно с базой данных MySQL, однако для нас важна лишь структура программы. Принципиально ничего не изменится, если вместо PHP мы применим ASP-, или JSP-документ, или сценарий Ruby.

Листинг 3.5. Код, генерирующий XML-данные на основе запроса к базе

```
<?php
// Информация для клиента о том, что ему пересылается
// XML-информация
header("Content-type: application/xml");
echo "<?xml version=\"1.0\" encoding=\"UTF-8\" ?>\n";
$db=mysql_connect("my_db_server","mysql_user");
// Получение данных из базы
mysql_select_db("mydb",$db);
$sql="SELECT id,title,description,price,colors,sizes"
  ."FROM garments WHERE category=\"{$cat}\"";
$result=mysql_query($sql,$db);
echo "<garments>\n";
// Просмотр набора результатов
while ($myrow = raysql_fetch_row($result)) {
 printf("<garment id=\"%s\" title=\"%s\">\n"
  ."<description>%s</description>\n<price>%s</price>\n",
 $myrow["id"],
 $myrow["title"],
 $myrow["description"],
 $myrow["price"]);
 if  (!is_null($myrow["colors"])){
 echo  "<colors>{$myrow['colors']}</colors>\n";
 }
 if  (!is_null($myrow["sizes"])){
 echo  "<sizes>{$myrow['sizes']}</sizes>\n";
 }
 echo "</garment>\n";
}
echo "</garments>\n";
?>
```

Код PHP в листинге 3.5 генерирует XML-документ. Если условиям запроса удовлетворяют только два изделия, этот документ будет выглядеть так, как показано в листинге 3.6. Отступы были добавлены лишь для того, чтобы код стал более удобным для восприятия. Язык XML очень часто используется для обмена данными между клиентом и сервером, кроме того, в главе 2 мы уже говорили об обработке XML-документа, полученного с сервера с помощью объекта XMLHttpRequest. В главе 5 мы рассмотрим другие варианты организации обмена.

Листинг 3.6. Пример данных, сгенерированных кодом из листинга 3.5

```
<garments>
 <garment id="SCK001" title="Golfers' Socks">
 <description>Garish diamond patterned socks. Real wool.
```

```
Real itchy.</description>
<price>$5.99</price>
<colors>heather combo,hawaiian medley,wild turkey</colors>
</garment>
<garment id="HAT056" title="Deerstalker Cap">
<description>Complete with big flappy bits.
As worn by the great detective Sherlock Holmes.
Pipe is model's own.</description>
<price>$79.99</price>
<sizes>S, M, L, XL, egghead</sizes>
</garment>
</garments>
```

Итак, мы получили серверную программу Web-приложения. Считаем, что Ajax-клиент способен правильно обработать полученные XML-данные. Попробуем представить себе дальнейшее развитие этой программы. Предположим, что ассортимент продукции расширился и нам надо ввести новые категории (например, Smart, Casual, Outdoor). Кроме того, необходимо реализовать поисковую функцию и установить ссылку на сервер химчистки. Очевидно, что XML-документ позволяет включить новые данные. Однако сможем ли мы повторно использовать имеющийся код и какие препятствия встретим при решении этой задачи?

Проблемы с повторным использованием кода

Организовать повторное использование сценария в том виде, в котором он существует в данный момент, непросто. Во-первых, мы "жестко" запрограммировали SQL-запрос в составе документа. Если мы захотим изменить критерии поиска, запрос нам придется генерировать по-другому. По мере добавления вариантов запроса в программе накопится большое количество выражений. Ситуация может развиваться еще хуже. Рассмотрим следующее выражение:

```
$sql="SELECT id,title,description,price,colors,sizes"
  ."FROM garments WHERE ".$sqlWhere;
```

Если мы разрешим передавать произвольные выражения WHERE как CGI-параметры, то можем получить следующий результат:

```
garments.php?sqlWhere=CATEGORY="Menswear"
```

Такое решение приводит к некорректной работе модели и представления, а также создает возможность для атак с применением SQL-выражений. Несмотря на то что современная система PHP имеет встроенные системы защиты, не стоит полагаться на них.

Во-вторых, мы "жестко" запрограммировали формат XML (он определяется структурой выражений `printf` и `echo`). Однако по ряду причин формат Данных может быть изменен. Не исключено, например, что руководство примет решение выводить не только цену, по которой продается товар, но и цену производителя.

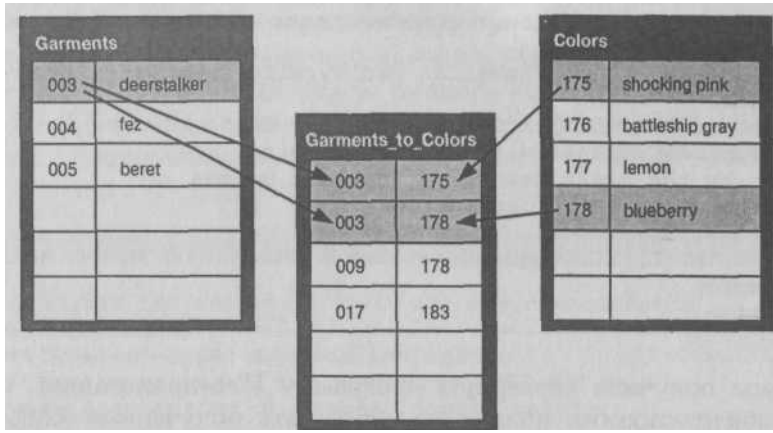


Рис. 3.7. Отношение "многие ко многим", реализованное в базе данных. В таблице Colors перечислены значения цвета для всех изделий, в результате исчезает необходимость хранить информацию о цвете в таблице Garments

В-третьих, для генерации XML-данных непосредственно используется набор результатов, полученный после обработки запроса к базе. На первый взгляд, кажется, что такое решение обеспечивает высокую эффективность работы, но при этом возможны две проблемы. Соединение с базой остается открытым в течение всего времени генерации XML-данных. В данном случае в цикле while не производятся сложные расчеты и соединение не будет открыто слишком долго, но в общем случае такое решение может стать причиной напрасного расходования ресурсов. Кроме того, подобный подход применим, только если мы рассматриваем базу данных как "плоскую" структуру.

3.4.2. Реструктуризация модели

На данном этапе работы мы храним списки цвета и размеров чрезвычайно неэффективно — помещаем значения, разделенные запятыми, в поля таблицы Garments. Чтобы преобразовать данные в соответствии с реляционной моделью, мы должны создать отдельную таблицу для хранения всех доступных значений цвета и вспомогательную таблицу, которая связывала бы изделия с цветом. Подобное отношение называется "многие ко многим" и показано на рис. 3.7.

Предположим, например, что вам нужна охотничья шляпа и вы хотите выяснить, шляпы каких цветов есть в наличии. Для этого вы обращаетесь к ключу garment_id в таблице Garments_to_Colors. Эти записи дают возможность получить первичные ключи в таблице Colors, и по ним мы видим, что в наличии имеются охотничьи шляпы цвета **shocking pink и blueberry,** а шляп цвета battleship gray нет в наличии. Можно сформировать и обратный запрос и использовать таблицу Garments_to_Colors, чтобы выяснить, какие изделия имеют требуемый цвет.

Теперь структура данных стала более строгой, но для того, чтобы получить информацию, надо формировать сложные запросы. Вместо того чтобы формировать такие запросы вручную, было бы лучше, если бы изделие можно было рассматривать как объект, в составе которого присутствовали бы массивы, задающие варианты цвета и размеры.

Инструмент Object-Relational Mapping

Существуют средства, способные выполнить описанную задачу за нас. Для этой цели подходит набор инструментов ORM (Object-Relational Mapping), которые автоматически преобразуют данные, содержащиеся в базе, в объект, хранящийся в памяти компьютера. При этом разработчик избавлен от необходимости формировать низкоуровневые SQL-выражения. Программисты, использующие PHP, могут обратиться к PEAR DB_DataObject, EZPDO (Easy PHP Data Objects) или Metastorage. Разработчики, применяющие Java, ограничены в выборе. Им доступен лишь продукт Hibernate (теперь он перенесен и на платформу .NET). Инструменты ORM — обширная тема, рассматривать которую мы сейчас не будем.

Рассматривая наше приложение с точки зрения MVC, легко увидеть, что применение ORM дает интересный побочный эффект — с самого начала у нас в руках оказываются основы модели. Мы можем написать программу генерации XML-данных, которая будет взаимодействовать с объектом Garment, предоставив ORM самостоятельно выполнять все необходимые действия с базой. Мы больше не привязаны к API конкретной базы данных и не должны учитывать особенности ее работы. В листинге 3.7 показаны изменения кода, связанные с переходом к ORM.

В данном случае мы определяем бизнес-объекты (т.е. модель) на PHP, используя Pear: :DB_DataObject, в результате чего наши классы становятся расширением базового класса DB_DataObject. Различные ORM делают это по-разному, но в результате мы получаем набор объектов, к которым можем обращаться как к обычному коду, абстрагируясь от сложных SQL-выражений.

Листинг 3.7. Объектная модель для приложения

```
require_once  "DB/DataObject.php";
class GarmentColor extends DB_DataObject {
 var $id;
 var  $garment_id;
 var  $color_id;
}
class Color extends DB_DataObject {
 var $id;
 var $name;
}
class Garment extends DB_DataObject {
 var $id;
 var $title;
 var  $description;
 var $price;
 var $colors;
```

```
var $category;
function getColors(){
if (!isset($this->colors)){
$linkObject=new GarmentColor();
$linkObject->garment_id = $this->id;
$linkObject->find();
$colors=array();
while ($linkObject->fetch()){
$colorObject=new Color(),•
$colorObject->id=$linkObject->color_id;
$colorObject->find();
while ($colorObj ect->fetch()){
$colors[] = clone($colorObject);
}
}
}
return $colors;
}
}
```

Помимо основного объекта Garment, мы определяем объект Color и метод объекта Garment, предназначенный для получения всех доступных цветов. Размеры поддерживаются аналогичным способом. Поскольку библиотека непосредственно не поддерживает отношение "многие ко многим", нам надо определить объект для связующей таблицы и организовать перебор в методе getColors(). Несмотря на это модель выглядит завершенной и удобна для восприятия. Рассмотрим, как можно применить данную модель к нашему документу.

Использование обновленной модели

Мы сгенерировали модель на основе более совершенной структуры базы данных. Теперь нам надо использовать ее в нашем PHP-сценарии. В листинге 3.8 показан код страницы, использующей объекты на базе ORM.

Листинг 3.8. Страница, использующая ORM для взаимодействия с базой

```
<?php
header("Content-type: application/xml");
echo "<?xml version=\"1.0\" encoding=\"UTF-8\" ?>\n";
 include "gannent_busines8_objects.inc"
$garment=new Garment;
$garment->category = $_GET["cat"];
$number_of_rows = $garment->find();
echo "<garments>\n";
while ( $garment->fetch()) {
 printf("<garment id=\"%s\" title=\"%s\">\n"
  ."<description>%s</descrJ^tion>\n<price>%s</price>\n",
 $garment->id,
 $garment->title,
 $garment->description,
 $garment->price
);
 Scolors»
```

```
$garment->getColors();
if (count($colors)>0){
echo "<colors>\n";
for($i=0;$i<count($colors);$i++){
echo "<color>{$colors[$i]}</color>\n";
}
echo "</colors>\n";
}
echo "</garment>\n";
}
echo "</garments>\n";
?>
```

Вместо того чтобы конструировать SQL-запросы, создается пустой объект Garment, который затем частично заполняется данными, представляющими собой критерии поиска. Поскольку модель включается из отдельного файла, мы можем повторно использовать ее для поиска. Средства представления XML также генерируются на основе модели. Наши последующие действия по реструктуризации направлены на разделение XML-данных и процесса их генерации.

3.4.3. Разделение содержимого и представления

Код представления пока еще не отделен от объекта. Причина в том, что формат XML связан с кодом, предназначенным для его анализа. Если мы работаем с несколькими страницами, будет полезно, если мы сможем изменить XML-формат в одном месте приложения, оставив другие фрагменты без изменений. В более сложном случае может понадобиться поддержка нескольких форматов, например, один для представления перечня товаров пользователю, а другой для управления самим магазином. Поэтому желательно определить каждый формат только один раз и обеспечить для них централизованное отображение.

Системы на базе шаблонов

Для решения данной задачи используется язык шаблонов. Система, поддерживающая шаблоны, получает текстовый документ, содержащий специальную разметку. Элементы разметки обозначают позиции реальных переменных. К языкам шаблонов можно отнести PHP, ASP и JSP, которые позволяют включать фрагменты кода в содержимое Web-страниц. Этим они принципиально отличаются от кодов, включающих содержимое, примерами которых являются Java-сервлеты или традиционные сценарии CGI. Несмотря на то что сценарии предоставляют достаточно мощные средства обработки запросов, используя их, очень трудно разделить бизнес-логику и представление.

Языки сценариев, ориентированные на конкретное применение, например PHP Smarty и Apache Velocity (система на базе Java; при переносе на платформу .NET она получила название N Velocity), предоставляют ограниченные возможности для создания кода. В ряде случаев поток управления

может содержать лишь ветвление (оператор if) и циклы (например, **операторы for и while). В** листинге 3.9 показан шаблон PHP Smarty для генерации XML-данных.

```
<?xml version="1.0" encoding="UTF-8" ?>
<garments>
{section name=garment loop=$garments}
 <garment id="{$garment.id}" title="{$garment.title}">
 <description>{$garment.description}</description>
 <price>{$garment.price}</price>
{if count($garment.getColors())>0}
 <colors>
{section name=color loop=$garment.getColors()}
 <color>$color->name</color>
{/section}
 </colors>
{/if}
 </garment>
{/section}
</garments>
```

Входной информацией для шаблона является переменная массива garments, содержащего объекты Garment. Блыпая часть шаблона генерируется процессором, а разделы в фигурных скобках интерпретируются как инструкции. Они либо заменяются именами переменных, либо интерпретируются как выражения ветвления и циклов. Структура выходного XML-документа, выраженная в виде шаблона, воспринимается гораздо лучше, чем код, подобный тому, который был представлен в листинге 3.7. Теперь поговорим о том, как применить шаблон к нашему документу.

Использование обновленного представления

Мы переместили определение XML-данных из основного документа в шаблон Smarty. В результате главная страница должна лишь установить процессор шаблона и передать необходимые данные. В листинге 3.10 отражены изменения, необходимые для этого.

Листинг 3.10. Использование Smarty для генерации XML-данных

```
<?php
header("Content-type: application/xml");
include "garment_business_objects.inc";
include "smarty.class.php";
$garment«new DataObjects_Garment;
$garment->category = $_GET["cat"] ;
$number_of_rows - $garment->find();
$smarty-new Smarty;
$3marty->assign('garments', $garments);
$smarty->display('garments_xml.tpl');
?>
```

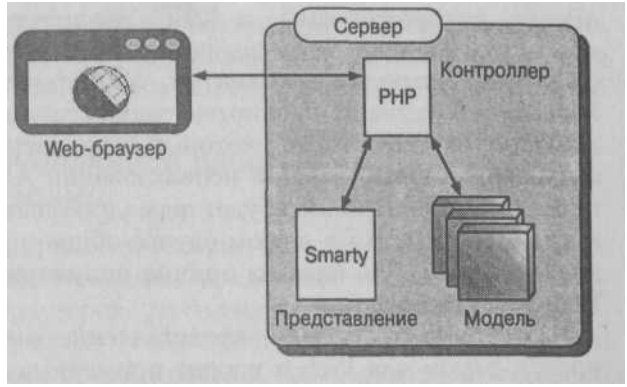


рис. 3.8. Архитектура MVC применима к Web-приложениям. Web-страница или сервлет действует как контроллер и обращается к модели для получения данных. Затем эти данные передаются файлу шаблона (представление), который генерирует содержимое, предназначенное для передачи клиентской программе. Заметьте, что в данной ситуации предусмотрено только чтение данных. При модификации модели поток событий несколько изменится, но роли составных частей приложения останутся прежними

Обычно действия с шаблонами насчитывают три этапа. Сначала мы создаем шаблон Smarty, затем заполняем его переменными. В данном случае переменная только одна, но мы можем использовать их столько, сколько нам надо. Например, если информация о пользователе хранится вместе с данными о сеансе, мы можем сформировать посредством шаблона персональное приветствие. После этого мы вызываем функцию `display()` и передаем ей имя файла шаблона.

Итак, нам удалось отделить представление от страницы с результатами поиска. Формат XML определяется только один раз, и мы можем организовать воспроизведение данных несколькими строками кода. Страница с результатами поиска теперь ориентирована на решение одной задачи и содержит только необходимую для этого информацию, а именно: параметры поиска и определение выходного формата. Если вы помните, мы хотели обеспечить возможность легко переходить к форматам, альтернативным XML. С помощью Smarty это достигается просто, мы лишь определяем дополнительный формат. Если мы хотим реализовать структуру, обеспечивающую минимум изменений, мы можем даже использовать шаблон внутри другого шаблона.

Если вы вспомните, в чем состоит архитектура "модель-представление-контроллер", то увидите, что наше приложение соответствует ей. Реализация приложения условно показана на рис. 3.8.

В нашем наборе объектов постоянное хранение модели в базе данных обеспечивается посредством ORM. Представлением служит шаблон, определяющий XML-формат. Контроллер — это страница "поиска по категории" и другие документы, которые мы создадим и которые будут объединять модель и представление.

Мы рассмотрели пример применения архитектуры "модель-представление-контроллер" к Web-приложению. Мы обсуждали серверные программы приложения Ajax, поддерживающего XML-документы, но нетрудно понять, как применить данную архитектуру к классическому Web-приложению, поддерживающему HTML-документы.

В зависимости от технологий, возможны вариации данного образа разработки, но общие принципы остаются теми же. Компоненты Enterprise Jav-

aBeans, поддерживаемые в J2EE, реализуют модель и контроллер и даже позволяют помещать их на различные серверы. Классы .NET делегируют функции контроллера объектам, специфическим для конкретных страниц. Базовые средства из библиотек типа Struts определяют *контроллер переднего плана* (front controller), который перехватывает все запросы к приложению и перенаправляет их. При использовании Apache Struts возможно решение, при котором контроллер будет переадресовывать пользователя от одной страницы к другой. Но в любом случае общие принципы архитектуры остаются неизменными. Это одна из причин популярности MVC среди разработчиков Web-приложений.

Архитектура "модель—представление-контроллер" упрощает работу над программами для Web и вполне применима не только для классических, но и для Ajax-приложений. Однако в Ajax архитектура MVC используется не единственным способом. В главе 4 мы рассмотрим разновидности данного образа, обеспечивающие преимущества структурированной разработки для всех компонентов приложения. Однако сначала рассмотрим еще один способ упорядочения Ajax-приложений.

Помимо реструктуризации кода, мы можем упростить работу над ним, используя базовые средства и библиотеки независимых разработчиков. По мере возрастания интереса к Ajax появляются новые средства разработки, j Рассмотрением наиболее популярных из них мы завершим данную главу.

3.5. Библиотеки независимых производителей

В большинстве случаев целью реструктуризации является устранение повторений кода. Для этого фрагменты кода оформляются в виде функций или объектов. Логичным завершением этих действий является помещение часто используемых функций в библиотеки, которые могут быть повторно использованы в других проектах. При этом уменьшается объем кода, создаваемого вручную, и повышается производительность труда программиста. Более того, поскольку библиотечный код уже был отлажен и протестирован в предыдущих проектах, качество программ должно возрасти.

На протяжении этой книги мы время от времени будем создавать простые, но универсальные средства разработки. Вы можете использовать их при работе над своими проектами. В главах 4 и 5 мы разработаем объект ObjectViewer, в главе 5 — объект CommandQueue, в главе 6 — средства оповещения, в главе 9 — элемент профилирования Stopwatch, а в приложении А — отладочную консоль. В конце глав 9 и 13 будут рассмотрены несложные примеры. Там же мы реструктуризируем созданные программы, превратив их в компоненты, пригодные для повторного применения.

Очевидно, что не мы одни интересуемся Ajax и JavaScript, поэтому многие средства для работы с ними, многократно проверенные разработчиками, сейчас доступны в Интернете.

В этом разделе мы рассмотрим некоторые библиотеки и наборы базовых средств, которые были созданы независимыми производителями и доступны для использования в Ajax-приложениях. На сегодняшний день популяр-

ность Ajax высока, поэтому мы не можем детально описать все приложения и постараемся дать вам лишь общее представление о том, какие продукты существуют и как они помогут вам навести порядок в коде ваших проектов.

3.5.1. Библиотеки, обеспечивающие работу с различными браузерами

Как было сказано в разделе 3.2.1, несоответствие браузеров не исчезло с появлением Ajax-приложений. Ряд библиотек предлагает функции, позволяющие скрыть несоответствие браузеров, рекомендуя единую точку обращения к соответствующим средствам (другими словами, они реализуют образ разработки Fagade). Одни из них ориентированы на конкретные функции, другие пытаются предоставить среду программирования с более развитыми возможностями.

Библиотека x

Библиотека x предназначена для решения задач, возникающих при написании DHTML-приложений. Она была впервые представлена в 2001 году и является развитием более ранней библиотеки CBE (Cross-Browser Extensions). Эта библиотека предоставляет функции, позволяющие обрабатывать в различных браузерах элементы DOM и стили, работать с моделями событий, а также поддерживает анимацию и перетаскивание объектов с помощью мыши. Библиотека x допускает работу с Internet Explorer, начиная с версии 4, и с последними версиями браузеров Opera и Mozilla.

Стиль кодирования библиотеки x основан на использовании функций; причем переменное количество параметров и отсутствие поддержки типов позволяют извлечь дополнительные преимущества. Например, для метода document.getElementById(), обрабатывающего только строковые данные, создана оболочка, которая принимает как строки, так и элементы DOM. Если передана строка, осуществляется преобразование идентификатора, а элемент DOM возвращается в неизменном виде. В результате вызов xGetElementByld() гарантирует, что параметр будет преобразован в узел DOM; при этом отпадает необходимость выполнять проверку и использовать условный оператор. Возможность замены элемента DOM текстовым идентификатором полезна при динамической генерации кода, например, при передаче строки методу setTimeout () либо при организации обратного вызова.

При работе со стилями для элементов DOM используются функции, посредством которых можно как получать, так и устанавливать значение. Рассмотрим следующее выражение:

```
xWidth(myElement)
```

В результате выполнения функции возвращается значение ширины для элемента DOM (где myElement — либо элемент, либо его идентификатор). Добавляя дополнительный параметр, можно использовать ту же функцию Для установки значения ширины.

```
xWidth(myElement,420)
```

Следовательно, чтобы задать ширину элемента равной ширине другого элемента, надо использовать выражение.

```
xWidth(secondElement,xWidth(firstElement))
```

Библиотека x не содержит кода для передачи запросов по сети, но она может быть полезна при создании пользовательских интерфейсов для Ajax-приложений.

Библиотека Sarissa

Библиотека Sarissa в основном ориентирована на работу JavaScript-сценариев с XML-данными. Она поддерживает MSXML ActiveX-компоненты Internet Explorer (начиная с версии 3) и базовые функции Mozilla, Opera, Konqueror и Safari. Расширенные средства, такие как XPath и XSLT, поддерживаются не для всех указанных браузеров.

Для разработчиков Ajax-приложений наиболее важными функциями являются поддержка объекта XMLHttpRequest в различных браузерах. Вместо того чтобы создавать объект типа Facade, Sarissa использует образ разработки Adapter для создания объекта XMLHttpRequest в тех браузерах, в которых отсутствует встроенный объект с таким именем (в частности, Internet Explorer). Код, реализующий описанные средства, конечно же, обращается к объектам ActiveX так, как было описано в главе 2, но эти действия остаются невидимыми для разработчика. Например, после импортирования библиотеки Sarissa следующий фрагмент кода будет выполняться независимо от используемого браузера:

```
var xhr = new XMLHttpRequest();
xhr.open("GET", "myData.xml");
xhr.onreadystatechange = function(){
 if(xhr.readyState == 4){
 alert(xhr.responseXML);
 }
}
xhr.send(null);
```

Сравните этот фрагмент с кодом, представленным в листинге 2.11, и заметьте, что вызовы функций API осуществляются так же, как и в браузерах Mozilla и Safari, содержащих встроенный объект XMLHttpRequest.

Как было сказано ранее, Sarissa также представляет универсальные средства для работы с XML-документами, в частности, позволяет выполнять сериализацию произвольных объектов JavaScript и представлять их в формате XML. Данный механизм может быть использован для обработки XML-данных, полученных с сервера в ответ на запрос. (Эту задачу и возможные ее решения мы рассмотрим в главе 5.)

Библиотека Prototype

Библиотека Prototype содержит функции общего назначения для программ на JavaScript. Основное внимание разработчики библиотеки уделили расширению возможностей самого языка и поддержке объектов. Дополнительные

возможности языка формируют специфический стиль программирования. Несмотря на то что сам код Prototype сложен для восприятия, так как он далек от стиля Java/C#, пользоваться самой библиотекой и другими библиотеками, созданными на ее основе, очень просто. Prototype молено рассматривать как библиотеку для разработчиков библиотек. При работе над Ajax-приложениями целесообразнее использовать библиотеки, созданные на основе Prototype, чем сами базовые средства. Некоторые из этих библиотек мы рассмотрим в следующем разделе. Здесь же ограничимся кратким обсуждением основных средств Prototype. Это нужно для того, чтобы понять стиль кодирования и лучше разобраться в продуктах Scriptaculous, Rico и Ruby on Rails.

Prototype обеспечивает "расширение" одного объекта другим путем копирования всех свойств и методов родительского объекта в дочерний объект. Эту особенность лучше рассмотреть на конкретном примере. Предположим, что мы определили родительский класс Vehicle.

```
function Vehicle(numWheels,maxSpeed){
 this.numWheels=numWheels;
 this.maxSpeed=maxSpeed;
}
```

Мы хотим создать конкретный экземпляр этого класса, представляющий пассажирский поезд. В дочернем классе также нужны средства, представляющие число вагонов и механизм для изменения этого значения. Пользуясь обычными средствами JavaScript, мы можем написать следующий код:

```
var passTrain=new Vehicle(24,100);
passTrain.carriageCount=12;
passTrain.addCarriage=function(){
 this.carriageCount++;
}
passTrain.removeCarriage=function(){
 this.carriageCount—;
}
```

Итак, мы реализовали функциональные возможности, необходимые для объекта passTrain. Анализируя код с точки зрения специалиста по проектированию программ, можно сказать, что в нем практически ничего не сделано для формирования привычного всем объекта. Prototype может помочь в решении подобных задач, определяя функции и расширяя с их помощью базовый объект. Определим расширенные возможности и оформим их в виде объекта.

```
function CarriagePuller(carriageCount){
 this.carriageCount=carriageCount;
 this.addCarriage=function(){
 this.carriageCount++;
 )
 this.removeCarriage=function(){
 this.carriageCount—;
 }
}
```

Затем объединим созданные средства с базовым объектом, обеспечив требуемое поведение.

```
var parent=new Vehicle(24,100) ;
var extension=new CarriagePuller(12);
var passTrain=Object.extend(parent,extension);
```

Заметьте, что родительский объект и объект расширения были определены независимо друг от друга, а лишь затем мы объединили их. Отношение "родительский-дочерний" имеет место между экземплярами, а не между классами Vehicle и CarriagePuller. Несмотря на то что данное решение не соответствует строго объектному подходу, весь код для конкретного действия, в нашем случае это движение вагонов, содержится в одном месте, а это упрощает его повторное использование. Если для небольших примеров подобные приемы могут показаться излишними, то для масштабных приложений они оказываются очень полезными.

Библиотека Prototype также предоставляет средства поддержки Ajax — объект, позволяющий извлекать объект XMLHttpRequest при работе на различных браузерах. Тип Ajax.Request используется для передачи запросов серверу посредством объекта XMLHttpRequest.

```
var req=new Ajax.Request('myData.xml');
```

В конструкторе используется стиль программирования, который часто встречается в библиотеках на базе Prototype. В качестве необязательного параметра указывается ассоциативный массив, позволяющий при необходимости задавать различные установки. Для каждой из них предусмотрено значение по умолчанию, поэтому указывать надо только те параметры, которые необходимо изменить. В конструкторе Ajax.Request посредством ассоциативного массива можно указать передаваемые данные, параметры запроса, HTTP-метод и функцию обратного вызова. Пример вызова Ajax.Request, в результате которого переопределяются некоторые установки, приведен ниже.

```
var req=new Ajax.Request(
 'myData.xml',
 {  method: 'get',
      parameters: { name:'dave',likes:'chocolate,rhubarb' },
      onLoaded: function(){ alert('loaded!'); },
      onComplete: function(){
      alert('done!\n\n'+req.transport.responseText);
      }
 }
);
```

Здесь посредством массива задаются четыре параметра. Поскольку в библиотеке Prototype по умолчанию используется метод post, то метод get приходится указывать явно. При использовании метода get строка параметров присоединяется к URL в первой строке запроса. Если бы мы использовали метод post, параметры помещались бы в тело запроса. Обработчики onLoaded и onComplete представляют собой функции обратного вызова, которые получают управление тогда, когда свойство readyState объекта XMLHttpRequest изменяется. Переменная req.transport в функции onComplete содержит ссылку на объект XMLHttpRequest.

На основе Ajax.Request определен тип Ajax.updater, который загружает фрагменты сценария, сгенерированные сервером, и выполняет их. Такой подход соответствует шаблону, ориентированному на сценарий. Мы подробно опишем его в главе 5.

На этом мы завершаем краткий обзор библиотек, предназначенных для работы с различными браузерами. Набор библиотек, описанных здесь, далеко не полон, а библиотеки, приведенные в качестве примера, выбраны почти случайным образом. Как уже было сказано ранее, разработки в этой области ведутся достаточно интенсивно, и в ближайшее время можно ожидать появления новых продуктов. В следующем разделе мы рассмотрим ряд компонентов, созданных на базе библиотек, рассмотренных ранее, и других библиотек.

3.5.2. Компоненты и наборы компонентов

Рассмотренные нами библиотеки обеспечивают поддержку различных браузеров на достаточно низком уровне. В частности, они позволяют работать с элементами DOM и загружать ресурсы с сервера. Данные инструменты существенно упрощают создание пользовательских интерфейсов и логики приложений, но, несмотря на это, объем работы остается гораздо большим, по сравнению тем, который требуется при использовании Swing, MFC или Qt.

По этой причине начали появляться компоненты и их наборы, ориентированные на работу с Ajax. В данном разделе мы рассмотрим некоторые из них. Как и прежде, мы ставим перед собой задачу не детально изучить, а лишь сформировать общее представление об этих компонентах.

Scriptaculous

Библиотека компонентов Scriptaculous создана на базе библиотеки Prototype, которую мы рассматривали в предыдущем разделе. На сегодняшний день Scriptaculous предоставляет лишь два набора функций, однако работа над данной библиотекой продолжается и разработчики планируют реализовать дополнительные возможности.

Библиотека Effects определяет визуальные анимационные эффекты, которые молено применить к элементам DOM для изменения их размеров, расположения и прозрачности. Элементы в составе Effects можно легко объединять друг с другом, реализуя вторичные эффекты. Например, вызов Puff() приводит к тому, что размеры элемента увеличатся, в то же время он станет более прозрачным и наконец исчезнет с экрана. Еще одна базовая функция, ParalleK), позволяет одновременно применить несколько эффектов. Библиотека Effects хорошо подходит для тех случаев, когда надо быстро реализовать визуальную обратную связь с пользователем. Пример ее применения будет приведен в главе 6.

Для применения эффекта обычно достаточно вызвать конструктор и передать ему в качестве параметра целевой элемент DOM либо его идентификатор, например:

```
new Effect.SlideDown(myDOMElement);
```

В основе эффектов, реализованных в данной библиотеке, лежит принцип перемещения объектов с учетом временных параметров и событий. Разработчику предоставляется возможность использовать линейное и синусоидальное перемещение, а также качание и пульсацию. Для создания эффектов, определяемых пользователем, достаточно объединить основные эффекты и задать соответствующие параметры. Подробное их обсуждение выходит за рамки нашего краткого обзора. Пример применения эффектов Scriptaculous будет приведен в главе 6 при разработке системы оповещения.

Набор Scriptaculous также содержит библиотеку, обеспечивающую перетаскивание объектов с помощью мыши. Эту возможность предоставляет класс Sortable. Он получает родительский элемент DOM и обеспечивает перетаскивание всех дочерних элементов. Параметры, передаваемые конструктору, позволяют задать обработчики обратного вызова, получающие управление при движении или опускании объекта, типы дочерних элементов и список целевых объектов (т.е. элементов, которые могут принять перетаскиваемый объект в том случае, если пользователь отпустит кнопку мыши в тот момент, когда курсор находится над одним из них). В качестве параметров могут быть также указаны объекты Effect. Их действие проявляется при нажатии кнопки мыши, в процессе движения курсора либо при отпускании кнопки мыши.

Rico

Подобно Scriptaculous, средства Rico созданы на базе библиотеки Prototype и также предоставляют настраиваемые эффекты и возможность перетаскивания объектов. Кроме того, средствами Rico поддерживается объект Behavior — фрагмент кода, применяемый к поддереву DOM для реализации интерактивных функций. Примером применения Behavior может служить компонент Accordion, который помещает набор элементов DOM в заданную область и расширяет их по одному. (Компонент такого типа обычно называют *полосой outlook* (outlook bar), поскольку он получил популярность благодаря использованию в Microsoft Outlook.)

Создадим простой компонент Rico Accordion. Нам потребуется родительский элемент DOM; каждый элемент, дочерний по отношению к нему, становится отдельной панелью в составе области. Для каждой панели мы определяем элемент div, содержащий еще два элемента div, которые задают заголовок и тело панели.

```
<div  id='myAccordion'>
 <div>
  <div>Dictionary  Definition</div>
  <div>
  <ul>
  <li><b>n.</b>A portable wind instrument with a small
  keyboard and free metal reeds that sound when air is
  forced past them by pleated bellows operated by the
  player.</li>
  <li><b>adj.</b>Having folds or bends like the bellows
  of an accordion: accordion pleats; accordion blinds.</li>
```

```
  </ul>
  </div>
  </div>
  <div>
  <div>A  picture</div>
  <div>
  <img  src='monkey-accordion.jpg'X/img>
  </div>
 </div>
</div>
```

Первая панель предоставляет статью словаря, определяющую слово accordion (аккордеон), а вторая панель содержит изображение обезьяны, играющей на аккордеоне (рис. 3.9). При воспроизведении без применения специальных средств эти два элемента размещаются один над другим. Однако мы присвоили элементу div верхнего уровня идентификатор и можем передать его объекту Accordion, код которого приведен ниже.

```
var  outer=$('myAccordion') ;
 outer.style.width='320px';
 new  Rico.Accordion{
  outer,
  {  panelHeight:400,
  expandedBg:'#909090',
  collapsedBg:'#404040',
  }
);
```

Первая строка выглядит несколько необычно. Символ $ допустим в имени переменной и, использованный так, как показано выше, ссылается на функцию в базовой библиотеке Prototype. Функция $() выполняет преобразование узлов DOM подобно тому, как это делает функция библиотеки x xGetElementByld(), рассмотренной в предыдущем разделе. Мы передаем конструктору объекта Accordion ссылку на преобразованный элемент DOM, а также массив параметров (подобный подход часто применяется в библиотеках, созданных на базе Prototype). В данном случае параметры лишь задают стили для визуальных элементов компонента Accordion. При необходимости можно также указать функцию обратного вызова, которая получит управление при открытии или закрытии панели. На рис. 3.9 показан эффект, полученный в результате применения объекта Accordion к элементам DOM. Объект Behavior предоставляет средства для создания" на базе популярных элементов разметки компонентов, пригодных для повторного использования, и обеспечивает разделение содержимого и интерактивного поведения. Разговор о принципах разработки пользовательского интерфейса средствами JavaScript мы продолжим в главе 4.

Внимание заслуживает также средство Rico, обеспечивающее поддержку запросов к серверу в стиле Ajax. Эти запросы реализуются посредством глобального объекта AjaxEngine. Возможности объекта AjaxEngine не ограничиваются использованием XMLHttpRequest в различных браузерах. Он определяет ответ в формате XML, состоящий из набора элементов response. Процессор Rico автоматически декодирует их и обеспечивает поддержку двух

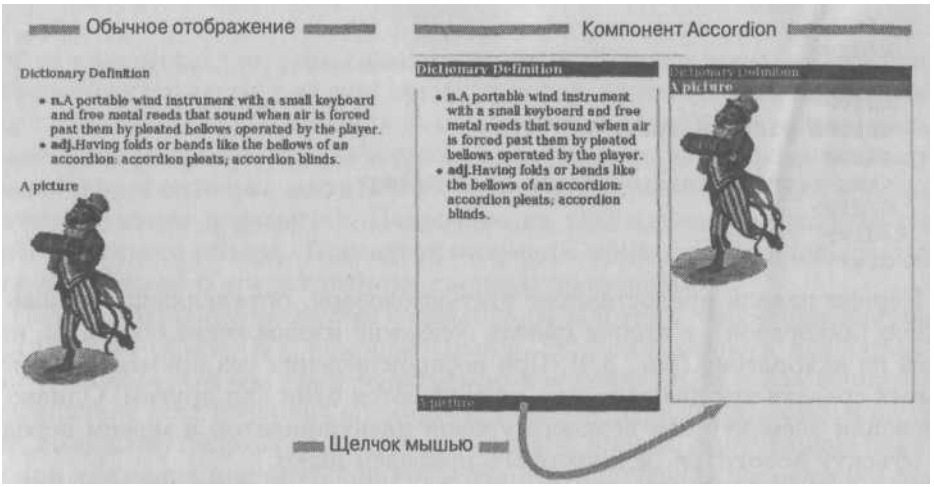


Рис. 3.9. Средства Behavior в составе Rico позволяют оформлять обычные узлы DOM в виде интерактивных компонентов. Для того чтобы сделать это, достаточно передать ссылку на узел верхнего уровня конструктору объекта Behavior. В данном случае объект Accordion применен к набору элементов div (его внешний вид, полученный без использования Accordion, показан слева). В результате формируется интерактивный компонент (он представлен справа), в котором по щелчку мыши открываются и закрываются отдельные панели

типов ответа. Один из этих типов предполагает непосредственное обновление элементов DOM, а второй — замену объектов JavaScript. Аналогичный механизм будет более подробно рассмотрен в разделе 5.5.3 при обсуждении взаимодействия клиента и сервера. Теперь перейдем к рассмотрению базовых средств, которые воздействуют на работу как клиента, так и сервера.

3.5.3. Элементы, располагаемые на стороне сервера

Средства, которые мы рассматривали до сих пор, выполняются исключительно в среде браузера и могут обсуживаться любым Web-сервером как обычные JavaScript-файлы. Сейчас мы рассмотрим программные компоненты, которые располагаются на стороне сервера и динамически генерируют JavaScript-код или элементы разметки HTML.

Эти компоненты гораздо сложнее рассмотренных ранее. Данные средства мы также не будем обсуждать подробно, а ограничимся лишь кратким рассмотрением их возможностей. К программным средствам, располагаемым на стороне сервера, мы вернемся в главе 5.

DWR, JSON-RPC и SAJAX

Начнем наш разговор о программных средствах, выполняющихся на стороне сервера, с обсуждения трех продуктов. Мы рассматриваем их совместно, поскольку в них, несмотря на различие языков, применяется один и тот же подход. SAJAX работает с разными языками, которые используются для создания программ, выполняющихся на стороне сервера, а именно: PHP,

python, Perl и Ruby. DWR (Direct Web Remoting) базируется на языке Java и вместо отдельных функций предоставляет методы объектов. JSON-RPC (JavaScript Object Notation-based Remote Procedure Calls) поддерживает работу с JavaScript для сервера, Python, Ruby, Perl и Java.

Все три продукта позволяют создавать на стороне сервера объекты, методы которых непосредственно доступны для запросов Ajax. В процессе работы Ajax-приложений часто используются функции, выполняющиеся на сервере и возвращающие результаты вычислений. Необходимость в подобных функциях возникает по разным причинам, например, с их помощью можно обрабатывать данные, полученные из базы. Рассматриваемые здесь средства обеспечивают доступ к таким функциям или методам из программ, выполняющихся в среде браузера, в частности, предоставляют клиентскому коду программные средства, реализующие модель.

Рассмотрим пример использования SAJAX для работы с PHP-функциями, определенными на сервере. В данном случае функция, выбранная в качестве примера, лишь возвращает строку текста.

```
<?php
function sayHello(name){
 return("Hello! {$name} Ajax in Action!!!!");
?>
```

Для того чтобы представить эту функцию JavaScript-программе на клиентском уровне, нам надо импортировать в PHP-программу процессор SAJAX и вызвать функцию sajax_export.

```
<?php
require('Sajax.php');
sajax_init();
sajax_export("sayHello");
?>
```

После создания динамической Web-страницы мы используем SAJAX для генерации JavaScript-оболочки экспортируемых функций. Сгенерированный код создает функцию JavaScript, заголовок которой идентичен функции на стороне сервера.

```
<script type='text/javascript'>
<?
 sajax_show_javascript();
?>

alert(sayHello("Dave"));

</script>
```

При вызове функции sayHello ("Dave") в среде браузера сгенерированный JavaScript-код формирует Ajax-запрос серверу, функция выполняется на стороне сервера и результаты возвращаются в составе HTTP-ответа. При разборе ответа извлекается возвращаемое значение и передается JavaScript-программе. Разработчику не приходится непосредственно применять технологии Ajax; все необходимые действия скрыты от него и выполняются библиотеками SAJAX.

Все три рассматриваемых здесь продукта осуществляют низкоуровневое отображение функций на стороне сервера в клиентские вызовы Ajax. Они автоматизируют рутинные операции, требующие много времени и усилий, но при их использовании возникает опасность доступа к серверной логике из Интернета. Этот вопрос мы подробнее рассмотрим в главе 5.

Остальные средства, рассматриваемые в этом разделе, действуют более сложным образом — они генерируют уровни пользовательского интерфейса для моделей, объявленных на сервере. Несмотря на использование технологий Ajax, они следуют своим собственным программным моделям. В результате работа с этими продуктами несколько отличается от написания универсальных Ajax-приложений.

Backbase

Backbase Presentation Server предоставляет обширный набор компонентов, связываемых в процессе работы с XML-дескрипторами. Они включаются в HTML-документы, сгенерированные сервером. Общий принцип работы аналогичен работе компонентов Rico Behavior, за исключением того, что вместо HTML-дескрипторов Backbase использует для обозначения компонентов пользовательского интерфейса произвольно задаваемый набор XHTML-элементов.

Backbase предоставляет реализацию серверных компонентов для Java и .NET и распространяется на коммерческой основе.

Echo2

Продукт NextApp Echo2 представляет собой процессор на базе Java, который генерирует богатые компоненты пользовательского интерфейса, объявленные на сервере. При загрузке в браузер компоненты работают автономно; взаимодействие с пользователями обеспечивает сценарий JavaScript. При необходимости они обращаются к серверу, используя очередь запросов, подобную той, которая применяется Rico.

Echo2 считается решением на базе Ajax. Для работы с ним не требуется знание HTML, JavaScript или CSS, если, конечно, пользователь не собирается расширять набор доступных компонентов. В большинстве случаев разработка клиентских приложений осуществляется с использованием только средств Java. Исходные коды Echo2 открыты. Допускается применение данного продукта в коммерческих целях.

Ruby on Rails

Ruby on Rails представляет собой набор средств для разработки приложений, написанных на языке программирования Ruby. В рамках данного продукта объединены решения для отображения серверных объектов в базу данных и представления содержимого с помощью шаблонов. В основном он соответствует архитектуре "модель-представление-контроллер" для сервера, которая была описана в разделе 3.4. Ruby on Rails обеспечивает быструю разработку как простых Web-узлов, так и средней сложности. Генерация ко-

пов общего назначения осуществляется автоматически. Количество установок, необходимых для получения работающего приложения, также сведено к минимуму.

Последние версии Rails обеспечивают поддержку Ajax посредством библиотеки Prototype. Prototype и Rails естественным образом сочетаются друг с другом, поскольку JavaScript-код для Prototype сгенерирован из программ Ruby и стили программирования совпадают. Как и в случае с Echo2, Rails не требует знания Ajax-технологий, в частности JavaScript, однако разработчик, знакомый с этим языком, может расширить средства поддержки Ajax.

На этом мы заканчиваем обзор продуктов независимых производителей для Ajax. Как уже было сказано ранее, работа над большинством рассмотренных здесь средств продолжается.

Разработчики многих библиотек придерживаются собственного стиля программирования, поэтому при написании примеров для данной книги мы старались передать характерные черты технологий Ajax и в то же время избежать глубокого изучения конкретных продуктов. Некоторые из рассмотренных здесь средств будут встречаться и далее в книге.

3.6. Резюме

В данной главе реструктуризация рассматривалась как средство для повышения качества кода и обеспечения дополнительной степени гибкости программ. Мы начали с того, что использовали реструктуризацию для упрощения доступа к объекту XMLHttpRequest. Был рассмотрен ряд образов разработки, которые можно применять при работе с Ajax для решения проблем, типичных для различных задач программирования. Образы разработки представляют собой полуформальные средства представления знаний разработчиков, которые решали задачи определенных типов. Они могут быть использованы при реструктуризации, преследующей определенные цели.

Образы Facade и Adapter предоставляют удобные способы устранения различий между разными реализациями. В Ajax-приложениях эти образы могут быть использованы для создания дополнительного уровня кода, скрывающего различия браузеров. Следует заметить, что несоответствие браузеров является.постоянным источником проблем при работе с JavaScript.

Observer — это гибкий образ для работы с системами, управляемыми событиями. Мы вернемся к его использованию в главе 4 при рассмотрении уровней пользовательского интерфейса приложения. Будучи использованным совместно с Command, предоставляющим удобный способ инкапсуляции действий пользователя, Observer может быть применен для поддержки ввода данных и отмены выполненных действий. Как вы увидите в главе 5, образ разработки Command также находит применение в организации взаимодействия клиента и сервера.

Образ разработки Singleton обеспечивает доступ к конкретным ресурсам. В Ajax этот образ часто применяется для управления обменом по сети. Данный вопрос также будет обсуждаться в главе 5.

В этой главе мы подробно обсудили архитектуру "модель-представление-контроллер", хорошо зарекомендовавшую себя в различных областях (в том числе и при создании программ для работы в Интернете). В частности, данную архитектуру удобно использовать для создания Web-приложений. Мы говорили о том, каким образом MVC способствует повышению степени гибкости программ, выполняемых на стороне сервера, в том числе обсудили важность уровня абстрактных данных и шаблонов.

Пример программного обеспечения интерактивного магазина продемонстрировал совместное применение образов разработки и реструктуризации. С первого раза трудно создать совершенный код, поэтому реструктуризация является сложным, но необходимым этапом работы, призванным обеспечить преимущества образов разработки для конкретной программы. Конечный результат существенно отличался от программы, полученной на первом этапе разработки.

И наконец, мы рассмотрели библиотеки и компоненты независимых производителей. Они также могут быть применены для упорядочения кода, создаваемого в рамках Ajax-проектов. В настоящее время доступны средства, которые могут быть использованы при решении различных задач: от формирования оболочек, скрывающих различия между браузерами, до программных решений, включающих создание клиентского и серверного кода. Кроме того, в главе описано несколько наиболее популярных продуктов, и к некоторым из них мы еще вернемся в этой книге.

В следующих двух главах мы применим полученные знания о реструктуризации и образах разработки при создании клиентских программ Ajax и систем, предполагающих взаимодействие клиента и сервера. Эти примеры подскажут вам способы решения задач, с которыми вы наверняка столкнетесь при разработке сложных Web-приложений.

3.7. Ресурсы

Мартин Фаулер (Martin Fowler), а также его соавторы Кент Бек (Kent Beck), Джон Брант (John Brant), Вильям Опдайк (William Opdyke) и Дон Роберте (Don Roberts) написали хорошее руководство по реструктуризации: *Refactoring: Improving the Design of Existing Code* (Addison-Wesley Professional, 1999).

Эрик Гамма (Erich Gamma), Ричард Хелм (Richard Helm), Ральф Джонсон (Ralph Johnson) и Джон Влиссидс (John Vlissides), известные как "команда четырех", написали книгу *Design Patterns* (Addison-Wesley Professional, 1995), которую высоко оценили многие разработчики программного обеспечения.

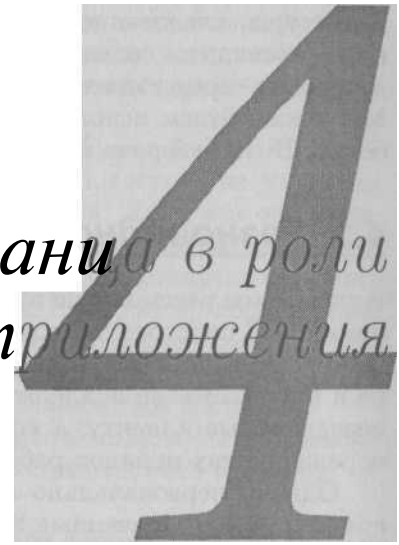
Впоследствии Эрик Гамма занялся архитектурой интегрированной среды разработки Eclipse (подробнее о ней речь пойдет в приложении А). С текстом интервью, в котором обсуждались Eclipse и образы разработки, можно ознакомиться по адресу http://www.artima.com/lejava/articles/gammadp.html.

Майкл Мехемофф (Michael Mahomoff) поддерживает Web-узел, посвященный образам разработки для Ajax (http://www.ajaxpatterns.org).

Часть II

Основные подходы к разработке приложений

Теперь, когда вы имеете общее представление об Ajax, пора рассмотреть основные способы разработки приложений. Наша задача — создать работоспособную, надежную программу, простую в сопровождении, которая предоставляла бы пользователю удобный интерфейс. В главе 4 мы рассмотрим вопросы создания клиентского кода и постараемся добиться, чтобы средства CSS, HTML и JavaScript как можно меньше зависели друг от друга. Глава 5 посвящена взаимодействию клиента и сервера.



Web-страница в роли приложения

- Организация сложного кода пользовательского интерфейса
- Использование образа разработки "модель-представление-контроллер" с JavaScript-кодом
- Разделение представления и логики для сопровождаемого кода
- Создание гибкого режима обработки событий
- Генерация пользовательского интерфейса

В главах 1 и 2 мы рассмотрели основные принципы Ajax как с точки зрения практичности программы, так и с позиции используемых технологий. В главе 3 речь шла о создании кода, пригодного для сопровождения. Чтобы сформировать такой код, мы использовали реструктуризацию и образы разработки. Для простых примеров, которые приводились в книге, такие сложные действия могли показаться излишними, однако, изучив глубже принципы программирования Ajax, вы убедитесь, что данные инструменты крайне необходимы.

В этой и следующей главах мы подробно обсудим вопросы, относящиеся к созданию сложного масштабируемого клиента Ajax, и принципы архитектуры, следование которым позволяет выполнить эту работу. Данная глава посвящена созданию клиентского кода, соответствующего архитектуре "модель-представление-контроллер", которая была рассмотрена в главе 3. Мы также будем использовать Observer и некоторые другие образы разработки. В главе 5 речь пойдет о взаимодействии клиента и сервера.

4.1. Разновидности архитектуры MVC

В главе 3 мы рассмотрели пример реструктуризации в соответствии с принципами MVC простого приложения, предназначенного для интерактивного магазина. Большинство разработчиков считают моделью информацию на сервере и программы ее поддержки, представлением — сгенерированные данные, передаваемые клиенту, а контроллером — сервлет или набор Web-страниц, определяющих порядок работы приложения.

Однако первоначально архитектура MVC была ориентирована на программы, предназначенные для настольных систем. В Ajax-приложениях существует несколько вариантов ее применения. Рассмотрим эти варианты более подробно.

4.1.1. Применение архитектуры MVC к программам различных уровней

Классическая архитектура "модель-представление-контроллер" в применении к Web описывает приложение лишь в общих чертах. Так, сгенерированный поток данных является представлением, CGI-сценарий или сервлет выполняет функции контроллера и т.д.

При разработке приложений для настольной системы архитектура MVC часто применяется в меньших масштабах. В соответствии ,с этой архитектурой могут создаваться даже такие простые компоненты, как кнопка. В этом случае элементы MVC можно описать следующим образом.

- Внутреннее представление состояний кнопки — например, нажата, отпущена, неактивна, — это модель. В компонентах Ajax она обычно реализуется посредством объекта JavaScript.
- Изображение компонента на экране (в случае пользовательского интерфейса он может быть сформирован из нескольких узлов DOM),

учитывающее различные состояния, с подсветкой и подсказкой, — это представление.

- Код, имеющий доступ к состоянию и позволяющий изменять изображение на экране, — это контроллер. Контроллером также являются обработчики событий (например, щелчок мышью на кнопке), но этот контроллер не имеет отношения к описанным выше представлению и модели.

Рассмотрим эти вопросы подробнее. Кнопка сама по себе демонстрирует крайне ограниченные варианты поведения. Ее состояние и внешний вид также изменяются крайне незначительно. Следовательно, применение к ней архитектуры MVC не даст практически никакого эффекта. Если мы рассмотрим более сложный компонент, например дерево или таблицу, то увидим, что он достаточно сложен для того, чтобы всерьез обсуждать вопрос использования MVC.

На рис. 4.1 условно показана архитектура MVC, применяемая к компоненту, реализующему древовидную структуру. Модель состоит из узлов дерева, каждый из которых имеет набор дочерних узлов. Для узла определены открытое и закрытое состояние. Каждый узел ссылается на некоторый бизнес-объект, представляющий файлы и каталоги. Представление состоит из пиктограмм и линий, расположенных на холсте. Контроллер поддерживает события, например, открытие и закрытие узла, отображение контекстного меню, и может инициировать обновление графического представления определенных узлов. В этом случае представление изменяет свое состояние.

Таким образом, архитектура MVC не обязательно должна быть привязана к знакомому всем сценарию Web-сервера. Обратимся к Web-браузеру.

4.1.2. Применение архитектуры MVC к объектам, присутствующим в среде браузера

До сих пор мы обращали внимание лишь на отдельные детали приложения. Но мы также можем целиком рассмотреть JavaScript-приложение, доставляемое браузеру в начале работы. Оно поддается структурированию в соответствии с архитектурой MVC и позволяет тем самым добиться разделения функций и ответственности между различными подсистемами.

На уровне браузера модель состоит из бизнес-объектов, представление — это страница, управляемая с помощью программы, а контроллером является сочетание всех обработчиков событий, связывающее пользовательский интерфейс с бизнес-объектами. На рис. 4.2 иллюстрируется использование MVC на этом уровне. Такой вариант применения архитектуры MVC, наверное, наиболее важен для разработчиков Ajax-приложений, так как богатое клиентское приложение Ajax естественным образом укладывается в ее рамки. Ниже мы рассмотрим данный вариант использования архитектуры "модель-представление-контроллер" и выгоды от его применения.

Если вы вспомните традиционное применение MVC для создания серверных программ (см. главу 3), то легко сделаете вывод, что в типичном Ajax-приложении можно выделить по крайней мере три уровня использования

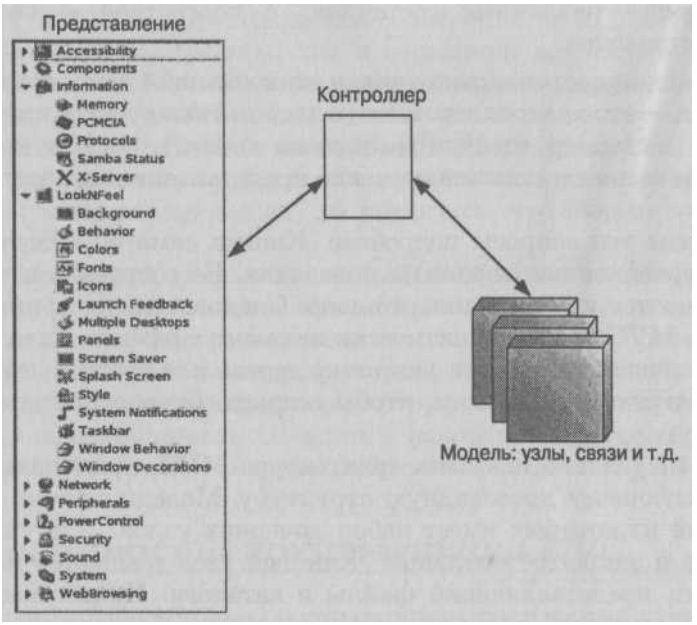


Рис. 4.1. Архитектура "модель-представление-контроллер" в применении к объекту, представляющему дерево. Представление состоит из элементов, отображаемых на экране и соответствующих элементам DOM. Моделью древовидной структуры является набор объектов JavaScript. Контроллер обеспечивает взаимодействие модели и представления

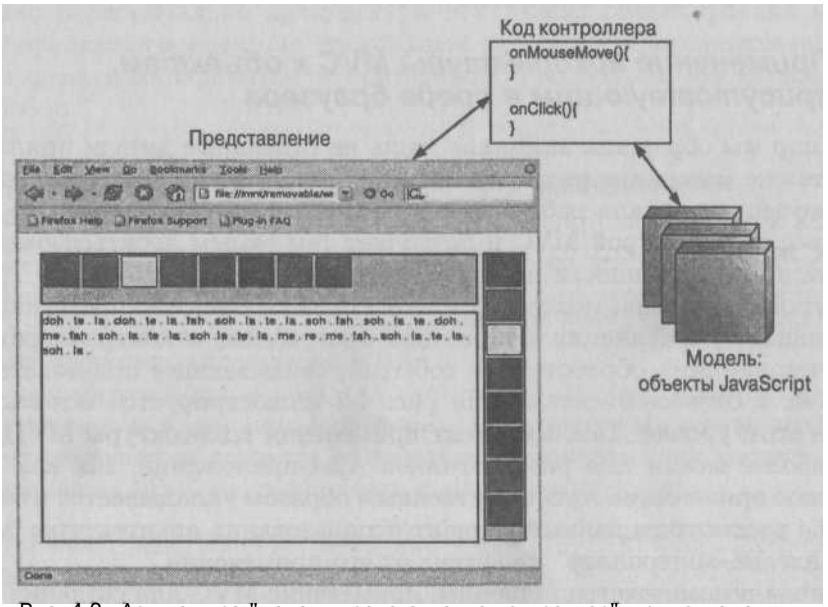


Рис. 4.2. Архитектура "модель-представление-контроллер", применяемая к клиентской части приложения Ajax. На этом уровне контроллером является JavaScript-код, связывающий пользовательский интерфейс с бизнес-объектами

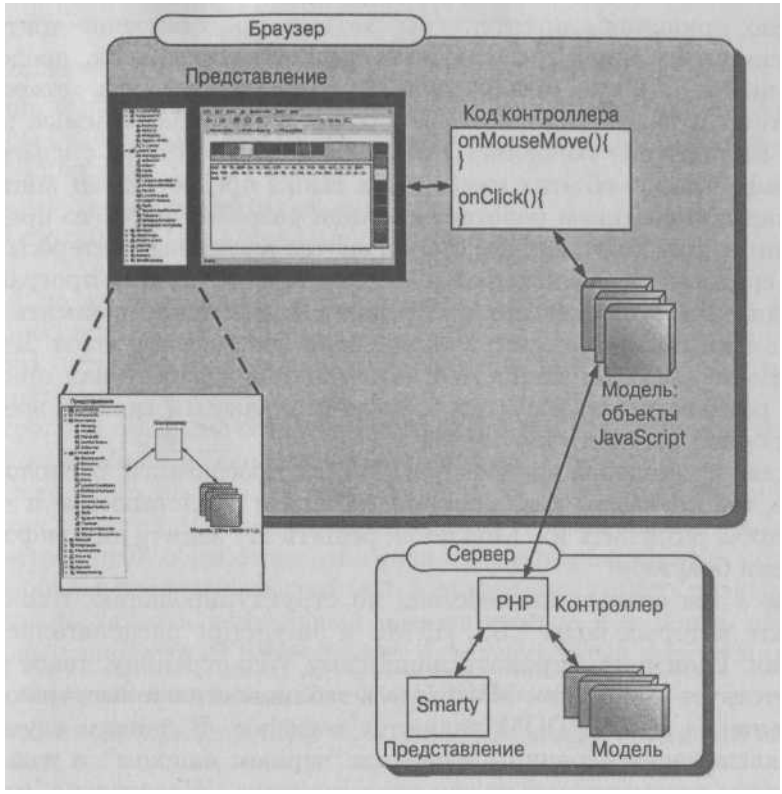


Рис. 4.3. Вложенная архитектура MVC

данной архитектуры. В жизненном цикле приложения средства MVC каждого уровня выполняют различные функции, но все они служат формированию понятного, хорошо организованного кода. На рис. 4.3 показано, как элементы MVC, присутствующие на различных уровнях, сочетаются друг с другом в рамках архитектуры приложения.

Как же данный подход влияет на нашу работу над кодом? В следующих разделах мы подробно рассмотрим использование MVC для определения структуры JavaScript-приложения, особенности написания кода и преимущества, получаемые при этом. Начнем наше рассмотрение с представления.

4.2. Представление в составе Ajax-приложения

С точки зрения JavaScript-приложения, доставленного браузеру в начале работы, представлением является отображаемая страница, состоящая из элементов DOM, основой для воспроизведения которых является HTML-разметка или действия программы. О принципах программной обработки элементов DOM см. в главе 2.

Согласно принципам архитектуры "модель-представление-контроллер" представление выполняет две основные функции. Во-первых, предоставляет визуальный интерфейс, посредством которого пользователь может выполнять действия и вызывать тем самым события, обрабатываемые контроллером. Во-вторых, оно обновляет само себя в соответствии с изменениями модели. Информацию об этих изменениях также предоставляет контроллер.

Если над приложением работает команда разработчиков, то представление становится той областью, где сталкиваются различные интересы. Интерактивные средства Ajax-интерфейса определяют не только программисты, но также дизайнеры и художники. Предлагать дизайнеру написать код или привлекать программиста к рисованию — явно бесполезная затея. Даже если кто-либо имеет две специальности и выступает при работе над приложением в двух разных ролях, желательно, чтобы в каждый момент времени он занимался одной конкретной задачей.

Обсуждая применение архитектуры MVC к программам, расположенным на сервере, мы показали, как "сплетаются" код и представление и что надо сделать, чтобы разделить их. Можно ли решить эту задачу для информации, имеющейся в браузере?

В главе 3 мы описывали действия по структурированию Web-страниц, в результате которых коды CSS, HTML и JavaScript располагались в разных файлах. Если рассматривать лишь саму Web-страницу, такое разделение соответствует принципам MVC. Здесь таблицы стилей выступают в роли представления, а HTML/DOM являются моделью. В данном случае средства воспроизведения страницы являются "черным ящиком", а коды HTML и CSS должны рассматриваться как представление. Желательно, чтобы они поддерживались независимо друг от друга. Поместив JavaScript-код в отдельный файл, мы уже сделаем существенный шаг по пути разделения функций дизайнеров и программистов. Однако это только начало.

4.2.1. Отделение логики от представления

Даже если JavaScript-код находится в отдельном файле, представление и логика (модель и контроллер) вполне могут оказаться перепутанными. Предположим, что мы создали встроенный обработчик события на языке JavaScript.

```
<div class='importButton'
  onclick='importData("datafeed3.xml", raytextbox.value);'/>
```

Сделав это, мы "жестко" закодировали бизнес-логику в составе представления. Что такое datafeed3? Как обрабатывать значение mytextbox.value? Почему функции importData() передаются два параметра и каково их назначение? Дизайнер должен знать об этом.

Функция importData() относится к бизнес-логике. Согласно принципам MVC, представление и модель не должны непосредственно взаимодействовать друг с другом, поэтому они разделены дополнительным уровнем. Предположим, что мы переписали элемент div следующим образом:

```
<div class='importButton' onclick='importFeedData()'/>
```

Теперь, если мы определим обработчик события так, как показано ниже, параметры будут инкапсулированы не в анонимном обработчике, а в функции importFeedData().

```
function importFeedData(event){
    importData("datafeed3.xml", mytextbox.value);
}
```

Такой подход позволяет использовать обработчик в любом месте программы, разделяя функциональные возможности и следуя принципу DRY (напомним, что данная аббревиатура означает "don't repeat yourself" — не повторяться).

Контроллер пока еще остается встроенным в HTML-код, что может стать источником проблем в больших приложениях.

Для того чтобы разделить контроллер и представление, мы присоединим обработчик события программными средствами. Вместо того чтобы реализовывать встроенный обработчик, мы укажем некоторый маркер, который впоследствии будет извлечен программой. Маркеры могут быть разными. В частности, можно указать уникальный идентификатор и задавать обработчики для каждого элемента. HTML-элемент будет переписан следующим образом:

```
<div class='importButton' id='dataFeedBtn'>
```

Приведенный ниже код надо включить в функцию обратного вызова window, onload.

```
var dfBtn=document.getElementById('dataFeedBtn') ;
dfBtn.onclick=importFeedData;
```

Если мы хотим связать определенные действия с несколькими событиями, нам понадобится маркер, не являющийся уникальным. Для этой цели можно использовать класс CSS.

Добавление обработчиков событий с помощью CSS

Рассмотрим простой пример, в котором события мыши связываются с клавишами виртуального музыкального инструмента. В листинге 4.1 мы определили простую страницу. В составе дескрипторов стили не указаны.

Листинг 4.1. Файл musical.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0
Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html> <head> <title>Keyboard</title> <link rel='stylesheet'
type='text/ess' href='musical.css'/> <script
type='text/javascript' src='musical.js'X/script> <script
type='text/javascript'> window.onload=assignKeys; </script>
</head>
<body>
<div id='keyboard' class='musicalKeys'>
  <! О — Клавиши на "клавиатуре" —>
  <div class='do musicalButton'X/div>
```

```
    <div class='re musicalButton'x/div>
    <div class='mi musicalButton'X/div>
    <div class='fa musicalButton'x/div>
    <div class='so musicalButton'x/div>
    <div class='la musicalButton'x/div>
    <div class='ti musicalButton'x/div>
    <div class='do musicalButton'x/div>
</div> <div id='console'  class='console'> </div> </body>
</html>
```

В декларации DOCTYPE мы указали определение Strict лишь для того, чтобы показать, что такое решение возможно. Элементу keyboard присвоен уникальный идентификатор, но в дескрипторах, соответствующих отдельным клавишам, идентификатор отсутствует. Заметьте, что для элементов, представляющих клавиши О, указаны два класса. Класс musicalButton — общий для всех клавиш, а второй класс позволяет различать их. Соответствующие стили определены раздельно в файле musical.ess (листинг 4.2).

Листинг 4.2. Файл musical .ess

```
.body{
  background-color: white;
} .musicalKeys{
  background-color: #ffeOdO;
  border: solid maroon 2px;
  width: 536px;
  height: 68px;
  top: 24px;
  left: 24px;
  margin: 4px;
  position: absolute;
  overflow: auto;
} .musicalButton{
  border: solid navy lpx;
  width: 60px;
  height: 60px;
  position: relative;
  margin: 2px;
  float: left;
}
.do{ background-color: red; }
.re{ background-color: orange; }
.mi{ background-color: yellow; }
.fa{ background-color: green; }
.so{ background-color: blue; }
.la{ background-color: indigo; }
.ti{ background-color: violet;
} div.console{
  font-family: arial, helvetica;
  font-size: 16px;
  color: navy;
  background-color: white;
  border: solid navy 2px;
  width: 536px;
```

```
  height: 320px;
  top: 106px;
  left: 24px;
  margin: 4px;
  position: absolute;
  overflow: auto;
}
```

Стиль musicalButton определяет свойства, общие для всех клавиш. Второй стиль лишь задает цвет клавиши. Заметьте, что позиция элемента верхнего уровня непосредственно определена в пикселях, а для клавиш мы задаем свойство float. В результате такого решения вступают в действие средства компоновки, встроенные в браузер, и клавиши размещаются вдоль горизонтальной линии.

Связывание обработчиков событий с элементами

JavaScript-код, показанный в листинге 4.3, связывает обработчики событий с клавишами.

Листинг4.3. Файл musical,js

```
function assignKeys(){
  // Получение родительского элемента div
  var keyboard=document.getElementById("keyboard");

  // Перечисление дочерних элементов
  var keys=keyboard.getEleraentsByTagName("div");

  if (keys){
    for(var i=0;i<keys.length;i++){
      var key=keys[i];
      var   classes=(key.className).split("    " ) ;
      if (classes && classes.length>=2
       SS
classes[1]=="musicalButton"){
        var note=classes[0];
        // Добавление атрибута
        key.note=note;
        key.onmouseover=playNote;
      }
    J
  }
} function playNote(event){
  // Извлечение дополнительного атрибута
  var note=this.note;
  var
console=document.getElementById("console");
  if (note && console){
    console.innerHTML+=note+" . ";
  }
}
```

Функция assignKeys () вызывается в составе обработчика window. onload. (Если определить window.onload непосредственно, это ограничит переносимость приложения.) Мы находим элемент keyboard по его уникальному идентификатору, а затем используем функцию getElementsByTagName () для перебора всех дочерних элементов div. Для этого необходимо иметь представление о структуре страницы, зато дизайнер получает возможность перемещать элементы div, соответствующие клавишам, любым удобным для него способом.

Каждый элемент DOM, представляющий клавишу, возвращает посредством свойства className строку символов. Для преобразования строки в массив мы используем встроенную функцию String.split, после чего проверяем, принадлежит ли элемент классу musicalButton. Затем мы читаем оставшуюся часть строки, которая представляет ноту, и связываем ее с узлом DOM посредством дополнительного свойства. Это свойство будет прочитано в обработчике события.

Воспроизводить звуки посредством Web-браузера затруднительно, поэтому в данном примере мы лишь отображаем ноты на "консоли" под изображением клавиатуры. На рис. 4.4 показано действие клавиатуры. В данном случае нам удалось довольно удачно разделить роли. Если дизайнер переместит элементы DIV, соответствующие клавиатуре и консоли, в другое место документа, приложение будет по-прежнему работать и риск случайно изменить логику обработки событий сводится к минимуму. По сути, HTML-страница выполняет функции шаблона, в состав которого мы включаем переменные и логику выполнения. Это позволяет отделить логику от представления. Мы обработали данный пример вручную, чтобы детально продемонстрировать наши действия. При работе над реальным приложением целесообразно использовать для получения того же результата библиотеки независимых производителей.

Библиотека Rico (http://www.openrico.org/) поддерживает объекты Behavior и позволяет реализовать интерактивные возможности поддерева DOM. Компонент Rico Accordion рассматривался в разделе 3.5.2. Разделить элементы разметки HTML и интерактивные функции можно посредством библиотеки Behaviour (см. ссылку в конце данной главы), созданной Беном Ноланом (Ben Nolan). Данная библиотека позволяет связывать с элементами DOM код, предназначенный для обработки событий, используя селекторы CSS (см. главу 2). В предыдущем примере функция assignKeys () выбирала элемент с идентификатором keyboard, а затем извлекала все содержащиеся в нем элементы div. Используя селектор, мы можем выразить то же правило следующим образом:

```
•keyboard div
```

Применяя CSS, мы можем связать стиль со всеми элементами keyboard посредством данного селектора. Библиотека Behaviour.js также позволяет использовать обработчики событий следующим образом:

```
var myrules={ 'tkeyboard div' :
function(key){
    var classes=(key.className).split(" ");
```

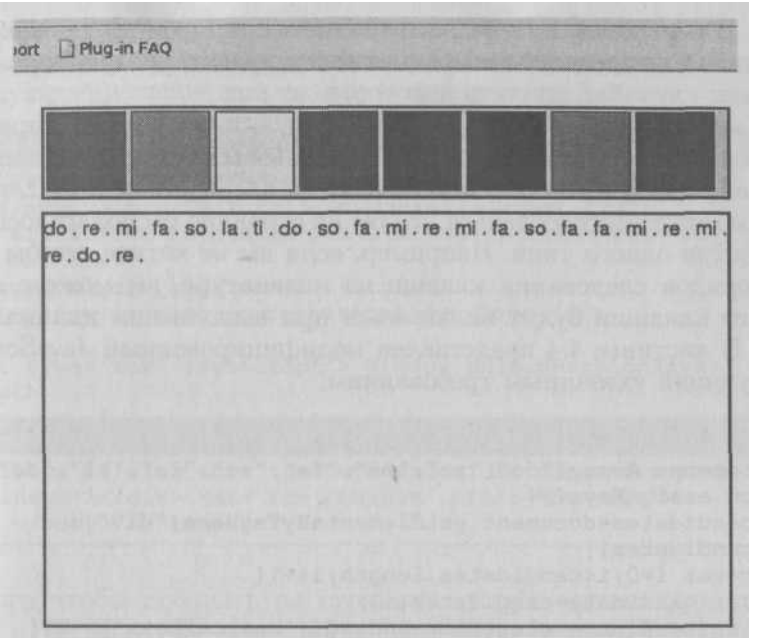


Рис. 4.4. Клавиатура "музыкального инструмента", представленная посредством браузера. Закрашенные области в верхней части окна соответствуют различным нотам. При перемещении указателя мыши над "клавишами" названия нот отображаются в области, расположенной ниже и выполняющей функции консоли

```
    if (classes && classes.length>=2
    &&
    classes[1]=='musicalButton')  {
      var note=classes[0];
      key.note=note;
      key.onmouseover=playNote;
    )
  }
};
```

Большая часть логики совпадает с предыдущим примером, но использование селекторов CSS представляет собой удобную альтернативу рассмотренному ранее решению, в особенности в тех случаях, когда надо одновременно реализовать несколько вариантов поведения.

В данном случае логика отделена от представления, однако, как мы вскоРе увидим, остается возможность включения средств представления в состав логики.

4.2.2. Отделение представления от логики

На данный момент мы добились того, что дизайнер может изменять внешний вид страницы, не затрагивая кода. Однако при существующем положении дел функционирование приложения еще существенно зависит от HTML-

документа. В частности, в HTML-коде задается порядок следования клавиш. Каждая клавиша определена посредством отдельного дескриптора div, и дизайнер может случайно удалить некоторые из них.

Если расположение клавиш — вопрос функционирования приложения, а не его оформления, то имеет смысл генерировать некоторые элементы DOM в программе, вместо того, чтобы объявлять их посредством HTML-кода. Кроме того, нам может потребоваться, чтобы на странице располагалось несколько компонентов одного типа. Например, если вы не хотите, чтобы дизайнер изменял порядок следования клавиш на клавиатуре, вы можете поставить условие, что клавиши будут включаться при выполнении инициализационного кода. В листинге 4.4 представлен модифицированный JavaScript-файл, соответствующий указанным требованиям.

Листинг 4.4. Файл musical_dyn_keys. js

```
var notes=new Array("do","re","mi","fa","so","la","ti","do");
function assignKeys(){
  var candidates=document.getElementsByTagName("div");
  if (candidates){
    for(var i=0;i<candidates.length;i++){
      var candidate=candidates[i];
      if (candidate.className.indexOf('musicalKeys')>=0){
        makeKeyboard(candidate);

    , '
  }
} function makeKeyboard(el){
  for(var i=0;i<notes.length;i++){
    var key=document.createElement("div");
    key.className=notes[i]+" musicalButton";
    key.note=notes[i];
    key.onmouseover=playNote;
    el.appendChild(key);
  }
}
function playNote(event){
  var note=this.note;
  var console=document.getElementById('console');
  if (note && console){
    console.innerHTML+«note+" . ";
  )
}                                                                    J
```

Ранее мы определяли клавиши в HTML-коде. Теперь мы задаем их в глобальном массиве JavaScript. Метод assignKeys () анализирует в документе все дескрипторы div, принадлежащие верхнему уровню, и находит те из них, свойство className которых содержит значение musicalKeys. Если такой элемент найден, предпринимается попытка заполнить его элементами div, соответствующими клавишам. Для этого используется функция makeKeyboard(). Она создает новые узлы DOM, а затем выполняет с ними такие же действия, которые ранее выполнялись с DOM-узлами, объявленными в HTML-коде. Функция обратного вызова playNote () действует так же, как и ранее.

Поскольку мы заполнили пустые элементы div элементами, представляющими клавиши, создание второго набора клавиш не представляет труда. Соответствующее решение представлено в листинге 4.5.

Листинг 4.5. Файл mus±cal_dyn_keys .html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtmll/DTD/xhtmll-strict.dtd">
<html> <title>Two Keyboards</title> <head> <link rel=*stylesheet'
type-'text/ess'
href-'musical_dyn_keys.ess'/>    <script
type='text/javascript'  src=•musical_dyn_keys.js'>
</script>
<script type='text/javascript'> window.onload=assignKeys;
</script>
</head>
<body> <div id='keyboard-top' class='toplong
musicalKeys'X/div> <div id='keyboard-side' class='sidebar
musicalKeys'X/div> <div id='console' class='console•>
</div> </body> </html>
```

Для того чтобы добавить на страницу вторую клавиатуру, достаточно включить в HTML-файл одну дополнительную строку. В данном случае мы не собираемся располагать клавиатуры одну под другой, поэтому стили, определяющие расположение, мы удалим из класса musicalKeys и оформим в виде отдельных классов. Модифицированные таблицы стилей показаны в листинге 4.6.

^Листинг 4.6. Модифицированные стили в файле musical_dyn_keys.css

```
/* Общие стили для клавиатуры */
.musicalKeys{
  background-color:  #ffe0d0;
  border: solid maroon 2px;
  position: absolute;
  overflow: auto;
  margin: 4px;
} /* Размеры и расположение первой клавиатуры */
.toplong{
  width: 536px;
  height: 68px;
  top: 24px;
  left: 24px;
} /* Размеры и расположение второй клавиатуры */
.sidebar{
  width: 48px;
  height: 400px;
  top: 24px;
  left: 570px;
}
```

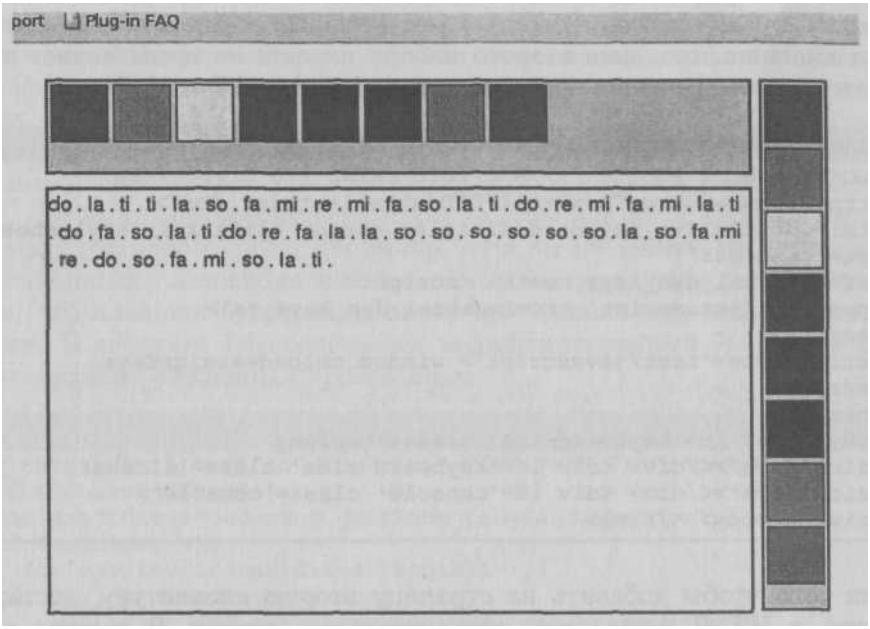


Рис. 4.5. Модифицированное приложение позволяет дизайнеру задавать несколько клавиатур. Используя CSS и средства воспроизведения в составе браузера, мы обеспечиваем горизонтальное и вертикальное расположение клавиш, не создавая для этого дополнительный JavaScript-код

В классе musicalKeys остались стили, общие для обеих клавиатур. Классы toplong и sidebar лишь задают их расположение.

Выполнив таким образом реструктуризацию, мы создали условия для повторного использования кода. Внешний вид клавиатуры частично определяется JavaScript-кодом; ее формированием занимается функция makeKeyboard(). Как видно на рис. 4.5, одна клавиатура располагается по горизонтали, а другая — по вертикали. Как мы достигли этого?

Функция makeKeyboard () позволяет без труда вычислить размер каждого элемента и определить его расположение. Если бы мы поступили так, то нам пришлось бы предпринимать меры для размещения клавиш по горизонтали или по вертикали. Специалисты, имеющие опыт разработки пользовательских интерфейсов для Java-программ и применявшие диспетчеры компоновки LayoutManager, могут склониться именно к такому решению. Если мы поступим подобным образом, контролировать внешний вид компонентов будут не дизайнеры, а программисты, в результате конфликты станут неизбежны.

Таким образом, принято решение, согласно которому функция makeKey-j board () влияет только на структуру документа. Размещение клавиш осу-! ществляется средствами браузера с учетом таблиц стилей; в данном случае используется стиль float. Важен тот факт, что размещение элементов контролирует дизайнер. Программная логика и представление — разделены.

Клавиатура — это относительно простой компонент. В более сложных случаях, например, при организации дерева, бывает трудно обеспечить желае-

мое размещение элементов посредством встроенных средств браузера, и программная реализация стилей становится неизбежной. Однако необходимо помнить о важности разделения представления и логики и стремиться добиться этого. Средства воспроизведения в составе браузера обычно обеспечивают высокую производительность и работают надежно. Чтобы добиться такого же качества JavaScript-программы, приходится затрачивать много усилий.

На этом мы заканчиваем работу над представлением. В следующем разделе будет рассмотрена роль контроллера в рамках архитектуры MVC и его реализация в Ajax-приложении посредством JavaScript-обработчиков событий.

4.3. Контроллер в составе Ajax-приложения

Контроллер в архитектуре MVC выполняет роль посредника между моделью и представлением. В программе с графическим пользовательским интерфейсом, в частности, в составе клиентских средств Ajax, контроллер состоит из обработчиков событий. Эволюция клиентских программ для Web привела к тому, что в современных браузерах поддерживаются две различные модели событий. Одна из них, классическая модель, относительно проста, однако в настоящее время происходит ее замена новой моделью обработки, спецификация которой разработана W3C. На момент написания данной книги новая модель была по-разному реализована в различных браузерах, и ее использование было сопряжено с проблемами. В данном разделе мы обсудим обе модели.

4.3.1. Классические JavaScript-обработчики

Реализация JavaScript в Web-браузерах позволяет определять код, выполняемый в ответ на событие. Обычно событиями являются действия с мышью или клавиатурой. В современных браузерах, поддерживающих Ajax, обработчики этих событий можно связать с большинством элементов, отображающихся на экране. Обработчики обеспечивают взаимодействие визуального пользовательского интерфейса, т.е. представления, и бизнес-объектов, составляющих модель.

Классическая модель поддержки событий достаточно проста и используется с момента появления JavaScript. В большинстве элементов DOM предусмотрены свойства, позволяющие связывать с ними функции обратного вызова. Например, для того, чтобы присоединить функцию, которая будет вызываться по щелчку на элементе myDomElement, мы должны написать следующее выражение:

```
myDomElement.onclick=showAnimatedMonkey
```

Здесь myDomElement — любой элемент DOM, к которому мы имеем доступ из программы. Функция showAnimatedMonkey() — это обычная функция JavaScript, которая определяется следующим образом:

Таблица 4.1. Свойства, позволяющие связывать обработчики событий с элементами DOM

Свойство	Описание
onmouseover	Активизируется, когда курсор мыши попадает в область, занимаемую элементом
onmouseout	Активизируется, когда курсор мыши покидает область, занимаемую элементом
onmousemove	Активизируется при каждом перемещении курсора мыши в пределах области, занимаемой элементом
onclick	Активизируется щелчком в области, занимаемой элементом
onkeypress	Активизируется щелчком мышью при условии, что элемент, над которым находится курсор, обладает фокусом ввода
onfocus	Активизируется, когда видимый элемент получает фокус ввода
onblur	Активизируется, когда видимый элемент теряет фокус ввода

```
function showAnimatedMonkey(){
   // Сложный код для поддержки движущихся изображений }
```

Заметьте, что при связывании обработчика события мы не вызываем функцию, а передаем объект Function, поэтому круглые скобки после имени не указываются. Ниже приведен пример распространенной ошибки.

```
myDomElement.onclick=showAnimatedMonkey();
```

Для некоторых программистов такая запись более привычна, однако приводит к результату, отличному от желаемого. Функция будет выполняться не в результате щелчка на элементе DOM, а в процессе присвоения, свойство onclick получит значение, возвращаемое функцией. Возможно, это сложное решение, при котором в результате выполнения одной функции возвращается ссылка на другую, но подобный подход используется достаточно редко и требует от программиста высокой квалификации. Еще раз покажем, как правильно выполнять связывание обработчика.

```
myDomElement.onclick=showAnimatedMonkey;
```

В результате ссылка на функцию обратного вызова присваивается элементу DOM, сообщая ему о том, что функция должна быть вызвана щелчком на элементе. Элемент DOM может содержать несколько свойств, предназначенных для связывания обработчиков событий. Свойства, часто используемые при создании графического пользовательского интерфейса, описаны в табл. 4.1. Встречавшиеся ранее свойства XMLHttpRequest.onreadystate и window. onload также предназначены для связывания обработчиков.

Механизм обработки событий имеет одну особенность, которую часто не учитывают программисты, использующие JavaScript. Эту особенность необходимо знать всем, кто собирается приступать к работе над клиентскими программами Ajax.

Предположим, что мы связали с элементом DOM функцию обратного вызова, используя для этого свойство onclick. Эта функция получает управление по щелчку мышью на элементе DOM. Однако контекст функции (т.е. значение, получаемое переменной this) соответствует узлу DOM, который получаем событие (объекты Function в языке JavaScript обсуждаются в приложении Б). Контекст зависит от того, как была объявлена функция. Этот эффект может привести к возникновению ошибки.

Рассмотрим пример. Предположим, что мы определили класс, представляющий объект кнопки. В нем указан узел DOM, обработчик обратного вызова и значение, которое отображается после щелчка мышью на кнопке. Все экземпляры кнопки одинаковым образом реагируют на щелчок, потому мы определяем обработчик как метод класса. Теперь обратимся к JavaScript-коду. Конструктор кнопки имеет следующий вид:

```
function Button(value,domEl){
  this.domEl=domEl;
  this.value=value;
  this.domEl.onclick=this.clickHandler;
}
```

Определим обработчик в составе класса Button.

```
Button.prototype.clickHandler=function(){
  alert(this.value);
}
```

Такое решение выглядит достаточно просто, но полученный результат — совсем не тот, которого можно было бы ожидать. В окне с сообщением отображается не значение, которое было передано конструктору, а сообщение undefined. Это происходит по следующей причине. Браузер вызывает функцию clickHandler() по щелчку на элементе DOM, поэтому задает контекст элемента DOM, а не JavaScript-объекта Button. Таким образом, this.value ссылается на свойство value элемента DOM, а не объекта Button. Это трудно предположить, глядя на определение обработчика события, не правда ли?

Разрешить данную проблему можно, модифицировав конструктор следующим образом:

```
function Button(value,domEl){
  this.domEl=domEl;
  this.value=value;
  this.domEl.buttonObj=this;
  this.domEl.onclick=this.clickHandler;
}
```

Элемент DOM по-прежнему не имеет свойства value, но содержит ссылку на объект Button, который используется для получения значения. Кроме того, нам надо изменить обработчик события.

```
Button.prototype.clickHandler=function(){
  var buttonObj=this.buttonObj;
  var value=(buttonObj && buttonObj.value) ?
    buttonObj.value : "unknown value";
  alert(value);
}
```

Узел DOM ссылается на объект Button, который ссылается на свойство value, и наш обработчик получает требуемое значение. Мы можем присвоить значение непосредственно узлу DOM, но, связывая ссылку с базовым объектом, мы получаем возможность работать с элементами произвольной сложности. Следует заметить, что в данном случае был реализован "мини-вариант" архитектуры MVC, согласно которой элемент DOM, выполняющий функции просмотра, отделен от объекта поддержки, являющегося моделью.

Мы рассмотрели классическую модель обработки событий. Основной недостаток этой модели состоит в том, что она допускает для элемента наличие лишь одной функции обработки. При обсуждении образа разработки Observer в главе 3 мы говорили о том, что может потребоваться связать с элементом Observable любое количество элементов Observer. Указанное здесь ограничение не является серьезным препятствием в работе над простым сценарием, но при переходе к более сложным клиентам Ajax оно может существенно помешать нам. Этот вопрос будет подробно рассмотрен в разделе 4.3.3, а пока перейдем к новой модели обработки событий.

4.3.2. Модель обработки событий W3C

Модель поддержки событий, предложенная W3C, обеспечивает большую степень гибкости, но она сложна в использовании. Согласно этой модели с элементом DOM может быть связано произвольное количество обработчиков. Более того, если действие происходит в области пересечения нескольких элементов, обработчики каждого из них имеют шанс получить управление и запретить остальные вызовы из стека событий. Этот эффект принято называть "проглатыванием" (swallowing) событий. Согласно спецификации осуществляются два прохода по стеку событий: первый — от внешнего элемента к внутреннему (т.е. от элемента, представляющего документ, к элементам, содержащимся в нем), в ходе которого происходит заполнение стека, а второй — от внутреннего элемента к внешнему. На практике различные браузеры реализуют подобное поведение по-разному.

В браузерах Mozilla и Safari функции обратного вызова, предназначенные для обработки событий, связываются с помощью функции addEventListener(), а для разрыва связи осуществляется обращение к removeEventListener(). Браузер Internet Explorer предоставляет для этих целей функции attachEvent() и detachEvent(). Объект xEvent, разработанный Майком Фостером (Mike Foster) и входящий в состав библиотеки x, предпринимает попытку реализовать средства обработки событий, маскирующие различия между браузерами. Для этой цели он создает единую точку входа в соответствии с образом разработки Fagade (см. главу 3).

Существуют также различия между браузерами, касающиеся обращений к обработчикам обратного вызова. В браузерах Mozilla, как и в классической модели обработки, контекст вызываемой функции определяется элементом DOM, получившим событие. В Internet Explorer контекст всегда определяется объектом Window, и в результате невозможно выяснить, какой элемент DOM вызвал функцию обработки. При написании обработчиков необходимо учитывать указанные различия, даже в случае применения объекта xEvent.

Следует также заметить, что ни одна из реализаций не предоставляет удобных средств получения списка всех обработчиков, связанных с элементом.

По этой причине мы не рекомендуем вам использовать новую модель обработки событий. Главный недостаток классической модели — отсутствие поддержки нескольких обработчиков — можно устранить, как вы узнаете далее, с помощью образов разработки.

4.3.3. Реализация гибкой модели событий в JavaScript

Из-за несоответствия средств поддержки новой модели событий в различных браузерах гибкие средства обработки действий пользователя пока нельзя считать доступными. В главе 3 мы описывали образ разработки Observer, который соответствует основным требованиям разработчиков и позволяет добавлять к источнику событий объекты Observer и удалять их. Таким образом обеспечивается необходимая степень гибкости. Очевидно, что специалисты W3C сделали все возможное, чтобы устранить недостатки классической модели, но производители браузеров не смогли обеспечить совместимость своих продуктов и, более того, в некоторых случаях допускали ошибки при реализации модели. Несмотря на то что классическая модель обработки событий не соответствует образу разработки Observer, мы постараемся исправить ситуацию, написав собственный код.

Управление несколькими функциями обратного вызова

Прежде чем реализовывать собственное решение, постараемся лучше понять проблему, а для этого рассмотрим простой пример. В листинге 4.7 приведен код документа, в котором элемент div реагирует на перемещение курсора мыши, выполняя два действия.

Листинг4.7. Файл mousemat.html

```
<html> <head> <link rel='stylesheet'
type='text/ess' href='mousemat.ess' /> <script
type='text/javascript'> var cursor=null;
window.onload=function(){
  var mat=document.getElementById('mousemat') ;
      mat.onmousemove=mouseObserver;
  cursor=document.getElementByld('cursor' ); }
  function mouseObserver(event){
  var e=event || window.event;
  writeStatus(e);
  drawThumbnail(e);
}
function writeStatus(e){
  window.status=e.clientX+","+e.clientY;
}
function drawThumbnail(e){
  cursor.style.left=((e.clientX/5)-2)+"px";
  cursor.style.top=((e.clientY/5)-2)+"px";
} </script> </head> <body> <div class='mousemat'
id='mousemat'X/div> <div class='thumbnail'
```

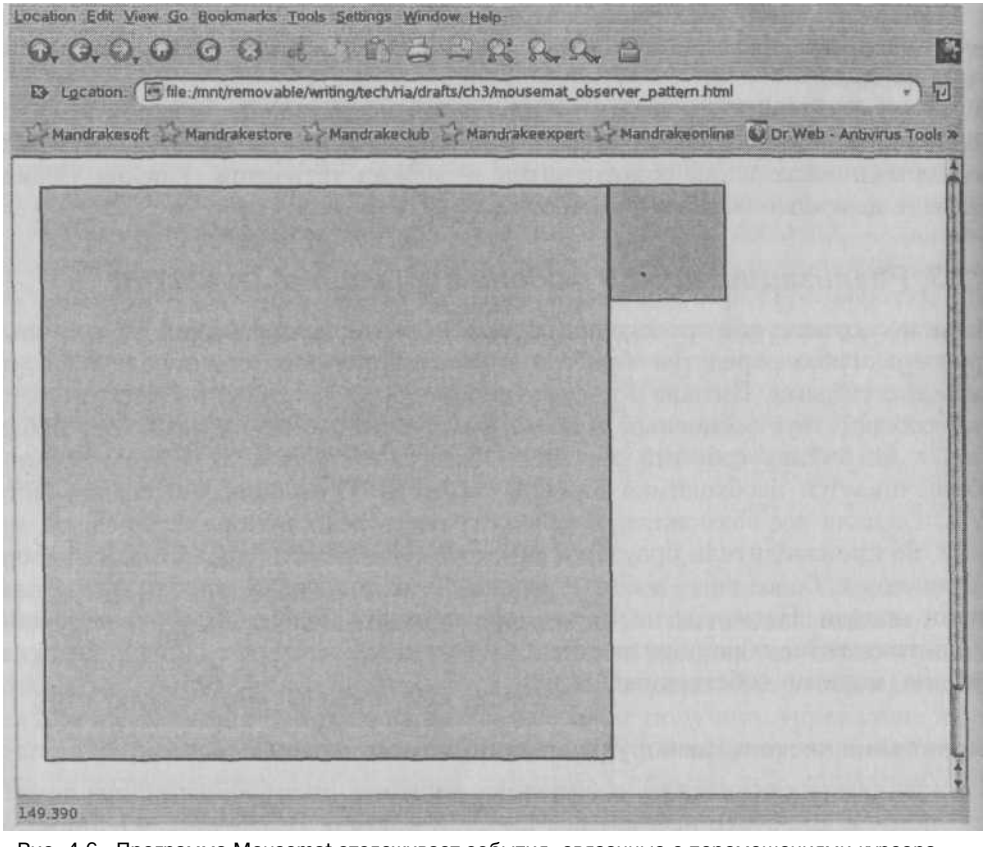


Рис. 4.6. Программа Mousemat отслеживает события, связанные с перемещениями курсора мыши в главной области, двумя способам: во-первых, координаты курсора мыши отображаются в строке состояния браузера, а во-вторых, в области малого размера перемещается точка, отражающая движения мыши

```
id='thumbnail'>
   <div class='cursor'
id='cursor'/> </div>
</body>
</html>
```

Во-первых, обработчик обновляет строку состояния браузера, вызывая для этого функцию writestatus(). Во-вторых, он обновляет небольшое изображение, изменяя в нем расположение точки, отражающей перемещение курсора мыши. Эти действия выполняются в функции drawThumbnail(). Внешний вид Web-страницы показан на рис. 4.6.

Эти два действия выполняются независимо друг от друга, и нам желательно обеспечить возможность менять их местами и включать между ними другие действия, являющиеся реакцией на перемещение курсора мыши.

Обработчиком события является функция mouseObserver(). (Первая строка в теле функции необходима лишь из-за различий браузеров. В отличие от Mozilla, Opera и Safari, браузер Internet Explorer не передает параметры обработчику обратного вызова, а сохраняет объект Event в window.event.) В данном примере обработчик по очереди вызывает функции writestatus() и drawThumbnail(). Программа выполняет свое назначения, и, поскольку она предельно проста, код функции mouseObserver() понятен. В общем случае нам надо разработать другой способ объединения обработчиков, который был бы пригоден для разработки сложных программ.

Реализация образа разработки Observer в JavaScript-программе

Решением данной задачи может быть универсальный объект-маршрутизатор, который связывал бы с целевым элементом стандартную функцию, получающую управление при возникновении события, и управлял бы набором функций-обработчиков. Для этого придется переписать код инициализации следующим образом:

```
window.onload=function() {
  var mat=document.getElementById('mousemat') ;

  var mouseRouter=new jsEvent.EventRouter(mat,"onmousemove");
  mouseRouter.addListener(writeStatus) ;
  mouseRouter.addListener(drawThumbnail);
}
```

Мы используем объект EventRouter и передаем ему в качестве параметра элемент DOM и тип события, которое хотим зарегистрировать. Затем мы включаем в объект-маршрутизатор функции-обработчики. Объект-маршрутизатор также поддерживает метод removeListener(), который мы здесь не используем. Применение нашего гипотетического объекта выглядит хорошо, но как мы реализуем сам объект?

Прежде всего нам надо написать конструктор объекта. В языке JavaScript это обычная функция. (В приложении Б содержится простое описание синтаксиса, используемого для создания объектов JavaScript. Если приведенный ниже код будет непонятным, обратитесь к этому описанию.)

```
j sEvent.EventRouter=function(el,eventType){
  this.lsnrs=new Array();:
  this.el=el;
  el.eventRouter=this;
  el[eventType]=j sEvent.EventRouter.callback;
}
```

Здесь мы определили массив функций-обработчиков; первоначально этот массив пуст. Затем мы получаем ссылку на элемент DOM и с помощью этой ссылки передаем элементу ссылку на текущий объект. Наши действия соответствуют образу разработки, описанному в разделе 3.5.1. Затем мы задаем в качестве обработчика события статический метод класса EventRouter с именем callback. Заметьте, что в JavaScript использование квадратных скобок а точки дают одинаковые результаты. Следовательно, два приведенных ниже выражения эквивалентны.

```
el.onmouseover
el['onmouseover']
```

Этот факт мы используем, передавая имя свойства в качестве параметра. Данное решение можно сравнить с механизмом отражения в Java или .NET.

Рассмотрим функцию обратного вызова.

```
jsEvent.EventRouter.callback=function(event){
  var e=event || window.event;
  var router=this.eventRouter;
  router.notify(e)
}
```

Контекстом для данной функции является не объект маршрутизации, а узел DOM. Мы извлекаем ссылку на EventRouter, которую присоединили к узлу DOM, используя прием, рассмотренный ранее. Затем вызываем метод notify () маршрутизатора, передавая ему в качестве параметра объект события.

Полностью код объекта-маршрутизатора показан в листинге 4.8.

Листинг 4.8. Файл EventRouter.js

```
var jsEvent=new Array();
jsEvent.EventRouter=function(el,eventType){
  this.lsnrs=new Array();
  this.el=el;
  el.eventRouter=this;
  el[eventType]=jsEvent.EventRouter.callback;
}
jsEvent.EventRouter.prototype.addListener=function(lsnr){
  this.lsnrs.append(lsnr,true);
}
jsEvent.EventRouter.prototype.removeListener=function(lsnr){
  this.lsnrs.remove(lsnr);
}
jsEvent.EventRouter.prototype.notify=function(e){
  var lsnrs=this.lsnrs;
  for(var i=0;i<lsnrs.length;i++){
    var lsnr=lsnrs[i]; lsnr.call(this,e);
  }
)
jsEvent.EventRouter.callback=function(event){
  var e=event I| window.event;
  var router=this.eventRouter;
  router.notify(e)
)
```

Заметьте, что некоторые из методов массива соответствуют не стандарту JavaScript, а расширенному определению массива, которое обсуждается в приложении Б. Функции addListener() и removeListener() легко реализовать, используя методы append() и remove(). Функции-обработчики вызываются с использованием метода Function, call(), первый параметр которого задает контекст функции, а последующие параметры (в данном случае — событие) передаются вызываемой функции.

Модифицированный вариант HTML-документа показан в листинге 4.9.

ЛИСТИНГ 4.9. Документ mousemat.html, использующий объект EventRouter

```
<html> <head> <link rel='stylesheet'
type='text/ess' href='mousemat.ess' /> <script
type='text/javascript' src='extras-array.js'X/script> <script
type='text/javascript' src='eventRouter.js'X/script> <script
type='text/javascript'> var cursor=null;
window.onload=function(){
  var mat=document.getElementById('mousemat');
 cursor=document.getElementById('cursor') ;
  var mouseRouter=new
jsEvent.EventRouter(mat,"onmousemove") ;
  mouseRouter.addListener(writeStatus) ;
  mouseRouter.addListener(drawThumbnail);
} function writeStatus(e){
  window.status=e.clientX+","+e.clientY
} function drawThumbnail(e){
  cursor.style.left-((e.clientX/5)-2)+"px";
  cursor.style.top=((e.clientY/5)-2)+"px";
} </script> </head> <body> <div class='mousemat'
id='mousemat'x/div> <div class=' thumbnail'
id='thumbnail'>
  <div class='cursor'
id='cursor'/> </div> </body> </html>
```

Встроенный код JavaScript существенно упрощен. В данном случае нам надо лишь создать объект EventRouter, передать ему функции обработки и реализовать эти обработчики. Дальнейшим развитием документа и поддерживающего его кода может стать включение флажков опций, посредством которых пользователь сможет динамически добавлять или удалять обработчики. Мы предлагаем читателю самостоятельно реализовать данную возможность.

На этом мы заканчиваем обсуждение уровня контроллера в Ajax-приложении и роли, которую образы разработки, в частности Observer, играют в создании понятного и пригодного для сопровождения кода. В следующем разделе мы рассмотрим последний элемент архитектуры MVC — модель.

4.4. Модель в составе Ajax-приложения

Модель представляет собой область применения приложения — интерактивный магазин, музыкальный инструмент и даже набор точек в пространстве. Document Object Model, или DOM, не является моделью Ajax-приложения. В данном случае роль модели выполняет код, написанный на языке JavaScript. Подобно другим образам разработки, архитектура MVC неразрывно связана с объектным подходом.

Разработчики JavaScript не задумывали его как объектно-ориентированный язык, однако в нем, не затрачивая слишком много усилий, можно организовать некоторое подобие объектов. Посредством механизма прототипов можно определить нечто, похожее на классы, а некоторые разработчики пытаются даже реализовать наследование. Эти вопросы мы обсудим в приложении Б. До сих пор, создавая на JavaScript приложение, соответствующее архитектуре "модель-представление-контроллер", мы придерживались стиля программирования, типичного для JavaScript, например, передавали объекты Function непосредственно обработчикам событий. Определяя модель, желательно использовать объекты JavaScript и придерживаться объектного подхода настолько, насколько это имеет смысл для данного языка. В последующих разделах мы покажем, как это можно сделать.

4.4.1. Использование JavaScript для моделирования предметной области

Обсуждая представление, мы все время упоминали элементы DOM. Говоря о контроллере, мы были ограничены рамками модели событий, реализованной в браузере. При создании модели мы будем иметь дело почти исключительно с кодом JavaScript и практически не будем учитывать возможности браузеров. Для тех программистов, которые вынуждены постоянно бороться с несовместимостью браузеров, подобные условия работы покажутся предельно комфортными.

Рассмотрим простой пример. В главе 3 обсуждалось приложение для интерактивного магазина, а внимание в основном уделялось генерации данных серверными программами. Эти данные определяли товары в терминах идентификаторов, имен и описаний. Кроме того, учитывались цена, цвет и размер. Вернемся к данному примеру и рассмотрим, что произойдет, когда информация будет доставлена клиенту. В процессе работы приложение получает несколько подобных потоков данных и должно хранить их в памяти. Область памяти для хранения данных можно рассматривать как кэш на стороне клиента. Есть возможность быстро отобразить находящуюся в нем информацию, не обращаясь к серверу. О преимуществах такого подхода см. в главе 1.

Определим простой JavaScript-объект, соответствующий изделию, поддерживаемому сервером. Пример такого объекта приведен в листинге 4.10.

Листинг 4.10. Файл Garment, js

```
var garments=new Array(); function
Garment(id,title,description,price){
  this.id=id;
  garments[id]=this;
  this.title=title;
  this.description=description;
  this.price=price;
  this.colors=new Object();
  this.sizes=new Object();
}
Garment.prototype.addColor(color){
```

```
        this.colors.append(color,true); }
Garment.prototype.addSize(size){
        this.sizes.append(size,true); )
```

Для хранения информации об изделиях определен глобальный массив. (Вы скажете, что использование глобальных переменных — плохой стиль программирования, и будете правы. При разработке реальных продуктов для этой цели обычно используются объекты пространств имен, однако в данном случае мы не будем поступать так, чтобы неоправданно не усложнять программу.) Это ассоциативный массив, для которого ключевыми значениями являются идентификаторы изделий. Такой подход гарантирует, что в каждый конкретный момент времени будет отображаться только одна ссылка на каждый тип товара. Все простые свойства, не являющиеся массивами, задаются в конструкторе. В момент определения массивы пусты. Для включения в них элементов используются специальные методы, которые обращаются к расширенному коду поддержки массивов (подробнее об этом — в приложении Б). Это также позволяет исключить дублирование данных.

Мы не создаем get- и set-методы и не реализуем полный контроль доступа посредством определения областей видимости для переменных и методов, что позволяют сделать объектно-ориентированные языки. Существуют способы обеспечения таких возможностей (они будут обсуждаться в приложении Б), но сейчас мы предпочитаем сделать модель как можно более простой.

При разборе потока XML-данных целесообразно сначала создать пустой объект Garment, а затем заполнять его информацией. Некоторые читатели удивятся, почему мы не предусмотрели более простой конструктор. На самом деле мы сделали это. Функции JavaScript позволяют задавать переменное количество параметров; недостающие параметры заменяются значениями null. Таким образом, два приведенных ниже вызова эквивалентны.

```
var garment=new Garment(123);
var garment=new Garment(123,null,null,null);
```

Идентификатор задавать необходимо, поскольку он используется в конструкторе для включения нового объекта в глобальный список изделий.

4.4.2. Взаимодействие с сервером

Для того чтобы генерировать объекты Garment в клиентской программе, нам надо организовать разбор XML-данных. О разборе мы уже говорили в главе 2, кроме того, мы вернемся к этому вопросу в главе 5, поэтому сейчас не будем детально обсуждать этот процесс. XML-документ содержит дескрипторы, атрибуты и содержимое элементов. Для чтения атрибутов мы используем свойство attributes и функцию getNamedItem(), а для чтения тела дескриптора — свойства firstChild и data. Например:

```
garment.description=descrTag.firstChild.data;
```

С помощью этого выражения можно произвести разбор следующего XML-Фрагмента:

```
<description>Large tweedy hat looking like an unappealing
strawberry </description>
```

Заметьте, что информация об изделиях автоматически помещается в глобальный массив по мере создания объектов; запись в массив предусмотрена в теле конструктора. Удалить изделие из массива также просто.

```
function unregisterGarment(id){
  garments[id]«null;
}
```

Данный фрагмент кода удаляет из глобального массива, выполняющего функцию реестра, информацию об изделии, но не разрушает уже созданный экземпляр объекта Garment. Поэтому имеет смысл проверить, используется ли объект Garment.

```
Garment.prototype.isValid=function(){
    return garments[this.id]!=null;
  }
```

На текущий момент мы определили на стороне клиента простой способ обработки информации, передаваемой с сервера. На каждом шаге обработки используются объекты, простые для восприятия и поддержки. Повторим еще раз основные этапы разработки. Сначала мы генерируем на стороне сервера объектную модель. Исходной информацией являются данные из базы. В разделе 3.4.2 было рассмотрено решение этой задачи с помощью инструмента ORM (Object-Relational Mapping), который естественным образом обеспечивает двустороннее взаимодействие объектной модели и базы данных. Мы оформляем данные в виде объектов, модифицируем и сохраняем их.

Далее используем систему шаблонов для генерации потока XML-данных и выполняем разбор этого потока, чтобы создать объектную модель на уровне JavaScript-программы. На этом этапе мы выполняем разбор вручную. Возможно, вскоре появятся специализированные библиотеки, ориентированные на решение подобных задач.

В приложении, предназначенном для администрирования, нам может понадобиться отредактировать данные, изменяя JavaScript-модель, а затем передать изменения серверу. При этом есть опасность появления двух копий модели, не синхронизированных друг с другом.

В классическом Web-приложении все "интеллектуальные" функции реализованы на сервере, следовательно, там расположена модель. Такое решение применяется независимо от используемого языка. В Aj ax-приложении мы распределяем обработку данных между клиентом и сервером, поэтому клиент получает возможность самостоятельно принимать некоторые решения, не обращаясь к серверу. Если клиент должен принимать только простые решения, мы можем написать код, ориентированный на конкретную задачу, но в этом случае ситуация будет похожа на классическое приложение, т.е. основные операции будут по-прежнему выполняться сервером с неизбежным прерыванием работы пользователя. Для того чтобы клиент смог принимать более серьезные решения, ему необходима информация о предметной области, т.е. ее модель.

Модель на сервере нужна в любом случае, так как именно там находятся важные ресурсы, например, база данных, средства доступа к существующим программам и т.д. Таким образом, модель предметной области на стороне клиента обязана взаимодействовать с моделью на стороне сервера. Что же из этого следует? На этот вопрос мы попытаемся ответить в главе 5, где продолжим разговор о взаимодействии клиента и сервера. Там же мы затронем вопросы разделения модели предметной области между различными уровнями.

На данный момент мы рассмотрели модель, представление и контроллер без учета их взаимодействия друг с другом. Закончим эту главу обсуждением вопросов совместной работы модели и представления.

4.5. Генерация представления на основе модели

формируя код, выполняемый на стороне клиента, в соответствии с архитектурой MVC, мы получили три отдельные подсистемы. Разделение функций позволяет создавать более понятные программы, но оно же приводит к увеличению объема кода. Противники применения образов разработки аргументируют свою позицию тем, что такой подход способен превратить решение простейшей задачи в сложнейшую процедуру (разработчикам, применяющим Enterprise JavaBeans, такая ситуация хорошо известна).

Создание приложений, насчитывающих несколько уровней, часто приводит к повторению информации на разных уровнях. Мы хорошо понимаем важность принципа DRY и знаем, что наилучший способ избежать дублирования — определять информацию на одном уровне и на ее основании генерировать остальные уровни. В данном разделе мы постараемся применить этот принцип на практике и опишем подход, позволяющий упростить реализацию архитектуры MVC. Обратим внимание на представление.

До сих пор мы рассматривали представление как код, созданный вручную и позволяющий отображать модель. Благодаря такому подходу мы могли достаточно свободно определять, что должен видеть пользователь. Однако в ряде случаев подобная свобода оказывается излишней, а программирование пользовательского интерфейса вручную всегда было скучным и утомительным занятием. Альтернативный подход предполагает автоматическую генерацию пользовательского интерфейса или хотя бы его компонентов на основе модели. Такой принцип уже использовался в среде Smalltalk и в системе Naked Objects Java/.NET. Язык JavaScript также вполне подходит для решения подобных задач. Рассмотрим, что можно сделать средствами отражения JavaScript, и создадим универсальный компонент "Object Browser", который может быть использован в качестве представления для любого объекта JavaScript.

4.5.1. Отражение объектов JavaScript

В большинстве случаев при написании кода для работы с объектом мы имели достаточно полное представление о том, что это за объект и что можно сделать с его помощью. Однако в некоторых случаях приходится создавать

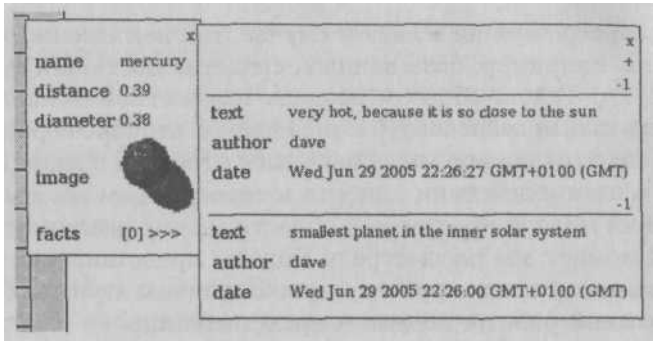


Рис. 4.7. В данном примере ObjectViewer используется для отображения иерархической системы планет, для каждой из которых поддерживается ряд свойств. Кроме того, в массиве хранится дополнительная информация

код "вслепую", не имея предварительных сведений об объекте. Именно так приходится поступать при генерации интерфейса для объектов, составляющих модель. В идеале было бы неплохо реализовать универсальное решение, которое подходило бы для любой предметной области: финансовой деятельности, электронной коммерции, визуализации результатов научных исследований и т.д. В данном разделе вы ознакомитесь с JavaScript-библиотекой ObjectViewer, которую вы, может быть, захотите использовать при разработке своих приложений. Для того чтобы помочь вам составить представление об этой библиотеке, на рис. 4.7 показано несколько уровней сложного графа, отображаемых с помощью ObjectViewer.

Объект, выбранный для просмотра, представляет планету Меркурий. Этот объект достаточно сложен. Помимо обычных свойств, значениями которых являются числа или строки, он содержит URL изображения и массив с дополнительными данными. ObjectViewer обрабатывает всю информацию, не имея предварительных сведений о типе объекта.

Процесс исследования объекта, выяснения его свойств и возможностей называется *отражением* (reflection). Читатели, имеющие опыт работы с Java или .NET, уже знакомы с этим термином. Возможности отражения, реализованные в языке JavaScript, мы рассмотрим подробно в приложении Б. Здесь же только скажем, что объект JavaScript можно исследовать так, как будто он представляет собой ассоциативный массив. Приведенный ниже фрагмент кода выводит информацию о свойствах объекта.

```
var description»"";
for (var i in MyObj){
  var property=MyObj[i];
  description+=i+" = "+property+"(BBSS)n";
} alert(description);
```

Представление данных посредством окна, предназначенного для вывода сообщений, — не самое удачное решение, так как это окно может не сочетаться с остальными компонентами интерфейса. Мы выбрали такой подход лишь для того, чтобы упростить пример. В листинге 4.11 представлен код ObjectViewer.

Листинг 4.11. Библиотека ObjectViewer

```
objviewer.ObjectViewer=function(obj,div,isInline,addNew){
  styling.removeAllChildren(div);
  this.object=obj;
  this.mainDiv=div;
  this.mainDiv.viewer=this;
  this.islnline=islnline;
  this.addNew=addNew;
  var table=document.createElement("table");
  this.tbod=document.createElement("tbody");
  table.appendChild(this.tbod);
  this.fields=new ArrayO;
  this.children=new ArrayO;
  for (var i in this.object){
    this.fields[i]=new objviewer.PropertyViewer(
      this, i
    );
  }
objviewer.PropertyViewer=function(objectViewer,name){
  this.objectViewer=objectViewer;
  this.name=name;
  this.value=objectViewer.object[this.name];
  this.rowTr=document.createElement("tr");
this.rowTr.className='objViewRow';
  this.valTd=document.createElement("td");
this.valTd.className='objViewValue';
  this.valTd.viewer=this;
  this.rowTr.appendChild(this.valTd);
  var valDiv=this.renderSimple();
  this.valTd.appendChild(valDiv);
  viewer.tbod.appendChild(this.rowTr);
}
objviewer.PropertyViewer.prototype.renderSimple=function()    {
    var valDiv=document.createElement("div");
    var valTxt=document.createTextNode(this.value);
    valDiv.appendChild(valTxt);
    if (this.spec.editable){
      valDiv.className+=" editable";
      valDiv.viewer=this;
      valDiv.onclick=objviewer.PropertyViewer.editSimpleProperty;
    }
    return valDiv;
}
```

В состав библиотеки входят два объекта: Obj ectViewer, предназначенный Для просмотра объекта и формирования HTML-таблицы, и PropertyViewer, который отображает имя и значение отдельного свойства и представляет их в виде строки таблицы.

В целом задача решена, но существует ряд проблем. Во-первых, ОЬ-j ectViewer исследует каждое свойство. Если вы добавите к прототипу вспомогательные функции, Obj ectViewer обнаружит их. Если вы обработаете таким способом узел DOM, то увидите все встроенные свойства и сможете оценить, насколько сложен элемент DOM. В большинстве случаев нет необхо-

димости представлять пользователю все свойства объекта. Выбрать свойства для отображения можно, присваивая объекту перед воспроизведением специальное свойство, представляющее собой массив. Этот подход иллюстрируется кодом, приведенным в листинге 4.12.

Листинг 4.12. Применение свойства objviewspec

```
objviewer.ObjectViewer=function(obj,div,isInline,addNew){
  styling.removeAllChildren(div);
  this.object=obj;
  this.spec=objviewer.getSpec(obj);
  this.mainDiv=div;
  this.mainDiv.viewer=this;
  this.islnline=islnline;
  this.addNew=addNew;
  var table=document.createElement("table");
  this.tbod=document.createElement("tbody");
  table.appendChiId(this.tbod);
  this.fields=newArray();
  this.children=new Array();
  for (var i=0,-i<this.spec.length;i++){
    this.fields[i]=new objviewer.PropertyViewer(
      this,this.specfi]
    );
  }
objviewer.getSpec=function (obj){
  return (obj.objViewSpec) ?
    obj.objViewSpec :
    objviewer.autoSpec(obj);
} objviewer.autoSpec=function(obj){
  var members=new Array();
  for (var propName in obj){
    var spec={name:propName};
    members.append(spec);
  }
  return members;
}
objviewer.PropertyViewer=function(objectViewer,memberSpec) {
  this.objectViewer=objectViewer;
  this.spec=memberSpec;
  this.name=this.spec.name;

}
```

Мы определили свойство objViewSpec, которое используется конструктором Objectviewer. Если данное свойство отсутствует, оно формируется в функции autoSpecO путем анализа объекта. Свойство objViewSpec представляет собой массив, каждый элемент которого является таблицей свойств. В данном случае мы генерируем только свойство name. Конструктору PropertyViewer передается описание свойства, из которого можно извлечь инструкции о порядке воспроизведения.

Предоставляя спецификацию для объекта, обрабатываемого Objectviewer, мы можем ограничить набор отображаемых свойств.

Вторая проблема, связанная с использованием ObjectViewer, состоит в том, что этот объект плохо обрабатывает сложные свойства. При добавлении к строке объектов, массивов и функций вызывается метод toString(). В случае объекта она возвращает совершенно неинформативное описание типа [Object object]. Для объекта Function возвращается исходный код функции. По этой причине необходимо различать разные типы свойств. Сделать это можно с помощью оператора instanceof. Учитывая выше названные особенности, рассмотрим, как можно улучшить средства отображения объекта.

4.5.2. Обработка массивов и объектов

Для того чтобы обеспечить поддержку массивов и объектов, надо дать возможность пользователю прослеживать их внутреннюю структуру путем применения к каждому свойству объекта ObjectViewer. Сделать это можно разными способами. В данном случае мы представим дочерние объекты в виде дополнительных окон, расположенных по принципу иерархического меню.

Чтобы получить требуемый результат, надо в первую очередь добавить к спецификации объекта свойство type и определить поддерживаемые типы.

```
objviewer.TYPE_SIMPLE="simple";:
objviewer.TYPE_ARRAY="array"; objviewer.TYPE_FUNCTION="function";
obj viewer.TYPE_IMAGE_URL="image ur1" ;
objviewer.TYPE_OBJECT="object";
```

Модифицируем функцию, которая генерирует спецификации объектов, не имеющих собственных описаний, как показано в листинге 4.13.

Листинг 4.13. Модифицированная функция autoSpec()

```
objviewer.autoSpec=function(obj){
  var members=new Array();
  for (var propName in obj){
    var propValue=obj[name];
    var propType=objviewer.autoType(value);
    var spec={name:propName,type:propType};
    members.append(spec);
  }

  if (obj && obj.length>0){
    for(var i=0;i<obj.length;i++){
      var propName="array ["+!+"]";
      var propValue=obj[i];
      var propType=objviewer.ObjectViewer.autoType(value);
      var spec={name:propName,type:propType};
      members.append(spec);
    )
  1
  return members;
}

objviewer.autoType=function(value){
  var type=objviewer.TYPE_SIMPLE;
  if ((value instanceof Array)){
```

```
    type=objviewer.TYPE_ARRAY;
  Jelse if (value instanceof Function){
    type=objviewer.TYPE_FUNCTION;
  }else if (value instanceof Object){
    type=objviewer.TYPE_OBJECT;
  }
  return type;
}
```

Заметьте, что мы также реализовали поддержку массивов с числовыми индексами, элементы которых нельзя обработать в цикле for... in.

Далее нам необходимо изменить объект PropertyViewer так, чтобы он учитывал различия типов и соответственно воспроизводил данные. Модифицированный вариант конструктора PropertyViewer показан в листинге 4.14.

Листинг 4.14. Модифицированный конструктор PropertyViewer

```
objviewer.PropertyViewer=function
  (objectViewer,memberSpec,appendAtTop){
   this.objectViewer=objectViewer;
   this.spec=memberSpec;     *
   this.name=this.spec.name;
   this.type=this.spec.type;
   this.value=objectViewer.object[this.name];
   this.rowTr=document.createElement("tr") ;

this.rowTr.className='objViewRow' ;
  var  isComplexType=(this.type==objviewer.TYPE_ARRAY
                    I| this.type==objviewer.TYPE_OBJECT);
  if ( !(isComplexType &&
this.objectViewer.isInline )

    this.nameTd=this.renderSideHeader() ;
    this.rowTr.appendChild(this.nameTd);

  this.valTd=document.createElement("td") ;

this.valTd.className='objViewValue• ;
  this.valTd.viewer=this;
  this.rowTr.appendChild(this.valTd) ;
  if (isComplexType){
    if (this.viewer.islnline){
      this.valTd.colSpan=2 ;
      var nameDiv=this.renderTopHeader();
      this.valTd.appendChild(nameDiv);
      var valDiv=this.renderInlineObject();
      this.valTd.appendChild(valDiv);
    }else{
      var valDiv=this.renderPopoutObject();
      this.valTd.appendChild(valDiv);


  else if
(this.type==objviewer.TYPE_IMAGE_URL){
    var vallmg=this.renderlmage();
```

```
      this.valTd.appendChild(vallmg);
    }
    else if
  (this.type==objviewer.TYPE_SIMPLE){
      var valTxt=this.renderSimple();
      this.valTd.appendChild(valTxt);
    }
    if (appendAtTop){
  styling.insertAtTop(viewer.tbod,this.rowTr);
    }else{
      viewer.tbod.appendChild(this.rowTr);
    }
  }
```

Для того чтобы обработать разные типы свойств, мы определили различные методы воспроизведения, реализация которых здесь обсуждаться не будет. Полностью исходный код Objectviewer молено скопировать с Web-узла, посвященного данной книге.

Итак, мы практически полностью решили задачу автоматизации представления модели. Для того чтобы объекты модели были видимыми, нам осталось лишь связать свойства objViewSpec с прототипами. Объект Planet, показанный на рис. 4.7, содержит в составе конструктора следующее'выражение:

```
this.objViewSpec=[
   {name:"name",     type:"simple"},
   {name:"distance", type:"simple",
editable:true},
   {name:"diameter", type:"simple",
editable:true},
   {name:"image",     type:"image url"},
   {name:"facts",     type:"array",   addNew:this.newFact,
inline:true } ];
```

Для данной спецификации использован принцип записи JSON (JavaScript object notation). Квадратные скобки обозначают числовой массив, а фигурные скобки — ассоциативный массив объектов (на самом деле оба типа массивов реализуются одинаково). Более подробно мы обсудим JSON в приложении Б.

В данном фрагменте кода остались некоторые выражения, которые мы еще не рассмотрели. Например, что означают addNew, inline и editable? Они оповещают представление о том, что данные разделы модели могут не только просматриваться, но и модифицироваться пользователем, а это область ответственности контроллера. Использованию контроллера посвящен следующий раздел.

4.5.3. Включение контроллера

Отображение модели — чрезвычайно важная функция, но для большинства приложений требуется также возможность модифицировать данные — редактировать документ, добавлять товары в "корзинку" покупателя и т.д. В роли посредника между интерактивными средствами, поддерживающими взаимодействие с пользователем и моделью, выступает контроллер. Сейчас мы добавим функции контроллера к объекту ObjectViewer.

Первое, что нам надо сделать, — это обеспечить возможность редактирования простых текстовых значений по щелчку на соответствующем элементе. Конечно, это должно происходить только в тех случаях, когда установки, произведенные посредством соответствующих флагов, разрешают его редактирование. В листинге 4.15 показан код, используемый для воспроизведения простого текстового свойства.

ЛИСТИНГ 4.15. Функция renderSimple ()

```
objviewer.PropertyViewer.prototype.renderSimple=function(){
    var valDiv=document.createElement("div");
    // Отображение значения, предназначенного для чтения
    var valTxt=document
       .createTextNode(this.value);
    valDiv.appendChild(valTxt);
    // О Если допустимо редактирование,
    // добавляются интерактивные средства
    if (this.spec.editable){
      valDiv.className+=" editable";
      valDiv.viewer=this;

valDiv.onclick=objviewer.PropertyViewer.editSimpleProperty;
    }
    return valDiv;
}
// © Начало редактирования
objviewer.PropertyViewer.editSimpleProperty=function(e){

  var viewer=this.viewer;
  if (viewer){
    viewer.edit();
  }
}
objviewer.PropertyViewer.prototype.edit=function(){
  if (this.type=objviewer.TYPE_SIMPLE){
    var editor=document.createElement("input");
    editor.value=this.value;
    document.body.appendChild(editor);
    var td=this.valTd;
    xLeft(editor,xLeft(td));
    xTop(editor,xTop(td));
    xWidth(editor,xWidth(td));
    xHeight(editor,xHeight(td));
    // © Замена метки на поле редактирования
    td.replaceChild(editor,td.firstChild);
    // О Связывание обработчика обратного вызова
```

```
    editor.onblur=objviewer.
      PropertyViewer.editBlur;
    editor.viewer=this;
    editor.focus();
  }
}
// © Окончание редактирования
objviewer.PropertyViewer
 .editBlur=function(e){
  var viewer=this.viewer;
  if (viewer){
    viewer.coiranitEdit(this.value);
  }
}
objviewer.PropertyViewer.prototype.commitEdit=function(value){
  if (this.type==objviewer.TYPE_SIMPLE){
    this.value=value;
    var valDiv=this.renderSimple();
    var td=this.valTd;
    td.replaceChild(valDiv,td.firstChild);
    // О Оповещение обработчиков
    this.objectViewer
      .notifyChange(this);
  }
}
```

Процесс редактирования свойства включает несколько этапов. Прежде всего нам надо связать обработчик onclick с элементом DOM, отображающим значение О. Понятно, что необходимость в этом возникает лишь тогда, когда свойство является редактируемым. Мы также связываем имя класса CSS с полями, допускающими редактирование, в результате эти поля изменяют цвет при попадании на них курсора мыши. Это сделано для того, чтобы пользователь имел информацию о том, что данное поле допускает редактирование.

Функция editSimpleProperty() © представляет собой простой обработчик событий. Он извлекает из узла DOM, на котором пользователь щелкнул мышью, ссылку на объект PropertyViewer и вызывает метод edit(). Способ связывания представления с контроллером знаком вам по разделу 4.3.1. Мы проверяем корректность типа свойства, а затем заменяем метку, допускающую только чтение, полем HTML-формы соответствующего размера, обеспечивающим редактирование. В этом поле отображается значение свойства ©. Мы также связываем с текстовой областью обработчик onblur О, который заменяет редактируемую область меткой, допускающей только чтение ©, и обновляет модель.

Таким образом, мы можем обрабатывать модель, но в общем случае при обновлении модели бывает необходимо выполнять другие действия. В этом случае включается в работу метод notifyChangeO объекта Ob j ectViewer ©, вызываемый из функции commitEdit(). Соответствующий код показан в листинге 4.16.

Листинг 4.16. Функция ObjectViewer.notifyChange()

```
objviewer.ObjectViewer.prototype
  .notifyChange=function(propViewer){
   if (this.onchangeRouter){
     this.onchangeRouter.notify(propViewer);
   }
   if (this.parentObjViewer){
this.parentObjViewer.notifyChange(propViewer);
   }
} objviewer.ObjectViewer.prototype
  .addChangeListener=function(lsnr){
   if (!this.onchangeRouter){
     this.onchangeRouter=new
jsEvent.EventRouter(this, "onchange");
   }
   this.onchangeRouter.addListener(lsnr);
}objviewer.ObjectViewer.prototype
  .removeChangeListener=function(lsnr){
   if (this.onchangeRouter){
     this.onchangeRouter.removeListener(lsnr);
   }
}
```

Проблема, с которой мы столкнулись, а именно оповещение процессов об изменениях в модели, идеально решается посредством образа разработки Observer и объекта EventRouter, о котором шла речь в разделе 4.3.3. Мы могли бы связать EventRouter с событием onblur редактируемых полей, однако сложная модель обычно предполагает большое их количество, а наш код не имеет доступа к деталям реализации ObjectViewer.

Вместо этого мы определим в ObjectViewer событие специального типа, onchange, и свяжем с ним EventRouter. Поскольку наши объекты ObjectViewer включены в состав древовидной структуры (это сделано для просмотра свойств, представляющих собой объекты и массивы), событие onchange должно передаваться родительским объектам. В общем случае мы можем присоединить обработчик к корневому объекту ObjectViewer, который мы создаем в приложении, и получать изменения свойств модели на более нижних уровнях.

В качестве простого примера можно привести обработчик событий, который отображает сообщение в строке состояния браузера. Объект верхнего уровня в модели планет — это солнечная система, поэтому мы можем написать следующий код:

```
var topview=new objviewer.ObjectViewer
   (planets.solarSystem,mainDiv);
topview.addChangeListener(testListener);
```

где testListener — функция поддержки события, которая выглядит следующим образом:

```
function testListener(propviewer){
  window.status=propviewer.name+" ["+propviewer.type+"] =
    "+propviewer.value;
}
```

Конечно, на практике при изменении модели выполняются более сложные действия, например обращение к серверу. В следующей главе мы рассмотрим способы взаимодействия с сервером и передачи ему объекта Objectviewer для дальнейшего использования.

4Л Резюме

"Модель-представление-контроллер" — это архитектурный шаблон, который в классических Web-приложениях обычно применяется к программам, располагаемым на стороне сервера. Мы продемонстрировали использование этой архитектуры для серверной части Ajax-приложения, генерирующей данные для клиента. Мы также применили этот подход при разработке клиентских программ, причем в процессе работы было найдено несколько интересных решений.

Рассматривая подсистему представления, мы продемонстрировали эффективный способ отделения ее от логики. Полученные результаты позволяют дизайнерам и программистам работать практически независимо друг от друга. Такая организация кода позволяет привести его в соответствие со структурой рабочей группы и квалификацией участников разработки, что, несомненно, положительно скажется на производительности труда.

Говоря о контроллере, мы обсудили различные модели поддержки событий и приняли решение придерживаться старой модели. Несмотря на то что она позволяет определять лишь одну функцию обратного вызова для каждого события, мы научились применять образ разработки Observer для создания гибких обработчиков, допускающих изменение конфигурации. Это удалось сделать на базе стандартной модели событий JavaScript.

Что касается модели, то мы сделали первые шаги по пути создания распределенного многопользовательского приложения. Разговор на эту тему будет продолжен в главе 5.

Для создания модели, представления и контроллера приходится затрачивать много усилий. Обсуждая объект Objectviewer, мы выяснили, что существуют способы, позволяющие автоматизировать взаимодействие элементов архитектуры MVC. Мы создали простую систему, способную представлять модель и позволяющую пользователю взаимодействовать с ней.

Разговор об образах разработки мы продолжим в следующей главе. В ней мы перейдем к рассмотрению взаимодействия клиента и сервера.

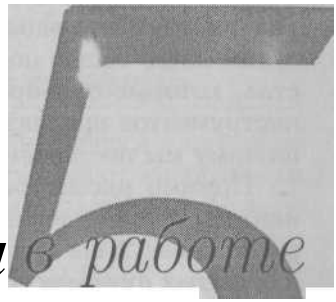
4.7. Ресурсы

Библиотека Behaviour, использованная в данной главе, находится по адресу http://ripcord.co.nz/behaviour/. Библиотеку x Майка Фостера можно скопировать, обратившись к серверу http://www.cross-browser.com.

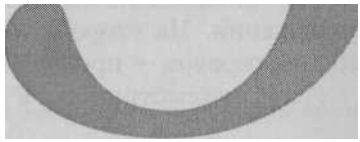
Попытки реализовать автоматическую генерацию представления на основе модели были предприняты в рамках проекта Naked Objects (http://

www.nakedobjects.org/). Книга Ричарда Посона (Richard Pawson) и Роберта Метьюса (Robert Matthews) *Naked Objects* (John Wiley & Sons, 2002) немного устарела, но в ней можно найти полезные сведения, касающиеся разработки компонентов архитектуры "модель-представление—контроллер".

Изображения планет, использованные для демонстрации возможностей Object Viewer, были предоставлены Jim's Cool Icons (http://snaught.com/jimsCoolicons/) и, как сказано на Web-узле, смоделированы с помощью POV-Ray, а в качестве текстуры использованы реальные изображения, полученные от NASA.



Роль сервера в работе Ajax-приложения



В этой главе...

- Использование существующих архитектур в Ajax-приложении
- Обмен содержимым, кодом сценариев и данными с сервером
- Передача обновленных данных серверу
- Объединение нескольких запросов в одном HTTP-обращении

В данной главе мы завершаем работу над созданием надежного и масштабируемого приложения, которая была начата в главе 4. Мы переходим от общих принципов к конкретным программам, которые сможем применить на практике. В главе 4 обсуждались способы структуризации клиентского кода, позволяющие достичь поставленной цели, а сейчас речь пойдет о программах, расположенных на стороне сервера и взаимодействующих с клиентом.

Вначале мы рассмотрим функции, выполняемые сервером, затем обсудим варианты архитектуры, реализованные в инструментах, предназначенных для создания программ на стороне сервера. На сегодняшний день в распоряжении разработчиков, особенно тех, кто использует Java, имеется целый ряд инструментальных средств. Мы не будем рассматривать их все, а обсудим лишь общие подходы, использованные при их создании, и преимущества, которые они предоставляют при разработке программ. Большинство инструментов предназначено для генерации классических Web-приложений, поэтому мы постараемся выяснить, насколько они применимы для Ajax.

Помимо рассмотрения общих вопросов, связанных с работой приложения, мы также уделим внимание деталям взаимодействия клиента и сервера. В главе 2 вы получили основные сведения об объекте XMLHttpRequest и скрытых фреймах. В данной главе мы вернемся к этому вопросу и обсудим использование различных образов разработки для передачи данных клиенту и выясним, есть ли альтернатива разбору XML-документов с применением методов DOM. В последнем разделе мы представим систему управления трафиком, связанным с обменом между клиентом и сервером в процессе работы приложения. На стороне клиента будет находиться очередь запросов, а на стороне сервера — процессы для их обработки.

Итак, рассмотрим роль сервера в Ajax-приложении.

5.1. Программы, выполняемые на сервере

В процессе работы Ajax-приложения сервер выполняет две основные функции, которые имеют существенное отличие. Прежде всего, он доставляет приложение браузеру. Мы считаем, что в процессе доставки данные не претерпевают изменения, поэтому реализуем средства, предназначенные для выполнения на стороне клиента, как набор файлов .html, .ess и .js. С передачей их клиенту справится даже самый простой сервер. Такое решение вполне жизнеспособно, но оно не единственное. Существуют альтернативные варианты. Средства, предназначенные для создания серверных программ, мы подробно обсудим в разделе 5.3.

Вторая задача сервера — это взаимодействие с клиентом, т.е. обработка запросов и подготовка ответов. Поскольку HTTP — единственный возможный в данной ситуации транспортный механизм, взаимодействие всегда должно начинаться по инициативе клиента. Сервер может только отвечать. В главе 4 мы говорили о том, что Ajax-приложение должно поддерживать модель предметной области как на стороне клиента (для обеспечения быстрого отклика), так и на стороне сервера (для доступа к ресурсам, например, к базе данных). Синхронизация этих моделей — сложная задача, и клиент без по-

мощи сервера не может справиться с ней. Вопросы работы сервера будут рассмотрены в разделе 5.5; там же предложены решения данной проблемы, основанные на использовании одного из образов разработки, описанных в главе 3.

Прочитав данную главу, вы узнаете, что доставить приложение клиенту и организовать взаимодействие с ним можно различными способами. Существует ли наилучшее решение? Можно ли обеспечить взаимную поддержку? Как различные решения будут работать с уже имеющимся программным обеспечением? Чтобы ответить на эти вопросы, необходимо описать возможности, имеющиеся в нашем распоряжении. Это мы постараемся сделать в данной главе. Рассмотрим варианты участия сервера в работе Web-приложения и особенности, связанные с применением инфраструктуры Ajax.

5.2. Создание программ на стороне сервера

В традиционных Web-приложениях программы, выполняющиеся на стороне сервера, обычно достаточно сложны. Именно они определяют особенности работы пользователя и хранят информацию о текущем состоянии. Приложение, как правило, ориентировано на определенный язык взаимодействия и учитывает ряд соглашений, которые задают порядок его работы. Возможные языки могут определяться архитектурой, операционной системой или аппаратными средствами. Выбор среды программирования — важный шаг в создании программ, поэтому рассмотрим возможности, имеющиеся в распоряжении разработчика.

5.2.1. Популярные языки программирования

Для создания программного обеспечения, предназначенного для выполнения на стороне сервера, могут использоваться различные языки. Несмотря на то что вся история всемирной сети очень коротка, отношение разработчиков к тем или иным языкам программирования не раз изменялись. В настоящее время чаще всего используются PHP, Java и классическая технология ASP; также становятся все более популярными ASP.NET и Ruby. Эти названия, конечно же, знакомы большинству читателей. Ajax имеет дело преимущественно с клиентским программным обеспечением, которое может взаимодействовать с серверными программами, написанными на любом из этих языков. Более того, при использовании Ajax не столь важно, какой именно язык применялся при создании серверных программ. В результате упрощается перенос Ajax-приложений с одной платформы на другую.

Часто базовые средства, используемые при написании серверной части Ajax-приложения, становятся более важными, чем язык реализации. Архитектура и базовые средства определяют структуру приложения и функции его основных компонентов. Для создания классических Web-приложений были разработаны различные архитектуры, однако поскольку жизненный цикл Ajax-приложений отличается от классического варианта программ, не все архитектуры оказываются пригодными. Вопросы разработки серверных программ и используемые при этом средства описаны далее в этой главе, а сейчас Мы рассмотрим базовые принципы Web-архитектуры.

5.2.2. N-уровневые архитектуры

Одним из основных понятий, используемых при разработке распределенных приложений, является *уровень* (tier). С одной стороны, уровень определяет круг вопросов, за решение которых отвечает приложение, а с другой — описывает изолированную подсистему, работающую на отдельной машине либо оформленную в виде отдельного процесса. Этим понятие уровней отличается от понятия ролей в архитектуре MVC. Модель, представление и контроллер не являются уровнями, поскольку они реализованы в рамках одного процесса.

Первые распределенные системы насчитывали два уровня — клиентский и серверный. Уровень клиента обычно располагался на настольной системе и взаимодействовал с серверным уровнем по сети. Для организации взаимодействия использовались сокеты, поддержка которых была реализована в виде библиотеки. В большинстве случаев серверный уровень является сервером базы данных.

Первые системы Web включали браузер, взаимодействующий с Web-сервером. Сервер обычно отвечал за передачу браузеру файлов, находящихся в файловой системе серверной машины.

По мере развития и усложнения Web-приложений потребовался доступ к базам данных. Двухуровневая модель "клиент/сервер" уступила место трехуровневой модели, в которой сам Web-сервер занял место промежуточного уровня между клиентом и СУБД. Дальнейшее развитие привело к выделению на промежуточном уровне средств представления и бизнес-логики. Они реализовывались либо в виде отдельных процессов, либо как модули в составе одной программы.

Современное Web-приложение обычно имеет два основных уровня. Уровень бизнес-логики моделирует предметную область и непосредственно взаимодействует с базой данных. Уровень представления получает данные от средств бизнес-логики и представляет их пользователю. Браузер занимает в этой системе место низкоуровневого клиента.

С появлением Ajax стал развиваться клиентский уровень. Если ранее средства представления полностью отвечали за поддержку работы пользователя, то сейчас их функции разделились между сервером и клиентом (рис. 5.1). Роль уровня представления на стороне сервера стала менее важной, поскольку некоторые вопросы обеспечения последовательности действий при работе пользователя полностью решаются на уровне клиента. Соответствующие программные средства создаются на языке JavaScript и работают в среде браузера.

Этот новый уровень предоставляет новые возможности, которые мы уже обсуждали в предыдущих главах. Однако вместе с тем он создает предпосылки для усложнения программ, в результате чего при разработке могут возникать дополнительные проблемы. Нам надо знать, как справиться с ними.

Рис. 5.1. В Ajax-приложении некоторые функции уровня представления переместились со стороны сервера на сторону клиента. В результате возник новый уровень — клиентский уровень представления

5.2.3. Управление моделью предметной области на стороне клиента и на стороне сервера

В приложении Ajax нам по-прежнему необходимо моделировать предметную область на сервере, "в непосредственной близости" от базы данных и других жизненно важных ресурсов. Однако для того, чтобы клиентская программа могла обеспечить быстрый отклик на действия пользователя, она должна обладать определенным "интеллектом", а это требует наличия на стороне клиента хотя бы частично реализованной модели. В результате возникает проблема синхронизации этих двух моделей.

Добавление дополнительного уровня всегда связано с увеличением сложности и возрастанием накладных расходов при взаимодействии. К счастью, данная проблема не нова. Аналогичные трудности возникали при разработке Web-приложений средствами J2EE, где были строго разделены уровень бизнес-логики и уровень представления. Модель предметной области поддерживается средствами бизнес-логики. К ней обращается уровень представления, на котором затем генерируется содержимое Web-страниц, передаваемых браузеру. В J2EE проблема была решена путем применения *объектов передачи* (transfer objects), которые представляют собой обычные объекты, передаваемые между уровнями. Они представляют уровню представления ограниченный доступ к модели предметной области.

Однако Ajax ставит новые задачи. При использовании технологии J2EE оба уровня реализовывались на одном и том же языке. Более того, в распоряжении разработчика был механизм удаленных процедур. При создании приложения Ajax такие условия, как правило, обеспечить невозможно. В принципе можно применить для'написания программ на стороне сервера язык JavaScript, однако такой подход нельзя назвать общепринятым, да и к тому же остается задача взаимодействия двух JavaScript-программ.

При организации взаимодействия уровней приходится решать две основные задачи: чтение данных с сервера и запись их на сервер. Подробно они рассмотрены в разделах 5.3-5.5. Перед тем как завершить разговор об архитектуре, мы должны рассмотреть основные принципы создания серверных программ, используемых в настоящее время. В частности, нас интересуют вопросы формирования модели предметной области, организации доступа к ней с уровня представления и ограничения, которые могут возникнуть при использовании Ajax.

Согласно данным исследований (ссылка на соответствующий обзор приведена в конце данной главы) в настоящее время только для Java существует более 60 вариантов базовых средств разработки (по правде говоря, такое обилие предложений скорее мешает, чем помогает в работе). Большинство из них различается лишь в деталях, и, независимо от языка реализации серверных программ, мы можем выделить для уровня представления несколько основных архитектурных принципов. Рассмотрим их.

5.3. Принципы создания программ на сервере

Базовые средства, применяемые для разработки программ на стороне сервера, важны для Ajax-приложений. Если код клиента генерируется на базе модели, поддерживаемой на сервере, их значение еще более возрастает. Если мы вручную создадим код клиента и оформим его в виде статических HTML-документов, содержащих JavaScript-код, то базовые средства не будут участвовать в доставке приложения, но данные, требуемые приложению в процессе работы, по-прежнему должны генерироваться динамически. Как было сказано ранее, программы на стороне сервера обычно разделяются на два уровня: модель предметной области и представление. В случае Ajax-приложения средства представления выполняют роль посредника между моделью и клиентской частью. Для того чтобы приложение работало эффективно, надо правильно организовать взаимодействие клиента и сервера, а для этого необходимо знать архитектуру серверных программ и, возможно, инструменты, применявшиеся при их создании.

Некоторые всерьез считают сервер в составе Web-приложения игрушкой в руках разработчика. Проблема поддержки работы пользователя посредством набора Web-страниц и обращения к другим системам на сервере, например СУБД, еще не решена по-настоящему. Web буквально наводнена неудачными системами и утилитами, а новые проекты появляются каждый месяц, а то и каждую неделю.

К счастью, среди этих разнообразных проектов можно выделить группы со сходными характеристиками. Анализируя основные подходы, можно описать четыре основных способа решения задач, поставленных перед сервером. Рассмотрим их и выясним, подходят ли они для нашей Ajax-модели.

5.3.1. Серверные программы, не соответствующие основным принципам разработки

Самая простая архитектура — это отсутствие таковой. Создание приложения без учета основных принципов разработки не обязательно означает отсутствие порядка. Возможно, в приложении будут удачно определены основные этапы работы пользователя и организовано обращение к ресурсам. Многие Web-узлы созданы именно таким образом. Каждая страница генерирует свое собственное представление и по-своему обращается к базе данных и другим системам. Во многих случаях при создании подобных приложений применяются библиотеки и вспомогательные функции. Приложение, созданное подобным образом, условно показано на рис. 5.2.

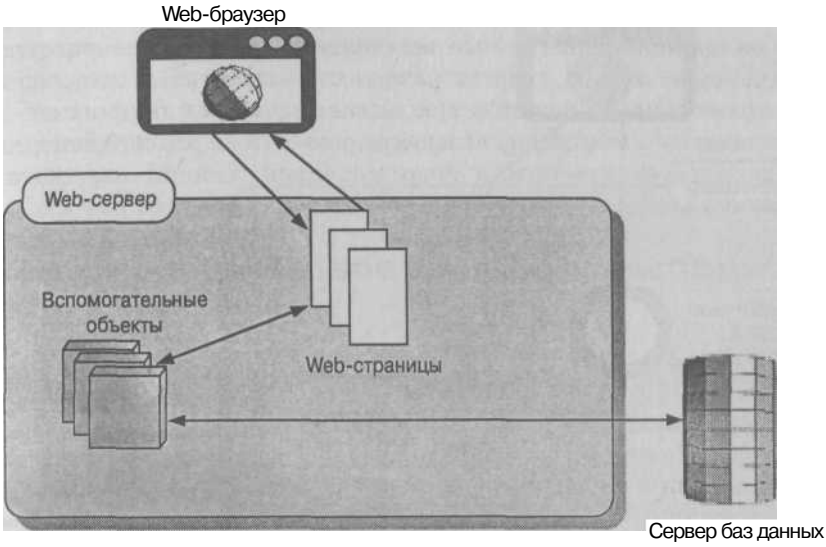


Рис. 5.2. Web-программа, созданная без учета основных принципов разработки. Каждая страница, сервлет или CGI-сценарий поддерживает собственную логику и особенности представления данных. Вспомогательные методы и объекты могут инкапсулировать низкоуровневые функции общего назначения, например обращение к базе данных

Применить данный подход для Ajax-приложения достаточно просто, при условии, что код клиента создается вручную. Генерация клиентской программы сервером — сложная задача, и ее рассмотрение выходит за рамки данной книги. Для доставки клиента надо определить основную страницу, содержащую JavaScript-код, таблицы стилей и другие ресурсы. Для доставки данных нам надо лишь заменить HTML-документы, генерируемые сервером, XML-данными или информацию в другом формате.

Главный недостаток подобного подхода в классическом Web-приложении состоит в том, что связи между документами распределены в коде самих документов. По этой причине средства, выступающие в роли контроллера, не локализованы. Если разработчику надо изменить порядок представления Web-страниц пользователю, ему необходимо изменять гипертекстовые ссылки в различных местах. Ситуацию можно несколько улучшить, помещая фрагменты данных, содержащих большое количество ссылок, например средства навигации, во включаемые файлы или генерируя их посредством вспомогательных функций, однако стоимость сопровождения все равно будет резко возрастать по мере усложнения приложения.

В Ajax-приложении проблема, возможно, будет не столь острой, поскольку гипертекстовые ссылки в этом случае встречаются несколько реже, однако включение инструкций и перенаправление их между страницами все равно усложнит разработку. В простых XML-документах включение и перенаправление не требуется, но в приложениях большого объема может возникать необходимость в пересылке документов со сложной структурой, сфор-

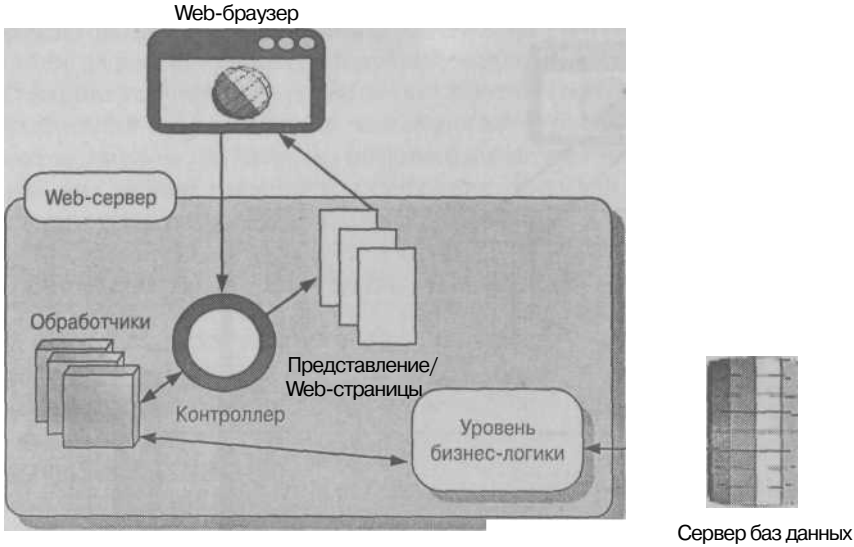


Рис. 5.3. Архитектура Model2. Единственная страница контроллера или единственный сервлет получает все запросы и распределяет их в соответствии со схемой работы пользователя. Запрос может быть передан для обработки вспомогательным классам или функциям и перед пересылкой браузеру передается компоненту, выполняющему роль представления (например, JSP- или PHP-документу)

мированных в результате совместной работы нескольких процессов. В ранних разработках для преодоления данной проблемы использовалась архитектура MVC. Приложения, созданные по такому принципу, часто встречаются в настоящее время.

5.3.2. Использование архитектуры Model!

Образ разработки Model2 является разновидностью MVC. Отличается он тем, что контроллер имеет единственную точку входа и единственное определение последовательности действий пользователя. В применении к Web-приложению это означает, что одна страница контроллера или один сервлет отвечает за маршрутизацию большинства запросов, передавая их различным службам и пересылая полученные данные представлению. Среди базовых наборов средств, поддерживающих Model2, наиболее известным является Apache Struts. Данной архитектуре также соответствует ряд других инструментов для Java и PHP. На рис. 5.3 условно показано Web-приложение, соответствующее архитектуре Model2

Как же применить данный принцип разработки к серверному приложению, взаимодействующему с клиентом Ajax? Model2 практически никак не определяет доставку клиентского приложения, потому что обычно она производится в начале работы и выполняется практически одинаково для всех пользователей. Централизованный контроллер может участвовать в процессе аутентификации, однако этот факт не влияет на доставку.

Данная архитектура хорошо подходит для доставки данных. Представление, созданное в рамках архитектуры Model2, практически не зависит от базовых средств, и мы можем без труда заменить HTML на XML или другой формат. Часть функций контроллера передается уровню клиента, но другие функции реализуются на стороне сервера посредством отображения.

Архитектура Model2 предоставляет классическому Web-приложению средства для выражения функций контроллера на высоком уровне абстракции, однако представление при этом обычно реализуется вручную. Другие принципы разработки дают возможность использовать высокоуровневые средства и для создания представления.

5.3.3. Использование архитектуры на базе компонентов

При создании HTML-страницы для классического Web-приложения в распоряжении автора имеется ограниченный набор компонентов пользовательского интерфейса — обычно этот набор исчерпывается элементами HTML-формы. Их возможности остаются неизменными уже около десяти лет, и они существенно проигрывают современным инструментам, предназначенным для создания интерфейсов. Если автор захочет реализовать нечто вроде древовидной структуры или таблицы с редактируемыми ячейками, он вынужден будет заняться низкоуровневым программированием. Это не идет ни в какое сравнение с уровнем абстракции, доступным разработчику, создающему программы для настольной системы и имеющему в своем распоряжении такие инструменты как MFC, GTK+, Cocoa, Swing или Qt.

Компоненты для Web

Принцип разработки, ориентированный на использование компонентов, повышает уровень абстракции при программировании пользовательских интерфейсов. В этом случае в распоряжение разработчика серверных программ предоставляются элементы, интерфейс которых не уступает компонентам графического интерфейса для настольных систем. В процессе воспроизведения компонентов, предназначенных для настольных систем, осуществляется рисование в графическом контексте. При этом осуществляются низкоуровневые обращения к средствам генерации графических примитивов, например, геометрических фигур, битовых карт и т.д. Воспроизведение компонентов для Web предполагает автоматическую генерацию потока HTML-данных и JavaScript-программ. В результате в среде браузера реализуются функции, типичные для настольных систем, и программист избавляется от необходимости выполнять операции на низком уровне. На рис. 5.4 условно показан принцип разработки, основанный на использовании компонентов.

Многие архитектуры на базе компонентов описывают взаимодействие с пользователем в терминах, применяемых при работе на настольных системах. Так, например, с компонентом Button может быть связан обработчик события, соответствующего щелчку мыши, поле редактирования может иметь обработчик valueChange и т.д. В большинстве архитектур обработка событий делегируется серверу посредством запроса, который формируется

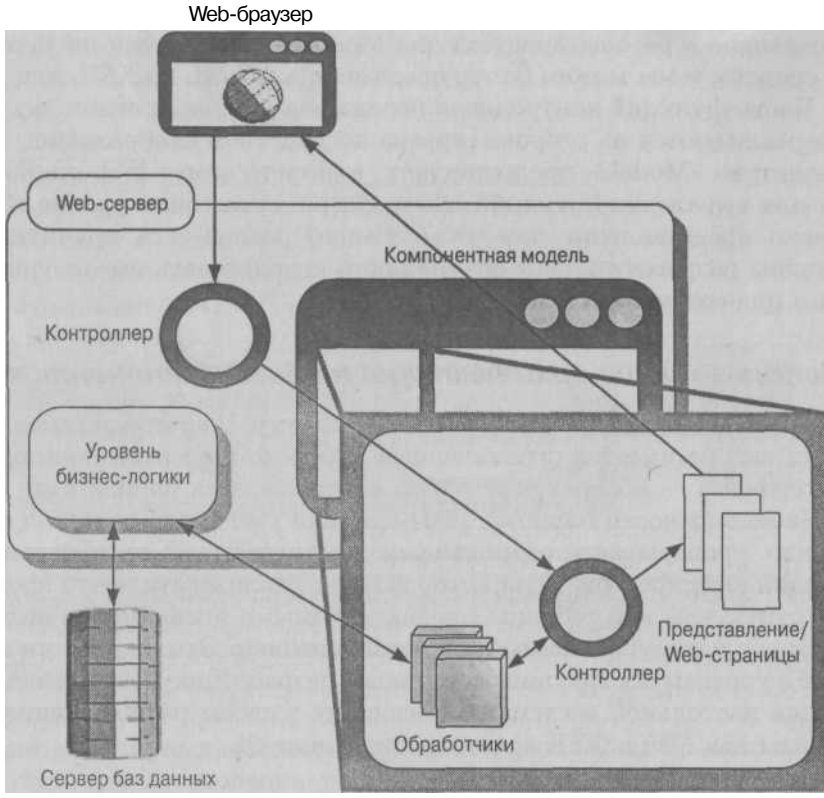


Рис. 5.4. Архитектура, основанная на использовании компонентов. Приложение описывается как набор компонентов, для воспроизведения которых браузеру передается поток HTML-данных и JavaScript-программ Каждый компонент содержит собственные "микрореализации" модели, представления и контроллера. Контроллер более высокого уровня обрабатывает запросы браузера к отдельным компонентам и **модели предметной области**

в ответ на действие пользователя. В "интеллектуальных" архитектурах обработка событий осуществляется незаметно для пользователя, а в других при возникновении каждого события обновляется вся страница. Как правило, в приложениях, созданных в виде набора компонентов, взаимодействие с сервером осуществляется более эффективно по сравнению, например, с приложениями Model2.

Одна из целей, преследуемых архитектурой, о которой идет речь, — это формирование различных типов пользовательского интерфейса на базе единого описания. Некоторые наборы базовых средств, например Windows Forms для .NET или JSF (JavaServer Faces), обеспечивают такую возможность.

Применимость архитектуры на базе компонентов для Ajax-приложений

Как же сочетается архитектура на базе компонентов с Ajax? На первый взгляд, взаимодействие должно быть органичным, поскольку оба подхода предполагают переход от интерфейса на основе документов к использованию компонентов. Может показаться, что данную архитектуру целесообразно применять для Ajax-приложений, в особенности тогда, когда клиентские средства генерируются программой и существуют включаемые средства воспроизведения, поддерживающие Ajax. Такой подход выглядит весьма привлекательным, поскольку в этом случае исчезает необходимость специального изучения разработчиками особенностей языка JavaScript и существует возможность реализовать систему воспроизведения на базе обычного HTML-кода.

Подобное решение хорошо подходит для тех приложений, для которых требуются лишь стандартные типы компонентов. Тем не менее некоторая степень гибкости все же теряется. Успех Google Maps (см. главу 1) в значительной степени обусловлен тем, что для данной системы определен собственный набор компонентов, от прокручиваемой карты до элементов масштабирования и всплывающей подсказки. Реализовать ту лее систему, используя стандартный набор компонентов, типичных для настольной системы, было бы гораздо сложнее, и конечный результат наверняка получился бы гораздо худшим.

Интерфейс многих приложений вполне укладывается в привычный набор компонентов. Именно для них целесообразно применять рассматриваемую здесь архитектуру. Необходимость компромисса между гибкостью и удобством разработки — общая проблема для многих решений, основанных на генерации кода.

Для того чтобы архитектура была пригодной для Ajax-приложения, она должна обеспечивать эффективную передачу данных в процессе работы. Здесь проблемы могут оказаться более серьезными, так как контроллер жестко "привязан" к уровню сервера и определен с помощью выразительных средств, типичных для настольных систем. Для Ajax-приложения, способного обеспечивать требуемый отклик на действия пользователя, необходимо иметь возможность определять собственные обработчики событий, которую сервер не всегда может предоставить. Однако данная архитектура имеет серьезные потенциальные возможности, и по мере возрастания популярности Ajax несомненно будут появляться новые решения. Система CommandQueue, которую мы рассмотрим в разделе 5.5.3, может стать существенным шагом по пути применения JSF и других подобных технологий, однако на сегодняшний день она еще не готова. Имеющиеся же в наличии базовые средства не предоставляют клиентам той свободы, которую хотелось бы видеть.

Будущее покажет, насколько хорошо удастся адаптировать системы на базе компонентов для Ajax. В настоящее время наблюдается рост интереса со стороны корпорации Sun и некоторых поставщиков JSF к инструментам, созданным с учетом Ajax. Поддержка некоторых функций, применимых для Ajax, уже реализована в .NET Forms; они также будут доступны в наборе инструментальных средств Atlas. (Ссылки на эти системы приведены в конЦе главы.)

Может возникнуть вопрос: как должны выглядеть инструментальные средства, созданные специально для Ajax? На сегодняшний день таковых еще нет, но возможно, что они будут разработаны на основе какой-либо из существующих систем.

5.3.4. Архитектуры, ориентированные на использование Web-служб

Последняя архитектура из рассматриваемых в данной главе, — это SOA (service-oriented architecture), т.е. архитектура, основанная на использовании служб. В данном случае служба — это нечто, к чему можно обратиться по сети, и получить в качестве ответа структурированный документ. Основное внимание здесь уделяется не содержимому, а данным, что вполне соответствует принципам Ajax. В настоящее время наиболее часто используются Web-службы, а XML в качестве основного языка также с энтузиазмом воспринимается разработчиками Ajax-приложений.

На заметку Термин Web-служба можно понимать как в широком, так и в узком смысле. Web-служба в узком смысле — это средства, использующие протокол SOAP. Web-служба в широком смысле — это любая система обмена данными, базирующаяся на протоколе HTTP, независимо от того, применяется ли при ее работе протокол SOAP или формат XML. XML-RPC, JSON-RPC и любая система, которую вы разработаете, используя объект XMLHttpRequest, будет Web-службой в широком смысле.

Используя Web-службу как источник данных, клиент Ajax может обеспечить высокую степень автономности, соизмеримую с той, какой обладает почтовый клиент, взаимодействующий с почтовым сервером. Возможности повторного использования элементов приложения превышают те, которые могут предоставить инструментальные средства, предполагающие использование компонентов. Так, клиент определяется лишь единожды и экспортируется для различных интерфейсов. Служба также определяется один раз и может быть использована различными клиентами, работающими независимо друг от друга. Таким образом, сочетание SOA и Ajax может стать основой достаточно мощной системы. В этом случае разделяются средства, предназначенные для генерации Ajax-приложений и для их обслуживания.

Предоставление доступа к объектам на стороне сервера

Многие инструментальные средства позволяют представить обычный объект, созданный на Java, C# или PHP, как Web-службу. При этом поддерживается отображение методов объекта в интерфейс Web-службы. Такую возможность предоставляют, в частности, Microsoft Visual Studio и Apache Axis for Java. Многие инструменты Ajax, например DWR (для Java) и SAJAX (для PHP, .NET, Python и некоторых других языков), обеспечивают данную возможность посредством клиентского кода, написанного на языке JavaScript.

Применение инструментальных средств может существенно помочь при здании приложения. Если же использовать их неумело, результат будет противоположным. Для того чтобы составить представление о правильном использовании инструментов, рассмотрим простой пример, в котором применяется Java DWR. Мы определим объект на стороне сервера для предоставления персональной информации.

```
package com.manning.ajaxinaction;
public class Person{
  private String name=null;
  public Person(){
  }
  public String getName(){
    return name;
  }
  public void setName(String name){
    this.name=name;
  }
}
```

Данный объект должен соответствовать спецификации JavaBeans. Это означает, что он должен предоставлять конструктор, вызываемый без параметров, и обеспечивать доступ к требуемым полям для чтения и записи посредством get- и set-методов. Далее мы сообщаем DWR о том, что данный объект должен быть доступен уровню JavaScript. Для этого мы редактируем файл dwr.xml.

```
<dwr>
  <init>
    <convert id="person" converter="bean"
      match="com.manning.ajaxinaction.Person"/>
  </init>
  <allow>
    <create creator="new" javascript="person">
      <param name="class"
value="com.manning.ajaxinaction.Person">
    </create>
  </allow>
</dwr>
```

В разделе `init` мы определяем средства преобразования нашего класса в тип bean, а в разделе allow определяем средства, представляющие данный объект JavaScript-программам в виде переменной person. Наш объект Person содержит только один общедоступный метод, getName(), таким образом, мы можем включить в код клиента Ajax следующее выражение:

```
var name=person.getName();
```

Затем можно получать требуемое значение с сервера в асинхронном режиме.

В нашем классе Person содержится только один метод. На первый взгляд может показаться, что им и исчерпывается набор средств, которые предоставляются клиенту. На самом деле это не так. Класс Person является подклассом java.lang.Object и наследует от него ряд общедоступных методов, например hashCode() и `tostring()`. К ним также может производиться об-

ращение. Эти "скрытые" средства не являются специфическими для DWR. Метод JSONRPCBridge. registerObject (), например, делает то же самое. Следует заметить, что DWR позволяет ограничить доступ к определенным методам, настраивая конфигурационный XML-файл. Однако по умолчанию доступ предоставляется ко всем методам. Такая проблема типична для решений, основанных на отражении. Мы встретились с ней в главе 4 при создании первых вариантов Objectviewer. Рассмотрим способы ее решения.

Предоставление ограниченного доступа

Если мы ненамеренно дадим возможность пользователям Web вычислять хэш-коды наших объектов, может ли это представлять опасность для нас? Для рассматриваемого примера, по-видимому, нет, так как суперклассом создаваемого нами класса является java.lang.Object, и эта ситуация вряд ли изменится. Для более сложных моделей, предоставляя информацию о суперклассе, мы не гарантируем, что программа в дальнейшем не подвергнется реструктуризации. Вполне может получиться так, что разработчик клиентской программы захочет воспользоваться методами, доступ к которым вы непреднамеренно предоставили. Затем, когда реструктуризированная модель будет доставлена на сервер, программные средства на стороне клиента внезапно откажутся работать. Другими словами, такое решение препятствует корректному разделению функций между клиентом и сервером. Если вы используете инструментальные средства, подобные DWR или JSON-RPC, вам следует тщательно взвешивать решения относительно состава интерфейса Ajax. Возможно, вам даже придется создать нечто вроде объекта Facade (рис. 5.5).

Использование в данной ситуации образа разработки Facade обеспечивает ряд преимуществ. Во-первых, как было сказано ранее, он позволяет без опасения реструктуризировать модель на стороне сервера. Во-вторых, такое решение упрощает общедоступный интерфейс, используемый клиентом. По сравнению с кодом, написанным для внутреннего применения, расходы на создание интерфейсов, предоставляемых другим элементам приложения, оказываются достаточно высокими. Необходимо создать полную документацию, иначе служба поддержки будет перегружена обращениями пользователей, пытающихся использовать созданный вами общедоступный интерфейс.

Еще одно преимущество Facade состоит в том, что данный образ разработки позволяет определять степень детализации предоставляемых услуг. Хорошая модель предметной области обычно содержит большое количество методов, выполняющих несложные действия. Таким образом, удовлетворяется требование детального управления работой кода на стороне сервера. Ajax-клиент предъявляет к интерфейсу Web-службы совершенно другие требования; они вытекают из наличия задержки, связанной с передачей данных по сети. Большое количество методов, решающих частные задачи, резко снизит практичность клиента, а сервер может не справиться с большим количеством запросов, передаваемых по сети.

Различие требований к интерфейсу, предъявляемых клиентом и сервером, можно проиллюстрировать на примере. Сравним разговор с обменом письмами. (Говоря о письмах, мы имеем в виду не электронную почту, а именно

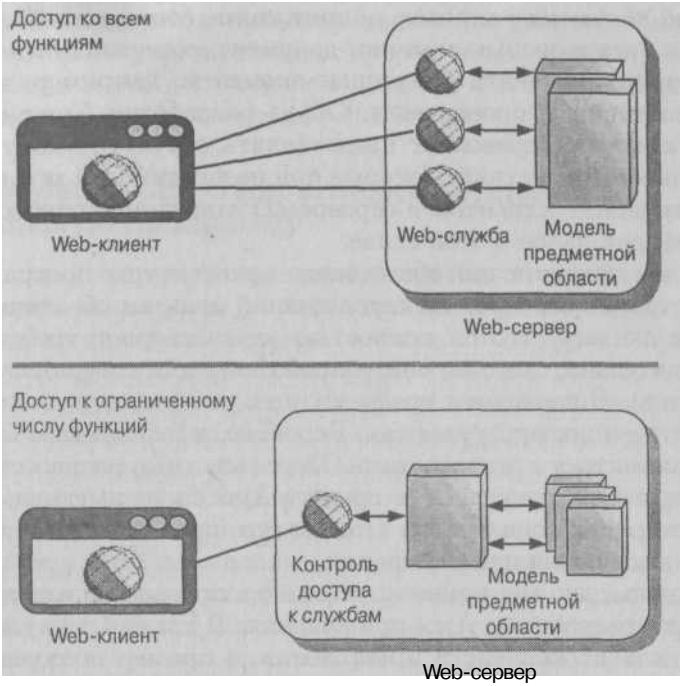


Рис. 5.5. Одна из систем предоставляет Ajax-клиенту все объекты как Интернет-службы, а при создании другой использован образ разработки Facade, который гарантирует доступ лишь к ограниченному набору функций. Уменьшая число общедоступных методов, мы снижаем риск повредить клиентский код при реструктуризации серверных программ

письма, написанные на бумаге, доставка которых занимает несколько дней.) Когда два человека беседуют, как минимум несколько фраз, которыми они обмениваются друг с другом, посвящено теме "как дела?". При написании письма никто не позволяет себе задать один подобный вопрос и ждать ответа на него. Отправитель подробно описывает, как его здоровье, как он провел отпуск, какие новости он узнал недавно, — все это оформляется в виде одного документа.

Путем объединения отдельных запросов, предназначенных для передачи по сети, в один большой документ архитектура, ориентированная на использование сетевых служб, позволяет более эффективно применять доступные сетевые ресурсы. Как правило, проблема быстродействия линий связи решается проще, чем проблема задержки при передаче по сети. Существует также проблема выработки стандартов для передачи больших объемов данных в формате XML посредством знакомого всем протокола HTTP, но ее Мы рассматривать не будем. Если мы проанализируем возможности, предоставляемые Ajax, то увидим, что в нашем распоряжении есть встроенные в браузер средства поддержки HTTP и XML, поэтому имеет смысл создавать распределенную модель предметной области на базе документов.

Типичный документ, например данная книга, состоит из абзацев, заголовков, таблиц и рисунков. Аналогично, документ, содержащийся в обращении к службе, может содержать различные элементы, например, запросы, данные для обновления и оповещения. Образ разработки Command, который обсуждался в главе 3, позволяет представлять структуру документа в виде последовательности действий (которые при необходимости можно отменять), передаваемых между клиентом и сервером. Реализация подобной структуры будет рассмотрена далее в этой главе.

На этом мы заканчиваем обсуждение архитектуры программного обеспечения на стороне сервера. На сегодняшний день ни об одном из ее вариантов нельзя сказать, что он полностью удовлетворяет требованиям Ajax. Это и неудивительно, так как они разработаны для совершенно других типов приложений. В настоящее время ведутся работы по реализации средств Ajax в существующих инструментах. Вероятно, в ближайшем будущем можно будет ознакомиться с результатами. Перед многими разработчиками стоит задача обеспечить взаимодействие средств Ajax с уже имеющимися системами. Мы будем рады, если анализ архитектур, проведенный в данной главе, поможет принять правильные решения.

Предположим, что мы приняли решение в дользу той или иной архитектуры и начали разработку Ajax-приложения. В главе 4 мы уже обсуждали архитектуру клиентской части приложения, а пример получения с сервера данных в формате XML был рассмотрен в главе 2. XML — удобный и популярный, но не единственный возможный формат для обмена данными между клиентом и сервером. В следующем разделе мы обсудим все возможности организации подобного взаимодействия.

5.4. Частные решения: обмен данными

В предыдущем разделе мы рассматривали глобальные архитектурные решения и обсуждали вопрос о том, насколько они подходят для создания Ajax-приложений. Мы постоянно подчеркиваем важность обмена данными между клиентскими программами и моделью предметной области, поддерживаемой на сервере. Прочитав материал, изложенный выше, можно сделать вывод, что, правильно выбрав базовые средства, мы автоматически решим последующие задачи. Увы, это совсем не так. И вы в этом убедитесь, прочитав остальную часть главы. Если мы внимательно рассмотрим вопрос обмена данными, то увидим, что для данной задачи существует множество решений. В последующих разделах мы кратко опишем их и попытаемся выяснить, какой язык наилучшим образом подходит для обмена данными в рамках приложения Ajax. Имея такую информацию, мы сможем принимать более обоснованные решения об использовании конкретных средств.

Задача обмена данными между программами, составляющими приложение Ajax, не имеет аналогов в области классического программирования и, следовательно, изучена крайне мало. Мы попытаемся исправить положение дел. В первую очередь выделим четыре категории взаимодействия с пользователем: с участием только клиента, ориентированное на содержимое, ори-

Ротированное на сценарий и ориентированное на данные. Взаимодействие, поддержке которого участвует лишь клиент, реализуется наиболее просто, мы вкратце обсудим его в следующем разделе, а затем на конкретном призере подробно рассмотрим остальные три категории.

5.4.1. Взаимодействие, затрагивающее только клиентскую программу

Данный тип взаимодействия характеризуется тем, что действия пользователя обрабатываются сценарием, который выполняется в среде браузера. Обращаться к ресурсам на стороне сервера (в частности, к уровню представления) нет необходимости, а это позволяет обеспечить быстрый отклик программы и снижает нагрузку на сервер. Подобное взаимодействие подходит для несложных вычислений, например, для подсчета общей суммы товаров, заказанных пользователем. Для того чтобы такой подход дал желаемые результаты, программы, расположенные на стороне клиента и поддерживающие взаимодействие, не должны меняться в течение сеанса работы пользователя и объем их должен быть небольшим. Для приложения, предназначенного для поддержки интерактивного магазина, эти требования выполняются автоматически. Число изделий, предлагаемых пользователю, относительно невелико (по крайней мере, не тысячи пунктов, как в случае библиотеки), и цены на товары не меняются каждую минуту (в отличие, например, от котировок акций на бирже). Данный тип взаимодействия уже обсуждался в главе 4 при рассмотрении контроллера, расположенного на стороне клиента, поэтому мы больше не будем уделять ему внимание.

Остальные три категории предполагают обращение к серверу и отличаются в основном передаваемыми данными. Главные различия между этими типами взаимодействия описаны в следующих разделах; там же рассмотрены преимущества и недостатки каждого из них.

5.4.2. Пример отображения информации о планетах

Перед тем как заняться изучением различных механизмов обмена данными, представим себе простой пример, на котором будем проверять то или иное решение. Наше приложение будет представлять информацию о планетах Солнечной системы. В главном окне представлено изображение Солнечной системы с пиктограммой для каждой планеты. На сервере хранятся сведения о планетах; щелчком на пиктограмме, соответствующей планете, надо отобразить эти сведения в окне (рис. 5.6). Сейчас мы не будем использовать объект Objectviewer, который рассматривался в главе 4, но еще вернемся к нему.

В данном случае нас больше всего интересует процесс доставки данных, которые должны быть отображены. Мы обсудим форматы данных, передаваемых сервером, но не будем углубляться в детали их генерации, так как этот вопрос был рассмотрен в главе 3. В листинге 5.1 показана заготовка клиентского кода приложения. На ее основе мы будем исследовать различные Механизмы доставки данных.

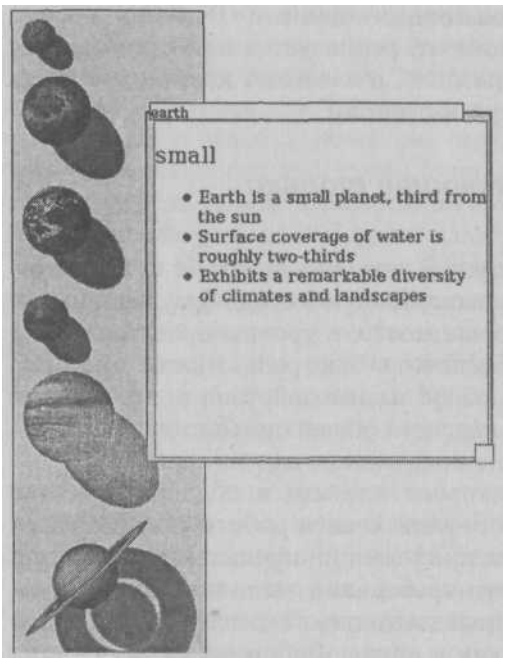


Рис. 5.6. После щелчка на пиктограмме, соответствующей планете, информация о ней выводится в окне

Листинг 5.1. Содержимое файла popups .html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"

"http://www.w3.org/TR/xhtmll/DTD/xhtmll-strict.dtd">
<html> <head> <title>Planet Browser</title>
<link rel=stylesheet type="text/css"
  href="main.css"/>
<link rel=stylesheet type="text/css"
  href="windows.ess"/>
<link rel=stylesheet type="text/css"
  href="planets.css"/>
// О Включение библиотек JavaScript <script
type="text/javascript"
  src="x/x_core.js"x/script>
<script type="text/javascript"
  src="x/x_event.js"x/script>
<script type="text/javascript"
  src="x/x_drag.js"></script>
<script type="text/javascript"
  src="windows.js"X/script>
<script type="text/javascript"
  src="net.js"></script>
<script type="text/javascript">
window.onload=function(){
  var pbar=document.getElementByld("planets");
  var children=pbar.getElementsByTagName("div");
  for(var i=0;i<children.length;i++){
```

```
    // © Связывание с пиктограммами обработчиков
событий
    children[i].onclick=showlnfo;
  }
J
</script>
</head> <body>
// 0 Включение пиктограмм, соответствующих планетам <div
class="planetbar" id="planets">
<div class="planetbutton" id="mercury">
 <img src="img/ball-mercury.gif"
alt="mercury"/> </div> <div class="planetbutton" id="venus">
 <img src="img/ball-venus.gif" alt="venus"/>
</div> <div class="planetbutton" id="earth">
 <img src="img/ball-earth.gif" alt="earth"/>
</div> <div class="planetbutton" id="mars">
 <img src="img/ball-mars.gif" alt="mars"/>
</div> <div class="planetbutton" id="jupiter">
 <img src="img/ball-jupiter.gif"
alt="jupiter"/> </div> <div class="planetbutton" id="saturn">
 <img src="img/ball-saturn.gif" alt="saturn"/>
</div> <div class="planetbutton" id="uranus">
 <img src="img/ball-uranus.gif" alt="uranus"/>
</div> <div class="planetbutton" id="neptune">
 <img src="img/ball-neptune.gif"
alt="neptune"/> </div> <div class="planetbutton" id="pluto">
 <img src="img/ball-pluto.gif" alt="pluto"/>
</div> </div>
</body> </html>
```

В состав нашего файла мы включили несколько JavaScript-библиотек О. Средства, предоставляемые net.js, обеспечивают поддержку низкоуровневых HTTP-запросов, используя для этой цели объект XMLHttpRequest, который обсуждался в главе 2. В файле windows, js определен объект окна, допускающего перетаскивание. Это окно мы используем для отображения информации. Детали реализации окна в данном случае не имеют значения; нас интересует только порядок вызова конструктора.

var MyWindow=new Window(bodyDiv,title,x,y, w,h);

где bodyDiv — это элемент DOM, который выводится в окне; title — строка, отображаемая в заголовке окна, а параметры x, y, w и h определяют начальные размеры окна. Задавая элемент DOM в качестве параметра, мы обеспечиваем достаточную гибкость при отображении данных к окне. Полностью код объекта Window можно скопировать с Web-узла, посвященного данной книге. Посредством HTML-кода, содержащегося в файле, мы определяем элемент div для каждой планеты ©, а в функции window.onload © связываем ° пиктограммами планет обработчики onclick. В качестве обработчиков используются функции showlnf o ( ), которые в данном листинге не определены. В этой главе мы обсудим несколько вариантов их реализации. Рассмотрим Действия, которые мы можем предпринять тогда, когда нам требуется загрузка данных.

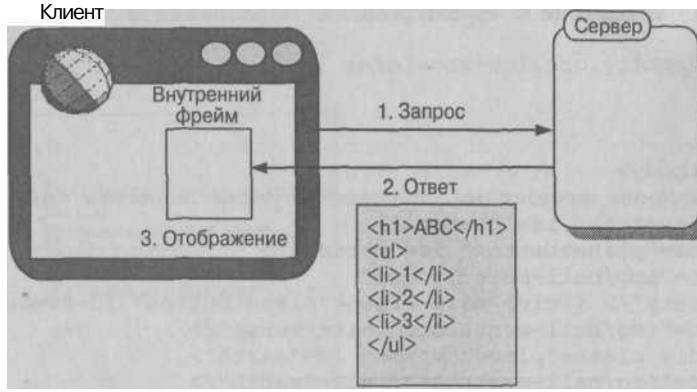


Рис. 5.7. Архитектура Ajax-приложения, ориентированная на содержимое. Клиент создает элемент IFrame и передает серверу запрос на получение информации. Содержимое генерируется моделью, представлением и контроллером уровня представления и возвращается элементу IFrame. На уровне клиента никакие требования к модели предметной области не предъявляются

5.4.3. Взаимодействие, ориентированное на содержимое

Наши первые шаги по использованию подхода Ajax напоминают действия, которые предпринимаются при создании классического Web-приложения. В этом нет ничего удивительного; как было сказано в главе 1, в названии первых велосипедов присутствовало слово "лошадь". Взаимодействие, ориен-: тированное на содержимое, соответствует классическому подходу, но может иметь место и в Ajax-приложениях.

Общие сведения

При взаимодействии, ориентированном на содержимое, HTML-данные гене! рируются сервером и отображаются в составе элемента IFrame, включенного на главную Web-страницу. Элементы IFrame обсуждались в главе 2. Их можно определить в HTML-коде страницы или сгенерировать с помощью программы. При программной генерации мы получаем динамический интерфейс, а окно браузера напоминает оконный диспетчер. На рис. 5.7 условно показана архитектура, ориентированная на содержимое.

В листинге 5.2 показана реализация обработчика событий рассматриваемого приложения. При создании обработчика использовался подход, ориентированный на содержимое.

Листинг 5.2. Содержимое файла ContentPopup.js

```
var offset=8;
function showInfo(event){
  var planet=this.id;
  var infoWin=new ContentPopup(
    "info_"+planet+".html",
    planet+"Popup",
```

```
    planet,offset,offset,320,320
  );
  offset+=32;
}
function'
ContentPopup(url,winEl,displayStr,x,y,w,h){
  var bod=document.createElement("div");
  document.body.appendChild(bod);
  this.iframe=document.createElement("iframe");
  this.iframe.className="winContents";
  this.iframe.src=url;
  bod.appendChild(this.iframe);
  this.win=new windows.Window(bod,displayStr,x,y,w,h);
}
```

Функция showlnf o () выполняет роль обработчика события для DOM-элемента, представляющего планету. В обработчике ссылка this указывает на элемент DOM. Чтобы определить, для какой из планет отображаются данные, используется идентификатор элемента.

Мы определили объект ContentPopup, который генерирует универсальный объект Window, создает элемент I Frame, предназначенный для просмотра содержимого, и загружает в него ресурс, определяемый URL. В данной ситуации мы лишь формируем имя статического HTML-файла и оформляем его в виде URL. В более сложном случае, если данные генерируются динамически, нам, возможно, придется добавить к URL строку параметров. Простой файл, предназначенный для отображения в составе iFrame (листинг 5.3), генерируется сервером.

Листинг 5.3. Содержимое файла info_earth.html

```
<html> <head>
<link rel=stylesheet type="text/css" href="../style.css"/>
</head>
<body class="info">
<div class="framedlnfo" id="info">
<div class="title" id="infotitle">earth</div>
<div class="content" id="infocontent">
A small blue planet near the outer rim of the galaxy,
third planet out from a middle-sized sun.
</div> </div> </body> </html>
```

В данном документе нет ничего примечательного: мы лишь используем обычный HTML-файл, как и для классического Web-приложения.

При использовании архитектуры, ориентированной на содержимое, клиенту требуется лишь ограниченная информация о бизнес-логике приложения, Достаточная для размещения элемента IFrame и формирования URL, определяющего содержимое для загрузки. Связь между клиентом и уровнем представления очень слабая; основная ответственность за предоставление содержимого возлагается на сервер. Преимущество данного типа взаимодействия состоит в том, что оно позволяет использовать готовые HTML-документы.

Он может оказаться полезен в двух случаях: при включении содержимого! предоставляемого внешними узлами, например, информации, получаемой от деловых партнеров, и при отображении данных, полученных от уже существующих приложений. В этих случаях использование формата HTML может обеспечивать высокую эффективность, и, как правило, не возникает необходимости преобразовывать информацию в другой вид. В качестве примера можно привести документы, содержащие справочную информацию. Использовать в приложениях Ajax подход, ориентированный на содержимое, целесообразно в тех случаях, когда в классических приложениях применяются всплывающие окна. Тем не менее данная архитектура имеет ограниченное применение. Причины этого нетрудно понять, зная ее недостатки.

Проблемы и ограничения

Поскольку архитектура, ориентированная на содержимое, мало отличается от привычных Web-страниц, ей присущи все ограничения данного подхода. Документ в элементе I Frame изолирован от остальной части страницы, в которую он включается. В результате информация на экране выглядит фрагментированной. Правда, при размещении содержимого возможны различные варианты. Несмотря на то что I Frame представляет собой прямоугольное окно, содержащее дочерний документ, для него может быть задан прозрачный фон, в результате чего содержимое окна будет смешиваться с содержимым родительского документа.

У некоторых разработчиков может возникнуть мысль использовать данный механизм для доставки динамических частей страницы, но в этом случае могут возникнуть проблемы при работе с элементами IFrame. Каждый элемент I Frame поддерживает собственный контекст, и объем вспомогательного кода, необходимого для обеспечения взаимодействия содержимого IFrame с родительским документом, может оказаться значительным. При взаимодействии со сценариями в других фреймах возникают еще более серьезные проблемы. Мы вернемся к этому вопросу при рассмотрении подхода, ориентированного на сценарий.

Нельзя также забывать о проблемах, связанных с практичностью традиционных Web-приложений. Во-первых, каждый запрос на получение содержимого предполагает повторное получение статических данных. Во-вторых, несмотря на то, что при обновлении данных в основном документе автоматически принимаются меры для подавления мерцания, в элементе IFrame этот эффект может присутствовать. Мерцания, правда, можно избежать, реализовав дополнительный код для отображения сообщения "поверх" фрейма.

Итак, в качестве первого термина нашего словаря, описывающего запросы к серверу Ajax, мы записали понятие "ориентированный на содержимое". Применимость данного подхода ограничена, однако учесть такую возможность все же необходимо. Существует ряд задач, которые нельзя удовлетворительно решить в рамках подхода, ориентированного на содержимое. В качестве примера таких задач можно привести обновление части компонента, например, отдельной пиктограммы или. строки таблицы. Единственный способ решить такую задачу — передавать JavaScript-код.

Варианты применения подхода, ориентированного на содержимое

До сих пор, рассматривая архитектуру, ориентированную на содержимое, мы предполагали, что для получения данных используется элемент IFrame. В качестве альтернативного решения можно рассмотреть генерацию фрагмента HTML-кода, как реакцию на асинхронный запрос, и присвоение ответа свойству innerHTML элемента DOM текущего документа. Более подробно мы рассмотрим данное решение в главе 12.

5.4.4. Взаимодействие, ориентированное на сценарий

Когда мы организуем передачу с сервера JavaScript-файла и выполнение этого файла в среде браузера, мы можем утверждать, что решаемая нами задача нетривиальна. Если JavaScript-код, предназначенный для передачи, генерируется программой, то выполняемые действия еще более сложны. Традиционно клиентские и серверные программы обмениваются данными друг с другом. Передача по сети исполняемого кода обеспечивает дополнительную гибкость. Поддержка мобильного кода реализована лишь в инструментах, предназначенных для создания корпоративных приложений, например Java и .NET. Для этой цели специально предусмотрены технологии RMI, Jini, и .NET Remoting Framework. Разработчики же простых Web-приложений решают подобные задачи постоянно! Логично предположить, что Ajax расширит наши возможности по работе с мобильным кодом.

Общие сведения

В классическом Web-приложении HTML-код и JavaScript-сценарий обычно располагаются в одном файле, и сценарий специально ориентирован на работу с конкретной Web-страницей. Используя Ajax, мы можем загружать сценарии и Web-страницы независимо друг от друга, а это дает возможность модифицировать страницу различными способами, в зависимости от полученного сценария. Коды, составляющие клиентскую часть приложения, могут быть существенно расширены в процессе работы. Это не только предоставляет новые возможности, но и, как вы вскоре увидите, создает проблемы. На рис. 5.8 условно показана архитектура, ориентированная на сценарии.

Первое преимущество данного подхода по сравнению с архитектурой, ориентированной на содержимое, состоит в том, что активные действия передаются элементам "второго плана", что предотвращает мерцание изображения.

Особенности генерируемых сценариев определяются потребностями клиентского уровня. Как и для любой задачи, предполагающей генерацию кода, Успех ее решения зависит от того, удастся ли добиться, чтобы сгенерированный код был простым и использовал существующие библиотеки. Библиотеки могут либо передаваться вместе с новым кодом, либо постоянно присутствовать на стороне клиента.

В любом случае данная архитектура предполагает тесную связь между Уровнями. Для генерации кода на стороне сервера необходимо иметь информацию об API на стороне клиента. При этом возникают две проблемы. Во-первых, модифицируя клиентский или серверный код, можно случайно на-

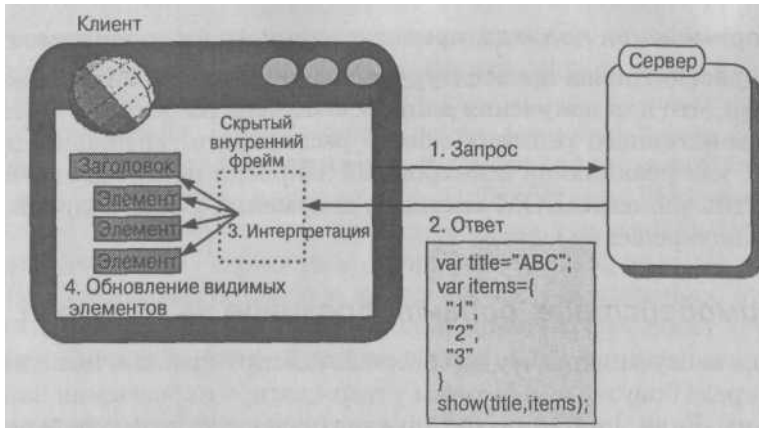


Рис. 5.8. Архитектура Ajax-приложения, ориентированная на сценарии. Клиентская программа передает серверу запрос на получение фрагмента JavaScript-кода. Полученный код интерпретируется. На стороне клиента обеспечиваются точки входа для сгенерированных сценариев, что позволяет сценарию управлять клиентской программой

рушить взаимодействие. Модульный подход к разработке и использование образа разработки Fagade могут несколько снизить подобную опасность. Во-вторых, поток JavaScript-кода ориентирован на конкретного клиента, и организовать его повторное использование намного сложнее, чем, например, потока XML-данных. Следует, однако, заметить, что пригодность к повторному использованию не всегда бывает важна.

Теперь вернемся к примеру отображения информации о планетах Солнечной системы. В листинге 5.4 показан простой API для отображения окон с данными.

Листинг 5.4. Функция showPopupO и вспомогательный код

```
var  offset=8;
function showPopup(name,description) {
  var win=new ScriptlframePopup
    (name,description,offset,offset,320,320);
  offset+=32;
}
function  ScriptlframePopup(name,description,x,y,w,h){
  var bod=document.createElement("div");
  document.body.appendChild(bod);
this.contentDiv=document.createElement("div");
  this.contentDiv.className="winContents";
  this.contentDiv.innerHTML=description;
  bod.appendChild(this.contentDiv);
  this.win=new windows.Window(bod,name,x,y,w,h);
}
```

Мы определили функцию showPopup(), которая получает два параметра и создает объект окна. В листинге 5.5 приведен пример сценария, вызывающего эту функцию.

Листинг 5.5. Содержимое файла script_earth.js

```
var name='earth'; var description="A small blue
planet near the outer rim of the galaxy,"
  +"third planet out from a middle-sized sun.";
showPopup (name,description);
```

В данном случае мы подготавливаем параметры (имя и описание) и вызываем функцию. Для того чтобы такой вызов стал возможен, мы должны загрузить сценарий с сервера и заставить браузер выполнить его. Существуют два способа, позволяющие сделать это. Рассмотрим каждый из них.

Загрузка сценариев в элементы IFrame

Если мы загрузим JavaScript-код, используя HTML-дескриптор <script>, сценарий будет автоматически обработан интерпретатором. С этой точки зрения элемент IFrame не отличается от обычного документа. Мы можем определить метод showlnf o () для создания IFrame и загрузить в него сценарий.

```
function showlnfo(event){
  var planet=this.id;
  varscriptUrl="script_"+planet+".html";
  var
dataframe=document.getElementById('dataframe');
  if (!dataframe){
    dataframe=document.createElement("iframe");
    dataframe.className='dataframe';
    dataframe.id='dataframe';
    dataframe.src=scriptUrl;
    document.body.appendChild(dataframe);
  )else{ dataframe.src=scriptUrl;
  }
}
```

Средства для обработки DOM уже знакомы вам. Если мы используем для загрузки сценария невидимый элемент IFrame, то можем сосредоточить внимание на самом сценарии, поскольку все остальные действия выполняются автоматически. Включим наш пример сценария в HTML-документ, как показано в листинге 5.6.

Листинг 5.6. Содержимое файла script earth.html

```
<html> <head> <script type='text/javascript'
src='script_earth.js'> </script> </head> <body> </body>
</html>
```

Если мы попытаемся загрузить этот код, он не будет работать, так как IFrame формирует собственный JavaScript-контекст и не может непосредственно обратиться к API, определенному в основном документе. Если в сценарии выполняется приведенное ниже выражение, браузер ищет функцию showPopup () в контексте IFrame.

```
showPopup (name, description);
```

Таблица 5.1. Результаты применения оператора `instanceof`

Элемент, в котором был создан объект	Элемент, в котором был вызван оператор `instanceof`	Результаты выполнения оператора `instanceof`
Документ верхнего уровня	Документ верхнего уровня	`true`
Документ верхнего уровня	`IFrame`	`false`
`IFrame`	Документ верхнего уровня	`false`
`IFrame`	`IFrame`	`true`

В простых ситуациях, подобных рассматриваемой, при наличии двух контекстов мы можем указать перед вызовом идентификатор top, формируя тем самым ссылку на документ верхнего уровня.

```
top.showPopup(name,description);
```

Если лее элемент I Frame содержится в составе другого элемента I Frame или если мы хотим запустить наше приложение в системе фреймов, ситуация усложняется.

В загружаемом нами сценарии используется функциональный подход. Если нам понадобится создать в I Frame экземпляр объекта, мы столкнемся с новой проблемой. Предположим, что в файле Planetlnfo.js определен объект Planetlnfo, к которому наш сценарий обращается следующим образом:

```
var pinfo=new PlanetInfo(name,description);
```

Для того чтобы такое обращение было возможно, мы должны импортировать Planetlnfo.js в контекст IFrame, указав дополнительный дескриптор <script>.

```
<script type='text/javascript'
src='Planetlnfo.js'></script> <script
type='text/j avascript'>
  var pinfo=new Planetlnfo(name,description);
</script>
```

Объект Planetlnfo, созданный в Iframe, будет вести себя так же, как и объект, созданный во фрейме верхнего уровня, но они имеют разные прототипы. Если впоследствии мы удалим элемент IFrame, а документ верхнего уровня сохранит ссылку на объект, созданный в составе IFrame, то последующие обращения к методам объекта невозможны. Более того, оператор instanceof даст совершенно неожиданные результаты (табл. 5.1).

Импортирование определения объекта в различные контексты — не такая простая задача, как это может показаться на первый взгляд. Решить проблему можно, определив в API документа верхнего уровня фабричный метод. Например:

```
function createPlanetlnfo(name,description){
  return new Planetlnfo(name,description);
}
```

При этом исчезает необходимость иметь ссылку на собственную версию объекта PlanetInfo.

```
<script type='text/javascript'>
  var pinfo=createPlanetInfo(name,description);
</script>
```

функция showPopupO в листинге 5.4 представляет собой фабрику для объекта ScriptIframePopup.

Данный подход обеспечивает корректную работу сценария. На каждой странице нам необходимо передавать HTML-код, выполняющий роль шаблона, но объем его значительно меньше, чем при использовании архитектуры, ориентированной на содержимое. Основной недостаток данного подхода связан с созданием отдельного контекста JavaScript. Ниже описан способ, позволяющий устранить этот недостаток.

Загрузка сценариев с использованием XMLHttpRequest и eval()

Подобно многим языкам сценариев, JavaScript содержит функцию eval(), которая позволяет передавать интерпретатору JavaScript произвольный текст. Некоторые считают, что использование функции eval() замедляет работу приложения. Это действительно так, если функция eval() систематически вызывается для малых сценариев. Однако, поскольку эта функция доступна нам, мы можем использовать ее для выполнения кода, загруженного с помощью объекта XMLHttpRequest. Если функцию eval() вызывать относительно редко для сценариев большого объема, она обеспечивает приемлемую производительность.

Переписав наш пример для работы с функцией eval(), получим следующий код:

```
function showInfo(event){
  var planet=this.id;
  var scriptUrl="script_"+planet+".js";
  new net.ContentLoader(scriptUrl, evalScript); )
function evalScript(){
  var script=this.req.responseText;
  eval(script); }
```

Теперь метод showInfoO использует объект XMLHttpRequest (помещенный в класс ContentLoader) для получения сценария с сервера; причем необходимость включать его в состав HTML-страницы отпадает. Вторая функция, evalScript(), представляет собой функцию обратного вызова; ссылка на нее передается ContentLoader. Когда функция evalScript() получает управление, может быть прочитано значение свойства responseText, принадлежащего объекту XMLHttpRequest. Весь сценарий выполняется не в отдельном контексте IFrame, а в контексте текущей страницы.

Теперь мы описали новый термин — "ориентированный на сценарий". Заметим, что данный подход допускает две реализации; посредством IFrame или eval(). Сравним подход, ориентированный на сценарий, с подходом, ориентированным на содержимое.

Проблемы и ограничения

При непосредственной загрузке сценария с сервера общий объем сообщения уменьшается; таким образом, сетевое соединение освобождается для других задач. Кроме того, в результате подобного подхода логика отделяется от представления, а изменение отображаемых данных не ограничивается прямоугольной областью на экране, как это происходит при использовании подхода, ориентированного на содержимое.

Очевидным недостатком является взаимная зависимость клиентского и серверного кода. JavaScript-код, получаемый от сервера, крайне трудно использовать в другом контексте, поэтому приходится специально разрабатывать программы для Ajax-клиентов. После начала эксплуатации изменить API клиента очень сложно.

В целом подход, ориентированный на сценарий, можно рассматривать как шаг в правильном направлении. Ajax-приложение начинает больше походить именно на приложение, а не на документ. Следующий вариант взаимодействия позволяет ослабить взаимную зависимость клиента и сервера.

5.4.5. Взаимодействие, ориентированное на данные

При использовании подхода, ориентированного на сценарий, программа, выполняющаяся в среде браузера, ведет себя подобно традиционному "толстому" клиенту, т.е. запросы к серверу на получение данных формируются в фоновом режиме и намного меньше влияют на работу пользовательского интерфейса. Однако сам сценарий представляет собой специализированный код клиента, выполняющегося в среде браузера.

Общие сведения

Иногда бывает необходимо организовать использование данных, полученных Ajax-клиентом, другими программами, например, интеллектуальными клиентами Java или .NET, либо программным обеспечением мобильных телефонов или КПК. В этих случаях JavaScript-инструкции плохо применимы; более уместным был бы "нейтральный" формат данных.

При взаимодействии, ориентированном на данные, сервер передает лишь данные, которые обрабатываются не интерпретатором JavaScript, а самой клиентской программой. Архитектура, ориентированная на данные, условно показана на рис. 5.9.

Большинство примеров, приведенных в данной книге, создано в рамках подхода, ориентированного на данные. Наиболее очевидный формат данных — XML, но возможны также и другие форматы.

Использование XML-данных

В настоящее время формат XML применяется очень часто. Среда браузера, в которой выполняется Ajax-приложение и, в частности, объект XMLHttpRequest, предоставляет средства поддержки XML-данных. Если объект XMLHttpRequest получает ответ типа application/xml или text/xml, он оформ-

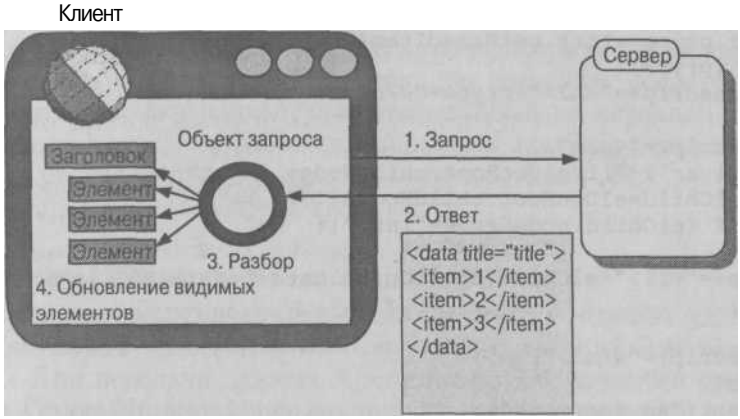


Рис. 5.9. В системе, ориентированной на данные, сервер в ответ на запрос возвращает поток низкоуровневых данных (представленных, например, в формате XML). Эти данные обрабатываются на уровне клиента и используются для обновления клиентской модели и (или) пользовательского интерфейса

ляет его в виде структуры DOM (Document Object Model). В листинге 5.7 приведен код приложения, отображающего данные о планетах, адаптированный для работы с информацией в формате XML.

Листинг 5.7. Содержимое файла DataXMLPopup.js

```
var offset=8; function
showPopup(name,description){
  var win=new DataPopup(name,description,offset,offset,320,320);
  offset+=32;
}
function
DataPopup(name,description,x,y,w,h){
  var bod=document.createElement("div");
  document.body.appendChild(bod);
this.contentDiv=document.createElement("div");
  this.contentDiv.className="winContents";
  this.contentDiv.innerHTML=description;
  bod.appendChild(this.contentDiv);
  this.win=new windows.Window(bod,name,x,y,w,h);
}
function showInfo(event){
  var planet=this.id;
  var scriptUrl=planet+".xml";
  new net.ContentLoader(scriptUrl,parseXML);
}
function parseXML(){
  var name="";
  var descrip="";
  var xmlDoc=this.req.responseXML;
  var elDocRoot=xmlDoc.getElementsByTagName("planet")[0];
  if (elDocRoot){
    attrs=elDocRoot.attributes;
```

```
    name=attrs.getNamedItem("name").value;
    var ptype=attrs.getNamedItem("type").value;
    if (ptype){
      descrip+="<h2>"+ptype+"</h2>";
    }
    descrip+="<ul>";
    for(var i=0;i<elDocRoot,childNodes.length;i++){
      elChild=elDocRoot.childNodes[i];
      if (elChild.nodeName=="info"){
descrip+="<li>"+elChild.firstChild.data+"</li>(BBSS)n";
      }
    }
    descrip+="</ul>";
  }else{
    alert("no document");
  }
  top.showPopup(name,descrip);
}
```

Функция showinfo() обращается к объекту XMLHttpRequest, содержащемуся в составе ContentLoader, и объявляет в качестве функции обратного вызова parseXML(). В данном случае функция обратного вызова сложнее, чем evalScript(), которую мы рассматривали в разделе 5.6.3. Это связано с тем, что нам необходимо поддерживать навигацию по структуре DOM, извлекать данные и вызывать метод *showPopup*. В листинге 5.8 содержится пример ответа, сгенерированного сервером в формате XML.

Листинг 5.8. Содержимое файла earth.xml

```
<planet name="earth" type="small">
  <info id="a" author="dave" date="26/05/04">
    Earth is a small planet, third from the sun
  </info>
  <info id="b" author="dave" date="27/02/05">
    Surface coverage of water is roughly two-thirds
  </info>
  <info id="c" author="dave" date="03/05/05">
    Exhibits a remarkable diversity of climates and landscapes
  </info>
</planet>
```

Существенным преимуществом XML является тот факт, что информация, представленная в этом формате, естественным образом структурирована. Так, в данном примере содержится ряд дескрипторов <info>, которые функция parseXML () преобразует в маркированный HTML-список.

Таким образом, благодаря использованию XML нам удалось разделить уровни сервера и клиента. Код клиента и сервера можно изменять, независимо один от другого, при условии, конечно, что они будут поддерживать формат документа. Однако подход, ориентированный на сценарий, имел очень важное преимущество: вся работа выполнялась JavaScript-интерпретатором. Рассмотрим решение, предполагающее использование JSON. Оно позволяет объединить преимущества обеих архитектур.

Использование JSON-данных

Имя для объекта XMLHttpRequest было выбрано не совсем удачно. На самом деле он может принимать не только XML, но и любую текстовую информацию. Для передачи данных Ajax-клиенту очень удобен формат JSON (JavaScript Object Notation), так как он позволяет представить в компактном виде граф объектов JavaScript. В листинге 5.9 показано, как молено адаптировать пример приложения, предоставляющего информацию о планетах, для использования JSON.

Листинг 5.9. Содержимое файла DataJSONPopup. j s

```
function showlnfo(event)
{
  var planet=this.id;
  var scriptUrl=planet+".json";
  new net.ContentLoader(scriptUrl,parseJSON);
}

function parseJSON()                                                    ,
{
  var name="";
  var descrip="";
  var jsonTxt=net.req.responseText;
  var jsonObj=eval("("+jsonTxt+")");
  name=j sonObj.planet.name
  var ptype=jsonObj.planet.type;
  if (ptypeH
    descrip+="<h2>"+ptype+"</h2>";
  )
  var infos=jsonObj.planet.info;
  descrip+="<ul>";
  for(var i in infos){
    descrip+="<li>"+infos[i]+"</li>\n";
  }
  descrip+="</ul>";
  top.showPopup(name,descrip);
```

Здесь мы также загружаем данные с помощью ContentLoader. Функцией обратного вызова в этом случае является parse JSON (). Текст ответа представляет собой JavaScript-выражение, поэтому мы можем создать граф объектов путем вызова eval ().

```
var jsonObj=eval("("+jsonTxt+")") ;
```

Заметьте, что перед обработкой выражения нам надо поместить его в скобки. Теперь мы можем обращаться к свойствам объекта по имени, а это позволяет сократить размеры кода и сделать его более удобным для восприятия, чем методы обработки структуры DOM, которые мы использовали при работе с XML. Здесь метод showPopup() не приводится, поскольку он выглядит точно так же, как и в листинге 5.7.

Какой же вид имеют JSON-данные? В листинге 5.10 показана информация о планете Земля, представленная как строка JSON.

Листинг 5.10. Содержимое файла earth, json

```
{ "planet":
 {
  "name": "earth",
  "type": "small",
  "info":
  [
    "Earth is a small planet, third from the sun",
    "Surface coverage of water is roughly two-thirds",
    "Exhibits a remarkable diversity of climates and landscapes"
  ]
 }
```

С помощью фигурных скобок определяются ассоциативные массивы, а квадратные скобки обозначают массивы с числовыми индексами. Допускается вложенность любых видов скобок. В приведенном примере мы определили объект planet, содержащий три свойства. Свойства name и type представляют собой обычные строки, а свойство info является массивом.

Формат JSON используется реже, чем XML, но JSON-данные могут быть обработаны любым интерпретатором JavaScript, включая Mozilla Rhino на базе Java и Microsoft JScript .NET. Библиотеки JSON-RPC содержат средства разбора JSON для различных языков программирования (соответствующие ссылки приведены в конце данной главы), а также инструмент JavaScript "Stringiner", предназначенный для преобразования JavaScript-объектов в строки JSON. Таким образом, JSON можно рассматривать как формат для двустороннего взаимодействия. Если интерпретатор JavaScript доступен и на стороне клиента, и на стороне сервера, целесообразно выбрать формат JSON. В рамках проекта JSON-RPC были также разработаны библиотеки, предназначенные для разбора и генерации JSON-сообщений и ориентированные на языки, которые часто используются при разработке программ, выполняемых на стороне сервера.

Теперь в нашем словаре появился термин "ориентированный на данные". Кроме того, мы выяснили, что для обмена данными можно использовать не только XML, но и другие форматы.

Использование XSLT

Альтернативой написанию программ для обработки дерева DOM и создания flTML-данных (этот подход был описан в разделе 5.7.3) является использование XSLT-преобразования для автоматического конвертирования XML-информации в формат XHTML. Этот подход представляет собой нечто среднее между взаимодействием, ориентированным на данные, и взаимодействием ориентированным на содержимое. С точки зрения сервера эта архитектура ориентирована на данные, а с точки зрения клиента — на содержимое. Такой подход позволяет быстрее и проще получить требуемый результат, но для него характерны те же ограничения, что и для взаимодействия, ориентированного на содержимое, а именно, данные ответа затрагивают лишь прямоугольную область в окне. XSLT-преобразование будет подрой гее обсуждаться в главе 11.

Проблемы и ограничения

Главный недостаток подхода, ориентированного на данные, состоит в том, что основная нагрузка, связанная с разбором информации, ложится на клиентскую программу. Это неизбежно ведет к усложнению кода на уровне клиента. Однако если данный подход получит широкое распространение, затраты на создание кода могут быть компенсированы за счет его повторного использования или оформления некоторых функций в виде библиотеки.

Три рассмотренных нами подхода описывают различные варианты систем: от классического Web-приложения до "толстого" клиента, характерного для настольных систем. К счастью, эти три архитектуры не являются взаимоисключающими и могут быть совместно использованы в рамках одного приложения.

Очевидно, что между клиентом и сервером осуществляется двухстороннее взаимодействие. В завершение данной главы рассмотрим, как клиент передает данные серверу.

5.5. Передача данных серверу

До сих пор мы уделяли внимание одной стороне взаимодействия — передаче сервером информации клиенту. Однако в большинстве приложений пользователю необходимо не только получать сведения о модели предметной области, но и управлять ею. В многопользовательской среде надо также обеспечить обновление отображаемых данных с учетом изменений, внесенных другими пользователями.

Рассмотрим сначала обновление с учетом изменений, которые мы вносим сами. Существуют два механизма передачи данных на сервер: HTML-формы и объект XMLHttpRequest. Рассмотрим кратко каждый из них.

5.5.1. Использование HTML-форм

В классическом Web-приложении элементы HTML-форм реализуют стандартный механизм ввода данных пользователем. Элементы формы объявляются с помощью средств разметки HTML.

```
<form method="POST" action="myFormHandlerURL.php">
  <input type="text" name="username"/>
  <input type="password" name="password"/>
  <input type="submit" value="login"/>
</form>
```

В результате обработки данного фрагмента кода отображаются два пустых поля редактирования. Если пользователь введет в полях значения dave и letmein и активизирует кнопку submit, будет сформирован HTTP-запрос POST и передан ресурсу myFormHandlerURL.php. В теле запроса будет содержаться текст username=dave&password=letmein. В большинстве систем Web-программирования разработчику непосредственно не доступны закодированные данные формы. Вместо этого ему предоставляются пары имя-значения в виде ассоциативного массива или специальных переменных.

В настоящее время авторы Web-страниц часто включают в них небольшие фрагменты JavaScript-кода, позволяющие проверить корректность введенных данных перед передачей их серверу. Для того чтобы организовать проверку, надо модифицировать нашу простую форму так, как показано ниже.

```
<form id="myForm" method="POST" action=""
      onsubmit="validateForm(); return false;">
  <input type="text" name="username"/>
  <input type="password" name="password"/>
  <input type="submit" value="login"/>
</form>
```

В этом случае функция, выполняющая проверку, будет иметь следующий вид:

```
function validateForm(){
  var form=document.getElementById('myForm');
  var user=form.elements[0].value;
  var pwd=form.elements[1].value;
  if (user && user.length>0 && pwd && pwd.length>0){
    form.action='myFormHandlerURL.php';
    form.submit();
  }
  else
  {
    alert("please fill in your credentials before logging in");
  }
}
```

Первоначально форма определяется без атрибута action. Обращение по реальному URL будет произведено только в случае положительного результата проверки. С помощью JavaScript-кода можно также модифицировать форму, организовав блокирование кнопки submit для предотвращения повторной передачи- данных, кодирование пароля перед формированием запроса и т.д. Эти и многие другие приемы многократно описаны в литературе, поэтому

^ibi не будем рассматривать их здесь. В главах 9 и 10 приведены примеры расширения возможностей HTML-форм.

Элемент формы можно также сформировать программно и реализовать передачу данных так, чтобы она не прерывала работу пользователя. Если мы сделаем форму невидимой, то сможем передавать данные незаметно для пользователя. Этот подход реализован в листинге 5.11.

Листинг 5.11. Функция submitData()

```
function addParam(form,key,value)
{
  var input=document.createElement("input");
  input.name=key;
  input.value=value;
  form.appendChild(input);
}
function submitData(url,data)
{
  var form=document.createElement("form");
  form.action=url;
  form.method="POST";
  for (var i in data){
    addParam(form,i,data[i]);
  }
  form.style.display="none";
  document.body.appendChild(form);
  form.submit();
}
```

Функция submitData () создает форму и в цикле включает в нее данные, используя для этого функцию addParam (). Обращение к функции submitData () имеет следующий вид:

```
submitData(
  "myFormHandlerURL.php",
  {username:"dave",password:"letmein"}
);
```

Для использования данного подхода не требуется создавать большой объем кода, но он не позволяет перехватывать ответ сервера. Мы можем поместить форму в невидимый элемент iFrame, а затем организовать разбор результатов, но подобное решение оказывается достаточно громоздким. В ряде случаев удобнее воспользоваться для этой цели объектом XMLHttpRequest.

5.5.2. Использование объекта XMLHttpRequest

Объект XMLHttpRequest уже рассматривался нами в этой главе, а также в главе 2. С точки зрения клиентского кода разница между чтением и обновлением незначительна. Нам надо лишь указать метод POST и передать параметры формы.

В листинге 5.12 показан код объекта ContentLoader, который мы раз-Я работали в разделе 3.1. Мы несколько изменили его для того, чтобы можнпИ было включать в запрос параметры и указывать произвольный HTTP-метод

Листинг 5.12. Объект ContentLoader

```
//O Для конструктора предусмотрены дополнительные параметры
net.ContentLoader=function
  (url,onload,onerror,  method,params,contentType)

  this.onload=onload;
  this.onerror=(onerror) ? onerror :
  this.defaultError;
  this.loadXMLDoc(url,method,params,contentType); }
  net.ContentLoader.prototype.loadXMLDoc
   =function(ur1,method,params,contentType){
  if (!method){
    method="GET";


  if (icontentType && method=="POST"){
    contentType="application/x-www-form-urlencoded";


  if (window.XMLHttpRequest){
    this.req=new XMLHttpRequest();


  else
  if (window.ActiveXObject){
    this.req=new ActiveXObject("Microsoft.XMLHTTP");


  if (this.req){
    try{
        this.req.onreadystatechange=net.ContentLoader.onReadyState;

      // HTTP-метод
      this.req.open( method,url,true);

      // Тип содержимого
     if (contentType){
        this.req.setRequestHeader("Content-Type", contentType);


      // Параметры запроса
      this.req.send( params);
    }catch (err){
      this.onerror.call(this);

  }
```

Конструктору объекта мы передаем несколько новых параметров О. Из них обязательными являются URL (в соответствии с атрибутом action формы) и обработчик onload. Можно также задать HTTP-метод, параметры и тип запроса. Заметьте, что при передаче пар ключ-значение посредством метода POST надо указать тип содержимого application/x-www-f orm-urlencoded. Если при вызове функции тип явно не задан, мы устанавливаем это значение автоматически. HTTP-метод используется при вызове метода open () объекта XMLHttpRequest а параметры — при обращении к методу send( ). Таким образом, вызов конструктора имеет следующий вид:

```
var loader=net.ContentLoader(
    'myFormHandlerURL.php', showResponse, null,
    ' POST', 'username=dave&password=letmein'
);
```

В результате выполнения этого выражения будет сформирован такой же запрос, как и при использовании метода submitData(), представленного в листинге 5.11. Заметьте, что параметры передаются в виде строки, которая кодируется так же, как и при передаче данных формы, например:

```
name=daveSjob=book&work=Ajax_In+Action
```

Так действуют основные механизмы передачи данных серверу. Они могут активизироваться в результате различных событий: ввода пользователем текста, перемещения мыши, перетаскивания объекта и т.д. В следующем разделе мы вернемся к объекту Objectviewer, рассмотренному в главе 4, и выясним, как можно управлять обновлением модели предметной области.

5.5.3. Управление обновлением модели

В главе 3 мы рассмотрели универсальный объект Objectviewer, предназначенный для представления сложных моделей, и обсудили простой пример, в котором данный объект используется для просмотра информации о планетах. Каждый из объектов, представляющих планеты Солнечной системы, содержит несколько параметров, а некоторые текстовые свойства — диаметр и расстояние от Солнца — мы определили как редактируемые. Изменение любого свойства перехватывается центральной функцией обработки событий, которую мы использовали для представления отладочной информации в строке состояния браузера. (Возможность записывать данные в строку состояния в последних версиях Mozilla Firefox ограничена. В приложении А мы рассмотрим простую консоль, поддерживаемую средствами JavaScript, которая позволяет отображать сообщения о состоянии системы при отсутствии в браузере строки состояния.) Средства обработки событий почти идеально подходят для передачи серверу информации об обновлениях.

Предположим, что на сервере выполняется сценарий updateDomainModel.jsp, который получает от клиента следующую информацию.

- Уникальный идентификатор планеты, информация о которой подлежит обновлению.
- Имя обновляемого свойства.
- Значение, присвоенное свойству.

Мы можем написать обработчик событий, который передавал бы серверу требуемую информацию.

```
function  updateServer(propviewer)
{
  var   planetObj=propviewer.viewer.object;
  var  planetId=planetObj.id;
  var  propName=propviewer.name;
  var   val=propviewer.value;
  net.ContentLoader
  (
    'updateDomainModel.jsp',  someResponseHandler,
    null, 'POST', 'planetId='+encodeURI(planetId)
    +'&propertyName='+encodeURI(propName)
    +'&value='+encodeURI(val)
  );
}
```

Этот обработчик надо связать с ObjectViewer.

```
myObjectViewer.addChangeListener(updateServer)  ;
```

Написать подобный код несложно, но результатом его выполнения будет передача по сети множества мелких фрагментов данных, что снижает эффективность работы. Для того чтобы получит контроль над трафиком, нам надо перехватывать обновления, помещать их в локальную очередь, а затем, во время бездействия приложения, передавать их серверу большими пакетами. Простая очередь для хранения информации об обновлениях, реализованная средствами JavaScript, показана в листинге 5.13.

ЛИСТИНГ 5.13. Объект CommandQueue

```
//O Создать обьект очереди
net.CommandQueue=function(id,url,freq)
{
  this.id=id;
  net.cmdQueues[id]=this;
  this.url=url;
  this.queued=new Array();
  this.sent=new Array();
  if (freq)
  {
    this.repeat(freq);
  }
}
```

```
// 0 Передать запрос серверу
net.CommandQueue.prototype.addCommand=function(command){
  if (this.isCommand(command))
  {
    this.queue.append(command,true);
  }
}
net.CommandQueue.prototype.fireRequest=function(){
  if (this.queued.length==0)
  {
    return;
  }

  var data="data=";
  for(var i=0; i<this.queued.length;i++){
    var cmd=this.queuedfi];
    if (this.isCommand(cmd)){
      data+=cmd.toRequestString();
      this.sent[cmd.id]=cmd;
    }
  }

  this.queued=new Array0;
  this.loader=new net.ContentLoader(
    this.url, net.CommandQueue.onload,net.CommandQueue.onerro:
    "POST",data
  );
}

// © Проверить тип объекта
net.CommandQueue.prototype.isCommand=function(obj)
{
  return
   (
    obj.implementsPropC'id")
    && obj.implementsFunc("toRequestString")
    && obj.implementsFunc("parseResponse")
  );
}

// О Выполнить разбор ответа сервера
net.CommandQueue.onload=function(loader)
  {
  var xmlDoc=net.req.responseXML;
  var elDocRoot=xmlDoc.getElementsByTagName("commands")[0];
  if (elDocRoot)
  {
    for(i=0;KelDocRoot.childNodes.length;i++)
    {
      elChild=elDocRoot.childNodes[i];
      if (elChild.nodeName=="command")
```

```
        {
          var attrs=elChild.attributes;
          var id=attrs.getNamedItem("id").value;
          var command=net.commandQueue.sent[id];
          if (command)
          {
            command.parseResponse(elChild);
          }
        }
      }
    }
  }

net.CommandQueue.onerror=function(loader){
  alert("problem sending the data to the server"); }

// © Опрос сервера
net.CommandQueue.prototype.repeat=function(freq){

  this.unrepeat();
  if (freq>0){
    this.freq=freq;
    var cmd="net.cmdQueues["+this.id+"].fireRequest()";
    this.repeater=setInterval(cmd,freq*1000);
  }
}

// 0 Отключить опрос сервера
net.CommandQueue.prototype.unrepeat=function()
{
  if (this.repeater)
  {
    clearInterval(this.repeater);
  }
  this.repeater=null;
```

При инициализации О объекта CommandQueue (он назван так потому, что в очереди содержатся объекты Command) указывается уникальный идентификатор, URL сценария на стороне сервера и в качестве необязательного параметра — флаг, указывающий на необходимость повторного опроса. Алы тернативный вариант — активизация опроса вручную. Каждый из этих режимов может быть полезен, поэтому оба они предусмотрены в коде объекта. Когда очередь формирует запрос серверу, все содержащиеся в ней команды преобразуются в строки и передаются серверу ©.

В составе объекта содержатся два массива. Массив queued предполагает указание числовых индексов; в него помещаются новые обновления. Массив sent — ассоциативный. Он содержит те обновления, которые были отправлены серверу, но ответы на которые еще не были получены. В обоих массивах содержатся объекты Command, интерфейс которых определяется функцией isCommand() ©. Эти объекты обладают следующими свойствами.

- Объект может иметь уникальный идентификатор.
- Объект допускает сериализацию для включения в состав запроса POST ©.
- Объект может выполнять разбор ответа сервера О и определять, успешно ли был обработан запрос и должны ли предприниматься дальнейшие действия.

Для проверки выполнения условий используется функция implements-Func(). Поскольку данный метод принадлежит базовому классу Object, может показаться, что он является стандартным инструментом JavaScript, но на самом деле мы объявляем функцию implements Func О в составе вспомогательной библиотеки.

```
Obj ect.prototype.implementsFunc=function(funcName){
  return this[funcName] && this[funcName] instanceof
Function; }
```

Прототипы JavaScript подробно описаны в приложении Б. Теперь вернемся к объекту, реализующему очередь. Метод onload очереди О ожидает ответа сервера, содержащего XML-документ. В составе документа должны присутствовать дескрипторы <command>, помещенные в дескриптор <commands>.

Методы repeat () © и unrepeat () © используются для управления объектом таймера при периодическом опросе сервера.

Объект Command, предназначенный для обновления свойств объекта, описывающего планету, показан в листинге 5.14.

Листинг 5.14. Объект UpdatePropertyCommand

```
planets.commands.UpdatePropertyCommand=function(owner,field,value)
{
  this.id=this.owner.id+"_"+field;
  this.obj=owner;
  this.field-field;
  this.value=value;
}

planets.commands.UpdatePropertyCommand.toRequestString=
function()
{
  return
  {
    type:"updateProperty",
    id:this.id,
    planetId:this.owner.id,
    field:this.field,
    value:this.value
  }.simpleXmlify("command");
}
```

```
planets.commands.UpdatePropertyCommand.parseResponse=
function(docEl)
{
  var attrs=docEl.attributes;
  var status=attrs.getNamedItem("status").value;
  if (status!="ok")
  {
    var    reason=attrs.getNamedItem("message").value;
    alert("failed to update "
     +this.field+" to "+this.value
     +"\n\n"+reason);
  }
}
```

Данный объект предоставляет уникальный идентификатор команды и инкапсулирует параметры, необходимые серверу. Функция toRequeststring() оформляет сама себя в виде фрагмента XML-кода, используя специальную функцию, которую мы присоединили к прототипу Object.

```
Object.prototype.simpleXmlify=function(tagname){
  var xml="<"+tagname;
  for (i in this){
    if (!this[i] instanceof Function)
    {
      xml+=" "+i+"="\""+this[i]+"\"";
    }
  }
  xml+="/>";-
  return xml;
)
```

В результате создается простой XML-дескриптор (для удобства восприятия он отформатирован вручную).

```
<command type='update Property'
  id='001_diameter'
  planetId='mercury'
  field='diameter'
  value='3'
/>
```

Заметьте, что уникальный идентификатор формируется из идентификатора планеты и имени свойства. Мы не можем передать серверу несколько вариантов одного и того же значения. Если перед передачей содержимого очереди мы несколько раз отредактируем свойство, каждое последующее значение будет заменять предыдущее.

В составе запроса POST, передаваемого серверу, может содержаться один или несколько описанных выше дескрипторов, в зависимости от частоты опроса и активности пользователя. Сервер обрабатывает каждую команду и сохраняет результаты, формируя ответ. Обработчик onload, принадлежащий CommandQueue, распознает дескрипторы в ответе, сравнивает их с объектами Command в массиве sent и вызывает метод parseResponse () объекта Command. Ответ может выглядеть следующим образом:

```
<commands>
  <command id='001_diameter' status='ok'/>
  <command id='003_albedo'
    status='failed' message='value out of range'/>
  <command id='004_hairColor'
    status='failed' message='invalid property name'/>
</commands>
```

Как видно из сообщения, информация о диаметре Меркурия была обновлена, но другие два обновления отклонены. Причины описаны посредством атрибута message. Пользователь оповещается о возникающих проблемах (обычно для этого используется функция `alert()`) и может предпринять ответные действия.

Компонент, обрабатывающий запросы на стороне сервера, должен выделять из данных запроса отдельные команды и передавать каждую команду соответствующему объекту обработчика. По мере обработки команд результаты записываются в HTTP-ответ. Код простого Java-сервлета, предназначенного для решения описанной задачи, представлен в листинге 5.15.

Листинг 5.15. Содержимое файла CommandServlet .Java

```
public class CommandServlet extends HttpServlet
{
  private Map commandTypes=null;
  public void init() throws ServletException
{
    ServletConfig config=getServletConfig();
    // О Конфигурация обработчиков
    commandTypes=new HashMap();
    boolean more=true;
    for(int counter=l;more;counter++){
      String typeName=config.getlnitParameter("type"+counter);
      String typelmpl=config.getlnitParameter("impl"+counter);

      if (typeName==null || typelmpl==null){
        more=false;
      }
      else
      {
        try{
          Class cls=Class.forName(typelmpl);
          commandTypes.put(typeName,els);
        }
        catch (ClassNotFoundException clanfex)
        {
          this.log(
            "couldn't resolve handler class name"+typelmpl);
        }
      }
    }
  }
```

```
protected void doPost
(
  HttpServletRequest req,
  HttpServletResponse resp
)

throws IOException{

  // 0 Обработать запрос
  resp.setContentType("text/xml");
  Reader reader=req.getReader();
  Writer writer=resp.getWriter();

  try{ SAXBuilder builder=new SAXBuilder(false);

    // © Обработать XML-данные
    Document doc=builder.build(reader);
    Element root=doc.getRootElement();

    if  ("commands".equals(root.getName())){
      for(Iterator iter=root.getChildren("command").iterator();
      iter.hasNext();)

      {
        Element el=(Element)(iter.next());
        String type=el.getAttributeValue("type");
        XMLCommandProcessor command=getCommand(type,writer);
        if  (command!=null)
        {
          // 0 Делегировать обработку
          Element result=command.processXML(el);
          writer.write(result.toString());
        }
      }
    }
    else
    {
      sendError(writer,
        "incorrect document format - "
        +"expected top-level command tag");
    }
  }
  catch  (JDOMException jdomex){
    sendError(writer,"unable to parse request document");
  }
}

private XMLCommandProcessor getCommand
(String type,Writer writer)
throws IOException{

  // © Соответствие обработчика команде
  XMLCommandProcessor cmd=null;
  Class cls=(Class)(commandTypes.get(type));
  if  (cls!=null){
```

```
        try{ cmd=(XMLCommandProcessor)(els.newlnstance());
        }catch  (ClassCastException  castex){
           sendError(writer,
             "class  "+cls.getName()
             +" is not a command");
        }
        catch  (InstantiationException  instex)
{
           sendError(writer,
             "not able to create class  "+cls.getName());
        }  catch  (IllegalAccessException illex)
{
           sendError(writer,
            "not allowed to create class  "+cls.getName());
        }
       }else{
         sendError(writer,"no command type registered for  "+type);

       }
       return  cmd;
     }
     private void sendError
      (Writer writer,String message) throws IOException{
       writer.write("<error msg='"+message+"'/>");
       writer.flush();
     }
_}
```

Данный сервлет поддерживает карту объектов XMLCommandProcessor, для конфигурирования которой используется интерфейс ServletConfig О. При использовании более мощных базовых средств можно воспользоваться для этой цели конфигурационным XML-файлом. При обработке запроса POST © для разбора XML-данных используется JDOM ©, а затем осуществляется перебор дескрипторов <coramand>, для которых атрибуты соответствуют объекту обработчика XMLCommandProcessor О. В карте содержатся определения классов, на основе которых в методе getCommand () мы создаем конкретные экземпляры, используя механизм отражения ©.

В интерфейсе XMLCommandProcessor объявлен единственный метод.

```
public interface XMLCommandProcessor {
   Element processXML(Element e1); }
```

В данном интерфейсе предполагается, что для представления XML-Данных будут применяться библиотеки JDOM. Объектами Element являются как параметры, так и возвращаемое значение. Простой класс, реализующий Данный интерфейс и предназначенный для обновления данных о планетах, показан в листинге 5.16.

Листинг 5.16. Содержимое файла PlanetUpdateCoimandProcessor. Java

```
public class PlanetUpdateCommandProcessor
  implements XMLCommandProcessor
  {
  public Element processXML(Element el)
  {
    // О Создать XML-узел результатов
    Element result=new Element("command");
    String id-el.getAttributeValue("id");
    result.setAttribute("id",  id);
    String status=null;
    String reason=null;
    String planetld=el.getAttributeValue("planetld");
    String field=el.getAttributeValue("field");
    String value=el.getAttributeValue("value");

    // @ Обращение к модели предметной области
    Planet planet=findPlanet(planetld);

    if (planet==null){
      status-"failed";
      reason="no planet found for id "+planetld;
    }else{
      Double numValue=new Double(value);
      Object[] args=new Object[]{ numValue
};
      String method = "set"+field.substring(O,1).toUpperCase()
        +field.substring(1);
      Statement statement=new Statement(planet,method,args);
      try {

        // © Обновить модель предметной области
        statement.execute();
        status="ok";
      } catch (Exception e) {
        status="failed";
        reason="unable to set value "+value+" for field "+field;
      }
    }
    result.setAttribute("status",status);
    if (reason!=null){
      result.setAttribute("reason",reason);
    }
    return result;
  }
  private Planet findPlanet(String planetld)
{
    // О Использовать ORM для модели предметной области
    return null;
  }

}
```

JDOM используется не только для разбора XML-данных, содержащихся в запросе, но и для генерации ответа. Корневой узел О и дочерние узлы создаются в методе processXML(). Для доступа к модели предметной области на стороне сервера используется метод f indPlanet () ©; ему передается идентификатор объекта. Реализация метода f indPlanet () здесь не рассматривается; заметим лишь, что в нем реализуется обмен с базой данных традиционными способами О. Для обновления модели предметной области © применяется механизм отражения.

Так в общих чертах выглядит архитектура, основанная на использовании очереди и используемая для объединения частных действий по обновлению модели предметной области в одну HTTP-транзакцию. Она позволяет не только синхронизировать модели на стороне клиента и на стороне сервера, но и эффективно управлять трафиком. Как было сказано в разделе 5.3, такое решение может быть использовано совместно с JSF и другими подобными средствами, когда структура пользовательского интерфейса и модель взаимодействия существенно зависят от сервера. В данном случае мы лишь реализовали эффективный способ обновления модели предметной области на разных уровнях.

На этом мы заканчиваем разговор о средствах обеспечения взаимодействия клиента и сервера и обзор основных элементов, определяющих структуру Aj ax-приложений. По сути, мы заложили основы языка запросов к серверу в рамках Ajax-приложения и получили представление о возможностях, доступных для его реализации.

5.6. Резюме

Мы начали данную главу с рассмотрения основных функций сервера в составе Ajax-приложения, а именно с доставки клиентского кода браузеру и предоставления клиенту данных в процессе работы. Рассматривая эти вопросы, мы учли наиболее популярные языки и инструменты, используемые для создания серверных программ. Они ориентированы на классические Web-приложения, и мы выяснили, насколько они подходят для Ajax. Инструменты, предназначенные для создания серверных программ, очень быстро изменяются, поэтому, вместо того, чтобы обсуждать конкретные продукты, мы разделили их на категории в зависимости от используемой архитектуры. В результате было выделено три основных направления: архитектура Model2, средства на основе компонентов и архитектура, ориентированная на использование служб. На сегодняшний день есть основания считать, что лучше всего для Ajax подходит SOA, однако попытки адаптировать другие архитектуры также могут Увенчаться успехом.

Далее мы выделили три основных подхода к решению задачи доставки данных от сервера клиенту: ориентированный на содержимое, ориентированный на сценарий и ориентированный на данные. Если классическое Web-приложение больше всего соответствует архитектуре, ориентированной на содержимое, то приложение Ajax больше укладывается в схему, ориентированную на данные. Взаимодействие, ориентированное на сценарии, занимает промежуточную позицию. Обсуждая архитектуру, ориентированную на данные, мы выяснили, что XML — не единственно возможный формат представления данных. В некоторых случаях при доставке информации клиенту может успешно применяться формат JSON.

И наконец, мы описали способы передачи данных на сервер, используя для этой цели HTML-формы и объект XMLHttpRequest. Мы также обсуждали возможности управления передачей данных путем создания на стороне клиента очереди объектов Command. Данный подход обеспечивает существенное повышение производительности, снижая как нагрузку на сервер, так и сетевой трафик. Переходя к взаимодействию, ориентированному на данные, мы отказываемся от действий, основанных на удаленном вызове процедур, и переходим к работе с документом.

Этой главой мы завершаем разговор о базовых технологиях Ajax. Мы обсудили все основные вопросы и даже затронули более сложные темы. В последующих трех главах мы вернемся к проблеме применимости программ и выясним, чем же отличаются поделки, пусть даже талантливые, от настоящих продуктов, в которых нуждаются пользователи.

5.7. Ресурсы

В данной главе были упомянуты некоторые инструменты разработки. Ниже приведены их URL.

Struts (http://struts.apache.org).

Tapestry (http://jakarta.apache.org/tapestry/).

JSF (http://java.sun.com/j2ee/javaserverfaces/faq.html).

PHP-MVC (http://www.phpmvc.net).

По данным исследований (http://wicket.sourceforge.net/Introduction.html), в настоящее время только для Java существует более 60 наборов базовых средств.

JSF — это категория инструментов, в которую входят различные продукты. Кито Манн (Kito Mann), автор книги *JavaServer Faces in Action* (Manning, 2004), поддерживает Web-портал, посвященный JSF (http://www.jsfcentral.com/). Материалы, подготовленные Грегом Мюрреем (Greg Murray) и его коллегами в ходе обсуждения тем Ajax и JSF, доступны по адресу https://bpcatalog.dev.java.net/nonav/ajax/jsf-ajax/frames.html. AjaxFaces представляет собой реализацию JSF для Ajax (http://www.ajaxfaces.com); этот продукт распространяется на коммерческой основе. Готовится модификация Apache Open Source MyFaces (http://myfaces.apache.org/sandbox/inputSuggestAjax.html) в соответствии с требованиями Ajax.

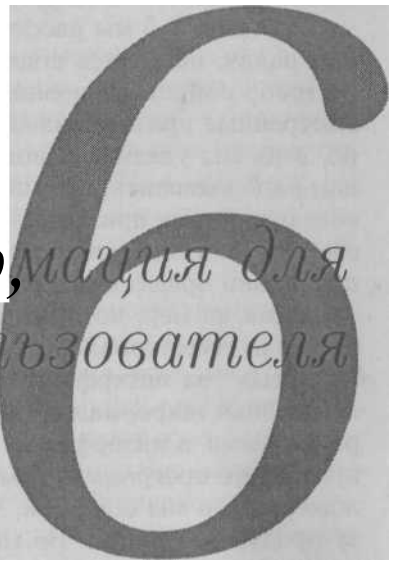
На момент написания данной книги Microsoft Atlas находился в процессе разработки, но первые реализации скоро должны быть доступны. Менеджер проекта Atlas, Скотт Гатри (Scott Guthrie), публикует материалы, имеющие отношение к Ajax (http://weblogs.asp.net/scottgu/archive/2005/06/28/416185.aspx).

Библиотеки JSON-RPC, ориентированные на различные языки программирования, находятся по адресу http://www.json-rpc.org/impl.xhtml.

Часть III

Создание профессиональных Ajax-приАожений

Теперь ваше Ajax-приложение может взаимодействовать с сервером, получать от него информацию и обновлять хранящиеся на нем данные. Однако, если вы хотите, чтобы ваш продукт был по достоинству оценен пользователями, вам рано останавливаться. В этой части вы узнаете, как добиться того, чтобы приложение было простым в использовании, защищенным и работало достаточно быстро.



Информация для пользователя

В этой главе...

- Основные характеристики практичного кода
- Принципы работы системы оповещения
- Набор базовых средств для оповещения
- Выделение измененных данных

В главе 1 мы обсуждали практичность — одно из основных качеств прикладной программы. Независимо от того, насколько грамотно написан код какие применены технические решения, если приложение недостаточно практично, у пользователя останется плохое впечатление от него. Может быть программистам это и покажется несправедливым, но положение вещей именно таково. Глядя на фотографию Альберта Эйнштейна, обыватель вряд ли поймет, что этот человек внес огромный вклад в формирование научной картины мира, но отметит его неряшливый внешний вид и всклокоченные волосы. Первое впечатление о человеке или предмете очень важно для формирования отношения к нему.

В главах 2-5 мы рассмотрели важные технологии и решили ряд интересных задач, пользуясь подходом, специфическим для Ajax. Дальнейший нащ разговор пойдет в основном о том, как оформить их. Надо заметить, что рассмотренные примеры внешне выглядят достаточно грубо. Это не удивительно, ведь мы уделяли основное внимание изяществу самого решения. Теперь нам надо выяснить, что мы можем сделать, чтобы пользователю понравилось созданное нами приложение и он согласился работать с ним по нескольку часов в день. Темы, рассмотренные в данной главе, призваны помочь читателю правильно представить Ajax-приложение внешнему миру.

Одна из мер, которые необходимо принять для того, чтобы ваши пользователи чувствовали себя комфортно, — информировать их о событиях, "скрытых" за интерфейсом. Более того, сделать это надо так, чтобы представленная информация органично сочеталась с остальными данными, отображаемыми в интерфейсе. Сведения об основных процессах, происходящих при работе программы, сами по себе еще не обеспечивают практичности приложения, но мы покажем, что подробная проработка деталей идет на пользу продукту в целом. Во многих Ajax-приложениях принимаются меры для оповещения пользователей, поэтому мы надеемся, что выработанные здесь решения вы сможете применить в своих проектах.

В данной главе мы рассмотрим несколько примеров, в которых пользователь информируется о возникающих событиях таким образом, что нормальный ход его работы не нарушается. Однако, перед тем как начать изучение данного вопроса, попробуем выяснить, что такое качественное приложение и как сделать его таковым.

6.1. Создание качественного приложения

Практичность — важнейшая характеристика Ajax-приложений, поскольку их пользователи представляют собой чрезвычайно непостоянную аудиторию. Возможность быстро и просто скопировать и запустить приложение — это не всегда благо. Пользователь, не затративший времени и усилий для того, чтобы начать работу с документом, может легко отказаться от него и перейти к другой Web-странице, каковых в глобальной сети насчитывается около восьми миллиардов. Ситуацию усложняет тот факт, что в настоящее время наблюдается сближение двух подходов к созданию пользовательского интерфейса: приложения для настольных систем и Web-страницы внешне ста-

ловятся похожи друг на друга. Создать в сложившихся условиях хороший интерфейс — непростая задача. Если вы не сможете ее решить, результаты ваших усилий по разработке бизнес-логики окажутся невостребованными.

В главе 1 мы рассматривали вопросы практичности с точки зрения пользователя. Теперь попытаемся выяснить, какими качествами должен обладать код, чтобы удовлетворять требованиям практичности. В следующих разделах будут описаны основные характеристики, ответственные за качество приложения.

6.1.1. Отклик программы

Одно из явлений, которые мешают использованию компьютера, — это прерывание нормального хода работы во время выполнения некоторых операций. Неопытные разработчики часто допускают типичную ошибку — блокируют пользовательский интерфейс на время записи на диск больших конфигурационных файлов. В результате пользователь "теряет нить", т.е. забывает, что он собирался сделать, так как ему необходимо внутренне перестроиться и перейти от модели предметной области, с которой он работал, к реалиям аппаратного обеспечения компьютера.

Планируя время отклика, важно представлять себе типичную аудиторию вашей системы. Если запись конфигурационного файла на локальный жесткий диск выполняется практически мгновенно, то запись того же файла на удаленный носитель в условиях большой загрузки локальной сети произойдет значительно медленнее, запись во флэш-память, подключенную черед порт USB, имеет свои особенности, и т.д. Разработчики Web-приложений при тестировании своих продуктов часто организуют взаимодействие клиента и сервера через интерфейс обратной петли; Это ошибка, так как становится невозможно оценить задержку, связанную с передачей данных. Поэтому все Web-приложения должны тестироваться в реальной сети или хотя бы в системе, имитирующей реальное прохождение трафика по линиям связи.

Помимо передачи данных по сети, на время отклика влияет также производительность клиентского кода. Производительность — чрезвычайно важная характеристика приложения, и мы уделим ей внимание в главе 8.

6.1.2. Надежность

Приложение считается надежным, если оно способно сохранять работоспособность в условиях рабочей станции, перегруженной другими задачами. Как отреагирует программа на временный выход из строя сети? Если некий процесс в течение пяти минут полностью займет ресурсы центрального процессора, возобновит ли после этого работу приложение? Недавно в рамках проекта, в котором участвовал один из авторов книги, проверялась надежность приложения. Одним из тестов было беспорядочное нажатие клавиш клавиатуры в течение десяти секунд. Другой тест предполагал перемещение курсора мыши по странице с периодическими щелчками правой и левой кнопками. Несмотря на кажущуюся примитивность данных тестов, они очень эффективны.

Какие недостатки станут видны в результате подобного тестирования? При этом может быть выявлена низкая эффективность обработчиков событий. Нажатие клавиш, перемещение курсора мыши и другие подобные действия требуют быстрой обработки, поскольку такие события могут повторяться часто. Они также помогают обнаружить нежелательную взаимную зависимость между компонентами. Например, модальное диалоговое окно в составе пользовательского интерфейса может блокировать доступ к остальным элементам приложения, а открытый пункт меню не позволит обратиться к модальному окну. Если подобная ситуация возникает достаточно редко, то, для того, чтобы обнаружить ее, могут потребоваться месяцы ежедневной работы пользователя. Если же приложение становится доступным тысячам пользователей, то хотя бы один из них столкнется с упомянутым эффектом уже через несколько часов. Описать подобную ошибку достаточно трудно, а воспроизвести ситуацию по описанию — еще сложнее. Вовремя выявляя проблемы и исправляя ошибки, вы увеличите общую надежность приложения. Это будет способствовать повышению надежности гораздо больше, чем выбивание дроби на клавиатуре.

Очень полезно, если в процесс тестирования приложения, помимо разработчика, будут вовлечены потенциальные пользователи. Новичок может дать полезную информацию о практичности приложения, а пользователь, хорошо знакомый с предметной областью, может оценить качество выполнения конкретных функций. Если тестированием занимается разработчик программы, он "видит" код и подсознательно избегает выполнять некоторые сочетания действий в определенном контексте. Конечный пользователь не обременен такими знаниями и свободен в выборе, если, конечно, разработчик не подсказывает ему, что надо сделать. Привлечение посторонних к проверке приложения ускоряет выявление ошибок и способствует повышению надежности.

6.1.3. Согласованность

Как уже упоминалось выше, на практичность Ajax-приложений существенное влияние оказывает тенденция к сближению программ интерфейсов для настольных систем и Web-страниц. Некоторые инструменты, ориентированные на Ajax, например Bindows, qooxdoo и Backbase, даже предлагают наборы компонентов, которые выглядят подобно кнопкам, деревьям, таблицам и другим элементам пользовательского интерфейса, типичным для настольных систем.

Теперь, чтобы создать качественное приложение, надо одновременно быть разработчиком Web-страниц, специалистом по вопросам практичности и конечным пользователем. Наилучший совет в данной ситуации — обеспечить согласованность различных частей приложения. Если в одной части программы для отображения окна с дополнительной информацией требуется один щелчок мышью, а в другой части такое же окно открывается двойным щелчком, пользователь будет испытывать неоправданные затруднения в процессе работы. Если же руководство по выбору пути в пределах Web-узла выполне-

до вами в виде говорящего поросенка, следите за тем, чтобы он внезапно не сменил одежду или не изменил произношение.

С точки зрения кода приложения согласованность и пригодность к повторному использованию неотделимы друг от друга. Если вы создадите в четырех местах программы кнопки, в ответ на активизацию которых будут выполняться стереотипные действия, а впоследствии по требованию заказчика измените три кнопки, но пропустите четвертую, то это будет первый шаг к рассогласованию компонентов. Если для обработки действий с различными кнопками вы используете один обработчик, это будет способствовать согласованности приложения. Такое решение повлияет не только на внешний вид и поведение кнопок, но и на другие особенности интерфейса, например, время тайм-аута или реакцию на неверно введенные данные.

6.14. *Простота*

Простота — важное качество приложения. Ajax позволяет разработчику дать волю фантазии и воплотить совершенно новые решения. Многие из них были невозможны потому, что отсутствовали необходимые для этого технологии. Однако в ряде случаев, прежде чем программировать то или иное средство, надо спросить себя, а необходимо ли оно в приложении. Меню, которые скачут по экрану, постепенно уменьшая амплитуду скачков, наверное, выглядят забавно. Возможно, программисту приятно было отвлечься на время от рутинной работы и сделать полушутливый интерфейс. Однако такое решение вряд ли будет одобрено пользователем, который должен работать с приложением по нескольку часов в день.

Итак, прежде чем реализовывать новый элемент или изменять поведение существующих компонентов, необходимо тщательно продумать, способствует ли подобное решение продуктивной работе пользователя. Во многих случаях при использовании Ajax ответ на заданный вопрос будет положительным, и разработчик сконцентрирует свое внимание на разработке тех средств, которые действительно улучшают качество приложения.

6.1.5. Как получить результат

Вряд ли ваше приложение в полной мере удовлетворяет всем перечисленным выше требованиям. Нам, во всяком случае, пока не удавалось создавать такие программы. Сказанное ранее можно рассматривать как характеристики идеального приложения. Стремление к идеалу — прекрасное качество. Его надо постоянно развивать, применяя грамотные подходы при разработке новых программ и реструктуризируя уже существующий код. Правильно выбрать точку приложения усилий — это искусство, граничащее с магией, и преуспеть в нем можно, только проверяя свои знания на практике. Если вы раньше не занимались реструктуризацией, начните с небольших программ и постоянно работайте над ними. Помните, что реструктуризация предполагает поэтапное улучшение кода, поэтому не стоит отвергать уже готовое решение и переделывать программу с нуля.

Далее в этой главе будет рассмотрен ряд средств, которые можно реализовать в рамках Ajax-приложения. Мы уделим много внимания оповещению пользователя о процессах, скрытых от него, например, о вычислениях или обмене по сети. Предоставляя пользователю информацию о ходе работы программы, мы повышаем качество взаимодействия с приложением. Реализуя оповещение посредством общего базового набора средств, мы обеспечиваем согласованность и упрощаем работу пользователя. Если все сообщения оформлены по одному и тому же принципу, пользователь лучше воспринимает их.

Рассмотрим различные способы оповещения пользователя о событиях, возникающих во время работы программы.

6.2. Предоставление сведений пользователю

При работе Ajax-приложения очень часто возникает необходимость обращения по сети и загрузки ресурсов с сервера. В получении результатов обычно участвуют функции обратного вызова. При синхронной обработке запросов к серверу процесс взаимодействия по сети так или иначе находит отражение в пользовательском интерфейсе. Инициализация запроса приводит к тому, что элементы интерфейса блокируются и в течение некоторого времени не реагируют на действия пользователя. После получения результатов содержимое окна клиентской программы обновляется и пользователь снова получает возможность взаимодействовать с приложением. Подобное решение реализуется очень просто, однако пользоваться такой программой неудобно. По этой причине мы будем использовать асинхронные запросы, однако при этом процесс обновления данных существенно усложняется.

6.2.1. Поддержка ответов на собственные запросы

Рассмотрим конкретный пример. В главе 5 мы создали приложение для просмотра информации о планетах. Оно позволяло пользователям обновлять некоторые свойства, в частности, задавать диаметр планеты и ее расстояние от Солнца (см. раздел 5.5). Информация, введенная пользователем, передавалась на сервер, а в ответе сервера содержались сведения о том, приняты или отвергнуты изменения. В разделе 5.5.3 было введено понятие очереди команд; следуя этому принципу, мы рассматривали каждый ответ сервера как оповещение о некоторых обновлениях, выполненных по инициативе конкретного пользователя. Ответ представлял собой XML-документ, содержащий информацию об успешно выполненных или отвергнутых командах.

```
<commands>
  <command id='001_diameter' status='ok'/>
  <command id='003_albedo' status»'failed'
  message='value out of range'/>
</commands>
```

Пользователь, работающий с приложением, редактирует свойство и принимается за другую задачу, может быть, даже переходит к другому окну. На-

пример, обновив сведения о диаметре планеты Меркурий, он больше не уделяет внимание этому вопросу. В это время значение для обновления оформляется в фоновом режиме в виде JavaScript-объекта Command, который помещается в очередь исходящих сообщений, а затем передается серверу. После этого объект Command перемещается в массив sent и извлекается оттуда при получении ответа. В составе ответа может содержаться информация сразу о нескольких обновлениях. Объект Command отвечает за обработку обновления и выполнение необходимых действий.

Давайте вспомним, какой вид имел код приложения перед началом реструктуризации. Ниже показан код метода parseResponse() объекта Command. Именно так мы сформировали его в главе 5.

```
planets.commands.UpdatePropertyCommand
  .parseResponse=function(docEl){
 var attrs=docEl.attributes;
 var status=attrs.getNamedItem("status").value;
 if (status!="ok"){
 var reason=attrs.getNamedItem("message").value;
 alert("failed to update "+this.field
 +" to "+this.value+"\n\n"+reason);
 }
}
```

Начнем его реструктуризацию. Если обновление было осуществлено успешно, ничего не происходит. Локальная модель предметной области уже была обновлена перед передачей информации на сервер, поэтому модели на стороне клиента и на стороне сервера синхронизированы. В случае неудачи мы генерируем сообщение. Такое сообщение легко сформировать, но, как мы увидим, оно плохо вписывается в концепцию практичной программы.

Давайте вернемся к пользователю, который уже успел забыть о планете Меркурий. Неожиданно он получает сообщение наподобие следующего: Failed to update albedo to 180 value out of range. Вне контекста такое сообщение совершенно неинформативно. Мы можем изменить текст, чтобы он выглядело приблизительно так: Failed to update albedo of Mercury..., но при этом действия пользователя все равно прерываются, а именно этого мы и хотели избежать, переходя к асинхронной обработке сообщений.

В данном случае может возникнуть и более серьезная проблема. При реализации полей редактирования передача данных серверу инициируется по событию onblur. Метод onblur() получает управление тогда, когда поле редактирования теряет фокус ввода, причем потеря фокуса может быть вызвана появлением окна с сообщением. Таким образом, если пользователь отредактировал свойство и начал вводить значение другого, ввод может прерваться на середине. В результате неполные данные будут переданы на сервер, а это приведет к генерации следующей ошибки, или, что еще хуже, модель может быть обновлена на основе некорректной информации.

Необходимо более элегантное решение, чем обычное окно с сообщением. Но, прежде чем заняться его разработкой, завершим разговор об обновлении Модели. Рассмотрим ситуацию, когда несколько пользователей одновременно обновляют модель предметной области.

12.2. Обработка обновлений, выполненных другими пользователями

'усматриваемое приложение допускает одновременную работу различных юльзователей, поэтому редактирование данных может осуществляться сра у с нескольких клиентских машин. Каждый пользователь хотел бы иметь перативную информацию об изменениях, внесенных другими участниками. 1одобные требования типичны для большинства приложений, предполагающих взаимодействие браузера с Web-сервером.

Для того чтобы учесть данное требование, мы должны модифицировать [ML-ответ и очередь объектов Command следующим образом. Для каждого запроса, предполагающего обновление модели предметной области на стороне ервера, должна генерироваться метка с информацией о времени (временная [етка). Процесс, выполняющийся на стороне сервера, который учитывает бновления, должен проверять модель, выявлять недавние изменения, внеенные другими пользователями, и включать их в возвращаемый документ. Ъперь полученный ответ будет выглядеть приблизительно так, как показао ниже.

```
• «responses updateTime='1120512761877'>
 <command id='001_diameter' status='ok'/>
 <command id='003_albedo' status='failed'
    message='value out of range'/>
 <update planetId='OO2' fieldName='distance'
    value='0.76' user='jim'/>
</responses>
```

Помимо дескрипторов <command>, содержащих идентификаторы объектов ommand, в составе ответа присутствует также дескриптор <update>, который данном случае определяет расстояние от Венеры до Солнца. Значение этого войства, равное 0,76, задал пользователь Jim. Кроме того, мы можем добаить к дескриптору верхнего уровня атрибуты, состав и назначение которых ы обсудим несколько позже.

Ранее запрос передавался серверу только в том случае, если в очереди рисутствовали команды. Теперь нам надо получать сведения об обновлении, следовательно, придется запрашивать сервер даже тогда, когда очередь пуга. Для того чтобы это стало возможным, нам надо изменить код в нескольих местах. Модифицированный объект CommandQueue показан в листинге 6.1. [зменения выделены полужирным шрифтом.

ЛИСТИНГ 6.1. Объект CommandQueue

```
// О Глобальная ссылка
 net.cmdQueues-new Array();
// © Дополнительные параметры функции
net.CommandQueue=function(id,url,onUpdate,freq){
 this.id=id;

 net.cmdQueues[id]-this;
 this.url=url;
 this.queued=new Array();
 this.sent=new Array();
```

```
 this.onUpdate-onOpdate;
 // 0 Инициализировать
повторные обращения
 if (freq){
 this.repeat(freq);
 }
 this.lastUpdateTime»O;
}
net.CommandQueue.prototype.fireRequest=function(){
 if (!this.onUpdate && this.queued.length==0){
 return;
 }
 // Временная метка
 var data=
 "lastUpdate«"+this.lastUpdateTime
+"&data=";
 for(var i=0;i<this.queued.length;i++){
 var cmd=this.queued[i];
 if (this.isCommand(cmd)){
 data+=cmd.toRequestString();
 this.sent[cmd.id]=cmd;
 }
 }
 this.queued=new ArrayO;
 this.loader=new net.ContentLoader(
 this.url,
 net.CommandQueue.onload,net.CommandQueue.onerror,
 "POST",data
 );
}

net.CommandQueue.onload=function(loader)   {
 var xmlDoc=net.req.responseXML;
 var elDocRoot=xmlDoc.getElementsByTagName("responses")[0];
 var lastUpdate=elDocRoot.attributes.getNamedItem("updateTime") ;
 if (parseInt(lastUpdate)>this.lastUpdateTime){
 // O Обновленная временная метка
 this.lastUpdateTime=lastUpdate;
 }
 if (elDocRoot){
 for(i=0; KelDocRoot.childNodes.length;i++){
 elChild=elDocRoot.childNodes[i];
 if (elChild.nodeName«-"command"){

 var attrs=elChild.attributes;
 var id=attrs.getNamedItem("id").value;
 var command=net.CommandQueue.sent[id] ;
 if (command){
 command.parseResponse(elChild);
 }
 }else if (elChild.nodeName=="update"){
 if (this.implementsFunc("onUpdate")){
 // © Обновленный обработчик
 this.onUpdate.call(this,elChild);
 }
 }
```

```
 )
 }
}
// 0 Опрос сервера
net.CommandQueue.prototype.repeat=function(freq){
 this.unrepeat();
 if (freq>0){
 this.freq-freq;
 var cmd="net.cmdQueues["+this.id+"].fireRequest{)" ;
 this.repeater=setlnterval(cmd,freq*1000);
 }
}
// О Включение/отключение повторных обращений
net.CommandQueue.prototype.unrepeat=function(){
 if (this.repeater){
 clearInterval(this.repeater);
 }
 this.repeater=null;
```

—1 •

Рассмотрим новые функции, которые мы реализовали, изменив код.

Во-первых, введена глобальная ссылка на очередь объектов Command О. С необходимостью этого шага приходится смириться, учитывая ограничения метода setlnterval(), которые мы рассмотрим несколько позже. Конструктору CommandQueue передается в качестве параметра уникальный идентификатор, и объект регистрируется в элементе массива, соответствующем идентификатору.

Теперь при вызове конструктора CommandQueue указываются два новых параметра ®. В качестве параметра onUpdate передается объект Function, используемый для поддержки дескрипторов <update>, который теперь может присутствовать в составе ответа. Параметр freq задает числовое значение, определяющее интервал в секундах между последовательными обращениями к серверу за информацией об обновлении. Если значение не равно нулю, конструктор инициализирует обращения к функции repeat() ©, в которой для организации повторного выполнения кода использован встроенный метод JavaScript setlnterval. Метод setlnterval() и метод setTimeout, предоставляющий аналогичные возможности, допускают параметры, заданные лишь в виде строк, поэтому нельзя непосредственно передавать ссылки на код, подлежащий выполнению. Чтобы обойти эту проблему, мы используем в теле функции repeat() глобальную переменную и уникальный идентификатор. Мы также сохраняем ссылку на временной интервал и можем в любой момент прекратить периодический опрос сервера, вызвав функцию unrepeat() в, в теле которой осуществляется обращение к clearlnterval.

Ранее, если очередь команд оказывалась пустой, метод f ireRequest О завершал работу. Теперь проверка выполняется несколько по-другому. При установленном обработчике onUpdate работа функции продолжается, независимо от того, есть ли в очереди элементы. Если очередь пуста, то серверу передается пустой запрос, на который сервер передает ответ, содержащий дескрипторы <update>. Вместе с результатами редактирования мы теперь по-

сылаем значение времени, сообщая серверу о моменте внесения изменений ©. Эту информацию сервер использует для того, чтобы правильно сформировать передаваемую информацию об обновлении. Значение времени хранится в виде свойства очереди команд и первоначально устанавливается равным нулю.

Информацию о времени мы передаем в формате, принятом в системе Unix для хранения даты, т.е. сообщаем число миллисекунд, прошедших с 1 января 1970 года. Такой выбор продиктован соображениями переносимости. Если бы мы выбрали формат более удобный для чтения, нам пришлось бы уделять внимание интернационализации и обеспечению адекватного представления на других платформах. Вопросы интернационализации имеют большое значение для Aj ax-приложений, так как для большинства из них планируется доступ по глобальной сети.

В функции onload() мы добавили код, необходимый для обновления сохраненной информации о времени О и для разбора дескрипторов <update> 0. Функция обработчика onUpdate вызывается в контексте очереди команд, а элемент DOM, соответствующей дескриптору <update>, передается ей в качестве параметра.

Для приложения, содержащего модель Солнечной системы, функция, которая выполняет роль обработчика обновления, представлена в листинге 6.2.

Листинг 6.2. Функция updatePlanets() Д Ц :

```
function  updatePlanets(updateTag){
 var  attribs=updateTag.attributes;
 var   planetld=attribs.getNamedltemC'planetld").value;
 var  planet=solarSystem.planets[planetld];
 if  (planet){
 var   fld=attribs.getNamedItem("fieldName").value;
 var   val=attribs.getNamedItem("value").value;
 if  (planet.fid){
 planet[fId]=val;
 }else{
 alert('unknown planet attribute  *+fld);
 }
 }else{
 alert('unknown planet  id  '+planetld);
 }
}
```

Атрибуты дескриптора <update> предоставляют нам всю информацию, необходимую для обновления модели предметной области на уровне JavaScript. Конечно, данные, полученные с сервера, могут быть некорректными, поэтому нам надо предусмотреть на этот случай соответствующие действия. Например, можно вывести сообщение с описанием проблемы, формируемое с помощью функции alert(). Этот вопрос обсуждался в разделе 6.2.1.

Теперь мы реализовали более сложное поведение очереди объектов Command, в частности, предусмотрели поддержку обновлений, выполненных другими пользователями, включили в состав данных, которыми обмениваются клиент и сервер, информацию о времени и предусмотрели включаемый об-

работник обновления. Итак, мы несколько продвинулись по пути решения вопроса об асинхронном обновлении модели предметной области и информировании пользователя о внесенных изменениях. В следующем разделе мы рассмотрим вопросы представления информации и влияние ее на нормальный ход работы пользователя. Там же мы постараемся отказаться от функции `alert()` в пользу более подходящего решения.

6.3. Создание системы оповещения

Функция `alert`, которую мы применяли до сих пор, позволяет реализовать лишь примитивное решение, типичное для ранних периодов развития JavaScript, когда Web-страницы были в основном статическими, а фоновая активность — минимальной. Внешний вид окна с сообщением не поддается контролю средствами CSS, поэтому, работая над профессиональным продуктом, желательно разработать механизм оповещения, используя технологии, применяемые для остальной части пользовательского интерфейса Ajax. Тем самым обеспечивается дополнительная степень гибкости приложения.

Если вы проанализируете существующие вычислительные системы, то увидите, что в них используются самые различные средства оповещения, различающиеся и по степени воздействия на пользователя. Менее всего привлекают внимание такие эффекты, как изменение вида курсора мыши (классический пример песочных часов в системе Windows) или добавление к пиктограмме, представляющей объект, дополнительных элементов, которые позволяют судить о состоянии папки. Такие простые индикаторы представляют минимум информации. Более детальные сведения о фоновых событиях можно получить, прочитав сообщение в строке состояния. И наконец, самым информативным является диалоговое окно. На рис. 6.1 показан пример использования соглашений об оповещении, принятых для окружения рабочего стола в системе Unix.

В данном случае папка `lost+found` не доступна текущему пользователю, поэтому к ней добавлено изображение замка. Строка состояния в нижней части окна предоставляет более подробную информацию о содержимом папки и в то же время не отвлекает пользователя от основной работы. Окно с сообщением об ошибке отображается тогда, когда пользователь пытается открыть папку, к которой он не имеет доступа. В этом случае действия пользователя прерываются явно, так как он должен немедленно отреагировать на сообщение.

В отличие от рассмотренных способов оповещения, функция `alert()` применима лишь для отдельных случаев. Хотя решение, предполагающее ее применение, реализуется просто, его нельзя назвать изящным. Если мы стремимся достичь надежности, согласованности и простоты, имеет смысл разработать базовые средства для оповещения пользователя, которые будут применяться во всем приложении. В следующих разделах мы постараемся сделать это.

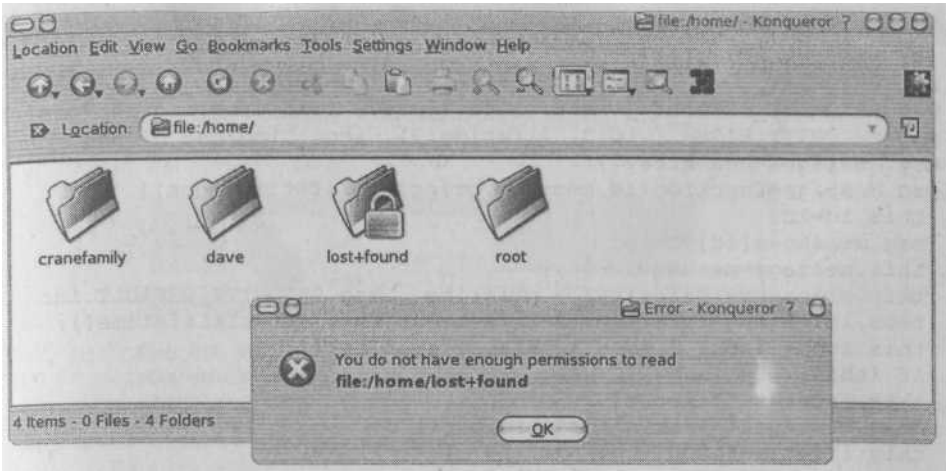


Рис. 6.1. Соглашения, принятые для предоставления информации о состоянии. Внешний вид пиктограммы отражает характеристики объекта (в данном случае права доступа), строка состояния позволяет получить общие сведения, а модальное диалоговое окно — подробную информацию об объекте. В данном случае в качестве примера приведена среда рабочего стола Unix, но аналогичные соглашения действуют во всех популярных графических интерфейсах

6.3.1 *Основные принципы оповещения*

В качестве первого шага нашей работы определим, как должно выглядеть сообщение, предоставляющее информацию пользователю. Логично предположить, что в него следует включить текстовое описание и, возможно, пиктограмму.

Если мы оповещаем пользователя о действиях, предпринимаемых в фоновом режиме, то некоторые сообщения окажутся важнее других. Вместо того чтобы в каждом отдельном случае решать, следует ли отобразить сообщение, лучше определить универсальные уровни приоритетов и применять их к каждому сообщению.

В общем случае нам необходимо сообщить пользователю некоторые сведения, которые тот может принять к сведению или отвергнуть. Некоторые из сообщений достаточно важны и должны отображаться до тех пор, пока пользователь не отключит их явным образом, а другие имеют смысл только в течение ограниченного времени. Ряд сообщений может удалиться без вмешательства пользователя. Например, информация о том, что клиент ожидает получения данных по сети, нужна лишь до активизации функции обратного вызова. Если данные будут приняты до того, как пользователь удалит сообщение, оно автоматически перестанет отображаться. В некоторых случаях, например при получении новостей, имеет смысл задать определенное время Жизни, по истечении которого сообщение должно быть удалено с экрана.

Объект Message, код которого приведен в листинге 6.3, создан с учетом этих требований. Этот объект реализует универсальный механизм оповещения пользователя. Установив модель оповещения, мы можем оформлять сообщения различными способами.

Листинг 6.3. Объект Message

```
var msg=new Objectt);
rasg.PRIORITY_LOWb { id:1, lifetime:30, icon:"img/msg_lo.png" J;
msg.PRIORITY_DEFAULT={ id:2, lifetime:60, icon:"img/msg_def.png" };
msg.PRIORITY_HIGH= { id:3, lifetime:-1, icon:"img/msg_hi.png" };
msg.messages=new Array();
msg.Message=function{id,message,priority,lifetime,icon){
 this.id=id;
 msg.messages[id]«this;
 this.message=message;
 this.priority=(priority) ? priority : msg.PRIORITY_DEFAULT.id;
 this.lifetime-(lifetime) ? lifetime : this.defaultLifetime();
 this.icon»(icon J ? icon : this.defaultlcon();
 if (this.lifetime>0){
 this.fader=setTimeout(
 "msg.messages['"+thia.id+"'].clear()",
 this.lifetime*1000
 );
 >
}
msg.Message.prototype.clear=function(){
msg.messages[this.id1«null;
)
msg.Message.prototype,defaultLifetime=function(){
 if (this.priority<=msg.PRIORITY_LOW.id){
 return msg. PRIORITY__LOW. lifetime;
 }else if {this.priority==msg.PRIORITY_DEFAULT.id){
 return msg.PRIORITY_DEFAULT.lifetime;
 }else if (this.priority>=msg.PRIORITY_HIGH.id){
 return msg.PRIORITY_HIGH.lifetime;
 1
)

msg.Message.prototype.defaultlcon=function(){
 if (this.priority<=msg.PRIORITY_LOW.id){
 return msg.PRIORITY_LOW.icon;
 Jelse if (this.priority==msg.PRIORITY_DEFAULT.id){
 return msg.PRIORITY_DEFAULT.icon;
 Jelse if (this.priority>=msg.PRIORITY_HIGH.id){
 return msg.PRIORITY_HIGH.icon;
 }
}
```

Для системы оповещения мы определили глобальный объект msg, выполняющий функции пространства имен. В нем содержится ассоциативный массив, позволяющий обращаться к любому сообщению по его уникальному идентификатору. Схема генерации идентификаторов зависит от конкретного приложения.

Уровни приоритета, низкий, средний (он же приоритет по умолчанию) и высокий, определяются с помощью констант. В соответствие каждому приоритету поставлены пиктограмма и время жизни (задаваемое в секундах). Пиктограмму и время жизни можно переопределить с помощью необязательных параметров, предусмотренных в конструкторе. Время жизни, равное -1.

I

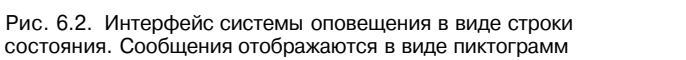


Рис. 6.2. Интерфейс системы оповещения в виде строки состояния. Сообщения отображаются в виде пиктограмм

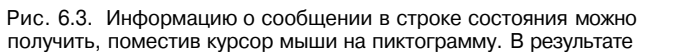


Рис. 6.3. Информацию о сообщении в строке состояния можно получить, поместив курсор мыши на пиктограмму. В результате на экране отобразится окно с подсказкой

>

(Задается для высокоприоритетных сообщений. Оно указывает на то, что сообщение не должно автоматически удаляться; решение о прекращении его ;Отображения принимает либо пользователь, либо функция обратного вызова.

Ы.2. Определение требований к пользовательскому интерфейсу

Применяя термины MVC, мы можем сказать, что нами только что была описана модель системы оповещения. Для того чтобы системой можно было пользоваться, надо определить представление. Существует много способов .визуального отображения информации, предназначенной для пользователя. В данном случае мы выберем вариант строки состояния, в которой информация будет отображаться в виде пиктограмм (рис. 6.2).

Пиктограмма с красным знаком X — стандартное средство оповещения на вязком уровне. Третья слева пиктограмма на рисунке — это шар синего цвета, отбрасывающий тень; данная пиктограмма переопределяет изображение X. используемое по умолчанию. Более подробную информацию о сообщении, ^соответствующем каждой пиктограмме в строке состояния, можно получить *$* окне подсказки (рис. 6.3).

[Такой механизм оповещения нельзя назвать навязчивым. Строка состоя- • *шя* занимает относительно мало места на экране, но появление новой пиктограммы заметно для пользователя. Срочные сообщения надо сделать более заметными, поэтому мы отображаем в строке состояния только низкоприо- ^ритетные сообщения. Информация с более высоким приоритетом выводится -В диалоговом окне (рис. 6.4), которое отображается на экране до тех пор, дюка пользователь не удалит его.

I Диалоговое окно может быть модальным и немодальным. В случае морального окна остальные элементы интерфейса блокируются до тех пор, по- *$&* окно не будет закрыто. Закрытое окно отображается в виде пиктограммы *к* правой части строки состояния. В следующих двух разделах мы обсудим реализацию описанных средств.

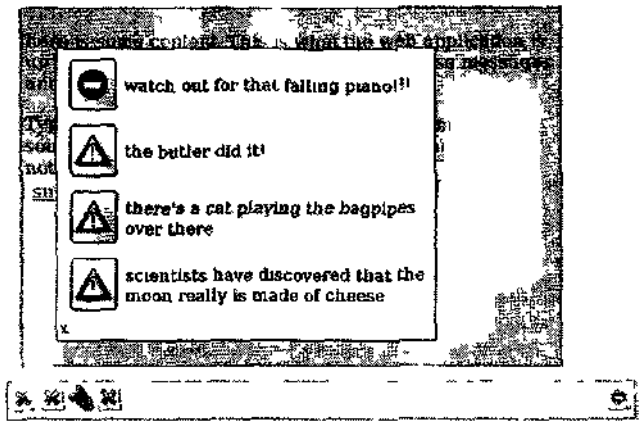


Рис. 6.4. Высокоприоритетные сообщения отображаются в диалоговом окне и располагаются в нем в соответствии с их приоритетами

6.4. Реализация базовых средств оповещения

Мы определили два основных элемента пользовательского интерфейса: строку состояния и всплывающее диалоговое окно. Теперь приступим к их реализации. Система оповещения достаточно сложна, поэтому разобьем работу над ней на отдельные этапы. Сначала модифицируем объект Message так, чтобы он мог обеспечить свое отображение в разных ситуациях: когда он представлен в строке состояния в виде пиктограммы, при выводе окна подсказки или в составе всплывающего диалогового окна. Начнем с реализации строки состояния.

6.4.1. *Отображение пиктограмм в строке состояния*

Строка состояния должна отображаться на экране и содержать пиктограммы, представляющие активизированные сообщения. Обязанности по воспроизведению конкретных пиктограмм мы делегируем объекту Message. Его можно описать в терминах архитектуры MVC. Средства отображения рассматриваются как представление, а интерактивные элементы выполняют роль контроллера. Если бы мы предполагали использовать произвольные механизмы отображения, то столкнулись бы с серьезными проблемами при реализации системы оповещения. Однако мы не планируем подобное решение. Более того, для обеспечения согласованности в пределах приложения надо использовать единый механизм оповещения. В листинге 6.4 показан метод, предназначенный для воспроизведения объекта Message.

Листинг 6.4. базовые 'Средства отображения сообщений

```
//O Воспроизведение сообщения
rasg.Message.prototype.render=function(el){
 if (this.priority<=msg.PRIORITY_LOW,id){
 this.renderSmall(el);
```

```
 lelse if (this.priority>=msg.PRIORXTX_DEFAULT.id){
 this.renderFull(el);
 }
)

// @ Вывести в качестве пиктограммы с поддержкой
// окна подсказки
msg.Message.prototype.renderSmall=function{el){
 this.icoTd=document.createElement("div");
 var ico=document.createElement("img");
 ico.src=this.icon;
 ico.classNarae="msg_smalI_icon";
 this.icoTd.appendChild(ico);
 this.icoTd.messageObj=this;
 this.icoTd.onmouseover=msg.moverIconTooltip,-
 this.icoTd.onmouseout=msg.moutlconTooltip;
 this.icoTd.onclick=msg.clicklconTooltip;
 el.appendChild(this.icoTd);
}

// ® Помещение курсора мыши над кнопкой
msg.moverIconTooltip=function(e){
 var event=e || window.event;
 var message=this.messageObj;
 var popped=message.popped;
 if ('popped){
 message. showi?opup(event, false);
 }
}
// О Перемещение курсора мши за пределы кнопки
msg.moutIconTooltip=function(e){
 var message=this.messageObj;
 var popped=rnessage.popped;
 var pinned=message.pinned;
 if {popped && 'pinned){
 message.hidePopup();
 }
)
//0 Щелчок мыппью
msg.clickIconTooltip=function{e){
 var event=e || window.event;
 var message=this.messageObj;
 var popped=message.popped;
 var pinned=message.pinned;
 var 6xpired=message.expired;
 if (popped && pinned){
 message.hidePopup{);
 if (expired){
 message.unrender();
 }
 }else{
 message.showPopup(event.true);
 }
}
// © Вывести окно с подсказкой
msg.Message.prototype.showPopup=function(event,pinned){
```

```
    this.pinned=pinned;
    if (!this.popup){
    this.popup=document.createEleraent("dxv");
    this.popup.className='popup';
    this.renderFull(this.popup);
    document.body.appendChild(this.popup);
    }
    this.popup.style.display='block';
    var popX=event.clientX;
    var popY=event.clientY-xHeight(this.popup)-12;
    xMoveTo(this-popup,popX,popY);
    if {msg.popper && msg.popper!=this){
    msg.popper.hidePopupf);
    }
    this.popped=true;
    msg.popper=this;
   }
   // Скрыть окно с подсказкой
  msg.Message.prototype.hidePopup=function(){
    if (this.popped){
    if (this.popup){
    this.popup.style.display='none';
    }
    this.popped=false;
    }
_}_____________________________________________•_____________m
```

Мы реализовали для объекта Message высокоуровневый код отображения и определили детали представления в строке состояния. Начнем с высокоуровневого кода. Метод render() О получает в качестве параметра элемент DOM. В зависимости от приоритета сообщения он передает управление либо методу renderSmall() ®, либо методу renderFulK) ©. В строке состояния представлены только сообщения с низким приоритетом, которые отображаются в виде пиктограмм. При помещении на такую пиктограмму курсора мыши выводится окно с подсказкой (рис. 6.2 и 6.3).

Функция rendersmalio отображает пиктограмму в составе элемента DOM и устанавливает обработчики событий, в процессе работы которых выводится подсказка.

Поскольку в данной главе мы стараемся придать Ajax-приложениям профессиональный вид, средства поддержки подсказки полностью определены. В работу с подсказкой вовлечены обработчики трех событий. Когда курсор мыши помещается над пиктограммой О, подсказка отображается на экране до тех пор, пока курсор не будет перемещен с данного элемента ©. По щелчку на пиктограмме подсказка "фиксируется" ©. В этом случае она будет удалена с экрана либо в результате повторного щелчка, либо при отображении другой подсказки (в каждый момент времени на экране может отображаться только одно окно с подсказкой).

6.4.2. Отображение подробных сообщений

В диалоговом окне выводятся высокоприоритетные сообщения либо сообщения, имеющие приоритет по умолчанию. Они отображаются в виде пиктограмм, сопровождаемых строкой текста (рис. 6.4). Такое же представление требуется нам для окон подсказки, соответствующих пиктограммам в строке состояния. Функция showPopupO, отвечающая за вывод подсказки, обращается к методу renderFull {), который отображает детальное сообщение. Этот же метод используется для представления информации в диалоговом окне. Такой подход исключает неоправданное дублирование кода и, кроме того, обеспечивает согласованность внешнего вида различных элементов пользовательского интерфейса. Код метода renderFull () приведен в листинге 6.5.

ЛИСТИНГ 6.5. Метод renderFullO

```
msg.Message.prototype.renderFull=functaon(el){

 var inTable=(el.tagName=="TBODY");
 var topEl=null;
 this.row=document.createElement("tr");
 if (!inTable){
 topEl=document.createElement{"table");
 var bod=document.createElement("tbody");
 topEl.appendChild(bod);
 bod.appendChild(this.row);
 }else{
 topEl=this.row;
 }

 var icoTd=document.createElement("td");
 icoTd.valign=center';
 this.row.appendChild(icoTd);
 var ico=document.createElement{"img");
 ico.src=this.icon;
 icoTd.className="msg_large_icon";
 icoTd.appendChild(ico);

 var txtTd=document.createElement("td");
 txtTd.valign-'top';
 txtTd.className="msg_text";
 this.row.appendChild{txtTd);
 txtTd.innerHTML=this.message;

 el.appendChild(topEl);
}
```

Метод renderFull () генерирует строку таблицы. Он проверяет элемент DOM и, если тот представляет собой дескриптор <tbody>, непосредственно включает его в таблицу. В противном случае генерируются необходимые дескрипторы <table> и <tbody>. Такой подход позволяет включать в таблицу, содержащуюся в диалоговом окне, несколько сообщений и в то же время обеспечивает корректное заполнение элемента в окне подсказки.

Заметьте, что мы не используем здесь методы W3C DOM, текст сообщения включается в состав таблицы посредством свойства innerHTML. Это открывает дополнительные возможности по представлению HTML-элементов по сравнению с генерацией обычного текстового узла.

6.4.3. Формирование готовой системы

На данном этапе нами полностью реализована поддержка пиктограмм в строке состояния и отображение текста сообщений. За вывод диалогового окна и строки состояний отвечает высокоуровневый метод render(), код которого показан в листинге 6.6.

Листинг 6.6. Функция msg. render ()

```
msg.render=function(msgbar){
 if (!msgbar){
msgbar='msgbar';
 }
 // О Обеспечить вывод строки состояния
msg.msgbarDiv=xGetElementById(msgbar);
 if (!msg.msgbarDiv){
msg.msgbarDiv=msg.createBar(msgbar);
 }
 styling.removeAllChildren(msg.msgbarDiv);
 var lows=new Array();
 var nveds=new ArrayO;
 var highs=new Array();
 // @ Сортировать сообщения в соответствии с приоритетом
 for (var i in msg.messages){
 var message=msg.messages[i];
 if (message){
 if (message.priority<=msg.PRIORITY__LOW.id) {
 lows.append (message);
 }else if (raessage.priority==msg.PRIORITY_DEFAULT.id){
meds.append(message);
 Jelse if (message.priority>=msg.PRIORITY_HIGH.id){
highs.append(message);
 }
 }
 }
 // © Воспроизвести низкоприоритетные сообщения
 for (var i=0;i<lows.length;i++){
lows fi].render(msg.msgbarDiv);
 }
 // О Воспроизвести сообщения с высоким и средним приоритетом
 if (meds.length+highs.length>O){
msg.dialog=xGetElementByld(msgbar+"_dialog");
 if (!msg.dialog){
msg.dialog=msg.createDialog(
msgbar+"_dialog",
msg.msgbarDiv,
 (highs.length>0) );
 // 0 Обеспечить отображение диалогового окна

 }
```

```
 styling.removeAHChildren(msg.dialog.tbod);
 for  (var  i=0;i<highs.length;i++){
 highs[i].render(msg.dialog.tbod);
 }
 for  (var  i=0;Kmeds.length;i++){
 meds⌠i].render(msg.dialog.tbod);
 }
 if  (highs.length>0){
 msg.dialog.ico.src=msg.PRIORITY_HIGH.icon;
 }else{
 msg.dialog.ico.src=msg.PRIORITY_DEFAULT.icon;
 }
 )
}
// 0 Создать строку состояния
msg.createBar=function(id){
 var  msgbar=document.createElement("div");
 msgbar.className='msgbar' ;
 msgbar,id=id;
 var  parentEl=document.body;
 parentEl.append{msgbar);
 return msgbar;
}
// в Создать диалоговое окно
msg.createDialog=function(id,bar,isModal){
 var  dialog=document.createElement("div");
 dialog.className='dialog' ;
 dialog.id=id;
 var  tbl=document.createElement("table");
 dialog.appendChiId(tbl);
 dialog.tbod=document.createElement("tbody");
 tbl.appendChild(dialog.tbod);
 varcloseB°atton=document. createElement ("div");
 closeButton.dialog=dialog;
 closeButton.onclick=msg.hideDialog;
 var  closeTxt=document.createTextNode("x");
 closeButton.appendChild(closeTxt);
 dialog.appendChild(closeButton);
 // О Добавить модальный уровень
 if  (isModal){
 dialog.modalLayer=document.createElement("div");
 dialog.modalLayer.className='modal' ;
 dialog.modalLayer.appendChild(dialog);
 document.body.appendChild(dialog.modalLayer);
 }else{
 dialog.clas sName+=' non-modal';
 document.body.appendChild(dialog);
 }

 dialog.ico=document.createElement("img");
 dialog.ico.className="msg_dialog_icon";
 dialog.ico.dialog=dialog;
 dialog.ico.onclick=msg-showDialog;
 bar.appendChild(dialog.ico);
 return  dialog;
}
```

```
// Скрыть диалоговое окно
msg.hideDialog=function(e){
 var dialogs(this.dialog) ? this.dialog : msg.dialog;
 if  (dialog){
 if  (dialog.modalLayer){
 dialog.modalLayer.style.display='none' ;
 }else{
 dialog.style.display='none' ;
 }
 }
}
// Отобразить диалоговое окно
msg.showDialog=function(e){
 var dialog=(this.dialog) ? this.dialog : msg,dialog;
 if  (dialog){
 if  (dialog.modalLayer){
 dialog.modalLayer.style.display*'block' ;
 }else{
 dialog.style.display='block';
 }
 }
}
```

К функции render() можно обращаться многократно. В процессе работы она проверяет присутствие компонентов пользовательского интерфейса О, © и по мере необходимости создает их, используя функции createDialog © и createBar ©. Для формирования компонентов интерфейса используются стандартные методы обработки элементов DOM. Обработчики событий позволяют отображать и скрывать диалоговое окно.

Для воспроизведения всех сообщений система сначала сортирует их по приоритету и размещает в трех временных массивах @. Затем низкоприоритетные сообщения выводятся в строке состояния ©, а остальные — в диалоговом окне, причем первыми располагаются высокоприоритетные сообщения О.

Для того чтобы реализовать модальное окно, мы помещаем видимые средства поддержки диалога в элемент DIV, занимающий весь экран, и блокирующий все события, связанные с мышью ©. Для модального элемента DIV определен белый фон с прозрачными пикселями. По этому признаку можно отличить модальное окно. Мы не используем установки прозрачности CSS, потому что в этом случае любой вложенный элемент будет также прозрачным. Файл CSS для системы оповещения представлен в листинге 6.7.

Листинг 6.7. Содержимое файла msg. ess

```
•msg_small_icon{
 height: 32px;
 width: 32px;
 position:relative;
 float:left;
}
• ™sg_jiialog_icon {
 height: 32px;
 width: 32px;
```

```
 position:relative;
 float:right;
1

•msg_large_icon{
 height: 64px;
 width: 64px;
}

.msg_text{
 font-family: arial;
 font-weight: light;
 font-size: 14pt;
 color: blue;
}

,msgbar{
 position:relative;
 background-color: white;
 border: solid blue 1px;
 width: 100%;
 height: 38px;
 padding: 2px;
}

.dialog{
 position: absolute;
 background-color: white-
 border: solid blue 1px;
 width: 420px;
 top: 64px;
 left: 64px;
 padding: 4px;
}

.popup{
 position; absolute;
 background-color: white;
 border: solid blue 1px;
 padding: 4px;
}

.non-modal{
}

.modal{
 position: absolute;
 top: Opx;
 left: Opx;
 width: 100%;
 height: 100%;
 background-image:url(img/modal_overlay.gif);
```

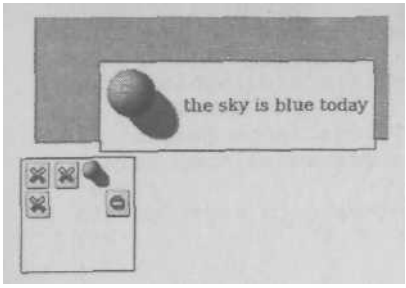


Рис. 6.5. Использование CSS-атрибутов f l o a t позволяет размещать пиктограммы в контейнере любой формы. В данном примере мы задали квадратную строку состояния; пиктограммы автоматически перекомпоновались *с* учетом выравнивания по левому и правому краю

Следует заметить, что в GSb-классах msg_small_icon и msg_dialog_icon используется стиль float. Класс msg_small_icon применяется для воспроизведения пиктограмм, соответствующих низкоприоритетным сообщениям. Эти пиктограммы располагаются в ряд, начиная с левого края. Для msg__dialog_icon задано выравнивание по правому краю. Базовые средства позволяют отображать строку состояния любой формы и любого размера. Плавающие элементы располагаются в строке состояния в ряд; при заполнении текущего ряда формируется следующий ряд (рис. 6.5).

Теперь, когда мы создали пользовательский интерфейс для объекта Message, надо модифицировать сам объект. Создаваемые сообщения должны иметь возможность включать самих себя в состав пользовательского интерфейса и удаляться по окончании действия. В листинге 6.8 показаны изменения, которые необходимо внести в состав кода, чтобы обеспечить эти возможности.

Листинг 6.8. Модифицированный объект Message

```
var msg=new Object();
msg.PRIORITY_LOW= { id:1, lifetime:30, icon:"img/msg_lo.png" };
msg.PRIORITY_DEFAULT={ id:2, lifetime:60, icon:"img/msg_def.png" J;
msg.PRIORITY_HIGH= { id:3, lifetime:-l, icon:"img/msg__hi .png" };
msg.messages=new ArrayO;
msg.dialog=null;
msg.msgBarDiv=null;
msg.suppressRender=false;
msg.Message=function(id,message,priority,lifetime,icon){
 this.id=id;
 msg.messages[id]=this;
 this.message=message;
 this.priority={priority) ? priority : msg. PRIORITY__DEFAULT. id;
 this.lifetime=(lifetime) ? lifetime : this.defaultLifetime();
 this.icon={icon) ? icon : this.defaultlcon();
 if (this.lifetime>0){
 this.fader=setTimeout(
 "msg.messages['"+this.id+"'].clear()",
 this.lifetime*1000
 );
 }
 // О Дополнительные параметры
 if (!msg.suppressRender){
 this.attachToBar();
```

```
 }
}

// @ Дополнительные параметры
msg.Message.prototype.attachToBar=function(){
 if (!msg.msgbarDiv){
 msg.render();
 }else if (this.priority==msg.PRIORITY_LOW.id){
 this.render(msg.msgbarDiv);
 lelsei
 if (!msg.dialog){
 msg.dialog=msg.createDialog(
 msg.msgbarDiv.id+"_dialog",
 msg.msgbarDiv,
 (this.priority==msg.PRIORITY_HIGH.id)
 );
 }
 this.render(msg.dialog.tbod);
 msg.showDialog();
 }
}

msg.Message.prototype.clear=function(){
 msg.messages[this.id]=null;
 // & Дополнительные параметры
 if (this.row){
 this.row.style.display='none' ;
 this. row.messageObj'^nullf-
 this.row^null;
 }
 if (this.icoTd){
 this.icoTd.style.displays'none' ;
 this.icoTd.messageObj=null;
 this.icoTd=null;
 }
```

Мы стремимся упростить работы с системой, поэтому при создании сообщения оно автоматически включается в состав пользовательского интерфейса О. Для того чтобы обеспечить воспроизведение нового сообщения, достаточно вызвать конструктор объекта. В зависимости от приоритета сообщения оно будет помещено либо в строку состояния, либо в диалоговое окно ©. Если нежелательно воспроизводить каждое сообщение, например, когда мы добавляем одновременно несколько сообщений, предусмотрен флаг, позволяющий отменить автоматическое отображение. В этих случаях после создания всех необходимых сообщений мы можем вручную вызвать функцию `msg.render()`.

Удаляя сообщения с помощью функции `clear()`, мы автоматически удаляем соответствующие элементы пользовательского интерфейса ©.

Итак, мы получили комплект базовых средств, упрощающих оповещение пользователей. Мы можем включать оповещение вручную или применять созданную систему в составе других компонентов, предназначенных для по-

вторного использования. В следующем разделе будет продемонстрирована совместная работа системы оповещения с объектом ContentLoader Пользователю будет предоставляться информация о загрузке объектов по сети.

6.5. Предоставление информации в запросах

В главе 5 мы представили объект ContentLoader, инкапсулирующий сетевой трафик. Применим систему оповещения для информирования пользователя о состоянии запроса. Начнем с формулирования общих требований.

При передаче запроса серверу мы должны сформировать низкоприоритетное сообщение о том, что данный запрос обрабатывается. Для того чтобы отделить запрос по сети от других оповещений с низким приоритетом, мы изменим внешний вид пиктограммы. Будем использовать для представления сообщения данного типа условное изображение земного шара. Оно присутствовало в программе просмотра информации о планетах, которая рассматривалась в главах 4 и 5.

По окончании обработки запроса сообщение должно быть удалено. Вместо него следует создать другое низкоприоритетное сообщение о том, что запрос обработан успешно, либо сообщение со средним приоритетом о возникновении ошибки.

Для того чтобы реализовать подобное поведение, нам надо создавать объекты Message на определенных этапах жизненного цикла запроса, а именно: при его инициализации, а также по завершении или получении информации об ошибках. Модифицированный код объекта ContentLoader показан в листинге 6.9.

Листинг б.Э.-Объект ContentLoader с оповещением пользователя

```
net.ContentLoader=function( ... )
    { ... } ;

net.ContentLoader.msgId=1;
net.ContentLoader.prototyped
 loadXMLDoc:function{url,method,params,contentType){
 if {Imethod){
 method="GET";
 }
 if {IcontentType && method=="POST"){
 contentType='application/x-www-form-urlencoded';
 ]
 if (window.XMLHttpRequest){
 this.req=new XMLHttpRequest();
 } else if (window.ActiveXObject){
 this.req=new ActiveXObj ect("Microsoft.XMLHTTP");
 ]
 if (this.reqH
 try{
 var loader=this;
 this.req.onreadystatechange=function(){
 loader.onReadyState.call(loader);
 )
```

```
this.req.open(method,url,true);
if (contentType)I
this.req.setRequestHeader('Content-Type', contentType);
}
// О Оповещение об инициализации запроса
this.notification=new msg.Message(
"netOO"+net.ContentLoader.msgId,
"loading "+url,
msg.PRIORITY_LOW.id,
-1,
"img/ball-earth.gif"
);
net.ContentLoader.msgId++;
this.req.send(params);
}catch (err){
this.onerror.call(this);
}
}
Ь

onReadyState:function(){
var req=this.req;
var ready=req.readyState;
if (ready==net.READY_STATE__COMPLETE) {
var httpStatus=req.status;
if (httpStatus==200 || httpStatus==*O) {
this.onload.call(this);
// @ Удаление существующего оповещения
this.notification.clear{);
}else{
this.onerror.call(this);
}
}
},

// © Сообщение об ошибке
defaultError:function(){
var msgTxt="error fetching data!"
+"<ulxli>readyState:"+this.req.readyState
+"<li>status: "+this.req.status
+"<li>headers: "+this.req.getAHResponseHeaders()
+"</ul>";
if (this.notification){
this.notification.clear();
J
this.notifxcation=new msg.Message(
"net__errOO"+net.ContentLoader.msgId,
msgTxt,msg.PRIORITY__DEFAULT.id
);
net.ContentLoader.msgId++;
}
```

Когда мы создаем запрос посредством вызова loadXMLDoct), мы формируем оповещение О и связываем ссылку на него с объектом ContentLoader. Заметьте, что время жизни устанавливается равным -1, поэтому сообщение не будет автоматически удаляться.

В методе onReadyState() мы удаляем сообщение при благополучном завершении операции €). При наличии ошибки мы вызываем метод defaultError(), который формирует соответствующее сообщение €>. Чтобы сообщение лучше воспринималось пользователями, вместо обычного текста для его формирования применяются средства HTML.

В листинге 6.10 показан пример страницы, в которой используется модифицированный объект ContentLoader

Листинг 6.10. Страница, предусматривающая оповещение пользователя

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0
Strict//EN"
 "http://www.w3.org/TR/xhtmll/DTD/xhtmll-strict.dtd">
<btml>
<head>
<title>Notifications test<Vtitle>
<link rel=stylesheet type="text/css" href="msg.ess"/>
<script type="text/javascript" src="x/x_core.js"X/script>
<script type="text/javascript" src="extras-array.js"x/script>
<script type="text/javascript" src="styling.js"></script>
<script type="text/javascript" src="msg.js"></script>
<script type="text/javascript" src="net_notify.js"X/script>
<script type="text/javascript">
window.onload=function(){
 msg.render('msgbar');
}
var msgld=1;

function submitUrl(){
 var url=document,getElementById{'urlbar').value;
 // О Запрос к. серверу
 var loader=new net.ContentLoader(url,notifyLoaded);
}

function notifyLoadedf){
 // © Оповещение о том, что ресурс загружен
 var doneMsg=new msg.Message(
 "done00"+msgld,
 "loaded that resource you asked for: "+this.url,
 msg.PRIORITY_LOW.id
 );
 msgld++;
 msg.render('msgbar');
}

</script>
</head>
<body>
<div class='content'>
<p>here is some content. This is what the web
```

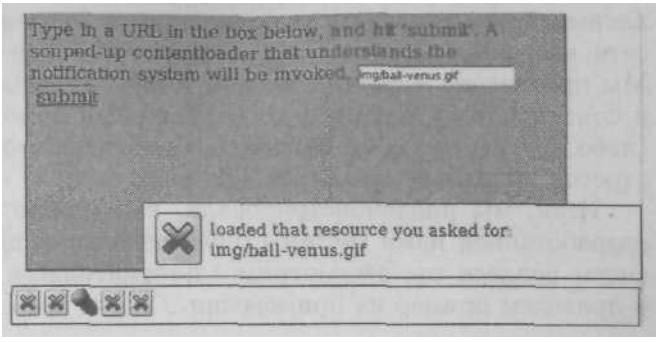


рис. 6.6. В окне подсказки, которое отображается при помещении курсора на пиктограмму, содержатся сведения об успешной загрузке ресурса

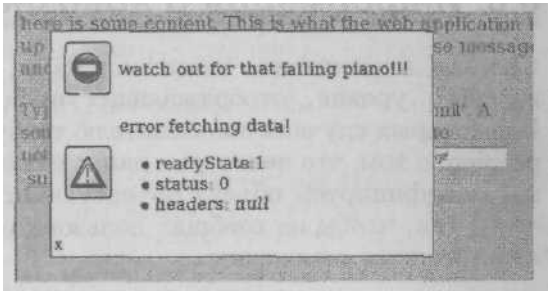


Рис 6.7. Если при обработке запроса возникла ошибка, информация о ней отображается в диалоговом окне (в данном случае в окне отображены два сообщения; второе информирует об ошибке)

```
application is up to when not being bugged silly
by all these messages and notifications and stuff.
<p>Type in a URL in the box below (from the
same domain, see Chapter 7), and hit 'submit'.
A souped-up contentloader that understands the
notification system will be invoked.
<input id='urlbar' type='text'/> 
<a href='javascript:submitUrl()'>submit</a>
</div>
<div id='msgbar' class='msgbar'></div>
</body>
</html>
```

Web-страница (рис. 6.6 и 6.7) содержит простую HTML-форму, в которой пользователь может ввести URL. По щелчку на ссылке submit предпринимается попытка загрузить ресурс, на который указывает URL O. В случае успешного завершения операции происходит обращение к функции обратного вызова — notifyLoaded(). Функция notifyLoaded() не выполняет с ресурсом никаких действий, а лишь сообщает о его загрузке, создавая объект Message ©.

Заметьте, что поведение в случае успешной обработки запроса не предусмотрено при реализации системы. Оно обеспечивается посредством обработчика onload. Благодаря такому решению система может быть адаптирована с учетом различных требований. В примере, приведенном в листинге *6.9,* поведение в случае ошибки было реализовано непосредственно в коде. В ре-

альных приложениях не каждая ошибка, связанная с передачей данных по сети, настолько важна, чтобы отображать сообщение о ней *в* диалоговом окне. Мы предлагаем читателю в качестве упражнения самостоятельно добавить к ContentLoader параметр, определяющий приоритет сообщения об ошибке (либо другим способом изменить обработчик onError для реализации менее строгой политики).

Итак, мы продемонстрировали, как можно обеспечить взаимодействие разработанной нами системы оповещения с существующим кодом. В следующем разделе мы рассмотрим альтернативные соглашения об оповещении и приведем пример их применения.

6.6. Информация о новизне данных

Система оповещения, созданная нами, предоставляет набор компонентов верхнего уровня, отображающих информацию о системной активности. В некоторых случаях пользователю требуются дополнительные сведения, например, о том, что некоторые данные были модифицированы. В этом разделе мы модифицируем объект ObjectViewer, который рассматривался в главах 4 и 5, так, чтобы он сообщал пользователю информацию о данных, недавно подвергшихся изменению.

6.6.1. Простой способ выделения данных

Начнем решение задачи с рассмотрения простого способа выделения данных путем подсветки; для этой цели будем использовать инвертирование изображения. В пользовательском интерфейсе ObjectViewer используются в основном синие и серые тона, поэтому красный цвет будет выделяться на фоне окружающих элементов. Первое, что нам нужно сделать, — это определить дополнительный класс CSS, представляющий недавно измененные данные.

```
.new{ background-color: IfOeOdO; }
```

Мы выбрали очень простой стиль. Для реального приложения этого недостаточно; чтобы интерфейс выглядел хорошо, надо приложить дополнительные усилия. В листинге 6.11 показаны изменения кода ObjectViewer, предназначенные для того, чтобы выделить свойства, недавно подвергшиеся редактированию. В данном случае признак новизны данных автоматически удаляется по истечении заданного интервала времени.

Листинг 6.11. ObjectViewer, предусматривающий выделение новых данных"

```
objviewer,PropertyViewer.prototype.commitEdit=function(value){

 if (this.type=objviewer.TYPE_SIMPLE){
 this.value=value;
 var valDiv=this.renderSimple();
 var td=this.valTd;
 td.replaceChild(valDiv,td.firstChild);
 this.viewer.notifyChange(this);
```

```
    // О Установить признак новизны
    this.setStatus(objviewer.STATUS_NEW);
    }
  }

  objviewer.STATUS_NORMAL=1;
  objviewer.STATUS_NEW=2;

  objviewer.PropertyViewer.prototype.setStatus=function(status){
    this.status=status;
    if (this.fader){
    clearTimeout(this.fader);
    }
    if (status==objviewer.STATUS_NORMAL){
    this.valTd.className='objViewValue' ;
    }else if (status==objviewer.STATUS_NEW){
    this.valTd.className='objViewValue new' ;
    var rnd="fade_"+new Date().getTime();
    this.valTd.id=rnd;
    this.valTd.fadee=this;
    // 0 Установить значение тайы-аута
    this.fader=setTimeout("objviewer.age('"+rnd+"')",5000);
    }
  }

  objviewer.age=function(id){
    var el=document.getElementById(id);
    var viewer=el.fadee;
    // © Восстановить состояние по окончании отсчета времени
    viewer.setStatus(objviewer.STATUS_NORMAL);
    el.id-"";
    el.fadee=null;
_}
```

Мы определили два варианта данных: "обычные" и "новые". Можно было бы использовать для их идентификации логическое значение isNew, но мы предпочли определить два признака. Такой подход допускает дальнейшее расширение; например, мы сможем ввести дополнительный признак для тех данных, которые были модифицированы и информация об обновлении которых передана на сервер. При фиксации изменяемых значений вызывается метод setStatus() О. В результате задается соответствующий класс CSS и для "новых" данных устанавливается значение таймера ©. После истечения пятисекундного интервала данные переходят в "нормальное" состояние. (Для реального приложения время, в течение которого информация будет отображаться как "новая", должно быть намного больше. Интервал в пять секунд выбран только для проверки и демонстрации возможностей данного решения.) Объект сохраняет ссылку на таймер и может переустановить его значение в случае, если до истечения заданного времени возникнут новые изменения.

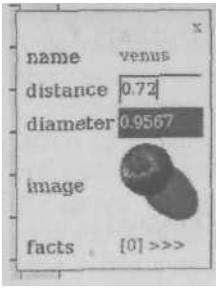


Рис. 6.8. Модифицированный объект Objectviewer отображает недавно модифицированное значение диаметра, выделяя его цветом фона

Учитывая ограничения метода setTimeout(), мы присваиваем обрабатываемому узлу DOM уникальный идентификатор. Это позволит снова обнаружить его тогда, когда таймер активизирует функцию аде() ©. Функция аде () "приводит в порядок" идентификатор и другие временные ссылки. На рис. 6.8 показан внешний вид Objectviewer после редактирования одного значения.

Через некоторое время данные перестанут считаться "новыми", и для них будет восстановлен нормальный стиль отображения.

Изменение цвета привлекает внимание пользователя к недавно измененному значению. Выделить данные можно также с помощью анимации. В следующем разделе вы увидите, что такое решение реализуется достаточно просто.

6.6.2. Выделение данных с использованием библиотеки Scriptaculous

Мы реализовали выделение данных путем установки стилей вручную. Такой подход был выбран отчасти потому, что результаты его использования хорошо видны на рисунках. При работе над реальными приложениями мы рекомендуем использовать объекты Effect библиотеки Scriptaculous, которые позволяют достаточно просто "оживить" отображаемые сообщения. Данную библиотеку мы вкратце рассмотрели в разделе 3.5.2, и вы уже знаете, что она позволяет путем выполнения несложных операций организовать различные анимационные эффекты.

Метод setstatus(), который был рассмотрен в листинге 6.11, несложно переписать для использования Scriptaculous. Предположим, что мы хотим, чтобы недавно отредактированные данные "вибрировали"[1] при отображении. Этого-легко добиться с помощью объекта Effect. Pulsate. Такой эффект привлекает внимание пользователя, но, к сожалению, его невозможно передать на статическом рисунке. Необходимые изменения в коде метода показаны в листинге 6.12.

Листинг 6.12. Выделение данных с помощью Scriptaculous

```
objviewer.PropertyViewer.prototype.setStatus=function(status){

 this.status=status;
 if (this.effect){
 this.effect.cancel();
 this.effect=null;                                          *
 }
 if (status==objviewer.STATUS_NEW){
 this.effect=new Effect.Pulsate(
 this.valTd,
 {duration: 5.0}
 )!
 )
```

Объект Pulsate берет на себя заботу по выполнению всей рутинной работы, и нам больше нет необходимости непосредственно заниматься установкой тайм-аута. Исчезает также необходимость в функции age(). Мы лишь вызываем конструктор Pulsate, передаем ему ссылку на элемент DOM и ассоциативный массив с теми параметрами, которые мы хотим изменить. По умолчанию эффект Pulsate проявляется в течение трех секунд. Мы увеличили этот интервал до пяти секунд, установив параметр duration.

Подобным образом можно выделить и другие данные, например, обновления, выполненные по инициативе сервера. Для того чтобы избежать конфликтов между различными эффектами, отключаем действующие эффекты. (В версии Scriptaculous 1.1 ко всем эффектам применима стандартная функция cancel().)

Эффекты, подобные описанному выше, обеспечивают мгновенную обратную связь с пользователем, привлекая его внимание к конкретному фрагменту данных. Диалоговое окно и строка состояния лучше подходят для представления более общей информации. Совместное использование различных способов выделения данных позволяет лучше организовать работу пользователя и повысить ее эффективность.

6.7. Резюме

В настоящей главе мы обсудили ряд способов, позволяющих создать более комфортные условия работы пользователя с Ajax-приложением. В начале главы рассматривались такие важные характеристики, как время отклика, надежность, согласованность и простота, которые в конечном итоге определяют качество приложения.

Основная часть главы была посвящена способам обеспечения обратной связи при работе пользователя с приложением. Мы реализовали отображение сообщений в строке состояния и в диалоговом окне, а также средства предоставления информации пользователю путем выделения фрагментов данных. Включение этих средств в реальную программу существенно повысит уровень информированности пользователя о процессах, происходящих в систе-

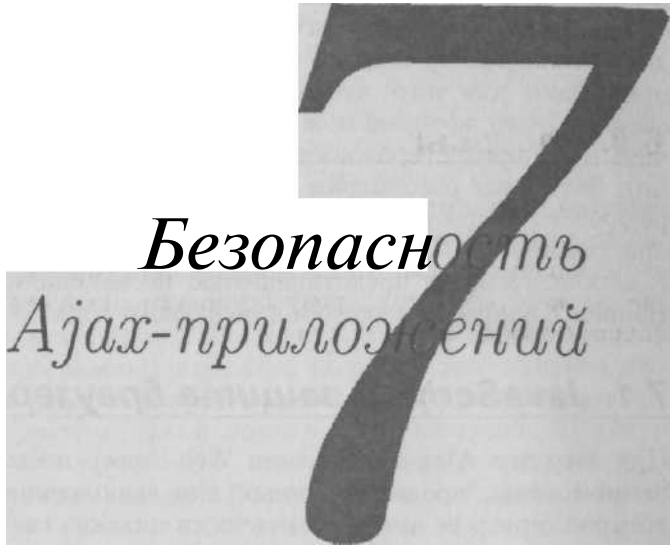
ме, а оформление их в виде повторно используемых компонентов позволяет упростить работу разработчика. В данной главе мы продемонстрировали, насколько просто включается система оповещения в уже готовый код. В наших примерах строка состояния использовалась для отображения сведений об обработке запросов сервером, а путем выделения фрагментов текста мы сообщали пользователю о данных, недавно подвергшихся изменениям. Данные средства оповещения были применены к объекту ObjectBrowser, предназначенному для отображения сведений о планетах Солнечной системы.

Интерфейс приложений несомненно имеет большое значение. Однако в следующих двух главах мы поговорим о не менее важных характеристиках, определяющих практичность приложения, таких как безопасность и производительность.

6.8. Ресурсы

Библиотеку Scriptaculous Effects можно найти по адресу http: //wihi .script. aculo.us/scriptaculous/list?category=Effects.

Дополнительные пиктограммы для примера оповещения были взяты из набора Nuvola, разработанного Дэвидом Виньони (David Vignoni) (http://www.icon-king.com/).



В этой главе...

- Модель безопасности
- Удаленные Web-службы
- Защита пользовательских данных, передаваемых по Интернету
- Защита потоков данных Ajax

Наличие средств защиты — важное свойство Интернет-служб. Система Web изначально не защищена, поэтому качество Ajax-приложения во многом зависит от наличия в нем средств обеспечения безопасности. Независимо от того, покупает ли пользователь товары в интерактивном магазине или приобретает право воспользоваться сетевыми услугами, он платит деньги, и этот факт еще больше повышает важность средств защиты.

Безопасность — обширная тема, которой посвящены многие книги. К средствам защиты Ajax во многом предъявляются те же требования, что и к системе защиты классического Web-приложения. Поэтому в данной главе мы будем уделять внимание только вопросам безопасности, специфическим для инфраструктуры Ajax. В первую очередь мы поговорим о передаче исполняемого кода по сети и мерах, которые производители браузеров принимают для того, чтобы сделать эту процедуру безопасной. Мы также обсудим шаги, которые можно предпринять для того, чтобы несколько смягчить меры предосторожности (естественно, в тех случаях, когда это допустимо). Затем мы рассмотрим вопросы защиты пользовательских данных, передаваемых серверу, и обеспечения конфиденциальности при работе с Ajax-приложениями. И наконец, уделим внимание защите служб, используемых Ajax-клиентами, и предотвращению несанкционированного доступа к ним. Начнем с вопросов передачи клиентского кода по сети.

7.1. JavaScript и защита браузера

При загрузке Ajax-приложения Web-сервер посылает браузеру набор JavaScript-команд, предназначенных для выполнения на удаленной машине, о которой сервер не имеет практически никаких сведений. Позволяя командам выполняться в среде браузера, пользователь тем самым высказывает доверие приложению и его авторам. Это доверие не всегда оправдано, поэтому производители браузеров предусматривают ряд мер, направленных на защиту системы и информации. В данном разделе мы рассмотрим эти меры и их влияние на работу программ. Мы также обсудим ситуации, в которых ограничения оказываются слишком строгими и которые желательно смягчить. Наибольший интерес у разработчиков Ajax-приложений вызывает возможность непосредственно обращаться к Web-службам независимых производителей.

Прежде чем приступать к подробному обсуждению данной темы, определим понятие мобильного кода. Любой фрагмент содержимого жесткого диска компьютера представляет собой двоичные данные. Несмотря на это, существует возможность отличить собственно данные от машинных инструкций, которые могут быть выполнены на компьютере. Обычные данные не выполняют никаких действий, по крайней мере до тех пор, пока они не будут обработаны некоторым процессом. В первых приложениях клиент/сервер клиентский код устанавливался на компьютере пользователя наравне с другими приложениями, и весь трафик, передаваемый по сети, представлял собой обычные данные. Однако в Ajax-приложениях JavaScript-код может быть выполнен. Помимо того что он предоставляет ряд интересных возможностей, которые не могут обеспечить обычные данные, он также может стать источ-

ликом опасности для системы. Назовем код мобильным, если он хранится на одной, машине и. может быть передан по сети для выполнения на другом компьютере. Компьютер, принимающий мобильный код, должен решить, доверяет ли он источнику кода. Это решение становится еще более ответственным, если исполняемый код получен из общедоступной сети. Необходим аргументированный ответ на вопрос, к каким из системных ресурсов можно предоставить доступ мобильному коду.

7.1.1. Политика "сервера-источника"

Как было сказано ранее, в среде Web-браузера выполняется код, написанный неким "третьим лицом", зачастую неизвестным пользователю. Выполнение мобильного кода, копируемого по сети, представляет потенциальную опасность для локальной системы. Для того чтобы уменьшить эту опасность, производители браузеров организуют запуск JavaScript-программ в специально сформированной среде, которая носит **название** "песочница" (sandbox). При этом программа имеет ограниченный доступ к ресурсам локальной системы либо не имеет его вовсе. Так, Ajax-приложение не может читать информацию из локальной файловой системы или записывать ее. Код Ajax-клиента также не может устанавливать сетевое соединение ни с одним сервером, за исключением того, с которого он был скопирован. Элемент IFrame, сгенерированный в результате выполнения программы, позволяет получать документы с любого узла и даже запускать их, но сценарии из различных фреймов не могут взаимодействовать друг с другом. Такой подход носит название политики "сервера-источника".

Рассмотрим очень простой пример. Определим в сценарии некоторую переменную.

```
x=3;
```

Во втором файле сценария попытаемся использовать ее.

```
alert(top.x+4);
```

Первый сценарий включен в документ верхнего уровня. По его инициативе был создан элемент IFrame и загружена страница, содержащая второй сценарий (рис. 7.1).

Если оба сценария получены из одной области или из одного домена, функция `alert()` выполняется успешно. В противном случае при выполнении второго сценария возникает ошибка.

7.1.2. Особенности выполнения сценариев в Ajax-приложении

При взаимодействии, ориентированном на сценарии (см. главу 5), JavaScript-код копируется с сервера и непосредственно выполняется на стороне клиента. В большинстве случаев клиент получает код с того же сервера, с которого он был скопирован сам. однако возможны случаи получения программного кода из другого домена. При этом возникает так называемый *кросс-сценарий*. Предоставляя право получать сценарии с произвольно выбранных узлов, мы тем самым формируем условия для возможной подмены документов или

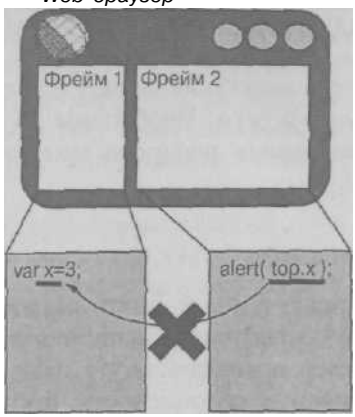


Рис. 7.1. Модель безопасности JavaScript запрещает сценариям, полученным из различных источников, взаимодействовать друг с другом

их искажения путем манипуляции DOM-информацией. Ограничения, предусмотренные в модели безопасности JavaScript, обычно обеспечивают реальную защиту от подобных явлений. Эта модель также предотвращает копирование клиентского кода Ajax на узлы, контролируемые злоумышленниками, и подмену сервера.

При взаимодействии, ориентированном на данные, риск значительно ниже, так как вместо исполняемого кода сервер предоставляет лишь данные. Тем не менее информация на серверах *злоумышленников* может быть подобрана так, чтобы нанести вред системе, используя известные недостатки в программах разбора. Таким образом, например, можно переопределить или удалить важную информацию или создать условия для неоправданно интенсивного потребления ресурсов.

7,1.3. Проблемы с поддоменами

Заметьте, что браузеры имеют весьма ограниченное представление о границах конкретной области, или домена, и в ряде случаев, когда можно было бы обойтись предупреждением, генерируют ошибку. Браузер идентифицирует домен по первой части URL и не делает попыток выяснить, указывают ли различные доменные имена на один и тот же IP-адрес. В табл. 7.1 приведено несколько примеров решений, принимаемых моделью безопасности браузера.

В случае поддоменов возможно выделить *для сопоставления* часть домена путем установки свойства document. domain. Так, например, мы можем включить приведенную ниже строку в сценарий, полученный с узла http: //www.myisp. com/dave,

document.domain^'myisp.com[1];

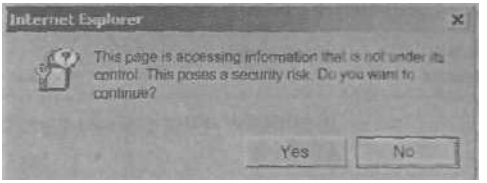
Этим мы разрешим взаимодействие со сценарием, полученным из под-Домена http://dave.myisp.com/, при условии, что в нем также будет установлено значение document. domain. Заметьте, что установить произвольное значение document.domain, например www.google.com, невозможно.

Таблица 7.1. Примеры применения браузером политики безопасности

URL	Разрешены ли кросс-сценарии	Описание
http://www.mysite.com/scriptl.js http://www.mysite.com/script2.js	Да	Сценарии получены из одного домена
http://www.mysite.com:8080/scriptl.js http://www.mysite.com/script2.js	Нет	Номера портов не совпадают (первый сценарий получен посредством порта 8080)
http://www.mysite.com/scriptl.js https://www.mysite.com/script2.js	Нет	Протоколы не совпадают (второй сценарий получен посредством защищенного протокола HTTP)
http://www.mysite.com/scriptl.js http://192.168.0.1/script2.js,	Нет	Доменное имя www.mysite.com соответствует адресу 192.168.0.1, но браузер не проверяет этот факт
http://www.mysite.com/scriptl.js http://scripts.mysite.com/script2.js	Нет	Поддомены считаются различными доменами
http://www.myisp.com/dave/scriptl.js http://www.myisp.com/eric/script2.js	Да	Несмотря на то что сценарии получены с узлов, контролируемых различными владельцами, домены считаются совпадающими
http://www.myisp.com/dave/scriptl.js https://www.mysite.com/script2.js	Нет	Имя www.mysite.com указывает на www.myisp.com/dave, НО браузер не проверяет этот факт

7.1.4. Несоответствие средств защиты в различных браузерах

Наш разговор о безопасности нельзя считать завершенным без рассмотрения существенного расхождения в средствах защиты разных браузеров. Система обеспечения безопасности Internet Explorer оперирует с набором "зон безопасности" с ограниченными правами. По умолчанию исполняемые файлы из локальной файловой системы имеют право взаимодействовать с узлами из всемирной сети, не оповещая об этом пользователя (по крайней мере, такое поведение предусмотрено в Internet Explorer 6). Таким образом, локальная



файловая система считается безопасной зоной. При попытке выполнения того же кода, но полученного с локального Web-сервера, отобразится диалоговое окно, показанное на рис. 7.2.

В случае необходимости можно написать сложное Ajax-приложение и проверять работоспособность больших фрагментов кода, оперируя данными из файловой системы. Временное исключение Web-сервера из работы системы несколько упрощает создание кода. Однако мы настоятельно рекомендуем разработчикам наряду с отработкой взаимодействия на локальной файловой системе проверять его на реальном Web-сервере. В браузере Mozilla понятие зон не используется. Приложение, загруженное из локальной системы, испытывает на себе те же ограничения, что и приложение, загруженное с Web-сервера. В Internet Explorer коды, полученные из различных зон безопасности, будут вести себя по-разному.

Итак, мы сформулировали основные ограничения, которые могут быть применены к сценариям в составе Ajax-приложений. Модель безопасности JavaScript в ряде случаев затрудняет работу, но в целом делает ее более надежной. Без нее доверие пользователей к службам Интернета и в том числе к средствам, предоставляемым Ajax, было бы столь низким, что их бы практически никто не применял.

В то же время бывает необходимо выполнять сценарии из доменов, отличающихся от того, из которого был получен сценарий. Такая необходимость возникает, например, при работе с рядом Web-служб. В следующем разделе вы увидите, как можно ослабить ограничения, связанные с защитой.

7.2. Взаимодействие с удаленным сервером

Средства защиты в составе браузера, конечно же, необходимы, но бывают ситуации, когда они лишь мешают нормальной работе. Чтобы система, обеспечивающая безопасность, работала эффективно, она обязана подвергать сомнению правомочность каждого действия. Тем не менее в ряде случаев приложению бывает необходимо обратиться к стороннему серверу. Теперь, когда вы имеете представление о политике безопасности браузера, давайте подумаем о том, как ослабить налагаемые ограничения. Мы рассмотрим два возможных решения этой задачи. Одно из них предполагает наличие дополнительного кода на стороне сервера, а второе затрагивает только клиентскую программу.

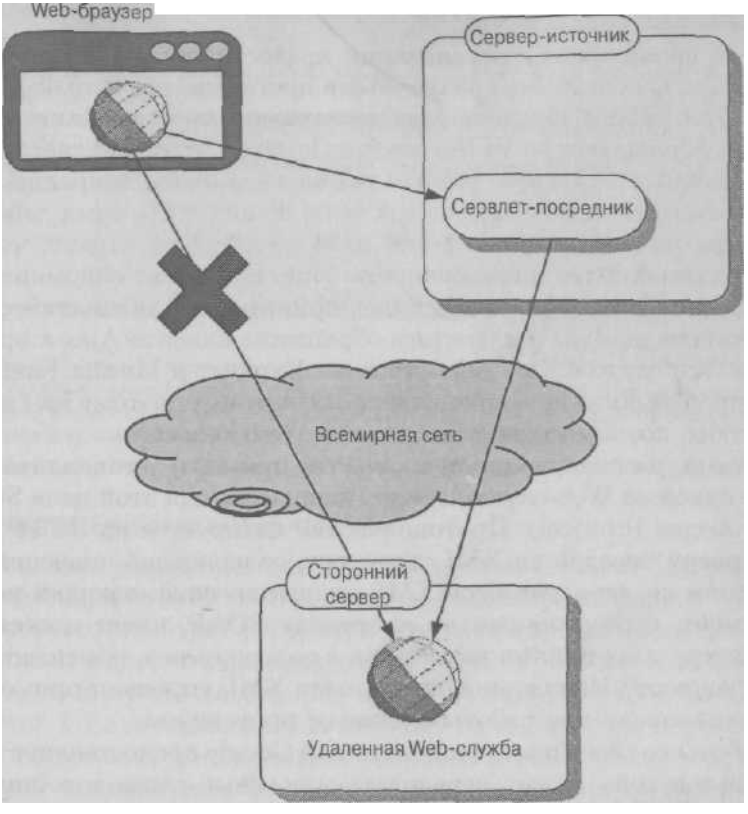


Рис. 7.3. Если A]ax-приложению необходимо получить доступ к ресурсам из другого домена, "сервер-источник" выступает а роли посредника, или прокси-сервера

7.2.1. Сервер в роли посредника при обращении к удаленной службе

В соответствии с политикой "сервера-источника" Ajax-приложение может копировать данные лишь из своего собственного домена. Если нам надо получать информацию с другого сервера, можно организовать обращение к нему не с клиентской машины, а с "сервера-источника", а ответ перенаправить клиенту (рис. 7.3).

В этом случае с точки зрения браузера данные поступают с того сервера, с которого загружена клиентская программа, что не противоречит политике "сервера-источника". Кроме того, перед перенаправлением информации клиенту на сервере может быть выполнена проверка на наличие недопустимых данных.

Недостатком такого подхода является увеличение нагрузки на сервер. Решение, которое мы рассмотрим ниже, предполагает обращение браузера непосредственно к требуемому серверу

7.2.2. Взаимодействие с Web-службами

В настоящее время многие организации предоставляют Web-службы, **котД** рые могут быть использованы различными программами, включая JavaScript, клиентов. При работе клиента Ajax желательно иметь возможность непосредственно обращаться к Web-службе. Помехой этому является политика "сервера-источника". Решить данную проблему можно, запрашивая из программы привилегии, необходимые для выполнения требуемых действий в сети. Запрос на получение привилегий отображается на экране, чтобы пользователь подтвердил его правомочность. Браузер может запомнить решение пользователя и в дальнейшем оно будет приниматься автоматически.

В данном разделе мы рассмотрим обращение клиента Ajax к произвольно выбранной Web-службе. Браузеры Internet Explorer и Mozilla Firefox обрабатывают запрос на получение привилегий по-разному, поэтому нам необходимо найти способы, позволяющие работать с ними обоими.

Программа, рассматриваемая в качестве примера, устанавливает взаимодействие с одной из Web-служб Google, используя для этой цели SOAP (Simple Object Access Protocol). Протокол SOAP базируется на HTTP. В составе запроса серверу передается XML-документ, содержащий значения параметров, а ответом сервера является XML-документ, описывающий результаты. XML-документ, пересылаемый по протоколу SOAP, имеет достаточно простую структуру и состоит из заголовков и содержимого, помещенных в своеобразный **"конверт".** Использование формата XML упрощает применение объекта XMLHttpRequest для работы с данным протоколом.

Для работы со своей поисковой машиной Google предоставляет интерфейс SOAP. Пользователь может передавать ключевые слова в составе запроса и получать в ответ XML-документ, содержащий результаты. Ответ в формате XML очень похож на данные, отображаемые Google на странице поиска. Каждый пункт набора результатов содержит заголовок, фрагмент текста, описание и URL. В ответе также указывается приблизительное число документов, удовлетворяющих условиям поиска.

Наше приложение представляет собой своеобразную игру с применением Интернета. В ней будет использоваться информация о количестве документов, полученных в результате поиска. Пользователю предлагается простая форма и случайным образом сгенерированное большое число (рис. 7.4). Задача пользователя — задать ключевые слова, при обработке которых сервером Google будет получено количество результатов, отличающихся от заданного числа не более, чем на 1000.

Мы будем взаимодействовать с SOAP-службой Google посредством XMLHttpRequest в составе объекта ContentLoader (его мы разработали в главе 3). Последний раз мы обращались к данному объекту в главе 6, где оснастили его некоторыми новыми возможностями. Если мы используем последнюю версию ContentLoader для взаимодействия с Google, то получим желаемые результаты при работе с браузером Internet Explorer, но не с Mozilla. Рассмотрим вкратце особенности поведения каждого браузера.

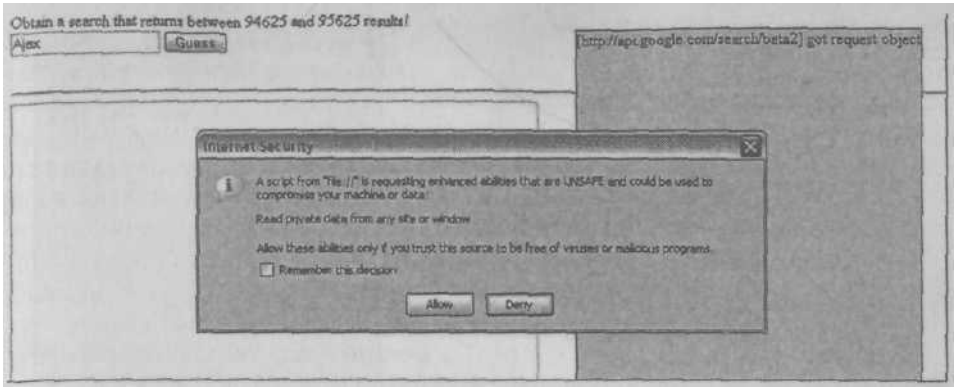


Рис. 7.4. Использование API SOAP, предоставляемого Google, в простом Ajax-приложении, Пользователь вводит ключевые слова для поиска. Число результатов, предоставляемых Google, должно укладываться в заранее заданный диапазон значений

Браузер Internet Explorer и Web-службы

Как было сказано ранее, система защиты Internet Explorer базируется на понятии зон безопасности. Если мы скопируем наше приложение с Web-сервера, даже локального, оно будет рассматриваться как представляющее опасность для системы. Когда мы в первый раз обратимся *к* Google, используя ContentLoader, то получим сообщение, подобное тому, которое было представлено на рис. 7.2. Если пользователь щелкнет на кнопке Yes, передача данного запроса, а также всех последующих запросов к тому же серверу будет разрешена. Если пользователь щелкнет на кнопке No, запрос будет отвергнут и управление получит обработчик ошибок ContentLoader. В данном случае обеспечивается средний уровень защиты, а пользователь испытывает небольшие неудобства в работе.

Как вы помните, при запуске Ajax-клиента из локальной файловой системы браузер Internet Explorer рассматривает эту среду как безопасную, и диалоговое окно не отображается.

Браузеры Mozilla, в том числе Firefox, используют более строгий подход к обеспечению защиты, и получить требуемые права сложнее.

Mozilla PrivilegeManager

Модель защиты браузера Mozilla основана на понятии привилегий. Считается, что каждое действие, независимо от того, является ли оно обращением к стороннему Web-серверу или чтением файлов из локальной файловой системы, может представлять опасность для системы. Чтобы выполнить действие, код приложения должен запросить соответствующие привилегии. Привилегиями управляет объект netscape.security.PrivilegeManager. Если мы хотим, чтобы клиентская программа Ajax взаимодействовала с сервером Google, нам надо сначала организовать обращение к PrivilegeManager. Firefox может быть настроен так, что PrivilegeManager даже не будет принимать обращения, причем для данных, обслуживаемых сервером, подобные

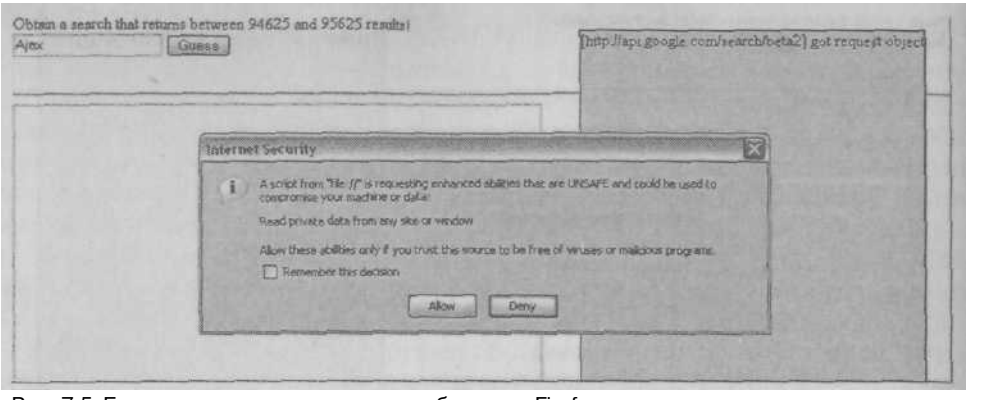


Рис. 7.5. Если приложение запрашивает у браузера Firefox дополнительные привилегии, на экране отображается окно с предупреждающим сообщением

установки принимаются по умолчанию. Таким образом, подход, рассматриваемый в данном разделе, в основном подходит для использования в сетях intranet. Поэтому последующий материал ориентирован в основном на разработчиков, которые создают программное обеспечение для внутренних сетей предприятия, а также на тех, кого интересуют особенности работы Firefox.

Запросить привилегии позволяет метод `enablePrivilege`. При обращении к нему выполнение сценария приостанавливается и на экране отображается диалоговое окно, показанное на рис. 7.5.

В окне выводится предупреждение о том, что сценарий собирается выполнить действие, которое потенциально может быть опасным. Пользователь может разрешить или запретить предоставление привилегий. В любом случае выполнение сценария возобновляется. Если привилегии получены, программа может выполнить требуемые действия. В противном случае она лишь пытается предпринять их, но в результате возникает ошибка.

Как вы уже знаете, Internet Explorer запоминает первое решение пользователя и, единожды отобразив предупреждающее сообщение, больше не повторяет его. Браузер Mozilla предоставляет привилегии только на время действия запросившей их функции, и, если пользователь не установит флажок опции Remember my decision, действия программы будут прерываться каждый раз, когда ей потребуются привилегии. {Как вы увидите, это может произойти дважды в течение одного запроса по сети.) Очевидно, что в данном случае требования безопасности и применимости противоречат друг другу.

Вернемся к объекту ContentLoader (мы уже обсуждали его в главах 3, 5 и 6). Постараемся ответить на вопрос, что надо сделать, чтобы передать запрос серверу Google. В модифицированном коде предусмотрены запросы к объекту PrivilegeManager (листинг 7.1). Мы также реализовали возможность формирования произвольных HTTP-заголовков. Это необходимо для создания сообщений SOAP.

Листинг 7.1. ContentLoader, взаимодействующий со средствами защиты

```
net.ContentLoader=function(
  url,onload,onerror,method,params,'contentType,
 headers,secure
){
  this.req=null;
  this.onload=onload;
  this,onerror=(onerror) ? onerror :
  this.defaultError;
   this.secure=secure;
  this.loadXMLDoc(url,method,params,contentType,  headers);
}
net.ContentLoader.prototype={
  loadXMLDoc:function(url,method,params,contentType,headers){
    if (!method){
      method="GET";
    }
    if {!contentType && method=="POST"){
      contentType^'application/x-www-foim-urlencoded';
    }
    if (window.XMLHttpRequest){
      this.req=new XMLHttpRequest();
    } else if (window.ActiveXObject){
      this.req=new ActiveXObject("Microsoft.XMLHTTP");
    }
     if (this.req){
       try{
        try{
          if (this.secure && netscape
            &6 netscape.security,PrivilegeManager.enablePrivilege) {
// O Получение привилегий для передачи запроса
            netscape.security.PrivilegeManager
              .enablePrivilege('UniversalBrowserRead');
          }
        }catch (err)U
        this.req.open(method,url,true);
        if (contentType){
          this.req.setRequestHeader('Content-Type', contentType);
        }
 // @ Добавление поля заголовка HTTP
        if (headers){
          for (var h in headers){
            this.req.setRequestHeader(h,headers[h]);
          }
        }
        var loader=this;
        this.req.onreadystatechange=function(){
          loader.onReadyState.call(loader);
        }
        this.req.send(params);
      }catch (err){
        this.onerror.call(this);
      1
    }
Ь
```

```
onReadyState:function(){
  var req=this.req;
  var ready=req.readyState;
  i f (ready==net.READY_STATE_COMPLETE){
    var httpStatus=req.status;
    if (httpStatus==200 I I httpStatus==O){
      try{
        if (this.secure && netscape
          && netscape.security.PrivilegeManager.enablePrivilege) {
          // Получение привилегий для раэбора ответа
netscape.security.PrivilegeManager
.enablePrivilege('UniversalBrowserRead');
        }
      }catch (err){}
      this.onload.call(this);
    }else{
      this.onerror.call(this);
    }
  }
b
  defaultError:function(){
    alert("error fetching data!"
      +"\n\nreadyState:"+this.req.readyState
      +"\nstatus: "+this.req.status
      +"\nheaders: "+this.req.getAHResponseHeaders());
  }
_J
```

К конструктору добавлены *дополнительные* параметры. Первый параметр представляет собой массив полей заголовка HTTP 0. Эти поля нам придется использовать при формировании запроса SOAP. Второй дополнительный параметр — логическое значение, выполняющее роль флага, который показывает, следует ли запрашивать привилегии на определенных этапах выполнения программы.

Когда мы запрашиваем привилегии, обращаясь к объекту netscape. PrivilegeManager, мы получаем их лишь на время выполнения текущей функции. Следовательно, нам надо запрашивать привилегии дважды: для обращения к удаленному серверу О и для чтения полученного ответа @. Функцию обработчика onload мы вызываем в области видимости функции onReadyState (), поэтому полученные привилегии будут действовать и для нее.

Браузер Internet Explorer не поддерживает PrivilegeManager, и при обращении к данному объекту будет сгенерировано исключение. Поэтому мы помещаем обращения в блок try.. .catch, в результате чего исключение будет перехвачено и должным образом обработано. Если приведенный выше код будет выполнен в среде Internet Explorer, в блоке try.. . catch возникнет ошибка и программа не будет завершена аварийно. Под управлением браузера Mozilla работа с PrivilegeManager не вызовет *исключения*.

Воспользуемся модифицированным объектом ContentLoader для передачи запроса Google. В листинге 7.2 приведен HTML-код, обеспечивающий работу приложения.

Листинг 7.2. Содержимое файла googleSoap.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
<title>Google Guessing</title>
  <script type="text/javascript" src='net_secure.js'></script>
  <script type="text/javascript" src='googleSoap.js'X/script>
  <script type='text/javascript'>
    var googleKey=null;
    var guessRange = 1000;
    var intNum = Math.round(Math.random()
      * Math.pow(10,Math.round(Math.random()*8)));
    window.onload = function(){
      document.getElementById("spanNumber")
        .innerHTML = intNum + " and "
        + (intNum + guessRange);
    }

</script>
</head>
<body>
  <form name="Forml" ons\ibmit="submitGuessO;return false;">
  Obtain a search that returns between 
  <span id="spanNumber"></span>  results!<br/>
  <input type="text" name="yourGuess" value="Ajax">
    <input type="submit" name="bl" value="Guess"/><br/xbr/>
    <span id="spanResults"></span>
  </form>
  <hr/>
  <textarea rows='24' cols='100' id='details'X/textarea>
</body>
</html>
```

В состав HTML-файла мы включаем элементы формы и код для вычисления достаточно большого псевдослучайного числа. Мы также объявляем переменную googleKey. Это ключ, позволяющий использовать API Google SOAP. Мы не приводим здесь реального ключа, так как это противоречило бы лицензионным соглашениям. Ключи предоставляются бесплатно и дают возможность выполнять в течение дня ограниченное число операций поиска. Процедура получения ключа достаточно проста. Для этого надо обратиться по URL, приведенному в конце данной главы.

Передача запроса

Основная часть работы выполняется функцией submitGuess (), которая вызывается при активизации кнопки sumit. Эта функция определена во включаемом JavaScript-файле. JavaScript-код этой функции, осуществляющий обращение к API Google, приведен в листинге 7.3.

Листинг 7.3. Функция submitGuess()

```
function submitGuess(){
// О Проверка ключа лицензирования
  if (igoogleKey){
    alert("You will need to get a license key "
      +"from Google,\n and insert it into"
      +"the script tag in the html file\n"
      +"before this example will run.");
    return null;
  }
  var myGuess=document.Forml.yourGuess.value;

// © Создание сообщения SOAP
  var strSoap='<?xml version="1.0" encoding="UTF-8"?>'
    +'\n\n<SOAP-ENV:Envelope'
    +' xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/" '
    +' xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"'
    +'xmlns :xsd="http://www. w3.org/1999/XMLScherna">
    +'<SOAP-ENV: BodyXns 1: doGoogleSearch'
    +' xmlns:nsl="urn:Google3earch"'
    +' SOAP-ENV:encodingStyle='
    +'"http://schemas.xmlsoap.org/soap/encoding/">'
    +'<key xsi:type="xsd:string">' + googleKey + '</key>'
    +'<q xsi:type="xsd:string">'+myGuess+'</q>'
    +'<start xsi:type="xsd:int">O</start>'
    +'<maxResults xsi:type="xsd:int">l</maxResults>'
    +'<filter xsi:type="xsd:boolean">true</filter>'
    +'<restrictxsi:type="xsd: string"X/restrict>'
    +'<safeSearch xsi:type="xsd:boolean">false</safeSearch>'
    +'<lr xsi:type="xsd:string"></lr>'
    +'<ie xsi:type="xsd:string">latinl</ie>'
    +'<oe xsi:type="xsd:string">latinl</oe>'
    +•</nsl:doGoogleSearch>'
    +'</SOAP-ENV:Body>'
    +'</SOAP-ENV:Envelope>';

// 0 Создание ContentLoader, в том числе
// предоставление URL Google API и
// передача дополнительных полей заголовка
  var loader=new net.ContentLoader(
    "http://api.google.com/search/beta2",
    parseGoogleResponse,
    googleErrorHandler,
    "POST",
    strSoap,
    "text/xml",
    {
      Man:"POST http://api.google.com/search/beta2 HTTP/1,1",
      MessageType:"CALL"
    },
    true
  );
}
```

Выполнение функции submitGuess () начинается с проверки наличия **ключа** лицензирования. Если ключ отсутствует, программа напоминает об этом пользователю О. Если вы скопируете код данного примера, значение ключа лицензирования будет установлено равным null, поэтому вам надо **цолучить** с сервера Google реальный ключ.

Следующая задача — создать SOAP-сообщение © большого объема, содержащее ключевые слова поиска и ключ лицензирования. Разработчики SOAP предполагали, что все основные действия будут автоматизированы, поэтому, создавая XML-файл вручную, мы отступаем от основных принципов работы со службами. Браузеры Internet Explorer и Mozilla предоставляют объекты, упрощающие взаимодействие посредством SOAP. Однако на наш взгляд, будет полезно хотя бы раз вручную сформировать SOAP-запрос и разобрать ответ.

Имея готовый XML-текст, мы создаем объект ContentLoader ©, при этом задаем SOAP XML в качестве содержимого HTTP-запроса, а также указываем URL API Google и поля HTTP-заголовка. В качестве типа содержимого указывается значение text/xml. Заметьте, что MIME-тип определяет тип данных в теле сообщения, а не тип информации, которую мы ожидаем получить в ответе (в данном случае эти типы совпадают). Последний параметр — значение true, которое указывает на то, что мы должны запрашивать необходимые права с помощью объекта PrivilegeManager.

Разбор ответа

Объект ContentLoader создает запрос, в ответ на который (с согласия пользователя) будет получен большой объем XML-данных. Ниже приведен фрагмент ответа, представляющего собой результаты поиска по ключевому слову Ajax.

```
<?xml version='1.O' encoding='utf-8'?>
<soap~env:envelope
  xmlns:soap-env="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/1999/xmlschema-instance"
  xmlns:xsd="http://www.w3.org/1999/xmlschema">
<soap-env:body>
  <nsl:dogooglesearchresponse xmlns:nsl~"urn:googlesearch"
    soap-env:encodingstyle="http://schemas.xmlsoap.org/soap/encoding/";
<return xsi:type="nsl:googlesearchresult">
<directorycategories
  xmlns:ns2="http://schemas.xmlsoap.org/soap/encoding/"
  xsi:type="ns2:array"
  ns2:arraytype="nsl:directorycategory[1]">

<estimateisexact xsi:type="xsd:boolean">false</estimateisexact>
<estimatedtotalresultscount
xsi:type="xsd:int">741000</estimatedtotalresultscount>

<hostname xsi:type="xsd:string"></hostname>
<relatedinformationpresent xsi:type="xsd:boolean">true
    </relatedinformationpresent>
<snippet xsi:type="xsd:string">de officinle site van afc
```

```
    &lt;b&gt;ajax&lt;/b&gt;.</snippet>
<summary xsi:type="xsd:string">official club site,
  including roster, history, wallpapers, and Video clips.
    &lt;brsgt; [english/dutch]</summary>
<title xsi:type="xsd:string">
&lt;b&gt;ajax&lt;/b&gt;.nl - splashpagina
</title>
```

Реальный SOAP-ответ содержит гораздо больший объем информации. В первой части ответа определены некоторые заголовки, имеющие отношение к доставке данных, в частности, там указан источник, из которого поступил ответ. В теле документа присутствуют элементы, описывающие объем набора результатов. В данном случае количество результатов приблизительно оценено как 741000. Также мы видим часть первого результата поиска — ссылку на страницу голландского футбольного клуба. В листинге 7.4 показан обработчик обратного вызова, с помощью которого мы осуществляем разбор результатов.

Листинг 7.4. Функция parseGoogleResponse ()

```
function parseGoogleResponse(){
  var doc=this.req.responseText.toLowerCase() ;
  document.getElementBy!d( 'details').value=doc;
  var startTag='<estimatedtotalresultscount xsi:type="xsd:int">';
  var endTag='</estimatedtotalresultscount>';
  var spotl=doc.indexOf(startTag);
  var spot2=doc.indexOf(endTag);
  var strTotall^doc.substring(spot1+startTag.length,spot2);
  var totall=parselnt(strTotall);
  var strOut^"";
  if(totall><intNum && totall<=intNum+guessRange){
    strOut+="You guessed right!";
  }else{
    strOut+="WRONG! Try again!";
  }
  strOut+="<br/>Your search for <strong>"
        +document.Forml.yourGues s.value
        +"</strong> returned " + strTotall + " results!";
  document.getElementByldf"spanResults").innerHTML = strOut;
```

В данном случае нас интересует не структура SOAP-сообщения, а лишь предполагаемое количество результатов. Ответ представляет собой корректный XML-документ, и мы могли бы использовать для его разбора свойство responseXML объекта XMLHttpRequest. Однако мы пойдем по пути наименьшего сопротивления и извлечем число результатов путем преобразования строки. Затем, выполняя операции со структурой DOM, мы сообщим пользователю о том, насколько удачно он подобрал ключевые слова. Для тех, кто хочет более подробно разобраться с данными SOAP, мы выведем весь XML-ответ в текстовой области.

Обеспечение доступа к PrivilegeManager в Firefox

Как было сказано ранее, PrivilegeManager может быть сконфигурирован так, что он не будет реагировать на запросы из программы. Для того чтобы выяснить, выполнены ли в браузере Firefox подобные установки, надо ввести в строке, предназначенной для указания адреса, about :config. В ответ будет выведена информация о текущей настройке. Используя поле редактирования, выполняющее фильтрацию, найдем пункт signed.applets.codebase_principal_support. Если значение данного свойства равно true, рассмотренный ранее код будет работать. В противном случае нам не удастся организовать взаимодействие с Google.

В ранних версиях Mozilla установки должны были выполняться вручную, после чего необходимо было перезапустить браузер. В Firefox, дважды щелкнув на соответствующей строке списка установок, можно заменить значение true на false и обратно. Изменения, вносимые таким образом, вступают в действие немедленно; при этом не только не требуется перезапускать браузер, но, если параметры отображаются на отдельной вкладке, не надо даже обновлять страницу.

Подписание клиентского кода для выполнения в среде Mozilla

Поскольку Internet Explorer не использует PrivilegeManager, многократного подтверждать правомочность действий программы не приходится. В среде Mozilla дважды отображается диалоговое окно, что создает определенные неудобства для пользователя (конечно, окно выводится только в случае, если конфигурация браузера позволяет использовать PrivilegeManager). Повторного отображения диалогового окна можно избежать, установив флажок опции Remember my decision (рис. 7.5), но мы, как разработчики, лишены возможности сделать это из программы.

Существует решение этой проблемы, но оно предполагает специальное оформление приложения. Web-приложение может быть подписано с помощью сертификата. Чтобы сделать это, нам надо представить приложение браузеру в виде JAR-файла — сжатого архива, содержащего все сценарии, HTML-страницы, изображения и другие ресурсы. Подготовленный JAR-файл можно подписать, используя сертификат, полученный у Thawte, VeriSign или Другой компании, занимающейся подобной деятельностью. Для обращения к ресурсам, содержащимся в подписанном JAR-файле. используются специальные выражения, подобные представленному ниже.

```
jar:http://myserver/mySignedJar.jar|/path/to/someWebPage.html
```

Когда пользователь копирует подписанное Web-приложение, он должен Лишь один раз подтвердить предоставление привилегий: повторно ему подобные вопросы не задаются.

Браузер Mozilla предоставляет бесплатные инструментальные средства Для подписания JAR-файлов. Те пользователи, которые собираются лишь поэкспериментировать с данной технологией, могут сгенерировать неаутен-Тифицируемые сертификаты. Для этой цели предназначена утилита keytool, поставляемая в составе Java Development Kit (JDK). Мы же рекомендуем

обзавестись настоящим сертификатом, который можно будет использовать в практических целях.

Подписанные JAR-файлы не являются переносимыми и могут работать только с браузерами Mozilla. По этой причине мы не будем подробно рассматривать процедуру подписания кода. Те, кого интересует данный вопрос могут воспользоваться информацией из источников, ссылки на которые приведены в конце данной главы.

На этом мы завершаем обсуждение вопросов взаимодействия приложений Ajax с удаленными службами. Мы выяснили, как приложение, выполняющееся в среде браузера, может обмениваться данными с сервером, с которого оно было скопировано, а при необходимости и со сторонними серверами. Программы, полученные с сервера, вряд ли смогут повредить вашей системе. Однако остается еще одна опасность, связанная с обменом информацией, особенно конфиденциальной. В следующем разделе мы поговорим о том, как скрыть данные пользователя от постороннего взгляда.

7.3. Защита конфиденциальной информации

Компьютер пользователя, на котором работает браузер, как правило, не имеет непосредственного соединения с сервером. Данные, передаваемые на сервер, проходят через промежуточные узлы (маршрутизаторы и прокси-серверы). Обычные HTTP-данные представляют собой незашифрованный текст, который можно прочитать в любой точке по пути следования пакета. Таким образом, информация может стать доступна любому, кто имеет контроль над промежуточными узлами.

7.3.1. Вмешательство в процесс передачи данных

Предположим, что вы написали приложение Ajax, которое передает по Интернету финансовую информацию, например, сведения о банковских счетах или номера платежных карточек. Если маршрутизатор работает корректно, он передает пакеты в неизменном виде и не читает их содержимое. Он лишь извлекает из заголовков данные, необходимые для выбора маршрута. Если же маршрутизатор контролируется злоумышленниками (рис. 7.6), он может собирать содержащиеся в пакетах сведения, выявляя, например, номера платежных карточек или почтовые адреса для рассылки сиама. Он даже может изменять маршрут так, чтобы пакет попал на другой сервер, или модифицировать передаваемые данные (чтобы, например, положить деньги на другой счет).

В Ajax-приложениях протокол HTTP используется как для копирования клиентского кода, так и для передачи запросов клиента серверу. Все способы взаимодействия, которые мы рассматривали до сих пор. — скрытые элементы iFrame, HTML-формы, объекты XMLHttpRequest — с этой точки зрения одинаковы. Любое Web-приложение, в том числе и инфраструктура Ajax, имеет ряд уязвимых мест, которыми могут воспользоваться злоумышленники. Атака путем перехвата передаваемых данных на промежуточном узле носит

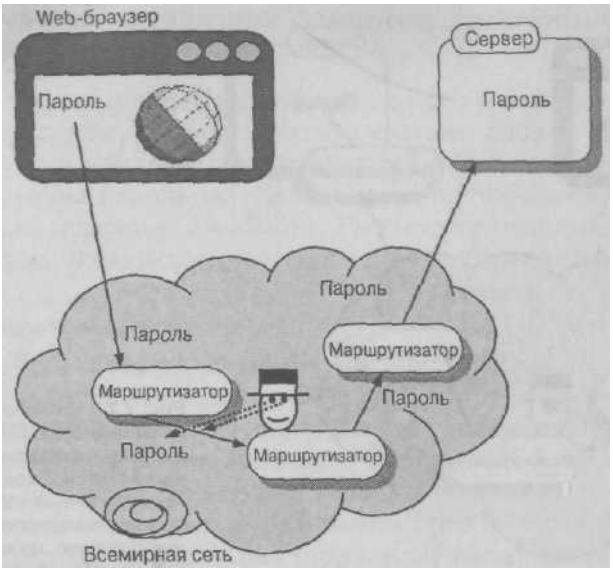


Рис. 7.6. При обычном HTTP-взаимодействии данные передаются по глобальной сети в виде незашифрованного текста, что позволяет злоумышленникам, контролирующим промежуточные узлы, прочитать информацию и даже изменить ее

название "человек посередине". Рассмотрим меры, которые можно принять для противодействия подобным атакам.

7.3.2. Организация защищенного HTTP-взаимодействия

Если вам необходимо защитить трафик между клиентской программой Ajax и сервером, самая очевидная мера, которую вы можете предпринять, — кодировать данные, используя защищенное соединение. Hypertext Transfer Protocol на базе Secure Socket Layer (HTTPS) реализует оболочку для обычного HTTP. Кодирование данных, передаваемых в обоих направлениях, осуществляется посредством пары ключей (открытого и закрытого). "Человеку посередине" по-прежнему доступно содержимое пакетов, но оно закодировано, и извлечь выгоду из данной информации невозможно (рис. 7.7).

Для того чтобы протокол HTTPS можно было применить на стороне браузера и на стороне сервера, необходима поддержка платформенно-ориентированного кода. Современные браузеры содержат встроенные средства для работы с HTTPS, и многие компании, предоставляющие на своих серверах пространство для Web-узлов, также предлагают защищенные соединения. При поддержке протокола HTTPS выполняются интенсивные вычисления, поэтому решение формировать передаваемые двоичные данные в программе на JavaScript вряд ли можно считать удачным. Мы не ставим задачу повторно реализовать с помощью JavaScript DOM, CSS или HTTP, поэтому

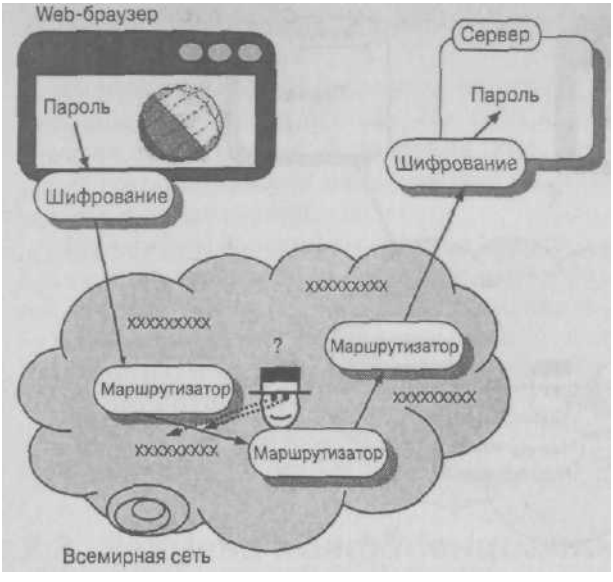


Рис. 7.7. При использовании защищенного HTTP-соединения данные, передаваемые в обоих направлениях, кодируются. На промежуточных узлах можно прочитать кодированную информацию, но ключ, требуемый для ее расшифровки, недоступен

HTTPS лучше всего рассматривать как используемую нами службу, а не как нечто, требующее переопределения.

Против использования HTTPS можно выдвинуть ряд аргументов. Во-первых, для кодирования и декодирования требуется большой объем вычислительных ресурсов. Это не создает проблему на стороне клиента, так как клиентская программа должна обрабатывать один поток данных. На сервере же дополнительная нагрузка крайне нежелательна. В особенности это важно на больших узлах. В классических Web-приложениях принято передавать посредством HTTPS только критичные данные, а обычное содержимое, например изображения или дескрипторы, формирующее общую структуру документа, пересылается посредством протокола HTTP. В Ajax-приложении необходимо учитывать влияние модели безопасности JavaScript, согласно которой `http://` и `https://` считаются различными протоколами. Во-вторых, HTTPS защищает только сам процесс передачи данных, но не обеспечивает безопасность приложения в целом. Если вы передадите по защищенному каналу номер платежной карточки, а затем поместите его в базу данных, в системе защиты которой имеются недостатки, ценная информация вполне может быть похищена.

Тем не менее HTTPS можно рекомендовать для передачи важных данных по сети. Стоимость такой передачи высока, и ее не всегда можно реализовать на небольших узлах. Если требования к защите не столь критичны, можно использовать обычный протокол HTTP для передачи шифрованных данных.

7,3-3. *Передача шифрованных данных в ходе обычного HTTP-взаимодействия*

Предположим, что вы поддерживаете небольшой Web-узел и при его работе не возникает потребности передавать секретные данные по защищенному каналу. Однако пользователи регистрируются на вашем узле, и вы хотите принять меры, чтобы пароль не попал в руки постороннему. В этой ситуации вам придет на помощью JavaScript. Рассмотрим основные принципы, на которых базируется возможное решение, а затем попробуем его реализовать.

Открытые и закрытые ключи

Если нам надо организовать передачу пароля, мы можем пересылать его в зашифрованном виде. Алгоритм шифрования преобразует входную строку в последовательность, с виду напоминающую случайный набор знаков. В качестве алгоритма кодирования можно выбрать MD5. Ряд возможностей, обеспечиваемых данным алгоритмом, позволяет применять его для обеспечения безопасности. Во-первых, в результате преобразования одного и того же фрагмента данных каждый раз будут получены одинаковые результаты. Во-вторых, вероятность того, что при преобразовании двух различных фрагментов будут получены одинаковые выходные данные, исчезающе мала. Благодаря этим особенностям данные, полученные в результате применения алгоритма, или МП5-дайджест, могут служить достаточно надежным идентификатором ресурсов. Есть и третья особенность. Она состоит в том, что алгоритма обратного преобразования не существует. Поэтому MD5-дайджест может пересылаться по сети в открытом виде. Если злоумышленник и перехватит его, он не сможет восстановить исходное сообщение.

Например, в результате преобразования строки Ajax in action будет сгенерирован MD5-дайджест 8bd04bbe6ad2709075458c03b6ed6c5a. Выходные данные будут неизменными при каждом преобразовании. Мы можем закодировать строку на стороне клиента и передавать ее в таком виде серверу. Сервер извлечет пароль из базы, закодирует его, используя тот же алгоритм, и сравнит две строки. Если они совпадут, регистрация будет считаться успешной. Заметьте, что пароль в исходном виде по сети не передается.

При регистрации на сервере MDS-дайджест нельзя непосредственно передавать по Интернету. Злоумышленник не может выяснить, на основе какой исходной строки был сгенерирован дайджест, но может использовать сам дайджест в процессе регистрации. Здесь на помощь приходит механизм открытого и закрытого ключа. Вместо того чтобы кодировать один пароль, мы зашифруем его вместе с псевдослучайной последовательностью символов, предоставленной сервером. Сервер при каждой регистрации генерирует новую псевдослучайную последовательность и передает ее по сети клиенту.

На уровне клиента пользователь вводит пароль, а мы присоединяем к нему строку, предоставленную сервером, и шифруем результат конкатенации. Переданная клиенту последовательность символов хранится на сервере. Получив от клиента идентификатор пользователя, сервер читает из базы пароль, присоединяет к нему запомненную последовательность симво-

лов, шифрует их и сравнивает результаты. При совпадении закодированных последовательностей регистрация считается успешной. В противном случае (например, если пользователь неправильно ввел пароль) процедура регистрации повторяется, но при этом уже используется другая последовательность символов.

Предположим, что сервер передал строку abed. MD 5-дайджест для последовательности символов Ajax in actionabed имеет вид e992dc25b<373842023 f06a61f03f3787. При следующей попытке регистрации клиенту будет передана строка wxyz. Присоединив ее к паролю и преобразовав результат, мы получим Ш)5-дайджест 3f2da3b3ee2795806793c56bf00a8b94. Злоумышленник видит каждое сообщение, может сохранить его, но не способен зарегистрироваться посредством имеющейся информации. Таким образом, несмотря на возможность перехвата сообщений, воспроизвести последовательность действий, необходимую для успешной регистрации, не удастся.

Псевдослучайная строка выполняет роль открытого ключа и доступна для всех. Пароль можно считать закрытым ключом. Он хранится в течение длительного времени и не должен быть доступен посторонним.

Реализация процедуры регистрации средствами JavaScript

Для реализации описанного решения необходимо, чтобы и на стороне клиента, и на стороне сервера присутствовал MD5-генератор. Для клиентских программ существует свободно распространяемая библиотека генерации, реализованная на JavaScript Полом Джонстоном (Paul Johnston). Для того чтобы воспользоваться возможностями этой библиотеки, надо лишь включить ее в состав программы и вызвать соответствующую функцию.

```
<script type='text/javascript' src='md5.js'X/script>
<script type='text/javascript'>var encrypted=str_md5('Ajax');
  // Выполнение требуемых действий ...
</script>
```

Для сервера алгоритм MD5 также поддерживается в ряде библиотек. В PHP, начиная с версии 3, содержится встроенная функция md5(). В классе Java, security.MessageDigest реализованы популярные алгоритмы шифрования, в том числе MD5. В .NET Framework доступен класс System .Security.Cryptography.MD5.

Данный подход имеет ограниченную область применения, так как, для того, чтобы выполнять сравнение кодированных строк, серверу должен быть заранее известен пароль. Таким образом, рассмотренный механизм является хорошей альтернативой передаче ресурсов по протоколу HTTPS, но он не может полностью заменить этот протокол.

Итак, на данный момент мы получили следующие результаты. Политика "сервера-источника" позволяет противодействовать передаче на клиентскую машину кода, способного повредить систему. Нам известны меры борьбы с атакой типа "человек посередине" при обмене данными клиента с сервером. В последнем разделе мы сосредоточим внимание на сервере, который также может стать объектом атаки. Выясним, как можно защитить Web-службы от несанкционированного доступа.

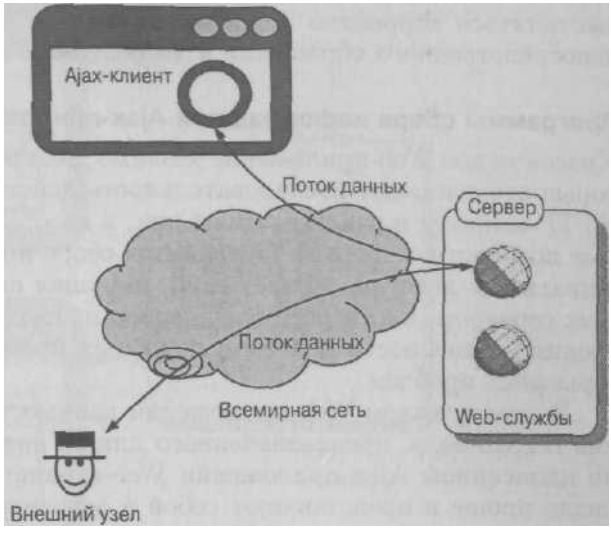


Рис. 7.8. В рамках инфраструктуры Ajax сервер предлагает свои услуги по Интернету Доступ к серверу осуществляется по стандартному протоколу, чаще всего HTTP. Ajax-клиент получает потоки данных с сервера. Вследствие того что Web-службы общедоступны, внешние узлы сети могут получать данные непосредственно, минуя клиента

7.4. Управление доступом к потокам данных Ajax

Рассмотрим стандартную архитектуру Ajax с точки зрения наличия уязвимых мест в защите. Клиент, выполняющийся в среде браузера, передает запросы серверу, используя протокол HTTP. Эти запросы обслуживаются процессами на стороне сервера (сервлетами, динамическими документами и программами других типов), которые возвращают данные клиенту. Такое взаимодействие условно показано на рис. 7.8.

Услуги сервера или документы становятся доступными внешним узлам автоматически вследствие самой природы Интернета. В некоторых случаях мы даже поощряем подобное взаимодействие с сервером, в частности, публикуем API таких служб, как eBay, Amazon и Google. Однако и в этом случае нельзя упускать из виду вопросы защиты. Меры, которые мы можем предпринять для обеспечения защиты, описаны в следующих двух разделах. Во-первых, мы можем создать интерфейс Web-служб, или API, таким образом, чтобы им невозможно было воспользоваться некорректно, например, заказать товары, не заплатив за них. Во-вторых, имеется возможность ограничить доступ к Web-службам.

7.4.1. Создание защищенных программ на уровне сервера

Говоря о Web-приложении, мы обычно имеем в виду некоторую последовательность выполняемых действий. Например, при работе с интерактивным магазином пользователь просматривает имеющиеся товары и помещает некоторые из них в **"корзинку".** Процесс выбора товаров четко определен: указывается адрес, по которому должны быть доставлены товары, способ оплаты и порядок подтверждения заказа. До тех пор, пока пользователь работает

с приложением, мы можем быть уверены, что взаимодействие будет осуществляться корректно. Если же некоторый узел сети предпримет попытку непосредственного обращения к Web-службе — возможны проблемы.

Программы сбора информации и Ajax-приложение

Классическое Web-приложение уязвимо для программ сбора информации, которые отслеживают последовательность действий пользователя, анализируя HTTP-запросы и извлекая те из них, в которых содержатся данные, введенные посредством формы. Программы сбора информации могут активно вмешиваться в действия приложения, нарушая порядок событий, обрабатываемых сервером, или перегружая сервер повторяющимися запросами. С точки зрения безопасности действия подобных программ могут стать источником серьезных проблем.

В классическом Web-приложении данные часто теряются в большом объеме HTML-кода, предназначенного для их внешнего оформления. В грамотно написанном Ajax-приложении Web-страницы, передаваемые клиенту, гораздо проще и представляют собой в основном структурированные данные. Разделение логики и представления является признаком профессионального подхода к написанию программ, но оно же упрощает работу программы сбора данных, поскольку информация, передаваемая сервером, предназначена в основном для разбора, а не для отображения посредством браузера. Когда внешний вид документа изменяется, программы сбора данных начинают работать менее надежно и могут даже вовсе выйти из строя. В Ajax-приложении правила обращения клиента к Web-службе меняются редко. Чтобы обеспечить целостность приложения, нам надо тщательно продумывать структуру высокоуровневого API, используемого при взаимодействии клиента и сервера. В данном случае под API мы понимаем не HTTP, SOAP или XML, а динамические страницы и параметры, передаваемые им.

Интерактивная игра "морской бой"

Чтобы лучше разобраться в том, как структура API Web-службы влияет на безопасность приложения, рассмотрим простой пример. Создадим интерактивную версию известной всем игра "морской бой". В процессе игры Ajax-клиент будет взаимодействовать с сервером, используя Web-службы. Нам надо принять меры для того, чтобы правила игры выполнялись даже в том случае, когда один из игроков попытается мошенничать: захватит контроль над клиентом, чтобы передавать данные серверу вне очереди.

Задача игрока — угадать расположение кораблей противника. Игра состоит из двух этапов. Сначала игроки расставляют свои корабли на игровом поле. Затем они по очереди называют позиции, пытаясь попасть на расположение корабля противника. Копии игровых полей в процессе игры хранятся на сервере. Каждый клиент также поддерживает модель своего поля и поле другого игрока. Вначале второе поле пустое, но по мере угадывания позиций оно заполняется (рис. 7.9).

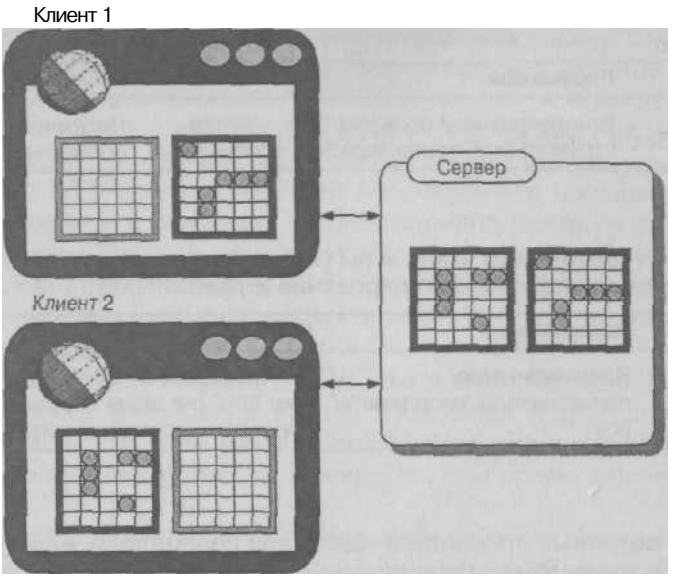


Рис. 7.9. Модели данных в Ajax-приложении, реализующем игру "морской бой" После расстановки кораблей на сервере располагаются модели игровых полей каждого участника. Вначале на стороне клиента содержится лишь модель своего игрового поля. Модель поля соперника формируется в процессе игры

Таблица 7.2. API для пошагового формирования игрового поля

URL	Параметры	Возвращаемые данные
clearBoard.do	Идентификатор пользователя	Подтверждение
positionShip.do	Идентификатор пользователя, длина корабля, координаты (x,y), ориентация (N.S.EwwW)	Подтверждение или сообщение об ошибке

Рассмотрим первый этап игры. Вначале игровое поле пустое. Затем пользователь размещает на нем условные изображения своих кораблей. Организовать вызов сервера в процессе расстановки кораблей можно двумя способами. Первый способ предполагает обращения для очистки поля и для постановки каждого корабля в конкретной позиции. В процессе начальных установок обращение к серверу произойдет несколько раз: один раз при очистке поля и по одному обращению для постановки каждого корабля. API, соответствующий данному подходу, показан в табл. 7.-2.

Второй способ предполагает очистку поля и установку всех кораблей в рамках одного обращения к серверу. В этом случае при настройке сервер вызывается лишь один раз. Соответствующий API показан в табл. 7.3.

Таблица 7.3. API, предполагающий формирование игрового поля в рамках одного запроса

URL	Параметры	Возвращаемые данные
setupBoard.do	Идентификатор пользователя, массив структур (x,y, длина корабля, ориентация)	Подтверждение или сообщение об ошибке

Таблица 7.4. API для второго этапа игры (используется независимо от того, какой подход был применен при конфигурировании игрового поля)

URL	**Параметры**	**Возвращаемые** данные
guessPosition.do	Идентификатор пользователя, координаты (x,y)	"hit" (попадание) "miss" (промах) или "not your turn" (не ваша очередь) плюс обновление с учетом хода другого игрока

Подходы, подобные описанным здесь, мы сравнивали между собой в главе 5 при обсуждении SOA. Однократное обращение по сети оказывается более эффективным и лучше обеспечивает разделение уровней. Кроме того, оно упрощает задачу обеспечения безопасности.

В рамках первого подхода клиент отвечает за контроль правильности количества и типов установленных кораблей. Модель, поддерживаемая на сервере, должна обеспечить проверку правильности конфигурации в конце настройки. Для второго подхода такая проверка может быть предусмотрена в формате документа, используемого при обращении к серверу.

По окончании настройки в действие вступает служба, обеспечивающая очередность ходов участников игры. Исходя из правил игры каждое обращение клиента должно соответствовать одному ходу — попытке угадать клетку, занимаемую каким-либо кораблем. Соответствующий API описан в табл. 7.4.

Если игра проходит корректно, оба пользователя по очереди обращаются к URL guessPosition.do. Сервер следит за очередностью ходов и в случае ее нарушения возвращает нарушителю сообщение not your turn.

Предположим теперь, что один из участников пытается играть нечестно. В его распоряжении клиентская программа, которая может обращаться к API Web-службы в любой момент. Что он может предпринять, чтобы получить преимущество в игре? Дополнительный ход сделать невозможно, так как сервер отслеживает эту ситуацию — очередность ходов заложена в API-

Единственный способ мошенничества — изменение положения корабля после окончания настройки. В рамках подхода, предполагающего многократные обращения к серверу, можно попытаться вызвать positionShip.do в процессе игры. Если серверная часть кода разработана грамотно, сервер определит, что это обращение противоречит правилам, и откажется обслуживать запрос. Однако, делая подобную попытку, игрок ничего не теряет, а задача разработчика серверной программы — предусмотреть подобную возможность и принять соответствующие меры.

Бели в процессе настройки разрешено лишь одно обращение к серверу, изменить позицию одного корабля невозможно без очистки всего игрового поля. Таким образом, изменить настройку в свою пользу в принципе невозможно.

Настройка, допускающая единственное обращение к серверу, ограничивает свободу действий нечестного участника, не затрагивая при этом интересы игрока, соблюдающего правила. Если модель на стороне сервера функционирует правильно, API, предполагающий многократные обращения к серверу, не должен предоставить возможность несанкционированного вмешательства, однако число точек входа, в которых такое вмешательство оказывается потенциально возможным, значительно больше. В результате разработчику серверной программы приходится выполнять намного больший объем работы по проверке безопасности.

В разделе 5.3.4 мы говорили о том, что для упрощения API желательно применять образ разработки Fagade. Подобный подход желательно использовать и сейчас, на этот раз для повышения уровня безопасности. Чем меньше точек входа, доступных из Интернета, тем проще реализовать требуемую политику.

Структура API может в какой-то степени защитить приложение от нежелательного воздействия извне, однако некоторые точки входа должны существовать для того, чтобы клиентская программа Ajax могла взаимодействовать с сервером. В следующем разделе мы рассмотрим способы их защиты.

7.4.2. Ограничение доступа к данным из Web

В идеале хотелось бы разрешить доступ к динамическим данным, подготовленным для нашего приложения, Ajax-клиенту (и, возможно, другим авторизованным программам) и запретить его всем остальным. Некоторые технологии, обеспечивающие работу с богатыми клиентами, дают возможность использовать специальные протоколы, однако в Ajax-приложениях наш выбор ограничен HTTP. Как вы уже знаете, защищенный протокол HTTP позволяет скрыть передаваемые данные от постороннего взгляда, но он не позволяет определить, кто обращается по конкретному URL.

К счастью, протокол HTTP обладает большими потенциальными возможностями, а объект XMLHttpRequest обеспечивает дополнительный контроль над ним. Когда запрос поступает на сервер, серверная программа имеет доступ к полям заголовка, из которых можно извлечь информацию об источнике запроса.

Фильтрация HTTP-запросов

Для написания конкретных примеров мы будем использовать язык Java. Существуют и другие технологии, позволяющие добиться аналогичных результатов. При создании Web-приложения на Java мы можем определить объект `javax.servlet.Filter`, который будет перехватывать запросы. В подклассе `Filter` переопределим метод `doFilter()` так, чтобы он анализировал HTTP-запрос перед тем, как принять решение о его передаче по назначению. В листинге 7.5 показан код простого фильтра, который перехватывает запрос,

а после либо продолжает его обычную обработку, либо появляется страница! с сообщением об ошибке.

Листинг 7.5. Java-фильтр, используемый в системе защиты

```
public abstract class GenericSecurityFilter implements Filter {
  protected String rejectUrl=null;
  public void init(FilterConfig config)
  throws ServletException {
// О Указание URL, соответствующего отвергнутому запросу
    rejectUrl=config.getInitFarameter("rejectUrl"); }
  public void doFilter(
    ServletRequest request, ServletResponse response,
    FilterChain chain)
    throws IOException, ServletException {
// © Проверка допустимости запроса
    if (isValidRequest(request)){
      chain.doFilter{request, response);
    }else if {rejectUrl!=null){
// 0 Обращение к URL, соответствующему отвергнутому запросу
      RequestDispatcher dispatcher
       =request.getRequestDispatcher(rejectUrl);
      dispatcher.forward(request, response); }
  }
  protected abstract boolean
    isValidRequest{ServletRequest request);
  public void destroy{){}
}
```

Фильтр представляет собой абстрактный класс, в котором объявлен абстрактный метод isValidRequest(). Решение о дальнейшей судьбе запроса принимается на основании его анализа. Если в результате вызова метода isValidRequest () © возвращается значение false, запрос перенаправляется по URL ©, определенному в конфигурационном файле Web-приложения О.

Рассматриваемый здесь фильтр предоставляет возможность создавать конкретные подклассы, т.е. его можно адаптировать для различных стратегий защиты.

Использование HTTP-сеанса

Для поддержки сеанса часто используется следующий подход. При регистрации пользователя создается объект-идентификатор. При последующих обращениях производится проверка на наличие этого объекта. Простой фильтр, решающий данную задачу, показан в листинге 7.6.

Листинг 7.6. Фильтр для проверки идентификатора сеанса

```
public class SessionTokenSecurityFilter
extends GenericSecurityFilter {
  protected boolean isValidRequest(ServletRequest request) {
    boolean valid=false;
    HttpSession session=request.getSession();
    if (session!=null){
      UserToken token=(Token) session.getAttribute{'userToken');
```

```
      if  (token!-null){
        valid=true;
      }
    }
    return  valid;
  }
}
```

Данный подход типичен для традиционных Web-приложений. Если проверка дает отрицательный результат, пользователю предлагается страница регистрации. В Ajax-приложениях ответ сервера может быть более простым по сравнению с традиционными Web-приложениями и содержать данные в формате XML, JSON либо в виде обычного текста. В ответ на получение определенных данных клиент может самостоятельно предоставить пользователю средства регистрации. В главе 11 мы обсудим реализацию средств регистрации в рамках приложения Ajax Portal.

Использование зашифрованных HTTP-заголовков

Существует еще один способ определения корректности запроса. Он предполагает добавление к HTTP-заголовку дополнительного поля и проверку его наличия посредством фильтра. В листинге 7.7 приведен пример фильтра, который ищет конкретное поле и проверяет совпадение зашифрованного значения с ключом, хранящимся на сервере.

Листинг 7.7. Фильтр для проверки поля HTTP-запроса "ШВШ!

```
public class SecretHeaderSecurityFilter
extends GenericSecurityFilter {
  private String headerName=null;
  public void init(FilterConfig config) throws ServletException {
    super.init(config);
// Имя поля задается как конфигурационный параметр
    headerName-config.getlnitParameter("headerName");
  }
  protected boolean isValidRequest(ServletRequest request)  {
    boolean  valid=true;
    HttpServletRequest   hrequest=(HttpServletRequest)request;
    if  (headerName!=null){
      valid=false;
// O Получение значения заголовка
      String   headerVal=hrequest.getHeader(headerName);
      Encrypter   crypt=EncryptUtils.retrieve(hrequest);
      if  (crypt!=null){
// © Сравнение значения заголовка
        valid=crypt.compare(headerVal);
      }
    }
    return  valid;
  }
```

При проверке запроса фильтр читает заголовок с определенным именем О и сравнивает его с кодированным значением, содержащимся на сервере ©. Данное значение не постоянно; оно генерируется для каждого конкретЛ ного сеанса, что затрудняет незаконное обращение к системе. Класс **ЕпЯ** crypter использует для генерации MX)5-значения Apache Commons Codec и javax.security.MessageDigest. Полностью набор классов можно **полД** чить, скопировав коды примеров для данной книги. Принцип формирования ЛГО5-значения в шестнадцатеричном формате показан ниже.

```
MessageDigest
digest=MessageDigest.getInstance("MD5");
byte[] data=privKey.getBytes();
digest.update(data);
byte[] raw=digest.digest(pubKey.getBytes());
byte[] b64=Base64.encodeBase64(raw);
return new String{b64);
```

где privKey и pubKey — соответственно закрытый и открытый ключ.

Чтобы настроить фильтр для проверки всех URL, соответствующих пути /Ajax/data, надо добавить к конфигурационному файлу web.xml нашего приложения следующее определение:

```
<filter id='securityFilter_I'>
  <filter-name>HeaderChecker</filter-name>
  <filter~class>
   com.manning.aj axinaction.web.SecretHeaderSecurityFilter
  </filter-class>
  <init-param id='securityFilter_l_param_l'>
    <param-name>rejectUrl</param-name>
    <param-value>/error/reject.do</param-value>
  </init-param>
  <init-param id='securityFilter_l_param_2'>
    <param-name>headerName</param-name>
    <param-value>secret-password</param-value>
  </init-param>
</filter>
```

Согласно данной конфигурации отклоненные запросы после проверки значения поля secret-password направляются URL /error/reject.do. Кроме того, мы определяем отображение фильтрации, в результате чего фильтр вступает в действие для любого запроса, соответствующего указанному пути.

```
<filter-mapping>
  <filter-name>HeaderChecker</filter-name>
  <url-pattern>/ajax/data/*</url-pattern>
</filter-mapping>
```

На стороне клиента для генерации МБ5-дайджеста Base64 используется библиотека Пола Джонстона (она упоминалась ранее в этой главе). Для того чтобы добавить поле HTTP-заголовка, мы вызываем метод setRequestHeadert).

```
function loadXml(url){
  var req=null;
  if (window.XMLHttpRequest){
    req=new XMLHttpRequest();
```

```
    } else if (window.ActiveXObject){
      req=new ActiveXObject("Microsoft.XMLHTTP");
    }
    if (req)f
      req.onreadystatechange=onReadyState;
      req.open(•GET',url,true);
      req.setRequestHeader('secret-password',getEncryptedKey());
      req.send(params);
    }
  }
```

Здесь функция кодирования создает MO5-дайджест Base64 для указанной строки.

```
var key="password";
function getEncryptedKey(){
  return b64_md5(key);
}
```

Данное решение предполагает передачу Ajax-клиенту значения переменной key. Ключ для сеанса можно передать по протоколу HTTPS при регистрации пользователя. Он должен представлять собой псевдослучайное значение, а не строку символов, например.

Положительная особенность данного решения состоит в том. что поле заголовка HTTP-запроса не может быть модифицировано посредством стандартной гипертекстовой ссылки или HTML-формы. Злоумышленникам придется программировать HTTP-клиент, а это требует определенного уровня подготовки. Очевидно, что по мере роста популярности объекта XMLHttpRequest среди разработчиков информация о том, как сформировать поле заголовка в составе запроса, будет становиться все более доступной. Следует заметить, что средствами, подобными Apache HTTPClient и Perl LWP::UserAgent, эту задачу можно было решить и раньше.

Фильтры и другие средства аналогичного назначения не являются непреодолимой преградой для тех, кто хочет обратиться к вашему узлу, но существенно усложняют эту задачу. Как и у любого разработчика, в распоряжении хакера имеются лишь ограниченные ресурсы, поэтому, защитив приложение несколькими способами, описанными выше, вы существенно снизите вероятность несанкционированного доступа к поддерживаемой вами службе.

На этом мы заканчиваем разговор о защите Ajax-приложений. Существуют вопросы, которые не нашли отражения в данной главе. Мы решили не обсуждать их лишь потому, что они типичны не только для инфраструктуры Ajax, но и для любых Web-приложений. Надежные средства аутентификации и авторизации помогут вам управлять доступом к службам. Стандартные HTTP-заголовки могут быть использованы для определения источника запроса, что затрудняет обращение к службе, минуя официальные каналы (однако не делает такое обращение невозможным). Те, кого заинтересовали вопросы безопасности Ajax-приложений, могут найти более подробные сведения в литературе, специально посвященной этой теме.

И наконец, помните, что безопасность — понятие относительное. Никакая защита не может быть абсолютно надежной. Максимум, чего можно добиться — опережать на шаг злоумышленников. Применение HTTPS и проверку

HTTP-запросов в тех случаях, когда такие меры оправданы, можно рассмат, ривать как шаги в нужном направлении.

7.5. Резюме

В данной главе обсуждались вопросы безопасности Ajax-приложений. Мы сосредоточили внимание на тех особенностях, которые отличают защиту инфраструктуры Ajax от защиты обычных Web-приложений. Сначала мы рассмотрели "песочницу" (среду для выполнения JavaScript-программ в составе Web-браузера) и правила, предотвращающие взаимодействие фрагментов кода, полученных из различных источников. Мы также рассмотрели, как можно сделать политику "сервера-источника" менее строгой и разрешить доступ к сторонним Интернет-службам, например к API Google.

Кроме того, мы рассмотрели способы защиты данных, которыми клиент обменивается с сервером. Достаточно надежную защиту обеспечивает протокол HTTPS, но существует и более простое решение, обеспечивающее безопасную передачу пароля средствами обычного протокола HTTP. И наконец, мы показали вам уязвимое место Ajax-приложений, связанное с передачей низкоуровневых данных с сервера. Поскольку в некоторых случаях это может представлять серьезную угрозу, мы рассмотрели варианты архитектуры сервера, минимизирующие опасность. Мы также обсудили средства на стороне сервера, усложняющие несанкционированный доступ к данным.

Вопросы, рассмотренные в данной главе, помогут вам обеспечить работу Ajax-приложений в реальных условиях. В следующей главе мы продолжим разговор о характеристиках приложений, влияющих на их практическое применение. На этот раз речь пойдет о производительности программ.

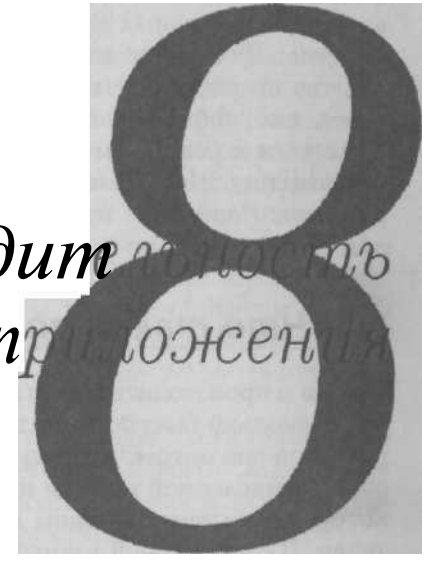
7.6. Ресурсы

Ключи для использования API Web-служб Google можно получить, обратившись по адресу http://www.google.com/apis/.

JavaScript-библиотеки Пола Джонстона, позволяющие создавать MD5-дайджесты, представлены по адресу http://pajhome.org.uk/crypt/md5/md5src.html. Для тех, кому хочется быстро проверить MD5 в действии, можно посоветовать URL генератора контрольных сумм (http: //www. f ilef ormat .info/tool/hash.htm?text* ajax+in+action).

Библиотеку Apache Commons Codec для Java, которую мы использовали при генерации Base64 MD5 на сервере, можно скопировать с узла http://j akarta.apache.org/commons/codec/.

В разделе 7.1 мы рассмотрели подписание JAR-файлов для создания защищенных приложений, ориентированных на браузеры Mozilla. Дополнительная информация по этому вопросу содержится в документе http://www.mozilla.org/projects/security/components/signed-scripts.html. Сведения об игре "морской бой" доступны по адресу http:// gamesmuseum.uwaterloo.ca/vexhibit/Whitehill/Battleship/.



Производительность приложения

В этой главе...

- Профилирование приложений Ajax
- Управление использованием памяти
- Программные решения, влияющие на производительность
- Особенности работы с конкретными браузерами

В предыдущих трех главах мы говорили о надежности приложения и erJj пригодности к сопровождению, в частности, о том, как оно будет работаЛ в реальных условиях и сможет ли разработчик модифицировать его при изменении требований к продукту. Упорядочить код приложения помогают образы разработки, а принцип разделения ответственности позволяет уменьшить взаимозависимость различных частей приложения. В результате становится возможным быстро внести изменения в программу, не нарушив ее способность выполнять основные функции.

Очевидно, что приложение будет реально применимо на практике только в том случае, если оно работает с надлежащей скоростью и не приводит к "зависанию" остальных программных компонентов, выполняющихся на той *же* машине. До сих пор мы действовали в идеальных условиях, т.е. предполагали, что на рабочей станции имеется бесконечный объем ресурсов, а браузер знает, как эффективно использовать их. В этой главе мы постараемся приблизиться к реальным условиям и обратим внимание на производительность приложения. Не будем забывать образы разработки и принципы реструктуризации. Известные программные решения не только предоставят термины *для* обсуждения, но и подскажут пути влияния на производительность.

8.1. Что такое производительность

Говоря о производительности, мы чаще всего учитываем два *основных* фактора; насколько быстро может работать приложение и какой объем системных ресурсов оно потребляет (из системных ресурсов нас обычно интересует лишь объем занимаемой памяти и загрузка центрального процессора). Программа, которая работает слишком медленно, непригодна для решения большинства задач. В современной многозадачной операционной системе программа, которая приводит к остановке остальных задач, не только бесполезна, но и вредна. Две указанные характеристики программы связаны между собой. Ситуацию можно было бы исправить, увеличив скорость процессора, однако неизвестно, существует ли процессор с требуемой тактовой частотой и оправдано ли его применение в конкретной ситуации. Мы, программисты, можем воздействовать лишь на логику приложения.

Вопросы производительности никогда не следует упускать из виду. Если мы забудем о ней, пользователи вскоре забудут о нас.

Язык программирования, подобно шахматам, можно рассматривать как независимый мир, управляемый определенным набором правил. Все, укладывающееся в рамки правил, определено и полностью предсказуемо. В этом детерминированном мире есть свое очарование, и хочется верить, что именно *его* правила описывают систему, с которой мы работаем каждый день. Этому способствует и тенденция к применению виртуальных машин, позволяющих создавать код без учета особенностей аппаратного обеспечения.

Желание работать в идеальной среде можно понять, но поддаваться ему пока рано. Современные операционные системы и прикладное программное обеспечение, в том числе и Web-браузеры, очень сложны и не всегда описываются простыми правилами. Чтобы написать код, выполняющийся на реаль-

дои машине, мы должны уметь заглянуть глубже спецификации DOM или стандарта ECMA-262, определяющего JavaScript, и учитывать реальные особенности конкретных браузеров. Если мы не будем знать "нижний уровень" иерархии программных средств, вряд ли нам удастся создать конкурентоспособный продукт. *

Если приложению требуется несколько секунд, чтобы обработать щелчок на кнопке, или несколько минут для пересылки содержимого формы, — это плохое приложение, независимо от того, насколько элегантными были решения разработчика. Если каждый раз, когда пользователь интересуется текущим временем, система запрашивает 20 Мбайт памяти, а потом освобождает лишь 15 Мбайт, пользователи вскоре откажутся от нее.

JavaScript не может обеспечить достаточно большое быстродействие, и как бы разработчики не усовершенствовали его, этот язык никогда не приблизится по скорости вычислений к языку C. Объекты JavaScript и элементы DOM требуют значительного объема памяти. Конкретные реализации браузеров далеки от совершенства и в ряде случаев напрасно расходуют память.

Производительность JavaScript-кода имеет большое значение для разработчиков Ajax-приложений, поскольку они вторгаются в ту область, которая еще не была исследована программистами. Объем JavaScript-кода в Ajax-программах намного больше, чем в классическом Web-приложении. Время жизни объектов JavaScript в Ajax-программах больше по сравнению с классическими приложениями, потому что полное обновление Web-страниц происходит гораздо реже.

В двух следующих разделах мы обсудим две составные части производительности, а именно скорость выполнения и потребление памяти. Завершится эта глава рассмотрением примеров, демонстрирующих важность образов разработки при работе с Ajax-программами и со структурой DOM.

8.2. Скорость выполнения JavaScript-программ

В мире, в котором мы живем, ценится скорость. Срок окончания проектов истекает "еще вчера". (Если вы живете в другом мире, пришлите нам письмо, а еще лучше иммиграционную карту.) Быстродействующая программа предпочтительнее более медленной, при условии, конечно, что они обе выполняют поставленную задачу. Нас как разработчиков кода должно волновать, насколько быстро он работает и как улучшить его.

Существует правило, согласно которому скорость выполнения программы определяется скоростью самой медленной подсистемы. Мы можем измерить время работы всей программы, но результат вряд ли будет полезен. Гораздо лучше будет, если мы сможем выяснить быстродействие отдельных подсистем. Измерение скорости выполнения отдельных фрагментов кода обычно называют *профилированием*. Создание высококачественного кода, как и создание любого произведения искусства, никогда не завершается, а лишь останавливается на важных этапах. (Плохой код отличается от хорошего тем, что его создание остановилось на этапе, который был абсолютно не интересен.) Путем оптимизации всегда можно хоть немного ускорить выполнение

программы. Ограничивающим фактором при этом является не уровень ма. стерства, а имеющееся в наличии время. Выяснив с помощью инструмента профилирования "узкие места"' нашего кода, мы можем сконцентрировать свои усилия для достижения наилучшего результата. Если же мы попытаемся оптимизировать код в процессе его написания, то вряд ли гарантированно получим хороший результат. "Узкие места" редко бывают именно там, где мы *ожидаем* их появления.

В данном разделе мы научимся измерять различными способами время выполнения кода. Кроме того, мы создадим простой профилировщик на JavaScript и попробуем поработать с уже существующим инструментом подобного назначения. Далее мы рассмотрим несколько простых программ и применим к ним профилировщик, чтобы выяснить наилучший способ их оптимизации.

8.2.1. Определение времени выполнения приложения

Самый простой инструмент измерения времени, имеющийся в нашем распоряжении, — это системный таймер, доступный из программы на JavaScript посредством объекта Date. Если мы создадим экземпляр Date, вызвав конструктор без параметров, то получим текущее время. Если одно значение Date вычесть из другого, разность будет представлена в миллисекундах. Пример использования объекта Date приведен в листинге 8.1.

Листинг 8.1. Измерение времени выполнения кода посредством Date

```
function  myTimeConsumingFunctionf){
  var beginning=new Date{);

  // Важные и длительные вычисления

  var ending=new Datef);
  var  duration=ending-beginning;
  alert("this  function  took  "+duration
   +"ms  to  do  something  interesting!");
```

Мы создаем объекты Date до начала и по окончании фрагмента кода, а затем определяем время выполнения этого фрагмента, вычисляя разность двух значений времени. В данном примере для представления информации о времени выполнения используется функция `alert()`, но такое решение подходит лишь для простейших случаев. Обычно информацию о времени выполнения записывают в файл протокола, однако модель безопасности JavaScript запрещает доступ к локальной файловой системе. Самый простой подход, доступный в Aj ax- приложении, — это запись данных о профиле в память и последующее воспроизведение их в форме отчета.

Заметьте, что, для того, чтобы результаты измерений были максимально достоверными, код профилирования должен выполняться настолько быстро, насколько это возможно. Запись переменной в память осуществляется намного быстрее, чем создание дополнительных узлов DOM.

В листинге 8.2 представлена простая библиотека, реализующая секундомер, которую можно использовать для профилирования кода. В процессе выполнения тестируемой программы информация о профиле хранится в памяти, а затем воспроизводится в виде отчета.

```
var stopwatch-new Object();
// Массив зарегистрированных таймеров
stopwatch.watches=new Arrayt);
// Точка входа для клиентского кода
stopwatch.getWatch=function(id,startNow){
  var watch-stopwatch.watches[id] ;
  if {!watch){
    watch=new stopwatch.Stopwatch(id);
  }
  if (startNow){
    watch.start();
  }
  return watch;
}
// Конструхтор объекта, выполняющего функции секундомера
stopwatch.StopWatch=function(id){
  this.id-id;
  stopwatch.watches[idl-this;
  this.events-new Array();
  this.objViewSpec-[
    {name: "count", type: "simple"},
    {name: "total", type; "simple"},
    {name: "events", type: "array", inline:true}
  ]?
)
stopwatch.Stopwatch.prototype.start^efunction(){
  this.current=new TimedEvent(};
}
stopwatch.Stopwatch.prototype.stop=function(){
  if (this.current){
    this.current.stop();
    this.events.append(this.current);
    this.count++;
    this.total+=this.current.duration;
    this.current—null;
  }
)
// Конструктор обработчика события измерения времени
stopwatch.TimedEvent=function(){
  this.start=new Date();
  this.objViewSpec=[
    {name: "start", type: "simple"},
    {name: "duration", type: "simple"}
  ];
}
stopwatch.TimedEvent.prototype.stop=function(){
  var stop=new Date O;
  this.duration=stop-this.start;
1
```

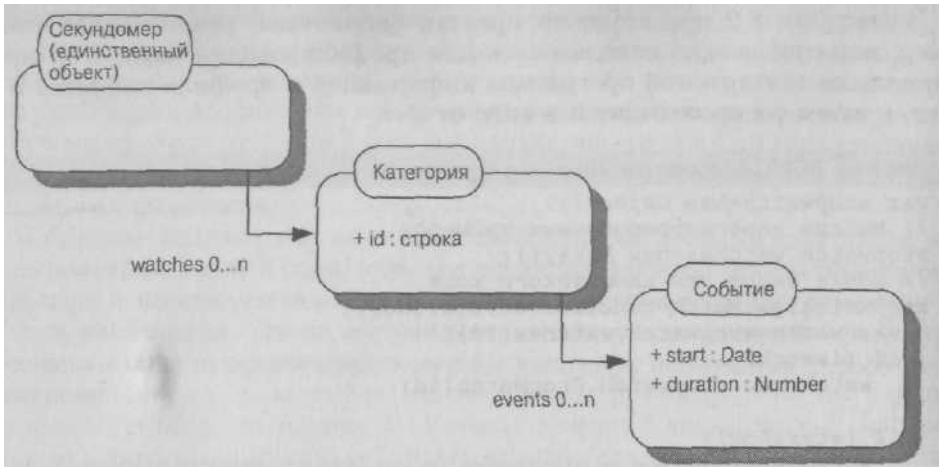


Рис. 8.1. Граф объектов библиотеки-секундомера. Каждая категория представляется объектом, который содержит предысторию событий. Все категории доступны посредством объекта `stopwatch.watches`, существующего в единственном экземпляре

```
// Генератор отчета о профиле
stopwatch.report=fvmction(div){
  var   realDiv=xGetElementById(div);
  var    report=new    objviewer.ObjectViewer(stopwatch.watches,realDiv)I
```

Наша система-секундомер поддерживает одну или несколько категорий, в каждой из которых существует одно активизированное событие и список предыдущих событий измерения времени. Когда клиентский код вызывает функцию `stopwatch, start()`, передавая ей в качестве параметра идентификатор, система создает для этой категории новый объект stopwatch либо повторно использует существующий объект. Затем клиент может многократно задавать команды секундомера `start()` и `stop()`. При каждом вызове функции `stop()` генерируется объект TimedEvent, в котором отмечается время запуска и длительность события. Если запустить секундомер несколько раз без промежуточных остановок, все обращения к `start()`, кроме последнего, будут проигнорированы.

Результатом работы секундомера является граф объектов категорий stopwatch, каждая из которых содержит предысторию измеряемых событий. Пример подобного графа показан на рис. 8.1.

По окончании сбора данных визуализируется граф объектов. Функция `render{}` использует для автоматической генерации отчета библиотеку Object Viewer (см. главу 5). В качестве упражнения мы предлагаем читателям реализовать вывод данных в формате CSV, что позволит включать их в файл.

В листинге 8.3 приведен пример применения кода секундомера к некоторой функции, выполнение которой занимает длительное время

листинг 8.3. Измерение времени с помощью библиотеки-секундомера

```
function myTimeConsuniingFvmctionO (
  var vatch=stopwatch.getWatch("my time consuming function",true);

  // Важные и длительные вычисления

  watch.stop();
}
```

Код секундомера можно достаточно просто включить в программу, подлежащую тестированию. Затем следует определить категории для различных целей. В данном примере мы назвали категорию по имени функции.

Прежде чем переходить к новому разделу, проиллюстрируем изученный материал на конкретном примере. Оценим время выполнения программы Mousemat, рассмотренной нами в главе 4. В данной программе за перемещениями мыши следят два процесса. Один из них выводит информацию о текущих координатах в строке состояния браузера, а другой отображает точку, координаты которой соответствуют текущему расположению курсора мыши. Оба процесса представляют нам полезную информацию, но создают большую нагрузку на центральный процессор. Нам надо выяснить, какой из этих процессов потребляет больше процессорного времени.

С помощью библиотеки-секундомер а мы можем достаточно просто реализовать функции профилирования для данной программы. В листинге 8.4 показана модифицированная Web-страница, содержащая новый элемент DIV, в котором отображается отчет профилировщика. Кроме того, в текст сценария включены обращения к секундомеру.

Листинг 8.4. Документ mousemat. html со средствами профилирования

```
<htmlxhead>
<link rel='stylesheet' type='text/ess' href='raousemat.ess' />
<link rel='stylesheet' type='text/ess' href='objviewer.ess' />
<script type='text/javascript' src='x/x_core.js'x/script>
<script type='text/javascript' src='extras-array.js*></script>
<script type='text/javascript' src='styling.js'></script>
<script type='text/javascript' src='objviewer.js'x/script>
<script type='text/javascript' src='stopwatch.js'X/script>
<script type='text/javascript' src='eventRouter.js'x/script>
<script type='text/javascript'>
var cursor=null;
window.onload=function(){
  var watch=stopwatch.getWatch("window onload",true);
  var mat=docu-ment.getElementById('mouseraat');
  cursor=document,getElementById('cursor');
  var mouseRouter=new jsEvent.EventRouter(mat,"onmousemove");
  mouseRouter.addListener(writestatus);
  mouseRouter.addListener(drawThumbnail);
  watch.stop();
}
function writeStatus(e)t
  var watch=stopwatch.getWatch("write status",true);
  window.status=e.clientX+","+e.clientY;
```

```
  watch.stop();
}
function drawThumbnail(e){
  var  watch=stopwatch.getWatch("draw  thumbnail",true);
  cursor.style.left=((e.clientX/5)-2)+"px";
  cursor.style.top=((e.clientY/5)-2)+"px";
  watch.stop{};
}
</script>
</head>
<body>
<div>
<a href='javascript:stopwatch.report("profiler")'>profile</a>
</div>
<div>
  <div class='mouseinat' id='mousemat'x/div>
  <div class='thumbnail' id='thumbnail'>
    <div class='cursor' id='cursor'x/div>
  </div>
  <div class='profiler objViewBorder' id='profiler'x/div>
</div>
</body>
</html>
```

Мы определили три контрольные точки: одну для window.onload и две ~ для обработчиков действий с мышью. Имена категорий отражают их назначение и используются при генерации отчета. Загрузим модифицированное приложение и проверим, как оно работает.

При перемещении курсора мыши наш профилировщик собирает данные, с которыми мы можем ознакомиться, щелкнув на ссылке, размещенной в верхнем левом углу. На рис. 8.2 показан внешний вид приложения после того, как пользователь выполнил несколько сотен передвижений мышью и задал команду генерации отчета.

Для браузеров Firefox и Internet Explorer мы получили сходные результаты: согласно нашим измерениям, на отображение данных в строке состояния расходуется приблизительно в четыре раза меньше времени, чем на движение точки в поле малого размера.

Заметьте, что событие window.onload обрабатывается за 0 миллисекунд. Такое значение получено из-за дискретности информации, предоставляемой объектом Date. При использовании данной системы профилирования мы выполняем все действия под управлением интерпретатора JavaScript, поэтому на наш код налагается ряд ограничений. При работе с Mozilla можно воспользоваться профилировщиком, встроенным в браузер. Мы рассмотрим его в следующем разделе.

8.2.2. Использование профилировщика Venkman

Для браузеров семейства Mozilla разработан большой набор встроенных мо-: Дулей, расширяющих их возможности. Одним из таких модулей является отладчик Venkman, позволяющий выполнять JavaScript-программу в пошаго-

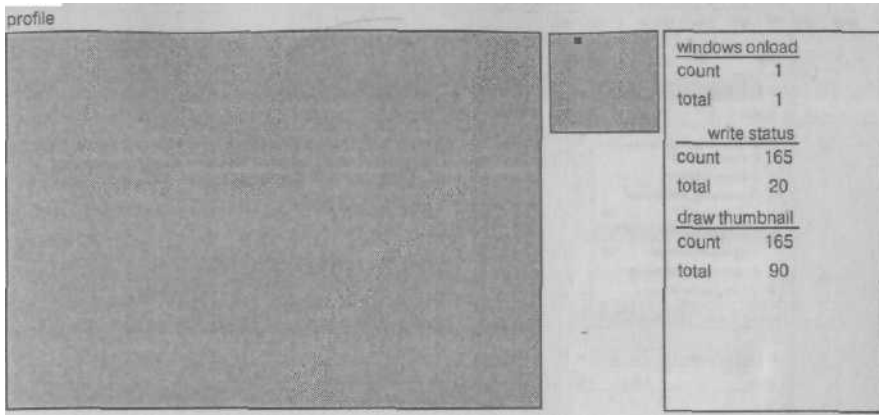


Рис. 8.2. Пример программы из главы 4, оснащенный профилировщиком на языке JavaScript Профилировщик генерирует отчет на основе данных, собранных в контрольных точках. В качестве контрольных точек мы выбрали событие window, onload, представление в поле малого размера текущей позиции курсора и вывод координат курсора в строке состояния Значение count указывает на то, какое количество записей было произведено для каждого фрагмента кода, a total — суммарное время, затраченное на выполнение данного фрагмента

вом режиме. Возможности отладчика Venkman мы обсудим в приложении А. Сейчас речь пойдет об инструменте, который не так широко известен, — о профилировщике кода.

Для того чтобы проверить код с помощью профилировщика Venkman, надо открыть интересующий документ и выбрать отладчик из меню Tools браузера. (Конечно, это возможно только в том случае, если расширение Venkman уже установлено. Инструкции по установке вы найдете в приложении А.) На панели инструментов отладчика находится пиктограмма с изображением часов и меткой Profile (рис. 8.3). После щелчка на кнопке к ней будет добавлена метка зеленого цвета.

С этого момента профилировщик Venkman тщательно отслеживает все действия, которые выполняются под управлением интерпретатора JavaScript. Перемещайте в течение нескольких секунд курсор мыши по полю, а затем снова щелкните на кнопке Profile панели инструментов отладчика, чтобы остановить процесс профилирования. В меню Window отладчика выберите пункт Profile=>Save Profile Data As. Профилировщик позволяет сохранить данные в различных форматах, в том числе CSV (для электронных таблиц), HTML (для генерации отчетов) и XML.

К сожалению, профилировщик Venkman генерирует слишком большой объем данных, в начале которых перечисляет URL chrome://. Это внутренние компоненты браузера, реализованные на JavaScript; нас они не интересуют. Помимо методов, код которых находится в составе HTML-доку мента, в регистрируемых данных находят отражение также функции каждой из используемых библиотек JavaScript, в том числе профилировщик stopwatch.js> описанный в предыдущем разделе. На рис. 8.4 показан раздел HTML-отчета, соответствующий главной HTML-странице.

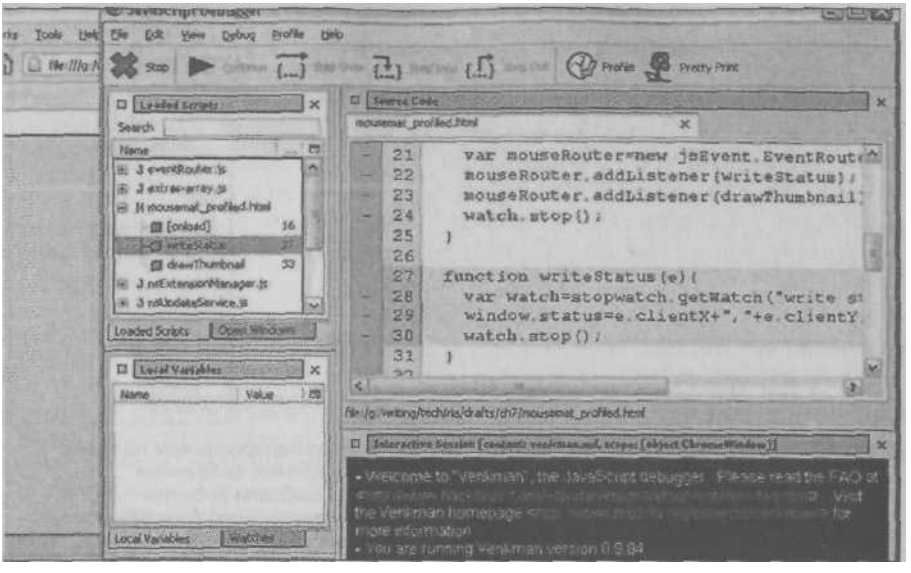


Рис 8.3. Отладчик Venkman для браузера Mozilla с активизированной кнопкой P r o f i l e Метка на кнопке означает, что система осуществляет сбор данных о времени, затраченном на выполнение всех загруженных сценариев (информация о них отображается на левой верхней панели)

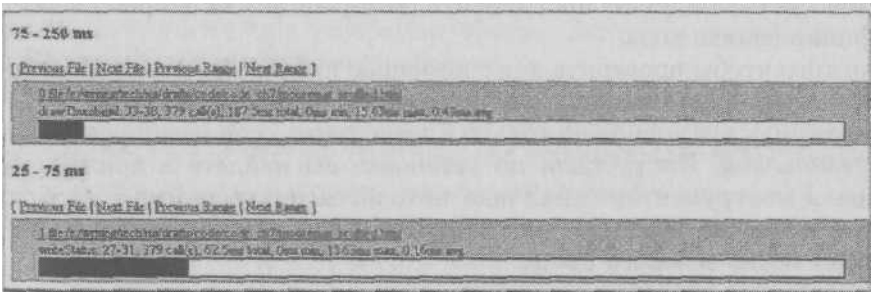


Рис. 8.4. Фрагмент отчета, сгенерированный профилировщиком Venkman В отчете отображаются число вызовов, общее, минимальное, максимальное и среднее время для каждого метода обрабатывающего в нашем примере действия с мышью

Данные, сгенерированные с помощью Venkman, хорошо согласуются с результатами, полученными с помощью объекта-секундомера, а именно: запись *в* строку состояния занимает приблизительно в три раза меньше времени, чем отображение точки, представляющей текущее положение курсора.

Venkman — полезный инструмент профилирования, не требующий модификации кода программы Если вам необходимо выполнить профилирование в нескольких браузерах, вам поможет библиотека-секундомер. В следующем разделе мы рассмотрим несколько примеров реструктуризации кода, направленной на ускорение его работы. Для проверки результатов будет использована библиотека-секундомер.

8.2.3. Оптимизация скорости выполнения Ajax-приложения

Оптимизация кода — это нечто, наверняка имеющее отношение к черной ма-
[illegible]йИ. Составление JavaScript-программ для Web-браузеров часто выполняется методом проб и ошибок- На основании таких предпосылок трудно вывести ясные правила оптимизации Ajax-кода. Оптимизация давно стала объектом 0Г>ток, многие из которых небезосновательны. И все же, используя библиотеку профилирования, знакомую вам из раздела 8.2.1, постараемся получить положительные результаты. В этом разделе мы рассмотрим три стратегии, направленные на повышение скорости выполнения программы, и применим их на практике.

Оптимизация цикла for

Первый из рассмотренных нами примеров демонстрирует ошибку, которую часто допускают программисты, и способы ее устранения. Она встречается не только в JavaScript-программах, и в данном случае нас интересует лишь код Ajax-приложения. В нашем примере выполняются длинные и бессмысленные вычисления. Они нужны нам для того, чтобы показать, сколько времени можно сэкономить, используя правильный подход. В данном случае мы будем вычислять последовательность чисел Фибоначчи, в которой каждое следующее число представляет собой сумму двух предыдущих. Приняв в качестве исходных данных две единицы, мы получим следующий набор чисел:

1, 1, 2, 3, 5, 8, ...

Программа на языке JavaScript, вычисляющая такую последовательность чисел, выглядит следующим образом:

```
function fibonacci(count){
  var a=1;
  var b=1;
  for(var i=0;Kcount;i++){
    var total=a+b;
    a=b;
    b=total;
  )
  return b;
}
```

Последовательность чисел Фибоначчи нужна нам лишь потому, что для ее вычисления требуется достаточно длительное время. Предположим теперь, что нас интересует сумма всех положительных целых чисел, не превышающих n-е число Фибоначчи. Код для вычисления этой суммы может иметь такой вид:

```
var total=0;
for (var i=0;i<fibonacci(count);i++){
  total+=i;
}
```

Результаты вычислений не имеют практического значения, но сама программа иллюстрирует проблему, которая часто возникает на практике, — попытку вычислить на каждой итерации некоторую величину. Приведенный

код работает неэффективно. Причина в том, что на каждом шаге цикла вызывается функция f ibonacci (count), несмотря на то, что результат ее выполнения остается постоянным. Правила записи цикла for таковы, что в некоторых случаях подобная ошибка может *остаться незамеченной. Код* надо переписать так, чтобы функция f ibonacci () вызывалась только один раз.

```
var total=0;
var loopCounter=fibonacci(count) ;
for (var i=0;i<
 loopCounter
;i++){
  total+=i;
}
```

Итак, мы оптимизировали код. Но насколько? Если оптимизации подлежит *большой объем* кода, необходимо знать, оправданы ли затраченные на нее усилия. Для того чтобы ответить на этот вопрос, мы поместим оба варианта кода в состав Web-страницы и туда же включим библиотеку-секундомер, позволяющую выполнить профилирование каждой функции. Код страницы показан в листинге 8.5.

Листинг 8.5. Профилирование цикла for

```
<html>
<head>
<link rel=stylesheet' type='text/ess' href='mousemat.css' />
<link rel='stylesheet' type='text/ess' href='objviewer.ess' />
<script type=' text/ javascript src=x/x__core. js'x/script>
<script type='text/javascript' src='extras-array.js'X/script>
<script type='text/javascript src='styling.js'x/script>
<script type='text/javascript' src='objviewer.js'X/script>
<script type='text/javascript' src='stopwatch.js'X/script>
<script type='text/javascript' src='eventRouter.j s'X/script>
<script type='text/javascript'>
function slowLoop(count){
  var watch=stopwatch.getWatch("slow loop",true);
  var total=0;
  // Вызов функции на каждом шаге цикла
  for (var i=0;i<fibonacci(count);i++){
    total+=i;
  }
  watch.stop();
  alert(total);
}
function fastLoop(count){
  var watch=stopwatch.getWatch("fast loop",true);
  var total=0;
  // Граничное значение вычисляется один раз
  var loopCounter=fibonacci(count);
  for (var i=0;i<loopCounter;i++){
    total+=i;
  }
  watch.stop();
  alert(total);
}
// Вычисление чисел Фибоначчи
```

Таблица 8.1. Результаты профилирования для оптимизации цикла for

длгоритм	Время выполнения (в миллисекундах)
Исходный вариант	3085
Оптимизированный вариант	450

```
function fibonacci(count){
  var a=l;
  varb=l;
  for{var i=0;Kcount;i++){
    var total=a+b;
    a=b;
    b=total;
  }
  return b;
}
function go(isFast){
  var   count=parselnt(document.getElementById{"count").value);
  if  {count==NaN){
    alert("please enter a valid number");
  }else if (isFast){
    fastLoop(count);
  }else{
    slowLoop(count);
  }
}
</script>
</head>
<body>
<div>
<a  href='javascript:stopwatch.report("profiler")'>profile</a> 
<input id='count' value='25'/> 
<a  href='javascript:go(true)'>fast  loop</a>snbsp;
<a  href='javascript:go(false)'>slow  loop</a>
</div>
<div>
  <div  class='profiler  objViewBorder'  id='profiler'></div>
</div>
</body>
</html>
```

Функции slowLoopO и fastLoopt) представляют два варианта алгоритма. Обращение к ним осуществляется из тела функции до(). На странице присутствуют гипертекстовые ссылки, позволяющие обращаться к каждому варианту цикла. Порядковый номер числа Фибоначчи задается с помощью поля редактирования, содержащегося в HTML-форме. Для того чтобы время вычислений находилось в разумных пределах, мы выбираем порядковый номер, равный 25. Третья гипертекстовая ссылка воспроизводит отчет о профилировании. Пример результатов проверки приведен в табл. 8.1.

Таким образом, мы видим, что вынесение *длительных* вычислений за пределы цикла for дало существенные результаты. Однако это не обязательно справедливо для каждой программы. Чтобы представлять себе реальные результаты выполненной работы, надо осуществить профилирование.

Следующий пример типичен для Ajax. В нем мы рассмотрим создание узла DOM.

Присоединение узлов DONI к документу

Для того чтобы отобразить данные в окне браузера, мы обычно создаем узлы DOM, а затем присоединяем их к дереву документа: либо к document.body, либо к другому узлу. Как *только* узел DOM становится частью документа, он воспроизводится. Таков порядок включения новых элементов, и изменить его невозможно.

Для отображения документа в окне браузера необходимо определить параметры, определяющие расположение элементов; а это часто предполагает большой объем вычислений. Если мы собираем сложный пользовательский интерфейс, имеет смысл создать сначала все узлы, соединить их друг с другом, а лишь потом включить всю полученную структуру в документ. В этом случае компоновка элементов на странице осуществляется лишь один раз. Рассмотрим простой пример создания элемента-контейнера, в который мы включаем большое количество узлов DOM. Говоря о примере, мы первым упомянули узел контейнера, логично будет, если именно его мы и создадим в первую очередь. Необходимый для этого код приведен ниже.

```
var container=document.createElement("div");
  container.className='mousemat';
  var outermost=document .getEleinentById( 'top') ;
  outermost.appendChildfcontainer);
  for(var i=0;i<count;i++){
    var node=dbcument.createElement('div' ) ;
    node.className='cursor' ;
    node.style.position='absolute';
    node.style.left=(4+parselnt(Math.random{)M92))+"px";
    node.style.top=(4+parselnt(Math.random()M92)H"px";
    container.appendChild(node);
```

Здесь outermost — существующий элемент DOM, к которому мы присоединяем наш контейнер, который, в свою очередь, включает несколько узлов DOM. Поскольку мы сначала включили в документ контейнер, а затем стали заполнять его, наш документ подвергся модификации *count+1* раз. Незначительно преобразовав код, мы изменим ситуацию.

```
var container=document.createElement("div") ;
container. classUame='mouseniat' ;
var outermost=document.getElementByldf'top') ;
for(var i=0;i<count;i++){
  var node=document.createElement('div') ;
  node.className='cursor' ;
  node.style.position^'absolute';
  node.style.left={4+parselnt(Math.random()*492)>+"px";
  node.style.top=(4+parseInt(Math.random()*492))+"px";
```

```
  container. appendChild(node);
)
outermost.appendChild(container);
```

По сути, мы изменили положение только одной строки, но теперь существующий документ модифицируется лишь один раз. В листинге 8.6 показан код страницы, на которой с помощью библиотеки-секундомера сравниваются два варианта действий по включению контейнера.

Листинг 8.6. Профилирование процедуры включения элемента DOM

```
<html>
<head>
<link rel='stylesheet' type='text/ess' href='mouseroat.ess' />
<link rel='stylesheet' type='text/ess' hre£='objviewer.ess' />
<script type='text/javascript' src='x/x_core.js'x/script>
<script type='text/javascript' src='extras-array. js'></script>
<script type='text/javascript' src='styling.js'X/script>
<script type='text/javascript' src='objviewer.js'x/script>
<script type='text/javascript' src='stopwatch.js'x/script>
<script type='text/javascript' src='eventRouter. js'X/script>
<script type='text/javascript'>
var cursor=null;
function slowNodes(count){
  var watch=stopwatch.getWatch("slow nodes",true);
  var container=document.createElement("div");
  container,className='mousemat' ;
  var outermost=document.getElementByldt'top');
  // Присоединение пустого контейнера
  outermost.appendChild(container);
  for(var i=0;i<count;i++)I
    var node=document.createElement('div') ;
    node.className='cursor';
    node.style.position='absolute' ;
    node.style.left=(4+parselnt{Math.random()*492))+"px";
    node.style.top=(4+parseIntiMath.randomO*'i92n+"px";
    container.appendChild(node);
  }
  watch.stop();
}
function fastNodes(count){
  var watch=stop-watch.getViatch("fast nodes", true);
  var container=document.createElement("div");
  container.className='mousemat' ;
  var outermost=document.getElementById('top' ) ;
  for(var i=0;i<count;i++){
    var node-document.createElement[*'div') ;
    node.className='cursor' ;
    node.style,position='absolute';
    node. style. left=H+parse!nt (Matti, random() *492)) +"px" ;
    node.style.top=(4+parselnt(Math.random()*492))+"px";
    container.appendChild(node);
  1
  // Присоединение контейнера, заполненного компонентами
  outermost.appendChild(container);
  watch.stop();
```

Таблица 8.2. Профилирование процедуры создания узлов DOM

Алгоритм	Число операций компоновки страницы	Время выполнения (в миллисекундах)
Исходный вариант	641	681
Оптимизированный вариант	1	461

```
}
function go(isFast){
  var count=parselnt(document.getElementById("count").value);
  if (count==NaN){
    alert("please enter a valid number");
  Jelse if (isFast){
    fastNodes(count);
  }else{
    slowNodes(count);
```

```
</script>
</head>
<body>
<div>
<a href='javascript:stopwatch.report("profiler")'>profile</a> 
<input id='count' value='640'/>Snbsp;
<a href='javascript:go(true)'>fast loop</a>snbsp;
<a href='javascript:go(false)'>slow loop</a>
</div>
<div id='top'>
  <div class='mousemat' id='mousemat'></div>
  <div class='profiler objViewBorder' id='profiler'X/div>
</div>
</body>
</html>
```

Как и ранее, Б нашем распоряжении есть две гипертекстовые ссылки, позволяющие вызвать "быструю" и "медленную" функцию. Параметром является значение, введенное в поле HTML-формы. На этот раз мы указываем, сколько узлов DOM следует включить в контейнер. Мы выбрали значение 640. Результаты проверки представлены в табл. 8.2.

И снова шаги по оптимизации, предпринятые из соображений здравого смысла, дали результат. Профилирование показывает различие между исходным и оптимизированным вариантами кода. В данном случае мы сэкономили приблизительно треть времени. При другом взаимном расположении узлов результаты будут иными. (Заметьте, что в примере использовалось только абсолютное позиционирование, что упрощает задачу системы компоновки.)

Наш последний пример имеет отношение к особенностям языка JavaScript. Мы сравним различные подсистемы и постараемся выявить "узкие места".

Минимизация "числа точек"

3 JavaScript, как и во многих других языках, можно обращаться к переменным, используя сложную иерархию объектов. При этом родительский объект отделяется от дочернего точкой. Например:

```
myGrandFather.clock.hands.minute
```

Это выражение представляет собой ссылку на минутную стрелку дедушкиных часов. Предположим, что мы хотим иметь ссылки на все три стрелки. Для этого нам надо написать следующие выражения:

```
var  hourHand=myGrandFather.clock.hands.hour;
var  minuteHand=myGrandFather.clock.hands.minute;
var   secondHand=myGrandFather.clock.hands.second;
```

Каждый раз, когда интерпретатор встречает имя, следующее после точки, он обращается к дочернему объекту. В предыдущем примере интерпретатору пришлось выполнить поиск дочернего объекта девять раз, причем некоторые из операций поиска дублировали друг друга. Перепишем пример следующим образом:

```
var  hands=myGrandFather.clock.hands;
var  hourHand=hands.hour;
var  minuteHand=hands.minute;
var   secondHand=hands.second;
```

Теперь поиск дочернего объекта осуществляется всего пять раз, а интерпретатор избавлен от лишней работы. В компилируемых языках, например в Java или C#, компилятор выполняет оптимизацию автоматически. Нам неизвестно, существует ли версия JavaScript- интерпретатор а, реализующая подобную возможность, и, если есть, в каком браузере она используется, поэтому позаботимся об оптимизации сами. Воспользуемся для этой цели библиотекой- секу ндомер ом.

Рассмотрим в качестве примера программу, которая вычисляет силу гравитационного взаимодействия двух тел, например Земли и Луны. С каждым телом связаны некоторые физические свойства, например, масса, положение, скорость и ускорение. Эти параметры используются при вычислении гравитационного взаимодействия. Поскольку мы обсуждаем обращение к объектам посредством имен, разделенных точками, оформим свойства в виде графа.

```
var  earth={
  physics:{
    mass:10,
    pos:{  x:250,y:250  },
    vel:{  x:0,   y:0   },
    ace:I  x:0,   y:0   }
  }
};
```

Объект верхнего уровня, physics, здесь лишний; мы включили его лишь Для увеличения количества точек при обращении к свойству.

Выполнение приложения происходит в два этапа. Сначала вычисляется состояние в фиксированные моменты времени — расстояние, сила притяжения, ускорение и другие характеристики, о которых многие из нас не вспо-

минали с университетских времен. Данные для каждого момента времени записываются в массив, в частности, в него помещаются сведения о положении каждого тела.

На втором этапе отображаются траектории тел; вычисления производятся на основании данных, подготовленных на первом этапе. Максимальные и минимальные значения координат используются для выбора области, в которой будут выводиться траектории. В реальном приложении имело бы смысл выводить данные в ходе имитации, но мы разделили программу на два этапа чтобы иметь возможность выполнить профилирование для каждого из них.

Как и ранее, определяем два варианта кода: неэффективный и оптимизированный- В неэффективном коде используем при обращении к объектам столько точек, сколько возможно. Ниже приведен фрагмент кода (на уравнение не стоит обращать внимание).

```
var    grav=(earth.physics.mass*moon.physics.mass)/(dist*dist*gravF);
var   xGrav=grav*(distX/dist);
var   yGrav=grav*(distY/dist);
moon.physics.ace.x=-xGrav/(moon.physics.mass);
moon.physics-ace.y=-yGrav/(moon.physics.mass);
moon.physics.vel.x+=moon.physics.ace.x;
moon.physics.vel.y+=moon.physics.ace.y;
moon.physics.pos.x+=moon.physics.vel.x;
moon.physics.pos.y+=moon.physics.vel. y;
```

Большое число повторяющихся операций поиска дочернего объекта оставляет простор для работы над кодом. Ниже приведен его оптимизированный вариант.

```
var mp=moon.physics;
var mpa=mp.acc;
var mpv=mp.vel;
var mpp-mp.pos;
var mpm^mp.mass;

var grav=(epm*mpm)/(dist*dist*gravF);
var xGrav=grav*(distX/dist);
var yGrav=grav*(distY/dist);
mpa.x=-xGrav/(mpm);
mpa.y=-yGrav/(mpm);
mpv.x+=mpa.x;
mpv.y+=mpa.y;
mpp.x+=mpv.x;
mpp.y+=mpv.y;
```

Мы выполнили необходимые преобразования в начале программы и сохранили результаты в переменных. В результате код стал более удобочитаемым и, что на данном этапе для нас более важно, снизилась нагрузка на интерпретатор. В листинге 8.7 показан полный код Web-страницы, позволяющий сравнить два алгоритма.

Листинг 8.7. Профилирование действий по обращению к переменным

```
<htmlxhead>
<link rel='stylesheet' type='text/ess' href='mousemat.css' />
<link rel='stylesheet' type='text/ess' href='objviewer.ess' />
```

```
<script type='text/javascript' src='x/x_core.js'x/script>
<script type='text/javascript' src='extras-array.j s'></script>
<script type='text/javascript' src^'styling.js'X/script>
<script type=" text /javascript' src='objviewer. js'></scro_pt>
<script type='text/javascript' src='stopwatch.js'X/script>
<script type='text/javascript' src^'eventRouter.js'X/script>
<script type='text/javascript'>
// Инициализация данных о планетах
var moon={
  physics:{
    mass:1,
    pos:{ x:120,y:80 },
    vel:[ x:-24, y:420 },
    ace:{ x:0, y:O }
  1
};
var earth={
  physics:{
    mags:10,
    pos:{ x:250,y:250 },
    vel:{ x:Q, y:0 },
    ace:{ x:0, y:O }
  1
};
var gravF=100000;
function showOrbit(count,isFast){
  // О Выбор типа вычислений
  var data=(isFast) ?
    fastData(count) :
    slowData(count);
  var watch=stopwatch.getWatch("render",true);
  // © Отображение орбиты
  var canvas=document.getElementBy!d('canvas');
  var dx=data.max.x-data.min.x;
  var dy=data.max.y-data.min.y;
  var sx=(dx==0) ? 1 : 500/dx;
  var sy=(dy==0) ? 1 : 500/dy;
  var offx=data.min.x*sx;
  var offy=data.min.y*sy;
  for (var i=0;i<data-path.length;i+=10){
    var datum=data.path[i];
    var dpm=datum.moon;
    var dpe=datum.earth;
    var moonDiv=document.createElement("div");
    moonDiv.className='cursor';
    raoonDiv.style.position='absolute';
    moonDiv.style.left=parselnt((dpm.x*sx)-offx)+"px";
    moonDiv.style.top=parselnt((dpm.x*sx)-offy)+"px";
    canvas.appendChiId(moonDiv);
    var earthDiv=document.createElement("div");
    earthDiv.className='cursor';
    earthDiv.style.position='absolute';
    earthDiv.style.left=parselnt((dpe,x*sx)-offx)+"px";
    earthDiv.style.top=parselnt((dpe.x*sx)-offy)+"px";
    canvas.appendChiId(earthDiv);
  }
```

```
  watch.stop();
}
// © Использование максимального количества операций
//  поиска дочерних объектов
function slowData(count){
  var watch=stopwatch.getWatch("slow orbit",true);
  var data={
    min:{x:0,y:0},
    max:{x:0,y:0},
    path:П
   };

  }
  watch.stop();
  return data;
}
// О Использование минимального количества операций
//  поиска дочерних объектов
function fastData(count){
  var watch=stopwatch.getWatch{"fast orbit",true);
  var data={
    min:{x:0,y:0},
    max:{x:0,y:0},
    path:[]
   };

  }
  watch.stop{);
  return data;
}
function go{isFast){
  var count=parselnt(document.getElementBy!d{"count").value);
  if (count==NaN){
    alert("please enter a valid number");
  }else{
    showOrbit(count,isFast);
  }
}
</script>
</head>
<body>
<div>
<a href=1javascript:stopwatch.report("profiler")'>profile</a> 
<input id='count1 value='640'/> 
<a href='javascript:go(true)'>fast loop</a> 
<a href=1javascript:go(false)'>slow loop</a>
</div>
<div id='top'>
  <div class='mouseraat' id='canvas'>
</div>
  <div class='profiler ob^ViewBorder' id='profiler'X/div>
</div>
</body>
</html>
```

•Таблица 8.3. Профилирование результатов для различных вариантов обращения к переменным

длТоритм	Время выполнения (в миллисекундах)
Исходный алгоритм вычислений	94
Оптимизированный алгоритм вычислений	57
Воспроизведение (среднее значение)	702

Структура страницы уже знакома вам. Функции slowData() © и fastData() О содержат два варианта кода, реализующего этап вычислений. Они используются на первом этапе вычислений при генерации структуры данных О. Мы не приводим здесь полный код вычислений так как его запись заняла бы много места. Различие между вариантами алгоритма ясны из приведенных выше фрагментов; кроме того, вы можете скопировать полный код примера. С каждой функцией, выполняющей вычисления, связан объект stopwatch, выполняющий ее профилирование. Обращение к функциям осуществляется в теле showOrbit (). На основании вычисленных данных эта функция отображает траектории ©. Ее профилирование осуществляется с помощью третьего секундомера.

Элементы интерфейса в данном случае те же, что и в предыдущих двух примерах. Посредством гипертекстовых ссылок вызывается "быстрый" и "медленный" варианты вычислений, а параметр вводится посредством поля редактирования. В данном случае параметром является число точек, для которых выполняется имитация. При активизации третьей ссылки отображаются результаты профилирования. В табл. 8.3 отображены данные, полученные для 640 точек.

И снова мы видим, что в результате оптимизации достигнуто значительное увеличение скорости — сэкономлено более трети времени, требовавшегося ранее для вычислений. Мы можем сделать вывод, что первоначальная догадка о том, что количество точек в сложных именах влияет на скорость вычислений, верна.

Но если мы рассмотрим все этапы работы, то увидим, что оптимизированная программа выполняется за 760 миллисекунд, а неоптимизированная — за 792 миллисекунды, т.е. выигрыш в результате оптимизации ближе к 5, чем к 40%. В данном случае узким местом приложения является не функция, выполняющая вычисления, а подсистема отображения. Таким образом, сосредоточив внимание на оптимизации процесса вычислений, мы приняли неправильное решение.

Этот пример демонстрирует еще один важный аспект профилирования. Одно дело — знать, что некоторый фрагмент кода можно оптимизировать, и совсем другое — представлять себе результаты, которые можно получить от оптимизации. Мы видим, что операции со структурой DOM занимают приблизительно в восемь раз больше времени, чем подготовка данных, но это справедливо лишь для данного конкретного примера. Возможно, что в других

ситуациях дело обстоит так же, но выяснить это можно, лишь выполнив измерения, причем желательно сделать их для разных платформ и браузеров.

На этом мы заканчиваем разговор о профилировании и скорости выполнения программ. Рассмотренные примеры дают полное представление о том. какую пользу можно извлечь, правильно выполняя профилирование при работе над проектом Ajax. Предположим, что благодаря профилированию вы до бились приемлемой скорости выполнения кода. Однако производительность зависит не только от скорости работы, но и от объема памяти, используемой приложением. Этому вопросу и будет посвящен следующий раздел.

8.3. *Использование памяти JavaScript-кодом*

Этот раздел посвящен вопросам управления памятью в Ajax-программах. Некоторые из рассмотренных здесь решений применимы для любой системы и любого языка программирования, другие специфичны для Ajax-приложений и даже для конкретных Web-браузеров.

Выполняющееся приложение запрашивает у операционной системы определенный объем памяти. В идеальном случае программа должна получить столько памяти, сколько необходимо для эффективного решения поставленной задачи, и вернуть неиспользуемую память системе. Если приложение написано непрофессионально, оно может потреблять неоправданно много памяти либо не возвращать память системе по окончании работы.

По мере перехода от простых Web-страниц к богатым клиентам Ajax качество управления памятью все больше влияет на время отклика программы и надежность ее работы. Применение стандартных программных решений помогает создать код, простой для восприятия и сопровождения, а также выявить причины утечки памяти и устранить их. (Утечкой памяти мы будем называть ситуацию, при которой программа запрашивает память у системы и не возвращает ее после завершения работы.)

Начнем с рассмотрения общих вопросов управления памятью.

8.3.1, Борьба с утечкой памяти

В любой программе может иметь место утечка памяти. Вопросы выделения памяти и ее освобождения требуют пристального внимания разработчиков, применяющих языки, в которых не предусмотрено автоматическое управление памятью, например язык C. В JavaScript имеется подсистема "сборки мусора", которая автоматически следит за выделенной памятью и освобождает ее, если она не используется. Это позволяет избежать многих проблем, типичных для языков, в которых отсутствуют подобные средства, однако было бы ошибкой считать, что при наличии подсистемы управления утечка памяти не может произойти.

Процедура "сборки мусора" пытается выяснить, можно ли удалить неиспользуемую переменную, при этом она учитывает возможность доступа к ней по сети или посредством ссылок, содержащихся в других переменных. Если получить доступ к переменной невозможно, она помечается как разрешен-

ная к удалению, и **на** следующем шаге занимаемая ею память освобождается (момент времени, когда это произойдет, предсказать невозможно). Забывая удалить ссылки на переменную после того, как работа с ней окончена, мы создаем условия для утечки памяти, которая может иметь место даже в языках с управлением памятью.

Рассмотрим простой пример. Определим объектную модель, описывающую домашних животных и их хозяев. Начнем с хозяев; для их описания используем объект Person.

```
function Person(name){
  this.name=name;
  this.pets=new Array();}
```

Хозяину может принадлежать один или несколько домашних любимцев. Когда человек приобретает животное, он сообщает о том, что у того теперь есть конкретный хозяин.

```
Person.prototype.addPet=function(pet){
  this.pets[pet.name]=pet;
  if (pet.assignOwner){
    pet.assignOwner(this);}
}
```

Аналогично, когда человек удаляет животное из списка своих любимцев, он сообщает животному о том, что оно больше никому не принадлежит.

```
this.removePet{petName)=function{
  var orphan=this.pets[petName];
  this.pets[petNarrLe]=null,•
  i£   (orphan.unassignOwner){
    orphan.unassignOwner(this);)
}
```

В любой момент времени хозяин знает, какие животные ему принадлежат, и может управлять списком своих любимцев посредством методов addPet() и removePet(). Человек информирует животное о принадлежности хозяину или ее отсутствии, считая, что животное автоматически начнет считать человека своим хозяином (подобное поведение можно предполагать по умолчанию, а необходимую проверку выполнять в процессе работы программы).

Определим два типа домашних животных: кошку и собаку. Эти животные различаются по своему отношению к хозяину, кошка не обращает внимания на тот факт, кому она **принадлежит,** а собака привязывается к человеку на всю жизнь. (Мы приносим извинения всему животному миру за столь вольное обобщение.)

Таким образом, определение кошки может выглядеть как показано ниже:

```
function Cat(name){this.name=name;}
Cat.prototype.assignOwner=function(person){}
Cat.prototype.unassignOwner=function(person){}
```

Кошка не интересуется фактом своей принадлежности кому-либо, поэтому тело соответствующих методов пусто.

Опишем теперь собаку. Она помнит, кому она принадлежит, и после того, как человек откажется от **нее,** по-прежнему считает его своим хозяином (многие собаки ведут себя именно так).

```
function Dog(name){this.name=name;}
Dog.prototype.assignOwner=function(person){
  this.owner=person;}
Dog.prototype.unassignOwner=function(person){
  this.owner=person;}
```

Объекты Cat и Dog представляют собой реализации класса Pet, но поведение их нельзя назвать корректным. Оно соответствует букве соглашения о принадлежности, но не его "духу". В Java или C# мы можем явным образом определить интерфейс Pet, но это не спасет классы, реализующие его, от неправильного поведения. При работе над реальными проектами специалисты, занимающиеся объектными моделями, тратят много времени, исправляя некорректное поведение реализаций интерфейсов.

Поработаем немного над моделью. Создадим три объекта.

1. jim — Person
2. whiskers — Cat
3. fido — Dog

В первую очередь создадим экземпляр Person (этап 1).

```
var jim=new Person("jim");
```

Далее сделаем человека владельцем кошки (этап 2). Экземпляр whiskers создается в процессе вызова метода addPet(), поэтому ссылка на объект Cat существует лишь до завершения метода. Однако объект jim также создает ссылку на whiskers, а она доступна в течение всего времени существования объекта jim.

```
jim.addPet(new Cat("whiskers"));
```

Пусть теперь jim владеет также собакой (этап 3). В отличие от whiskers, мы объявим для этого объекта глобальную переменную.

```
var fido=new Dog("fido");
jim.addPet(fido);
```

Однажды jim решает избавиться от кошки (этап 4).

```
jim.removepet("whiskers") ;
```

Затем он решает, что собака ему тоже не нужна (этап 5). Наверное, он собирается переехать в другой город.

```
jim.removePet("fido");
```

Мы также больше не интересуемся этим человеком и удаляем ссылку на него (этап 6).

```
jim=null,-
```

И наконец, мы удаляем ссылку на fido (этап 7).

```
fido=null;
```

По окончании этапа 6 мы считаем, что избавились от объекта jim, поскольку присвоили соответствующей переменной значение null. На самом деле ссылка на него существует в fido и доступна посредством выражения fido.owner.

Система "сборки мусора" не затронет этот объект, и он останется незамеченным в области памяти, называемой "куча" (heap), интерпретатора JavaScript. Tlйшь на этапе 7, когда переменной fido присваивается значение null, объект -i^m становится недоступным, и память может быть освобождена.

В нашем простом сценарии проблема не является серьезной, к тому же оНа быстро решается, но данный пример иллюстрирует, как решения, не зависящие друг от друга, на первый взгляд, влияют на процесс "сборки мусора'. Объект fido не обязательно будет удален сразу же после jim, и если мы предусмотрим возможность хранить информацию о нескольких предыдущих владельцах, то в памяти может остаться целый набор объектов Person, ведущих призрачное существование, но занимающих реальные ресурсы. Если бы мы объявили fido при вызове метода, а ссылку на объект Cat поместили в глобальную переменную, то такое явление не возникло бы. Для того чтобы оценить, насколько серьезную проблему может создать подобное поведение объекта fido, нам надо ответить на следующие вопросы.

1. Какой объем памяти он может занимать с учетом ссылок на объекты, которые в других обстоятельствах были бы удалены? Мы знаем, что в данном примере fido может ссылаться только на один объект Person. Но даже в этом случае не исключено, что объект Person содержит ссылки на 500 объектов Cat, поэтому объем расходуемой памяти может быть существенным.
2. В течение какого времени память будет оставаться занятой? В рассмотренном примере ответом на этот вопрос будет "не очень долго". Однако впоследствии между удалением jim и fido могут появиться команды, выполнение которых будет занимать длительное время. Более того, язык JavaScript позволяет создавать программы, управляемые событиями, и удаление jim и fido может происходить при обработке различных событий. В этом случае, чтобы предсказать, в течение какого времени будет занята дополнительная память, пришлось бы анализировать типичные сценарии работы пользователя.

Ни на один из приведенных выше вопросов непросто ответить. Самое лучшее, что можно сделать, — это задавать их себе в процессе создания и модификации кода и выяснять в процессе тестирования, правильны ли наши предположения.

Мы рассмотрели основные принципы управления памятью. Однако существуют проблемы, специфические для приложений Ajax. Обсудим их в следующем разделе.

8.3.2. Особенности управления памятью в приложениях Ajax

Мы рассмотрели понятия, лежащие в основе управления памятью, для многих языков программирования. Отслеживать объем используемой памяти и доступность переменных важно при работе над любым приложением, однако есть также вопросы, специфические для Ajax. Ajax-программы ра-

ботают в управляемой среде, контейнере, который предоставляет некото! рые функции и запрещает пользоваться другими. Это несколько меняет общую картину.

В главе 4 мы разделили Aj ax-приложение, на три подсистемы — модель I представление и контроллер. Модель, как правило, состоит из обычных объектов JavaScript. Разработчик определяет эти объекты и предусматривает создание их конкретных экземпляров. Представление в основном состоит из узлов DOM — объектов, которые браузер предоставляет JavaScript-программе. Контроллер является связующим звеном между моделью и представлением. Реализуя контроллер, необходимо уделять особое внимание управлению памятью.

Устранение циклических ссылок

В разделе 4.3.1 мы рассмотрели принцип обработки событий, позволяющий связать объектную модель предметной области (часть подсистемы модели) с узлами DOM (частью подсистемы представления). Попробуем применить данный подход для задач, связанных с управлением памятью. Ниже приведен конструктор модели предметной области, соответствующей кнопке.

```
function Button(value,domEl){
  this,domEl=domEl;
  this.value=value;
  this.domEl.buttonObj=this;
  this.domEl.onclick=this.clickHandler;
}
```

Заметьте, что между элементом DOM domEl и сахмим объектом создаются двусторонние ссылки. Ниже приведена функция обработки событий, на которую ссылается конструктор

```
Button.prototype.clickHandler=function(event){
  var buttonObj=this.buttonObj;
  var value=(buttonObj && buttonObj.value) ?
    buttonObj.value : "unknown value";
  alert(value);
}
```

Как вы помните, функция обработки события вызывается не объектом Button, а узлом DOM; этот факт влияет на формирование контекста. Ссылка представления на модель нужна нам для того, чтобы обеспечить взаимодействие с уровнем модели. В данном случае мы читаем свойство value. В других ситуациях вызываются функции объектов, составляющих модель предметной области.

Объект типа Button доступен до тех пор, пока хотя бы один из остальных доступных объектов содержит ссылку на него. Точно так же элемент DOM доступен до тех пор, пока на него ссылаются другие доступные элементы. Если элемент DOM присоединен к главному документу, он остается доступен, даже если в программе нет на него ни одной ссылки. Таким образом, если элемент DOM, являющийся частью документа, связан с объектом Button, последний не будет удален системой "сборки мусора" *до* тех нор, пока мы явным образом не разорвем эту связь.

Если модели предметной области взаимодействуют со структурой DOM, зозможно существование локального объекта JavaScript, на который не ссылается ни одна глобальная переменная, определенная в программе, но который, тем не менее, доступен посредством DOM. Для того, чтобы создать условия для удаления этого объекта подсистемой "сборки мусора", нам надо раписать простую функцию очистки. (Поскольку такая функция вызывается вручную, ее можно рассматривать как шаг в сторону объектов C++.) Для объекта Button эта функция будет иметь следующий вид:

```
Button.prototype.cleanUp=function(){
  this.domEl.buttonObj=null;
  this.domEl=null;
}
```

Первая строка в теле функции удаляет ссылку узла DOM на объект. Вторая строка удаляет ссылку объекта на узел DOM. Данная функция не разрушает узел, а лишь присваивает переменным, в которых ранее хранились ссылки, значение null. В данном случае узел DOM передавался конструктору нашего объекта в виде параметра, поэтому мы не отвечаем за его дальнейшую судьбу. В других ситуациях такая ответственность может быть возложена на нас, поэтому рассмотрим вопрос управления этим узлом.

Управление элементами DOM

В приложениях Ajax, в особенности тех, в которых используются сложные модели предметной области, узлы DOM создаются и включаются в состав дерева документа из программы. В классических Web-приложениях формирование узлов DOM чаще всего производится при первой загрузке документа на основании HTML-описания. Объект Objectviewer, рассмотренный в главах 4 и 5, и подсистема оповещения (см. главу 6) содержат объекты модели предметной области, способные к отображению. Отображение осуществляется путем создания элементов DOM и присоединения их к главному документу. Дополнительные возможности предполагают дополнительную ответственность. Если узел создан программой, мы обязаны позаботиться и о его удалении.

Ни спецификация DOM, разработанная W3C, ни ее реализации в популярных браузерах не предусматривают способа удаления созданного узла DOM. Единственное, что мы можем сделать, — это разорвать связь между узлом и деревом документа и надеяться, что после этого узел будет обнаружен и удален системой "сборки мусора".

Рассмотрим простой пример. Приведенный ниже фрагмент сценария демонстрирует всплывающее окно с сообщением. Для его обнаружения используется выражение document.getElementById().

```
function Message(txt, timeout){
  var box=document.createElement("div");
  box.id="messagebox";
  box.classname="messagebox";
  var txtNode=document.createTextNode(txt);
  box.appendChild(txtNode);
  setTimeout("removeBox('messagebox')", timeout);
}
```

```
function removeBox(id){
  var box=document.getElementByld(id);
  if (box){
    box.style.displays'none';
  }
}
```

При обращении к функции Message () отображается окно с сообщением и устанавливается таймер, который по прошествии определенного времени вызывает другую функцию, удаляющую окно.

Переменные box и txtNode создаются локально и исчезают из области видимости сразу после того, как выполнение функции Message () завершается, но созданные узлы документа остаются доступными, так как они была присоединены к дереву DOM.

Функция- removeBox () удаляет узел DOM по окончании работы с ним. С технической точки зрения сделать это можно различными способами. В примере, рассмотренном выше, мы удалили окно, запретив его отображение. В этом случае элемент, хотя и не отображается, продолжает занимать память. Если нам потребуется снова отобразить его, мы сделаем это без труда.

В случае необходимости мы также можем исключить узел DOM из документа и считать, что *др* того момента, когда система "сборки мусора" обнаружит и удалит его, пройдет немного времени. Заметьте, что мы не разрушаем переменную и не можем контролировать период, в течение которого объект останется в памяти.

```
function removeBox(id){
  var box=document.getElementByld(id);
  if (box && box.parentNode){
    box.parentNode.removeChild(box);
  }
}
```

Мы можем сформулировать два принципа удаления элемента пользовательского интерфейса: *удаление путем сокрытия* (Remove By Hiding) и *удаление путем отсоединения* (Remove By Detachment). Для объекта Message не определены обработчики событий — по прошествии определенного времени он исчезает. Если мы свяжем объект модели предметной области и узлы DOM в двух направлениях, как мы сделали это для объекта Button, и захотим выполнить удаление путем отсоединения, мы должны будем явным образом вызвать функцию cleanUpO.

Оба подхода имеют свои преимущества и недостатки. Главный аргумент в пользу выбора одного из них — это необходимость повторно использовать узел DOM. Если речь идет об окне, предназначенном для отображения сообщения, то, вероятнее всего, оно понадобится нам в дальнейшем, поэтому целесообразно прибегнуть к удалению путем сокрытия. Если же речь идет *о* специфическом элементе, например об узле сложной древовидной структуры, проще будет разрушить элемент по окончании работы с ним, чем хранить ссылки на неиспользуемые узлы.

Выбрав принцип удаления путем еокрытия, мы должны принять соответствующие меры для того, чтобы повторное использование узлов DOM стало возможным. Модифицируем функцию создания окна с сообщением так, чтобы она сначала проверяла, существует ли узел, и создавала новый только в случае необходимости. Перепишем конструктор объекта Message следующим образом:

```
function Message(txt, timeout){
  var box=document.geElementById{"messagebox");
  var txtNode=document.createTextNode(txt);
  if (box==null){
    box=document.createElement("div");
    box.id="messagebox";
    box.classname="messagebox";
    box.style.display='block';
    box.appendChild(txtNode);
  }else{
    var oldTxtNode=box.firstChild;
    box.replaceChild(txtNode,oldTxtNode);
  }
  setTimeout("removeBox('messagebox')",timeout);
}
```

Теперь мы можем сравнить два принципа создания элемента пользовательского интерфейса. Назовем их *созданием при любых условиях* (Create Always) и *созданием при отсутствии* (Create If Not Exists). Создание при любых условиях использовалось нами в исходном примере, а создание при отсутствии — в его модифицированном варианте. Поскольку идентификатор заложен в коде программы, в каждый момент времени может отображаться одно сообщение (такое ограничение нас вполне устраивает). Если мы имеем возможность связать объект модели предметной области с узлом DOM, предназначенным для повторного применения, объект может быть использован для хранения первоначальной ссылки на узел DOM, в результате принцип создания при отсутствии не противоречит наличию нескольких экземпляров объекта.

На заметку При написании приложения Ajax важно помнить о том, что занимать память могут не только объекты, ссылки на которые хранятся в переменных, но и элементы DOM. Также необходимо принимать во внимание тот факт, что элементами DOM можно управлять и удалять их по окончании работы. Если в программе используются как элементы DOM, так и обычные переменные, желательно создавать специальные функции очистки для того, чтобы не допускать существования циклических ссылок.

В следующем разделе мы рассмотрим еще один вопрос, который необходимо принимать во внимание, создавая Ajax-программу, предназначенную для работы в Internet Explorer.

Особенности работы в среде Internet Explorer

В разных браузерах реализованы различные варианты "сборки мусора", **ков** торые могут отличаться друг от друга. Конкретный механизм работы данной системы *в* Internet Explorer мало кому известен в деталях, но участники группы новостей comp. lang. JavaScript сходятся во мнении о том. что эта система испытывает серьезные трудности при обработке циклических ссылок между элементами DOM и обычными объектами JavaScript. По этой причине желательно вручную удалять подобные ссылки.

В качестве примера рассмотрим следующий код, в котором создаются циклические ссылки:

```
function MyObject(id){
  this.id=id;
  this.front=document.createElement("div");
  this.front.backingObj=this;
1
```

Здесь тип MyObject определен пользователем. Каждый экземпляр объекта ссылается на узел DOM с помощью свойства this.front, а в узле DOM с помощью this .backingObj поддерживается обратная ссылка на объект JavaScript.

Для того чтобы удалить циклическую ссылку после окончания работы с объектом, мы можем использовать метод, подобный следующему:

```
MyObject.prototype.finalize=function(){
  this.front.backingObj=null;
  this.front=null;
}
```

Устанавливая значения обоих свойств равным null, мы разрушаем циклическую ссылку.

Для очистки дерева DOM может быть применен универсальный способ, который состоит в обходе дерева и удалении ссылок с определенным именем или определенного типа. Ричард Корнфорд (Richard Cornford) предложил подобную функцию для обработчиков событий, связанных с элементами DOM (ссылка на нее приведена в конце главы).

На наш взгляд, универсальные подходы можно применять лишь в крайних случаях, поскольку обработка больших древовидных структур, типичных для богатых клиентов Ajax, будет происходить слишком медленно. Специальные функции, предусмотренные в коде программы, позволяют выполнять очистку тогда, когда она действительно необходима.

Еще одна особенность Internet Explorer — это наличие высокоуровневой недокументированной функции CollectGarbageO- В Internet Explorer б эта функция существует и может быть вызвана, но похоже, что она не выполняет никаких действий. Нам, во всяком случае, не удалось выявить разницу в объеме используемой памяти до и после обращения к пей.

Теперь, когда вы имеете сведения об управлении памятью, научимся измерять ее объем и применять результаты измерений в реальных приложениях.

8.4. Разработка с учетом производительности

В начале данной главы мы выяснили, что производительность приложения определяется скоростью его выполнения и объемом занимаемой памяти. Мы также упомянули о том, что стандартные программные решения могут помочь добиться высокой производительности программ.

В этом разделе вы узнаете, как измерить объем памяти, занимаемой реальным приложением. Здесь же мы рассмотрим простой пример, показывающий, что причины слишком большого потребления памяти программой иногда могут быть выявлены лишь в результате глубокого анализа используемых алгоритмов.

8.4.1. Измерение объема памяти, занимаемой приложением

Для измерения скорости выполнения программы мы могли применять как JavaScript-код, а именно объекты Date, так и специальные программы. В JavaScript отсутствуют встроенные средства для определения объема используемой памяти, поэтому для решения этой задачи надо полагаться на внешние инструменты. К счастью, такие инструменты существуют.

Выяснить, какой объем памяти потребляет браузер в процессе выполнения приложения, можно различными способами. Самый простой, способ сделать это — анализировать выполняемые процессы с помощью утилит, ориентированных не конкретную операционную систему. В Windows это Task Manager, а в Unix — команда top, вызываемая с консоли. Рассмотрим каждый из этих инструментов.

Windows Task Manager

Программа Windows Task Manager (рис. 8.5) существует для различных версий системы, однако пользователям Windows 95 и 98 она недоступна. Windows Task Manager предоставляет информацию обо всех процессах, выполняемых в операционной системе, и об используемых ими ресурсах. Обычно данная программа вызывается из меню, отображаемого после нажатия комбинации клавиш <Ctrl+Alt+Delete>. Интерфейс Task Manager состоит из нескольких вкладок. Нас интересует вкладка Processes.

Строка, выделенная на рисунке, сообщает нам, что в текущий момент Firefox использует около 38 Мбайт памяти. Информация о потребляемой памяти отображается в столбце Mem Usage. В некоторых версиях Windows пользователь может добавить дополнительные столбцы, используя меню View=>Select Columns (рис. 8.6).

Отображая наряду с Memory Usage значение Virtual Memory Size, можно получить полезную информацию о работе программы. Если Memory Usage представляет активизированную память, выделенную приложению, то Virtual Memory Size отображает информацию о неактивизированной памяти, записанной в раздел свопинга. При минимизации Windows-приложения значение в столбце Mem Usage существенно уменьшается, а Virtual Memory Size становится более или менее постоянным, показывая, какой объем ресурсов приложение может потреблять в дальнейшем.

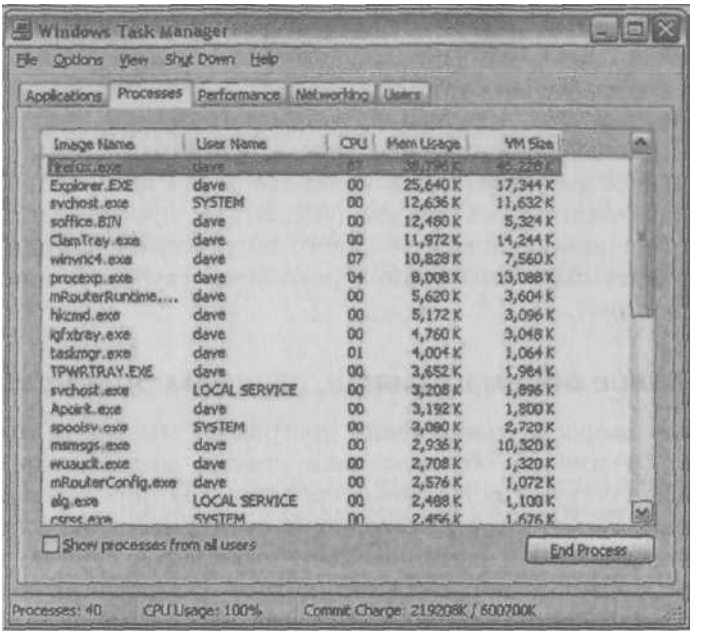


Рис. 8.5. Программа Windows Task Manager отображает список выполняемых процессов и используемую ими память Процессы отсортированы по убыванию объема занимаемой памяти

Утилита top

Консольная программа top, предназначенная для выполнения в системе Unix (а также в Mac OS X), предоставляет приблизительно ту же информацию, что и Windows Task Manager (рис. 8.7).

Как и в случае программы Task Manager, в каждой строке представлены активизированный процесс, объем потребляемой памяти, нагрузка на центральный процессор и некоторые другие сведения. Управление программой top осуществляется с клавиатуры, команды описаны в справочной системе, кроме того, информацию о них можно найти в Интернете. Мы не будем рассматривать подробно программу top, а также аналогичные средства, предоставляющие графический интерфейс, например GNOME System Manager.

Инструменты с расширенными возможностями

Помимо рассмотренных простых инструментов, существуют более мощные средства, которые позволяют контролировать использование памяти, а также предоставляют подробные сведения о внутреннем состоянии операционной системы. Мы не можем уделить внимание всем существующим инструментам, поэтому вкратце рассмотрим лишь два из них. Эти программы распространяются бесплатно, и, на наш взгляд, их использование может помочь разработчикам.

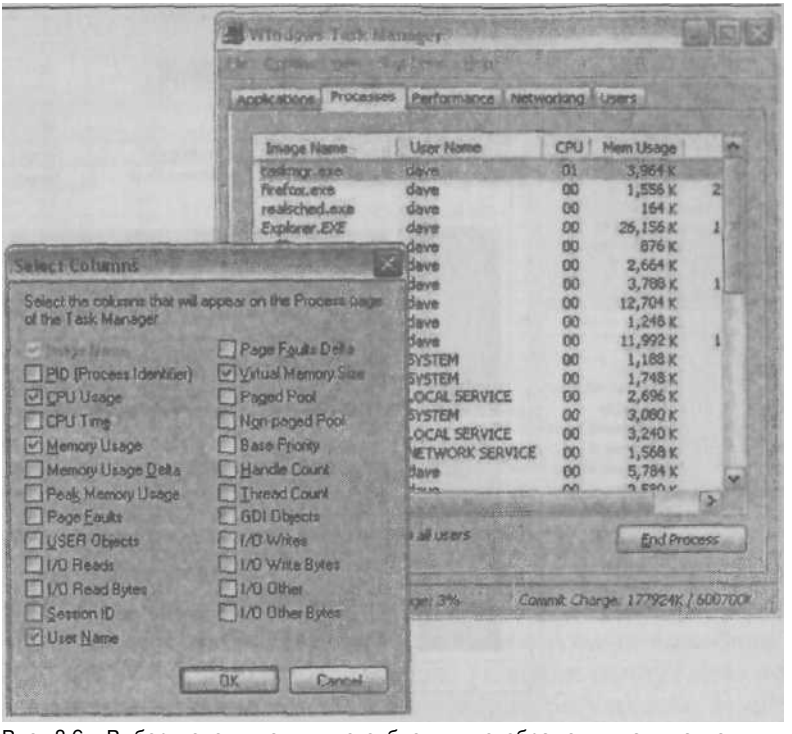


Рис. 8.6. Выбор дополнительных столбцов для отображения на вкладке Processes программы Task Manager. Virtual Memory Size отображает общий объем памяти, выделенной процессу

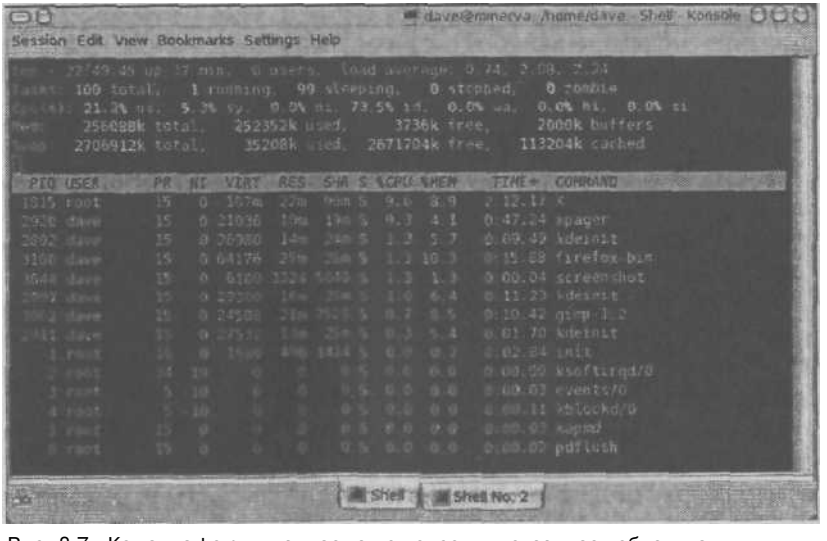


Рис. 8.7. Команда top, вызываемая с консоли, показывает объем памяти, используемой приложением, и нагрузку на центральный процессор

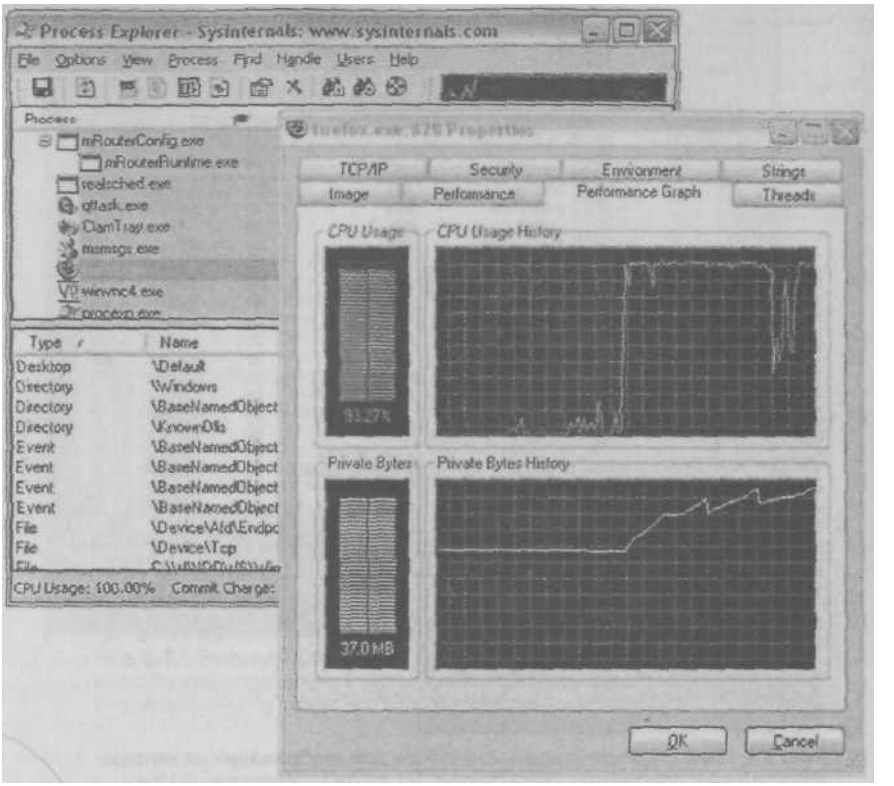


Рис. 8.8. Process Explorer предоставляет подробные сведения об использовании памяти и процессора каждым процессом и позволяет отслеживать особенности работы различных браузеров на *машине под* управлением Windows. В окне, приведенном на этом рисунке, отображается информация о выполнении в среде Mozilla f irefox теста, олисанного в разделе 8 4.2

Инструмент Process *Explorer* производства Sysinternal.com (рис. 8.8) часто в шутку называют "диспетчером задач на стероидах". Он выполняет те же функции, что и Task Manager, *но предоставляет* подробную информацию об использовании памяти и центрального процессора каждым процессом. С его помощью мы можем, например, изучить особенности работы браузера Internet Explorer или Firefox.

Дж. Г. Веббер (J. G. Webber) разработал Drip (ссылка на соответствующий ресурс приведена в конце данной главы) — простой, но мощный инструмент, ориентированный на Internet Explorer и предоставляющий информацию об использовании памяти. Он непосредственно запрашивает Web-браузер об известных ему узлах D0M, в том числе о тех, которые не связаны с деревом документа (рис. *8.9).*

Даже такие простые инструменты могут много сказать о состоянии выполняющегося Ajax-приложения.

Мы рассмотрели детали и особенности, влияющие на производительность

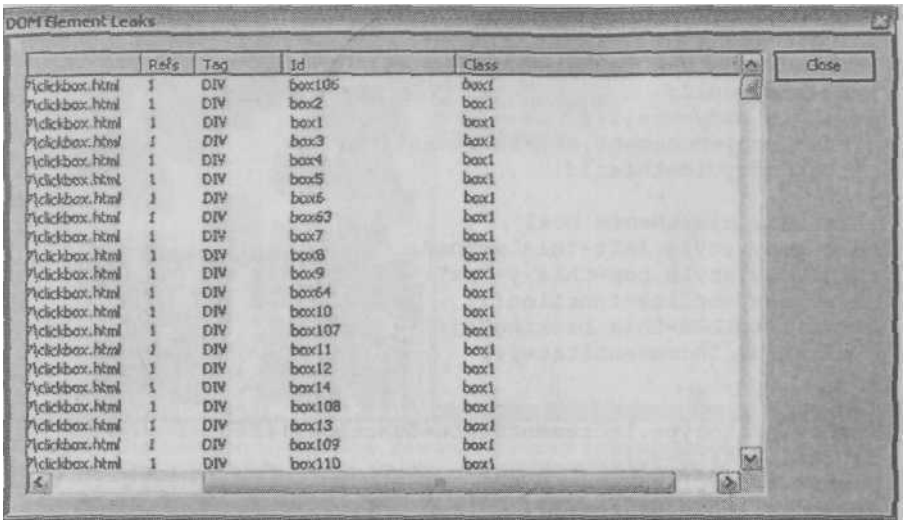


Рис. 8.9. Инструмент Drip предоставляет подробную информацию о внутреннем состоянии дерева DOM браузера Internet Explorer

отдельных фрагментов кода. Если мы напишем Aj ax-приложение хотя бы *среднего* размера, различные программные решения начнут взаимодействовать друг с другом, давая подчас неожиданные результаты. В следующем разделе мы рассмотрим пример, подчеркивающий, насколько важно знать конкретные особенности выполнения кода.

8.4.2. *Простой пример управления памятью*

На данный момент вы уже имеете основные сведения об управлении памятью и знакомы с некоторыми подходами, упрощающими создание интерфейсных элементов из программы. При разработке реального приложения Ajax мы используем ряд приемов, результаты применения которых могут в общем случае влиять друг на друга. Каждое программное решение оказывает влияние на производительность; то же справедливо для их совместного применения. Результаты использования различных подходов удобно проиллюстрировать на конкретном примере, который мы и рассмотрим в данном разделе.

При работе программы, рассматриваемой здесь в качестве примера, создаются и удаляются компоненты ciickBox. Они называются так потому, что представляют собой небольшие квадраты, реагирующие на щелчки мышью. Поведение каждого компонента определяется следующим кодом:

```
function CiickBox(container){
  this.x=5+Math.floor(Math.random*)*370);
  this.y=5+Math.floor(Math.random()*370);
  this.id="box"+contailer-boxes.length;
  this.state=0;
  this.render();
  container.add(this);
```

```
}

ClickBox.prototype.render=function(){
  this.body=null;
  if  (this.body==null){
    this.body=document.createElement("div");
    this.body.id=this.id;
  }
  this.body.className='box1';
  this.body.style.left=this.x+"px";
  this.body.style.top=this.y+"px";
  this.body.onclick=function(){
    var  clickbox=this.backingObj;
    clickbox.incrementState();
  }
}
ClickBox.prototype.incrementState^function(){
  if  (this.state==O){
    this.body.className='box2' ;
  }else if  (this.state==l){
    this.hide{);
  }
  this.state++;
}
ClickBox.prototype.hide=function(){
  var  bod=this.body;
  bod.className='box3' ;
}
```

Компонент ClickBox имеет красный цвет, если отображается впервые. После того как пользователь один раз щелкнет на нем мышью, цвет компонента изменяется на синий. После второго щелчка компонент удаляется с экрана. Данное поведение реализуется путем создания двусторонних ссылок между объектами модели предметной области и элементами DOM, представляющими их на экране.

В программе для каждого ClickBox поддерживается уникальный идентификатор, информация о позиции, сведения о внутреннем состоянии (сколько щелчков было сделано) и тело компонента. Телом компонента является узел DOM типа div. Узел DOM содержит ссылку на объект модели в переменной backingObj.

В программе также определен класс Container, содержащий объекты ClickBox. В данном классе содержится массив компонентов и уникальный идентификатор самого контейнера.

```
function  Container(id){
  this.id=id;
  this.body=document.getElementBy1d(id) ;
  this.boxes=new Array();
}
Container.prototype.add=function(box){
  this.boxes[this.boxes.length]=box;
  this.body.appendChild(box.body);
}
Container.prototype.clear=function(){
```

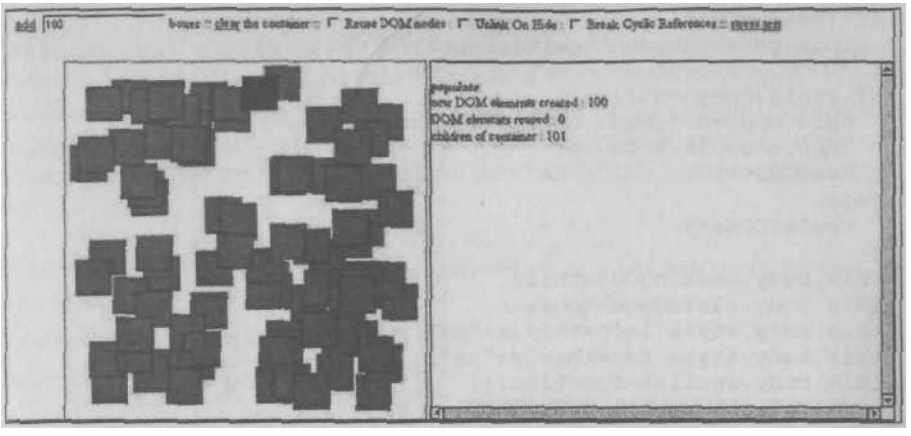


Рис. 8.10. Так выглядит приложение, демонстрирующее использование памяти, после создания первых 100 компонентов. Компоненты реагируют на щелчки мышью

```
  for(var i=0;i<this.boxes.length;i++){
    this.boxes[i].hide();
  }
  this.boxes^new Array();
  report("clear");
  newDOMs=0;
  reusedDOMs=0;
}
```

Внешний вид окна приложения показан на рис. 8.10.

На отладочной панели, расположенной справа, отображается информация о внутреннем состоянии системы. На состояние системы влияют события, например, добавление компонентов к контейнеру или их удаление.

Данная программа была написана для того, чтобы в процессе ее работы можно было поэкспериментировать с созданием и удалением элементов DOM и обработкой циклических ссылок. Пользователь может выбирать различные приемы работы с программой, устанавливая или удаляя флажки опций в HTML-форме, расположенной на странице. При активизации ссылок, управляющих добавлением элементов в контейнер или их удалением, реализуются программные решения, определяемые набором установленных флажков. Рассмотрим каждый из этих флажков и соответствующий ему код.

Опция Reuse DOM Nodes

Данная опция определяет, должен ли компонент ClickBox при отображении пытаться найти существующий узел DOM и создавать новый узел, только если поиск закончился неудачей. Другими словами, флажок этой опции осуществляет переключение между режимами создания при любых условиях и создания при отсутствии (см. раздел 8.3.2). Модифицированный код, отвечающий за воспроизведение компонента, выглядит следующим образом:

```
ClickBox.prototype.render=function(){
  this.body^null;
```

```
  if (reuseDOM){
    this.body=document.getElementById(this.   id);
  }
  if (this.body==null){
    this.body=document.createElement("div");
    this.body.id=this.id;
    newD0Ms++;
  }else{
    reusedDOMs++;
  }
  this.body.backingObj=this;
  this.body.className='boxl';
  this.body.style.left=this.x+"px";
  this.body.style.top=this.y+"px";
  this.body.onclick=function(){
    var clickbox=this.backingObj;
    clickbox.incrementState();
  }
}
```

Опция Unlink On Hide

Когда компонент ClickBox удаляется из контейнера (либо после второго щелчка мышью, либо в результате вызова Container. clear()), данная опция определяет, должно ли производиться удаление путем сокрытия или удаление путем отсоединения. Об этих принципах удаления см. в разделе 8.3.2.

```
ClickBox.prototype.hide=function(){
  var  bod=this.body;
  bod.className='box3';
  if   (unlinkOnHide){
    bod.parentNode.removeChiId(bod};
  }

}
                                /
```

Опция Break Cyclic References

Данная опция также имеет отношение к процессу удаления компонента ClickBox. Она определяет, должны ли присваиваться значения null свойствам элемента D0M и объекта модели предметной области, т.е. должны ли удаляться циклические ссылки.

```
ClickBox.prototype.hide=function(){
  var  bod=this.body;
  bod,className='box3' ;
  if   (unlinkOnHide){
    bod.parentNode.removeChild(bod);
  }
 if  {breakCyclics){
    bod.backingObj=null;
    this.body=null;
  I
}
```

Управляющие элементы формы позволяют пользователю помещать з контейнер новые компоненты ClickBoxes и очищать контейнер. Приложением можно управлять вручную, но для удобства сбора результатов мы написали также тестирующую функцию, которая имитирует некоторые действия пользователя. Приведенная ниже последовательность действий повторяется 240 раз.

1. Включение 100 компонентов в контейнер с помощью функции populate ().
2. Добавление 100 компонентов.
3. Очистка контейнера.

Код функции stressTest() приведен ниже.

```
function stressTest() {
  for (var i=Q;i<240;i++){
     populate (100);
     populate(100);
     container.clear();
  }
  alert("done");
{
```

Заметьте, что в данном случае проверяется добавление компонентов к контейнеру и удаление их, а не поведение элементов ClickBox по щелчку мышью.

Данная проверка умышленно упрощена. Если вам надо лишь выяснить, увеличивается или уменьшается объем потребляемой памяти при изменении алгоритма, мы рекомендуем создавать для ваших приложений такие же простые тесты. Создание тестовых сценариев уже само по себе требует определенного мастерства, в частности, от разработчика требуется знание типичных приемов проектирования.

Выполнение тестового сценария занимает около минуты. В течение этого времени браузер не реагирует на действия пользователя. Если число итераций выбрать слишком большим, браузер может "зависнуть" на длительное время. Если число повторений мало, изменения потребляемой памяти не будут заметны. Мы выбрали 240 итераций и считаем, что для нашей машины это вполне приемлемое значение. Вы же можете изменить его, исходя из конкретных условий.

Запись изменения занимаемой памяти — сравнительно простая задача. Мы выполняем тестирование в операционной системе Windows при работающей программе Task Manager. Объем потребляемой памяти регистрируется непосредственно после загрузки тестовой страницы, а затем снова после вывода окна, сообщающего о том, что тестирование завершено. В системе Unix для тестирования может быть использована утилита top или другой инструмент аналогичного назначения (см. раздел 8.4.1). После каждой проверки мы закрываем браузер, чтобы обеспечить равные условия тестирования.

Таблица 8.4. Результаты тестирования

Идентификатор	Флажок Reuse DOM Nodes	Флажок Unlink On Hide	Флажок Break Cyclic References	Занимаемая память после окончания работы (Internet Explorer)
A	Сброшен	Сброшен	Сброшен	166 Мбайт
B	Сброшен	Сброшен	Установлен	84,5 Мбайт
C	Сброшен	Установлен	Сброшен	428 Мбайт
D	Установлен	Сброшен	Сброшен	14,9 Мбайт
E	Установлен	Сброшен	Установлен	14,6 Мбайт
F	Установлен	Установлен	Сброшен	574 Мбайт
G	Установлен	Установлен	Установлен	14,2 Мбайт

Такова методология проверки. В следующем разделе мы обсудим ее результаты.

8.4.3. Как уменьшить объем используемой памяти в 150 раз

Выполняя тестирование и пользуясь данными, предоставляемыми Windows Task Manager, можно сделать вывод о том, что объем используемой памяти зависит от алгоритмов, выбранных посредством флажков опций. Результаты проверки приведены в табл. 8.4.

Тестирование производилось на рабочей станции с тактовой частотой процессора 2,8 ГГц и 1 Гбайт оперативной памяти, работающей под управлением операционной системы Windows 2000 Workstation. В качестве браузера был выбран Internet Explorer 6. Во всех случаях начальное потребление памяти составляло приблизительно 11,5 Мбайт. Информация об использовании памяти отображалась в столбце Mem Usage на вкладке Processes программы Task Manager (см. раздел 8.4.1).

Поскольку мы сопоставляем реальные значения, следует заметить, что само приложение занимает достаточно много памяти. Ajax-программы считаются "тонкими" клиентами, однако при определенных ошибках в коде или их сочетании они могут потреблять невероятно много памяти.

Необходимо также заметить, что выбор конкретных программных решений оказывает огромное влияние на объем занимаемой памяти. Рассмотрим результаты более подробно. При трех сочетаниях флажков опций после отображения и удаления всех компонентов clickBox объем памяти оказывался менее 15 Мбайт. При остальных наборах опций это значение возрастало до 80, 160 и даже до 430 и 580 Мбайт. Учитывая, что сам браузер потребляет 11,5 Мбайт памяти, размер дополнительно использованной памяти лежал в пределах от 3,5 до 570 Мбайт, т.е. мы можем сократить его в 150 раз, правильно выбрав программные решения. Стоит также отметить, что браузер продолжал функционировать при любой утечке памяти.

Ни один из алгоритмов нельзя признать виновником происходящего. *у[ежду* ними существует очень сложная взаимозависимость. Сравнивая, например, проверки A, D и F, мы видим, что переключение значения опции Reuse DOM Nodes дает огромную экономию памяти {больше 90%), в то же время переключение Unlink On Hide приводит к тройному увеличению ее похребления. В данном конкретном случае причины происходящего понятны — поскольку связи для узлов DOM разрываются, они не могут быть найдены путем вызова document .getElementById(), следовательно, их невозможно использовать повторно. Аналогично, изменение значения опции Unlink on Hide приводит к увеличению потребления памяти по сравнению с исходной конфигурацией (проверки C и A). Перед тем как объявлять опцию Unlink on Hide виновницей излишнего расхода памяти, сравним проверки E и G — в правильном контексте она дает некоторый положительный результат.

Интересно отметить, что в этом "состязании" нет победителя: три совершенно различные комбинации значений опций дают лишь незначительное различие в объеме потребляемой памяти. Во всех трех случаях установлен флажок Reuse DOM Nodes, но эта же опция в другом сочетании приводит к максимальному расходу памяти. Однозначного вывода из имеющихся результатов сделать нельзя, но мы можем выделить сочетания алгоритмов, которые дают хорошие результаты, и другие наборы, которые использовать не следует. Если мы проанализируем программные решения и дадим им имена, будет значительно проще применять их при работе над приложением для получения приемлемых результатов. Если мы не будем использовать фиксированный набор программных решений, а будем по наитию создавать каждую подсистему, работающую с DOM, каждый новый фрагмент кода будет давать непредсказуемые результаты: возможно, его выполнение приведет к значительному расходованию памяти, а может быть, он будет работать достаточно экономно.

В ходе подобных проверок разработчик получает информацию, позволяющую решить многие вопросы, связанные с разработкой богатых DHTML-клиентов, выполняющихся в среде браузера в течение длительного времени, и выявить ряд ошибок.

Чтобы реально контролировать вопросы, связанные с использованием памяти, надо помнить о них в процессе разработки. Модифицируя код, полезно подумать о том, как эти изменения скажутся на использовании памяти. Кроме того, после внесения изменений всегда надо проверять, как память используется программой.

Несмотря на то что стандартные программные решения помогают в разработке, проблемы с памятью вновь могут возникнуть при незначительном изменении условий. Так, например, мы знаем, что алгоритм устранения циклических ссылок может давать неожиданные результаты, взаимодействуя с решением, которое предполагает создание узлов лишь в случае их отсутствия. Детально представляя себе суть того или иного программного решения, вы уменьшаете вероятность возникновения нежелательных последствий.

Использование типичных программных решений упрощает задачу создания тестовых сценариев. Написание программ или сценариев для тестирова-

ния, наверное, самая сложная часть работы, поскольку для ее выполнений надо знать, как пользователь будет взаимодействовать с приложением. Не исключено, что программа будет предназначена для нескольких категорий пользователей; в этом случае придется разработать несколько тестирующих сценариев. Используя одну программу, имитирующую некоего усредненного пользоваеля. можно упустить из виду важные детали, оказывающие существенное влияние на результаты работы. И наконец, готовую программу проверки нельзя рассматривать как нечто неизменное. При модификации приложения вполне может потребоваться внесение изменений в сценарий тестирования.

8.5. Резюме

Производительность любой компьютерной программы в основном определяется скоростью ее выполнения и объемом потребляемой памяти. В случае приложения Ajax мы имеем дело со специальной средой, имеющей мало общего с операционной системой и, тем более, с аппаратным обеспечением. Тем не менее, выбирая то или иное программное решение, мы можем существенно влиять на производительность программы.

Мы рассмотрели вопросы профилирования, причем использовали для создания профилей как библиотеки JavaScript, так и встроенные инструменты, например отладчик Venkman. Профилирование позволяет нам выявить узкие места системы, а также определить исходные точки отсчета для оценки влияния вносимых изменений. Сравнивая результаты профилирования до и после модификации кода, мы можем оценить влияние внесенных изменений на работу программы.

Мы также рассмотрели вопросы управления памятью и показали, как можно избежать утечки памяти при работе программы. При этом было уделено внимание как универсальным решениям, так и подходам, ориентированным на конкретные обстоятельства, например, использованию структуры DOM или работе с Internet Explorer.,Вы научились определять объемы памяти, потребляемой приложением, используя для этой цели инструменты, ориентированные на операционные системы Windows и Unix.

И наконец, мы рассмотрели пример, демонстрирующий важность принятия тех или иных решений при написании программы. Программные решения, примененные при разработке приложения, могут существенно влиять на потребление памяти и возхможность управления ею.

Работая над повышением производительности программы, никогда нельзя утверждать, что конечная цель достигнута, — на каждом этапе работы остается возможность некоторой оптимизации кода. Поэтому, оптимизируя Ajax-приложение, важно вовремя сказать "'достаточно" и не затрачивать большие усилия на получение скромных результатов. Для того чтобы принять решение об окончании работы по оптимизации, необходима информация о реальной производительности программы. В данной главе были рассмотрены инструменты, способные предоставить эту информацию разработчику.

8^6. Ресурсы

0 данной главе мы рассмотрели несколько инструментов.

- Drip, детектор утечки памяти, разработанный Джоэлом Веббером (Joel Webber) и ориентированный на Internet Explorer. Этот продукт доступен по адресу http://www.outofhanwell.com/ieleak/.
- Профилировщик Venkman (http://www.svendtofte.com/code/learning_yenkman/advanced).
- Process Explorer (http://www.sysinternals.com).

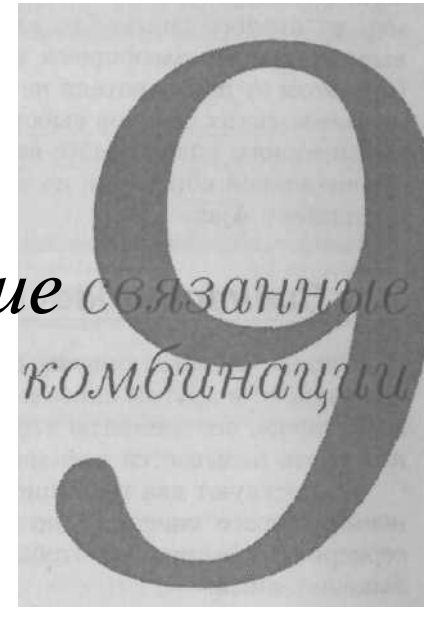
Официальная информация о проблемах использован ш памяти браузером Internet Explorer и путях их устранения представлена по адресу http://msdn.microsoft.com/library/default.asp?url=/library/en-us/lETechCol/dnwebgen/ie_leak__patterns.asp. Сведения о решениях, предложенных Ричардом Корнфордом (Richard Cornford), можно получить, осуществив поиск посредством Google Groups. В качестве ключевых слов надо указать cornford javascript fixCircleRef s () (полный URL слишком длинный, чтобы приводить его здесь).

Часть IV

Ajax в примерах

X? данной части приводится пять завершенных проектов Ajax, демонстрирующих весь процесс создания неотразимых интерактивных элементов для вашего Web-приложения. В каждом случае мы последовательно разберем код одного примера и покажем, как он работает. Затем реструктуризируем код, чтобы вы могли использовать его в собственных проектах. Предлагаемые примеры охватывают весь спектр действий, доступных при использовании Ajax: от улучшения элементов форм до разработки завершенных порталов, сообщающихся как с вашими серверными процессами, так и со стандартными службами Интернета. При реализации кода серверной стороны мы специально задействовали смесь популярных языков программирования, поэтому в данном разделе вам встретятся PHP, Java, VB.Net и C#. В коде, который можно загрузить с Web-сайта данной книги, представлено несколько реализаций серверного кода, разобранного в данной главе. Не скучайте!

9 Динамические связанные комбинации



В этой главе...

- Клиентский код JavaScript
- Серверный код VB.NET
- Формат обмена данными
- Реструктуризация для повторного использования
- Динамические окна выбора

Если вы когда-либо покупали рубашку в интерактивном магазине, то мог-Я ли столкнуться со следующей проблемой. Вы выбираете размер рубашки из раскрывающегося списка, а из следующего списка выбираете ее цвет. Затем вы отправляете заполненную форму и получаете сообщение 'Товара нет на складе", написанное гигантскими буквами. С чувством глубокого разочарования вы щелкаете на кнопке Назад или на предложенной ссылке, а затем выбираете новый цвет.

Используя Ajax, подобного срыва планов можно избежать. Например списки выбора можно связать так, чтобы, когда пользователь выберет размер из первого списка, во второй список непосредственно из базы данных выводились все имеющиеся в наличии расцветки рубашек данного размера (при этом от пользователя не требуется обновлять страницу). Ранее связывание нескольких списков выбора для выполнения указанной задачи требовало интенсивного кодирования на JavaScript с использованием массивов или до-1полнительной обработки на сервере, но теперь есть лучший способ, который предлагает Ajax.

9.1. Сценарий двойной комбинации

В двойных связных списках содержимое одного списка зависит от варианта, выбранного в другом списке. Когда пользователь выбирает значение из первого списка, все элементы второго динамически обновляются. Обычно подобная схема называется *сценарием двойной комбинации* (double-combo script). \

Существуют два традиционных решения проблемы динамического заполнения второго списка: одно реализуется на стороне клиента, второе — на сервере. Напомним их, чтобы понять принцип их действия и вопросы, связанные с ними.

9.1,1. Недостатки клиентского решения

Традиционно первое, о чем думают разработчики, — это использовать решение, переносящее всю работу на сторону клиента. В подобном решении, основанном на JavaScript, значения списка жестко кодируются на Web-странице в массивах JavaScript. После того как вы выбрали размер рубашки, сценарий заполняет следующий список, выбирая значения из массива. Данное решение показано на рис. 9.1.

С данным клиентским методом связана одна проблема: поскольку пользователь не связывается с сервером, на момент первого выбора массив может оказаться устаревшим. Другую проблему составляет время загрузки исходной страницы, на которое очень сильно влияет количество позиций в обоих списках. Представьте себе магазин с тысячей предметов; в массив JavaScript необходимо поместить цену каждой вещи. Поскольку код, представляющий данный массив, будет частью страницы, пользователю придется довольно долго ждать первой загрузки сайта (пользователь никак не может заблаговременно получить данную информацию). С другой стороны, метод JavaScript имеет и одно преимущество: после первой загрузки все происхо-

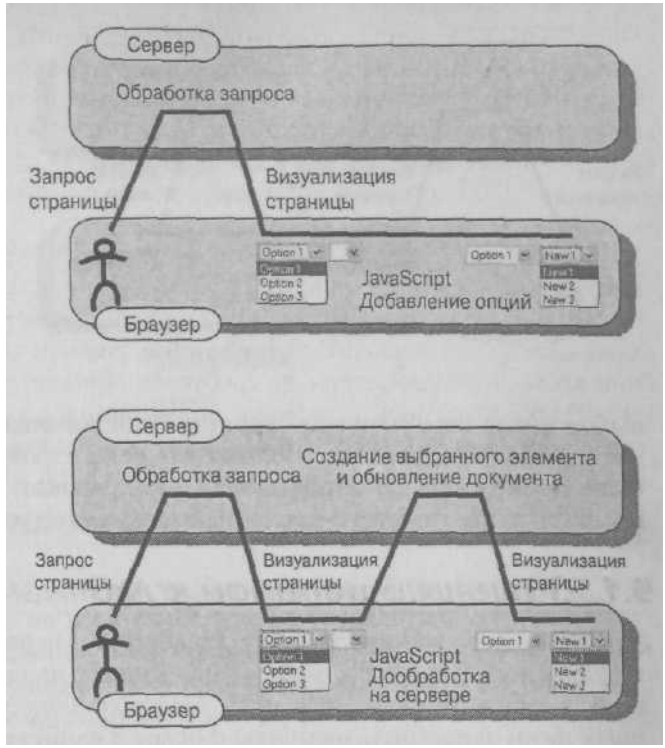


рис. 9.1. Клиентское решение

Рис. 9.2. Метод дообработки на сервере

дит достаточно быстро. Никакой заметной задержки между выбором позиции первого списка и появлением второго не наблюдается. Таким образом, данный метод подходит только в том случае, если требуется реализовать лишь несколько связных списков, которые не очень сильно повлияют на время загрузки страницы.

9.1.*2. Недостатки клиентского решения*

Следующее традиционное решение заключается в возврате формы на сервер Для дообработки. В данном методе обработчик событий onchange в первом списке инициирует возврат страницы на сервер с помощью метода submit (), введенного в JavaScript-представление формы. В результате страница отправляется на сервер, донося выбранный пользователем элемент первого списка. Сервер, в свою очередь, запрашивает базу данных согласно значению, которое выбрал пользователь, и динамически заполняет второй список новыми значениями, вызывая повторное отображение страницы. Действия данного серверного метода представлены на рис. 9.2.

Недостатком описанного серверного метода является число обращений к серверу; при каждой перезагрузке страницы наблюдается задержка, вызванная необходимостью копирования по сети всей страницы. Визуально данная структура выглядит так, как показано на рис. 9.2. Кроме того, на стороне сервера необходимо реализовать дополнительный код, отвечающий за

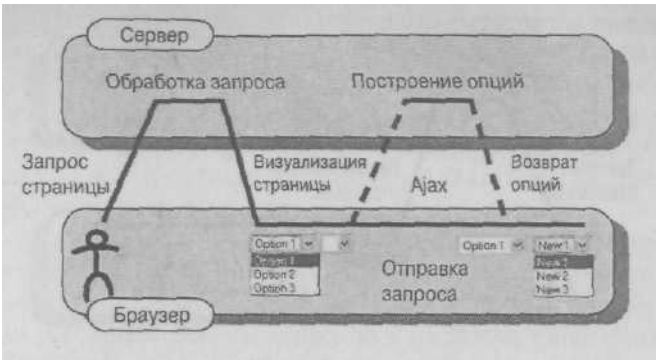


Рис. 9.3. Решение, предлагаемое Ajax

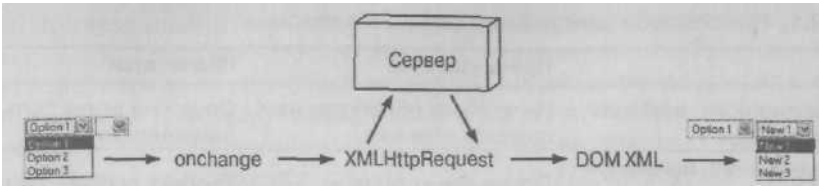
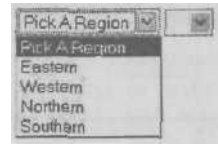


Рис. 9.4. Архитектура клиентской части приложения с показанной инфраструктурой Ajax

Рис. 9.5. Доступные элементы первого списка select



9.2. Архитектура клиента

Большинство разработчиков, обычно занимающихся написанием кода для сервера, практически ничего не знают об архитектуре клиента. В нашем случае все не так страшно, поскольку нам требуется лишь несколько действий, позволяющих занести ряд позиций во второй список. Если вам уже приходилось реализовывать описанные выше комбинированные списки (с использованием JavaScript или серверного подхода), то вы уже примерно знаете, с чем вам придется столкнуться.

Как видно на рис. 9.4, взаимодействие данного приложения со стороной клиента проходит в три этапа. Первый этап — это построение исходной формы. Затем пользователь выбирает элемент из первого списка select. На втором этапе требуется создать объект XMLHttpRequest, взаимодействующий с сервером. Этот объект передает выбор пользователя серверу вместе с именами формы и элементами управления, которые будут обновлены при получении отклика сервера. Третий этап — это добавление содержимого XML-отклика сервера ко второму элементу select. Для разбора XML-отклика используются методы DOM XML, реализованные на JavaScript.

Сейчас мы разберем первые два этапа, которые выполняются до отправки запроса Ajax серверу. Третий этап (взаимодействие DOM с документом XML-отклика сервера) мы подробно разберем в разделе 9.4, поскольку, перед тем, как завершить реализацию архитектуры клиента, нам нужно будет поговорить о сервере.

9.2.1. Разработка формы

В данном примере форма содержит два элемента select. Изначально первый из них хранит значения, а второй список пуст. Внешний вид полученной формы показан на рис. 9.5.

Таблица 9.1. Три способа заполнения элемента формы

Метод	Преимущества	Недостатки
Жестко закодировать значения в элементе select	Не требует обработки на стороне сервера	Опции не могут быть динамическими
Заполнить значения, используя сценарий серверной части приложения	Опции могут быть динамическими и заполняться из базы данных	Требует дополнительной обработки на сервере
Использовать Ajax для заполнения формы значениями; для извлечения значений требуется дообработка на сервере	Можно связать с другими значениями на странице	Требует дополнительной обработки на сервере

Как показано в табл. 9.1, первый элемент формы можно заполнить тремя различными способами.

Первый метод — жестко закодировать значения в элементе select. Данный метод хорош, когда вы используете несколько опций, которые не будут меняться. Второй метод — заполнить значения, используя сценарий серверной части приложения. При таком подходе опции заполняются при визуализации страницы, что позволяет извлекать их из базы данных или файла XML. Третий метод — использовать для заполнения значений инфраструктуру Ajax; этот метод требует дообработки на сервере (извлечения значений), но не требует повторной визуализации всей страницы.

В данном примере мы жестко закодируем значения в списке, поскольку используется всего четыре опции, которые не являются динамическими. Наиболее удачным решением задачи динамической загрузки значений в первый список является использование серверного сценария, заполняющего список в процессе визуализации страницу. В данном случае для заполнения первого списка Ajax использовать не стоит (если только содержимое списка не зависит от других значений, выбранных пользователем в форме).

Как показано в листинге 9.1, первый список должен содержать обработчик событий onchange, добавленный к элементу select. Этот обработчик событий вызывает функцию FillTerritoryO (JavaScript), которая инициирует процесс заполнения второго списка, отправляя запрос серверу.

Листинг 9.1. Форма со связными списками

```
<form name="Forml"> <select name="ddlRegion"
  onchange="FillTerritory(this,document.Forml.ddlTerritory)">
    <option value="-l">Pick A Region</option>
    <option value="l">Eastern</option>
    <option value="2">Western</option>
    <option value-"3">Northern</option>
    <option value="4">Southern</option>
</select> <select name="ddlTerritory"></select> </form>
```

Код, приведенный в листинге 9.1, создает форму, инициирующую процесс fillTerritory{) при выборе элемента из первого списка. Функции FillTer-i-itoryO мы передаем две объектные ссылки на элемент. Первая из них — объект списка, к которому прикреплен обработчик событий, вторая — список, который требуется заполнить. Далее нам необходимо разработать код серверной части приложения для FillTerritory(), который бы отправлял „аш запрос серверу.

9.2.2. Разработка взаимодействия клиент/сервер

Основной целью функции FillTerritory() является сбор информации, необходимой для отправки запроса серверу. Эта информация включает выбранную опцию из первого списка, имя формы и имя второго списка. Имея эти данные, мы можем использовать функции Ajax в нашей библиотеке JavaScript для отправки запроса серверу. Итак, первое, что требуется сделать, — это реализовать функциональные возможности Ajax. Код, требуемый для связи с внешним файлом JavaScript net.js (определяет объект Content-Loader), тривиален. Все, что требуется, — добавить приведенную ниже строку между дескрипторами заголовка вашего документа HTML.

```
<script type="text/javascript" src="net.js"x/script>
```

Объект ContentLoader выполняет всю работу, определяя, как отправить запрос серверу, скрывая при этом весь код (зависящий от используемого браузера) в удобном интерфейсном объекте, который мы ввели в главе 3. Это позволяет обмениваться данными с сервером, не обновляя страницу.

Добавив функциональные возможности Ajax, мы сможем построить функцию FillTerritoryO, показанную в листинге 9.2; данный код также включается в заголовок нашего документа.

Листинг 9.2. Функция FillTerritoryU инициализирует запрос Ajax

```
<script type="text/javascript">
function FillTerritory(oElem,©Target){
// О Получить значение, выбранное в списке
  var strValue = oElem.options[
oElem.selectedlndex].value;
// © Установить целевой URL
  var url = "DoubleComboXML.aspx";
// © Построить строку параметров
  var strParams = "q=" + strValue +
"&f=" + oTarget.form.name +
"&e=" + oTarget.name;
// О Инициализировать загрузчик содержимого
  var loaderl = new
  net.ContentLoader(url,FillDropDown,null,
"POST",strParams);
```

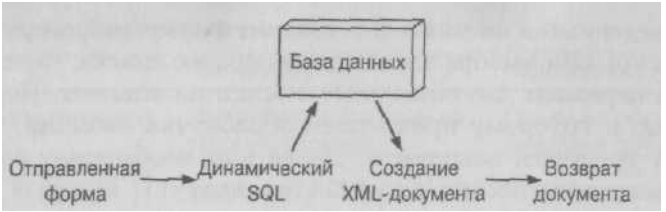


Рис. 9.6. Поток процесса на стороне сервера

Функция FillTerritory() принимает два параметра, *в* данном случае передаваемые от обработчика событий onchange из первого списка. Эти параметры — ссылки на первый и второй элементы select О. Мы обращаемся к значению, которое пользователь выбрал в первом списке ®, и устанавливаем URL целевого серверного сценария ©. Далее мы создаем параметры, которые будут отправлены серверу: формируем строку с таким же синтаксисом, как и строка запроса, используя амперсанд для разделения пар "имя-значение". В данном примере мы отправляем значение, представляющее выбранную область, как q, имя формы —- как f, а имя второго элемента select — как e. Серверная часть кода использует значение выбранной области для запроса базы данных и отправит имена формы и элемента select клиенту в ответном документе XML. С помощью этой информации клиент определит, какую форму и элемент управления обновить. Создав строку параметров, остается только инициировать процесс Ajax.

Чтобы начать процесс О, мы вызываем конструктор ContentLoaderO и передаем целевой URL, функцию, вызываемую при получении отклика сервера, функцию обработки ошибок, используемый метод HTTP и отправляемые параметры. В данном случае при возврате данных с сервера будет вызываться функция FiliDropDown(), в качестве функции обработки ошибок устанавливается функция, по умолчанию принятая ContentLoader, кроме того, мы используем запрос POST.

В это время ContentLoader ожидает возврата от сервера документа XMI* Мы продолжим разбор клиентской части кода в разделе 9.4, а пока изучим, что должен сейчас сделать сервер.

9.3. Реализация сервера: VB.NET

В коде серверной части приложения необходимо реализовать извлечение территорий, принадлежащих к выбранной клиентом области, из базы данных и возврат их клиенту в документе XML. Согласно набору элементов, полученных из запроса SQL, создается документ XML, который и возвращается клиенту. Поток процесса на стороне сервера показан на рис. 9.6.

Код на стороне сервера активизируется запросом объекта ContentLoader со стороны клиента. Вначале код сервера извлекает значение параметра запроса q, представляющего выбранную область. Согласно значению q создается динамический запрос SQL, который, обращаясь к базе данных, находит

пары "текст-значение" для второго раскрывающегося списка. Затем данные, полученные из базы данных в ответ на запрос, форматируются в формате XML и возвращаются клиенту. Прежде чем написать код, выполняющий описанные действия, нам необходимо определить основную структуру документа XML.

9.3.1- Определение формата XML-ответа

Нам требуется создать простой документ XML, возвращающий клиенту результаты нашего запроса к базе данных. Этот документ будет содержать опции, которыми будет заполняться второй список. Для представления каждой опции требуются два элемента: один, содержащий текст опции, второй — ее значение.

Документ XML в нашем примере содержит корневой элемент selectchoice, вмещающий единственный элемент selectElement, за которым следует один или несколько элементов entry. Элемент selectElement содержит имена формы HTML из раскрывающегося списка, который будет заполнен результатами, полученными из базы данных. Каждый элемент entry содержит два дочерних элемента, optionText и optionValue, которые вмещают значения, представляющие описание и идентификатор каждой территории. Описанная структура показана в листинге 9.3.

Листинг 9.3. Пример формата XML-ответа

```
<?xml version="1.0" ?>
<selectChoice>
  <selectElement>
    <formName>Forml</formName>
    <formElem>ddlTerritory</formElem>
  </selectElement>
  <entry>
    <optionText>Select A Territory</optionText>
    <optionValue>-l</optionValue>
  </entry>
  <entry>
    <optionText>TerritoryDescription</optionText>
    <optionValue>TerritoryID</optionValue>
  </entry>
</selectChoice>
```

Изучая пример документа XML в листинге 9.3, обратите внимание на наличие позиции, содержащей текст опции "Select A Territory" ("Выберите территорию"). Это первая опция, отображаемая в списке, она предлагает пользователю выбрать значение. Код сервера включает это значение в начало каждого ответного документа до получения динамических опций из базы данных.

Теперь, определив ответный документ, можно писать код, динамически создающий XML-документ и возвращающий его клиенту.

9.3.2. Написание кода сервера

Код VB.NET серверной части приложения достаточно прямолинеен. Мы запрашиваем базу данных, которая возвращает набор записей. После этого мы циклически проходим по набору записей и создаем XML-документ, который затем возвращаем клиенту. Если мы не найдем ни одной записи, тогда элементы `entry` не создаются и статическая опция "Select A Territory" опускается. Как показано в листинге 9.4, серверная часть кода не очень сложна. Она просто содержит операторы, извлекающие значения формы, отправленные на сервер, устанавливает тип содержимого, выполняет поиск и выдает документ XML.

В приведенном примере использована база данных Northwind из SQL Server (Microsoft).

Листинг 9.4. DoubleConiboXML.aspx. vb: создание XML-ответа на сервере^

```
// Реализация метода Page_Load
Private Sub Page_Load( _
            ByVal sender As System.Object, _
            ByVal e As System.EventArgs) _
            Handles MyBase.Load
// О Установить тип содержимого
  Response.ContentType = "text/xml"
// © Извлечь отправленные данные
  Dim strQuery As String
  strQuery • Request.Form("q")
  Dim strForm As String
  strForm *= Request.Form("f")         \       _ /
  Dim strElem As String
  strElem = Request.Form("e")
// © Создать утверждение SQL
  Dim strSql As String = "SELECT " &
                "TerritoryDescription, " & _
                "TerritorylD" & _
                " FROM Territories" & __
                " WHERE regionid = " & _
                strQuery & " ORDER BY " & _
                "TerritoryDescription"
  Dim dtOptions As DataTable
// О Выполнить утверждение SQL
  dtOptions = FillDataTable(strSql)
// © Начать документ XML
  Dim strXML As StringBuilder
  strXML = New StringBuilder("<?xml " & _
                    "version=""1,0"" ?>")
  strXML.Append{"<selectChoice>")
  strXML.Append("<selectElement>")
  strXML.Append("<formName>" & _
                strForm & _
                "</formName>")
  strXML.Append("<formElem>" & _
                strElem &
                "</formElem>")
  strXML.Append("</selectElement>")
```

```
//O Проверить наличие результатов
  If dtOptions.Rows.Count > 0 Then
//в Добавить первый элемент выбора
    strXML.Append("<entry>")
    strXML.Append{"<optionText>" & _
                  "Select A Territory" & _
                  "</optionText>")
    strXML.Append("<optionValue>-l" & _
                  "</optionValue>")
    strXML.Append("</entry>")
// © Циклически пройти по набору результатов и добавить элементы XML

    Dim row As DataRow
    For Each row In dtOptions.Rows
      strXML.Append("<entry>")
      strXML.Append("<optionText:>" & __
            row("TerritoryDescription") & _
            "</optionText>")
      strXML.Append("<optionValue>" & _
                    row("TerritoryID") & _
                    "</optionValue>")
      strXML. Append. ("</entry>")
    Next
  End If
// © Вернуть документ XML
  strXML.Append("</selectChoice>")
  Response.Write(strXML.ToString)
End Sub
Public Function FillDataTable( _
                 ByVal sqlQuery As String) __
                 As DataTable
  Dim strConn As String = _
        "Initial Catalog - Northwind; " fi _
        "Data Source=127.0.0.1; " & _
        "Integrated Security=true;"
  Dim cmdl As _
  New SqlClient.SqlDataAdapter{sqlQuery, _
                               strConn)
  Dim dataSetl As New DataSet
  cmdl.Fill(dataSetl)
  cmdl.Dispose()
  Return dataSetl.Tables(O)
End Function                                                    _ _ •
```

Установив тип содержимого страницы O равным text/xml, мы гарантируем, что XMLHttpRequest правильно разберет отклик сервера на стороне клиента.

Из параметров запроса ©, полученных от клиента, мы получаем значение выбранной области, имя формы HTML и имя элемента. Чтобы лишний раз перестраховаться, мы можем добавить в этом месте проверку, гарантирующую, что данные значения не пустые. Если в ходе проверки не будет найдено хотя бы одно требуемое значение, сценарий может возвращать сообщение об ошибке. Кроме того, прежде чем приложение войдет в производственную

среду, мы также должны проверить возможность атаки через внедрение SQe кода. Это защитит базу данных от злоумышленных запросов.

Получив значение выбранной области, необходимо сгенерировать строку SQL, чтобы мы могли извлечь из базы данных соответствующие территории ©. Нас интересуют два столбца: TerritoryDescription и TerritorylD из таблицы Territories базы данных. Значение области мы вводим в условие WHERE выражения SQL. Чтобы гарантировать, что в создаваемом списке результаты будут идти в алфавитном порядке, мы дополнительно устанавливаем, условие ORDER BY равным TerritoryDescription. Далее следует выполнить выражение SQL O. В данном случае мы вызываем функцию FillDataTable() создающую соединение с сервером базы данных, выполняющую запрос и возвращающую результаты в таблицу данных.

Итак, мы получили результаты запроса SQL, теперь требуется создать первую часть документа XML ©, который обсуждался в листинге 9.2. Мы начинаем документ и добавляем объект selectElement, содержащий значения параметров formName и formElem, полученные из параметров запроса.

В этом месте нужна проверка: вернул ли запрос SQL какие-либо результаты ©. Если результаты есть, мы добавляем к XML-документу вводную опцию "Select A Territory" ©.

Далее мы циклически проходим по результатам, представленным в DataTable ©, заполняя значением столбца TerritoryDescription дескриптор optionText. а значением столбца TerritorylD — дескриптор optionValue. Вложив пары "описание-идентификатор" в дескриптор позиции, мы получаем легкий способ циклического прохода значений на стороне клиента с помощью методов DOM XML (реализованных на JavaScript). Заполнив полученными результатами документ XML, нужно закрыть корневой элемент selectChoice и записать ответ на выходной странице 0. Клиенту возвращается ответный документ XML, а объекту ContentLoader сообщается, что обработка на стороне сервера завершена. Далее объект ContentLoader вызывает на стороне клиента функцию FillDropDown(), которая обработает созданный нами XML-документ.

Подытожим, что мы сделали на стороне сервера. Мы взяли значение из выбранного элемента списка и организовали запрос к базе данных, не отправляя на дообработку на сервере всю страницу. Затем мы сгенерировали документ XML и вернули его клиенту. Следующий этап обработки снова будет происходить на стороне клиента, где мы должны преобразовать элементы XML в опции второго списка.

9.4. *Представление результатов*

Итак, теперь в документе XML у нас есть результаты запроса базы данных, и мы собираемся пройти по его элементам, используя программный интерфейс приложения DOM JavaScript. С помощью функции getElementsByTag^' Name {) мы можем легко "перепрыгивать" на любой элемент документа. Эта Функция с помощью имени элемента находит его в DOM (немного похоже иа то, как выдвигались алфавитные закладки в старомодном Rolodex). По-

дольку многие элементы в документе XML могут иметь одинаковые име-
а функция getElementsByTagName () в действительности возвращает массив
цементов, выстроенных по порядку их появления в документе.

д.4.1. Навигация в документе XML

Наконец-то мы можем завершить клиентскую часть сценария, добавляющую опции во второй список. Имена формы и элемента select, которые мы должны заполнить, задаются в документе XML вместе со всеми доступными опциями списка. Чтобы выделить эти опции и вставить их в элемент select, нам необходимо пройти по всем элементам документа.

Как только объект ContentLoader получит с сервера документ XML, он вызывает функцию FillDropDown(), приведенную в листинге 9.2. В функции FillDropDown() мы проходим по элементам entry документа XML и создаем для каждого из них новый объект Option. Данные объекты представляют пары "текст-значение"', которые будут добавлены во второй список. В листинге 9.5 функция FillDropDown() показана полностью.

Листинг 9.5. Обновление страницы с использованием данных из XML-ответа

```
//О Получить ответный XML-документ
function FillDropDown(){
// © Получить имя формы и элемента select
  var xmlDoc = this.req.responseXML.documentElement;
  var xSel = xmlDoc.
      getElementsByTagName('selectElement')[01;
  var strFName = xSel.
      childNodes[0],firstChild.nodeValue;
  var strEName = xSel.
      childNodes[1].firstChild.nodeValue;
// © Получить ссылку на элемент select
  var objDDL = document.forms[strFName].
      elements[strEName];
  objDDL.options.length = 0;
// О Последовательно пройти по XML-документу, добавляя опции
  var xRows = xmlDoc.
    getElementsByTagName('entry'     );
  for{i=0;i<xRows.length;i++){
    var theText = xRows[i].
        childNodes[0].firstChild.nodeValue;
    var theValue = xRowsfi].
        childNodes[l].firstChild.nodeValue;
    var option - new Option(theText, theValue);
    try{ objDDL.add(option,null);
    }catch (e){
      objDDL.add{option,-I);
    }
  }
```

Функция FillDropDown() вызывается объектом ContentLoader. как толь. ко он получит и разберет XML-ответ. Объект ContentLoader вызывается в функции FillDropDown() посредством ссылки this, и с его помощью у», получаем ответный документ responseXML Как только у нас будет ссылка на объект documentElement отклика О, мы можем использовать функции DOM JavaScript для навигации по его узлам. Первая информация, которую нам нужно получить, — целевой список select, к которому мы добавим новые опции. Мы ищем элемент, именуемый selectElement, используя функцию getElementsByTagName () и выбирая первый элемент из возвращенного этой функцией массива. После этого мы можем переходить к дочерним узлам @. Первый дочерний узел содержит имя формы, а второй — имя списка select.

Используя два указанных значения, мы обращаемся собственно к самому целевому списку © и очищаем все существующие опции, устанавливая длину массива его опций равной 0. Теперь мы можем добавлять новые опции к списку. Нам требуется доступ к элементам entry документа XML, поэтому мы еще раз вызываем функцию getElementsByTagName(). На этот раз требуется циклически пройти по массиву элементов, который возвращает эта функция, и получить пары "текст-значение" из каждого элемента О. Первый узел-потомок каждого элемента entry представляет собой текст опции, который будет показан пользователю, второй — значение. Получив два этих значения, мы создаем новый объект Option, передающий текст опции как первый параметр конструктора и значение опции — как второй. Затем к целевому элементу target добавляется новая опция, и процесс повторяется, пока не будут добавлены все новые опции. Сигнатура метода для select.add() зависит от браузера, поэтому мы используем оператор try .. . catch, пока не получим искомые сведения. Это все, что нужно-было сделать для завершения нашей структуры связанных списков. Теперь можно загружать HTML-страницу, выбирать регион и наблюдать, как второй раскрывающийся список заполняется прямо из базы данных. На рис. 9.7 показано, как это выглядит. В данном примере из первого списка выбран регион Eastern, затем из базы данных извлекаются и отображаются во втором списке соответствующие территории. Далее из первого списка выбирается регион Southern, после чего второй список заполняется соответствующими территориями.

Как видно на рис. 9.7, нам осталось сделать еще одно: изменить внешний вид списка, сделав его более привлекательным. Размер второго списка увеличивается при заполнении опциями. Данное смещение размера можно исправить, применив к элементу правило CSS (Cascading Style Sheet — каскадная таблица стилей).

9.4.2. Применение каскадных таблиц стилей

Каскадные таблицы стилей позволяют менять визуальные свойства элемента. Мы можем изменить цвет и семейство шрифта, ширину элемента и т.д. На рис. 9.7 видно, что второй элемент select изначально имеет ширину всего несколько пикселей, поскольку не содержит ни одной опции. При выборе из первого списка региона Eastern второй элемент select расширяется Это

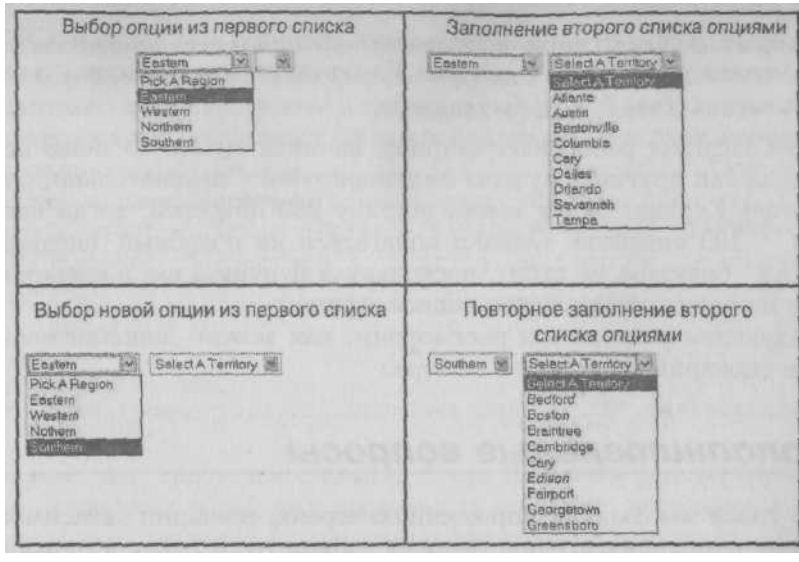


Рис. 9,7. Зависимые списки в действии

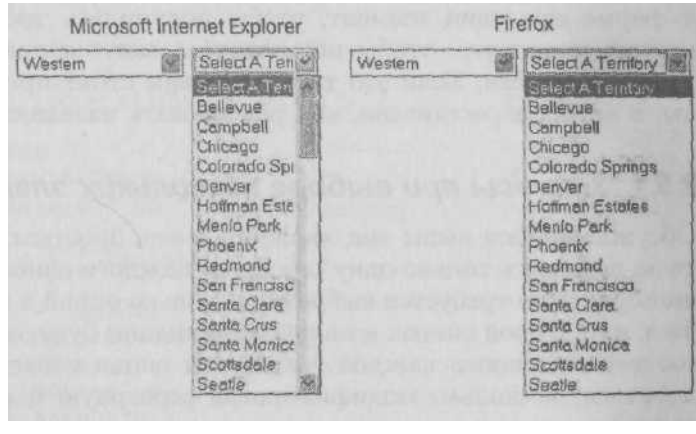


Рис. 9.8. В различных браузерах элемент s e l e c t визуализируется по-разному

изменение размера режет глаза и может раздражать пользователя. Чтобы решить эту проблему, можно задать ширину списка.

<select name="ddlTerritory" style="width:200px"x/select>

Однако так мы не снимем всех проблем, если, например, одно из отображенных значений больше установленной нами ширины. В Firefox, если элемент находится в фокусе, опции раскрывающегося списка показываются так, чтобы был виден весь их текст. Однако в Microsoft Internet Explorer текст усекается и не виден пользователю, как показано на рис 9.8.

Чтобы избежать проблем, связанных с Internet Explorer, необходимо установить ширину списка равной ширине наибольшей опции. В большинстве случаев единственной возможностью определения требуемого числа пикселей является метод проб и ошибок

Некоторые разработчики используют специальные уловки CSS только для того, чтобы установить в Internet Explorer большую ширину элемента: 1

```
style="width:100px;_width:250px"
```

Internet Explorer распознает ширину, начинающуюся со знака подчеркивания, тогда как другие браузеры ее игнорируют. Следовательно, рамки выбора Internet Explorer будут иметь ширину 250 пикселей, тогда как другие браузеры — 100 пикселей. Однако полагаться на подобные "специфические особенности" браузера не стоит, поскольку в будущем их, возможно, исправят, а это нарушит отображение ваших страниц.

В следующем разделе мы рассмотрим, как можно дополнительно улучшить наш сценарий зависимого выбора.

9.5. Дополнительные вопросы

В данной главе мы создали упрощенную версию сценария зависимого выбора. Серверу отправлялся единственный параметр, в ответ возвращался набор результатов для одного выбранного пункта. Возможно, вам кажется, что приложение должно работать иначе. Допустим, было бы неплохо добавить к форме еще один элемент, чтобы получилась тройная зависимость. Возможно, вы хотите, чтобы пользователь мог выбирать несколько элементов в первом списке. Если это так, тогда вам стоит прочесть следующие разделы, в которых рассказано, как реализовать названные идеи.

9.5.1. Запросы при выборе нескольких элементов

Обсуждавшийся выше код является очень простым,-он позволяет пользователю выбирать только одну опцию из каждого списка. В некоторых случаях пользователю требуется выбрать несколько опций в первом списке. Это означает, что второй список в нашей комбинации будет заполняться значениями, соответствующими каждой выбранной опции первого списка. Этого можно добиться, несколько модифицировав серверную и клиентскую часть нашего приложения.

Первое, что требуется сделать, — это настроить первый список так, чтобы разрешить выбор нескольких позиций. Для этого к дескриптору select нужно добавить атрибут multiple. Чтобы задать количество отображаемых опций, можно добавить атрибут size. Если значение size меньше числа опций, список будет прокручиваемым (и можно будет выбирать элементы, которые не видны в текущий момент).

```
<select name="ddlRegion" multiple size="4"
  onchange="FillTerritory(this,document.Forml.ddlTerritory)">
  <option value="l">Eastern</option>
  <option value="2">We3tern</option>
  <option value="3">Northern</option>
  <option value="4">Southern</option>
</select>
```

Далее необходимо изменить функцию FillTerritory<). Вместо того чтобы **просто** обращаться к выбранному номеру элемента select, нам требуется последовательно пройти по всем опциям и найти все выбранные значения. Затем значения всех выбранных опций добавляются в строку параметров.

```
function FillTerritory(oElem,oTarget){
  var url = 'DoubleComboMultiple.aspx';
  var strParams — "f=" + oTarget.form.name + "&e=" + oTarget.name;
  for(var i=0;KoElem.options.length;i++)[
    if(oElem.options[i].selected){
      strParams += "&q=" + oElem.options[i].value;
    }
  }
  var loaderl = new
  net.ContentLoader(url,FillDropDown,null,"POST",strParams);
}
```

Последнее, что требуется сделать, — это изменить код серверной части сценария, реализовав обработку нескольких значений, переданных в запрос. В .NET несколько значений, разделенных запятыми, представляются в одной строке. Чтобы получить отдельные элементы, необходимо расщепить строку на массив. Затем с помощью циклического прохода массива задается условие WHERE для оператора SQL.

```
Dim strQuery As String - Request.Form("q")
Dim strWhere As String = ""
Dim arrayStrt) As String = strQuery.Split(",")
Dim i As Integer
For Each i In arrayStr
  If strWhere.Length > 0 Then
    strWhere - strWhere & " OR "
  End If
  strWhere = strWhere & " regionid = " & i
Next
Dim strSql As String = "SELECT " & _
         " TerritoryDescription, " & _
         " TerritorylD" & __
         " FROM Territories" & _
         " WHERE " & strWhere & _
         " ORDER BY TerritoryDescription"
```

После таких изменений пользователь сможет выбирать несколько регионов из первого списка выбора, после чего во втором списке появятся территории, соответствующие каждому выбранному региону.

9.5.2. Переход от двойного связного выбора к тройному

Переход от двойного связного списка к тройному требует минимальных изменений (в зависимости от того, насколько мы хотим изменить логику сервера). Первый вариант заключается в том, чтобы перенести нашу логику на несколько серверных страниц и на каждой из них запустить свой запрос. Такое решение означает добавление к каждому обработчику onchange нужного списка еще одного параметра, представляющего URL вызываемого серверного сценария.

Возможен и другой вариант: добавить к серверному коду оператор £jfl else или switch-case. Чтобы использовать структуру if-else, нужно опр^ делить, какой запрос следует выполнять для возврата соответствующих зн&. чений. Простейшая проверка: решить, какой запрос SQL использовать, ос. новываясь на имени заполняемого элемента select. Таким образом, пр> реализации тройной комбинации мы можем проверять значение переменной strElem. При этом нам не потребуется изменять обработчики событий onchange в клиентской части кода.

```
Dim strSql As String
If strElem - "ddlTerritory" Then
    strSql = "SELECT TerritoryDescription, " $ _
           " TerritorylD" & __
           " FROM Territories" & _
           • WHERE " & strWhere & _
           " ORDER BY TerritoryDescription"
Else
    strSql = "SELECT Column1, Column2" &
           " FROM TableName" & __
           " WHERE " & strWhere &
           " ORDER BY Column2"
End If
```

При таком решении (и при условии, что раскрывающиеся списки заполнены уникальными именами) на странице можно будет использовать множественные связанные списки, не разделяя логику на различные серверные страницы.

9.6. *Реструктуризация*

Как вы думаете, чего не хватает в изложенном материале? Обобщения! Мы описали отличную модную технику реализации двойных списков, которая, однако, требует небольшой доводки, чтобы ее можно было обобщить. Мы перейдем к этому, так что держитесь! Но вначале давайте рассмотрим кое-что еще более фундаментальное: инкапсуляцию некоторых элементов внутренней "кухни" Ajax. Для начала можно взять объект net.ContentLoader, кратко представленный в главе 3 и более подробно рассмотренный в главе 5. Давайте создадим этот объект, чтобы еще лучше понять Ajax. В идеальном случае данный элемент должен стать нашим объектом-'тгомощником" Ajax, который формирует всю инфраструктуру Ajax. Это позволит сосредоточиться на разработке вопросов, связанных только с зависимыми списками, а также сократить код, требуемый всеми остальными компонентами. Наш улучшенный объект net.ContentLoader в идеальном случае должен инкапсулировать состояние и поведение, требуемые для выполнения перечисленных ниже задач.

- Создание объекта XMLHttpRequest, который был бы понятен всем браузерам и не зависел от отправки запросов. Это позволит вызывающей стороне использовать метод создания независимо от остальной части объекта. Данная возможность полезна, если вызывающая сторона использует другую идиому, каркас или механизм для действия запроса/отклика.

- Предложить более удобный API для обработки параметров запроса. В идеальном случае вызывающая сторона должна просто передавать от приложения состояние и разрешать "помощнику" net. ContentLoader решать все вопросы, связанные с созданием строк запроса.
- Маршрутизация отклика обратно к компоненту, который знает, как его обработать, и выполнение соответствующей обработки ошибок.

Ну что ж, начнем сборку объекта net .ContentLoader, а затем переформулируем наш сценарий двойного списка как компонент.

9.6.1. Новый и улучшенный объект netContentLoader

Рассмотрим вначале, как следует изменить конструктор.

```
net.ContentLoader = function! component, url,
method,   requestParams ) {
// Состояние net.ContentLoader
  this.component = component;
  this.url = url;
  this.requestParams = requestParams;
  this.method = method;
}
```

Данный конструктор вызывается с четырьмя аргументами. Первый, component, обозначает объект, пользующийся услугами данного "помощника". Вспомогательный объект будет предполагать, что component имеет метод aj axUpdate () для обработки откликов и метод handleError () для обработки состояний ошибки. Подробнее этот вопрос мы рассмотрим позже. Второй аргумент, url, обозначает URL, вызываемый данным "помощником" для асинхронного получения данных с сервера. Параметр method обозначает метод HTTP-запроса. Допускаются значения GET и POST. Наконец, аргумент requestParameters представляет собой массив строк вида key=value, которые обозначают параметры запроса, передаваемые запросу. Это позволяет вызывающей стороне задавать набор параметров запроса, которые не меняются между запросами. Эти параметры будут присоединены к дополнительным параметрам запроса, передаваемым в рассмотренный ниже метод sendRequest Таким образом, наш объект-'помощник" может следующим образом создаваться клиентом:

```
var str = "Eastern";
var aComp - new SomeCoolComponent(...);
var ajaxHelper = new net.ContentLoader( aComp,
                 "getRefreshData.aspx", "POST",
                 [ "query=" + str, "ignore_case=true" ]
);
```

Рассмотрим теперь оставшуюся часть API. Здесь следует обратить в: ше внимание на стилистическую природу примера кода. Область действ» методов этого объекта распространяется на объект-прототип, прикрепле; ный к функции конструктора. Данная технология является стандартной *n* написании объектно-ориентированного кода JavaScript, поскольку примен

ет определения метода ко всем экземплярам объекта. Тем не менее сущ^. ствует несколько способов, позволяющих задать такое поведение синтаксически. Одним из наших любимых (его структура позаимствована из библиотеки prototype.js, внедренной в Ruby On Rails) является буквальное создание объекта-прототипа.

```
net.ContentLoader.prototype = {
// Первый метод, прикрепленный к прототипу
  methodl: function(a, b, c) {
```

Ь

```
// Второй метод
  method2: function() { },
  method3: function(a) { }
};
```

С точки зрения синтаксиса достоинством данного подхода является лаконичность. Как читать данный фрагмент? Внешние скобки представляют литерал объекта, а содержимое — это разделенный запятыми список пар "свойство-значение" в объекте. В данном случае наши свойства являются методами. Пары "свойство-значение" задаются как имя свойства, затем следует двоеточие, а после него ~ значение свойства. В этом случае значения (или определения, если вам угодно) являются литералами функций. Все просто, правда? Достаточно помнить, что методы, которые будут использоваться начиная с этого момента, предполагаются содержащимися внутри литеэала объекта-прототипа (как показано выше). Кроме того, обратите внимание на то, что последнее свойство не требует (а следовательно, может не №геть) после себя запятой. Теперь давайте вернемся к рассматриваемой задаче: сборке API.

API должен удовлетворять упомянутым выше требованиям, поэтому рас:мотрим их последовательно. Первое, что нам требуется, — обработка создания объекта XMLHttpRequest независимо от браузера. Звучит, как метод! С счастью, мы уже реализовывали его несколько раз. Все, что требуется сейгас, — создать его как метод нашего "помощника" (листинг 9.6), чтобы нам ольше никогда не приходилось его писать.

Листинг 9.6. Метод getTransport

```
getTransport: function() {
  var transport;
// Родной объект
  if ( window.XMLHttpRequest )
    transport = new XMLHttpRequest();
// Объект IE ActiveX
  else if ( window.ActiveXObject ) {
    try { transport = new ActiveXObject('Msxml2.XMLHTTP'); }
    catch(err) {
      transport = new ActiveXObject('Microsoft.XMLHTTP');
    }
  }
  return transport;
},
```

Данный код не требует очень подробных объяснений, поскольку данную тему мы рассматривали неоднократно, однако на этот раз мы привели аккуратно упакованный метод, обеспечивающий работающий во всех браузерах объект Ajax транспорта данных для обработки нашей асинхронной связи. Второе упомянутое нами требование касалось обеспечения более удобного API для обработки параметров запроса. Чтобы его можно было использовать во множестве приложений, очевидно, что отправляемый запрос будет требовать в качестве параметров значения времени выполнения. Мы уже записали некоторое исходное состояние, представляющее параметры запроса, постоянные для различных запросов, но кроме этого нам требуются значения времени выполнения. Рассмотрим поддержку приведенного ниже кода.

```
//Предполагается инициализация со значениями времени
выполнения
var a,b,c;
var ajaxHelper = new net.ContentLoader(... );
ajaxHelper.sendRequest( "paraml=" + a, "param2=" + b,
                        "param3=" + c );
```

Итак, при данном требовании к использованию определяется sendRequest, как показано в листинге 9.7.

```
sendRequest: function() {
//О Записываем аргументы в массив
  var requestParams = [];
  for ( var i = 0 ; i < arguments.length ; i++ ) {
    requestParams.push(arguments[i]);
  }
// © Создаем запрос
  var request = this.getTransport();
  request.open( this.method, this.url, true );
  request.setRequestHeader( 'Content-Type',
                            'application/x-www-form-urlencoded');
// © Задается обратный вызов
  var oThis = this;
  request.onreadystatechange = function() {
                          oThis.handleAjaxResponse(request) };
// О Отправляется запрос
  request.send( this.queryString(requestParams) );
```

Данный метод расщепляет процесс отправки запроса на четыре этапа. Рассмотрим эти этапы подробнее.

- Данный этап использует тот факт, что JavaScript создает псевдомассив arguments, область действия которого распространяется на функцию. Как можно догадаться по имени, агдитег^эсодержит аргументы, переданные функции. В данном случае ожидается, что аргументы - это строки вида key=value. Пока что мы просто копируем их в массив первого класса. Кроме того, обратите внимание, что все переменные, созданные в этом методе, начинаются с ключевого слова var. Хотя компилятору

JavaScript сильно бы понравилось, если бы мы оставили ключевое слово varB покое, то, что мы так не поступили, очень важно Почему? Если мы опустим это ключевое слово, переменная создается с глобальной областью действия — т.е. видимой для всего кода в вашей вселенной JavaScript! Это может вызвать нежелательное взаимодействие с остальным кодом {например, в использованном вами чьем-либо сценарии переменная названа точно так же, как и у вас). Короче говоря, подобная ситуация гарантирует кошмар отладки. Позаботьтесь о себе сами — приучитесь использовать переменные с локальной областью действия везде, где это возможно.

© Здесь наш метод использует метод getTransport, определенный в листинге 9.6, для создания экземпляра объекта XMLHttpRequest. Затем обрабатывается запрос, и его заголовок Content-Type инициализируется, как в предыдущих примерах. Ссылка на объект хранится в локальной переменной request.

© На данном этапе решается задача обработки отклика. Вы наверняка пытаетесь догадаться, зачем была создана переменная oThis. Обратите внимание на следующую строку, где фигурирует анонимная функция, реагирующая на изменение статуса готовности (onreadystatechange) нашего объекта запроса и ссылающаяся на oThis. Данный прием называется замыканием. В силу того что внутренняя функция обращается к локальной переменной, создается неявный контекст выполнения (или область действия), позволяющий поддерживать ссылку после выхода из замыкающей функции. (Подробнее о замыкании речь пойдет в приложении Б.) Это позволяет реализовать обработку отклика Ajax посредством вызова метода первого класса на нашем объекте ajaxHelper.

О Наконец мы отправляем запрос Ajax. Обратите внимание на то, что массив, созданный нами на этапе 1, передается методу queryString, который преобразовывает его в одну строку. Эта строка становится телом запроса Ajax. В действительности метод queryStringHe является частью открытого контракта, который мы обсуждали ранее, но относится к вспомогательным методам, облегчающим чтение и понимание кода. Рассмотрим его подробнее *в* листинге 9.8.

```
queryStrxng: function(args) {
// Постоянные параметры
  var requestParams = [];
  for ( var i = 0 ; i < this.requestParams.length ; i++ ) {
    requestParams.push(this.requestParams[i]);
  }
// Параметры времени выполнения
  for ( var j = 0 ; j < args.length ; j++ ) {
    requestParams.push(args[j]);
  }
  var queryString - "";
  if ( requestParams SS requestParams.length > 0 ) {
    for ( var i = 0 ; i < requestParams.length ; i++ ) {
```

```
        queryString += requestParams[i] + •&';
      }
      queryString • queryString.substring{0, queryString.length-1);
    }
    return queryString;
```

Данный метод принимает параметры запроса, с которыми был создан объект net.ContentLoader, а также дополнительные параметры времени выполнения, переданные в метод sendRequest и помещенные им в общий массив. Затем массив обрабатывается и превращается в строку запроса. Результат описанных действий представлен ниже.

```
var helper = new net.ContentLoader{ someObj, someUrl, "POST",
  ["a-one", -b-two"]
  );
var str = ajaxHelper.queryString(
  ["c-three", "d-four"]
  );
str -> "a*=one&b=two&c=three&d=four"
```

Последнее, что нам требуется сделать для получения полнофункционального вспомогательного объекта, — это реализовать сотрудничество с компонентом с целью обработки отклика, поступающего от Ajax. Если вы были внимательны, то, возможно, уже знаете имя данного метода. Действительно, наш метод sendRequest уже задал, как он будет обрабатывать отклик свойства onreadystatechange запроса.

```
request.onreadystatechange • function(){
  oThis.handleAjaxResponse(request)
}
```

Все правильно, осталось всего лишь реализовать метод handleAjaxResponse. Реализация этого метода приведена в листинге 9.9.

```
handleAjaxResponse: function(request) {
  if ( request.readyState == net.READi_STATE_COMPLETE ) {
    if ( this.isSuccess(request) )
// Компонент сообщения с откликом
      this.component.ajaxUpdate(request);
    else
// Компонент сообщения с ошибкой
      this.component.handleError(request);
  }
},
isSuccess: function(request){
  return request.status =e 0
    II (request.status >= 200 && request.status < 300);
```

Все, что делает метод, — проверяет, равно ли значение состояния readystate четырем (указывает на завершение), и уведомляет this, component, что отклик доступен. Но мы еще не закончили. Существовало еще требование соответствующей обработки ошибок. Но что такое "соответствующая"? Этого мы сказать не можем. Обработка ошибок — это решение, которое нужно отложить для другого объекта. Следовательно, мы предполагаем, что наш клиент this, component, имеет метод handleError, который и выполняет необходимые действия, когда отклик Ajax поступает как-то не так. Этот компонент может в свою очередь делегировать решение другому объекту, но это уже не проблемы нашего вспомогательного объекта. Мы обеспечили механизм и *позволим другому объекту* обеспечить семантику. Как говорилось ранее мы предполагаем, что this . component имеет методы ajaxupdate и handleError. Это неявный созданный нами контракт, поскольку JavaScript не относится к языкам со строгим контролем типов, которые могут принудительно вводить подобные ограничения.

Поздравляем! Вы трансформировали net .ContentLoader Б гибкий вспомогательный объект, который выполняет всю грязную работу Ajax для ваших DHTML-компонентов, лоддерживающих Ajax. А если у вас есть DHTML-компонент, который еще не поддерживает Ajax, теперь он станет проще! Кстати, нам еще нужно написать компонент двойных списков.

9.6.2. Создание *компонента двойного списка*

Мы немного поработали с объектом net.ContentLoader, чтобы существенно упростить нашу задачу, к *которой* теперь пора вернуться. Предположим, что нам, первоклассным разработчикам, поручили создать сценарий Связанного выбора, который можно использовать повторно во многнх^онтекстах в приложении или нескольких приложениях. В связи с этим для удовлетворения поставленным требованиям необходимо рассмотреть несколько вопросов.

- Предположим, мы не можем (или не хотим) вручную менять HTML-разметку окон выбора. Так может быть, например, если мы не отвечаем за производство разметки. Возможно, элемент select генерируется JSP или другим дескриптором на языке сервера. Возможно, какой-то дизайнер написал HTML-код, а мы желаем по возможности сохранить его без изменений, чтобы не допустить дополнительного этапа переверстки страницы.
- Нам требуется сценарий связывания, который может использовать различные указатели URL и параметры запроса для возврата данных option. Кроме *того,* требуется, чтобы структура допускала будущую настройку.
- Необходимо, чтобы поведение, реализованное в сценарии зависимого выбора, можно было *потенциально использовать в* нескольких наборах дескрипторов select на одной странице, а также, чтобы существовала возможность реализации тройных или четверных комбинаций, как обсуждалось ранее.

Начав с первой задачи (максимально сохранить HTML-разметку), предположим, что характерным представителем HTML-кода, с которым нам придется работать, является разметка, приведенная в листинге 9.10.

```
<html>
<body>
<form name="Forml">
  <select id="region" name="region" >
    <options... >
  </select>
  <select id="territory" name="territory" />
</form>
</body>
</html>
```

Нам нужен компонент *DoubleCombo,* который можно присоединить к документу, возложив на него всю магию двойной комбинации. Поэтому пойдем с конца и рассмотрим, как бы нам хотелось, чтобы выглядела разметка, а затем сообразим, как ее реализовать. Изменим разметку так, чтобы она выглядела подобно коду, приведенному в листинге 9.11.

Листинг 9.11. Модифицированная HTML-разметка . '[5!a]ТЯИЯНИИТТ

```
<html>
<head>

  <script>
    function inj ectComponentBehaviors() {
      var doubleComboOptions = {};
<!— Компонент DoubleCombo —>
 new DoubleCombo('region', 'territory',
[1]DoubleComboXML.aspx', doubleComboOptions );
    }
  </script>
</head>
<body onload=
 •injectComponentBehaviors()"
>
<form name="Forml">
  <select id="region" narae="region" >
    <option value="-l">Pic]c A Region</option>
    <option value="1">Eastern</option>
    <option value="2">Western</option>
    <option value="3">Northern</option>
    <option value="4">Southern</option>
  </select>
  <select id="territory" name="territory" />
</form>
</body>
</html>
```

д

Разметка изменилась следующим образом

- Создана функция, инжектирующая в документ требуемое нам поведение
- К элементу тела, вызывающему данную функцию, добавлен обработчик onload.

Обратите внимание на то, что в элементе <body> страницы никаких модификаций нет. Как отмечалось ранее — это хорошо. Таким образом, мы уже удовлетворили первому требованию. Но давайте посмотрим на функцию injectComponentBehaviors() — похоже, что есть еще одна работа. Действительно, нам нужно создать объект JavaScript, именуемый DoubleCombo в котором было бы реализовано все поведение, необходимое для поддержки функциональных возможностей двойной комбинации.

Логика компонента DoubleCombo

Для начала рассмотрим внимательнее семантику создания нашего компонента. Функция injectComponentBehaviors() создает объект DoubleCombo, вызывая его конструктор. Этот конструктор определен в листинге 9.12.

. . Листйнг-9,12. Компонент DoubieComboDoubleCombo

```
function DoubleCombo( masterld, slaveld, url,
options ) {
// Инициализация состояния
  this.master - document.getElementByld(masterld);
  this.slave = document.getElementById(slaveld);
  this.options = options;
  this.ajaxHelper " new net.ContentLoader( this, url, "POST",
                          options.requestParameters I I, И );
// Инициализация поведения                          - —
  this.initializeBehavior();
}
```

Данная конструкция должна быть вам уже знакомой; наша функция конструктора инициализирует состояние объекта DoubleCombo. Описания аргументов, которые необходимо передать конструктору, приведены в табл. 9.2.

Рассмотрим природу состояния, поддерживаемого объектом DoubleCombo, — особенно URL и опции. Две данные составляющие состояния удовлетворяют второму указанному ранее функциональному требованию, т.е. наш компонент может задействовать для извлечения данных любой URL, кроме того, этот компонент можно настраивать с помощью параметра options. Пока что мы предполагаем, что в объекте опций найдем только одно — свойство requestParameters. Однако, поскольку параметр options — это просто общий объект, мы можем приписать ему любое свойство, необходимое для облегчения последующей настройки. Наиболее очевидными "довесками" объекта опций являются стилевое оформление классов CSS и прочее из этой же серии. Однако стиль и функции связных списков очевидно являются независимыми концепциями, поэтому решение вопросов стилевого оформления мы оставляем дизайнеру страницы.

Таблица 9.2. Описание аргументов

дргумент	Описание
masterId	Идентификатор элемента разметки, соответствующего основному элементу select. Выполнение операции выбора на данном элементе определяет значения, отображаемые вторым элементом select
slaveId	Идентификатор элемента разметки, соответствующею зависимому элементу select. Значения этого элемента будут меняться в зависимости от выбора пользователем значения основного элемента select
options	Общий объект, предоставляющий другие данные, требуемые сценарием

Мы уверены, что для многих из вас самой интересной частью конструктора являются последние две строчки:

```
this.ajaxHelper = new net.ContentLoader( this, url, "POST",
                                         options.requestParameters || []
>;
```

Разумеется, мы знаем, что наш компонент требует возможностей Ajax. Благодаря везению и толике планирования мы уже имеем объект, выполняющий большую часть всей работы, выпадающей на долю Ajax, — речь идет об объекте net.ContentLoader, который мы предусмотрительно написали ранее. Объект DoubleCombo просто передает себя (посредством this) как параметр компонента вспомогательному объекту ContentLoader. Кроме того, как целевой URL запросов Ajax этому вспомогательному объекту передается параметр url, и с помощью строки "POST" задается метод HTTP-запроса.

Наконец, свойство requestParameters объекта опций (или пустой массив, если не была определена ни одна опция) передается как массив "постоянных" параметров, которые будут отправляться со всеми запросами Ajax. Напомним также, что поскольку мы передали this как аргумент компонента, объект DoubleCombo обязан реализовать указанный ранее неявный контракт с объектом net.ContentLoader. Другими словами, мы должны реализовать методы ajaxUpdate() и handleError(). Чуть позже мы разберем этот момент подробно, а пока рассмотрим последнюю строку нашего конструктора:

```
this.initializeBehavior();
```

Наконец-то наш конструктор делает что-то, похожее на желаемое поведение. Да, наступил момент, которого мы так долго ждали: реализация поведения. Все, что мы будем делать далее, непосредственно связано с обеспечением функциональных возможностей зависимых списков. Поэтому без лишних слов рассмотрим данный метод (а также все остальные методы DoubleCombo, которые нам потребуются). Благодаря тому что мы упорядочили всю инфраструктуру, наша задача уже не выглядит устрашающе. Помните, что все методы, которые встретятся при разборе данного примера, предполагаются внедренными в литеральный объект-прототип (точно так же, как мы делали при реализации net.ContentLoader).

```
DoubleCombo.prototype = {
  // все методы ... .
};
```

Ну что ж, заглянем под капюшон. Ниже приведен метод initializeBehavior().

```
initializeBehavior: function() {
  var oThis = this;
  this.master.onchange = function() { oThis.masterComboChanged(); },

)/
```

Коротко и ясно. Данный метод помещает обработчик событий onchange в основной элемент select {ранее это делалось в самой разметке HTML). При срабатывании обработчик событий вызывает на нашем объекте masterComboChangedO другой метод.

```
masterComboChanged: function() {
  var query = this.master.options[
                this.master.selectedlndex].value;
  this.ajaxHelper.sendRequest( 'q=' + query );
Ь
```

Удивительно, но все по-прежнему коротко и ясно. Все, что требуется от этого метода, — это создать параметр запроса и отправить наш запрос Ajax. Поскольку вся работа Ajax была вынесена в другой объект, все действия формулируются с помощью одной строки кода. Напомним, что sendRequest {) создаст и отправит XMLHttpRequest, который направит отклик обратно нашему методу ajaxUpdate (). Запишем все сказанное.

```
ajaxUpdate: function(request) {
  var slaveOptions = this.createOptions (              ^/
                     request.responseXML.documentElement);
// Очистить существующие опции
  this.slave.length = 0;
// Заполнить новые опции
  for ( var i = 0 ; i < slaveOptions.length ; i++ )
    try{
      this.slave.add(slaveOptions[i], null);
    }catch (e){
      this.slave.add(slaveOptions[i],-1);
    }
```

Ь

Данный метод принимает ответный XML-документ от объекта request и передает его методу createOptions(), который создает элементы option нашего зависимого элемента select. Затем метод просто очищает и повторно заполняет зависимый элемент select. Метод createOptions(), хотя *я* не является частью никакого публичного контракта, относится к вспомогательным методам, которые облегчают чтение и понимание кода. Его реализация и еще один вспомогательный метод, getElementContent (), показаны в листинге 9.13.

Листинг 9.13. Методы заполнения

```
createOptions: function(ajaxRespon.se) {
  var newOptions = [];
  var entries = ajaxResponse.getElementsByTagName('entry');
  for ( var i — 0 ; i < entries.length ; i++ ) {
    var text = this.getElementContent(entries[i],
               'optionText');
    var value = this.getElementContent(entries[i],
                'optionValue');
    newOptions.push( new Option{ text, value ) );
  }
  return newOptions;
```

Ь

```
getElementContent: function(element,tagName) {
  var childEleraent = element.getElementsByTagName(tagName)[0];
  return {childElement.text != undefined) ? childElement.text :
                                            childElement.textContent;
```

Ь

Данные методы выполняют всю тяжелую работу, действительно извлекая значения из ответного документа XML и создавая по ним объекты опций. Напомним, что структура XML-отклика выглядит следующим образом:

```
<?xml version»'* 1.0" ?>
<selectChoice>

  <entry>
    <optionText>Select A Territory</optionText>
    <optionValue>-l</optionValue>
  </entry>
  <entry>
    <optionText>TerritoryDescription</optionText>
    <optionValue>TerritoryID</optionValue>
</entry>
</selectChoice>
```

Метод createOptions() последовательно проходит по всем элементам entry в XML-документе и извлекает текст из элементов optionText и optionValue посредством вспомогательного метода getElementContent(). Касательно метода getElementContent() стоит отметить только то, что он использует IE-совместимый атрибут text элемента XML, если он существует; в противном случае применяется стандартизованный W3C атрибут textContent.

Обработка ошибок

Ну вот и все. Вернее, почти все. Мы реализовали все линии поведения, необходимые для раскрытия потенциала данного компонента. Но постойте! Мы говорили, что обработка состояний ошибки также будет. Вы можете сказать, что нам требуется еще реализовать метод handleError(), чтобы с объектом net.ContentLoad.er можно было нормально работать. Ну так давайте реализуем его и действительно завершим эту задачу. Итак, какое восстанавливающее действие требуется при наличии ошибки? Пока что мы этого сказать не можем. Вообще-то, этот вопрос должно решать приложение, использующее

наш компонент DoubleComtoo. Похоже, что при такой формулировке нашлась работа нашему объекту опций (помните, мы передавали его конструктору?У Рассмотрим возможность такого контракта. Что будет, если мы создадим нащ компонент связных списков с кодом, подобным приведенному ниже?

```
function myApplicationErrorHandler(request) {
  // Функция приложения, отвечающая за обработку ошибок
}
var comboOptions = { requestParameters: [
                     "param1=one", "param2=two" ],
 errorHandler: myApplicationErrorHandler
 };
var doubleCombo • new DoubleCombo( 'region', 'territory',
                                   'DoubleComboXML.aspx',
                                    comboOptions );
```

В этом сценарии, мы позволяем приложению определить функцию myApplicationErrorHandler(). Именно в реализацию данного метода мы можем поместить логику приложения, обрабатывающую ошибочные ситуации, Это может быть предупреждение. Или менее навязчивое сообщение "упс!" в стиле GMail. Суть такого решения заключается в том, что мы делегируем принятие решения приложению, использующему наш компонент. Как и ранее, мы предоставляем механизм и позволяем кому-то еще обеспечить семантику. Следовательно, теперь нам нужно написать метод handleError() объекта DoubleCombo.

```
handleError: function(request) {
  if ( this.options.errorHandler )
    this.options.errorHandler(request);
}
```

Счастье компонента

Поздравляем! Наконец-то мы сделали все. У нас есть *общий* компонент, который мы можем создать с идентификаторами любых двух элементов select и некоторой информацией по конфигурации, также у нас есть возможность зависимого выбора. И тут... *распахивается дверь*[1].

Пятница, 14:45, входит менеджер, ничего не смыслящий в программировании. "Джонсон, — восклицает он. — Нам нужна поддержка подтерриторий! ... И она должна быть готова к утру понедельника!" Драматическая пауза. "О-ох!" — выдавливаете вы из себя. Затем вы собираетесь и говорите: "Я сделаю это, сэр. Даже если мне придется проработать все выходные". Он вручает вам новый дизайн страницы:

```
<form>
  <select id="region" name="region"><select>
  <select id="territory" name="territory"></select>
  <select id="subTerritory" name="subTerritory"X/select>
</form>
```

Начальство уходит. Вы открываете HTML-страницу в Emacs, поскольку так вам удобнее. Переходите сразу к заголовку. Курсор мигает — вы начинаете набирать.

```
<script>
  function injectComponentBehaviors() {
    var optsl *• { requestParameters: "master=region" };
    var opts2 = { requestParameters: "master=territory" J;
    new DoubleCombo( 'region[1],
                     'territory',
                     'DoubleComboXML.aspx', optsl );
    new DoubleCombo( 'territory',
                     [1]subTerritory',
                     'DoubleComboXML.aspx', opts2 );
</script>
```

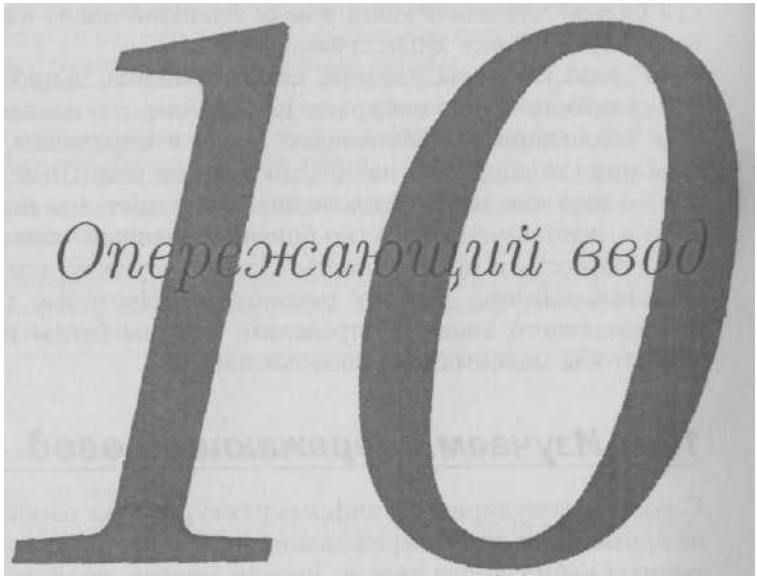
Вы нажимаете клавишу, запуская макрос, который отформатирует ваш код. Сохраняете. Бросаете через плечо: "Я буду работать из дома", проходя мимо офиса менеджера в 14:57. Дома вы плюхаетесь на диван и думаете про себя: "Да, я крут!" Ладно, хватит фантазий. Запомните, что вы сделали, и забудьте обо всем остальном.

9.7. Резюме

Двойная комбинация элементов select представляет собой эффективный метод создания для пользователя динамических элементов формы. Мы можем использовать обработчики событий JavaScript для отслеживания изменений в одном элементе select и инициации процесса обновления значений второго элемента. С помощью Ajax мы можем избежать длительного времени загрузки страницы, характерного для решений JavaScript. Используя Ajax, мы можем организовать запрос к базе данных, не отправляя всю страницу на сервер для дообработки и не разрушая взаимодействие пользователя с формой. Ajax позволяет создавать Web-приложения, более близкие к клиентским приложениям.

Используя приведенный код, вы сможете разработать более сложные формы, не утруждаясь обычными проблемами дообработки страниц на сервере. Благодаря возможностям расширения сценария на различные комбинации списков ваши пользователи смогут более точно проходить через несколько уровней опций, получая информацию или продукты, которые им требуются.

Наконец, мы немного переработали код, создав для себя промышленный компонент, облегчающий потенциальное повторное использование и настройку. По нашему мнению, мы инкапсулировали данные функциональные возможности в самодостаточный компонент, и нам никогда не придется писать его повторно. По мнению пользователей, исчезла вероятность получения сообщения "продукт недоступен" при покупках в интерактивных магазинах. Все счастливы.



В этой главе...

- Опережающий ввод
- Кэширование результатов поиска на стороне клиента
- Библиотека Rico
- Библиотека Prototype
- Функция JavaScript apply()
- Параметризация компонентов

Опережающий ввод является одной из основных сфер применения Ajax благодаря которым эта инфраструктура заняла свое место в мире программирования. Возможности Google Suggest впечатляют пользователей, обнаруживающих список предположительных вариантов того, что они набирают (и даже сейчас, спустя много месяцев, эта возможность все еще активно обсуждается). Действие этого приложения подобно работе помощника, сидящего рядом с вами и подсказывающего, что вам нужно. Одни люди думают что предлагаемые значения хранятся на странице, другие полагают, что для получения данных применяется скрытый элемент IPrame. В действительности Google для извлечения новых значений после нажатия каждой клавиши использует объект XMLHttpRequest.

В этой главе вы узнаете, как отправлять запросы Ajax в то время, как пользователь что-то набирает. Кроме того, мы исследуем недостатки доступных реализаций опережающего ввода и определим, как избежать их в нашем приложении. Для начала мы примем понятный низкоуровневый подход. После того как наше приложение заработает, мы перейдем на высокий уровень и используем объектно-ориентированный подход, предлагающий большую гибкость и практичность. Однако, прежде чем мы приступим к созданию приложения, давайте рассмотрим некоторые типичные составляющие опережающего ввода и определим, как мы будем разрабатывать приложение, чтобы максимально использовать их.

10.1. Изучаем опережающий ввод

С ростом популярности инфраструктуры Ajax одной из самых модных сфер ее применения стал опережающий ввод. Сценарии, заранее предлагающие варианты набираемого текста, писали многие люди, реализовывавшие различные варианты взаимодействия с сервером. Одни решения имели свои недостатки, другие подходы были чересчур агрессивными. Чтобы получить общее представление о данном вопросе, вначале оценим ряд функциональных возможностей, реализованных во многих приложениях упреждающих предположений, а затем кратко рассмотрим Google Suggest. После этого мы разработаем свое приложение.

10.1.1. Типичные элементы приложений опережающего ввода

В настоящее время существует множество приложений опережающего ввода — от простейших до достаточно сложных. Все они преследуют одну цель, просто в некоторых реализованы более модные интерфейсы с затухающими переходами. Введите в какой-либо поисковой программе "type-ahead suggest Ajax" — и вы получите приличное число результатов.

Изучив несколько представителей данного множества, можно заметить, что все они построены примерно по одной схеме.

Таблица 10.1. Проблемы приложений опережающего ввода

Проблема	Результат
Пообработка на сервере после каждого нажатия клавиши	Чрезмерное потребление полосы пропускания. Представьте, какая ширина полосы требуется для обработки действий 1 000 пользователей, набирающих 10-символьные слова
Данные не кэшируются	Запросы бомбардируют базу данных постоянно, даже если в ответ на запрос возвращаются пакеты данных, ранее уже передававшиеся клиенту
Модем 56K	Да, еще существуют пользователи, применяющие удаленный доступ по модему; для них передача информации с сервера будет сопровождаться заметными паузами
Возврат слишком большого числа результатов	Если не ограничить количество результатов, клиенту будет отправлено чересчур много данных (соответственно увеличится время отправки ответа)
Слишком модный интерфейс	Множество "наворотов" может мешать, увеличивая время визуализации
Быстрая печать	Некоторые люди способны печатать быстро. Определенные сценарии рассчитаны на людей, долго ищущих на клавиатуре нужную букву

1. Вы набираете символ.
2. Приложение отправляет запрос на сервер.
3. Данные возвращаются клиенту.
4. Клиент принимает данные и отображает результаты в таблице, элементе div, текстовом окне и т.п.

Тем не менее существуют моменты, в которых некоторые сценарии проявляют себя не лучшим образом. Разработчикам следует учитывать ширину полосы пропускания, пропускную способность сервера, а также конфигурацию клиента. Если вы забудете об этих факторах, использование Ajax может вызвать отрицательную реакцию пользователя (а нужна положительная).. Возможные проблемы, довольно распространенные в приложениях Ajax, перечислены в табл. 10.1.

Разработчики, как правило, забывают о полосе пропускания. Даже один пользователь, вводящий простое слово, может несколько раз сталкиваться с дообработкой на сервере. Объедините это с проблемой быстрой печати, и число запросов в секунду станет даже больше, чем в приложениях, не использующих Ajax.

Наличие оперативного интерфейса также очень важно. Чем больше времени уходит на визуализацию средств управления, тем позже пользователь их увидит и тем менее эффективным будет приложение. Кроме того, возможны задержки в работе, из-за завышенной нагрузки на сервер. Если запросы формируются слишком часто или возвращают чересчур много данных, пострадает оперативность пользовательского интерфейса.

Рис. 10.1. Google Suggest предлагает различные варианты после ввода буквы "e"

Для повышения оперативности интерфейса можно каптировать данные на стороне клиента. (Данные можно кэшировать и на сервере, но это другая тема, более знакомая разработчикам классических Web-приложений.) Система опережающего ввода обычно возвращает тем меньше результатов, чем больше набрано символов, причем, как правило, эти результаты входили в самый первый предложенный набор. Реализация системы "в лоб" отбрасывает предыдущие запросы в ответ на набор новых букв и извлекает с сервера новые данные. Более интеллектуальная система может оставить результаты первоначального запроса, отсеивая неподходящие элементы по мере ввода пользователем символов и обновляя пользовательский интерфейс без обращения к серверу за обработкой каждого нового символа. Такое решение улучшает приложение путем повышения оперативности и сокращения требуемой полосы пропускания. Мы просто обрабатываем имеющийся набор результатов, что ускоряет извлечение нужной информации и снимает необходимость в дополнительных этапах обработки на сервере.

Пожалуй, хватит теории. В следующем разделе мы рассмотрим готовую реализацию опережающего ввода.

10.1.2. Google Suggest

Некоторые люди считают Google Suggest вершиной приложений опережающего ввода. Google Suggest — это быстрое, доброжелательное к пользователю приложение, эффективно выполняющее свою работу. На рис. 10.1 показано, как Google Suggest предлагает варианты после ввода буквы "e".

На рис. 10.1 число предложений ограничено десятью. Зная, какой базой данных располагает Google, можно было бы ожидать и миллиарды предложений. При отборе информации, отображаемой пользователю, Google использует специальный алгоритм. Чтобы понять, как работает написанный Google код JavaScript, разработчики внимательно его разобрали (помните, что код JavaScript невозможно скрыть полностью, хотя можно запутать его анализ).

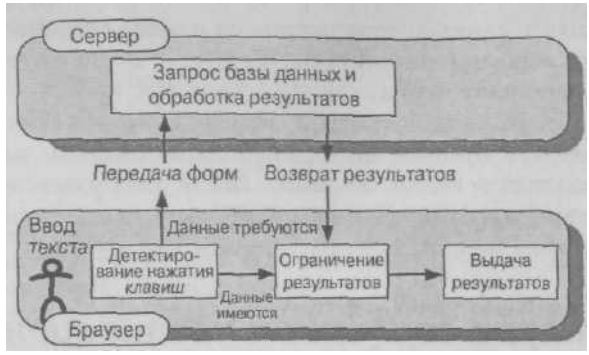


Рис. 10.2. Архитектура системы опережающего ввода

Одной из наиболее впечатляющих особенностей Google Suggest является поддержка быстрого набора — для ограничения числа обращений к серверу за короткий промежуток используются различные таймеры. Данный вопрос чрезвычайно важен для Google из-за огромного числа пользователей. Большое число обращений к серверу может порождать проблемы, кроме того, ограничение этого числа позволяет экономить ресурсы.

Пример Google вдохновляет нас (и многих других). В следующем разделе мы соберем лучшие из рассмотренных элементов и разработаем собственную схему опережающего ввода —- приложение Ajax.

10.1.3. Ajax как средство опережающего ввода

Средства опережающего ввода нашего приложения будут пытаться максимально уменьшить нагрузку на сервер. Небольшие сайты не могут обработать трафик, допустимый для Google, Amazon и Yahoo!, поскольку не имеют мощных регуляторов распределения нагрузки и множества серверов, обрабатывающих запросы. Следовательно, чем больше мы сэкономим полосу, тем дешевле будет эксплуатация небольших сайтов.

Для достижения поставленной цели будем использовать сервер только тогда, когда нам потребуются новые данные. В этом приложении мы используем основанное на сценарии взаимодействие клиент/сервер. Сервер форматирует результаты в виде выражений JavaScript, которые можно выполнить на стороне клиента. Данные, возвращаемые сервером, содержат только текст предлагаемых вариантов и идентификационное поле, которое мы желаем связать с этими данными. Это идентификационное поле можно сравнивать со значением или дескриптором опции. Большинство сценариев не позволяет отправлять идентификатор; в нашем это возможно. Далее браузер обрабатывает возвращенные данные как код JavaScript. Диаграмма этого процесса приведена на рис. 10.2.

Видно, что та часть сценария, которая собственно и представляет собой инфраструктуру Ajax, позволяет захватывать данные с сервера. Сервер возвращает клиенту текстовый документ, содержащий выражение JavaScript.

Клиент обрабатывает данные, содержащиеся в этом выражении, и проверяет имеются ли в них искомые результаты. Если да. то пользователю предлагав ются на выбор несколько вариантов. Все кажется просто, пока вы не заду, маетесь о том, сколько кода JavaScript задействовано в описанном процессе Впрочем, если разбить процесс на ряд небольших этапов, задача написание сценария опережающего ввода с минимальной нагрузкой на сервер оказывается сравнительно простой. В данном проекте самым простым будет код серверной части приложения, поэтому начнем мы именно с него.

10.2. Структура серверной части сценария: C#

Схема опережающего ввода, которую мы будем разрабатывать, состоит из трех частей: сервера, базы данных и клиента. На практике база данных может представлять собой, например, XML-файл, но общая суть от этого не меняется.

10.2.1. Сервер и база данных

Код сервера и базы данных можно писать одновременно, поскольку соединение с базой данных нам требуется только для получения информации. В данном примере мы используем базу данных Northwind (Microsoft), получая необходимые данные из таблицы Products, однако вы можете работать с любой удобной для вас базой данных.

Для отправки запроса от клиента серверу и получения в ответ отформатированной структуры данных нужен объект XMLHttpRequest. В нашем случае динамически создается текстовый документ, заполняемый информацией, полученной из запроса к базе данных. Этот текстовый документ будет содержать код JavaScript, создающий массив с данными. Основные этапы создания массива JavaScript приведены в листинге 10.1,

Листинг 10.1. Сценарий typeAheadData.aspx.cs

```
// О Инициализировать код при загрузке страницы
private void Page_Load(object sender,
System.EventArgs e)
{
// @ Установить тип содержимого
  Response.ContentType - "text/html";
// © Запросить элемент формы
  string strQuery =
    Request.Form.Get("q").ToString();
  string strAny = "";
// О Объявить строку
  if  (Request.Form.Get("where").ToLower()
    == "true")
  {
    strAny = "%";
  }
// © Построить выражение SQL
  string strSql = "Select top 15 " +
```

```
            "ProductName, " +
            "Productld FROM Products " +
            "WHERE ProductName like '" +
            strAny + strQuery + "%"
            '" ORDER BY ProductName";
// 0 Инициализировать запрос к базе данных
  DataTable dtQuestions = FillDataTable(
                          strSql);
// © Построить массив JavaScript
  System.Text.StringBuilder strJSArr =
    new System.Text.StringBuilder(
    "arrOptions = new Array(");
    int iCount = 0;
// © Последовательно пройти по результатам
    foreach (DataRow row in
      dtQuestions.Rows)
    {
      if (iCount > 0)
      { strJSArr.AppendC[1],"); }
      strJSArr.Append("[");
      strJSArr.Append("\"" +
        row["ProductName"].ToString() + "\",");
      strJSArr.Append("\"" +
        row["Productid"].ToString() + "\"");
      strJSArr.Append("]");
      iCount++;
    }
    strJSArr.Append(");");
// 0 Записать строку на страницу
    Response.Write(strJSArr.ToString());
}
// © Выполнить запрос
public static DataTable
    PillDataTable(string sqlQuery)
{
string strConn = "Initial Catalog = "+
      "Northwind;Data Source=127.0.0.1; "
      "Integrated Security=true;";
    SqlConnection conn = new SqlConnection(strConn);
    SqlDataAdapter cmdl = new SqlDataAdapter();
    cmdl.SelectCommand • new SqlCommand(sqlQuery,conn);
    DataSet dataSetl = new DataSetO;
    cmdl.Fill(dataSetl);
    icmdl.Dispose();
    conn.Close();
    return dataSetl.Tables[0];
```

Код, приведенный в листинге 10.1, позволяет получать значения от клиента и передавать данные в строку, формируя массив JavaScript. Этот массив возвращается клиенту и обрабатывается. Данное поведение требуется инициализировать при загрузке страницы документа О. Первое, что мы делаем после возврата строки, — проверяем, что тип содержимого страницы установлен равным text/html ©.

Код клиентской стороны помещает значения на нужную страницу посред, ством объекта XMLHttpRequest. Следовательно, мы должны запросить в элементе формы q текст, представляющий предположительные варианты ввода ©. В отличие от многих схем опережающего ввода, мы позволяем пользователям находить результаты в середине слова, поэтому объявляем строку о отвечающую за обработку такой ситуации. Сценарий клиента передает булеву строку с элементом формы where. Если ее значение равно `true`, мы добавляем % перед началом термина, чтобы разрешить поиск с любого места строки

Теперь мы можем сформировать выражение SQL ©, которое позволит получать из базы данных значения, соответствующие тому, что вводит пользователь. Чтобы минимизировать нагрузку на пользователя, мы ограничим поиск: возвращать можно всего 15 значений. Затем мы инициализируем © процедуру запроса базы данных © и возвращаем таблицу данных с доступными результатами поиска.

Получив результаты из базы данных, можно формировать массив JavaScript *О.* Затем мы циклически проходим по множеству записей ©, создавая двухмерный массив с именами вариантов и идентификаторами. После завершения цикла мы пишем нашу строку на страницу ©, чтобы ее могло использовать выражение JavaScript.

Если вы используете серверный язык со страницей кода и HTML-страницей, то с HTML-страницы необходимо удалить все дополнительные дескрипторы. Если используется такая же страница C#, как в нашем примере, на ASPX-странице должен присутствовать единственный указанный ниже дескриптор. - s.

```
<%@ Page Language="c#" AutoEventWireup="false"      \
Codebehind="TypeAheadXML.aspx.cs"                   \
Inherits="ChapterlOCS.TypeAheadXML"%>
```

Если мы не уберем дополнительные дескрипторы HTML, которые по умолчанию присутствуют на ASPX-странице, мы не получим допустимый текстовый файл (с кодом JavaScript), т.е. наши методы DOM JavaScript не смогут использовать данные. Чтобы гарантировать правильность данных, которые возвращаются клиенту, необходима быстрая проверка.

10.2,2. Тестирование серверного кода

При работе с Ajax тестирование серверного кода необходимо, поскольку JavaScript известен своими проблемами, причины которых часто бывает трудно обнаружить. Консоль JavaScript от Mozilla улучшает ситуацию, но чтобы минимизировать шансы на ошибку всегда стоит проверять корректность работы сервера.

При тестировании серверного кода можно пойти двумя путями. Поскольку объект XMLHttpRequest использует метод HTTP POST, мы можем либо создать простую форму с двумя текстовыми окнами и отправить ее на серверную страницу, либо закомментировать строки, проверяющие отправку формы, и задать в этом месте некоторые значения. Выберем второй вариант. Как видно из фрагмента кода, приведенного в листинге 10.2. мы заком-

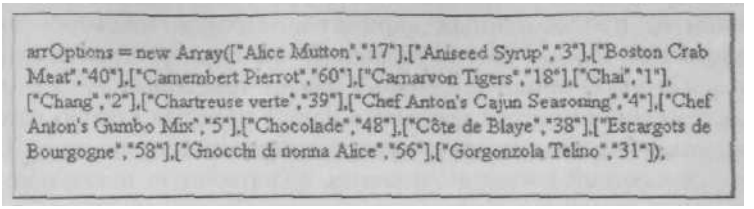


Рис. 10.3. Строка JavaScript, получаемая при тестировании на выходе серверной страницы

ментировали операторы запроса формы, заменив их жестко закодированными значениями.

Листинг 10.2. Фрагмент кода для тестирования

```
//string strQuery =
Request.Form.Get("q")-ToString();
string strQuery = "a";
string strAny = "";
//if (Request.Form.Get("where").ToLower() =- "true")
//{
  strAny = "%";
//}
```

В данном коде запрограммирован поиск всех слов, содержащих букву "a". Следовательно, при его выполнении объявление массива JavaScript выглядит так, как показано на рис 10.3. Мы видим, что приведенная строка выглядит правильно. Таким образом, комментарии и строки с присвоением значений можно убирать и продолжать написание сценария опережающего ввода. Возможно, вам интересно, где же выполняется кэширование данных, возвращаемых с сервера. За это отвечает сторона клиента. Сервер вызывается только раз: когда вводится первый символ или когда число результатов больше или равно 15. Нет никакого смысла запрашивать данные, если мы получим подмножество данных, полученных ранее.

Таким образом, мы завершили создание серверной части кода и переходим к клиентской.

10.3. Структура клиентской части сценария

Каркас клиентской части сценария содержит объект Ajax, именуемый XMLHttpRequest, и много DHTML-кода. Начнем с создания текстовых окон.

10.3.1. HTML

Используемый HTML-код очень прост, поскольку мы работаем всего с тремя элементами формы: двумя текстовыми окнами и скрытым полем. Первое текстовое окно представляет собой элемент формы с предложениями по опережающему вводу. Скрытое поле принимает значение элемента, выбранного

пользователем из предложенных вариантов. Второе текстовое окно проем не дает отправить форму на дообработку на сервер при нажатии клавиши <Enter>. По умолчанию в форме с одним текстовым полем с клавишей <Enter> соотнесена отправка формы. Самый простой способ отказаться от такого поведения — добавить на страницу второе текстовое поле. Если вы до. бавляете опережающий ввод на страницу, содержащую несколько элементов формы, дополнительное окно вам не требуется. Таким образом, мы приходи^ к общей структуре HTML-документа, показанной в листинге 10.3.

Листинг 10.3. Структура HTML-страницы с опережающим вводом

```
<form name="Forml" AUTOCOMPLETE="off" ID="Forml">
  AutoComplete Text Box: <input type="text" name="txtUserInput" />
  <input type="hidden" name="txtUserValue" lD="hiddenl" />
  <input type="text" name="txtlgnore" style="display:none" />
</form>
```

В листинге 10.3 мы добавили форму с отключенным автоматическим заполнением. Нам пришлось так сделать, чтобы браузер не помещал значения в поле при первой загрузке страницы. Описанная возможность очень удобна, но в данном случае она нарушит нашу схему опережающего ввода. Обратите внимание на то, что приведенный фрагмент используется только в Internet Explorer и не дает встроенным раскрывающимся спискам автозаполнения взаимодействовать с нашими раскрывающимися списками DHTML. Другие браузеры данный атрибут проигнорируют. Кремле того, мы добавили текстовое окно txtuserinput, скрытый элемент txtUserValue и ложное текстовое окно txtlgnore. С окном txtlgnore, используемым для предотвращения автоматической отправки формы, также соотнесен стиль CSS. согласно которому это окно невидимо. Данный результат можно получить и другими способами, но предложенный нами вариант является самым простым и быстрым решением. Итак, мы добавили в форму текстовые поля и можем писать код JavaScript.

10.3.2. JavaScript

Код JavaScript приложения с опережающим вводом выполняет три основные задачи.

- Наблюдение за действиями пользователя (вводом с помощью клавиатуры и мыши).
- Отправка данных на сервер/получение данных с сервера.
- Формирование HTML-содержимого, с которым может взаимодействовать пользователь.

Прежде чем .мы начнем кодирование, стоит подробнее остановиться на том, что же мы собираемся получить.

Когда пользователь набирает букву, становится видимым скрытый элемент span с информацией, имеющей отношение к набранной букве. На

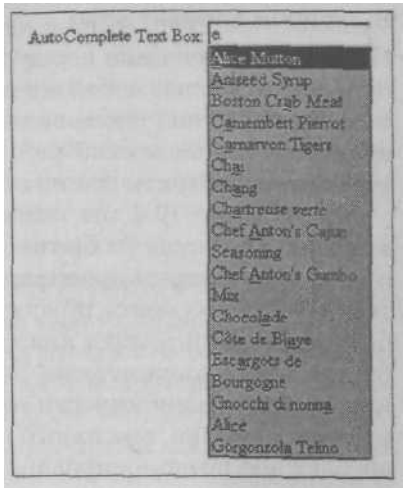


Рис. 10.4. Приложение опережающего ввода в действии

рис. 10.4 во всех вариантах подчеркнута буква "a" — буква, присутствующая в текстовом поле. Первая позиция списка выделена. Нажимая клавиши со стрелками вверх и вниз, можно выделять другие позиции. Нажав клавишу <Enter>, вы выберете выделенный вариант. Кроме того, нужную позицию можно выбрать, щелкнув на одном из слов списка.

Из-за сложности сценария объяснение может выглядеть немного непоследовательным, поскольку для реализации опережающего ввода требуется множество функций. Одна из этих функций отслеживает ввод с клавиатуры, вторая — загружает текст и код JavaScript, третья — создает список, четвертая — подчеркивает набранные буквы и т.д. Для удобства вы можете загрузить код этого сценария с Web-сайта книги и изучать его в своем любимом редакторе.

Добавление внешнего файла JavaScript (Ajax)

Чтобы добавить в наше приложение функциональные возможности Ajax, мы должны включить в дескриптор заголовка внешний файл JavaScript net.js (см. главу 3). Этот файл содержит объект ContentLoader, позволяющий инициировать запрос Ajax без всяких проверок if-else.

```
<script type="textjavascript" src="net.js"></script>
```

Чтобы присоединить внешний файл, мы добавляем дескриптор JavaScript и включаем атрибут src, задающий внешний файл. Ссылка на файл организовывается точно так же, как ссылка на изображение или файл CSS. Названный файл определяет, как отправлять информацию на сервер, скрывая весь зависящий от браузера код за удобным интерфейсным объектом. Таким образом, мы сможем отправлять и извлекать данные с сервера, не обновляя страницу. Подключив данный файл к нашему проекту, мы можем начинать разработку опережающего ввода.

Выходной элемент span

На рис. 10.4 показано серое окно, содержащее все доступные варианты. Эт$_0$ окно представляет собой элемент span HTML, который динамически Быстра. ивается точно под текстовым окном. Вместо того чтобы добавлять на страницу элемент span всякий раз, когда нам потребуется использовать сценарий мы можем добавить его на страницу из сценария.

В листинге 10.4 мы создаем новый элемент span с помощью DOM при загрузке страницы (событие onload). Мы вставляем span в страницу HTML с идентификатором spanOutput и именем класса CSS span Text Dropdown. Затем к телу документа присоединяется новый дочерний элемент — span. Добавленная нами ссылка класса CSS позволяет так присваивать правила, чтобы мы могли динамически размещать элемент span на странице. Поскольку нам придется динамически изменять его положение на экране в зависимости от расположения текстового окна, в соответствующем классе CSS мы задаем абсолютное позиционирование.

Листинг 10.4. Код JavaScript, отвечающий за вывод списка на экран

```
window.onload = function(){
  var elemSpan = document.createElement("span");
  elemSpan.id « "spanOutput";
  elemSpan.className = "spanTextDropdown";
  document.body.appendChild(elemSpan);
_ }
```

Для динамического добавления элемента span на страницу используется обработчик событий onload. Благодаря этому нам не требуется вручную добавлять этот элемент на страницу г>ри каждомчиспользовании сценария. Для создания списка применяется метод DOM, именуемый createElement. Затем нам нужно присвоить нашему новому элементу span идентификатор ID и атрибут className. Создав эти атрибуты, мы можем добавить элемент на страницу. Теперь создадим класс CSS (листинг 10.5), который позволит динамически располагать элемент на странице.

Листинг 10.5. Класс CSS для раскрывающегося списка

```
span.spanTextDropdown{ position: absolute;
                       top: Opx;
                       left: Opx;
                       width: 150px;
                       z-index: 101;
                       background-color: #C0COCO;
                       border: lpx solid #000000;
                       padding-left: 2px;
                       overflow: visible;
                       display: none;
```


Изначально элемент span располагается в произвольной точке экрана, заданной с помощью параметров top и left. Мы задаем ширину элемента

д$_0$ умолчанию и устанавливаем значение свойства z-index, соответствующее самому верхнему слою страницы. Правило CSS также позволяет определять $_c$тиль фона и границу нашего списка, чтобы он выделялся на странице. Свойство display задается равным попе, поэтому наш элемент невидим пользователю в момент первоначальной загрузки страницы. Как только пользователь начинает вводить данные в поле с опережающим вводом, свойство display меняется, поэтому мы можем увидеть результаты.

Связывание с текстовым окном структуры опережающего ввода

Поскольку возможности опережающего ввода могут нам понадобиться в нескольких полях, необходимо придумать, как приписать различным элементам различные свойства. Данные свойства определяют реакцию сценария. Мы задаем их так, чтобы текст сопоставлялся с учетом регистра, сравнивались вес элементы текста, предусматривалось время ожидания, а также выполнялись другие условия, которые мы опишем ниже. Чтобы достичь желаемого, можно создать объект, содержащий все необходимые параметры, уникальные для отдельного текстового окна. В таком случае, когда текстовое окно будет находиться в фокусе, мы сможем обращаться к объекту, присоединенному к элементу, и получать необходимые настройки. В листинге 10.6 показано, как создать новый объект, позволяющий организовать список параметров, которые мы приписали текстовому окну.

Листинг 10.6. Создание индивидуального объекта

```
function SetProperties(xElem,xHidden,xserverCode,
  xignoreCase,xmatchAnywhere,xmatchTextBoxWidth,
  xshowNoMatchMessage,xnoMatchingDataMessage,xuseTimeout,
  xtheVisibleTime){
    var props={
      elem: xElem,
      hidden: xHidden,
      serverCode: xserverCode,
      regExFlags: { (xignoreCase) ? "i" : "" ),
      regExAny: ( (xmatchAnywhere) ? " " : " " ) ,
      matchAnywhere: xmatchAnywhere,
      matchTextBoxWidth: xmatchTextBoxWidth,
      theVisibleTime: xtheVisibleTime,
      showNoMatchMessage: xshowNoMatchMessage,
      noMatchingDataMessage: xnoMatchingDataMessage,
      useTimeout: xuseTimeout
    1;
    AddHandler(xElem);
    return props;
```

Первым шагом при создании наших объектов для опережающего ввода является написание новой функции setProperties(), которая может присваивать свойства объекту. В данном примере мы будем передавать этой функции несколько параметров. В их число входит текстовое окно, с которым соотнесено средство опережающего ввода; скрытый элемент, используемый

для хранения значения; указатель URL на страницу сервера, булева переменная, указывающая игнорировать регистр при поиске; булева переменная указывающая искать текст в любом месте строки; булева переменная, указывающая отображать сообщение "нет соответствий"; отображаемое сообщение-булева переменная, определяющая, следует ли скрывать варианты по прошествии указанного периода времени; а также временной интервал, в течение которого демонстрируются варианты.

Описанный набор параметров слишком велик, чтобы передавать его функции. Мы должны взять эти параметры и присвоить их нашему объекту. Для этого используем форму записи JavaScript Object Notation (JSON) которая подробно описана в приложении Б. Ключевое слово определяется перед двоеточием, затем следует значение. Трактовка параметров ignoreCase и matchAnywhere несколько сложнее. Вместо хранения булева значения мы записываем в свойство эквивалентное регулярное выражение. В данном случае мы используем символ i для обозначения "игнорировать регистр" и ″ — для отметки начала строки в регулярных выражениях. Выбор такого подхода объясняется тем, что нам проще задать параметры регулярных выражений, чем использовать оператор if при каждом вызове функций.

Последним действием нашей функции является связывание с текстовым окном обработчиков событий. В данном примере мы вызываем функцию, автоматически добавляющую обработчики событий. Код этой функции будет приведен чуть ниже, а пока вызовем функцию SetProperties() для создания нашего объекта. Код, приведенный в листинге 10.7, выполняется обработчиком события onload (загрузка страницы) и позволяет задавать свойства текстового окна.

Листинг 10.7. Инициализация сценария, . Щ ^ ^ ^ ^ ^ Ш Я

```
window.onload = function(){
  var elemSpan = document.createElement("span");
  elemSpan.id = "spanOutput";
  elemSpan.className = "spanTextDropdown";
  document.body.appendChild(elemSpan);
  document.Forml.txtUserInput.obj =
    SetProperties(document.Forml.txtUserInput,
  document.Forml.txtUserValue,'typeAheadData.aspx',
  true,true,true,true,"No matching Data",false,null);
  }
```

Обработчики событий должны быть заданы при загрузке страницы. Следовательно, мы должно присвоить их обработчику событий window.onload, который мы создали ранее для добавления нового элемента span. В данном тримере мы используем только одно текстовое окно с опережающим вводом. Цалее нам потребуется обратиться к элементу формы, с которым мы желаем связать возможности опережающего ввода, и добавить к нему новое свойство >bj. Вместо того чтобы использовать глобальные переменные, присвоим дангому свойству наш индивидуальный объект, чтобы для получения значений с нему можно было обращаться из сценария.

В качестве ссылки используется функция SetPropertiesO. Затем мы присваиваем все параметры, созданные в листинге 10.6. Здесь важно ответить, что мы ссылаемся на два элемента формы, созданных в листинге Ю.З, и вызываем серверную страницу typeAh.eadData.aspx, созданную в листинге 10.1. Теперь, когда обработчик onload инициализировал процесс, мы можем добавлять обработчики событий, которые вызывает функция SetPropertiesO-

Обработчики событий

Чтобы мы могли определить действие пользователя, выполненное в текстовом окне, и предложить варианты опережающего ввода, требуется добавить к форме обработчики событий. В связи с этим нужно знать два основных момента: набирает ли пользователь что-либо на клавиатуре, и завершил ли он работу с текстовым полем. Для детектирования действий пользователя мы используем обработчики событий, приведенные в листинге 10.8.

Листинг 10.8. Добавление обработчиков **событий**, - • ;

```
var
 isOpera=(navigator-userAgent.toLowerCase().indexOf{"opera")!= -1);
     function AddHandler(objText){
       objText.onkeyup = GiveOptions;
       objText.onblur = function(){
         if(this.obj.useTimeout)StartTimeout();
       1
       if(isOpera)objText.onkeypress = GiveOptions;
_}
```

Листинг 10.8 начинается с выражения, определяющего используемый браузер. Вообще, в данном примере мы будем несколько раз использовать детектирование браузера, поскольку Opera детектирует ввод с клавиатуры не так, как другие браузеры. Указанный способ позволяет проще всего выяснить, используется ли в качестве браузера Opera, но он является не самым надежным, поскольку может имитировать поведение других браузеров.

Наша функция AddHandler () получает ссылку на текстовое окно. Эта ссылка позволяет добавлять к элементу обработчики событий onkeyup и onblur. Обработчик событий onkeyup запускает функцию GiveOptions () при отпускании клавиши клавиатуры. Следовательно, когда пользователь набирает пятибуквенное слово, функция GiveOptions запускается пять раз при отпускании клавиш.

Обработчик событий onblur, который мы связали с текстовым окном, вызывает функцию StartTimeout () (она представлена в листинге 10.19) в тот момент, когда текстовое окно выходит из фокуса. Действия, вследствие которых окно может потерять статус находящегося в фокусе, включают щелчки на других частях экрана или нажатие клавиши <TaЬ>.

Почему мы особо обратили внимание на детектирование браузера Opera? Дело в том, что он не так реагирует на обработчик событий onkeyup, как другие браузеры, — при срабатывании onkeyup Opera не показывает значе-

ние в текстовом окне, в которое в текущий момент вводится текст. Прооге.\щ решается добавлением обработчика событий onkeypress. Из приведенного кода видно, что мы проверили используемый браузер с помощью булевой переменной isOpera, а затем присвоили текстовому окну обработчик событий onkeypress. Благодаря этому обработчику событий Opera служит нашим целям так же, как и другие браузеры. Поскольку теперь мы можем детектировать то, что пользователь вводит с клавиатуры, мы можем определить какие действия требуются в функции GiveOptions ().

Обработка нажатия клавиш

Мы собираемся создать функцию GiveOptions (), которая вызывается при нажатии клавиш. Основных задач у этой функции две: определить действие в зависимости от нажатой клавиши и решить, требуется ли использовать Ajax для получения данных с сервера или можно задействовать уже имеющуюся информацию. Таким образом, функция GiveOptions () требуется для того же, что и кэширование данных, рассмотренное в разделе 10.1.1. Используя для обработки дополнительных нажатий клавиш клиентский код, мы уменьшаем потребление полосы пропускания приложением опережающего ввода. Чтобы реализовать кэш доступных вариантов, установим на стороне клиента несколько глобальных переменных. Список глобальных переменных, с которых мы начнем нашу работу, приводится в листинге 10.9.

Листинг 10.9. Глобальные переменные, используемые в проекте

```
var arrOptions = new Arrayf);
var strLastValue = "";
var bMadeRequest;
var theTextBox;
var objLastActive;                          \.
var currentValueSelected = -1;                 \
var bNoResults = false;                          \
var isTiming = false;                             \                    _
```

Первая глобальная переменная — arrOptions — ссылается на массив, который содержит все доступные опции из запроса сервера. Следующая переменная — strLastValue — содержит последнюю строку, которая находилась в текстовом окне. Переменная bMadeRequest представляет собой булеву метку, благодаря которой мы узнаем, что запрос уже отправлен серверу (поэтому нам не нужно отправлять дополнительные запросы). Эта метка подходит для решения проблемы быстрого набора, поэтому нам не нужно вводить специальные таймеры, как в Google Suggest.

Переменная theTextBox будет содержать ссылку на текстовое окно, находящееся в фокусе, а objLastActive — ссылку на последнее активизированное текстовое окно. Таким образом определяется, требуется ли обновление набора данных при переключении пользователем текстовых окон. В нашем примере видимое текстовое окно всего одно, но если данное решение будет реализовано в окне с несколькими текстовыми окнами, нам потребуется знать, какое из них находится в фокусе. Следующая переменная. currentValueS-

greeted, действует подобно переменной selectedlndex списка. Если ее значение равно -1, не выбирается ничего. Последняя необходимая на текущий иомент переменная — булева bNoResults. Посредством этой переменной сообщается, что результатов нет, поэтому мы можем их не искать. Переменная ^sTiming позволяет определять, запущен ли таймер на странице. Этот таймер нужен для того, чтобы скрыть от пользователя предлагаемые варианты
в период бездействия.

Возможно, вы не совсем поняли роли всех этих глобальных переменных, но. начав их использовать, вы разберетесь с ними лучше. Ссылаясь на эти глобальные переменные, мы можем создать функцию GiveOptionsO, которая вызывается при нажатии клавиш в текстовом окне. Функция GiveOptions(), приведенная в листинге 10.10, позволяет определять действие, которое пользователь произвел в текстовом окне.

Листинг 10.10. Код JavaScript, отвечающий за детектирование нажатия клавиш

```
//O Детектировать нажатие клавиши
function GiveOptions(e){
  var intKey « -1;
  if(window.event){
    intKey = event.keyCode;
    theTextBox = event.srcElement;
  }
  else{
    intKey = e.which;
    theTextBox = e.target;
  }
// © Обновить таймер
  if(theTextBox.obj.useTimeout){
    if(isTiming)EraseTimeout();
    StartTimeout();
  }
II © Проверить, существует пи текст
  if(theTextBox.value.length == 0
    && !isOpera){
    arrOptions = new ArrayO;
    HideTheBox();
    strLastValue = "";
    return false;
  }
// O Определить функциональные клавиши
  if(objLastActive == theTextBox)(
    if(intKey — 13){
      GrabHighlighted();
      theTextBox.blur();
      return false;
    }
    else if(intKey == 38){
      MoveHighlight(-l);
      return false;
    }
    else if(intKey == 40){
      MoveHighlight(l);
      return false;
```

```
// © Обработать действия, соотнесенные с нажатиеы клавиш
  if(objLastActive != theTextBox ||
    theTextBox.value
    .indexOf{strLastValue) != 0 ||
    ((arrOptions,length==0 ||
    arrOptions.length==15 )
    SS IbNoResults) ||
    (theTextBox.value.length
      <= strLastValue.length)){
    objLastActive = theTextBox;
    bMadeRequest = true
    TypeAhead(theTextBox.value)
  }
  else iff!bMadeRequest){
    BuildList(theTextBox.value);
  }
// ® Записать то, что ввел пользователь
  strLastValue = theTextBox.value;
}
```

Если пользователь набирает слово, то наша функция либо начинает новый поиск, проверяя через сервер наличие соответствующих данных, либо работает с кэшированным набором результатов. Если извлекать новые данные с сервера не требуется, вызывается функция BuildListO, ограничивающая набор результатов. Подробнее данный вопрос рассмотрен ниже, в разделе "Создание окна результатов".

Функция GiveOptionsO объявляется с параметром e, что позволяет детектировать источник события. Прежде всего нам нужно объявить локальную переменную intKey, содержащую код нажатой пользователем клавиши О. Чтобы определить, какая клавиша нажата, необходимо узнать, какой метод нужен для работы выбранного пользователем браузера. Если поддерживается свойство window.event — мы имеем дело с Internet Explorer. Чтобы получить код клавиши, применяется свойство event. keyCode, а для получения объекта, соотнесенного с текстовым окном, — свойство event.srcElement. Для получения кода клавиши в других браузерах применяется параметр e.which, ссылка на объект текстового окна находится с помощью e.target.

Далее нужно проверить, использует ли текстовое окно таймер, согласно которому окно скрывается через некоторое время ©. Для этого необходимо обратиться к свойству obj текстового окна (мы создали его ранее) и булевой переменной useTimeout. Если таймер запущен, мы останавливаем его и перезапускаем, вызывая функции EraseTimeout() и StartTimeout() (мы напишем их в разделе "Использование таймеров JavaScript").

Затем мы проверяем, имеется ли что-либо в текстовом окне ©. Если окно пусто, вызываем функцию HideTheBoxO (разрабатывается в разделе "Установка выбранного значения"), устанавливающую значение strLastValue равным null, и возвращаем значение false, чтобы выйти из функции. Если текстовое окно содержит текст, мы продолжаем выполнять функцию. До того

как мы начнем детектирование клавиши со стрелками и <Enter>, мы должны убедиться, что текущее активизированное текстовое окно совпадает с последним активизированным текстовым окном.

Первой клавишей, за нажатием которой мы должны следить, является <Enter>. имеющая код 13 О. Нажатие клавиши <Enter> позволит нам захватить значение выбранной позиции раскрывающегося списка и поместить его Б видимое текстовое окно. Таким образом, мы вызываем функцию GrabHighlighted() (ее мы тоже напишем в разделе "Установка выбранного значения"). После этого мы снимаем фокус с текстового окна и выходим из функции.

Далее нам требуются клавиши со стрелками вверх и вниз, имеющие коды 38 и 40 соответственно. При нажатии клавиш со стрелками выделение позиции перемещается вверх-вниз по списку. Выделение, показанное на рис. 10.4, имеет вид темно-серой полоски. Нажав клавишу со стрелкой вниз, можно выделить следующую позицию списка. Подробности реализации этого процесса рассмотрены ниже, в разделе "Выделение позиции". Пока же мы просто отметим, что при нажатии клавиши со стрелкой вниз функции MoveHighlight() отправляется значение 1, а при нажатии клавиши со стрелкой вверх —'значение -1.

Если "особые" клавиши не нажимались, мы проверяем, требуется ли для получения значений обращение к серверу, или мы можем взять их из списка, полученного ранее ©. В этом месте мы снова используем реализованный в нашем сценарии механизм кэширования, ограничивая число обращений к серверу (а также нагрузку на сервер). Итак, с помощью различных проверок нам нужно выяснить, требуются ли новые результаты. Вначале мы определяем находится ли в фокусе в текущий момент окно, активизированное последним. Далее проверятся, что текст, в текущий момент набранный пользователем в текстовом окне, отличается от предыдущего только добавлением новых символов в конце. Если результатов для вывода на экран не найдено или наше множество результатов насчитывает не больше пятнадцати элементов, необходимо получить данные с сервера. Последняя проверка позволяет убедиться, что длина текущего значения больше предыдущего. Получать данные с сервера требуется только тогда, когда это покажет любая из проверок. В таком случае мы устанавливаем статус objLastActive для текущего текстового окна. Затем задается булево значение, показывающее, что был отправлен запрос (это нужно для того, чтобы мы не отправили несколько запросов), и вызывается функция TypeAhead(), отвечающая за захват значений.

После этого мы устанавливаем текущую строку в текстовом окне равной доследней известной строке ®. Это значение будет использовано еще раз, когда при вводе следующего символа мы проверим, требуется ли запрашивать с сервера новые данные. Таким образом, далее мы должны рассмотреть обращение к серверу для получения данных.

10.3.3. Обращение к серверу

Объект xMLHttpRequest позволяет передавать текст из текстового окна на сервер и получать результаты с сервера. В данном случае мы отправляем данные на сервер, поскольку серверная страница, созданная в листинге 10.1 обращается к элементам, отправленным в форме. Как показано в листинге 10.11, в объекте ContentLoader требуется задать положение страницы на сервере, функцию, вызываемую после завершения выполнения этого объекта а также параметры формы, которые будут отправлены форме.

Листинг 10.11. Функция Ajax, используемая для отправки данных на сервер \

```
function TypeAhead(xStrText){
  var strParams • "q=" + xStrText +
    "&where=" + theTextBox.obj.matchAnywhere;
  var loaderl = new net.ContentLoader(
    theTextBox.obj.serverCode, BuildChoices,null,"POST",strParams);
)
```

При вызове функции TypeAhead() из функции GiveOptions () мы (для выполнения поиска) передаем ей текущее значение строки из текстового окна. Нам требуется создать строку параметров, strParams, содержащую значение строки в текстовом окне, а также булево значение matchAnywhere. Оба названных элемента использовались в листинге 10.1 для получения результатов поиска. Затем мы начинаем загружать документ, вызывая Content-Loader. Для этого в качестве двух первых параметров ContentLoader мы отправляем URL серверной страницы и функцию JavaScript, которая будет вызвана при возврате результатов. Третий параметр равен null, поскольку мы собираемся игнорировать все сообщения об ошибках. Благодаря этому поле с опережающим вводом будет выглядеть подобно обычному текстовому полю. Последние два параметра указывают ContentLoader поместить данные на сервер и отправить параметры формы, содержащиеся в строке strParams.

После возврата результатов с сервера вызывается функция Build-Choicest), позволяющая завершить обработку данных на стороне клиента. Разрабатывая код серверной части сценария, мы возвращали результаты в виде двухмерного массива JavaScript. Этот массив содержал пары "текст-значение" предлагаемых вариантов выбора. Тем не менее в отклике мы имеем просто строку символов. Следовательно, нужно взять эту возвращенную строку и отформатировать ее как массив JavaScript. Соответствующая функция, обрабатывающая информацию, полученную от ContentLoader с помощью метода eval(), приводится в листинге 10.12.

Листинг 10.12. Преобразование свойства responseTextB массив JavaScript

```
function BuildChoices(){
  var strText = this.req.responseText;
  eval(strText);
  BuildList(strLastValue);
  bMadeRequest = false;
```

Свойство responseText возвращенного объекта запроса позволяет получать текст из запроса Ajax. Чтобы данную возвращенную строку можно было использовать в нашем коде JavaScript, необходим метод evalO, нужным образом обрабатывающий строку, переданную ему в качестве аргумента. В данном случае этот метод распознает, что строка представляет собой объявление переменной для создания нового массива. Если бы мы просто записали строку на страницу, она не была бы доступна для выражения JavaScript. Как правило, разработчики не приветствуют использование метода eval{) из-за его известной медлительности. Однако в данном случае он позволяет отказаться от циклического прохода XML-документа на стороне клиента для получения значений. Теперь мы можем вызвать функцию BuildListO, форматирующую и отображающую возвращенные результаты. Кроме того, мы устанавливаем значение булевой переменной bMadeRequest равным false, сообщая оставшейся части сценария, что запрос к серверу завершен.

Создание окна результатов

Использование JavaScript для обработки текущего документа обычно считается признаком DHTML. В данном примере мы принимаем двухмерный массив и превращаем его в строки текста на экране. Вернувшись к рис. 10.4, мы увидим список слов с подчеркнутыми фрагментами — частями текста, совпадающими с тем, что ввел в окне пользователь. В нашем приложении данные слова будут отображаться как элемент span.

Функция BuildList(), созданная в листинге 10.13. использует три функции: поиск слов по шаблону, установку положения окна и форматирование результатов с использованием подчеркивания.

Листинг 10.13. Форматирование результатов в формат отображения

```
function BuildList(theText){
  SetElementPosition(theTextBox);
// Установить положение элемента                          ,
  var theMatches - MakeMatches(theText);
// Отформатировать соответствия
  theMatches = theMatches.join().replace{/\,/gi,"");
// Показать результаты
  if(theMatches.length > 0){
    document.getElementByld("spanOutput")
       .innerHTML = theMatches;
    document.getElementByldf
      "OptionsList_0").className =
      "spanHighElement";
    currentValueSelected = 0;
    bNoResults = false;
  }
//Не показывать варианты
  else{
    currentValueSelected = -1;
    bNoResults = true;
    if{theTextBox.obj.showNoMatchMessage)
      document.getElementByld(
        "spanOutput").innerHTML =
```

```
            "<span    class='noMatchData'>"    +
            theTextBox.obj
            . noMa.tchinqDataMessa.ge    +
            "</span>";
        else      HideTheBox();
    }
```

Функция BuildListO, приведенная в листинге 10.13, принимает строку введенную пользователем, и форматирует результаты. Первое, что мы должны сделать, — это динамически разместить элемент *span непосредственно* под текстовым окном, в котором реализован опережающий ввод. Для этого мы вызываем функцию SetElementPositionO (подробнее об этом — ниже, в разделе "Динамическая установка положения элемента"). Расположив элемент span в нужном месте страницы, мы можем манипулировать массивом, отыскивая соответствия с использованием функций MakeMatches() (речь о ней пойдет в разделе "Использование регулярных выражений"). Эта функция возвращает массив, содержащий только информацию, согласующуюся с введенной пользователем. В отличие от большинства интерактивных приложений опережающего ввода мы не требуем обработки на сервере, а с помощью JavaScript ограничиваем результаты на стороне клиента.

Функция MakeMatches () форматирует результаты и возвращает их в виде массива. Затем мы превращаем этот массив в строку, используя метод join. Если длина строки *больше* 0, тогда мы можем отображать результаты в виде списка, задав его свойство innerHTML. Затем мы выбираем первый элемент списка и устанавливаем его свойство className, чтобы выделить этот элемент.

Если возвращенный массив не содержит данных, мы отображаем сообщение "до matches" ("нет соответствий"), если текстовое окно это позволяет. Мы знаем, что соответствий нет, поскольку проверяем, что значение currentselectedValue установлено равным -1. Если никаких сообщение отображать не требуется, мы скрываем окно.

Итак, мы завершили выполнение функции BuildListO и теперь можем создавать все функции, которая она вызывает. Первой из них рассмотрим SetElementPosition().

Динамическая установка положения элемента

За расположение текстового окна ввода на странице отвечает процессор браузера. Создавая раскрывающийся список с предложениями DHTML, мы желаем выровнять его точно по текстовому окну. Поскольку нам требуются координаты раскрывающегося списка, то приходится решать *довольно* сложную задачу нахождения положения элемента (в нашем случае — текстового окна), которое не задано жестко. Такие элементы размещаются на странице относительно — без указания абсолютных координат верхнего левого угла. Попытавшись сослаться на левую верхнюю точку текстового окна, мы получим в ответ неопределенную *строку.* Следовательно, *для* определения по-

дожения элемента нужно использовать возможности JavaScript — тогда мы сможем выровнять окно с вариантами выбора непосредственно под текстовым окном. Динамическое размещение элемента списка с выравниванием его под текстовым окном представлено в листинге 10.14.

Листинг 10.14. Динамическое нахождение положения элемента

```
function SetElementPosition(theTextBoxInt){
  var selectedPosX - 0;
  var selectedPosY = 0;
  var theElement « theTextBoxInt;
  if (!theElement) return;
  var theElemHeight = theElement.offsetHeightr
  var theElemWidth = theElement.offsetWidth;
  while(theElement != null)I
    selectedPosX += theElement.offsetLeft;
    selectedPosX += theElement.offsetTop;
    theElement = theElement.offsetParent;
  }
  xPosElement • document.getElementById("spanOutput");
  xPosElement.style.left = selectedPosX;
  if(theTextBoxInt.obj.matchTextBoxWidth)
    xPosElement.style.width • theElemWidth;
  xPosElement.style.top = selectedPosY + theElemHeight
  xPosElement.style.display = "block";
  if(theTextBoxInt.obj.useTimeout){
    xPosElement.onmouseout = StartTimeout;
    xPosElement.onmouseover = EraseTimeout;
  }
  else{
    xPosElement.onmouseout • null;
    xPosElement.onroouseover *= null;
  }
}
```

В листинге 10.14 мы объявляем функцию SetElementPosition(), принимающую один параметр: ссылку на объект текстового окна. Значения двух локальных переменных, selectedPosX и selectedPosY, устанавливаются равными 0. Эти две целочисленные переменные используются для расчета положения элемента. Ссылка на текстовое окно задается в другой локальной переменной. Для получения ширины и высоты текстового окна применяются свойства offsetHeight и offsetWidth.

Для последовательного прохода дерева документа используется цикл while. Дерево документа позволяет получить координаты X и У положения элемента относительно его родителя. Последовательно проходя *всех* предков искомого элемента, мы можем найти его точное положение, прибавляя смещения к созданным выше двум локальным переменным.

Получив положение текстового окна, мы можем извлекать объектную ссылку списка, использованную для задания верхней левой точки раскрывающегося списка с предложениями. Затем мы изучаем объект obj текстового окна, проверяя, должно ли его свойство width согласовываться с шириной текстового окна. Если соответствующее булево значение равно true, мы

устанавливаем ширину списка. Если значение равно false, ширина задается равной значению, указанному в таблице стилей. Последнее, что от нас требуется, — изменить параметр видимости списка, чтобы он больше не скрывался от пользователя. Для этого свойство display устанавливается равным block.

Подытожим: мы правильно разместили список и сделали его видимым пользователю. Теперь можно разрабатывать код, заполняющий список предположениями.

Использование регулярных выражений

Поскольку приложение предполагает поиск фрагментов строки, напомним, что одним из лучших инструментов поиска соответствий являются регулярные выражения, обладающие к тому же хорошей гибкостью. Функция MakeMatchesO, которую мы создадим ниже, позволяет находить в списке вариантов слова, согласующиеся с тем, что пользователь ввел в текстовом окне. Это означает, что мы можем обойтись без обращения к серверу после каждого нажатия клавиши, так как необходимый отбор вариантов производится функцией на стороне клиента. Код, приведенный в листинге 10.15, позволяет сэкономить полосу пропускания, ограничивая множество полученных результатов.

Листинг 10.15. Ограничение результатов с помощью регулярных выражений

```
var countForld = 0;
function MakeMatches(xCompareStr){
  countForld = 0;
  var matchArray = new Array();
  var regExp = new RegExp(theTextBox.obj.regExAny +
    xCompareStr,theTextBox.obj.regExFlags);
  for(i=0;i<arrOptions.length;i++){
    var theMatch = arrOptions[i][0].match(regExp);
    if(theMatch){
      matchArray[matchArray.length]=
        CreateUnderline(arrOptions[i][0], xCompareStr,i);

}
```

Мы создаем функцию MakeMatches (), которая принимает один параметр: строку, введенную пользователем. Затем мы присваиваем переменной countForld значение 0 и создаем локальную переменную массива matchArray. (Обратите внимание на то, что countForld — это глобальная переменная. Это позволяет немного упростить структуру примера. Позже мы избавимся от этой переменной.) Суть данной функции заключается в создании регулярного выражения, находящего варианты, согласующиеся с тем, что вводит пользователь. Поскольку мы уже определили параметры для регулярного выражения, создавая код, приведенный в листинге 10.6, сейчас нам требуется сослаться на объект текстового окна. Далее мы добавим свойство regExAny, позволяющее согласовывать текст, начиная с начала или любого места стро-

$#• Свойство regExFlags позволяет определять, следует ли при поиске соответствий игнорировать регистр.

По завершении работы с регулярными выражениями мы последовательно проходим массив arrOptions, проверяя, действительно ли находящиеся в нем варианты согласуются с нашим регулярным выражением. Если да, то МЫ добавляем текст в массив matchArray после вызова функции CreateUnderline(), которая форматирует код, отображаемый на экране.

Завершив проход по всем элементам массива, мы возвращаем найденные соответствия основной функции BuildList{), после чего отобранные варианты отображаются на экране. Функция MakeMatches() предоставляет механизм кэширования, о котором мы уже говорили выше. Вместо повторного обращения к серверу за ограничением множества результатов после каждой набранной буквы мы используем регулярные выражения, сокращая уже имеющийся набор вариантов. Последним этапом обработки результатов является вызов функции CreateUnderline (), которая обеспечивает нужное форматирование.

Обработка строк

Последний этап форматирования строк, позволяющий пользователю видеть их и взаимодействовать с ними, заключается в следующем: в строках, находящихся в списке, подчеркивается фрагмент текста, отображенный в текстовом окне, и с каждой позицией списка соотносится обработчик событий onclick, позволяющий пользователю выбирать позицию с помощью мыши. Создание строки, отформатированной нужным образом с помощью регулярных выражений, представлено в листинге 10.16.

Листинг 10.16. Обработка строк средствами JavaScript

```
var undeStart = "<span class='spanMatchText'>";
var undeEnd = "</span>";
var selectSpanStart = "<span style='width: 100%,-display:block; '
  class='spanNormalElement' onmouseover=•SetHighColor(this)' ";
var selectSpanEnd ="</span>";
function CreateUnderline(xStr,xTextMatch,xVal){
  selectSpanMid = "onclick='SetText(" + xVal + " ) ' " +
    "id='OptionsList_"+countForId+ "'theArrayNumber='"+ xVal +'">";
  var regExp = new RegExp{theTextBox.obj.regExAny +
    xTextMatch,theTextBox.obj.regExFlags);
  var aStart = xStr.search(regExp);
  var matchedText = xStr.substring(aStart,
                        aStart + xTextMatch.length);
  countForId++;
  return selectSpanStart + selectSpanMid +
    xStr.replace(regExp,undeStart +
    matchedText + undeEnd) + selectSpanEnd;
```

В листинге 10.16 определяются две переменные, которые содержат строки, используемые для привязки класса CSS к фрагменту текста, согласующемуся с заданной строкой. Благодаря этому можно легко определить :тиль нужного текста. Первая переменная, undestart, содержит открывающий дескриптор span; вторая, undeEnd, — соответствующий закрывающий дескриптор.

Следующие две переменные формируют контейнер для всей строки. Этот сонтейнер позволяет манипулировать цветом фона и определять, щелкнул ли юльзователь на ячейке. Чтобы визуально выделить ячейку, на которую на-!еден указатель мыши, мы добавили в переменную selectSpanStart событие louseover. Переменная selectSpanEnd представляет собой закрывающий декриптор элемента span.

Функция CreateUnderline() вызывается функцией MakeMatches<), коюруго мы написали чуть выше. MakeMatches () принимает три параметра: троку, введенную пользователем, текст предлагаемого варианта, а также начение данной опции. Используя переданные данные, можно создать об-•аботчик onclick и добавить к списку идентификатор. Обработчик событий nclick позволяет выбрать один из предложенных вариантов, а идентифиатор — использовать DOM для выбора варианта из списка.

Для сопоставления текста, введенного пользователем, с фрагментами редлагаемых вариантов мы снова используем регулярные выражения. Таим образом мы можем вставить в строку созданные блоки с подчеркиванием. [тобы определить, в каком месте строки находится соответствие, применяет-^ метод search. Найдя положение искомой строки, мы можем получить подгроку, в которой можно сохранить исходное форматирование. Далее значене счетчика countForld увеличивается на 1, и мы возвращаем отформатиро-шную строку, объединяя все созданные элементы span. Теперь текст отфорратирован, осталось дописать классы CSS, добавленные к элементам span.

Элементам span были присвоены имена классов CSS, поэтому от нас 3 требуется вручную редактировать код JavaScript, изменяя определенные юйства текста. Это позволяет подогнать текстовое окно с опережающим юдом под любую цветовую схему, изменив несколько правил CSS.

```
span.spanMatchText{ text-decoration: underline;
                    font-weight: bold; }
span.spanNormalElement{ background: #C0C0C0; }
span.spanHighElement{ background: #000040;
                      color: white; cursor: pointer; }
span.noMatchData{ font-weight: bold; color: #0000FF; }
```

Напомним (см. рис. 10.4), что текст списка, согласующийся с текстом на, подчеркнут и выделен полужирным. Оба этих свойства заданы в прале CSS span.spanMatchText. Стиль блока по умолчанию (серый цвет фо) представлен в правиле span. spanNormalElement. К выбранному элементу именяется правило CSS span.spanHighElement. На рисунке также видно, о фон имеет темно-серый цвет, а текст представлен белым. Кроме того, рсор был заменен указателем, поэтому пользователь знает, что вариант жно выбрать с помощью мыши. К любому из этих элементов можно добагь и другие свойства: шрифт, размер, границу и т.д. Итак, мы сформулиро-

вали правила таблицы стилей и закончили работу по выводу результатов на экран. Осталось только реализовать обработку клавиш со стрелками и клавиши <Enter> и создать таймер, который будет скрывать список вариантов в моменты бездействия.

Выделение опций

ранее в этой главе мы показали, как отслеживать нажатия клавиш со стрелками вверх и вниз, чтобы пользователь перемещал selectedIndex вверх-вниз по списку, не прибегая к помощи мыши. При нажатии клавиш со стрелками мы получали 1 ("сместить выбор вниз") или -1 ("сместить выбор вверх"). При перемещении выбора мы применяем правила классов CSS к элементам span. Кроме того, мы создали глобальную переменную currentValueSelected, в которую помещаем номер текущей позиции. Функция MoveHighlight(), приведенная в листинге 10.17, предлагает более богатый пользовательский интерфейс, поскольку взаимодействует и с мышью, и с клавиатурой.

Листинг 10.17. Изменение имен классов CSS с помощью JavaScript

```
function MoveHighlight(xDir){
  if (currentValueSelected >= 0){
    newValue = parseInt(currentValueSelected) + parseInt(xDir);
    if(newValue > -1 && newValue < countForId){
      CurrentValueSelected = newValue;
      SetHighColor (null);
    1
  }
}
function SetHighColor(theTextBox){
  if(theTextBox){
  currentValueSelected =
  theTextBox.id.slice(theTextBox.id.indexOf("_")+1,
  theTextBox.id.length);
  }
  for{i = 0; i < countForId; i++){
    document.getElementById('OptionsList_' + i).className =
      'spanNormalElement';
  }
  document.getElementById('OptionsList__' +
    currentValueSelected).className = 'spanHighElement';
}
```

Функция MoveHighlight t) позволяет пользователю использовать для выбора клавиши со стрелками вверх и вниз. Эта функция принимает один параметр, xDir, обозначающий направление, в котором должно сместиться выделение. Прежде всего мы должны убедиться, что у нас есть варианты на выбор и можно получить новое значение. Далее мы проверяем, что это новое значение принадлежит разрешенному диапазону. При положительном ответе мы устанавливаем состояние currentValueSelected и переходим к следующей функции, SetHighColor(), выделяя новую выбранную позицию.

Функция SetHighColor() вызывается двумя различными событиями: нажатием клавиш со стрелками и наведением указателя мыши на позицию (обработчик событий onmouseover). Эта функция предназначена для удаления подсветки с последней выбранной опции и добавления ее к новой выбранной позиции. Событие onmouseover (см. листинг 10.16) принимает в качестве аргумента объект блока предложенных вариантов; следовательно, мы должны отсечь идентификатор и получить порядковый номер блока. Требуемое значение передают клавиши со стрелками, поэтому нам не нужно ничего делать так как функция moveHighlight () уже установила currentValueSelected.

Далее мы последовательно проходим все дескрипторы span и устанавливаем их класс CSS равным spanNormalElement. В результате их внешний вид соответствует невыбранной позиции. Завершив проход, мы добавляем класс CSS к выбранной опции. Таким образом, две созданные выше функции позволяют пользователю выделять нужную позицию с помощью мыши или клавиатуры. Теперь нам осталось взять это выбранное значение и добавить его в текстовое окно.

Установка выбранного значения

Мы разрабатываем сценарий, который позволил бы пользователям, выбирая из предложенных вариантов, уменьшать объем работы, требуемой для заполнения поля формы. Для этого нам нужно взять номер выбранного пользователем элемента и установить текст в текстовом окне и значение в скрытом текстовом поле. Чтобы поведение нашего списка вариантов было похоже на поведение элемента HTML select, мы разработали три функции, приведенные в листинге 10.18.

Листинг 10.18. Обработка нажатия клавиш со стрелками и щелчков мышью

```
function SetText(xVal){
  theTextBox.value = arrOptions[xVal][0]; //set text value
  theTextBox.obj .hidden, value •» arrOptions [xVal] [1] ;
  document.getElementByldf"spanOutput").style.display = "none";
  currentValueSelected = -1; //remove the selected index
}
function GrabHighlighted(){
  if(currentValueSelected >= 0){
    xVal = document.getElementByldf"OptionsList_" +
      currentValueSelected).getAttribute("theArrayNumber");
    SetTextfxVal);
    HideTheBoxO ;
  }
}
function HideTheBoxf){
  document.getElementByldf"spanOutput").style.display = "none";
  currentValueSelected = -1;
  EraseTimeout();
```

Функция GrabHighlighted() позволяет получить текст и значение выбранной позиции. Итак, вначале нам нужно проверить, выбрал ли пользователь значение. Если да, тогда мы получаем порядковый номер массива arrOptions, в котором располагается текст. Для этого мы берем установленное ранее значение атрибута theArrayNumber, затем вызываем функцию SetText(), устанавливающую текст значения выбранной опции в соответствующие элементы формы.

Функция SetText() использует порядковый номер, переданный ей как параметр для индексирования массива arrOptions. Видимый пользователю текст устанавливается путем индексирования первого индекса массива. Скрытый элемент формы получает номер второго элемента, записанного в массиве. После того как мы извлечем значения, мы удалим список вариантов с экрана, вызвав функцию HideTheBox().

Функция HideTheBox() позволяет удалять блок spanOutput из поля зрения. Для этого мы обращаемся к блоку и устанавливаем его свойство style.display равным none. Для удаления выбранного индекса присваиваем переменной currentValueSelected значение -1. Все запущенные таймеры удаляются посредством функции EraseTimeout(), разработкой которой мы займемся в следующем разделе.

Использование таймеров JavaScript

Этот раздел является последним, в которым фигурирует JavaScript. Написав его, мы завершим создание приложения опережающего ввода, и вы сможете немного отдохнуть от всего этого кода. Итак, таймер, приведенный в листинге 10.19, является еще одним элементом, делающим интерфейс пользователя немного богаче. Он используется для того, чтобы укрыть блок выбора после определенного периода бездействия. В данном случае мы используем метод JavaScript setTimeout(), выполняющий выражение по прошествии определенного времени. Это время задается в миллисекундах и привязывается к объекту, созданному в листинге 10.6. Отметим, что функция setTimeout() будет вызвана только в том случае, если присвоить параметру объекта useTimeout значение true.

Листинг 10.19. Присоединение и удаление запланированных событий

```
function EraseTimeout(){
  clearTimeout(isTiming); isTiming • false;
}
function StartTimeout{){
  isTiming = setTimeout("HideTheBox()",
  theTextBox.obj.theVisibleTime);
```

Функция StartTimeout() устанавливает таймер при выполнении функции. Для инициализации таймера мы присваиваем переменной isTiming значение setTimeout. Метод setTimeout должен вызвать функцию HideTheBox() по прошествии установленного времени, которое задается с помощью `theVisibleTime`.

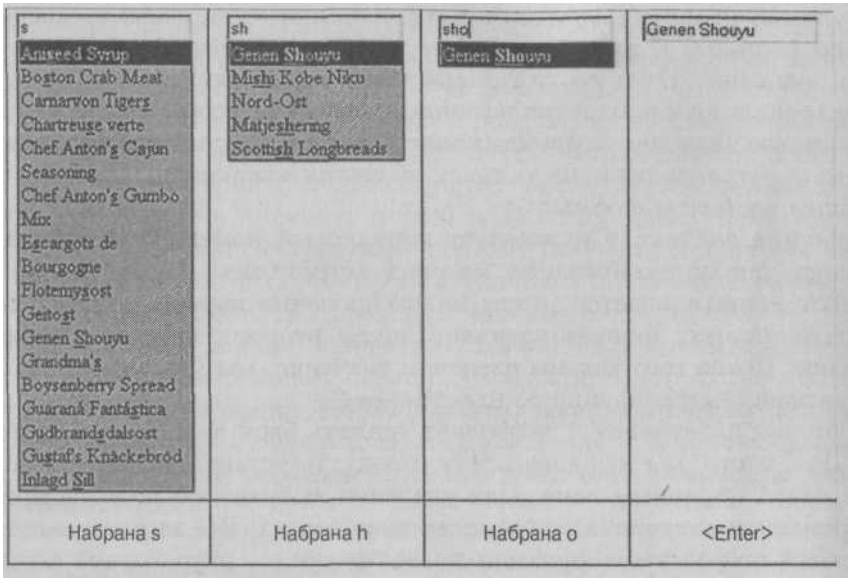


Рис. 10.5. Ход проекта опережающего ввода

Единственное, что мы еще должны сделать, — это удалить блокировку по времени. Для этого мы создаем функцию EraseTimeout (), использующую встроенную функцию JavaScript clearTimeout () и предотвращающую срабатывание HideTheBox(). Значение булевой переменной isTiming устанавливается равным false.

Написав последнюю строку кода, мы можем запускать проект опережающего ввода! Запишите проект, откройте его и начинайте набирать слово. В действии процесс выбора из предлагаемых вариантов выглядит так, как показано на рис. 10.5. При наборе первой буквы (s) число вариантов превышает 15. Вторая буква *(h)* уменьшает их число до пяти. Третья (о) сокращает список до одной позиции, которую мы выбираем, нажимая клавишу <Enter>. Добавив этот проект к любой форме, вы можете повысить эффективность работы пользователей, позволив им не набирать слова целиком.

10.4. Дополнительные возможности

Мы разрабатывали сценарий так, чтобы на странице можно было реализовать несколько элементов с опережающим вводом. Все, что нам для этого требуется, — добавить для каждого элемента объявления с новыми вызовами функции SetProperties (). Минусом данного метода является то, что для заполнения раскрывающихся списков различными значениями мы должны обращаться к нескольким страницам сервера. В большинстве случаев это не проблема, единственным отличием данных методов, скорее всего, будет выражение SQL.

Мы можем предложить более продуманное решение этой проблемы, введя в специальный объект дополнительный параметр и отправив его на сервер. Есть и другая возможность — работать с тем, что мы имеем, и внести в код минимальные изменения. В таком случае простое решение включает изменение одной строки кода и добавление оператора if в код серверной части сценария.

Итак, нам требуется каким-то образом научиться различать элементы на сервере, определяя, какой элемент потребовал дообработки. Сообщить о различии проще всего с помощью имени, содержащегося в элементе. В данном случае мы ссылаемся на имя текстового окна. Для внедрения описанной возможности мы изменили строку параметров так, как показано в листинге 10.20.

Листинг 10.20. Измененная функция TypeAhead()

```
function  TypeAhead(xStrText){
  var strParams = "q=" + xStrText + "&where=" +
    theTextBox.obj.matchAnywhere + "&name=" + theTextBox.name;
  var loaderl = new net.ContentLoader(theTextBox.obj.serverCode,
    BuildChoices,null,"POST",strParams);
}
```

Немного изменив переменную strParams в функции TypeAhead(), в параметрах формы, отправляемых на сервер, мы теперь передаем имя текстового окна. Это означает, что мы можем обращаться к этому значению на сервере и использовать оператор if-else либо case для запуска другого запроса. В таком случае нам не требуется реализовывать несколько страниц, соответствующих различным элементам.

10.5. Реструктуризация

Разработав достаточно сильный набор функций, обеспечивающих возможности опережающего ввода, мы можем подумать, как реструктуризировать все эти возможности в более удобном для использования виде. То, что мы создали на текущий момент, обеспечивает функциональные возможности, необходимые для предоставления на выбор ряда вариантов. Однако данная структура имеет свои недостатки с точки зрения работы, требуемой от разработчика при внедрении ее на Web-страницу (или на 20-30 Web-страниц).

Давайте на минуту представим, что перед нами, как перед главным архитектором использующей Ajax Web-структуры, поставлена задача написать компонент опережающего ввода, который будет использовать вся компания. После завершения вводного собрания мы получаем краткий перечень функциональных требований. Не совсем еще понимая суть дела, мы просматриваем этот документ (табл. 10.2).

По мере изучения списка нам в голову начинают приходить различные ᵣысли. Прежде всего, руководство, похоже, не понимает концепцию приоритета. Но поскольку этого стоило ожидать, давайте рассмотрим суть — требо-

Таблица 10.2. Наши функциональные требования

Номер	Описание требования	Приоритет
1	Компонент должен работать с существующей разметкой HTML, не требуя никакого ее изменения. Допускается лишь простая модификация заголовка с целью внедрения линии поведения компонента	1
2	Компонент должен без дополнительных усилий поддерживать многократное использование на одной странице	1
3	Должна существовать возможность индивидуальной настройки каждого экземпляра компонента Под этим имеются в виду как аспекты поведения (например, учет регистра, поиск с любого места), так и стилевое оформление CSS	1
4	Компонент не должен вводить глобальные переменные. Компания пользуется сторонними библиотеками JavaScript, и глобальное пространство имен уже достаточно засорено. Использование любых глобальных имен (кроме названия самого компонента) строго запрещено	1
5	Компонент должен предоставлять разумные значения по умолчанию для всех конфигурационных опций	1
6	Компонент должен работать в Internet Explorer и Firefox	1
7	Для уменьшения работ по кодированию, требуемых для улучшения качества и надежности решения, компонент должен быть создан на основе структуры с открытым исходным кодом	1
8	Кстати, если получится, сделайте это до конца недели	1

вания. Несмотря на то что мы проделали немалую работу, имеющийся сценарий не удовлетворяет даже половине из них. Сценарий уже готов, поэтому о требовании 7 можно забыть: нам не нужно уменьшать объем работ. Очевидно, что требованию 8 сценарий также удовлетворяет (по той же причине). Он поддерживает различные браузеры, поэтому снимаем и требование 6. А вот что касается остального, то определенную работу проделать все же придется. У нас есть всего неделя, поэтому начнем.

10.5.1. День 1: план разработки компонента TextSuggest

Прежде всего нам нужно определиться с тем, как поднять производительность и уложиться в отведенное время. Один из лучших способов — переложить работу на других- Если кто-то может сделать часть работы, пусть он ее для нас сделает. В данном случае мы собираемся воспользоваться библиотекой с открытым исходным кодом Rico (http: //openrico. org) и расширением Prototype.js (http://prototype.conio.net/). Библиотека Rico предлагает некоторую инфраструктуру Ajax, эффекты и вспомогательные методы, которые повысят скорость нашей разработки. Prototype предлагает инфраструктуру Для прекрасных синтаксических идиом, благодаря которым наш код будет выглядеть понятнее и потребует меньше времени на разработку. Поэтому давайте внимательно изучим последствия использования Prototype и Rico.

prototype

prototype предлагает разработчикам несколько расширений основного объекта JavaScript, а также несколько функций, способствующих поддержанию хорошего стиля кодирования. Ряд из них мы используем в нашем примере.

Объект **Class**

Объект Class, представленный в библиотеке Prototype, имеет единственный метод create(), отвечающий за создание экземпляров, которые могут иметь любое число методов. Метод create{) возвращает функцию, вызывающую другой метод того же объекта — initialize(). Звучит немного сложно, но на практике все просто По сути, таким образом формируется синтаксическая основа для задания типов в JavaScript. Идиома выглядит следующим образом:

```
var TextSuggest = Class.create();
TextSuggest.prototype = {
// Вызывается в процессе создания
  initialize: function( pi, p2, p3 ) {
  h

};
```

В данном фрагменте кода создается то, что можно считать "классом" (хотя сам язык не поддерживает такой концепции), и определяется функция-конструктор initialize(). Клиент компонента может создавать экземпляры класса с помощью приведенной ниже строки кода.

```
var textSuggest = new TextSuggest(pi, p2, p3);
```

Метод **extend O**

Библиотека Prototype расширяет базовый объект JavaScript и добавляет метод, именуемый extend(), открывая этот метод для всех объектов. Метод extend() принимает в качестве параметров два объекта: базовый объект и объект, который будет его расширять. Свойства расширяющего объекта переносятся на базовый объект. Это позволяет использовать механизм расширения объектов на уровне экземпляров. Этой возможностью мы воспользуемся позже, когда будем реализовывать для настраиваемых параметров компонента TextSuggest значения по умолчанию.

Метод **bind/bindAsEventListener()**

Библиотека Prototype также добавляет два метода к объекту Function: bind() и bindAsEventListener(). Эти методы предлагают синтаксически элегантный способ создания замыкания функций. Напомним, как мы создавали замыкания в других примерах:

```
oThis • this;
this.onclick = function() { oThis.callSomeMethod() };
```

С помощью метода bind() библиотеки Prototype того же результата можно добиться гораздо проще.

```
this.onclick = this . callSomeMethod.binci(this);
```

API bindAsEventHandler() передает методу объект Event и сглаживает различия между Internet Explorer и стандартизованной W3C моделью событий!

Метод $ — синтаксическая "конфетка"

В JavaScript в названиях методов можно использовать определенные специальные символы, например знак доллара ($). В библиотеке Prototype этот малоизвестный факт используется для инкапсуляции в метод одной из наиболее распространенных задач в DHTML-программировании: извлечения элемента из документа на основе его идентификатора. Таким образом, в нашем коде мы сможем писать конструкции следующего типа:

```
$('textField').value = aNewValue;
```

При этом мы обходимся без указанных ниже громоздких структур.

```
var textField = document.getElementByIdf'textField' )
textField.value = aNewValue;
```

Rico

Используя Rico, мы получаем Prototype бесплатно. Посмотрим, что нам требуется от Rico. Rico предлагает богатый набор линий поведения, возможностей перетаскивания и кинематических эффектов, но поскольку мы пишем компонент, использующий единственное текстовое поле, то большинство из доступных возможностей нам не понадобится. Однако еще есть прекрасный обработчик Ajax и некоторые вспомогательные методы, предлагаемые Rico. Вспомогательные методы Rico мы рассмотрим по ходу разбора примера, а сейчас остановимся на предлагаемой Rico инфраструктуре Ajax. Возможности Ajax Rico публикуются посредством единственного объекта, доступного для документа ajaxEngine. API ajaxEngine предоставляет поддержку для регистрации логических имен для запросов и регистрации объектов, знающих, как обрабатывать ответы Ajax. Рассмотрим, например, следующий код:

```
ajaxEngine.registerRequest( 'getlnvoiceData',
                            'someLongGnarlyUrl.do' );
ajaxEngine.registerAjaxObject( 'xyz', someObject );
```

В первой строке кода регистрируется логическое имя потенциально громоздкого URL Ajax. Далее при отправке запросов можно использовать это логическое имя, не отслеживая упомянутый громоздкий URL. Пример такого использования приведен ниже.

```
ajaxEngine.sendRequest('getlnvoiceData', request parameters ... );
```

Метод registerRequest () локализует использование указателей URL — теперь они встречаются только в одном месте, обычно это обработчик событий onload в разделе тела. Если URL требуется изменить, это можно сделать в месте регистрации, не затрагивая остальную часть кода-

Метод registerAjaxObject() иллюстрирует регистрацию объекта обработки Ajax. В предыдущем примере подразумевалось, что в ответах необходимо обращаться к объектной ссылке someObject с помощью логического имени xyz, причем эта ссылка необходима для обработки ответов Ajax посредством метода ajaxUpdate ().

Исходя из того, что мы используем описанные функциональные возможности объекта ajaxEngine, нам осталось только рассмотреть ответный XML-

документ, ожидаемый процессором Ajax. Этот документ немного отличается от динамически генерируемого сценария JavaScript, который возвращался в предыдущей версии данного примера, но Rico ожидает получить XML. Все элементы <response> документа должны находиться внутри элемента верхнего уровня <ajax-response>. Внутри указанного элемента сервер может возвращать столько элементов <response>, сколько требует приложение. Такая возможность очень удобна, поскольку позволяет серверу возвращать ответы, обрабатываемые различными объектами, обновляющими потенциально несвязанные области Web-страницы, — например, для обновления области состояния, области данных и конечной области вывода. XML-документ для предыдущего примера приведен ниже.

```
<ajax-response>
  <response type="object" id="xyz">
    ... the rest of the XML response as normal ...
  </response>
  <response...>
    more response elements if needed..
  </response>
</ajax-response>
```

Данный XML-документ указывает ajaxEngine, что данный запрос должен обработать объект, зарегистрированный с идентификатором xyz. Процессор Ajax находит объект, зарегистрированный с именем xyz, и передает содержимое соответствующего элемента <response> методу ajaxUpdate().

Вот и все. День получился довольно коротким. Некоторое время мы потратили на изучение структур с открытым исходным кодом, на которые мы можем опираться, и разработали план внедрения их в требуемый компонент. Мы еще не написали ни строчки кода, но решили, как ускорить дальнейшую работу над проектом. Кроме того, мы подобрали платформу, которая поможет повысить производительность работы (удовлетворив при этом требованию 7 технического задания). Кодирование начнется завтра.

10.5.2. День 2: создание TextSuggest — понятного и настраиваемого компонента

Теперь, когда у нас есть хорошая технологическая платформа, мы можем создавать на ней свой компонент. При работе над проектами часто удобно идти от желаемого результата, заранее думая о контракте нашего компонента. Напомним наше первое требование.

> Требование 1. Компонент должен работать с существующей разметкой HTML, не требуя никакого ее изменения. Допускается лишь простая модификация заголовка для внедрения линии поведения компонента.

Из-за этого требования мы оставляем нетронутым практически все, что находится внутри элемента <body>. Поэтому предположим, что нам требуется ввести сценарий в HTML посредством HTML-кода, подобного приведенному в листинге 10.21.

Листинг 10.21. HTML-разметка компонента TextSuggest

```
<html>
<head>
  <script>
    var suggestOptions = { /*details to come*/ };
    function injectSuggestBehavior() { // Создать компонент в <head>
      suggest = new TextSuggest(
          'fieldl', 'typeAheadData.aspx', suggestOptions );
      } >;
  </script>
</head>
<body onload="injectSuggestBehavior()">
  <form name="Forml">
    AutoComplete Text Box:
    <input type="text"  id—"fieldl" name="txtUserInput"> _-
  </form>
</body>
</html>
```

Смысл приведенного HTML-кода заключается в том, что нам требуется создать объект с идентификатором текстового поля, к которому мы будем привязаны, указателем URL источника данных Ajax и набором конфигурационных объектов, которые еще будут заданы. (Чтобы это сработало, текстовое поле должно иметь идентификатор.) Все, что находится внутри элемента <body>, остается без изменений. Разобравшись с этим, займемся конструктором. Имя создаваемого компонента TextSuggest помещается в глобальное пространство имен с помощью функции-конструктора, которая (как вы помните) генерируется методом Clas s. create {) библиотеки Prototype, как показано в листинге 10.22.

Листинг 10.22. Конструктор TextSuggest

```
TextSuggest = Class.create();
TextSuggest.prototype = {  initialize:
 function{anld, url,
options)  {//О Ссылка на входной элемент
    this.id = anld;
    this.textlnput = $(this.id);
    var browser = navigator.userAgent.toLowerCase();
// © Детектировать тип браузера
    this.isIE =
      browser.indexOff"msie") != -1;
    this-isOpera =
      browser.indexOf{"opera")!= -1;
// © Установить значения по умолчанию
      this.suggestions = [];
      this.setOptions(options);
      this.initAjax(url);
      this.injectSuggestBehavior();
  Ь
```

Разберем этот конструктор. Как упоминалось ранее, конструктору передается идентификатор текстового ввода, с которым нужно связать предложение вариантов. Для поля ввода хранится ссылка как на идентификатор, так л на элемент DOM О. Далее определяется браузер пользователя и записывается состояние, которое понадобится компоненту позже, когда потребуется информация о среде времени выполнения этого браузера ©. В данном случае специальный код требуется только для Internet Explorer и Opera, поэтому проверяется использование только этих браузеров.

Настройка Ajax и введение в код нужной линии поведения будет рассмотрен позже ©. Пока же (до конца дня) сосредоточимся на возможности настройки компонентов. Как вы помните, ранее была создана функция Setproperties(), содержащая все настраиваемые аспекты нашего сценария.

```
function SetProperties
            (xElem, xHidden, xserverCode,
            xignoreCase, xmatchAnywhere,
            xmatchTextBoxWidth, xshowNoMatchMessage,
            xnoMatchingDataMessage, xuseTimeout,
            xtheVisibleTime){

}
```

Написанный код удовлетворяет требованию, которое касается возможности настройки, но не обеспечивает удобный API или подходящие значения по умолчанию. Для решения этих проблем вводится объект опций, который передается конструктору. Объект опций содержит свойство для каждого конфигурационного параметра компонента опережающего ввода. Теперь нужно заполнить опции конфигурационными параметрами.

```
var suggestOptiona - { matchAnywhere : true, ignoreCase : true };
function injectSuggestBehavior() {
suggest = new TextSuggest('fieldl', 'typeAheadXML.aspx',
     suggestOptions ); } );
```

Данная простая идиома предлагается следующие дополнительные преимущества.

- Благодаря ей сигнатура конструктора выглядит понятнее. Сигнатура клиентских страниц, на которых используется наш компонент, может создаваться всего с тремя параметрами.
- Можно вводить дополнительные конфигурационные параметры, не меняя контракт конструктора.
- Можно написать изящную функцию setoptions(), предоставляющую значения по умолчанию для всех незаданных свойств, и позволить пользователю задавать только те свойства, которые он желает изменить.

В последнем пункте описано именно то, что делает метод setOptions(), включенный в конструктор ранее ©. Разберем его.

```
setOptions: function(options) {
  this,options • {
    suggestDivClassName: 'suggestDiv',
```

```
      suggestionClassName:   'suggestion',
      matchClassName  :  'match',
      matchTextWidth  :  true,
      selectionColor  :   '#ЫсО9С,
      matchAnywhere  :  false,
      ignoreCase  :  false,
      count  :  10
    }.extend(options  |I  ({});
  Ь
```

В данном коде задаются все свойства объекта опций, имеющие подходящие значения по умолчанию. Затем вызывается метод extend() библиотека Prototype, который переопределяет свойства, переданные во время создания. В результате получается смешанный объект опций, содержащий и значения по умолчанию, и переопределенные значения, заданные в одном-объекте! В использованном примере были переопределены булевы свойства matchAnywhere и ignoreCase (новые значения — true). Значения конфигурационных свойств объясняются в табл. 10.3.

Обратите внимание на наличие нескольких опций, задающих, какие имена классов CSS должны генерироваться внутренне при создании структуры HTML для всплывающего списка предположений. Напомним технические требования, которые приводились в табл. 10.2.

> Требование 3. Должна существовать возможность индивидуальной настройки каждого экземпляра компонента. Под этим имеются в виду как аспекты поведения (например, учет регистра, поиск с любого места), так и стилевое оформление CSS.
>
> Требование 5. Компонент должен предоставлять разумные значения по умолчанию для всех конфигурационных опций.

В готовом компоненте описанный механизм конфигураций будет использоваться для настройки на уровне экземпляров поведения (например, чув-J ствительности к регистру) и стилевого оформления (например, имен клас-! сов CSS).

Таким образом, к концу второго дня работы над компонентом мы взяли хороший старт. Был получен нормальный конструктор, создана основа для настройки компонента, а теперь пора включить в наш проект принципы Ajax.

10.5.3. День 3: включаем Ajax

Ну что, подключим к работе Ajax? Без Ajax компонент TextSuggest подобен гамбургеру без говядины. Мы не относимся неуважительно к вегетарианцам, но сейчас пришло время говядины. Вы уже видели подготовку к включению Ajax при разборе конструктора. Как вы помните, в конструктор был включен метод initAjaxf), который выполняет настройку, требуемую для обсуждавшейся выше поддержки Ajax. Реализация данного метода приведена ниже.

```
initAjax:  function(url)  {
  ajaxEngine.registerRequest(  this.id  +  '_request',  url  );
  ajaxEngine.registerAjaxObject{  this.id  +  '_updater',  this  );},
```

Таблица 10.3. Значения конфигурационных свойств

Значение	Объяснение
ggestDivClassName	Задает имя класса CSS для элемента div, в котором будут содержаться предлагаемые варианты
ggestionClassName	Задает имя класса CSS для элемента span, генерируемого для каждого предлагаемого варианта
matchClassName	Задает имя класса CSS элемента span, содержащего фрагмент предлагаемого варианта, согласующийся с вводом пользователя
inatchTextwidth	Булево значение, указывающее, должен ли размер элемента div, генерируемого для списка предположений, согласовываться с шириной текстового поля, с которым он соотнесен
selectionColor	Задает шестнадцатеричное значение (или любое другое приемлемое значение, используемое CSS для задания цвета), определяющее цвет фона списка предлагаемых вариантов
matchAnywhere	Булево значение, которое задает, с какого места строки должно выполняться сопоставление, — с начала или с любой позиции
ignoreCase	Булево значение, указывающее, должен ли учитываться регистр при поиске соответствий
count	Максимальное число показываемых предположений

Напомним, что метод registerRequest () предоставляет аппарату Ajax логическое имя указателей URL, которые будут вызываться для данного запроса посредством метода sendRequest{). При условии, что нам требуется поддержка нескольких компонентов опережающего ввода на странице, имеющих различные указатели URL (но использующие единственный элемент ajaxEngine), необходимо сгенерировать для каждого из них уникальное логическое имя. Таким образом, имя для запроса мы генерируем, основываясь на идентификаторе компонента, который считается уникальным. Тот же принцип применяется при обработке регистрации. Мы регистрируем this как объект, обрабатывающий ответные сообщения, направленные к сгенерированному нами идентификатору.

Пожалуй, стоит проиллюстрировать сказанное на примере. Предположим, что мы привязали предложение вариантов к полю с идентификатором id='fieldl', а затем эффективно зарегистрировали себя как 'fieldl_updater'. XML-документ, возвращаемый этому компоненту, должен содержать элемент, выглядящий примерно так, как показано ниже.

```
<ajax-response>
  <response type='object' id=  'fieldl^updater' >.
    ... тот же xml-текст, что и ранее.
  </response>
</ajax-response>
```

Во внутреннем представлении запросы отправляются следующим образом; *Щ*

```
ajaxEngine.sendRequest{
  'fieldl^request',1param1=vall1, 'param2=val2', . . .
);
```

Учитывая вышесказанное, для получения Ajax-модификации нашего компонента необходимо отправить запрос и обработать ответный документ. Посмотрим, как это можно сделать.

Предлагаемый текст — отправка запроса Ajax

Разумеется, до отправки запроса придется сделать кое-что еще. Текстовый ввод должен сгенерировать событие onchange, которое мы будем ждать, и при выполнении определенных условий отправлять запрос о предполагаемых вариантах. Пока что написанные нами фрагменты кода находятся не на своих местах, но мы это поправим. Мы по-прежнему можем оперировать терминами контрактов и ответственности метода, которые желательно независимо реализовать в конечном продукте. Поэтому предположим, что некоторый фрагмент кода (который мы еще напишем) решает, что необходимо отправить запрос и получить ряд предполагаемых вариантов. Назовем эту функцию sendRequestForSuggestions () и реализуем ее следующим образом:

```
sendRequestForSuggestions; function() {
  if ( this.handlingRequest ) {
    this.pendingRequest = true; return;
  }
  this.handlingRequest = true;
  this.callRicoAjaxEngine();
Ь
```

Единственное, что делает приведенный код, — это вызывает th.is.callRicoAjaxEngine () при условии, что запрос не обработан. Этот простой механизм присваивает внутреннему булеву свойству this.handlingRequest значение true после создания запроса Ajax и false (рассмотрено ниже) — после обработки запроса. Если метод вызывается в момент обработки запроса, булеву свойству this.pendingRequest присваивается значение true. Это состояние сообщает обработчику, что, возможно, ему придется отправить еще один запрос после обработки текущего. Далее мы рассмотрим метод callRicoAjaxEngine {), приведенный в листинге 10.23.

Листинг 10.23. Использование a j axEngine (Rico)

```
callRicoAjaxEngine: function() {
// Построить массив параметров
  var callParms = [];
  callParms.push( this.id + '^request');
  callParms.push{ "id=" + this.id);
  callParms.push( 'count=' + this.options.count);
  callParms-push( 'query=' + this.lastRequestString);
  callParms.push( 'match_anywhere=1 + this.options.matchAnywhere);
  callParms.push( 'ignore_case=' + this.options.ignoreCase);
  var additionalParms = this.options.requestParameters t{ [];
  for( var i=0 ; i < additionalParms.length ; i++ )
    callParms.push{additionalParms[i]);
```

```
// Отправить запрос Ajax
  ajaxEngine.sendRequest.apply( ajaxEngine, callParms );
Ь
```

Чтобы понять, что же делает данный метод, вначале необходимо рассмотреть механизм JavaScript, используемый в последней строке метода.

ajaxEngine.sendRequest.apply(ajaxEngine, callParms);

Здесь применяется метод applyO, доступный для всех функциональных объектов (подробности — в приложении Б). Проиллюстрируем его использование на простом примере.

```
Greeter,prototype.greetPeople = function(strl, str2)
{
  alert('hello ' + strl + 'and ' + str2)
};
```

Предположим, что имеется экземпляр Greeter, именуемый friendlyperson, и в качестве аргумента ему нужно передать метод greetPeople{). Однако у нас нет параметров удобного для передачи вида, а имеется массив элементов people. В подобном случае выгодно использовать метод apply. Соответствующий код можно записать следующим образом:

```
var people = [ "Joe", "Sally" ];
friendlyPerson.greetPeople.apply( friendlyPerson, people);
```

Метод apply() преобразовывает массив, переданный как второй аргумент, в параметры метода первого класса и вызывает метод на объекте, переданном как первый параметр. Таким образом, приведенный код эквивалентен строчке

```
friendlyPerson.greetPeople( people[0], people[1]);
```

Вернемся к нашей задаче. Нам требуется вызвать метод sendRequest() процессора ajaxEngine, принимающий в качестве первого параметра логическое имя запроса плюс переменное число строковых параметров вида key=value, представляющих параметры запроса. Здесь нужно быть очень аккуратными, поскольку мы запрашиваем параметры из различных источников и не знаем, сколько этих параметров. Рассмотрим код еще раз.

```
// О
var callParms - [];
callParms.push( this.id + '^request');

callParms.push( 'ignore_case=' + this.options.ignoreCase);
// ©
var additionalParms = this.options.requestParameters || [];
for( var i=0 ; i < additionalParms.length ; i++ )
  callParms.push(additionalParms[i]) ;
```

Массив параметров, отправляемый методу sendRequest() посредством apply, заселяется смешанной информацией: внутренним состоянием объекта, такими элементами, как идентификаторы и lastRequeststring, а также определенными свойствами объекта options (например, count, matchAnywhere, ignoreCase) О.

Тем не менее необходимо еще обеспечить механизм, позволяющий пользователю компонента передавать помимо указанной информации внешние параметры ©. Для этого мы проверяем существование свойства объекта options, именуемого requestParameters. Если его значение не равно null, оно предполагается равным массиву строк вида key=value. Этот массив последовательно проходится и добавляется к массиву callParms, уже заполненному параметрами компонента. Наконец, запрос отправляется.

```
ajaxEngine.sendRequest.apply{ ajaxEngine, callParms);
```

Все! С отправкой запроса мы разобрались. Теперь будем надеяться, что сервер находится в рабочем состоянии, получим ответный документ. Давайте разбираться, как мы будем обрабатывать ответ, когда он к нам поступит.

Предлагаемый текст — обработка ответа Ajax

Если мы собираемся обеспечить надежную отправку запросов, нам придется много поработать, поэтому лучше убедиться в надлежащей обработке ответного документа, иначе весь тяжкий труд будет напрасным. Напомним, что механизм ajaxEngine (Rico) направляет запрос к отправившему его методу ajaxUpdatet), передавая содержимое элемента <response>. Следовательно, необходимо написать метод a j axUpdate (), который будет точкой входа в процедуру обработки запроса. Метод ajaxUpdateO показан в листинге 10.24 вместе со своими вспомогательными методами разбора createSuggestionsO и getElementContent().

Листинг 10.24. Обработка ответа Ajax j M H H B H H H H H H B H ^ I

```
ajaxUpdate: function( ajaxResponse ) {
// Создать предположительные варианты
  this.createSuggestions( ajaxResponse );
  if ( this.suggestions.length == 0 ) {
    this.hideSuggestions{);
    S( this.id + "_hidden").value = "";
  }
// Создать и показать пользовательский интерфейс
  else {
    this.updateSuggestionsDiv{);
    this.showSuggestions();
    this.updateSelection(0);
  }
// Завершить обработку ответа
  this.handlingRequest = false;
  if ( this.pendingRequest ) {
    this.pendingRequest = false;
    this-lastRequestString = this.textlnput.value;
// Отправить другой запрос
    this.sendRequestForSuggestions();
  }
},
createSuggestions: function(ajaxResponse) {
  this.suggestions * [];
  var entries ~ ajaxResponse.getElementsByTagNamef'entry');
  for ( var i = 0 ; i < entries.length ; i++ ) {
```

```
      var strText = this.getElementContent(
                    entries[i1.getElementsByTagName('text') [0] );
      var strValue = this.getElementContent{
                      entries[i].getElementsByTagName('value')[0]);
      this.suggestions.push({ text: strText, value: strValue });

    }
  ),
  getElementContent: function(element) {
    return element.firstChild.data;
```

_ J

Поскольку мы собираемся сконцентрироваться исключительно на реализации в нужных местах механизмов Ajax, многие элементы будут рассмотрены на высоком уровне, а обработка ответа описана алгоритмически. Итак, сначала необходимо разобрать отклик посредством метода createSuggestions(), преобразовав его во внутреннее представление предлагаемых вариантов, содержащихся в свойстве suggestions. Это свойство представляет собой массив объектов, содержащих текст и значение (соответствуют элементам <text> и <value> каждого элемента <entry> XML-документа).

Остальная часть алгоритма, метод ajaxUpdate(), достаточно очевидна и понятна. Если предлагаемые варианты не найдены, всплывающее окно скрывается, а внутреннее значение, хранимое компонентом посредством скрытого поля, обнуляется. Если предполагаемые варианты найдены, создается раскрывающийся пользовательский интерфейс, заполненный предложениями, затем он отображается на экране с первой выделенной позицией. На этом этапе запрос считается обработанным, поэтому значение рассмотренного ранее свойства this.handlingRequest устанавливается равным false. Наконец, метод ajaxUpdatet) проверяет, имеются ли незавершенные запросы. Если да, значение метки pendingRequest устанавливается равным false, lastRequestString присваивается текущее значение поля ввода и посредством sendRequestForSuggestions() инициируется следующий цикл запроса.

Таким образом, к концу третьего дня мы получили полный цикл запроса/ответа с поддержкой Ajax. За сегодняшний день сделано довольно много: мы разобрались в структуре с открытым исходным кодом, которая позволила полностью "активизировать Ajax" (требование 7), а также гарантировали возможность настройки и поддержки нескольких экземпляров на одной странице (требования 2 и 3). Собственно созданием, размещением, показом и сокрытием пользовательского интерфейса мы займемся на пятый день. Пока же рассмотрим события компонента и подготовим обработку действий с клавиатурой и мышью.

10.5.4. День 4: обработка событий

Теперь, активизировав Ajax в нашем компоненте опережающего ввода, можно разбираться с событиями, порождаемыми реакцией поля на ввод с клавиатуры. Внимательный читатель уже догадался, что код, запускающий этот процесс, уже предусматривался в конструкторе, будучи спрятанным в вызове

метода injectSuggestBehavior{). Данный код инициирует все модификации DOM существующей разметки, включая обработку событий, дополнительный ввод, и контейнер для предположений. Все эти модификации програь*: мируются, HTML-код страницы затрагиваться не будет (см. требование 1) Введение требуемого поведения показано в листинге 10.25.

Листинг 10.25. Введение требуемого поведения

```
injectSuggestBehavior: function() {
  if ( this.isIE ) {
// Убрать взаимодействие с Internet Explorer
    this.textlnput,autocomplete="off";
  }
  var JceyEventHandler =
    new TextSuggestKeyHandler(this);
// Создать контроллер
  new Insertion.After( this.textlnput,
                       '<input type="text" id-"' + this.id +
                       '_preventtsubmit'+
                       '" style="display:none"/>' );
  new Insertion.After( this.textlnput,
                       '<input type="hidden" name="'+
                      this.id+'_hidden'+
                      '"Tr id='"+this.id+'_hidden'+"'/>' );
  this.createSuggestionsDivf);
// Создать пользовательский интерфейс
},
```

Данный метод вначале проверяет, не используется ли Internet Explorer. Если используется, то собственному свойству autocomplete присваивается значение off. Благодаря этому встроенные возможности автоматического заполнения не взаимодействуют с нашим всплывающим окном. Далее создается объект TextSuggestKeyHandler, который будет перенаправлять события нужным методам. Действительно, механика событий достаточно запутана, поэтому мы выделяем поведение в отдельный объект, который рассмотрим чуть позже. Следующий метод вводит несколько входных элементов в разметку. Напомним, что в предыдущем цикле разработки кода мы добавили скрытое поле ввода для хранения значения компонента и невидимое текстовое поле, препятствующее отправке формы при нажатии клавиши <Enter>. Поскольку первое требование технической задачи запрещает вмешиваться в HTML-код, вся "грязная работа" выполняется программно — с помощью двух вызовов Insertion.After(). Возможность использования Insertion.After() появилась благодаря библиотеке Prototype. Наконец, вызывается функция createSuggestionsDiv(), создающая элемент div, вмещающий пользовательский интерфейс с предлагаемыми вариантами.

Объект TextSuggestKeyHandler

Как уже говорилось выше, распределение событий мы решили вынести в отдельный класс контроллера. В таком решении нет ничего нового или необычного, просто так удобнее разделить ответственность класса. В реальной жиз-

01 можно еще отделить все компоненты дизайна, создав явные классы модели
1(представления и обеспечив полный шаблон MVC Это предлагается сделать читателю в качестве самостоятельного упражнения, но в главе 13 будет показано, как разбить архитектуру приложения чтения RSS-сообщений на набор классов, согласующихся с традиционным шаблоном MVC.

Контроллер создается точно так же, как и основной класс, — с использованием Class.create() и метода initialize(). Конструктор TextSuggestKeyHandler показан в листинге 10.26.

Листинг 10.26. Конструктор TextSuggestKeyHandler

```
TextSuggestKeyHandler = Class.create();
TextSuggestKeyHandler.prototype = {
  initialize: function( textSuggest ) {
    this.textSuggest = textSuggest;
    this.input = this.textSuggest.textlnput;
// Ссылка на TextSuggest
    this.addKeyHandling();
  h
  // остальная часть API
_ Ь
```

В процессе создания контроллер использует ссылку на компонент опережающего ввода, а также "родное" поле ввода HTML-формы. Затем с помощью this.addKeyHandlingO он добавляет в поле ввода обработчики событий. Метод addKeyHandling() показан в листинге 10.27.

Листинг 10.27. Обработчик действий с клавиатурой

```
addKeyHandling: function() {
  this.input.onkeyup =
    this.keyupHandler.bindAsEventListener(this);
  this.input.onkeydown =
    this.keydownHandler.bindAsEventListener(this);
  this.input.onblur =
    this.onblurHandler.bindAsEventListener(this);
  if ( this.isOpera )
    this.input,onkeypress =
      this.keyupHandler.bindAsEventListener(this);
```

В данном методе устанавливаются все события, наступление которых требуется отслеживать, а также код для браузера Opera, упоминавшийся в первом цикле разработки данного сценария. Напомним, что метод bindAsEventListener ( ) представляет собой механизм замыкания, полученный благодаря библиотеке Prototype. Данный механизм позволяет нашим обработчикам вызывать методы первого класса на контроллере и сводит к общему знаменателю модели событий Internet Explorer и W3C. Очень хорошо. Методы keyupHandlerO, keydownHandler(), onblurHandler() и их вспомогательные методы представляют собой реструктуризированный вариант уже рассмотренного кода (с минимальными изменениями). Все эти методы будут описаны

ниже с указанием их отличий от первоначального сценария. Начнем с обсу>кдения метода keydownHandler(), показанного Блистинге 10.28, и того, как он обрабатывает выделение.

```
keydownHandler: function(e) {
  var upArrow = 38;
  var downArrow =40;
  if ( e.keyCode =- upArrow ) {
    this.textSuggest.moveSelectionUp();
    setTimeout( this.moveCaretToEnd.bind(this), 1 );
  }
  else if ( e.keyCode == downArrow ) {
    this.textSuggest.moveSelectionDown();
  }
```

—h

Наиболее существенные отличия от исходного сценария с точки зрения функциональных возможностей претерпела обработка клавиш со стрелками. В компоненте TextSuggest клавиши со стрелками управляют движением выделения, основываясь на событии onkeydown, а не на onkeyup. Это сделано исключительно для повышения практичности. Иногда пользователя сбивает с толку то, что выделение остается на своем месте при нажатии клавиши и начинает перемещаться после того, как клавиша отпущена. В общем, перемещение выделения обрабатывается методом keydownHandler (). Обратите внимание на то, что методы управления выделением являются методами компонента TextSuggest. Поскольку контроллер во время создания записал ссылку на компонент, он может вызывать эти методы посредством объектной ссылки this. textSuggest. Для полноты изложения мы приводим в листинге 10.29 методы управления визуальным выделением.

Листинг 10.29. Методы TextSuggest, отвечающие за визуальное выделение *п*

```
raoveSelectionUp: function{)   {
  if ( this.selectedlndex > 0 ) {
    this.updateSelection(this.selectedlndex  -  1);
  }
```

Ь

```
moveSelectionDown: function() {
  if ( this.selectedlndex < (this.suggestions.length - 1) ) {
    this.updateSelection<this.selectedlndex  +  1);
  }
```

Ь

```
updateSelection: function(n) {
  var span = $(this.id + "_"+this.selectedlndex);
  if (span){
    span.style.backgroundColor = "";
// Очистить предыдущее выделение
  }
  this.selectedlndex = n;
  var span = 5(this.id+"_"+this.selectedlndex);
  if (span){
```

```
    span.style.backgroundColor =
      this,options.selectionColor;
  }
```

Реальную работу по изменению визуального состояния выделенного элемента выполняет метод updateSelection(). Он обновляет элемент span, созданный в списке выбора (соответствующий код мы напишем на пятый день), и устанавливает его свойство style.backgroundColor равным значению, заданному как options.selectionColor в объекте Configuration нашего компонента.

Перед тем как закрыть тему обработки нажатия клавиш, следует учесть еще один момент. Поскольку при обработке действий с клавишами мы учитываем их нажатие, а не отпускание, необходимо изменить значение клавиши со стрелкой вверх с поведения по умолчанию на перемещение назад каретки в текстовом поле. Для этого применяется метод moveCaretToEndt), вызываемый с миллисекундной задержкой (реализована с помощью setTimeout). Реализация метода moveCaretToEndt) приведена в листинге 10.30.

Листинг 10.30. Метод moveCaretToEnd () компонента TextSuggest • ' 9 | Щ Н

```
moveCaretToEnd: function() {
  var поз = this.input.value.length;
  if (this.input.setSelectionRange) {
    this.input.setSelectionRange(pos,pos);
  }
  else if(this.input.createTextRange){
    var m = this.input.createTextRange{};
    m.moveStart('character', pos);
    ra.collapse();
    m.select();
  }
```

Ь

Рассмотрим теперь обработку отпускания клавиш. Реализация отпускания немного проще, чем реализация нажатия клавиш. Все, что требуется, — это перенаправить событие, учитывая текст в поле ввода и нажатую клавишу. Рассмотрим подробнее код, приведенный в листинге 10.31.

Листинг 10.31. Обработка в TextSuggest отпускания клавиш , у

```
keyupHandler: function(e) {
  if ( this.input.length == 0 && !this.isOpera )
    this.textSuggest.hideSuggestions();
  if ( 'this.handledSpecialKeys(e) )
    this.textSuggest.handleTextlnput() ;
```

Ь

```
handledSpecialKeys: function(e) {
  var enterKey = 13;
  var upArrow = 38;
  var downArrow • 40;
  if ( e.keyCode == upArrow II e.keyCode == downArrow ) {
```

```
    return true;
  }
  else if ( e.keyCode == enterKey ) {
    this.textSuggest.setInputFromSelection{);
    return true;
  }
  return false;
```

Обработчик отпускания клавиш вначале проверяет, содержится ли текст в поле ввода. Если нет, он сообщает компоненту TextSuggest скрыть всплывающий список предлагаемых вариантов. Далее обработчик проверяет, не была ли нажата одна из специальных клавиш: клавиша со стрелкой вверх, клавиша со стрелкой вниз или <Enter>. Если была нажата одна из клавиш со стрелками, метод не выполняет никаких действий, поскольку нажатие этой клавиши уже было обработано. Однако если была нажата клавиша <Enter>, метод сообщает компоненту TextSuggest установить входное значение на основе выбранной в текущий момент позиции списка предположений. Наконец, если поле ввода содержит значение и нажатая клавиша не является специальной, то обработчик сообщает компоненту TextSuggest о наличии некоторого входа, который нужно обработать с помощью метода textsuggcrest.handleTextInput(). Данный метод компонента TextSuggest в конечном счете активизирует структуру Ajax, которую мы успешно настраивали вчера. Код обработчика handleText!nput() представлен в листинге 10.32.

;• Листинг 10.32. Обработчик ввода текста "ШЯШШШИИВ;. >

```
handleTextInput: function{) {
  var previousRequest =
    this.lastRequestString;
// Предыдущее запрашиваемое значение
  this.lastRequestString =
    this.textInput.value;
// Текущее запрашиваемое значение
  if { this.lastRequestString == "" )
    this.hideSuggestions();
  else if ( this.lastRequestString != previousRequest ) {
    this.sendRequestForSuggestions();
// Запрос данных средствами Ajax
  }
```

Вначале метод handleTextlnputO устанавливает значение локальной переменной previousRequest равным предыдущему значению this.last-RequestString. Затем он устанавливает значение свойства lastRequest-String равным текущему содержимому поля ввода, так что теперь можно сравнить эти два значения и убедиться, что мы не пытаемся запросить [T]У информацию, которая у нас уже есть. Если запрос представляет собой пустую строку, всплывающий список будет скрыт. Если запрос отправлен для получения новой информации, метод handleTextlnput() вызывает ме-

тод sendRequestForSuggestions() (который мы написали вчера) для вызова основанного на Ajax источника данных и получения с сервера ряда предположений. Если запрос идентичен последнему посланному запросу, никакие действия не предпринимаются. Похоже, что наконец-то фрагменты начинает собираться в общую картину. Создание, настройка конфигурации, работа инфраструктуры Ajax, обработка событий — мы действуем так, как будто знали, что придется делать. Идем впритык — сейчас уже четвертый день работы!

Класс контроллера должен охватывать еще один класс — обработчик событий onblur, который представляет собой очень простой метод, устанавливающий значение текстового поля согласно текущему выделению и скрывающий предлагаемые варианты. Его реализация выглядит следующим образом:

```
onblurHandler: function(e) {
  if ( this.textSuggest.suggestionsDiv.style.display == " " )
    this.textSuggest.setInputFromSelection();
  this.textSuggest.hideSuggestions();
}
```

Обработчики onblurHandler и handledspecialKeys обращаются к методу компонента TextSuggest, который мы еще не рассматривали, — setInputFromSelection( ). По сути, этот метод делает то же, что делала ранее функция SetText() — принимает выделенное в текущий момент предположение; присваивает свой текст полю ввода, а значение — скрытому полю; скрывает список предположений. Реализация этого метода приведена ниже.

```
setInputFromSelection: function() {
  var hiddenInput « $( this.id + "_hidden" );
  var suggestion ~ this.suggestions[ this.selectedIndex ];
  this.textInput.value = suggestion.text;
// Обновить видимое значение
  hiddenInput.value = suggestion.value;
// Обновить скрытое значение
  this.hideSuggestions();
}
```

Возможно, для выполнения всего, что было запланировано на сегодня, пришлось немного поработать сверхурочно. Однако в результате мы создали класс controller, обрабатывающий все, что касается управления событиями. Для автоматического создания замыканий и унификации моделей событий Internet Explorer и W3C мы использовали метод bindAsEventListener() библиотеки Prototype. Мы реализовали обработчики нажатия/отпускания клавиш, скрывающие в себе все сложности обработки выделенной позиции и обычного текстового ввода. Мы гарантировали, что запросы будут отправляться только для получения новой информации; разобрались с показом и сокрытием в нужные моменты пользовательского интерфейса; программно обновили DOM для управления скрытым входным значением и невидимым текстовым полем, предотвращающим отправку формы при нажатии клавиши <Enter>. Кроме того, мы обработали обновление скрытого и видимого значений компонента TextSuggest. На пятый день мы доведем до ума и реструктуризируем наш компонент, реализовав все методы, требуемые для создания

всплывающего окна, размещения его в нужном месте, показа и сокрытия а также обработки событий, связанных с мышью. Неясный свет в конце тунеля становится все ближе и отчетливее.

10.5.5. День 5: *пользовательский интерфейс всплывающего окна с предлагаемыми вариантами*

Теперь, когда мы полностью вошли в работу, пришло время разобраться со всеми незавершенными моментами. Пока что мы создали инфраструктуру для обеспечения настроек и значений по умолчанию, обработку запросов и ответов Ajax, а также события, которые все это связывают. Остались вопросы графической реализации. Относительно пользовательского интерфейса необходимо решить следующие вопросы.

- Создание пользовательского интерфейса всплывающего окна с предлагаемыми вариантами. Это подразумевает создание элемента div для списка и элемента span для каждого варианта.
- Размещение всплывающего окна в нужном месте.
- Заполнение всплывающего окна предлагаемыми вариантами.
- Показ и сокрытие вариантов.

Создание всплывающего окна с предлагаемыми вариантами

Вернемся немного назад и рассмотрим реализацию метода injectSuggestBehavior(). Напомним, что данный код представлял собой точку входа для всех манипуляций DOM, производимых компонентом TextSuggest.

```
injectSuggestBehavior: function() {
  // Введение поведения DOM HTML ...
  this.createSuggestionsDiv();
}
```

В последней строке метода injectSuggestBehavior() вызывается метод createSuggestionsDiv(), создающий самый внешний контейнер div всплывающего окна с предлагаемыми вариантами. Поскольку это контейнер всех артефактов GUI, именно с него логично начать рассмотрение кода пользовательского интерфейса. Подробности его реализации показаны в листинге 10.33.

Листинг 10.33. Создание пользовательского интерфейса окна с вариантами

```
createSuggestionsDiv: function() {
  this.suggestionsDiv = document.createElement("div");
// О Создать элемент div
  this.suggestionsDiv.className = this.options.suggestDivClassName;
// © Определить стиль элемента div
  var divStyle = this.suggestionsDiv.style;
// © Добавить стиль поведения
  divStyle.position = 'absolute';
  divStyle.zIndex = 101;
  divStyle.display = "none";
```

```
//O Вставить в документ
  this.textlnput.parentNode.appendChild   (this.suggestionsDiv);
```

Метод создания контейнера имеет четыре основных обязательства, перечисленных в листинге. Прежде всего он должен создать элемент div посредством API createElement () документа О. Далее он должен определить стиль элемента div согласно клиентской конфигурации ©. Напомним, что одно из технических требований заключалось в том, чтобы стили CSS каждого экземпляра компонента допускали индивидуальную настройку. Для этого атрибут className элемента div устанавливается согласно свойству suggestDivClassName объекта опций. Значение этого свойства по умолчанию было задано (в методе setOptions) равным suggestDiv. Таким образом, если пользователь не укажет явно иное значение свойства, мы получим значение по умолчанию. Данное решение удобно, поскольку оно позволяет клиенту использовать таблицу стилей по умолчанию с нашими именами классов для определения стилей всех экземпляров компонента TextSuggest, используемых в приложении. Кроме того, можно предоставлять и другие таблицы стилей (например, заказные или реализованные в продукте), переопределяющие стандартные имена стилей. Кроме того, значение параметра suggestDivClassName может переопределяться на самой странице, формируя стилевое оформление компонента в пределах страницы или экземпляра. Звучит достаточно гибко.

Существуют определенные аспекты стиля всплывающего окна, которые мы не можем менять и которые относятся к поведенческим стилям, поэтому определяем их стиль явно, используя атрибут style элемента ©. Обратите внимание на то, что любой стиль, установленный программно посредством атрибута style, переопределяет все, что задано с помощью атрибута CSS className (обычно для этого используется таблица стилей). Перечислим упомянутые аспекты.

- Объявление position='absolute' (компонент должен управлять положением элемента div).
- Объявление zlndex=101 (всплывающее окно должно располагаться поверх любого другого элемента страницы).
- Объявление display="none" (всплывающее окно должно скрываться от пользователя до тех пор, пока его не активизирует ввод с клавиатуры).

Обратите внимание на то, что значение 101 переменной zlndex выбрано практически произвольно. Наконец, метод вводит элемент div в документ в качестве потомка текстового поля О. Родительский элемент в данном случае значения не имеет, поскольку div будет располагаться посредством абсолютного позиционирования.

Размещение всплывающего окна

Если мы создали всплывающее окно, рано или поздно его потребуется показать, но до этого его необходимо разместить в нужном месте При отображе-

нии всплывающего окна на экране требуется, чтобы оно располагалось сразу под текстовым полем и выравнивалось по его левому краю. Метод positionSuggestionsDiv, выполняющий данные действия, приведен в листинге 10.34.

Листинг 10.34. Размещение всплывающего пользовательского интерфейса

```
positionSuggestionsDiv: function!) {
  var textPos = RicoUtil.toDocumentPosition{this.textlnput);
  var divStyle = this.suggestionsDiv.style;
  divStyle.top = (textPos.y + this.textlnput.offsetHeight)
                 + "px";
  divStyle.left = textPos.x + "px";
  if ( this.options.matchTextWidth )
    divStyle.width = (this.textlnput.offsetWidth —
                      this.padding()) + "px";
```

Напомним, что в предыдущей версии сценария мы написали метод для расчета абсолютного положения текстового поля. В реструктуризированной версии мы будем полагаться на вспомогательный метод от Шсо — toDocumentPosition(). Все, что требуется, — использовать данный метод для получения опорной точки и выполнения соответствующих расчетов, помещающих всплывающее окно ниже текстового поля, выровненным по его левому краю. Затем мы проверяем существование конфигурационной опции matchTextWidth, и если ее значение равно true, то задаем ширину элемента div равной ширине введенного текста. Обратите внимание на то, что мы выровняли ширину с помощью значения заполнения. Это было необходимо, поскольку стиль элемента div может задаваться извне с помощью класса CSS. Мы не знаем, какие поля и границы пользователь приписал нашему компоненту, возможно, они нарушат наше визуальное выравнивание по ширине текстового поля. Поэтому напишем метод padding() (листинг 10.35), который вычислит величину заполнения и полей слева и справа, а затем вычтет их из общей ширины.

Листинг 10,35. Расчет заполнения[слева и справа

```
padding: function() {
  try{
    var styleFunc = RicoUtil.getElementsComputedStyle;
    var lPad = styleFunc( this.suggestionsDiv,
                          "paddingLeft",
                          "padding-left" );
    var rPad = styleFunc( this.suggestionsDiv,
                          "paddingRight",
                          "padding-right" );
    var lBorder = styleFunc( this.suggestionsDiv,
                             "borderLeftwidth",
                             "border-left-width" );
    var rBorder = styleFunc( this.suggestionsDiv,
                             "borderRightWidth",
                             "border-right-width" ) ;
    lPad = isNaN(lPad) ? 0 : lPad;
    rPad = isNaN(rPad) ? 0 : rPad;
    lBorder = isNaN(lBorder) ? 0 : lBorder;
```

```
    rBorder » isNaN(rBorder) ? 0 : rBorder;
    return parselnt(lPad) + parselnt(rPad) +
           parselnt(lBorder) + parselnt(rBorder);
  }catch (e){ return 0; }
```

Чтобы получить рассчитанный стиль элемента (действительное значение атрибута, вне зависимости от того, как оно было задано), придется немного потрудиться. Для решения этой задачи Internet Explorer предлагает собственный атрибут currentstyle для каждого элемента, браузеры Mozilla используют метод getComputedStyle () свойства defaultview документа. Все эти механизмы требуют также конкретной спецификации запрашиваемого атрибута. Атрибут currentstyle (Internet Explorer) ожидает, что внешний вид будет задан с помощью связывания в стиле JavaScript (например, borderRightWidth), тогда как getComputedStyle() (Mozilla) ожидает, что атрибуты будут заданы согласно синтаксису таблиц стилей (например, border-right-width). К счастью, библиотека Rico предлагает метод, решающий все эти проблемы, — RicoUtil.getElementsComputedStyle(). От нас требуется только передать ему элемент, имя атрибута Internet Explorer и имя атрибута Mozilla, а метод возвращает значение. В данном случае метод принимает величину границ и полей слева и справа, суммирует их и возвращает результат.

Метод Rico.getElementsComputedstyle() известен некорректной работой с некоторыми версиями Safari, поэтому мы вручную задаем возвращаемое значение по умолчанию в блоке try.. .catch.

Содержимое всплывающего окна

Итак, у нас есть код, создающий и размещающий всплывающее окно. Далее требуется написать метод, заполняющий это окно предлагаемыми вариантами. Напомним, что метод ajaxUpdateO разбирает ответный XML-документ, превращая его в массив объектов-предположений. Кроме того, если существует хотя бы одно предположение, вызывается метод this.updateSugggestionsDiv(). Данный метод преобразовывает внутреннее представление набора предлагаемых вариантов в реальные элементы span всплывающего окна (элемента div). Рассмотрим, как это происходит.

```
updateSuggestionsDiv: function() {
  this.suggestionsDiv.innerHTML = "";
// Удалить предыдущий контекст
  var suggestLines •> this,createSuggestionSpans();
// Создать новый контекст
  for (var i = 0; i < suggestLines.length; i++)
  this.suggestionsDiv.appendChild(suggestLines[i]);
),
```

Данный метод обманчиво прост — на самом деле предстоит еще многое сделать. Указанный метод устанавливает значение свойства innerHTML элемента suggestionsDiv, созданного ранее в виде пустой строки (чтобы уничтожить все предыдущее содержимое). Затем вызывается функция createSuggestionSpans (), создающая блок span для каждого варианта в массиве

предложений. Наконец, созданные блоки последовательно обрабатываются и добавляются к элементу div. Вот здесь уже начинается реальная работа. Рассмотрим подробнее функцию createSuggestionSpans (), приведенную в листинге 10.36, и разберем, как создаются блоки-варианты.

Листинг 10,36. Создание позиций списка предлагаемых вариантов

```
createSuggestionSpans: function() {
  var regExpFlags «• "";
  if { this.options.ignoreCase )
    regExpFlags - 'i';
  var startRegExp = " ";
  if { this.options.matchAnywhere )
    StartRegExp = ' ';
  var regExp — new RegExp( startRegExp +
                              this.lastRequestString,
                           regExpFlags );
  var suggestionSpans = [};
  for ( var i = 0 ; i < this.suggestions.length ; i++ )
    suggestionSpans.push(
      this.createSuggestionSpan( i, regExp ) );
  return suggestionSpans;
Ь
```

Данный метод вначале изучает объект опций и находит значение свойств ignoreCase и matchAnywhere. Благодаря этому мы определим соответствующие параметры регулярного выражения, которое облегчит извлечение из ответного документа части строки, совпадающей с текстом, набранным пользователем. Затем метод последовательно обрабатывает свойство suggestions (массив объектов, имеющих свойства .text и .value). Для каждого предлагаемого варианта в массиве вызывается метод createSuggestionSpan() с номером предположения и созданным ранее регулярным выражением. Таким образом, всю основную работу выполняет метод createSuggestionSpan(), показанный в листинге 10.37.

Листинг 10.37. Создание блока с предлагаемым вариантом

```
createSuggestionSpan: function( n, regExp ) {

  var suggestion = this.suggestions[nj;
  var suggestionSpan = document.createElement("span");
  suggestionSpan.className = this.options.suggestionClassName;
  suggestionSpan. style, width=* '100%';
  suggestionSpan.style.display = 'block';
  suggestionSpan. id = this.id + "__" + n;
  suggestionSpan.onmouseover =
              this.mouseoverHandler.bindAsEventListener(this);
  suggestionSpan.onclick =
              this.itemClickHandler.bindAsEventListener(this);
  var textValues - this.splitTextValues( suggestion.text,
                                  this.lastRequestString.length,
                                  regExp );
  var textMatchSpan = document.createElement("span");
  textMatchSpan.id - this.id + "_match_" + n;
```

```
  textMatchSpan.className = this.options.matchClassName;
  textMatchSpan.onmouseover =
              this.mouseoverHandler.bindAsEventListener(this);
  textMatchSpan.onclick =
              this.itemClickHandler.bindAsEventListener(this);
  textMatchSpan.appendChild(
    document.createTextNode(textValues.mid) );
  suggestionSpan.appendChild(
    document.createTextNode(textValues.start) );
  suggestionSpan.appendChild(textMatchSpan);
  suggestionSpan.appendChild(
    document.createTextNode(textValues.end) );
  return suggestionSpan;
Ь
```

Задача кажется непосильной, но сдаваться еще рано. Данный метод, возможно, выглядит сложнее, чем он есть на самом деле, хотя он действительно отвечает за многое. Пожалуй, стоит немного притормозить и рассмотреть его с точки зрения результата: получения HTML-кода предложений. Представим HTML-разметку предлагаемого варианта, которая выглядит примерно следующим образом:

```
<span>before <span>
 matching text
</span>, and after</span>
```

Мы сильно упростили реальную картину, чтобы проиллюстрировать структуру. Предположим, что пользователь ввел слова "matching text", а в базе данных имеется значение "before matching text, and after". Это значение будет предложено в качестве предполагаемого варианта, но, кроме того, с элементом span будут связаны дополнительные атрибуты, облегчающие идентификацию, стилевое оформление и обработку событий. Работу по отсечению фрагментов текста до и после соответствия выполняет следующая строка кода:

```
var textValues = this.splitTextValues(
suggestion.text, this.lastReguestString.length, regExp );
```

Возвращенное значение textValues представляет собой объект, имеющий три свойства: .start, .mid и .end. Таким образом, в приведенном примере textValues — это объект, который выглядит приблизительно так:

```
textValues = { start: 'before ',
               mid: 'matching text',
               end: ', and after' };
```

Наконец, ниже показана реализация метода splitTextValues ().

```
splitTextValues: function{ text, len, regExp ) {
  var startPos = text.search(regExp);
  var matchText = text.substring( startPos, startPos + len );
  var startText = startPos == 0 ?
                  "" : text.substring(O, startPos);
  var endText = text.substring( startPos + len );
  return { start: startText, mid: matchText, end: endText };
h
```

Рассмотрев структуру блока предлагаемого варианта, поговорим о важных атрибутах, сгенерированных для блока. Внешний и внутренний блоки создаются с именами классов CSS на основе значения свойств suggestionClassName и matchClassName объекта Options соответственно. Благодаря классам CSS полностью настраиваемым является не только suggestionsDiv но и вся внутренняя HTML-структура каждого предлагаемого варианта.

Из других важных атрибутов, генерируемых в блоке, стоит упомянуть идентификаторы, которые позволяют обработчикам событий извлекать нужные блоки. Далее в блоки необходимо поместить обработчик событий onmouseover; это позволит компоненту обновлять выбор, переводя его на позицию, над которой в настоящее время находится указатель мыши. Кроме того, с каждым предлагаемым вариантом необходимо соотнести обработчик событий onclick, чтобы после щелчка на строке предположения значение этой строки помещалось в текстовое поле. Реализация описанных обработчиков событий показана в листинге 10.38.

Листинг 10.38, Взаимодействие мыши с позицией списка

```
mouseoverHandler: function(e) {
  var src = e.srcElement ? e.srcElement : e.target;
  var index = parselnt (src. id. substring(src.id.lastIndexOf ('__')+l));
  this.updateSelection(index);
K
itemClickHandler: function(e) {
  this.mouseoverHandler(e);
  this.hideSuggestions() ;
  this.textlnput.focus();
```

Обработчик mouseoverHandler () просто находит цель события и вычленяет сгенерированный нами идентификатор, представляющий номер предлагаемого варианта. Затем можно использовать метод updateSelection(}, написанный на четвертый день, обновляя выбор и переводя его на позицию, над которой в текущее время находится указатель мыши.

Подобным образом обработчик itemClickHandler () также должен обновить выбор, поэтому он просто вызывает отвечающий за это обработчик mouseoverHandler(). Затем itemClickHandler() должен скрыть всплывающее окно с предлагаемыми вариантами, вызывая метод hideSuggestions () и возвращая текстовое поле в фокус, чтобы пользователь мог продолжать набор.

Наконец, мы завершили создание всплывающего окна. Сосредоточимся теперь на невероятно простой задаче его показа и сокрытия.

Отображение и сокрытие всплывающего окна

Создав код, обрабатывающий все сложные детали создания всплывающего списка с предлагаемыми вариантами, необходимо создать и код, отображающий и скрывающий этот список. К счастью, это довольно прямолинейный

процесс, знакомый любому эпизодическому разработчику DHTML-страниц. Отображение и сокрытие элемента обычно выполняются посредством манипуляций со свойством display стиля элемента. Наш компонент — не исключение. Поэтому изучите листинг 10.39, содержащий код с реализацией отображения и сокрытия всплывающего списка.

Листинг 10.39, Отображение и сокрытие всплывающего списка вариантов

```
showSuggestions: function() {
  var divStyle = this.suggestionsDiv.style;
  if ( divStyle.display == " )
    return;
  this.positionSuggestionsDiv();
// Разместить всплывающий список
  divStyle.display = ";
// Показать всплывающий список
>.
// Скрыть всплывающий список
hideSuggestions: function() {
  this.suggestionsDiv.style.display =
    'none';
Ь
```

В приведенном коде мы манипулируем свойством style.display метода suggestionsDiv, чтобы отобразить (посредством значения, равного пустой строке) и скрыть (посредством попе) всплывающее окно. Метод showSuggestions() выполняет дополнительную работу по размещению окна в нужном месте перед его отображением. Вот и все! Действительно все! Компонент готов. Подведем итоги.

10.5.6. Итоги

Да, это был достаточно сложный компонент с большим числом элементов. Мы создали компонент с возможностью повторного использования, которым можно гордиться. Разработанный компонент TextSuggest обрабатывает большое число конфигурационных параметров; допускает расширение; не нагружает сервер; к тому же он ненавязчив, работает во всех браузерах, имеет простой API... Другими словами, он удовлетворяет всем требованиям, перечисленным в табл. 10.2. Полностью исходный код компонента можно найти на сайте http://www.manning.com/crane (http://www.dialektika.com/); библиотеку Rico — на http://openrico.org/, а библиотеку Prototype — на сайте http://prototype.conio.net/.

10.6. Резюме

Предложение вариантов вводимого текста позволяет пользователям экономить время, предлагая им на выбор ту информацию, которая, возможно, им требуется. Иногда пользователь, набравший всего несколько букв, сразу получает нужные данные. В данной главе были рассмотрены недостатки су-

ществующих реализаций и разработано приложение, позволяющее обойтись без ненужных обращений к серверу благодаря интенсивной обработке на стороне клиента. При создании динамического пользовательского интерфейса допускающего взаимодействие с клавиатурой и мышью, мы использовании DHTML. В данном примере показано, как с помощью Ajax обеспечить гладкое взаимодействие с сервером без нарушения взаимодействия пользователя с Web-страницей. Кроме того, предложенный сценарий хорошо работает с браузерами, не поддерживающими Ajax, поскольку в подобных случаях текстовое окно опережающего ввода действует как обычное текстовое окно, в котором пользователи могут вводить данные (просто они не получают готовые предположения по поводу вводимого текста). Наконец, мы убрали объектно-ориентированную оболочку JavaScript, реструктуризировав сценарий в удобный конфигурируемый и полезный компонент TextSuggest.

11 Улучшенный Web-портал Ajax

В этой главе.

- Создание портала Ajax
- Реализация каркаса регистрации
- Создание динамических окон
- Запоминание состояния окна
- Адаптация кода библиотеки .

В настоящее время вес больше и больше компаний создают внутренние сети на основе порталов. Порталы предлагают пользователю удобный шлюз для получения больших объемов информации на одной странице Благодаря этому пользователю не требуется заходить на множество Web-сайтов, чтобы получить требуемую информацию. Интерактивные порталы, подобные Yahoo!, позволяют получать новости, прогнозы погоды, результаты спортивных соревнований, почту, игры и многое другое на одной странице. Другим примером портала является принадлежащий Amazon поисковый портал A9.com, позволяющий выполнять поиск во многих областях без переходов на отдельные страницы С его помощью на одной странице можно искать Web-страницы, книги, изображения и многое другое. В A9. com для отображения информации на экране используется Ajax. Это производит невероятно благоприятное впечатление, поскольку пользователю не требуется сидеть и ждать повторной визуализации страницы, когда будут отображены новые результаты поиска.

В данной главе мы вводим инфраструктуру Ajax в портал, улучшая процесс входа пользователя в систему и запоминание системой пользователей. Проект "Портал" позволит пользователю настраивать структуру портала с минимальными усилия.ми. Пользователь даже не поймет, что его действия вызывают отправку информации на сервер, где будет запоминаться точное положение объектов на странице. Это означает, что *его* личные настройки будут одинаковыми при каждом входе в систему. При создании портала мы вначале примем низкоуровневый поход и реализуем основной каркас портала менее структурированным образом, чтобы прояснить саму концепцию портала. Затем более подробно рассмотрим портал, используя объектно-ориентированный подход. Итак, прежде чем мы приступим к реализации портала, рассмотрим несколько существующих порталов и подумаем, как с помощью Ajax сделать их удобнее.

11.1. Эволюционирующий портал

Порталы развились из простых сайтов, позволяющих получать электронную почту и выполнять поиск, которые эволюционировали до изысканных конструкций, позволяющих быстро получать большой объем информации при минимальных усилиях. Для сравнения: прежде приходилось посещать один сайт для чтения новостей, другой — для получения прогноза погоды, на третьем располагались комиксы, четвертый специализировался на поиске и т.д. В подобных случаях приходилось либо использовать тонны закладок для сайтов, которые мы посещали ежедневно, либо запоминать, какие адреса нужно вводить в окно браузера.

11.1.1. Классический портал

Все мы уже привыкли к классическим порталам (мы используем их много лет). Во многих внутренних сетях компаний они применяются для повышения производительности, поскольку все находится в одном месте. Классический

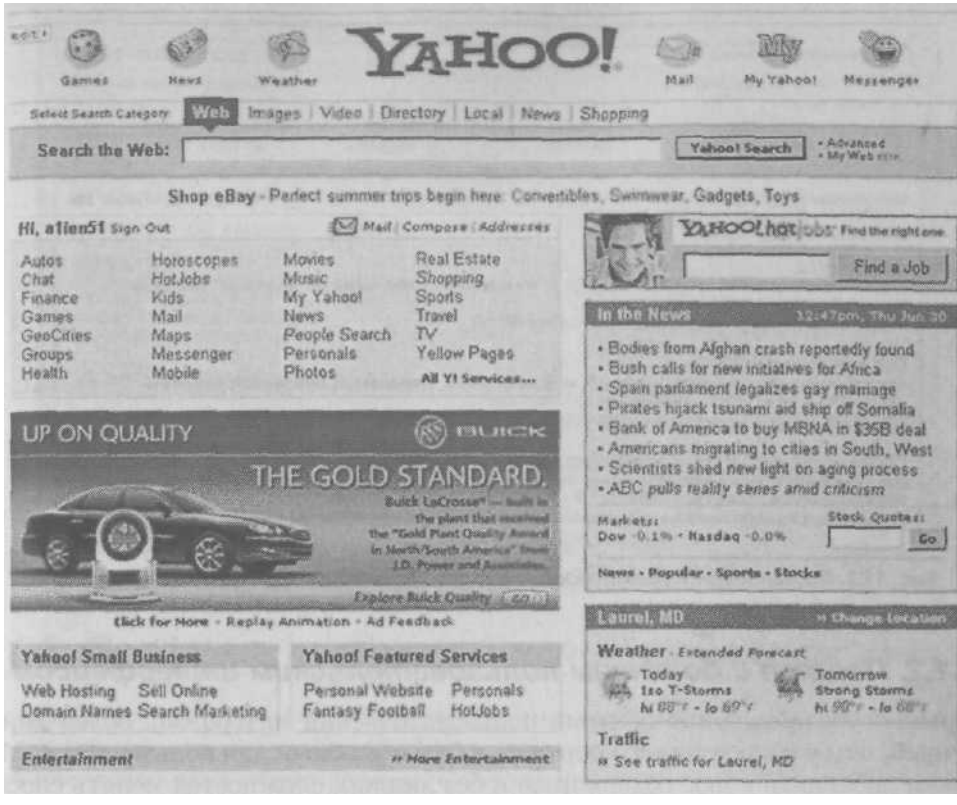


Рис. 11.1. Портал Yahoo! отображает заказную информацию

портал позволяет пользователю входить в систему и настраивать содержимое по своему усмотрению. Например, портал компании может содержать один набор настроек для агента по продажам, а другой — для программиста. Обоим сотрудникам может пригодиться окно с календарем проектов компании, но не обоим требуются графики продаж или сообщения об ошибках приложения. Ограничивая информацию, предоставляемую сотрудникам, мы повышаем и безопасность, и производительность, поскольку им не приходится искать нужную информацию по всей внутренней сети компании.

Еще одним классическим порталом является Yahoo!. Заходя на Yahoo!, мы можем получать почту, менять настройки прогноза погоды, чтобы они соответствовали нашему региону, изменять внешний вид страницы и др. Как видно на рис. 11.1, портал Yahoo! можно настроить под требования пользователя.

Для этого в Yahoo! реализован механизм технических страниц, на которых пользователь может менять необходимые данные. На одной из таких страниц можно выбрать родной город, чтобы отображаемый прогноз погоды касался только этой местности. На рис. 11.1, например, задан Мэриленд (США). Хотя возможность подобной настройки хороша уже сама по себе, хорошее впечатление пользователя можно усилить, применив Ajax так же. как сделала компания Amazon при создании портала A9.com.

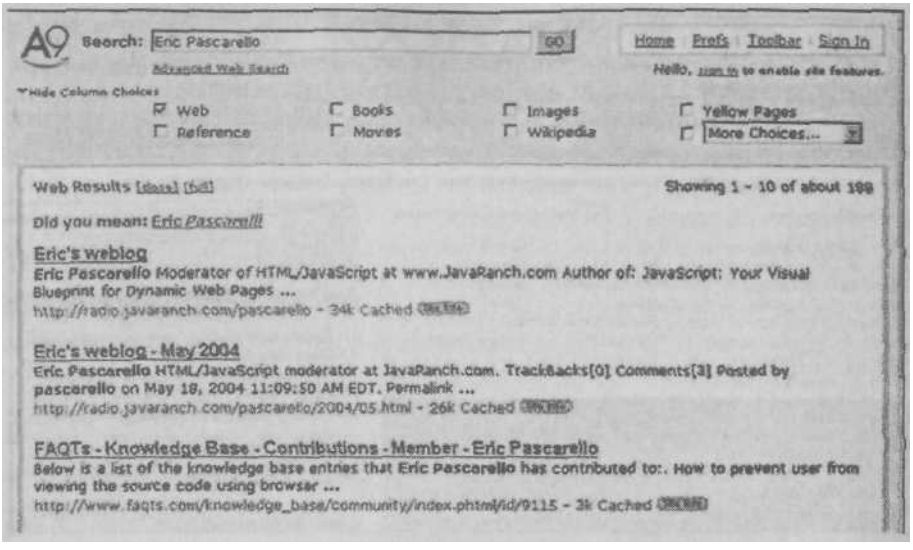


Рис. 11.2. Портал A9.com с результатами поиска в Web информации об Эрике Паскарелло

11.1.2. Портал с богатым пользовательским интерфейсом

Портал Ajax предлагает богатый пользовательский интерфейс, более динамичный, чем у классического портала, и более удобный для пользователя. Мы можем добавлять новое содержимое и без лишних сложностей менять способ его представления. Прекрасным примером этого легкого взаимодействия является поисковый портал Amazon A9.com. Посмотрим, как *он* работает. На рис. 11.2 показан поиск информации об Эрике Паскарелло с единственным установленным флажком Web.

Теперь сузим результаты поиска. Нам требуется книга, написанная данным человеком, поэтому мы устанавливаем флажок Books. В правой стороне страницы появляется панель Book Results. Как показано на рис 11.3, на экран выводятся результаты поиска книг Эрика Паскарелло без повторной обработки всей страницы на сервере.

Другой пример использования Ajax для повышения удобства портала связан с настройкой конфигурации портала. Ajax позволяет задействовать пользовательский интерфейс как инструмент управления конфигурацией — вместо того чтобы для настройки параметров переходить на другую Web-страницу, пользователь щелкает на объектах, находящихся в отображенном окне. Пользователь может динамически менять размеры и располагать элементы на экране, настраивая портал согласно своим предпочтениям.

Итак, ознакомившись с несколькими преимуществами портала Ajax, мы можем переходить к архитектуре портала, который планируем создать.

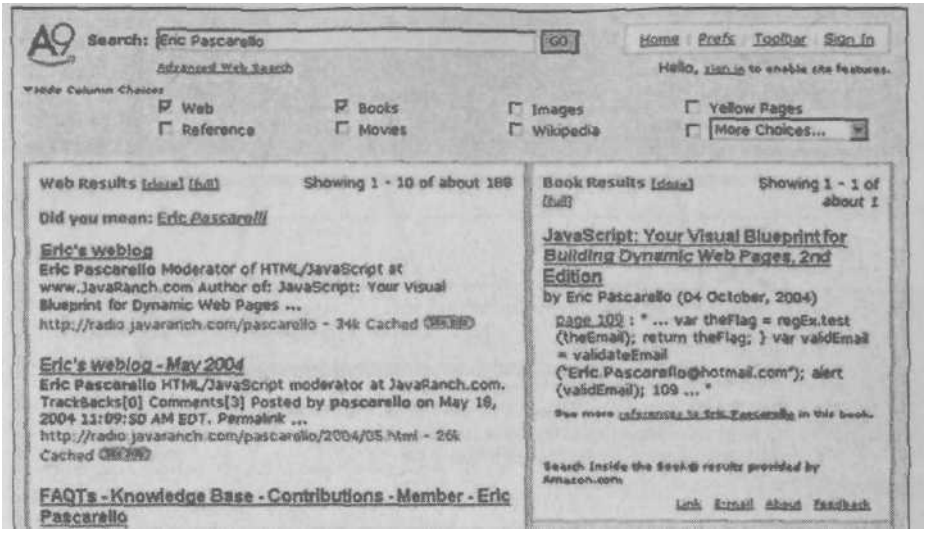


Рис. 11.3. Портал A9.com с результатами поиска и колонкой Book Results

11.2. Создание портала с использованием Java

Чтобы создать в высокой степени настраиваемый портал Ajax для большого числа пользователей, необходим код клиентской части сценария, код серверной части сценария и база данных. Сторона клиента обрабатывает взаимодействие пользователей с окнами — перетаскивание, отправку данных на сервер с помощью Ajax, Сервер, в свою очередь, обрабатывает сеансы пользователей, передачу данных клиенту и взаимодействие с базой данных. База данных в одной таблице содержит имена и пароли наших пользователей, а во второй — метаданные окна портала (положение, размер и содержимое). Выполнение данного проекта будет состоять из большого числа этапов, поскольку он является очень динамическим. Чтобы начать данный проект, рассмотрим его структуру (рис. 11.4).

Сама идея создания портала с богатым пользовательским интерфейсом для взаимодействия с сервером, использующим Ajax, кажется очень сложной, но вы не поверите, насколько просто можно реализовать данный проект. Архитектура портала, показанная на рис. 11.4, состоит из двух основных частей: исходной регистрации и динамического взаимодействия с окнами. Таким образом, процессы можно разбить на две основные группы и использовать функциональные возможности Ajax для удовлетворения требованиям обеих групп. Первый этап заключается в проверке регистрационных данных пользователя согласно информации, содержащейся в базе данных, второй — это взаимодействие с элементами DHTML и возврат значений клиенту.

В данной главе для обработки большого количества клиентского кода мы используем библиотеку DHTML. Она позволяет разрабатывать настраиваемые окна, использующие элементы IFrame для отображения содержимого.

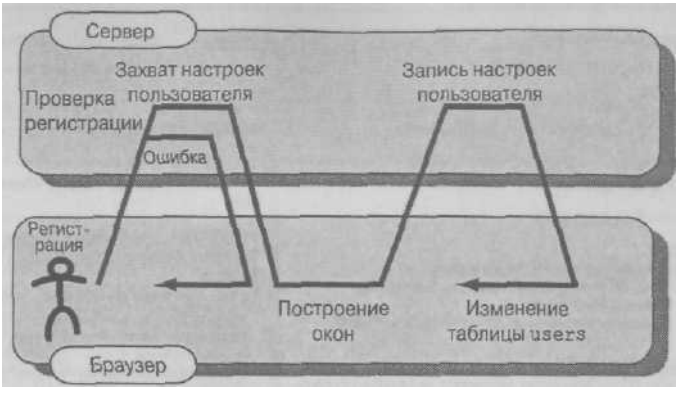


Рис. 11.4- Схема действии портала Ajax Пользователи входят на портал и управляют своими окнами. Изменения записываются автоматически в фоновом режиме

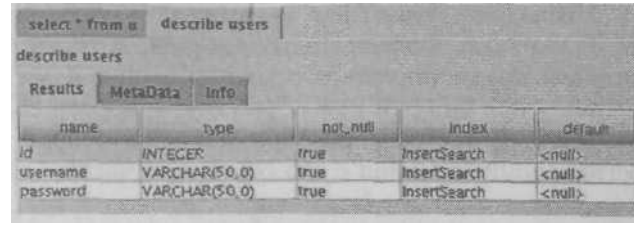


рис. 115. Свойства таблицы users в SQL Squirrel — графической программе работы с базами данных

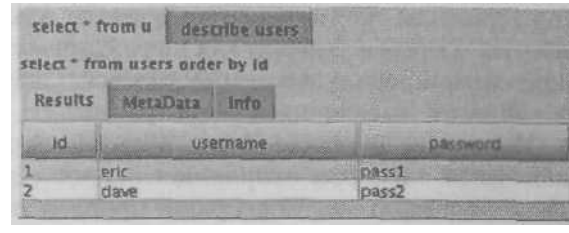


Рис. 11.6. Содержимое таблицы users

11.3.1. Таблица пользователя

Первой таблицей, с которой мы сталкиваемся в базе данных, является таблица users, содержащая три столбца В данном проекте мы будем использовать только необходимый минимум информации, но в зависимости от наших требований ее объем может увеличиваться. Три столбца таблицы, id, username и password, создаются с помощью стандартного SQL-выражения, приведенного в листинге 11.1. На рис. 11.5 показана таблица в клиентском приложении базы данных SQL Squirrel (http://squirrel-sql.sourceforge.net).

Листинг 11.1. Схема таблицы users

```
create table users(
    id int primary key unique not null,
    username varchar(50) not null,
    password varchar(50) not null
);
```

После создания таблицы необходимо создать несколько пользователей. В данном случае мы жестко кодируем имена пользователей и пароли. Как видно на рис. 11.6, в таблицу добавлены два пользователя с идентификационными номерами 1 и 2. Эти номера потребуются нам позже.

Далее необходимо создать учетные записи для пользователей, представленных в таблице. В нынешнем состоянии портал не представляет административный пользовательский интерфейс, облегчающий добавление новых пользователей, поэтому мы должны сделать это вручную — с помощью удобного инструмента базы данных. Разработка интерфейса управления пользователями на основе Ajax возможна, но для экономии места мы не будем ее рассматривать в данной книге.

Последнее, что от нас требуется, — сопоставить с таблицей права доступа. В учетных записях пользователей, которые будут обращаться к таблице **должны** предоставляться права на чтение и запись. Не задав права доступа, мы получим ошибки при обработке запросов SQL.

Итак, у нас есть таблица users, и теперь можно писать код процесса регистрации, начиная с его серверной части.

11.3.2. Серверная часть кода регистрации: Java

Код серверной части сценария портала Ajax по своей сути прост, но его разработка потребует множества шагов, поскольку нужно реализовать довольно много функциональных возможностей. Сейчас мы сконцентрируем внимание на вопросах, касающихся регистрации.

Напомним процесс. Когда пользователь входит на портал, код клиентской части сценария отправляет запрос серверу, передавая в нем информацию о пользователе. Серверный процесс, перехватывающий данный запрос, определяет, верна ли предоставленная информация. Если да, то запускается процесс создания окон портала. Если предоставленная пользователем информация неверна, на клиентскую страницу возвращается сообщение об ошибке.

Поскольку мы работаем на Java, то для обеспечения безопасности всех взаимодействий с сервером будем использовать сервлетный фильтр. Если вы не знакомы с этим термином, поясним: фильтр — это логика, которую можно связать с одним или несколькими ресурсами и которая может модифицировать запрос до его передачи целевому сервлету. Использование фильтров для обеспечения безопасности обсуждалось в главе 7. Если вы используете систему, не поддерживающую фильтры, то можете создать вспомогательный объект или функцию, проверяющую, зарегистрировался ли пользователь, и вызывать эту функцию вручную в начале каждой страницы, которую требуется защитить. Код фильтра регистрации приведен в листинге 11.2.

>, листинг 11.2. LoginFiiter. j ava: код серверной части процесса регистрации

```
public class LoginFiiter implements Filter {
    public void initfFilterConfig config)
    throws ServletException { }
    public void doFilter(
        ServletRequest request,
        ServletResponse response,
        FilterChain filterChain)
    throws lOException, ServletException {
        boolean accept=false;
            HttpSession session=(
             (HttpServletRequest)request).getSession();
            User user=(User)
                 (session.getAttribute("user"));
// О Проверить сеанс для объекта User
            if (user==null){
            accept=login(request);
// © Запросить аутентификацию
             }else{
            accept=true;
```

```
// © Разрешить вход
            }
                if (accept){
                    filterChain.doFilter
                        (request,response);

// О Отправить запрос дальше
                }else(
                    Writer writer=response.getWriter();
                    writer .write                          .. .
                        (JSUtil.getLoginError());
// © Вернуть код ошибки
                    writer,flush();
                    writer.closet);
                }

    }
    private boolean login(ServletRequest request){
// 0 Получить из запроса информацию о пользователе
            String user=request.getParameter("username");
            String password=request.getParameter("password");
        User userObj=findUser(user,password);
        if (userObj!=null){
                HttpSession session=
                    ((HttpServletRequest)request).getSession(true);
// О Записать сеанс для будущего использования
            session.setAttribute("user",userObj);
        }
        return (userObj!=null);
    }
    private User findUser(String user, String password) {
        User userObj=null;
            Connection conn=DBUtil.getConnection();
// © Сформировать выражение SQL
        try{
            String sql="SELECT id FROM users WHERE username='"
                +user+"' AND password™'"+password+"'";
            Statement stmt=conn.createStatement();
            ResultSet rs=stmt.executeQuery(sql);
            if (rs.next()){
// © Создать объект User
                int id=rs.getlnt("id");
                userObj=new User(id,user);
            }
        Icatch (SQLException sqlex){
        }
        return userObj;
    }
    public void destroyO { J
>_____M
```

В данном случае мы применяем фильтр, проверяющий, присутствует л* уже в сеансе объект User О. Если да, то мы его принимаем ©; в противного случае производится аутентификация с использованием имени пользовател*

и пароля, указанных в запросе ©. Если запрос принят, он передается сервлету О; в противном случае возвращается указание отобразить сообщение об ошибке ©. Весь сгенерированный код JavaScript заключается в интерфейсный объект JSUtil. Ниже показан метод, генерирующий сообщение об ошибке.

```
public static String getLoginError() {
    StringBuffer jsBuf=new StringBuffer()
        .append("document.getElementByldf 'spanProcessing')(BBSS)n")
        .append(" .innerHTML = ")
        .append("'The Username and Password are invalid';(BBSS)n");
    return jsBuf.toStringf);
}
```

За аутентификацию отвечает метод login (), приведенный в листинге 11.2. Мы извлекаем из запроса имя пользователя и пароль ®, а затем вызываем функцию finduserf), обращающуюся к базе данных и извлекающую соответствующую строку *О.* (Мы абстрагируемся от всех вопросов, связанных с базой данных; нас интересует только объект DBUtil.) Если строка, соответствующая данным пользователя, найдена, функция возвращает объект User ©, который затем записывается в сеансе *О* и используется при следующих прохождениях через данный фильтр, — далее нам уже не понадобится предоставлять в строке запроса имя пользователя и пароль, поскольку объект User уже будет присутствовать в сеансе.

Описанный подход обладает еще одной приятной особенностью — он облегчает выход пользователя из системы. Все, что нам нужно, — удалить объект User из сеанса.

Как показано в листинге 11.3, сам по себе объект User представляет структуру базы данных.

If' Листинг^ 13. User.java

```
public class User {
    private int id=-l;
    private String userName-null;
    public User(int id, String userName) {
        super();
        this.id = id;
        this.userName = userName;
    }
    public int getld() { return id;}
    public String getUserNamef) { return userName;}
```

В данном объекте мы не будем записывать поле пароля. При работе с порталом он нам не понадобится, кроме того, его использование в сеансе представляет угрозу безопасности! Поэтому работу над процессом регистрации на стороне сервера можно считать законченной. Ничего необычного. Теперь пойдем дальше и посмотрим, как с этим кодом будет реагировать клиентская часть приложения.

11.3.3. Структура регистрации (клиентская часть)

Клиентская часть кода регистрации состоит из двух частей. Первая — визуальная, которую пользователь может видеть и с которой может взаимодействовать. Данный HTML-код мы создадим динамически; вы удивитесь, насколько просто создать структуру с помощью элементов div, span и правил CSS.

Вторая часть сценария — код Ajax или JavaScript, отправляющий запрос на сервер и обрабатывающий данные. В данном случае мы собираемся ввести метод JavaScript eval (). Этот метод обрабатывает переданную ему строку как код JavaScript. Если строка содержит имя переменной, метод создает переменную. Если в качестве входного параметра eval () получает вызов функции, метод выполняет эту функцию. Метод eval() — довольно мощный, но его производительность может быть невысокой из-за сложности задействованных операций.

Структура HTML

Как и в предыдущих главах, для формирования структуры страницы мы не будем использовать таблицу. Использование таблиц для создания структуры увеличивает время визуализации страницы, а поскольку мы используем Ajax, то хотелось бы, чтобы все элементы нашего сценария работали и реагировали как можно быстрее. Нам необходимо поместить текстовое окно, поле пароля и кнопку отправки в форме, которую мы можем отправить на сервер. Кроме того, нам потребуется элемент span, чтобы мы могли отобразить поступившее с сервера сообщение об ошибке, если имя пользователя или пароль окажется недействительным. Помещая всю форму в элементы div и span, мы форматируем HTML-код, получая заголовок портала. В листинге 11.4 показана структура HTML-кода заголовка регистрации.

Листинг 11.4. HTML-структура регистрации *:ЫЁ*

```
<!— О Определить форму —>
<form name="Forml">
<i— Q Добавить заголовок —>
    <div id="header">
<!— © Ввести элемент span для регистрации —>
        <span id="login">
            Name:
<!— О Добавить текстовое поле для имени пользователя —>
            <input type="text" name="username">
<!— © Добавить элемент для пароля —>
            <br>Password:
        <input type="password" name="password">
            <br/>
<!— О Ввести обрабатывающий элемент span —>
            <span id="spanProcessing"x/span>
            <input type="button" name="btnSub" value-"login"
<!— © Добавить кнопку "Отправить" —>
            onclick="LoginRequest()">
        </span>
```

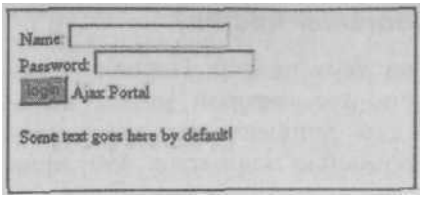


Рис. 11.7. Регистрационная форма HTML без правил CSS

```
<!— © Добавить текст —>
        <span id="sloganText">Ajax Portal</span>
    </div>
<!— © Добавить контекст по умолчанию —>
    <div id="defaultContent">
        <p>Some text goes here!</p>
    </div>
    <div id="divSettings" class="hidden"X/div>
</form>
```

Вначале мы добавляем форму О к HTML-документу. Эта форма обеспечивает семантически значимый контейнер для текстовых окон. Кроме того, она предоставляет путь для отступления — аутентификации без использования Ajax, т.е. посредством обычной отправки формы. Мы создаем заголовок div ©, вмещающий в себя весь наш код. Затем добавляем элемент span ©, содержащий текстовое поле имени пользователя О или поле пароля 0, обрабатывающий элемент span 0 и кнопку отправки ©.

С кнопкой, используемой для отправки данных на сервер, должен соотноситься обработчик событий onclick. Этот обработчик инициализирует Ajax, вызывая функцию JavaScript LoginRequestt) (объясняется ниже, в разделе "Код регистрации (JavaScript)").

Чтобы завершить работу с заголовком, осталось добавить текст © портала и место для содержимого по умолчанию ©, которое будет отображаться при загрузке страницы. В элементе div с именем defaultContent можно отобразить любое сообщение. В данном примере мы просто помещаем в него строку текста, но вообще можно добавлять ссылки, изображения, текст и все, что нам будет угодно. Далее мы записываем HTML-форму; ее непрезентабельный внешний вид без применения к элементам правил CSS показан на рис. 11.7.

Чтобы украсить эту "пресную" структуру, с элементами нужно соотнести правила CSS. Поскольку мы присвоили элементам собственные идентификаторы, это упрощает процесс. Чтобы обратиться к идентификатору элемента, поместим перед ним знак #• Таблицу стилей можно добавить как внешний или встроенный файл, используя дескриптор <style>. В данном случае используется встроенный дескриптор <style>, добавляемый в заголовок документа. Как показано в листинге 11.5, мы добавляем правила CSS, меняющие цвета, шрифты, размеры, положения, поля и т.д.

Листинг 11.5. Правила CSS регистрационной формы

```
<style type="text/css">
<!— О Элементы html и body —>
    html, body{ margin: Opx; padding:Opx;
                          height:100%; }
<!— © Определить стиль элемента заголовка -->
#header{ background-color: #C0C0C0;
                 height: ЮОpx;
                 border-bottom: lpx solid black;
                 font-weight: bold; J
<!— © Разместить элемент span регистрационной формы —>
#login{ text-align: right; float: right;
                margin-top:15px;
                margin-right:15px; )
<!-- О Отформатировать текст заголовка -->
#sloganText{ font-size: 25px;
                          margin-left: 15px;
                          line-height: ЮОpx; }
</style>
```

Вначале мы удаляем все поля или заполнения из тела О документа. Высоту (свойство height) мы задаем равной 100%. Важно отметить, что нам необходимо задать данные свойства в дескрипторах HTML и body, поскольку различные браузеры получают данную информацию из одного из указанных дескрипторов.

Разрабатывая стилевое оформление заголовка ©, мы можем задать цвет фона элемента div. Кроме того, можно установить высоту и добавить границу кнопки, чтобы отделить заголовок от содержимого. Помимо этого, можно изменить свойства шрифтов, которые мы посчитаем нужными.

Мы принимаем регистрационную информацию © и перемещаем ее в правую сторону экрана. Для этого используется свойство float со значением right. Для выравнивания текстовых блоков применяется свойство text-align, поэтому содержимое элемента span также выравнивается по правому полю. Благодаря этому текстовые окна выглядят более унифицированно. В противном случае они не были бы выровнены правильно, поскольку имя строки короче, чем пароль. Кроме того, для выравнивания положения регистрационной информации можно добавить поля, чтобы правый край соответствующего окна не примыкал непосредственно к границе элемента div заголовка.

В заключение необходимо определить стиль текста заголовка О. Задавая высоту строки (свойство lineheight) равной высоте элемента div, мы позволяем центрировать текст заголовка вертикально. Кроме того, свойства шрифта задаются так, чтобы текст был заметен. Далее мы добавили поле, поэтому первая буква в слове Ajax не выровнена по краю заголовка. После применения правил CSS к заголовку можно записать документ и посмотреть, как таблицы CSS изменили его внешний вид (рис 11.8).

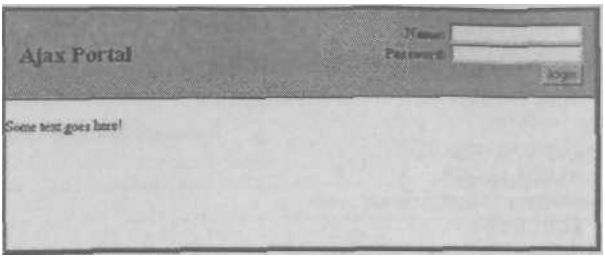


Рис. 11.8. Страница регистрации портала Ajax с разработанной таблицей стилей

На рисунке видно, что текстовые окна выровнены по правому краю, а текст заголовка — по левому. Таким образом, мы взяли стандартную HTML-структуру и создали привлекательную шапку страницы регистрации, не использовав при этом ни одной таблицы. Разработав стилевое оформление заголовка, можно добавить к форме ряд функциональных возможностей. Например, нам нужны возможности JavaScript, чтобы мы могли передать запрос на сервер, не отправляя всю страницу.

Код регистрации (JavaScript)

Написав код регистрации JavaScript, мы, используя возможности Ajax, сможем передавать имя пользователя и пароль на сервер, не отправляя при этом всю страницу. Для этого потребуется обратиться к внешнему файлу JavaScript net.js, содержащему объект ContentLoader, чтобы мы могли задействовать Ajax для отправки и извлечения запроса.

```
<script type="text/javascript" src="net.js"x/script>
```

Файл ContentLoader выполняет все действия, связанные с отправкой ин-3 формации на сервер, сокрытием кода, зависящего от конкретного браузера, за удобным интерфейсным объектом, введенным в главе 3. Итак, обратив- ' шись к файлу net.js, мы можем выполнить запрос. Этот запрос инициируется щелчком на кнопке в нашей форме регистрации. Сама форма при этом должна выполнить три действия: во-первых, сообщить пользователю, что его запрос обрабатывается; во-вторых, собрать информацию и, в-третьих, отправить запрос на сервер (листинг 11.6).

Листинг 11.6. Запрос регистрации XMLHttpRequest

```
function LoginReguest(){
    document.getElementById("spanProcessing").innerHTML =
        " Verifying Credentials";
    var url = 'portalLogin.servlet';
    var strName = document.Forml.username.value;
    var strPass = document.Forml.password.value;
    var strParams = "user="+strName + "&pass=" + strPass
    var loaderl = new net.ContentLoader(
        url,CreateScript,null,"POST",strParams);
```

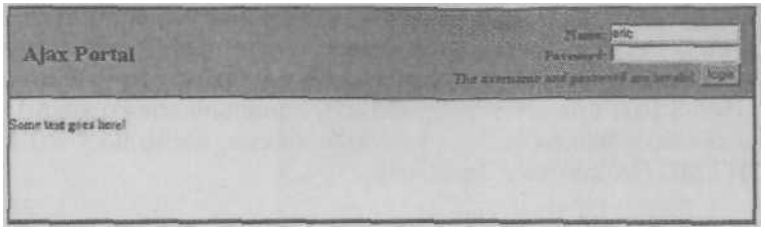


Рис. 11.9. Сообщение об ошибке, вызванное предоставлением неверных сведений

Прежде чем отправлять информацию на сервер, необходимо отобразить для пользователя сообщение о том, что щелчок на кнопке позволяет ему войти в систему. Благодаря этому пользователь не будет повторно щелкать на кнопке, думая, что ничего не происходит.

Итак, мы получаем поля имени пользователя и пароля и помещаем их в строку, отправляемую на сервер. Значения отправляются на сервер с помощью объекта ContentLoader, принимающего в качестве параметров URL, функцию, вызываемую при удачном завершении действия, функцию, вызываемую в случае ошибки, действие формы POST, а также строку, содержащую отправляемые параметры. Рассмотрим подробнее функцию, вызываемую при успешном возврате данных с сервера и обрабатывающую эти данные: CreateScript{).

```
function CreateScript() {
    strText - this.req.responseText; eval(strText);
}
```

При создании серверной части сценария мы возвращали текстовые строки, содержащие выражения JavaScript в свойстве responseText возвращаемого объекта. Чтобы эффективно использовать выражения JavaScript, их необходимо обработать с помощью метода eval(), точно определяющего, что содержит строка, и выполняющего указанные в ней действия. В данном случае строка либо будет содержать сообщение об ошибке, сгенерированное при сбое LoginFilter, либо код создания окна, если фильтр пропускает данные к SelectServlet (листинг 11.8).

Из чего состоит строка? В данном приложении мы не собираемся возвращать в ответ на запрос XML-документ, как мы поступали во многих других примерах. Вместо этого мы вернем структурированные выражения JavaScript, которые смогут использовать метод eval(). Используя термины, сформулированные в главе 5, можно сказать, что наше решение ориентировано скорее на сценарий, чем на данные. Как и ранее, мы приняли данный подход только ради разнообразия. В качестве среды передачи при разработке кода портала можно использовать XML или JSON.

Теперь можно сохранить портал и запустить его, чтобы посмотреть, как работает процедура регистрации. Как видно на рис. 11.9, в поля были введены неверное имя пользователя и пароль.

Ниже кнопки "login" на рис. 11.9 отображено сообщение об ошибке, **информирующее** пользователя, что предоставленные им данные неверны.

Если, с другой стороны, регистрация прошла успешно, запрос перенаправляется на основную страницу портала. В этом случае следующим шагом является создание окон. Чтобы получить намеченный богатый пользовательский интерфейс, нам придется разработать довольно много кода DHTML, но на самом деле эта тяжелая работа уже выполнена, поскольку мы используем готовую DHTML-библиотеку JavaScript.

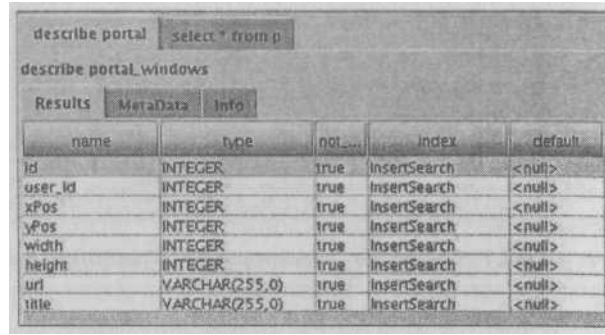
11.4. Реализация окон DHTML

Наш портал Ajax имеет богатый пользовательский интерфейс, позволяющий пользователю динамически размещать окна. Кроме того, пользователь может устанавливать размер окна (желательную ширину и высоту). При изменении указанных настроек мы можем с помощью Ajax реализовать взаимодействие с сервером и записать новые значения в собственной базе данных (причем пользователь даже не будет знать об этом). Чтобы разрешить подобное поведение, требуется разработать таблицу базы данных, в которой будут храниться свойства окна — высота, ширина и положение. Код серверной части сценария должен получать данные значения и соответствующим образом обновлять величины в базе данных. Написание DHTML-кода, совместимого с основными браузерами, может оказаться сложным, поэтому для реализации перетаскивания и изменения размеров окна мы используем сценарий библиотеки DHTML. Данная библиотека, JSWindow.js, представляет собой внешний файл JavaScript, содержащий код всех требуемых нам функций . (Эту библиотеку можно найти на Web-сайте данной книги.) Чтобы активизировать инфраструктуру Ajax, нам потребуется лишь незначительно изменить код данной библиотеки.

11.4.1. База данных окон портала

Нам нужна таблица базы данных, способная содержать свойства нескольких окон DHTML для каждого пользователя. Каждый пользователь может иметь несколько строк в данной таблице — по одной на каждое окно портала. Таблица применяется для извлечения последнего запомненного положения и размера окна при первом входе пользователя в систему. Когда пользователь что-то меняет, значения обновляются, чтобы при следующем посещении окна располагаться точно так же. Для создания таблицы portal_windows используется следующий SQL-код:

```
create table portal_windows(
    id int primary key not null, user_id int not null,
    xPos int not null, yPos int not null,
    width int not null, height int not null,
    url varchar(255) not null, title varchar(255) not null
);
```



name	type	not_...	index	default
id	INTEGER	true	InsertSearch	<null>
user_id	INTEGER	true	InsertSearch	<null>
xPos	INTEGER	true	InsertSearch	<null>
yPos	INTEGER	true	InsertSearch	<null>
width	INTEGER	true	InsertSearch	<null>
height	INTEGER	true	InsertSearch	<null>
url	VARCHAR(255,0)	true	InsertSearch	<null>
title	VARCHAR(255,0)	true	InsertSearch	<null>

Рис. 11.10. Структура таблицы portal_windows

Каждый пользователь может иметь несколько окон с различными настройками. Эта проблема решается следующим образом. Столбец user_id связан с базой данных пользователей. Каждое окно должно иметь идентификатор и первичный ключ, которые можно использовать для хранения и обновления свойств. Не забудьте добавить к столбцу идентификатора окна автоматический инкремент. Данный идентифицирующий столбец используется кодом Ajax и библиотекой окон DHTML для получения и обновления свойств окон, определенных пользователем.

Для хранения координат x и y нашего окна DHTML потребуются два столбца. Данная информация определяет положение окна на экране относительно левого верхнего окна браузера. Данные столбцы, содержащие координаты, обозначаются xPos и yPos. Кроме того, нужны еще два свойства окна DHTML: ширина и высота. Они хранятся в таблице как целочисленные величины.

Последние два столбца базы данных определяют URL содержимого окна, а также заголовок содержимого, присвоенный пользователем для быстрого доступа. Все свойства, фигурирующие в таблице portal_windows базы данных, показаны на рис 11.10.

Далее требуется ввести определенные значения, заданные по умолчанию, и выполнить тестирование. В таблицу users мы можем добавлять любое число пользователей. Как показано на рис. 11.10, для пользователя 1 мы добавили три окна DHTML.

На рис. 11.11 показано, как три параметра окна DHTML предоставляю! информацию, требуемую для создания трех окон на экране, имеющих различные размеры и положения. В данном случае в трех окнах отображаются тр* Web-сайта: JavaRanch, Google и Eric's Ajax Blog. Создав таблицу базы данных, мы можем предоставлять указанную информацию пользователю прр его входе на портал. Насколько это просто, вы узнаете в следующем разделе

11.4.2. Серверный код окна портала

Предположим, что запрос регистрации прошел через фильтр безопасности Следующий этап — извлечение списка окон портала для аутентифицированного пользователя и возврат JavaScript-кода, сообщающего браузеру, чтс

describe portal | select * from p

select * from portal_windows order by id,user_id

Results | MetaData | Info

id	user_id	xPos	yPos	width	height	url	title
1	1	612	115	615	260	http://www.javaranch.com	JavaRanch
2	1	10	115	583	260	http://www.google.com	Google
3	1	10	387	1220	300	http://radio.javaranch.com/pascarello	Ajax Blog!

Рис. 11.11. Данные, введенные для пользователя с идентификатором 1

нужно отображать на экране. В связи с этим мы определяем объект PortalWindow, представляющий строку информации в базе данных, как показано в листинге 11.7.

Листинг 11.7. PortalWindow.java

```
public class PortalWindow {
    private int id=-1;
    private User user=null;
    private int xPos=0;
    private int yPos=0;
    private int width=0;
    private int height=0;
    private String url=null;
    private String title=null;

    public PortalWindow(
        int id. User user, int xPos, int yPos,
        int width,int height,
        String url, String title
    ) {
        this.id = id;
        this.user = user;
        this.xPos = xPos;
        this.yPos = yPos;
        this.width = width;
        this.height = height;
        this.url = url;
        this.title = title;
    }

    public int getHeight() {return height;}
    public void setHeight(int height) {this.height » height;}
    public int getId() {return id;}
    public void setId(int id) {this.id = id;}
    public String getTitle() {return title;}
    public void setTitle(String title) {this.title = title;}
    public String getUrl() {return url;}
    public void setUrlfString url) {this.url = url;}
    public User getUser() {return user;}
    public void setUser(User user) {this.user = user;}
    public int getWidth() {return width;}
    public void setWidth(int width) {this.width • width;}
    public int getXPos() {return xPos;}
```

```
    public void setXPos(int pos)  {xPos = pos;J
    public int getYPos()  {return yPos;}
    public void setYPos(int pos)  {yPos = pos;}
    }
```

Как и прежде, объект представляет собой достаточно прямолинейное отображение структуры базы данных. В реальной задаче мы, возможно, использовали бы для облечения жизни ORM-систему, например Hibernate или iBATIS, но сейчас нам требуется, чтобы все было довольно простым и платформенно-независимым. Обратите внимание на то, что мы предоставляем для этого объекта как методы установки, так и методы получения, поскольку нам нужно, чтобы объекты обновлялись динамически в ответ на события на стороне клиента. Запрошенный при регистрации URL, portalLogin, servlet, отображается в сервлет, извлекающий все окна портала для этого пользователя и возвращающий в ответ указания JavaScript. Основной сервлет показан в листинге 11.8.

Листинг 11.8. Сервлет SelectServlet.Java

```
public class SelectServlet extends HttpServlet {
    protected void doPost(
        HttpServletRequest request,
                HttpServletResponse response
    ) throws ServletException, IOException {
        HttpSession session=request.getSession();
        User user={User)
// О Проверить сеанс
            (session.getAttributef"user"));
        StringBuffer jsBuf=new StringBuffer();
                if (user==null){
                    jsBuf.append(JSUtil.logout());
                }else{
                    List windows=DBUtil
// 0 Определить объект
                        .getPortalWindows(user);
// в Использовать объект JSUtil
                    j sBuf.append(JSUtil.initUI()  );
               for (Iterator iter=windows.iterator();iter.hasNext();)
                PortalWindow window=(PortalWindow)(iter.next());
                session.setAttribute("window_"+window.getld(),window)
                jsBuf.append
// О Объявить окно портала
                                (JSUtil.initWindow(window));
                    }
                J
        Writer writer=response.getWriter();
//  0 Записать в выходной поток.
        writer.write(jsBuf.toString());
        writer.flush();
    }
```

Здесь мы снова используем объект DBUtil для абстрагирования взаимодействий с базой данных и JSUtil — для генерации кода JavaScript. DBUtil предоставляет метод getPortalWindows () ©, принимающий в качестве аргумента объект User. Один из таких объектов фигурировал в информации о сеансе, поэтому теперь мы его извлекаем О. Фактический код JavaScript генерируется (как и ранее) объектом JSUtil. создающим код инициализации пользовательского интерфейса ©, объявляющим окна портала, извлеченные из базы данных О. и записывающим их непосредственно в выходной поток сервлета 0.

Напомним коротко, что представляют собой вспомогательные объекты которые мы постоянно используем, — DBUtil и JSUtil. Объект DBUtil используется для получения списка окон портала. Как отмечалось выше, в реальной задаче этот процесс было бы неплохо автоматизировать, используя Hibernate или похожую систему; но с обучающей целью мы используем приведенный в листинге 11.9 метод объекта DBUtil, являющийся реализацией доступа к таблице portal_windows в базе данных. В данном случае применяется прямолинейный код SQL, но представленный принцип легко перенести на любой удобный для вас язык.

Листинг 11.9. Метод getPortalWindows О. - ^Щ

```
public static List getPortalWmdows (User user){
    List list=new ArrayList();
    Connection conn=getConnection();
    try{
        String sql="SELECT * FROM portal_windows "
            +"WHERE user_id="+user.getId();
//  О Построить выражение SQL
            Statement stmt=conn.createStatement();
            ResultSet rs=stmt.executeQuery(sql);
            PortalWindow win=null;
            while (rs.next()){
// @ Последовательно пройти результаты
                int id=rs.getInt("id");
                int x=rs.getInt("xPos");
                int y=rs.getInt("yPos");
                int w=rs.getInt("width");
                int h=rs.getInt("height");
                String url=rs.getString("url");
                String title-rs.getString("title");
                win=new PortalWindow(
// © Добавить объект
                    id,user,x,y,w,h,url,title) ;
                list.add(win); }
        rs .close();
        strat.close();
    }catch (SQLException sqlex){
    }
    return list;
}
```

Итак, мы формируем выражение SQL O, последовательно проходим по сгенерированному этим выражением набору результатов © и во всех случаях добавляем к нашему списку объект PortalWindow ©.

Далее мы используем вспомогательный объект JSUtil, генерируем код инициализации и объявляем объекты окна на JavaScript. Соответствующие методы по сути представляют собой упражнения по конкатенации строк, поэтому мы не будем приводить здесь класс целиком. Принцип его действия демонстрируется в приведенном ниже коде.

```
public static String initwindow(PortalWindow window) {
    StringBuEfer jsBuf=new StringBuffer{)
        .append("CreateWindow{new NewWin('")
        .append(window.getId())
        .append("',")
        ,append(window.getXPos())
        .append(",")
        .append(window.getYPos())
        .append(",")
        .append(window.getWidth())
        .append(",")
        .append(window.getHeight())
        .append(",'")
        .append(window.getUrK))
        .append("','")
        .append(window.getTitle{))
        .append^")); (BBSS)n");
    return jsBuf.toString();
}
```

Метод initwindow () генерирует код JavaScript для инициализации одного окна портала. Код JavaScript успешного запроса может выглядеть приблизительно так, как показано ниже: для всех окон последовательно вызывается функция initwindow () (для улучшения читаемости код был соответствующим образом отформатирован):

```
document.getElementByld('login' )
    .innerHTML='Welcome back!'
document.getElementByld('defaultConten')
    .style.display=tquotenonetquote;
CreateWindow(
    new NewWin(
        tquoteltquote,612,115,615,260,
        tquotehttp://www.javaranch.comtquote,tquoteJavaRanchtquote
    )
);

CreateWindow(
    new NewWin(
        tquote2tquote,10,115,583,260,
        tquotehttp://www.google.comtquote,tquoteGoogletquote
    )
>;

CreateWindow(
    new NewWin(
```

```
            tquote3tquote,10,387,1220,300,
            tquotehttp://radio.javaranch.com/pascarellotquote,
            tquoteAjax Blog!tquote
        )
    >;
```

Поскольку мы уже вошли в систему, текстовое окно регистрации и кнопку "Отправить" можно убирать; вместо них отображается приветственное сообщение. Чтобы разместить это сообщение, необходимо будет скрыть содержимое, по умолчанию располагающееся на экране. Для этого свойство display элемента DOM defaultContent устанавливается равным попе, поэтому оно убирается из поля зрения пользователя.

Обрабатывающий окно оператор JavaScript состоит из двух частей. Первая часть представляет собой вызов функции CreateWindow(), являющейся частью добавленной нами библиотеки JavaScript. В вызове функции мы вызовем новый конструктор объекта. Конструктор создает класс окна, облегчающий обращение к свойствам окна. Функция JavaScript, создающая класс окна, должна получить параметры OKHaid, width, height, xPos, yPos, url и title. Когда сервлет возвратит эту строку клиенту, метод JavaScript eval () выполнит ее.

Как правило, мы придерживаемся договоренностей относительно генерации кода, дающих простой, повторяемый код, вызывающий функции нашей библиотеки JavaScript. Код инициализации можно поместить в один вызов, привязав его к клиенту, но реализовать данную возможность мы предоставляем читателю в качестве самостоятельного упражнения.

Использованная нами библиотека JavaScript создает плавающие окна JavaScript. Давайте посмотрим, как сделать эти функции создания окон доступными на стороне клиента.

11.4.3. Добавление внешней библиотеки JavaScript

Как отмечалось ранее, мы используем библиотеку DHTML, которую можно загрузить с Web-сайта www.manning.com. Файл JSWindow. js содержит все методы DOM JavaScript, необходимые для создания элементов окон. Кроме того, библиотека применяет обработчики событий к объектам окон, так что мы можем использовать возможности перетаскивания. Готовые библиотеки кода удобны тем, что позволяют сокращать время разработки и обычно работают во всех браузерах.

Первым делом требуется переименовать файл, чтобы мы могли модифицировать его. Запишем файл JavaScript под именем A j axWindow. j s в рабочей папке.

Чтобы использовать функции, содержащиеся в AjaxWindow. js, необходимо сослаться на внешний файл JavaScript посредством дескриптора script. В дескрипторе JavaScript element мы используем атрибут src. Элемент script, связывающийся с требуемым файлом .js, необходимо включить в заголовок нашей HTML-страницы.

```
<script type="text/javascript" src="AjaxWindow.js"X/script>
```

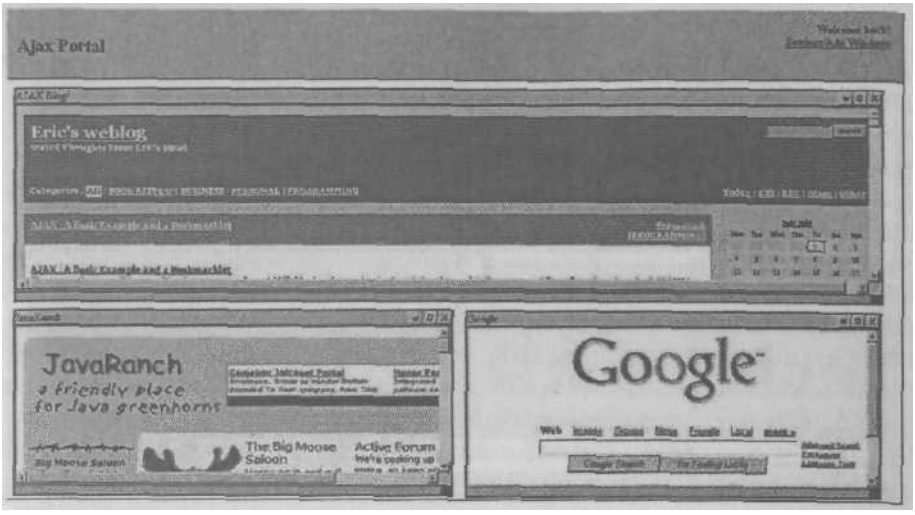


Рис. 11.12. Портал Ajax с тремя открытыми на экране окнами

Кроме того, нам потребуется таблица стилей окон DHTML, чтобы мы могли определить стилевое оформление окна. Для этого загрузите файл AjaxWindow.css с Web-сайта www.manning.com (или www.dialektika.com) и присоедините его к нашему сценарию, используя дескриптор link и атрибут href.

```
<link rel="stylesheet" type="text/css"
    href="AjaxWindows. css"x/link>
```

Итак, мы присоединили к HTML-странице файлы JavaScript и CSS, поэтому теперь можно проверить, правильно ли мы это сделали. Кроме того, проверим, правильно ли код серверной части сценария вызывает библиотеку JavaScript. Если код присоединен правильно и мы надлежащим образом получаем данные с сервера, то после входа в систему с использованием имени пользователя и пароля из базы данных мы, согласно хранимой в базе информации, должны получить три окна, показанных на рис 11.12. Помните, что созданная нами библиотечная функция формирует окна и связывает с ними все требуемые функциональные возможности. Выглядит, как волшебство, поскольку мы вызываем одну функцию, и все работает.

Открыв окна портала, можно проверить функциональные возможности, встроенные в библиотеку DHTML (функциональные возможности Ajax мы пока не рассматриваем). Вообще, в портале можно реализовать следующие дополнительные возможности.

- Щелчок на кнопке с пометкой О максимизирует и минимизирует окно.
- Щелчком на кнопке со знаком х окно можно скрыть.
- Щелчком на кнопке с буквой w можно открыть окно DHTML в собственном раскрывающемся окне.

Рис. 11.13. Другое расположение окон портала Ajax

- Нажав левую кнопку мыши в то время, когда курсор наведен на заголовок окна, окно можно захватить и с помощью мыши переместить на другое место.
- Отпустив левую кнопку мыши, мы завершаем операцию перетаскивания.

Для изменения размера окна необходимо щелкнуть на зеленом квадратике в правом нижнем углу окна и перетащить его в требуемое положение.

В качестве иллюстрации к сказанному обратите внимание на то, что окна на рис. 11.13 располагаются не так, как на рис. 11.12.

Теперь мы можем с помощью библиотеки размещать окна и изменять их размеры; далее требуется внести изменения во внешний файл .js. Данные изменения позволят вызывать функцию, с помощью Ajax отправляющую в базу данных новые значения свойств.

11.5. Возможность автоматического сохранения

Использование Ajax позволяет реализовать функцию автоматического сохранения, которая может вызываться любым событием (причем пользователь об этом знать не будет). Обычно пользователь инициирует отправку информации на сервер, щелкая на некоторой кнопке. В данном случае функция автоматического сохранения будет запускаться событием onmouseup, которое завершает процессы перетаскивания и изменения размеров. Если бы мы по событию onmouseup инициировали обычную отправку формы, мы бы нарушили взаимодействие пользователя с порталом и потеряли все, чего с таким трудом добивались. Используя Ajax, мы делаем процесс непрерывным.

11.5.1. Адаптация библиотеки

Как отмечалось ранее, код библиотек DHTML JavaScript обычно приемлем во всех основных браузерах, что позволяет нам использовать его, не задумываясь о вопросах совместимости. Изучая код внешнего JavaScript-файла AjaxWindow.js, вы обнаружите много интересных функциональных возможностей (поскольку соответствующий код довольно длинный, мы его рассматривать не будем). Существуют функции, отслеживающие движения мыши, имеется функция, отвечающая за создание окон. Вы можете использовать функции, устанавливающие положение окна, а также функцию, задающую его размер. Имея в своем распоряжении все эти функции, нам недостает лишь одного — возможности записи состояния окна в базу данных. Для реализации соответствующей функции нам потребуется Ajax.

Адаптация библиотеки DHTML под использование Ajax

Функции библиотеки DHTML, реализующие перетаскивание и изменение размеров окон, используют множество обработчиков событий и методов DOM для преодоления несогласованности браузеров. Перетаскивание и изменение размеров окон завершается при отпускании кнопки мыши. Следовательно, нам нужна функция, вызываемая обработчиком события onmouseup в файле AjaxWindow.js. Эта функция содержит приведенный ниже код, который выполняется после отпускания кнопки мыши.

```
document.onmouseup = function(){
    bDrag = false;
    bResize = false;
    intLastX = -1;
    document.body.style.cursor = "default";
    elemWin="";
    bHasMoved = false;
}
```

В данном коде множество булевых переменных получает **значение false,** что обозначает отмену соответствующих действий. Курсор возвращается к значению по умолчанию. Нам необходимо изменить строку, в которой отменяется ссылка elemWin. Сейчас нам требуется взять эту ссылку и передать ее другой функции, чтобы инициализировать объект XMLHttpRequest и передать информацию на сервер.

Хотя иногда адаптация библиотек под нужды текущего проекта требует длительного процесса проб и ошибок, в данном случае требуемые функциональные возможности достаточно очевидны. Просто добавьте в код обработчика события onmouseup строку, выделенную полужирным шрифтом.

```
document.onmouseup = function(){
    bDrag = false;
    bResize - false;
    intLastX * -1;
    document.body.style.cursor - "default";
    if(elemWm && bHasMoved)SaveWindowProperties(elemWin);
    bHasMoved = false;
}
```

Полужирная строка в приведенном фрагменте кода проверяет, произошло ли перемещение или изменение размеров объекта и существует ли еще элемент. Если пользователь не выполнил ни одного из указанных действий, то нет никаких причин отправлять запрос на сервер. Если же действие было выполнено, мы передаем ссылку на элемент функции SaveWindowProperties () которая инициирует запрос на сервер.

Получение свойств активизированного элемента

После того как пользователь переместил элемент или изменил его размеры, необходимо обновить информацию на сервере в соответствии с новыми параметрами. Для размещения элементов и задания их ширины и высоты библиотека окон DHTML использует CSS. Это означает, что нам требуется всего лишь получить идентификатор базы данных, координаты и размер окна. Координаты и размер можно получить, изучив параметры CSS, соотнесенные с окном, в текущий момент находящимся в фокусе. Затем эти новые параметры можно отправить на сервер, чтобы с помощью Ajax записать в базе данных (листинг 11.10).

р;Листинг;11:10,;Функция **Sa&ff&SowPropertiesO** ' ^ Ч / -*Щ*

```
function SaveWindowProperties(){
    winProps - "ref=" +
        elemWin.id;
//О Получить идентификатор окна
    winProps +• "&x=" +
// © Найти положение окна
        parselnt(elemWin.style.left);
// © Выяснить размер окна
    winProps += "&y=" +
        parselnt(elemWin.style.top);
    winProps += "&w=" +
        parselnt{elemWin.style.width);
    winProps +•= "&h=" +
        parselnt(elemWin.style.height);
// © Вызвать функцию Settings
    Settings("saveSettings",winProps);
    elemWin = "";
// © Удалить ссылку на элемент
```

Как показано в листинге **11.11,** мы получаем идентификатор окна О, обращаясь к объекту окна. Полученный идентификатор был присвоен окну при его создании библиотекой. При присвоении идентификатора библиотека добавляет win перед номером из столбца id базы данных (это видно из кода JavaScript создания окна).

Координаты x и y левого верхнего угла окна находятся © с помощью свойств left и top таблицы стилей. Кроме того, мы используем таблицы стилей для получения размера © окна, обращаясь к его свойствам width И height.

Получив информацию, мы можем вызывать функцию Settings () (ее код приводится ниже) для отправки запроса на сервер О. Вызвав эту функцию, мы должны удалить из глобальной переменной elemWin объект, сопоставленный с элементом ©. Для этого переменной elemWin присваивается значение, равное пустой строке. Завершив работу над функцией SaveWindowProperties (), можно инициировать тихий запрос Ajax на сервер с помощью функции JavaScript Settings ().

11.5.2. Автоматическая запись информации в базе данных

Ajax позволяет так отправлять информацию на сервер, чтобы пользователь об этом и не подозревал. Такой подход реализован в двух проектах, рассмотренных в данной книге. Мы можем легко отправлять запросы на сервер, отслеживая нажатие клавиш, как в сценарии опережающего ввода (см. главу 10), и движение указателя мыши (как в данной главе). Подобная невидимая передача информации очень удобна для разработчиков, поскольку мы можем обновлять пользовательские настройки, не требуя от него никаких дополнительных действий. В большинстве случаев, чем меньше приходится делать пользователям, тем больше они довольны. В данном приложении все, что нам требуется для инициации запроса XMLHttpRequest, — чтобы пользователь отпустил кнопку мыши. Таким образом, сейчас самое время рассмотреть процесс отправки запроса на сервер.

Клиент: отправка незаметного запроса

В данном случае процесс XMLHttpRequest не требует ничего сложного. Пользователь, взаимодействующий с формой, сам незаметно передает нашей функции все свойства формы. Итак, вначале нам необходимо инициализировать объект XMLHttpRequest.

```
function  Settings(xAction,xPararas){
    var url ° xAction + ".servlet";
    var strParams = xParams;
    var loader! - new net.ContentLoader(url,
                                        BuildSettings,
                                        ErrorBuildSettings,
                                        "POST",
                                        strParams);
}
```

Рассмотрим функцию Settings () подробнее. В качестве параметра ей передается активизированная строка, содержащая все свойства окна. В нее мы добавляем все параметры, отправляемые на сервер. Если обращение к серверу пройдет успешно, загрузчик вызовет функцию BuildSettings (). Если произойдет какой-то сбой, вызывается функция ErrorBuildSettings ().

```
function  BuildSettings(){
    strText = this.req.responseText;
    document.getElementById("divSettings").innerHTML = strText;
}
function  ErrorBuildSettings(){
    alert(tquoteThere was an error trying to connect
```

```
            to the server.tquote);
        document.getElementByld("divSettings").style.display =  "none";
    }
```

Приведенная выше функция BuildSettings () является очень условной — мы просто завершаем объект XMLHttpRequest, полученный с сервера. Еще мы можем поместить сообщение в строку состояния портала, указывающее, что • информация на сервере была обновлена. Если в процессе обновления информации возникли проблемы, в строку состояния можно добавить сообщение об ошибке. Кроме того, генерируется предупреждение, сообщающее пользователю об ошибке, однако это нарушает процесс взаимодействия с порталом. Готовый механизм уведомления был рассмотрен в главе 6. его интеграцию в систему портала предлагаем вам выполнить в качестве самостоятельного упражнения. Посмотрим теперь, что происходит на сервере.

Сервер: сбор информации с клиента

Все, что нам осталось сделать, — извлечь значения, отправленные в форме. Эти значения были посланы объектом XMLHttpRequest, активизированным обработчиками событий onmouseup. Теперь нам необходимо создать запрос SQL, содержащий эту информацию, и обновить запись в базе данных, записав в нее новые данные. Для этого мы определим класс UpdateServlet, показанный в листинге 11.11.

НЦПистинг 11.11. Класс Upda^erviet. *3^ШШШШШШШШШЩШЯЩ*

```
public class UpdateServlet extends HttpServlet {
    protected void doPost{
        HttpServletRequest  request,
        HttpServletResponse response
    )throws. ServletException,  IOExceptionf
        String windowld*
// О Извлечь из запроса уникальный идентификатор
            request.getParameter("ref");
        HttpSession  session=request.getSession();
        PortalWindow  window=(PortalWindow)

// © Извлечь из сеанса объект Window
            (session.getAttribute
                {"window_"+windowld));
        window.setXPos(getlntParamfrequest,"x"));
        window.setYPos(getlntParamfrequest,"y"));
        window,setWidth(getIntParam(request,"w"))  ;
        window.setHeight(getIntParam(request,"h"));

// в Записать изменения
        DBUtil.savePortalWindow(window);
        Writer  writer=response.getWriter();

// Вернуть простой текстовый отклик
        writer.write("Save  Complete");
        writer.flush();
    J
    private  int  get!ntParam(HttpServletRequest  request,
```

```
            String param) {
            String str=request.getParameter(param) ;
            int result=Integer.parselnt(str);
            return result;
        }
_ }
```

Используя идентификатор окна как параметр запроса О, мы можем извлечь из сеанса объект PortalWindow © и обновить его геометрию, опираясь на другие параметры запроса. Затем на объекте DBUtil вызывается другой метод, записывающий настройки окна портала в базе данных ©. Как и ранее, реализация, представленная в листинге 11.12, очень проста, и ее легко перевести на другие языки.

```
public static void savePortalWindow(PortalWindow window){
    Connection conn=getConnection();
    int x=window.getXPos();
    int y=window.getVPos();
    int w=window.getWidth{};
    int h=window.getHeight();
    int id=window.getld();
    String sql="UPDATE portal_windows SET xPos="+x
        +", yPos="-t-y
        +",width="+w
        +",height="+h
        +" WHERE id="+id;
    try{
        Statement stmt=conn.createStatement();
        stmt.execute(sql);
        stmt.close();
    }catch (SQLException sqlex){
    }
>
```

Код, приведенный в листинге 11.12, достаточно логичен. Мы считываем нужные детали из объекта PortalWindow и соответствующим образом создаем обновляющее выражение SQL. В данном случае мы будем возвращать не JavaScript-код, а простое текстовое подтверждение.

Чтобы проверить новые функциональные возможности, войдем на портал в качестве предполагаемого пользователя. Перетащим окна по экрану и изменим их размеры, чтобы изменить настройки, заданные по умолчанию. Закроем браузер, инициировав завершение сеанса. При повторном открытии браузера и входе в систему (под тем же именем) видим, что окна находятся в том же положении, в котором мы их оставили. Переместим окна в новое положение и изучим таблицу базы данных. Видим, что настройки пользователя записываются автоматически, причем пользователь об этом даже не подозревает.

Итак, мы сформировали все базовые функциональные возможности для рабочей системы портала, в том числе несколько элементов, которые классические Web-приложения просто не могли бы дать. Имеется еще несколько элементов, которые можно отнести к категории полезных, например, возможности добавления, удаления и переименования окон. Из-за ограниченного размера книги мы не будем их рассматривать. Впрочем, вы можете загрузить полный код приложения-портала, в котором можно добавлять, удалять, переименовывать и настраивать свойства окон, не покидая единственной страницы портала. Если у вас возникнут какие-либо вопросы, касающиеся приведенного кода, или вам потребуется более подробное его объяснение, вы всегда можете связаться с авторами через www.manning.com.

Разработанный код можно назвать грубым, однако он эффективен, так что мы можем демонстрировать работу его отдельных компонентов. Пожалуй, пришло время передать его группе, ответственной за реструктуризацию, и посмотреть, как связать все элементы воедино и облегчить повторное использование системы.

11.6. Реструктуризация

Как вы видели, концепция основанного на Ajax клиента портала, взаимодействующего с серверной частью портала, является довольно убедительной. В разделе, посвященном реструктуризации клиентской части кода мы рассмотрим наш компонент как объект, выступающий в роли арбитра команд портала, отправляемых менеджеру портала на сервере. В процессе реструктуризации мы постараемся выделить те части кода, которые могут со временем меняться, и максимально облегчить эти изменения. Поскольку портал представляет собой крупномодульный компонент, а мы в какой-то степени можем распоряжаться содержимым страницы, нас не ограничивает требование не вмешиваться в HTML-разметку страницы (как в двух предыдущих проектах).

Однако, прежде чем рассматривать семантику клиентской части сценария, остановимся на контракте с сервером. Наша предыдущая реализация серверного кода была написана на Java, поэтому возможности аутентификации обеспечивал сервлетный фильтр(один сервлет возвращал конфигурации окна, второй — записывал конфигурации окна). Подобным образом, для добавления новых окон и удаления текущих мы создадим дополнительные автономные сервлеты. В Web-приложениях Java сервлеты можно достаточно гибко отобразить в URL; данная возможность определена в файле web.xml, содержащемся в загружаемом Web-архиве (.war). Например, функция SelectServlet, возвращающая сценарий, определяющий исходные окна, была отображена в URL portalLogin.servlet.

Одним из достоинств инфраструктуры Ajax является слабая связь между клиентом и сервером. В нашем примере портала в качестве внутреннего интерфейса используется Java, но нам совсем не обязательно привязываться к таким особенностям Java, как сервлетные фильтры и гибкая перезапись URL. Альтернативная внутренняя архитектура может ис-

пользовать диспетчер запросов, при которой единственный сервлет (страница PHP или ресурс ASP.NET) принимает все входящие запросы, а затем считывает параметр, задающий тип выполняемого действия (специфика передачи параметра определяется методом: GET или POST). Например, строка параметров для входа на портал может выглядеть как action=login&userid=userspassword=password. Используя Java, мы можем реализовать подход с помощью диспетчера запросов, присваивая определенный префикс URL, например .portal, сервлету диспетчера, что позволит записывать такие URL, как login.portal.

В реструктуризированном компоненте мы обобщим предположения относительно внутренней архитектуры, разрешив либо архитектуру диспетчера запросов, либо вариант с множественными адресами, использованный в реализации Java. Тем не менее от нас не требуется полная гибкость, поэтому мы предопределим набор команд, предположительно понятных для внутренней архитектуры портала и охватывающих регистрацию, отображение для пользователя окон портала, а также добавление и удаление окон из портала. Учитывая указанные изменения серверной части приложения, вернемся к реализации клиентской части.

Обсуждение реструктуризации портала начнем с переопределения контракта использования с точки зрения HTML-кода страницы, а затем приступим к реализации. Напомним, что введение HTML-страницы в сценарий портала происходит посредством регистрации, а точнее — с помощью кнопки входа в систему.

```
<input type="button" name="btnSub" value="login"
       onclick="LoginRequest(tquotelogintquote)">
```

Мы изменили обработчик события onclick, теперь это вызов функции, которая будет использовать наш компонент-портал. Предположим, что этот компонент обрабатывается с помощью сценария, выполняемого после загрузки. Характерный пример того, как это должно выглядеть, приведен в листинге 11.13.

Листинг 11.13. Создание портала и вход в систему

```
function createPortal() {
    myPortal = new Portal(
//O Основной URL портала
        tquoteportalManagertquote,
        {
// © Необязательные параметры
            messageSpanId: tquotespanprocessingtquote,
            urlSuffix: tquote.portaltquote }) ;
// © Вызвать для загрузки окон
    myPortal.loadPage(Portal-LOAD_SETTINGS_ACTION);
    document.getElementById(tquoteusernametquote).focus{) ;
}
function login() {
myPortal.login(document.getElementById(tquoteusernametquote).value,
               document.getElementById(tquotepasswordtquote).value);
```

В такой семантике использования функция createportal(), которая должна вызываться сразу после загрузки страницы, создает экземпляр компонента портала. Первый ее аргумент — это основной URL серверного приложения портала О, второй предоставляет необязательные параметры, используемые для ее настройки под конкретный контекст 0. В данном случае мы сообщаем функции идентификатор элемента DOM, в который следует записывать сообщения о состоянии, и имя параметра запроса, который определит совершаемое действие. После создания на портале вызывается API loadPage, загружающий окна портала, если в сеансе сервера присутствует регистрационное имя пользователя ©. Если в систему никто не входил, сервер возвращает пустой сценарий, оставляя на экране только форму регистрации.

Функция login() представляет собой вспомогательную функцию, вызывающую метод login{) компонента портала, передающую в качестве аргументов имя пользователя и пароль. Согласно данному контракту обработчик событий onclick кнопки регистрации вызывает метод login() страницы.

```
<input type="button" name="btnSub" value="login" onclick="login()">
```

11.6.1. Определение конструктора

Разобравшись с использованием компонента с точки зрения страницы, займемся реализацией логики. Начнем с конструктора.

```
function Portal( baseUrl, options ) {
    this.basellrl = baseUrl;
    this.options - options;
    this.initDocumentMouseHandler();
```

В качестве первого аргумента приведенный конструктор принимает URL средства управления порталом Ajax на сервере, а в качестве второго — объект опций, используемый при определении конфигураций. Напомним, что в первом варианте данного сценария использовался сервлетный фильтр и два сервлета, выполняющих внутреннюю обработку. Далее мы будем предполагать, что все запросы к внутренней части портала будет перехватывать один сервлет, или ресурс portalManager (см. листинг 11.13). Если нам требуется сконфигурировать портал с учетом внутренней архитектуры, не использующей единственный диспетчер запросов, мы можем передать конструктору различные аргументы, например:

```
myPortal = new Portal(
    tguotedatatquote,
    { messageSpanld: tquotespanProcessingtquote,
        urlSuffix: tquote.phptquote }
);
```

Таким образом, мы передадим основной URL "данных" и, поскольку в массиве опций определен параметр actionParam, добавим к пути URL команду с суффиксом .php, получая в результате URL, подобный data/login,php. Следовательно, мы получаем всю необходимую на данный момент гибкость. Вопрос превращения опций в URL будет рассмотрен в разделе 11.6.3. Сейчас же мы перейдем к следующему заданию. Итак, из по-

следней строки конструктора следует необходимость адаптации библиотеки AjaxWindows.js.

11.6.2. Адаптация библиотеки AjaxWindows.js

Напомним, что реализация данного портала использует внешнюю библиотеку AjaxWindows.js для создания отдельных окон портала и управления их размером и положением на экране. В связи с этим нам, в частности, требуется адаптировать библиотеку для отправки запросов Ajax менеджеру портала, чтобы записать настройки после события mouseup. Мы отслеживаем данное действие потому, что теоретически им заканчиваются все операции перемещения и изменения размеров. Первое, что мы сделали для выполнения данной адаптации, — скопировали код библиотеки AjaxWindows.js и изменили в нем фрагмент, помещающий в документ обработчик событий mouseup. Если рассматривать библиотеку AjaxWindow.js как продукт стороннего производителя, то недостатки данного подхода очевидны. Мы отошли от кода чужой библиотеки, т.е. модифицировали исходный код и поведение библиотеки так, что они перестали быть совместимыми с версиями, поддерживаемыми авторами библиотеки. Если библиотека изменится, нам придется согласовывать его со своими изменениями при выходе каждой следующей версии. Мы ничего не сделали, чтобы изолировать место изменения и сделать его как можно более "безболезненным". Поэтому рассмотрим менее радикальный подход к адаптации и выясним, можно ли как-то исправить данную ситуацию. Напомним, что последняя строка конструктора выглядела следующим образом:

```
this.initDocumentMouseHandler();
```

Метод initDocumentMouseHandler() представляет оперативную адаптацию библиотеки AjaxWindows.js. Он, как и ранее, просто перезаписывает обработчик document.onmouseup, но делает это уже в нашем собственном коде. Теперь логика, требуемая для адаптации внутри метода портала handleMouseUp(), реализована в нашем методе (листинг 11.14).

Листинг 11.14. Адаптация обработчика AjaxWindows.js

```
initDocumentMouseHandler: function() {
    var oThis = this;
    document.onmouseup = function() { oThis.handleMouseUp(); };
Ь
handleMouseUp: function() {
    bDrag = false;
    bResize = false;
    intLastX = -1;
    document.body.style.cursor = "default";
    if { elemWin && bHasMoved )
            this.saveWindowProperties(elemWin.id);
    bHasMoved = false;
Ь                                .                                Л
```

Это решение уже гораздо лучше, но это еще не все. Если библиотека AjaxWindows.js определяет обработчик mouseup в именованной, а не в ано-

нимиой функции, то этот обработчик можно сохранить под другим именем и вызывать из нашего собственного обработчика. Таким образом, мы не будем дублировать логику, уже определенную в библиотеке AjaxWindows.js. Данный подход иллюстрируется в приведенном ниже коде.

```
function ajaxWindowsMouseUpHandler() {
    // logic here...
}
document.onmouseup = ajaxWindowsMouseUpHandler;
```

Функция ajaxWindowsMouseUpHandler () представляет собой обратный вызов, определенный внешней библиотекой AjaxWindows.js. Как показано ниже, ее применение позволит сохранить определение метода и использовать его позже.

```
initDocumentMouseHandler: function() {
    this.ajaxWindowsMouseUpHandler =
        aj axWindowsMouseUpHandler;
// О Сохранить нашу собственную ссылку
    var oThis = this;
    document.onmouseup = function() { oThis.handleMouseUp(); };
Ь
// © Вызвать библиотечную функцию
handleMouseUp: function() {
    this.aj axWindowsMouseUpHandler();
// © Добавить функциональные возможности Ajax
    if { elemWin && bHasMoved )
        this.saveWindowProperties(eleraWin.id) ;
>,
```

Теперь наш метод handleMouseUp () не должен дублировать функциональные возможности библиотеки AjaxWindows.js. Необходимые возможности мы вызываем © посредством записанной ссылки О, а затем добавляем функциональные возможности Ajax ©. Если же обработчик mouseup библиотеки Ajax Windows в будущем изменится, то эти изменения не потребуют модификации нашего кода. Это уже более приятная ситуация с точки зрения управления изменениями. Разумеется, она предполагает, что подразумеваемый контракт с библиотекой не изменится — имеются в виду две глобальные переменные, elemWin и HasMoved. Поскольку в текущий момент библиотека определяет обработчик mouseup как анонимную функцию, мы по-прежнему можем записать ссылку на существующую функцию обработки события mouseup, используя следующую строку кода:

```
this.ajaxWindowsMouseUpHandler - this.document.onmouseup;
```

Таким образом, мы добиваемся того же результата, что и ранее, но теперь решение гораздо изящнее, поскольку в данной ситуации контракт гораздо слабее. Приведенное решение основывается на том, что мы включили наши эиблиотеки сценария в правильном порядке, т.е. библиотека AjaxWindows.js уже выполнила код, помещающий в документ обработчик mouseup. Кроме то-^о, предполагается, что никакая другая библиотека не поместила в документ зругой обработчик событий mouseup или реализовала другой интерфейсный подход, подобный нашему.

Пожалуй, это все, на что можно надеяться при адаптации библиотеки. Перейдем теперь к API портала. Изучая метод handleMouseUp(), можно догадаться об одной из трех команд портала, которые должен иметь компонент портала. При отпускании кнопки мыши вызывается метод saveWindowProperties(), записывающий состояние и положение текущего окна. Ниже мы подробно рассмотрим детали этого процесса, а также разберем другие API команд портала.

11.6.3. Задание команд портала

Как обсуждалось выше, наш компонент портала, в первую очередь, является отправителем команд. Отправляемые команды представляют собой запросы Ajax к серверной системе управления порталом. Понятие команд и формальную структуру Command в Ajax мы уже рассматривали в главах 3 и 5. Теперь мы изучим другую возможность использовать данные знания.

Итак, мы организовали в портале поддержку таких команд: регистрация, загрузка и запись настроек. Еще мы собираемся ввести возможность добавления и удаления окон, о которой мы уже упоминали, хотя и не показывали ее полную реализацию. Перечисленные возможности можно представлять как методы нашего портала. Однако, прежде чем мы начнем рассмотрение кода, выполним подготовительную работу, которая поможет нам в локализации изменений. Прежде всего, мы имеем в виду имена самих команд. Давайте определим символы для всех имен команд, чтобы их можно было использовать в любой части компонента. Рассмотрим следующий набор символов:

```
Portal.LOGIN_ACTION » "login";
Portal.LOAD__SETTINGS_ACTION - "PageLoad";
Portal.SAVE_SETTINGS_ACTION = "UpdateDragWindow";
Portal.ADD_WINDOW_ACTION - "AddWindow";
Portal.DELETE_WINDOW_ACTION - "DeleteWindow";
```

Хотя используемый язык и не поддерживает напрямую константы, предположим, что, используя договоренность о прописных буквах, можно считать указанные значения постоянными. Мы можем просто провести данные строковые литералы через наш код, но данный подход слишком неаккуратен. При подобном использовании констант наши "магические" строки будут располагаться в одном месте. Если контракт сервера изменится, мы сможем к этому приспособиться. Представим, например, каким образом может измениться контракт сервера (табл. 11.1).

Теперь, когда мы можем обращаться к командам с помощью указанных символов, рассмотрим общий механизм передачи серверу команд управления порталом. Нам потребуется вспомогательный метод, в общем виде отправляющий серверу Ajax-команды портала. Рассмотрим данный контракт использования.

```
myPortal.issuePortalCommand( Portal.SAVE_SETTINGS_ACTION,
  "setting1=" + setting1Value, "setting2=" + setting2Value, ... )-"
```

В данном сценарии мы рассматриваем метод issuePortalCommand(), принимающий в качестве первого аргумента имя команды (например, одну из

Таблица 11.1. Изменения открытого контракта

Изменение контракта сервера	Требуемое действие
Переименована команда (например, PageLoad заменена глагольной формой LoadPage)	Заменить правую часть оператора присваивания константы LOAD_SETTINGS_ACTION новым значением. Остальная часть кода не меняется
Сервер больше не поддерживает команду	Удалить контракт и выполнить глобальный поиск всех упоминаний команды. В каждом случае произвести необходимую модификацию кода
Сервер поддерживает новую команду	Добавить константу для этой команды и использовать ее имя в коде

наших констант) и переменное число аргументов, соответствующих параметрам, которые ожидает/требует команда. Как и следовало ожидать, параметры имеют точно такую же форму, которая требуется методом sendRequest() объекта net.ContentLoader. Определенный нами метод issuePortalCommand() можно реализовать следующим образом:

```
issuePortalCommand: function{ commandName ) {
//O Получить параметр действия
    var actionParam - this.options[tquoteactionparamtquote];
// © Получить суффикс URL
    var urlSuffix = this.options[tquoteurlSuffixtquoteJ;
    if ('urlSuffix) urlSuffix="";
    var url = this.baseUrl;
    var callParms •" [];
    if (actionParam){ callParms.push(
// © Применить параметр действия
    actionParam + "=" + commandName );
    }else{
// O Применить суффикс URL
        url += "/" + commandName + urlSuffix; }
    for ( var i = 1 ; i < arguments.length ; i++ )
        callParms.push( argumentsfi] );
    var ajaxHelper = new
// 0 Создать объект ContentLoader
        net.ContentLoader( this, url, "POST", [] );
    ajaxHelper.sendRequest
// 0 Отправить запрос
        ,apply( ajaxHelper, callParms );
),
```

Данный метод создает URL, основываясь на конфигурационных опциях, обсуждавшихся в разделе 11.6.1, Если мы установили значение actionParam O, оно будет добавлено к параметрам с помощью POST, передаваемым на сервер ©. Если нет, мы добавим команду к пути URL O, присоединяя суффикс URL, если он указан в опциях ©. Первый аргумент функции — имя команды. Все остальные аргументы рассматриваются как параметры запроса. Затем сформированный URL передается объекту ContentLoader ©, и, как показано в предыдущем примере, отправляется запрос со всеми предоставленными

параметрами ©. Благодаря данному методу все API команд нашего портала будут очень лаконичными . Еще один "бонус" подобного общего метода заключается в том, что мы можем поддерживать новые команды, появляющиеся на сервере, не меняя код клиентской части приложения. А теперь рассмотрим команды, которые мы уже знаем.

регистрация

Напомним, что обработчик событий onclick нашей кнопки регистрации инициирует вызов метода login() страницы, который в свою очередь вызывает указанный выше метод. Функция login (по крайней мере, с точки зрения сервера) представляет собой команду, которую сервер должен обработать, проверив предоставленную пользователем регистрационную информацию, а затем (если эта информация верна) ответив так же, как команда load-page. Учитывая сказанное, рассмотрим реализацию функции login(), показанную в листинге 11.15.

```
login: function(userName, password) {
    this.userName = userName;
    this.password - password;
    if ( this.options.messageSpanId )
        document.getElementById(
            this.options.messageSpanId).innerHTML =
            "Verifying Credentials";
    this.issuePortalCommand( Portal.LOGIN_ACTION,
     "user=" + this.userName, "pass=" + this.password );
},
```

Данный метод помещает сообщение "Verifying Credentials" в элемент span, определяемый настраиваемой опцией this.options.messageSpanId. Затем он отправляет команду login внутреннему коду портала, передавая регистрационную информацию, полученную методом в виде параметров запроса. Всю сложную работу выполняет метод issuePortalCommand{).

Загрузка настроек

Напомним, что функция createPortai() нашей страницы вызывает указанный выше метод для загрузки исходной конфигурации окон портала. Для загрузки настроек страницы применяется метод, который даже проще рассмотренного выше метода регистрации. Это просто удобная интерфейсная оболочка вокруг команды issuePortalCommand{). В качестве единственного своего параметра, используемого сервером для загрузки нужных настроек окон, она передает имя пользователя.

```
loadPage: function(action) {
    this.issuePortalCommand( Portal.LOAD_SETTINGS_ACTION,
          "user=" + this.userName, "pass=" + this.password ); },
```

Запись настроек

Метод записи настроек также достаточно прост. Напомним, что данный метод вызывается адаптированной библиотекой AjaxWindows.js при наступлении события mouseup и производит запись всех операций перемещения и изменения размеров.

```
saveWindowProperties: function(id) {
    this.issuePortalCommand( 'portal.SAVE_SETTINGS_ACTION,
            "ref=" + id, "x=" + parseInt(elemWin.style.left),
            "y=" + parseInt(elemWin.style.top),
            "w=" + parseInt(elemWin.style.width),
            "h=" + parseInt(elemWin.style.height) );
    elemWin = null; },
```

Добавление и удаление окон

Хотя мы не полностью разработали концепцию добавления и удаления окон (по крайней мере, с точки зрения пользовательского интерфейса, удобного для инициации этих действий), мы можем определить командные методы API, поддерживающие указанные операции.

```
addWindow: function(title, url, x, y, w, h) {
    this.issuePortalCommand( Portal.ADD_WINDOW_ACTION,
                "title-" + title, "url-" + url, "x=" + x,
                "y=" + y, "w=" + w, "h=" + h ); },
deleteWindow: function(id) {
    var doDelete =
        confirm("Are you sure you want to delete this window?");
    if(doDelete) this.issuePortalCommand(
          Portal.DELETE_WINDOW_ACTION, "ref=" + id ); },
```

Данный метод завершает обсуждение программных интерфейсов, требуемых для поддержки команд портала. Рассмотрим теперь обработку Ajax на портале.

11.6.4. Обработке средствами Ajax

Как уже отмечалось, в данном примере мы используем технологию Ajax для обработки ответов. В частности здесь реализуется взаимодействие, ориентированное на сценарии. Описанная технология основана на том, что ожидаемый ответ сервера представляет собой приемлемый код JavaScript. При таком подходе очень желательно, чтобы клиенту для понимания ответа не требовалось выполнять какую-либо сортировку или синтаксический анализ. Ответ вычисляется с помощью метода JavaScript eval(), и в дальнейшем с клиента снимается вся ответственность. Недостаток данного подхода состоит Б ТОМ, что вся ответственность возлагается на сервер, который отвечает за понимание клиентской объектной модели и генерирует синтаксически верный ответ, согласующийся с требованиями языка (JavaScript). Второй недостаток описанного подхода частично устраняется с помощью использования для определения откликов популярной разновидности рассматриваемой технологии — JSON. Существуют определенные серверные библиотеки, помогающие гене-

рировать отклики JSON (см. главу 3), хотя они ближе к тому, что в главе 5 называлось подходом, ориентированным на данные.

Пока же мы собираемся придерживаться взаимодействия, ориентированного на сценарии, поэтому сейчас обратимся к нашей реализации и посмотрим, что можно сделать для его развития. Начнем с функции ajaxupdate() и вспомогательной функции runScript().

```
ajaxUpdate:    function(request){this.runScript(request.responseText);'
runScript: function(scriptText){eval(scriptText);},
```

Как обсуждалось выше, обработка ответа слишком проста. Все, что мы делаем, — это вызываем метод runScriptO с параметром responseText ("текст ответа"), и runScript() применяет функцию eval() к тексту отклика. У вас может возникнуть вопрос: "Почему бы вообще не избавиться от метода runScript() и просто вызывать eval() из метода ajaxUpdate()?" Да, это действительно допустимый и полезный подход. Тем не менее иногда удобно иметь метод, инкапсулирующий концепцию запуска сценария. Например, что будет, если мы добавим в реализацию runScript() этап предварительной или последующей обработки? Опять же мы изолировали место изменения. К счастью, метод ajaxUpdate() не замечает изменения, и мы получаем новое поведение. Одной из интересных сфер применения описанной технологии может быть препроцессор, выполняющий замещение значений, перед выполнением функций, располагающихся на стороне клиента.

В завершение обсуждения обработки Ajax рассмотрим первостепенную по важности тему обработки ошибок. В связи с этим напомним, что метод handleError(), так же, как и метод ajaxUpdate(), является неявным контрактом, требуемым для сообщения с net.ContentLoader. Реализация метода handleError() приведена ниже.

```
handleError: function(request) {
    if (this.options.messageSpanId) document.getElementById
            (this.options.messageSpanId).innerHTML «•
            "Oops! Server error. Please try again later.";}.
```

Данный метод проверяет существование конфигурационного свойства messageSpanId и при его наличии использует его в качестве элемента, отображая сообщение "Oops!" в пользовательском интерфейсе. Фактический текст сообщения также можно представить как параметр с помощью объекта опций. Сделать это предлагается читателям в качестве самостоятельного упражнения.

Таким образом, мы завершаем разговор о реструктуризации компонента-портала. Мы создали обманчиво простой механизм, поддерживающий управление порталом на основе Ajax. Потратим несколько минут и подытожим, чего же мы добились в ходе реструктуризации.

11.6.5. Выводы

По нескольким параметрам разработка данного компонента достаточно отличается от всех остальных примеров, приведенных в данной книге. Во-первых, мы разработали крупномодульный компонент, предоставляющий возможно-

сти портала на основе Ajax. Вряд ли какой-нибудь разработчик пожелает поместить систему-портал в угол своей страницы! Во-вторых, мы использовали технологию обработки откликов Ajax в виде кода JavaScript. При реструктуризации данного компонента мы старались изолировать точки изменения. Это было проиллюстрировано несколькими способами.

- Мы создали понятный способ адаптации библиотеки AjaxWindows.js.
- Изолировали строковые литералы как псевдоконстанты.
- Написали общий метод генерации команд.
- Концепцию запуска ответного сценария Ajax изолировали с помощью метода.

11.7. Резюме

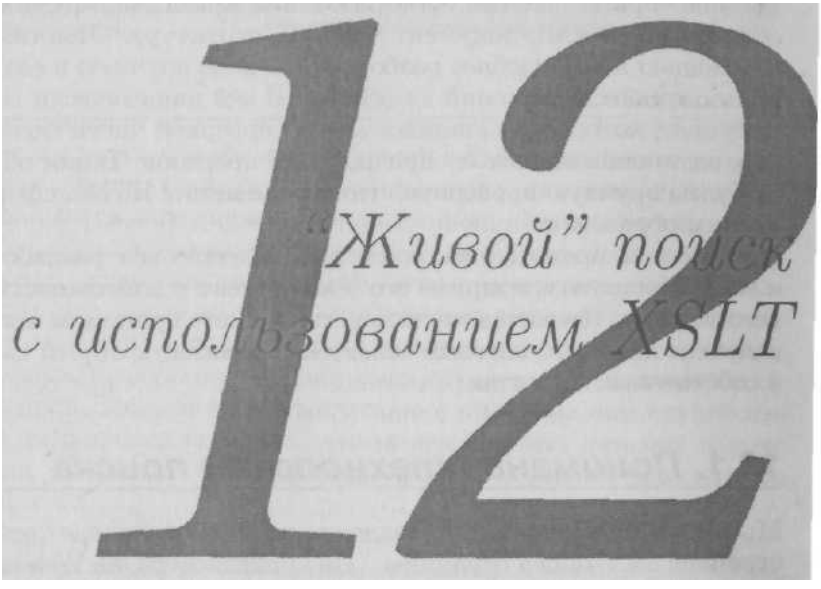
Портал — это, пожалуй, один из наиболее мощных инструментов, которые могут находиться в распоряжении компании. Компания может настроить бизнес-логику, позволяющую пользователям просматривать только ту информацию, которая их касается. Кроме того, порталы разрешают пользователям настраивать внешний вид окна согласно собственным предпочтениям, что позволяет повысить производительность труда, поскольку страница выглядит точно так, как того желает пользователь.

Используя Ajax для реализации портала, можно сохранить все функциональные возможности в одной области, не требуя отправки на сервер множества страниц. Теперь при выходе пользователя в систему уже не нужно думать, какое действие соотнесено с кнопкой "Назад". Истории прохождения страниц уже нет, поскольку мы никогда не покидаем единственную страницу. Мы говорили о недостатках выхода со страницы, но данная проблема решается с помощью запроса сервера с использованием Ajax.

Кроме того, мы отправляем запросы на сервер, не уведомляя пользователя о записи данных. Инициируя Ajax посредством обработчиков событий, мы можем записывать данные быстро, не нарушая взаимодействие пользователя с порталом. Портал, использующий Ajax, переводит богатый пользовательский интерфейс на новый уровень производительности.

В последнем разделе данной главы мы рассмотрели реструктуризацию кода портала, а в предыдущих разделах сосредоточили внимание на создании компонента с возможностью повторного использования, который можно поместить на существующую страницу. В данном случае такой вариант не подходит, поскольку портал представляет собой оболочку, в которой располагаются все остальные компоненты. При реструктуризации мы акцентировали внимание на повышении удобства эксплуатации кода, изолируя строковые константы, создавая общие методы и более четко отделяя библиотеки сторонних производителей от нашего кода.

В данной главе мы генерировали простые ответные XML-документы от сервера и вручную декодировали их с использованием JavaScript. В следующей главе мы рассмотрим альтернативный подход: использование таблиц стилей XSLT на стороне клиента для преобразования абстрактного XML-кода непосредственно в HTML-разметку.



В этой главе.

- Технологии динамического поиска
- Использование XSLT для трансляции XML в HTML
- Закладки на динамическую информацию
- Создание компонента "'живого" поиска

Инфраструктура Ajax позволяет сделать так, чтобы во время выполнения сервером трудоемких операций пользователь не терял контроль над тем, что происходит на стороне клиента. Если процесс обработки довольно длительный, пользователю можно предоставить анимированное GIF-изображение уведомляющее его о происходящем. Кроме того, во время выполнения серверного процесса пользователь сможет производить другие действия, при этом не думая, что браузер "завис".

В данной главе мы используем данную технологию Ajax для создания "живого" поиска. С помощью XSLT (Extensible Stylesheet Language Transformations — расширяемые преобразования языка таблиц стилей) мы будем преобразовывать XML-документ в HTML-структуру. Отметим, что трансляция с помощью XSLT удобнее разбора XML-кода вручную и создания HTML с использованием выражений JavaScript. В ней динамически генерируется XML-документ, которым мы заменяем код серверной части сценария и JavaScript-код, составлявший основу предыдущих проектов. Таким образом, мы больше не будем вручную проверять, что все элементы HTML сформированы надлежащим образом.

Как и в предыдущих примерах, вначале мы разработаем код "в лоб", а затем реструктуризируем его в компонент с возможностью повторного использования. Прочитав эту главу, вы поймете принципы использования XSLT с Ajax и получите готовый компонент поиска, который сможете применять в собственных проектах.

12.1. Понимание технологий поиска

Мы уже привыкли, что выполнение поиска на сервере "дополняется" демонстрацией застывшей страницы. (По крайней мере, на Web-сайтах, не располагающих кластером из 1 200 серверов, менее чем за секунду выполняющих поиск на 8 миллиардах страниц.) Чтобы убрать паузу, некоторые разработчики создают всплывающие окна и фреймы. Иногда для обработки используется дополнительное окно, что приветствуется пользователями, но также создает определенные проблемы. Используя Ajax, мы можем избежать использования фреймов и привычных задержек в классических операциях отправки форм.

12.1.1. Классический поиск

Рассмотрим классический процесс поиска. Включив форму поиска в Web-сайт, мы предполагаем отправлять один или несколько элементов на сервер для дообработки. В качестве примера можно привести основную страницу поиска Google. Эта страница (www.google.com) содержит единственное текстовое окно и две кнопки поиска. В зависимости от выбранного действия форма либо перенаправит нас на список записей (с помощью которых мы можем переходить на требуемые страницы), либо переведет на страницу, соответствующую одной из позиций названного списка. Подобная структура прекрасно подходит для страницы, с которой не связаны никакие другие функциональные возможности, однако если поиск является частью большого

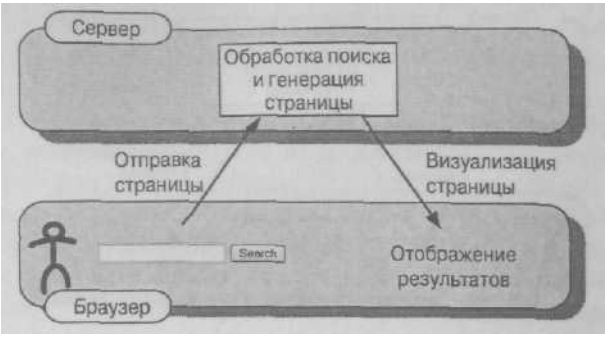


Рис. 12.1. Классическая модель поиска с обозначенными этапами обработки

проекта, описанное решение может порождать проблемы — потерю состояния страницы, очистку полей формы и т.д. На рис. 12.1 приведена диаграмма классической модели поиска, когда вся страница отправляется на сервер для обработки, а возвращается совершенно новая страница с результатами

Одним из источников задержек является то, что запросы к базе данных могут обрабатываться довольно долго. Действия с браузером недоступны пользователю до тех пор, пока не будут отображены результаты, при этом страница кажется "зависшей". Иногда разработчики пытаются как-то бороться с данным периодом неактивности, например, уведомлять пользователя о том, что процесс идет. Здесь важно отметить, что проблема недоступности не ограничивается операциями поиска. С ней можно столкнуться при обновлении или удалении записей из базы данных, запуске сложной транзакции на стороне сервера и т.д.

В качестве средства борьбы с описанным явлением разработчики используют анимированные GIF-изображения (например, заменяющие собой строку состояния), вызываемые в то время, когда сервер обрабатывает отправленный запрос. Вопрос о том, как это можно сделать, является одним из самых популярных на различных форумах, подобных JavaRanch (www.JavaRanch.com). Однако проблемой анимированных GIF-изображений является то, что они запускаются не всегда. В Microsoft Internet Explorer GIF-анимация часто замирает на первой картинке, не демонстрируя циклически предполагаемый набор изображений. Разработчики знают, что некоторые пользователи, не видя анимации, считают, будто браузер "завис", и щелкают на кнопке обновления или закрывают браузер.

Классическая форма поиска подвержена тем же проблемам, которые описывались в предыдущих примерах, требовавших повторной визуализации страницы. Из-за загрузки новой страницы, которая отображается в начале, а не в том месте, до которого была прокручена на момент инициации обновления, информация о прокрутке страницы может теряться. Данные, введенные в поля формы, могут исчезать, и пользователю придется вводить их заново. Разработчики пытаются решать эти проблемы с помощью фреймов и всплывающих окон, но-прл этом порождают еще больше проблем. Что ж, давайте посмотрим, почему это происходит.

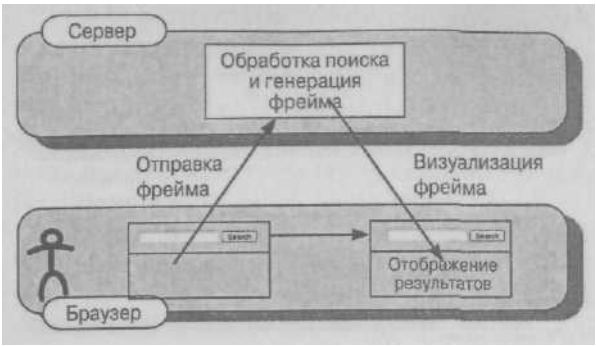


Рис. 12.2. Процесс поиска с использованием фреймов

12.1.2. Недостатки использования фреймов и всплывающих окон

Для решения проблем кажущегося "зависания" страниц, потери положения прокрутки и т.п. разработчики традиционно применяют фреймы (обычные и IFrame) и всплывающие окна. Фреймы и всплывающие окна позволяют продолжать обработку в другой части Web-страницы, так что пользователь может работать с фрагментом формы, инициировавшим обработку. Причем не только пользователь может манипулировать формой, параллельно могут выполняться и другие функции JavaScript.

Фреймы и всплывающие окна имеют и другие преимущества. Использование фреймов позволяет прокручивать возвращаемый набор записей, оставляя при этом элементы поисковой формы в поле зрения пользователя. Всплывающее окно позволяет отображать результаты в отдельном окне, вынося обработку из основного окна. Правильным образом организовав связь родительского окна с дочерним, можно при возврате результатов передавать данные из дочернего окна в родительское. Всплывающие окна невероятно полезны при внедрении поиска в больших формах, где пользователю требуется определенная трудно запоминаемая информация. Кроме того, можно сделать так, чтобы окно закрывалось после завершения обработки. Это полезно, когда требуется обновить страницу, не передавая на сервер никаких данных.

На рис. 12.2 показано, как реализован поиск во фрейме. Нижний фрейм отвечает за отправку поискового запроса на сервер, позволяя обрабатывать результаты. Поскольку поиск инициировал нижний фрейм, верхний фрейм окна по-прежнему доступен пользователю (в отличие от классической схемы поиска, показанной на рис. 12.1).

Хотя описанные подходы и решают одну проблему, они приводят к возникновению других. Фреймы были и остаются одним из самых страшных кошмаров разработчиков. Основная проблема, связанная с ними, — это навигация, поскольку мы не знаем, как фрейм будет взаимодействовать с браузером. Мы не знаем, как повлияет на фрейм кнопка "Назад". Вернет ли фрейм нас на нужную страницу, уничтожит ли всю структуру фреймов или просто не сработает? Как правило, именно такие вопросы возникают при тестировании. А что произойдет, если открыть страницу в браузере, не поддерживающем структуры фреймов? Чтобы избежать последней проблемы,

на страницу потребуется включить сценарий детектирования фреймов, из-за чего приложение станет более громоздким, управлять кодом станет сложнее, кроме того, возрастет его общая сложность.

С другой стороны, всплывающие окна можно заблокировать (что пользователи часто и делают). Вообще, всплывающие окна не должны представлять проблем, если они явно инициируются пользователем. Однако они также могут генерироваться браузером автоматически в ответ на событие onload или onunload, причем часто не допускается открывать такие окна, так как они, как правило, используются как реклама. Некоторые пользователи блокируют вообще все всплывающие окна — т.е. они никогда не получат результатов поиска, поскольку необходимое окно не откроется.

При использовании всплывающих окон могут возникнуть и другие проблемы, например, открытие дочернего окна под родительским, — в таком случае всплывающего окна просто не видно. Другая проблема встречается при выполнении некоторых действий в родительском окне. Если пользователь щелкает на ссылке или обновляет страницу, это действие может нарушить связь потомка с родителем, т.е. разрушить сообщение между окнами. При обновлении страницы объект всплывающего окна уничтожается; никаким разумным способом переносить объект со страницы на страницу нельзя.

Как видите, хотя использование фреймов и всплывающих окон решает проблемы, присущие традиционной отправке формы, эти решения могут порождать еще большие проблемы. Впрочем, для решения этих проблем мы можем использовать Ajax. Ajax обрабатывает связь с сервером независимо от страницы браузера, что позволяет воспроизводить анимацию и поддерживать состояние страницы; при этом не нужно заботиться о пользователях, которые блокируют всплывающие окна и закрывают окно, думая, что оно "зависло".

12.1.3. "Живой" поиск с использованием Ajax и XSLT

Функциональные возможности поискового элемента Web-сайта можно улучшить, сделав поиск "живым"; именно так некоторые разработчики называют поиск с использованием Ajax. Для выполнения такого поиска не требуется передавать на сервер всю страницу для обработки (как при традиционном поиске), а это означает, что можно поддерживать текущее состояние страницы. Кроме того, без особых проблем можно запустить JavaScript и GIF-анимацию, поскольку результаты отображаются в браузере с помощью innerHTML или других методов DOM.

Допустим, что у нас есть средство поиска, инициирующее длительную транзакцию базы данных, в ходе которой страница выглядит недоступной. Используя Ajax, можно запустить анимацию в начале транзакции. Когда придет время выводить результаты, свойству CSS display анимированного изображения можно присвоить значение none, и анимация просто исчезнет. Можно также поместить анимированное изображение там, куда планируется выводить результаты. После завершения транзакции изображение будет заменено результатами. В любом случае пользователь может использовать форму, в то время как объект XMLHttpRequest обрабатывает данные сервера.

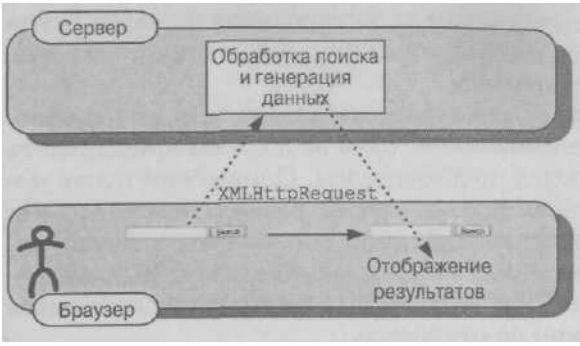


Рис. 12.3. Ход процесса при использовании инфраструктуры Ajax. Процесс, запущенный на стороне сервера, генерирует данные, которые код клиентской части приложения вставляет непосредственно на страницу. Такая реализация процесса требует меньшей полосы пропускания и имеет более дружественный пользовательский интерфейс

Рассмотрим Google Maps — популярный пример того, как пользователь может работать с приложением, пока на сервере выполняется обработка данных. Предположим, что мы отправляем серверу запрос о ресторанах на Главной улице и при этом продолжаем работать с картой, пока сервер его обрабатывает. Нам не нужно ждать, как при обычной отправке формы. Процесс, запущенный на стороне сервера, возвращает результаты на страницу, где они отображаются для пользователя. Точно так же "живой" поиск позволяет пользователю взаимодействовать со страницей, пока сервер обрабатывает данные. Ход данного процесса показан на рис. 12.3.

Использование Ajax для обработки поиска и длительных транзакций позволяет устранить проблемы, с которыми мы сталкивались в прошлом. Возможность "живого" поиска полезна не только при использовании поисковой машины, подобной Google или Yahoo!, но и при потребности в менее масштабном поиске. Например, с помощью "живого" поиска можно обратиться к таблице базы данных и извлечь информацию для одного из полей формы (например, адреса), основываясь на том, что уже ввел пользователь, при этом не мешая пользователю в это время заполнять другие поля. Любую длительную транзакцию с сервером можно превратить в "живой" процесс, когда сервер последовательно и ненавязчиво обновляет информацию, предоставляемую клиенту (см. главу 6). С помощью Ajax передачу данных можно сделать более эффективной и предоставлять результаты клиенту в более богатой среде.

12.1.4. Возврат результатов клиенту

Когда сервер готов вернуть клиенту результат "живого"' поиска, это можно сделать несколькими способами. Результаты можно отформатировать в виде XML-кода, обычного текста или HTML-дескрипторов. В предыдущих примерах на сервере создавался XML-документ. Затем на стороне клиента с помощью JavaScript-кода вызывались методы DOM XML, с помощью которых согласно последовательной обработке узлов XML формировалась таблица результатов. При таком подходе требовалось два цикла. Первый был задействован при формировании XML-докумецта на сервере, второй цикл применялся Для создания HTML-таблицы на стороне клиента.

Цикла DOM XML на стороне клиента можно избежать, если перед отправкой клиенту сформировать на сервере вместо XML-файла таблицу HTML. Используя такую технологию, мы соединяем дескрипторы HTML в большую строку, подобно тому, как мы формировали XML-документ. Однако теперь вместо дескрипторов XML применяются элементы таблицы. Строка HTML-кода возвращается клиенту, где ее можно присвоить свойству innerHTML элемента. В данном случае используется свойство responseText объекта XMLHttpRequest, поскольку нам не требуется проходить по узлам.

Недостаток описанной технологии заключается в том, что мы должны динамически обработать данные и создать таблицу (либо на стороне сервера, либо на стороне клиента). Если в будущем потребуется изменить формат таблицы, то в зависимости от сложности таблицы это может оказаться проблематичным. Добавление или удаление столбца может представлять проблему, поскольку мы должны будем изменить код внутри цикла. Кроме того, следует учесть, что в нашей строке содержатся кавычки; необходимо убедиться, что при создании строки они были представлены правильными управляющими последовательностями. Кроме того, если мы введем JavaScript-код в дескриптор HTML, то получим еще больше двойных и одинарных кавычек, с которыми нужно что-то делать, — необходимо проверить, что все дескрипторы правильно отформатированы и закрыты. Единственная возможность сделать все это — изучить текст после создания строки.

Чтобы избежать данных проблем, мы можем использовать XSLT. Применяя инфраструктуру Ajax, можно объединить XSLT-файл с XML-документом и отобразить результаты, не прибегая к использованию методов DOM. Если разработчик знает XSLT, но не является асом в использовании JavaScript, данное решение может быть превосходным.

Обсуждая поиск Ajax, следует отметить, что он не требует дообработки на сервере, следовательно, URL страницы не изменяется, чтобы соответствовать результатам поиска. Таким образом, создавая закладку на URL, мы не получим требуемых результатов поиска. В классических приложениях поиска, подобных Google, мы можем легко копировать URL со страницы, сформированной при поиске, вставить его в электронное письмо, и когда получатель щелкнет на такой ссылке, он увидит искомые результаты. Однако при Ajax-поиске такую возможность необходимо закодировать отдельно. Подробности этого процесса рассмотрены в разделе 12.5.4.

12.2. Код клиентской части сценария

Технология форматирования XML-данных с использованием XSLT довольно популярна, поскольку XML-файл обладает структурой, легко поддающейся обработке. В предыдущих проектах (например, в приложении опережающего ввода, рассмотренном в главе 10) мы использовали JavaScript, XML и DOM для создания отображаемого HTML-кода. В данном примере для получения того же эффекта мы будем использовать XSLT.

XSLT позволяет форматировать данные, формируя структуру HTML в другом файле и объединяя его с документом XML. Файл XSLT отвечает

за все, кроме навигации по узлам XML и построения таблиц, меню и HTML-структуры. Используя Ajax, мы можем извлечь статический или динамический файл XML и статический или динамический файл XLST с сервера, объединив их на стороне клиента для создания HTML-доку мента. Всю работу с XSLT можно выполнить и на стороне сервера, но мы будем рассматривать преобразования на стороне клиента.

12.2.1. Настройка клиента

В данном проекте мы рассмотрим поиск в телефонной книге по имени пользователя. Для этого используется одно текстовое окно и одна кнопка Отправить. Форма поиска показана в листинге 12.1.

Листинг 12.1. Форма клиентской части приложения

```
<form name="Forml" ID="Forml"
    onsubmit="GrabNumber();return  false;">
// O Добавить обработчик onsubmit
    Name: <input name="user" type="text"/>
// 0 Вставить текстовое окно
    <input type="submit" name="btnSearch"
        value="Search" />
// © Добавить кнопку отправки
    <br/Xbr/>
    <div id="results"></div>
// O Добавить элемент div для результатов
</form>
```

Для инициализации "живого" поиска к дескриптору form необходимо добавить обработчик событий. Обработчик событий onsubmit O перехватывает нажатие клавиши <Enter>, если указатель мыши расположен в текстовом окне, а пользователь щелкает на кнопке Отправить. Данный обработчик событий вызывает функцию GrabNumber(), инициирующую XMLHttpRequest без возврата формы на страницу. (В реальной ситуации необходимо проверить, не отключил ли пользователь JavaScript. В таком случае форма будет отправляться на сервер, и для поддержки подобных пользователей можно использовать классическую форму поиска. Впрочем, в данном проекте мы такую возможность не рассматриваем.)

Созданная нами форма является базовым вариантом, содержащим только обработчик событий, инициирующий XMLHttpRequest. Для сбора пользовательских критериев поиска к форме добавлены текстовое окно © и кнопка Отправить ©. Если мы хотим чего-то необычного, к текстовому окну можно еще добавить обработчик событий onblur, вызывающий функцию GrabNumber(); в таком случае поиск будет запущен тогда, когда текстовое окно перестает находиться в фокусе В данном примере активизация поиска связана с обработчиком событий onsubmit.

Затем в документ добавляется элемент div O, в который будут выводиться результаты поиска. Его можно разместить в любом месте станицы. в котором мы желаем видеть результаты. К div добавляется идентифика-

тор, поэтому мы можем ссылаться на этот элемент при добавлении результатов и GIF-анимации. Отметим, что использовать данный элемент для вывода результатов не обязательно. Их можно выводить в ячейку таблицы или даже в элемент span. Фактически можно использовать любой элемент HTML, свойством innerHTML которого мы можем манипулировать. Мы выбрали элемент div потому, что это блочный элемент, содержащий разрыв строки перед началом и после конца элемента. Кроме того, div занимает 100% доступной ширины браузера, из-за чего он больше подходит для выдачи пользователю больших таблиц результатов.

Важно отметить, что обработчик событий onsubmit должен возвращать значение false при выполнении обработчика. Это уведомляет браузер, что форму не нужно отправлять на сервер, что инициировало бы обновление всей страницы и прервало бы JavaScript-программу формы. Обработка возвращаемого значения приведена в листинге 12.2.

12.2.2. Инициализация процесса

В данном примере мы используем на сервере два файла: документ XML и документ XSL. Документ XML динамически создается PHP, когда этого требует клиент. PHP-код принимает то, что пользователь ввел на страницу, инициирует запрос к базе данных, а затем форматирует результаты в виде XML-документа. Статический документ XSL превращает наш динамический XML-файл в документ HTML. Поскольку документ XSL статический, его не требуется создавать серверу в момент клиентского запроса, это можно сделать заблаговременно.

Точно так же, как и в других проектах, рассмотренных в данной книге, для инициализации объекта XMLHttpRequest мы используем отдельную функцию. Процесс сбора необходимой информации и вызов функции представлен в листинге 12.2.

Листинг 12.2. Функция инициализации

```
function GrabNumber(){
// Создать функцию
    var urlXML='PhoneXML.php?q=' + document.Forml.user.value;
// О Сформировать URL XML
    var urlXSL=•Phone.xsl';
// © Сформировать URL XSL
    var newlmg=document.createElement('img');
// © Создать элемент изображения
    newlmg.setAttribute('src',   'images/loading.gif);
// О Установить источник
    document.getElementById("results").appendChild(newlmg);
// © Добавить изображение на страницу
    LoadXMLXSLTDoc{urlXML,urlXSL,"results");
// 0 Начать загрузку
```

Данная функция собирает информацию, требуемую для вызова сервера, устанавливает изображение, которое будет отображаться в процессе обра-

ботки, и вызывает сервер, который динамически создаст ответные данные, основываясь на отправленном значении строки запроса. Первым параметром функции LoaciXMLXSLTDoc () является URL страницы PHP, которая генерирует XML-документ, объединенный со строкой запроса, сформированной ссылкой на значений поля HTML-формы О. Второй параметр — это имя XSLT-файла ©, используемого в преобразовании XML-данных. Третий параметр, требуемый функцией LoadXMLXSLTDoc (), представляет собой идентификатор элемента div, в который следует помещать результаты поиска. Идентификатор — это строковое имя выходного элемента, а не ссылка на объект; в данном случае в качестве идентификатора используется строка "results".

На следующем этапе мы с помощью методов DOM добавляем на Web-страницу изображение-индикатор. Создается элемент изображения © и устанавливается атрибут источника изображения О. Этот созданный элемент добавляется к элементу div с результатами ©. Таким образом, когда наша функция вызывается из обработчика событий onsubmit формы, на страницу помещается файл изображения. Вообще, для пользователя важно создать визуальную обратную связь — сообщение или изображение, — указывающую, что обработка находится в процессе. Благодаря этому пользователь не будет повторно щелкать на кнопке отправки форы, думая, что ничего не происходит (помните, процесс Ajax — "незаметный").

Последний этап — это вызов функции LoadXMLXSLTDoc () ®, инициирующей процесс отправки информации на сервер. Функция LoadXMLXSLTDoc (), описанная в разделе 12.4, обрабатывает вызов объекта ContentLoader (), который запрашивает документы с сервера. Задавая в качестве параметра выходное положение (а не кодируя значение жестко в функции LoadXMLXSLT-Doc {)), мы можем многократно использовать данную функцию на одной странице, не требуя для разделения функциональных возможностей добавления множества процедур или операторов if. Следовательно, мы перенаправляем результаты различных поисковых запросов на различные части страницы. Однако, прежде чем мы все это сделаем, давайте посмотрим, как создать на сервере документы XML и XSLT.

12.3. Код серверной части приложения: PHP

В данном разделе мы создадим для проекта динамический XML-документ, используя PHP — популярный язык подготовки сценариев с открытым исходным кодом (как вы знаете, инфраструктура Ajax совместима с любым серверным языком или платформой). XML-документ генерируется динамически по набору результатов, полученному в ответ на запрос клиента к базе данных. Кроме того, мы покажем, как создать статический документ XSLT, который расположен на сервере и извлекается каждый раз, когда запрашивается динамический файл. Оба указанных документа возвращаются клиенту независимо, когда объект ContentLoader запрашивается в двух отдельных запросах (листинг 12.7). XSLT-код преобразовывает наш динамический XML-документ на стороне клиента и создает HTML-таблицу, которая отображается пользователю.

12.3.1. Создание XML-документа

Поскольку мы используем XSLT, нам нужен структурированный XML-документ, представляющий собой простую запись информации, чтобы XSL-файл мог выполнить стандартное преобразование. В данном проекте мы создадим динамический XML-файл, когда клиент затребует PHP-файл.

Разработка XML-структуры

Прежде чем мы начнем создание XML-файла, необходимо создать шаблон для него. Этот шаблон должен отражать структуру данных, возвращаемых при поиске. В выбранной формулировке задачи (телефонная книга) мы будем возвращать название компании, имя контактного лица, страну и телефонный номер. В листинге 12.3 показан базовый XML-шаблон, содержащий четыре поля.

Листинг 12.3. Базовый XML-файл

```
<?xml version="1.0" ?>
    <phonebook>
        <entry>
            <company>Company Name</company>
            <contact>Contact Name</contact>
            <country>Country Name</country>
            <phone>Phone Number</phone>
        </entry>
    </phonebook>
```

Первым элементом является phonebook. Следующий — элемент entry, содержащий подэлементы со всеми деталями, которые связаны со всеми контактными телефонами, найденными в запросе. Если у нас есть пять результатов, в XML-документе будет пять элементов entry. Имя компании отображается в элементе company. Кроме того, мы добавили имя контактного лица, название страны и номер телефона. Мы не ограничены только указанными полями; в зависимости от того, какую информацию необходимо отобразить, поля можно добавлять и удалять.

Если результаты не найдены, то вместо отображения предупреждающего сообщения можно создать элемент, отображающий эту информацию для пользователя. STO облегчит нам задачу возврата результата пользователю и не потребует дополнительного кода в клиентской части приложения. Код, приведенный в листинге 12.4, практически не отличается от кода из листинга 12.3, но на этот раз мы вводим текст в элементы XML, которые желаем показать пользователю, чтобы сообщить, что результаты не найдены.

Листинг 12.4. Файл XML без результатов

```
<?xml version="1.0" ?>
    <phonebook>
        <entry>
            <company>No Results</company>
// О Вместо имени компании отображается "No Results"
            <contact>N/A</contact>
```

```
// © Вместо оставшихся полей отображается "N/A"
            <country>N/A</country>
            <phone>N/A</phone>
        </entry>
    </phonebook>
```

С помощью данного кода мы отображаем пользователю единственную строку, сообщающую, что затребованная информация отсутствует. В дескрипторе company О отображается информация, сообщающая, что результатов нет. В других дескрипторах 0 пользователю сообщается, что информации нет. Если мы не желаем отображать текст "N/A" ("не доступно"), можно добавить вместо него неразрывный пробел, , что позволит показать ячейки таблицы. Если бы мы не добавили вообще никакой информации, ячейки в таблице не появились бы.

Как видите, формат XML имеет очень простую структуру. Если бы данный XML-файл был статическим, пользователю было бы относительно просто добавить в файл нового абонента. Поскольку же он создается динамически, нам потребуется цикл, создающий XML-документ по набору результатов.

Создание динамического XML-документа

Как всегда, мы создаем XML-документ на сервере. Придерживаясь политики использования в примерах различных серверных языков, серверный код дайной главы написан на PHP. Еще раз напомним, что инфраструктуру Ajax можно создать с помощью любого серверного языка, и мы будем описывать только сам принцип реализации серверного кода, не вдаваясь в детали. Итак, в листинге 12.5 показан код серверной части приложения. Код получает параметр строки запроса и генерирует множество результатов запроса базы данных. Затем мы проходим по множеству результатов, создавая в XML-файле согласно приведенному в листинге 12.4 шаблону элемент для каждого номера телефона, полученного в ответ на запрос.

Листинг 12.5. Сценарий phoneXML.php: генерация XML-документа на сервере

```
<?php
// О Объявить тип MIME
header("Content-type: text/xml");
echo("<?xml version='1.0' ?>\n");
// © Соединиться с базой данных
$db=mysql__connect("localhost","ajax","action");
rnysql_select_db("ajax",$db);
$result = mysql_query("SELECT *
    FROM Contacts WHERE ContactName like '%".
// © Заполнить запрос
$_GET['q']."%"',$db); ?>
<phonebook>
<?
// О Проверить результаты
if ($myrow = mysql_fetch_array($result)) {
    do {
// © Пройти по множеству результатов
```

```
?>
<entry  id='<?=$rayrow['id']?>001'>
    <company><?=$myrow['companyNarae']?></company>
    <contactx?=$myrow['contactName']?></contact>
    <countryX?=$myrow['country']?></country>
    <phone><?=$myrow[ 'phone' ] ?x/phone>
</entry>
<?
    }while ($myrow - mysql_fetch_array{$result));
}else{
?>
<entry  id='001'>
// @ Показать пустой набор данных
    <company>No  Results</company>
    <contact>N/A</contact>
    <country>N/A</country>
    <phone>N/A</phone>
    </entry>
<? } ?>
</phonebook>
```

Чтобы данная динамическая генерация XML-документа удалась, тип MIME документа необходимо установить равным text/xml О; если пропустить этот этап, XSLT-преобразование может не произойти (особенно в браузерах Mozilla и Firefox).

Требуемые нам данные хранятся в таблице базы данных, и нам необходимо просто выбрать нужные записи. В данном случае, для того чтобы максимально упростить работу, мы для непосредственного общения с базой данных используем встроенные функции MySQL языка PHP. Мы соединяемся с базой данных ajax, запущенной на локальном сервере базы данных, используя имя пользователя ajax и пароль action ©. После этого создается строка SQL-запроса; для заполнения оператора WHERE применяется параметр request, переданный в клиентском коде ©.

Для более надежной реализации серверной части приложения рекомендуется не связываться напрямую с базой данных, как в приведенном коде, а использовать структуру, подобную DB_DataObject (Pear) (см. главу 3). Впрочем, текущая реализация очень проста, и читатели, желающие самостоятельно протестировать рассматриваемый пример, могут ее легко настроить.

Получив множество результатов, мы проверяем наличие в нем данных О, а затем либо последовательно проходим по нему © для создания записей телефонной книги, либо выдаем сообщение "No Results" *0*.

12.3.2. Создание документа XSLT

Используя XSLT, наш XML-файл с помощью пары строк кода можно преобразовать в красивую таблицу HTML. XSLT-документ разрешает сопоставление с шаблоном, если оно необходимо для отображения данных в любом требуемом формате. При сопоставлении с шаблоном для отображения данных применяется структура-шаблон. При этом мы проходим по узлам дере-

ва источника, используя XSLT. XSLT-доку мент принимает структурированный XML-файл и преобразует его в формат, который удобен для просмотра, обновления и изменения. В нашем случае XSLT-доку мент определяется статически.

Структура XSLT

XSLT-преобразование содержит правила для перевода исходного дерева в конечное. Весь XSLT-процесс заключается в сопоставлении со структурой-шаблоном. Когда элементы исходного дерева соответствуют заданной структуре, согласно шаблону документа создается конечное дерево.

Структура конечного дерева не обязательно должна быть связана со структурой исходного. Следовательно, исходный XML-файл можно преобразовать в любой требуемый формат. Использовать только табличное представление набора данных не обязательно.

XSLT-преобразование называется таблицей стилей, поскольку оно определяет стилевое оформление конечного дерева. Таблица стилей содержит правила шаблона, состоящие из двух частей. Первая часть — это структура-шаблон, с которой сравниваются узлы исходного дерева. Обнаружив соответствие, XSLT-процессор задействует вторую часть — шаблон, содержащий дескрипторы для построения исходного дерева.

Создание XSLT-документа

Сформировать XSLT-преобразование для данного проекта сравнительно просто. Поскольку мы собираемся получить таблицу, никакое необычное сопоставление с шаблоном не потребуется; мы просто последовательно пройдем по всем узлам-элементам исходного дерева. Ниже мы разработаем шаблон, формирующий таблицу HTML с четырьмя столбцами. Соответствующий XSLT-файл для данного проекта показан в листинге 12.6.

i Листинг 12.6. XSLT-файл

```
<?xml version="1.0" encoding=>"ISO-8859-l"?>

//  О Установить версию XML и кодировку
<xsl:stylesheet version="l.O"
xmlns:xsl=
// © Задать пространство имен XSLT
    "http://www.w3.org/1999/XSL/Transform">
// © Установить правила шаблона
<xsl:template match="/">
// О Добавить элемент table
    <table id="tablel">
// © Создать строку заголовка
        <tr>
            <th align="left">Company</th>
            <th align="left">Contact</th>
            <th align="left">Country</th>
            <th align="left">Phone</th>
        </tr>
```

```
// 0 Последовательно пройти по элементам телефонной книги
        <xsl:for-each
            select="phonebook/entry">
// © Отформатировать выходные данные
        <tr>
            <tdxxsl:value-of  select="company"/></td>
            <td><xsl:value-of  select="contact"/></td>
            <td><xsl:value-of  select="country"/></td>
            <td><xsl:value-of  select="phone"/></td>
        </tr>
        </xsl:for-each>
    </table>
</xsl:template>
</xsl:stylesheet>
```

При создании XSLT-преобразования необходимо указать кодировку и версию XML O, а также задать пространство имен XSLT ©. Пространство имен определяет правила и спецификации, которым должен соответствовать документ. Элементы в пространстве имен XML распознаются в исходном документе, но не распознаются в документе результатов. Для определения всех наших элементов в пространстве имен XSLT применяется префикс xsl. Далее молено установить правило шаблона — искать структуру / ©, которая соответствует всему документу.

Теперь можно начинать создание шаблона таблицы, в которой отображаются наши результаты. Мы добавляем дескриптор table O, сопоставляющий с таблицей идентификатор. После этого вводится строка заголовка таблицы ©, вмещающая имена столбцов, указывающих пользователю, какая информация содержится в таблице.

Последовательно проходя по множеству узлов исходного дерева, мы получаем остальные строки таблицы. В данном случае используется цикл for-each ©, в процессе обработки записей выдающий узлы, расположенные в phonebook/entry.

Поскольку мы последовательно проходим по дереву документа, необходимо выбрать значения столбцов. Чтобы выбрать значения из узлов, используется оператор value-of в, извлекающий значение элемента XML и добавляющий его в выходной поток преобразования. Чтобы задать элемент XML, текст которого мы желаем извлечь, используем с именем элемента атрибут select. Сформировав XSLT-фаЙл и создав код для динамической генерации документа XML, можно завершить создание JavaScript-кода и посмотреть, как объединение XSLT-преобразования со структурированным XML-файлом позволяет получить удобную для просмотра таблицу.

На следующем этапе мы снова возвращаемся на сторону клиента, извлекающего файлы, созданные только что с помощью HTTP-отклика.

12.4. Объединение документов XSL и XML

Возвращаясь на сторону клиента, мы сталкиваемся с задачей объединения полученных с сервера документов XSL и XML. При использовании XSLT-преобразования следует помнить, что браузеры по-разному объединяют документы указанных типов. Следовательно, вначале необходимо проверить, какой метод поддерживает браузер, чтобы загрузить и объединить два документа.

Как и ранее, мы используем объект ContentLoader (см. главу 3). Данный объект содержится во внешнем JavaScript-файле net.js. Этот файл определяет, как отправлять информацию на сервер, скрывая все отличия работы браузеров за удобным интерфейсным объектом.

```
<script type="text/javascript" src="net.js"X/script>
```

Теперь можно начинать процесс получения файлов с сервера и объединения их на стороне клиента. В листинге 12.7 приведена функция LoadXMLXSLTDoc(), вызываемая из функции GrabNumber(), представленной в листинге 12.2. Функция GrabNumber() передает значения URL, генерирующего XML-данные, XSL-файл и идентификатор элемента, в который должны выводиться данные. Имея три указанных значения, мы можем загрузить два документа и объединить их после завершения загрузки. Как показано в листинге 12.7, для объединения файлов XML и XSL потребуется специальный код. предусматривающий возможность использования различных браузеров.

ЛИСТИНГ 12.7. ФУНКЦИЯ LoadXMLXSLTDoc . Я Н Н Н Н Н Ш Н В Н Н

```
// О Обьявить глобальные переменные

var xmlDoc;
var xslDoc;
var objOutput;
// © Обнулить переменные
function  LoadXMLXSLTDoc(urlXML,urlXSL,elementID){
    xmlDoc=null;
    xslDoc=null;
// & Определить выходной элемент
    objOutput = document.getElementById{
    elementld);
// О Загрузить файлы XML и XSL
    new net.ContentLoader(urlXML,onXMLLoad);
    new net.ContentLoader(urlXSL,onXSLLoad);
}
// 0 Обработать XML-докуыеит
function onXMLLoad(){
    xmlDoc=this.req.responseXML;
    doXSLT();
}
// 0 Обработать XSL-документ
function onXSLLoad(){
    xslDoc=this.req.responseXML;
    doXSLT();
}
// О Проверить, загрузились ли документы
```

```
function doXSLT{){
    if (xmlDoc==null || xslDoc==null){ return; }
// © Преобразовать XML-докуыент
    if (window.ActiveXObject){
        objOutput.innerHTML=xmlDoc.transformNode(xslDoc);

    }
    else{
        var xsltProcessor = new XSLTProcessor();
        xsltProcessor.importstylesheet(xslDoc);
        var fragment =xsltProcessor.transformToFragment(
                            xmlDoc,document);
        objOutput.innerHTML = "";
        objOutput.appendChild(fragment);
    }
```

Чтобы упростить клиентскую часть сценария, необходимо объявить три глобальные переменные О, которые будут вмещать три различных объекта. Первые две переменные, xmlDoc и xslDoc, предназначены для хранения файлов XML и XSL, возвращаемых с сервера. Третья переменная, objOutput, содержит объектную ссылку на элемент DOM, в который будут выводиться результаты. Определив указанные переменные, можно создать функцию LoadXMLXSLTDoc (), вызываемую из функции GrabNumber ().

Поскольку мы загружаем два документа, нужно определить момент, когда оба они будут доступны. Для этого мы проверим, получили ли уже переменные xmlDoc и xslDoc соотнесенные с ними документы. Прежде чем начать, мы должны присвоить обеим этим переменным значение null @. Это гарантирует, что переменные не содержат никаких данных, даже если функция запускается на странице несколько раз. Чтобы задать объект для вывода результата объединения, из вызова функции берется идентификатор переданного элемента ©. Далее мы дважды вызываем функцию Content-Loader — один раз для документа XML и один раз для документа XSL О. При каждом вызове функция ContentLoader получает URL, а затем вызывает другую функцию для загрузки документов. Функция onXMLLoadO © загружает возвращаемый XML-документ в глобальную переменную xmlDoc, а затем вызывает функцию doXSLT () для последующей обработки. Функция onXSLLoadO © загружает XSL-документ в глобальную переменную xslDoc и также вызывает функцию doXSLT ().

Обработка не может продолжаться, пока не будут загружены оба документа, а мы никак не можем определить, который из них будет загружен первым, поэтому функция doXSLT () вначале проверяет, загружены ли оба документа. Она вызывается дважды — после загрузки документа XML и документа XSL. При первом вызове этой функции одна из глобальных переменных все еще имеет значение null, поэтому мы завершаем выполнение функции 0. При следующем вызове выполнение функции не завершается, поскольку ни одна переменная не имеет значения null. Теперь загружены оба документа, и можно выполнять XSLT-преобразование ©.

После загрузки обоих документов необходимо преобразовать XML-документ с применением XSLT. Изучая код, приведенный в листинге 12.7. видим, что требуемого можно достичь двумя различными способами в зависимости от браузера. Internet Explorer использует функцию transf ormNode {), тогда как Mozilla — объект XSLTProcessor. В связи с этим ниже мы подробно рассмотрим две различные реализации преобразования.

12.4.1. Совместимость с браузером Microsoft Internet Explorer

Браузер Internet Explorer позволяет преобразовать XML-документ согласно XSLT-доку менту с помощью всего пары строк кода. В нашем проекте используется метод transf ormNode (), принимающий документы XML и XSLT и объединяющий их.

```
if (window.ActiveXObject){
    objOutput.innerHTML=xmlDoc.transformNode(xslDoc); }
```

Вначале мы определяем, поддерживает ли браузер метод transf ormKode (). Для этого необходимо проверить, поддерживает ли браузер объект ActiveX. Если объект поддерживается, к глобальной переменной, содержащей XML-данные, применяется метод transformNode(), передавая этой переменной глобальную переменную, содержащую XSLT-данные. Результат данного преобразования добавляется к свойству innerHTML выходного элемента, который теперь будет содержать отформатированные результаты поиска.

Итак, мы разобрались, как форматировать результаты для Internet Explorer. Теперь сделаем то же для браузеров, совместимых с Mozilla.

12.4.2. Совместимость с браузерами Mozilla

Для браузеров Mozilla необходимо использовать объект XSLTProcessor, позволяющий объединять документы XML и XSL. Обратите внимание на то, что хотя браузеры Opera и Safari поддерживают объект XMLHttpRequest, они не поддерживают объект XSLTProcessor, и в них невозможен запуск рассматриваемого проекта без поддержки с сервера (подробнее данный вопрос рассмотрен в разделе 12.5.2). В листинге 12.8 реализовано преобразование XML-документа с использованием XSLT и отображение отформатированного множества результатов.

Листинг 12.8. Вызов XSLT для браузеров Mozilla

```
else{
    var xsltProcessor = new XSLTProcessor();
    xsltProcessor.importstylesheet(xslDoc);
    var fragment^xsltProcessor.transformToFragment(xmlDoc,document)I
    objOutput.innerHTML = "";
    objOutput.appendChild(fragment);
```

_ >

рис. 12.4. Отображение результатов "живого" поиска Ajax

Прежде всего необходимо инициализировать объект XSLTProcessor, позволяющий объединять файлы XML и XSL. С помощью метода importStylesheet объекта XSLTProcessor можно импортировать файл XSL, что позволит нам в дальнейшем присоединить его к файлу XML. Загрузив в процессор файл XSL, остается преобразовать XML-документ. Для этого снова используется объект XSLTProcessor, на этот раз с новым методом transformToFragmentO. Метод transformToFragment () принимает файл XML и объединяет его с XSL, возвращая отформатированное конечное дерево.

Для замещения содержимого элемента result свойству innerHTML присваивается значение, равное пустой строке. Таким образом, со страницы удаляется анимация ожидания. Наконец, берется результат, полученный от метода transformToFragment (), и добавляется к элементу result. Теперь пользователь видит отформатированные результаты поиска.

В приведенном коде было введено несколько новых концепций, в том числе объект XSLTProcessor, позволяющий сочетать произвольные файлы XML и XSL. Для объединения подобных документов в браузерах Mozilla и Firefox требуется использовать больше методов DOM. В Internet Explorer для преобразования документа XML нужна единственная строка кода. Окончательные итоги обоих преобразований идентичны: отображаемые результаты поиска отформатированы согласно данным XSL-файла.

Завершив разработку клиентской части сценария, можно сохранить документы и протестировать полученное приложение "живого" поиска. Введите в текстовое окно какой-либо текст и щелкните на кнопке Search ("поиск"). Результаты должны появиться в табличной форме, подобной показанной на рис. 12.4.

Как видите, мы создали XSL-документ и можем успешно производить поиск. Таблица, показанная на рис. 12.4, выглядит довольно скучно, поскольку не содержит форматирования. Это означает, что нам осталось разработать стилевое оформление таблицы результатов, чтобы сделать ее более наглядной. Для этого нам потребуются каскадные таблицы стилей (Cascading Style Sheets-CSS).

12.5. Последние штрихи

Объединив документы XML и XSL для получения результатов, мы должны улучшить стиль отображения результатов поиска, применив к соответствующим элементам правила CSS. Стилевое оформление элементов облегчает восприятие результатов пользователями. Первое, что нужно сделать для улучшения удобства пользователя, — это применить правила CSS к элементам HTML.

12.5.1. Применение каскадных таблиц стилей

О каскадных таблицах стилей (Cascading Style Sheets — CSS) речь шла в главе 2. Благодаря им результаты будут выглядеть профессионально при минимальных усилиях с нашей стороны, при этом представление результатов будет отделено от структуры документа и логики преобразования. Если вдруг когда-нибудь менеджера или клиента начнут раздражать выбранные цвета, соответствующие изменения можно будет выполнить легко и быстро. Если мы участвуем в большом проекте с отдельными командами дизайна и кодирования, CSS поможет нам не мешать друг другу. Таблицу стилей можно привязать к поисковой странице как внешний файл, также ее код можно внедрить непосредственно на страницу. Использование внешнего файла CSS предпочтительнее, поскольку он кэшируется браузером и уменьшает время загрузки страницы в будущем. Используемые нами правила таблицы стилей показаны в листинге 12.9.

Листинг 12.9. Каскадная таблица стилей

```
//O Стиль таблицы
table{
        border: lpx solid black;
        border-collapse: collapse;
        width: 50%;
}
// © Стиль ячеек таблицы
th, td{
    border: lpx solid black;
    padding: 3px;
    width: 25%;
}
// @ Стиль ячеек заголовка
th{
        background-color: #A0A0A0;
}
```

Первое правило CSS применяется к дескриптору table O. В данном случае мы хотим, чтобы рамка вокруг таблицы формировалась сплошной линией толщиной один пиксель. Свойству border-collapse таблицы присваивается значение collapse. Такая модель CSS, по сути, позволяет унифицировать свойства таблицы. Границы имеют равную толщину — границы соседних ячеек считаются общими, так что между ячейками не возникает границ двойной или тройной толщины. Последним изменением дескриптора table является присвоение значения свойству width. Поскольку столбцов не много, ширина таблицы устанавливается равной 50% ширины содержащего ее элемента div. Все столбцы таблицы будут содержать небольшой объем информации, поэтому большая разрядка в данном случае не нужна.

Определив стиль элемента table, необходимо отформатировать тело и заголовок таблицы ©. Точно так же, как и для всей таблицы, рамку для тела и заголовка мы задаем в виде сплошной линии толщиной один пиксель. Кроме того, добавим небольшое заполнение, чтобы текст не "слипался" с граница-

Рис. 12.5. Результаты "живого" поиска Ajax с примененными к элементам стилями CSS

ми ячейки. Значение свойства width ячеек мы задаем равным 25% ширины таблицы, чтобы все четыре столбца имели одинаковый размер

Последний этап стилевого оформления таблицы с помощью правил CSS — изменение свойства ячеек заголовка, чтобы визуально отделить их от ячеек тела таблицы. Мы обращаемся к ячейке заголовка © и изменяем цвет ее фона (background-color) на оттенок серого. Здесь же можно изменить и другие свойства — font-weight, color и т.д. Завершив разработку свойств таблицы стилей, мы записываем документ и снова запускаем поиск. Теперь отформатированная таблица выглядит так, как показано на рис. 12.5.

Видно, что таблица имеет структуру, полученную путем применения свойств CSS к элементам таблицы. Если в таблице стилей требуется внедрить больше функциональных возможностей, можно добавить ссылки на файл XSLT и получить еще более гибкую структуру. CSS позволяет настраивать таблицу любым удобным способом, однако поиск можно улучшить и по другим направлениям.

12.5.2. Улучшение поиска

Одним из достоинств Ajax является легкость передачи информации на сервер. Данный проект — это всего лишь упражнение по реализации поиска с использованием Ajax и применением XSLT для отображения таблицы результатов при минимальных усилиях. "Живой" поиск можно совершенствовать до бесконечности. Рассмотрим, например, несколько путей его улучшения.

Включение новых возможностей

В созданной нами форме для выполнения поиска применяется единственное текстовое окно и одна кнопка отправки. Однако мы вполне можем задействовать в поиске множество параметров, например, дополнительно учитывать при поиске имя контактного лица или страны. Все, что для этого требуется, — послать на сервер чуть больше параметров и приказать серверному коду обработать их. Сказанное означает, что пользователи могут искать информацию и с помощью альтернативных вариантов, что сделает форму полезнее.

В данный сценарий можно добавить и другие возможности Ajax, например, реализовав двойные связанные комбинации (см. главу 9), что позволит отсеять результаты, отображаемые для пользователя. Кроме того, можно реализовать технологии, описанные в главе 10, введя в текущий сценарий возможности опережающего ввода.

Поддержка браузерами Opera и Safari

Напомним, что браузеры Opera и Safari не поддерживают ни метод transformNodeO, ни объект XSLTProcessor. Обеспечить требуемую поддержку можно двумя способами. Первый вариант — использовать Ajax для отправки файлов на сервер, где код серверной части приложения объединит документы XML и XSL. Затем результат преобразования можно извлечь с помощью одного объекта Content Loader, а не двух независимых элементов — данных XML и таблицы стилей XSLT. Такое решение будет не самым лучшим, поскольку для выполнения преобразования клиенту дважды придется обращаться к серверу.

Второй вариант — с помощью Ajax отправить на сервер всю страницу. В таком случае сервер обработает полученный материал и объединит документы XML и XSL. Такой подход лучше, поскольку он позволяет использовать поиск всем пользователям. Если кто-то работает со старой версией браузера, не поддерживающей объект XMLHttpRequest, это не будет препятствовать использованию формы. Если же мы будем полагаться только на технологию Ajax, пользователи, которым она не доступна, не смогут извлечь два требуемых файла для обработки. Поэтому мы примем второй подход, в котором поддержка Ajax не является необходимым условием. Чтобы добавить требуемые функциональные возможности, понадобится в двух местах изменить функцию LoadXMLXSLTDoc {), как показано в листинге 12.10. Во-первых, нужно переписать первый оператор if, добавив проверку наличия процессора XSLT. Во-вторых, следует добавить вариант else, инициирующий отправку формы на сервер.

Листинг 12.10. Функция LoadXMLXSLTDoc G **поддержкой Opera** и Safari

```
function LoadXMLXSLTDoc(urlXML,urlXSL,elementID){
    var reqXML; var reqXSL;
if {window.XMLHttpRequest && window.XSLTProcessor){
    //... do Mozilla client XSLT
} else if (window.ActiveXObject){
    //... do Internet Explorer client XSLT
} else{ document.Forml.onsubmit = £unction(){return true;(
       document.Forml.submit(); }
```

_ >

В листинге 12.10 мы переместили обработчик событий onsubmit внутрь ветки else условной проверки, чтобы можно было отправить форму на сервер. Если бы мы не убрали обработчик событий onsubmit, форму невозможно было бы вернуть на сервер.

Далее серверная часть приложения должна выполнить всю необходимую обработку и поместить элемент в форму. В результате мы получим возможность быстрого отклика для тех пользователей, которые могут самостоятельно объединить XSLT с JavaScript, при этом мы не отказываемся от тех пользователей, браузеры которых не поддерживают Ajax или XSLTProcessor. Помните, что Ajax позволяет не визуализировать повторно всю страницу, теряя при этом ее текущее состояние. Благодаря Ajax мы можем не думать

о прокрутке до нужного места, заполнении полей формы и т.д. Поскольку мы к тому же сумели решить проблему поддержки Opera и Safari, в вопросе целесообразности использования XSLT-преобр азов алия у нас стало меньше еще одним аргументом против.

12.5.3. Использовать ли XSLT

В процессе работы над проектом вполне может возникнуть ситуация, когда вам придется отстаивать перед руководством или членами команды желание использовать XSLT. Возможно, вам зададут такой вопрос: "Вы динамически генерируете на сервере файл XML, так почему же нельзя точно так же сгенерировать таблицу результатов?"

По сути, мнение наших коллег-разработчиков заключается в том, что для отображения результатов пользователю мы делаем,больше работы, чем требуется. Действительно, когда браузер визуализирует таблицу результатов, на Web-странице должна быть выполнена дополнительная обработка. Вместо одного файла, код Ajax загружает два, которые впоследствии необходимо объединить.

Мы без проблем могли бы сгенерировать таблицу на сервере. Таблица результатов отображалась бы с использованием свойства responseText объекта XMLHttpRequest и возвращенного значения к HTML-элементу. Данный метод нормальный и жизнеспособный, причем содержит на один этап меньше, чем предложенный выше.

Однако при построении HTML-таблицы на стороне сервера следует подумать еще и о тех усилиях, которые придется приложить, если потребуется изменить что-то. Как обсуждалось ранее в данной главе, при построении таблицы разработчики сталкиваются с множеством проблем. Нужно подумать о кавычках, синтаксисе дескрипторов, атрибутах, обработчиках событий и о многом другом. Если пользователю потребуется изменить порядок столбцов в таблице результатов, нам придется перекодировать очень много фрагментов приложения.

Используя XSLT, мы выносим построение таблицы из кода серверной части приложения. Сервер может построить упрощенную версию таблицы результатов в формате XML. Этот формат достаточно понятен и позволяет легко выявлять ошибки. Кроме того, XSLT-файл выглядит подобно HTML-странице. Нам не придется подсчитывать кавычки или выполнять поиск в строке, чтобы определить, имеется ли в ней дескриптор. Используя XSLT, мы можем просто посмотреть и сказать, что все правильно.

Кроме того, мы можем воспользоваться созданным Web-дизайнером шаблоном таблицы и поместить его в файл XSLT. Если нам когда-либо придется что-то изменить (например, переставить столбцы), все наши действия сведутся к операциям "вырезать-вставить". Нам не придется морочить себе голову и проверять, нормально ли расставлены дескрипторы после модификации. Используя XSLT, мы убираем из динамического кода обработку HTML-страницы. Благодаря этому внешний вид таблицы результатов можно будет изменить без особых проблем.

Наконец, использование JavaScript позволяет очень легко сделать то, что при выполнении преобразования на сервере было бы просто невозможным.

- Извлекать различные документы XSL, основываясь на теме, размерах экрана, языке и т.д.
- Извлекать документы XML и XSL без помощи сервера.
- Изучать журнал (log file) XML на локальном компьютере, не имея контроля над структурой документа XML.

Наверное, вы уже поняли, что Ajax может существенно облегчить выполнение ваших повседневных задач.

Тем не менее мы не рассмотрели еще один вопрос, связанный с "живым" поиском Ajax: как позволить пользователю сформировать закладку на страницу результатов.

12.5.4. Решение проблемы закладок

При поиске с использованием Ajax мы сталкиваемся с проблемой: создать закладку на страницу традиционным способом не получается. (Та же проблема возникает при использовании фреймов и всплывающих окон.) Закладки позволяют возвращаться в дальнейшем к результатам поиска без ввода запроса; кроме того, закладки удобно отправлять друзьям и коллегам с помощью электронной почты или приложений обмена мгновенными сообщениями. Поскольку поиск Ajax не требует дообработки на сервере, в процессе поиска мы не изменяем URL страницы, следовательно, сделав закладку на этот URL, мы просто отметим отправной пункт приложения, а не результаты, которые предполагалось сохранить.

Легким решением данной проблемы является добавление линии поведения, позволяющей запоминать результаты поиска. Мы можем создать динамическую строку, которая будет использоваться для создания динамической закладки. Эта динамическая строка будет содержать прямую ссылку на ту страницу, на которой она находится , и включать параметр строки запроса с искомым значением. Таким образом, записывая данную строку на страницу для формирования ссылки, пользователь может либо создать на нее закладку (щелкнув правой кнопкой мыши на ссылке), либо скопировать ссылку (при этом будут автоматически записаны условия поиска). Возможность считывания значения строки запроса мы опишем после того, как разберемся с созданием ссылки.

Ссылку можно создать при выполнении функции GrabNumber(). В наш документ добавляется еще один элемент span, в который мы поместим ссылку. В данном случае элемент span имеет идентификатор spanSave, как показано в листинге 12.11 (вызов метода getElementByld). Отметим, что элемент span можно разместить в любом месте страницы (что достаточно удобно Для пользователя).

Листинг 12.11. Функция GrabNumber, содержащая ссылку-закладку

```
function GrabNumber(){
    var strLink = "<a href=" +
        location.href.split("?")[0] + "?q=" +
        document-Forml.user.value + '">Save Search</a>";
    document.getElementById("spanSave").innerHTML = strLink;
```

Код, приведенный в листинге **12.11,** генерирует динамическую ссылку на текущую страницу поиска и добавляет параметр строки запроса q со значением, указанным в текстовом окне. Именно параметр строки запроса позволит нам в будущем "вспомнить" произведенный поиск. Далее эта новая ссылка добавляется в элемент span на странице, чтобы пользователь мог выбрать ссылку и отправить ее другим пользователям или создать закладку на нее, щелкнув правой кнопкой мыши на ссылке и добавив ее в папку Избранное. В листинге 12.12 мы получаем значение строки запроса из URL во время загрузки страницы, а затем выполняем поиск автоматически, чтобы отобразить результаты.

Листинг 12.12. Получение значения строки запроса и выполнение поиска

```
window.onload = function(){
    var strQS = window.location.search;
    var intQS = strQS.indexOf("q=");
    if(intQS != -1){
        document.Forml.user.value = strQS.substring(intQS+2);
        GrabNumber();
    }
}
```

К объекту окна мы добавили обработчик события onload, который выполнит функцию при загрузке страницы. Проверим, содержится ли значение строки запроса в URL; если да, то мы получаем значение. Значение строки запроса помещается в текстовом окне, а затем автоматически выполняется функция GrabNumber(), формирующая таблицу результатов. Добавление указанного кода позволяет создавать закладки на страницы поиска и получать результаты поиска на странице без ввода условий поиска вручную. Благодаря этому наш проект Ajax становится еще более дружественным к пользователю.

12.6. Реструктуризация

Пришло время вывести **"живой"** поиск XSLT на следующий уровень, используя (вы догадались!) компонентное представление. Мы должны взять полученный стильный сценарий и реструктуризировать в объектно-ориентированный повторно используемый компонент. Начнем с XSLT-обработки на стороне клиента. Отметим, что данный пример отличается от

всех остальных Б ТОМ смысле, что все вопросы, связанные с обработкой ответа методами DOM. он решает с помощью XSLT. Ну что ж, начнем! Мы должны так изменить XSLT-обработку, чтобы ее можно было использовать и с другими компонентами (а не только с "живым" поиском). Сделав это, мы сосредоточимся на такой реструктуризации "живого" поиска, чтобы его можно было быстро добавить на любую страницу как удобный настраиваемый компонент.

12.6.1. Объект XSLTHelper

Нам пришлось много потрудиться, чтобы выяснить все подробности XSLT-обработки на стороне клиента. Например, мы узнали, что для выполнения указанного преобразования можно задействовать абсолютно разные API, в зависимости от того, на какие браузеры мы рассчитываем — Internet Explorer или Mozilla. Кроме того, каждый API имеет собственные индивидуальные особенности. Поэтому было бы стыдно не инкапсулировать эти тяжким трудом полученные знания, чтобы коллегам, которые пойдут вслед за нами, не пришлось с таким же трудом разбираться с кажущимися простыми XSLT-преобразованиями. Следовательно, создадим объект XSLTHelper, инкапсулирующий все вопросы, касающиеся XSLT.

Вся XSLT-обработка обычно требует два источника информации: преобразовываемого XML-документа и XSL-документа, содержащего правила преобразования. Учитывая это, можем написать конструктор вспомогательного объекта, с помощью которого сохраним это состояние.

```
function XSLTHelper( xmlURL, xslURL ) {

    this.xmlURL = xmlURL;
    this.xslURL = xslURL;
}
```

Пожалуй, этот конструктор является самым простым из всех, использованных в данной книге. Он просто записывает URL упомянутых выше документов: документа с XML-данными и документа с XSLT-преобразованием. Однако, прежде чем мы уверимся в совершенной простоте всей предстоящей работы, нужно придумать API, поддерживающий постепенное ухудшение условий. Нам нужно, чтобы сценарий выполнял XSLT-обработку, только если браузер ее поддерживает. Таким образом, если мы собираемся написать объект-"помощник", было бы неплохо, чтобы его API сообщал клиенту, может ли он выполнять XSLT-преобразования. Однако создание дополнительного объекта для выяснения, поддерживает ли браузер XSLT, — не очень хорошая идея. Решением такой проблемы является использование функции API, область действия которой распространяется не на объект-прототип, а на саму Функцию конструктора. Данную функцию можно рассматривать как статический метод в мире Java. В конечном итоге мы хотим, чтобы клиент мог написать код, который выглядит примерно следующим образом:

```
XSLTHelper.isXSLTSupported();
```

вместо того, чтобы создавать объект так, как показано ниже.

```
var helper = new XSLTHelper( 'phoneBook.xml',
                'transformation.xsl' ) ;
var canDoThis = helper.isXSLTSupported();
```

Для этого предоставим нашим пользователям статический метод, который выражается таким образом:

```
XSLTHelper.isXSLTSupported = function() {
    return (window.XMLHttpRequest && window.XSLTProcessor) ||
                    XSLTHelper.isIEXmlSupported();
}
XSLTHelper.isIEXmlSupported = function() {
    if ( ! window.ActiveXObject )
        return false;
    try { new ActiveXObject("Microsoft.XMLDOM");
            return true; }
    catch(err) { return false; }
>
```

Здесь мы не использовали ничего нового. Логика идентична определенной ранее; мы просто инкапсулировали информацию о наличии поддержки XSLT. Наверняка кто-то нас за это поблагодарит. Теперь же мы можем конкретизировать оставшуюся часть API XSLTHelper.

Не будем все усложнять. Как вы относитесь к тому, чтобы выполнение XSLT-обработки требовало от клиентов нашего класса вызова единственного метода? Для этого мы включим во вспомогательный объект дополнительные методы, разделяющие ответственности всей внутренней логики, но для клиентов мы реализуем один общий API. Семантика данного объекта будет следующей:

```
var helper = new XSLTHelper ( 'phoneBook.xml',
        'transformation.xsl' ) ;
helper.loadView{ 'someContainerId' );
```

В данном примере документ phoneBook. xml нужно преобразовать в HTML с помощью документа transformation.xsl, а полученный в результате HTML-код следует поместить в элемент с идентификатором someContainerId. Кроме того, зададим, что функция loadview() может принимать в качестве параметров либо строку, представляющую идентификатор элемента, либо сам элемент. Внутренний код функции сам определит, с чем он имеет дело, и отреагирует соответствующим образом. Да, кстати, если клиент не собирается повторно использовать экземпляр нашего вспомогательного объекта, все сказанное можно выразить в одной строке кода:

```
new XSLTHelper('phoneBook.xml',
                'transformation.xsl').loadView('someContainerId');
```

Определив API и его семантику, мы можем реализовать его, как показано в листинге 12.13.

Листинг 12.13. Метод loadview

```
loadView: function ( container ) {
// О Проверить поддержку XSLT
    if ( ! XSLTHelper.isXSLTSupportedO )
        return;
// © Заново инициализировать состояние вспомогательного объекта
    this.xmlDocument = null;
    this.xslStyleSheet = null;
    this.container = $(container);
// © Запросить документы
    new Ajax.Requestt this.xmlURL,
        {onComplete: this.setXMLDocument.bind(this)J );
    new Ajax.Request( this.xslURL,
        {method:"GET", onComplete:
    this.setXSLDocument.bind(this)} ) ;
Ь
```

Первое, что делает метод loadView(), — это убеждается в том, что используемый браузер поддерживает XSLT О. Вообще-то, клиент уже должен был это сделать (как в нашем предыдущем примере), но если нам попался безответственный пользователь, лучше лишний раз все перепроверить. Затем метод присваивает значение null переменным состояния, которые будут хранить документы XML и XSL, и устанавливает ссылку на обновляемый контейнер ©. Наконец, метод отправляет запросы Ajax для извлечения документов XML и XSL ©, Когда сервер пришлет затребованный документ XML, будет вызван метод setXMLDocument (). Аналогично, когда сервер вернет документ XSL, вызывается метод setXSLDocument (). Реализация данных функций показана в листинге 12.14.

Листинг 12.14. Настройка документов XML и XSL

```
setXMLDocument: function(request) {
    this.xmlDocument = request.responseXML;
    this.updateViewIfDocumentsLoaded();
},
setXSLDocument: function(request) {
    this.xslStyleSheet = request.responseXML;
    this.updateViewIfDocumentsLoaded();
```

Указанные методы сопоставляют с переменными состояния объекта XSLTHelper документы XML и XSL. Затем они вызывают метод updateViewifDocumentsLoaded(), который проверяет, инициализированы ли оба документа, и в случае положительного ответа обновляет представление. Метод updateViewIfDocumentsLoaded() реализован следующим образом:

```
updateViewIfDocumentsLoaded: function() {
    if ( this.xmlDocument == null |I this.xslStyleSheet == null )
        return;
    this.updateView();
Ь
```

Как только с сервера поступят оба документа, мы сможем обновить пользовательский интерфейс. Мы узнаем, что оба отклика поступили, когда обе переменные состояния, this .xmlDocument и this .xslStyleSheet, будут иметь значение, не равное null. Соответствующий метод обновления updateView() показан в листинге 12.15.

```
updateView: function () {
    if { ! XSLTHelper.isXSLTSupportedO )
        return;
    if { window.XMLHttpRequest && window.XSLTProcessor )
        this.updateViewMozilla();
    else if ( window.ActiveXObject )
        this.updateViewIE();
```

ь

Как уже отмечалось, для каждого поддерживаемого типа браузеров требуется отдельная реализация. Детали соответствующих процессов мы рассмотрим по отдельности, начав с реализации для браузера Mozilla, показанной в листинге 12.16.

Листинг 12.16. Обновление представления в браузере Mozilla

```
updateViewMozilla: function!) {
// Инициализировать трансформер
    var xsltProcessor =* new XSLTProcessor();
    xsltProcessor.importStylesheet(this.xslStyleSheet);
    var fragment - xsltProcessor.
// О Выполнить XSLT-преобразование
        Trans formToFragment(
            this.xmlDocument, document);
// @ Обновить пользовательский интерфейс
    this.container.innerHTML = "";
    this.container.appendChild(fragment);
```

L

Обновление визуального представления в поддерживаемых браузерах (Internet Explorer и Mozilla) включает два основных этапа: О выполнение XSLT-преобразования и © обновление пользовательского интерфейса согласно полученным результатам. Напомним, что результат процесса преобразования в браузере Mozilla — это фрагмент документа, добавляемый к элементу с помощью appendChild(), тогда как преобразование в Internet Explorer дает строку, добавляемую посредством свойства innerHTML. Таким образом, реализация функции updateViewlE () должна выглядеть следующим образом:

```
updateViewIE: function() {
    this.container.innerHTML =
        this.xmlDocument.transformNode{this.xslStyleSheet);
```

ь

В реализации для браузера Internet Explorer задействованы те же два этапа, что и ранее, причем на этот раз они записаны гораздо компактнее, по

скольку и применение преобразования, и обновление пользовательского интерфейса представляется одной строкой кода. Относительно того, какая из реализаций эффективнее — решать вам.

Итак, объект xSLTHelper готов, и у нас есть понятный, простой метод API, в который включены все действия, связанные с XSLT-преобразованиями. Данный вспомогательный объект наверняка будет'очень полезным и точно будет стоить тех усилий, которые мы в него вложили.

А теперь вернемся к "живому" поиску и попытаемся получить компонент с простой структурой.

12.6.2. Компонент "живого" поиска

Ну вот, в заднем кармане у нас припрятана удобная поддержка XSLT, и теперь мы реализуем сценарий "живого" поиска в виде компонента, который должен удовлетворять требованиям, обсуждавшимся в разделах о реструктуризации других глав. Нам требуется хороший API, возможность настройки при наличии необходимого числа значений по умолчанию, малозаметность для HTML-страницы, содержащей компонент, а также возможность использования на одной странице нескольких экземпляров компонента. Итак, разработаем понятное объектно-ориентированное решение, основным принципом которого будет инкапсуляция каждой ответственности в отдельный метод. Одна ответственность — один метод. Запомним этот принцип и начнем, как обычно, — с построения.

С точки зрения состояния структура "живого" поиска должна отслеживать больше элементов, чем любой другой написанный нами компонент. Она должна знать, где взять документы XML и XSL, какое поле инициирует поиск, какой URL страницы нужно использовать для поддержки закладок. В общем, соответствующий конструктор будет более громоздким, чем конструкторы, описанные в предыдущих главах. Тем не менее мы все равно сможем им управлять. Итак, посмотрим, как выглядит конструктор "живого" поиска.

```
function LiveSearch( pageURL, loohupField, xmlURL,
xsltURL, options ) {
this.pageURL = pageURL;
this.lookupField = lookupField;
this.xmlURL = xmlURL;
this.xsltURL = xsltURL;
this.setOptions{options);
// О Настроить компонент
var oThis - this;
lookupField.form.onsubmib • function(){
    oThis.doSearch(); return false; };
// Перейти к предыдущему поиску
this.initialize();
}
```

Первые четыре аргумента конструктора были названы выше: URL страницы, поле поиска и URL двух документов. Последним параметром является уже привычный параметр options, используемый для настройки конфигура-

ции компонента. Аргумент options передается методу setQptions(), который обеспечивает значения по умолчанию для всех настраиваемых данных О. Рассмотрим кратко этот метод.

```
setOptions: function(options) {
    this.options — options;
    if ( !this.options.loadinglmage )
        this.options.loadinglmage = 'images/loading.gif';
    if ( !this.options.bookmarkContainerId )
        this.options.bookmarkContainerId = 'bookmark';
    if { !this.options.resultsContainerId )
        this.options.resultsContainerId - 'results';
    if { !this.options.bookmarkText )
        this.options-bookmarkText = 'Bookmark Search';
>'
```

В данном примере метод setOptions{) не настолько лаконичен, как в компоненте TextSuggest (см. главу 10), когда с помощью метода extend() библиотеки Prototype мы получили аккуратное и компактное выражение. Тем не менее сейчас метод выполняет те же действия и предоставляет значения по умолчанию для изображения, отображаемого в процессе загрузки, идентификатора элемента, содержащего закладку, идентификатора элемента, содержащего результаты, и, наконец, текста сгенерированной закладки. Как и в прошлый раз, если в объекте options были переданы значения этих свойств,* они заменяют указанные значения по умолчанию. Полученный в результате объект options представляет собой смешанный набор значений по умолчанию и значений, заданных пользователем. Затем данные опции используются в соответствующих местах сценария для настройки конфигурации компонента.

Разобравшись с настройкой конфигурации и значениями по умолчанию, вернемся к конструктору и напомним две ненавязчивые строки кода:

```
var oThis = this;
lookupField.form.onsubmit = function(){
    oThis.doSearch{); return false; };
```

Напомним, что в первоначальном варианте сценарий модифицировал HTML-страницу, помещая в форму поиска обработчик событий onsubmit.

```
<form name="Forml" onsubmit="GrabNumber();return false;">
```

Поскольку мы пытаемся сделать компоненты максимально ненавязчивыми (по крайней мере, с точки зрения того, сколько HTML-кода нужно изменить для использования компонента), две приведенные выше строки кода конструктора предлагают те же функциональные возможности, что и упомянутая модификация HTML-страницы. С точки зрения именования отличие заключается в переименовании GrabNumberO в более универсальный вариант doSearch(), причем doSearch() — это метод нашего компонента, а не глобальная функция. Кстати, посмотрим, как реализован метод doSearcht):

```
doSearch: function{) {
    if ( XSLTHelper.isXSLTSupported() )
        this.doAjaxSearch();
    else this.submitTheFormt);
},
```

Наш "разумный" компонент знает, что он должен убедиться в поддержке XSLT, а лишь затем пытаться выполнить XSLT-обработку, поэтому метод поиска использует написанный ранее API xsLTHelper и определяет, использовать ли XSLT-обработку или активизировать стандартную отправку формы. Действительно, довольно разумно. Клиент может просто вызывать метод doSearch(), не задумываясь о тонкостях использования XSLT. Обо всем позаботимся мы. Поэтому сейчас подробно рассмотрим два варианта поиска. Поскольку отправка формы проще, начнем именно с нее.

```
submitTheForm: function() {
    var sesrchForm = this.lookupField.form;
    searchForm.onsubrait « function() { return true; };
    searchForm.submit();          t
Ь
```

Указанный метод находит с помощью поля поиска ссылку на соответствующую форму и изменяет ее обработчик событий onsubmit на функцию, возвращающую значения true. Это позволяет явно возвращать запрос поиска на сервер. Затем метод вызывает метод submit () "родной" формы HTML, который инициирует традиционную отправку формы. В данном сценарии компонент предполагает наличие в форме соответствующего атрибута действия; также считается, что результат действия возвращает соответствующую страницу с результатами поиска.

Далее рассмотрим реализацию поиска средствами Ajax.

```
doAj axSearch: function() {
// О Показать изображение в процессе загрузки
       this.showLoadinglmage();
       var searchUrl = this.appendToUrl( this.xmlURL, 'q',

// © Сформировать URL операции поиска
                        this.lookupField.value);

// © Выполнить XSLT-обработку
    new XSLTHelpertsearchUrl, this.xsltURL).loadView(
              this.options.resultsContainerld);

// О Обновить закладку
       this.updateBookmark ();
Ь
```

Метод doAj axSearch () выполняет те же действия, что и первоначальный вариант нашего сценария, однако теперь каждый этап обработки помещен в отдельный метод, отвечающий за свою часть работы. Вы можете возразить и сказать, что данный метод имеет четыре ответственности. Но на самом деле ответственность одна: поиск. Однако она состоит из четырех частей, каждая из которых формулируется как ответственность. Рассмотрим их.

- *Показать изображение в процессе загрузки.* Поиск начинается с вызова метода, демонстрирующего изображение "идет загрузка". Используемое изображение определяется объектом опций.

```
showLoadingImage: function() {
    var newImg = document.createElement('img');
    newImg.setAttribute('src', this.options.loadingImage );
    document.getElementById(
    this.options.resultsContainerId).appendChild(newImg);
},
```

0 *Сформировать URL операции поиска.* URL операции поиска формируется с использованием атрибута xmlURL, который был передан в момент создания с параметром q=, к которому присоединено значение, находящееся в текущий момент в поле поиска. Объединение (конкатенация) выполняется методом, проверяющим URL на существование предыдущей строки запроса (таким образом гарантируется использование правильных разделителей параметров).

```
appendToUrl: function(url, name, value) {
    var separator = '?';
    if (url.indexOf(separator) > 0; ) separator = '&';
    return url + separator + name + '=' + value;
},
```

© *Выполнить XSLT-обработку и обновить пользовательский интерфейс.* Поскольку мы предусмотрительно вынесли в отдельный метод задачу XSLT-обработки на стороне клиента, данная внешне сложная ответственность выполняется посредством единственной строки кода.

```
new XSLTHelper(searchUrl,
    this.xsltURL).loadView(this.options.resultsContainerId);
```

О *Обновить закладку.* После того как пользователь инициировал поиск, необходимо обновить закладку. Данная ответственность выполняется с помощью метода updateBookmark().

```
updateBookmark: function() {
    var container • document.getElementByIdt
        this.options.bookmarkContainerId);
    var bookmarkURL = this.appendToUrl(
        this.pageURL, 'q', this.lookupField.value );
    if ( container )
        container.innerHTML = '<a href^""' +
            bookmarkURL + '" >'+this.options.bookmarkText + '</a>';
}
```

Этот метод берет из объекта опций элемент container и текст закладки. URL сгенерированной закладки представляет собой значение, переданное конструктору в форме аргумента pageURL. К этому указателю добавляется параметр q= со значением текущего поиска. Согласно всем полученным значениям обновляется свойство innerHTML контейнера и получается соответствующий URL.

Если закладка была записана и использована для возврата на страницу, пользователь переходит на страницу с параметром q=someValue. Но что инициирует поиск, чтобы мы получили требуемый результат? Напомним, что в последней строке конструктора вызывался метод this.initialize();. Мы

еще не оговаривали, что этот метод делает, поэтому теперь самое время это уточнить. Как вы, возможно, догадались, метод initialize() требуется для поддержки концепции закладки. Реализация этого метода выглядит следующим образом:

```
initialize: function() {
    var currentQuery = document.location.search;
    var qlndex = currentQuery.indexOf('q=');
    if ( qlndex != -1 ) {
        this.lookupField.value =
            currentQuery.substring( qlndex + 2 );
        this.doSearch();
    }
Ь
```

Метод initialize() принимает текущее положение документа и ищет в строке запроса параметр q. Если он существует, метод выделяет значение параметра и передает его в поле поиска. Затем он инициирует поиск посредством метода doSearch(). Что и требовалось получить.

12.6.3. Выводы

Остановимся на мгновение и рассмотрим, чего же мы добились. Мы написали вспомогательный класс XSLTHelper, инкапсулирующий все тяжким трудом добытые знания по обеспечению поддержки XSLT в браузере клиента. Мы в полной мере использовали данный вспомогательный класс в компоненте 'живого" поиска. Мы написали простой, но настраиваемый компонент "живого" поиска, принимающий очень мало информации и превращающий Web-страницу пользователя в отзывчивое поисковое животное. При написании компонента мы придерживались классического объектно-ориентированного :тиля, иллюстрирующего простую структуру и разделение ответственности. В целом совсем неплохо для одного дня работы.

12.7. Резюме

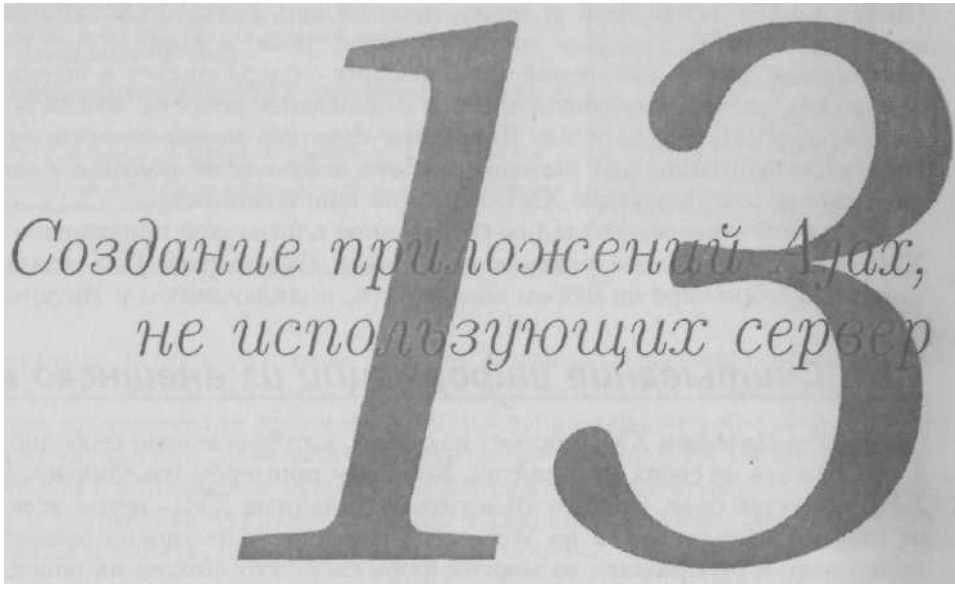
3 данной главе мы воспользовались обычной поисковой страницей и добашли в нее функциональные возможности Ajax. Главной особенностью напего поиска является то, что он не запрещает пользователю взаимодейство-!ать с окном во время выполнения обработки запроса на стороне сервера. Это означает, что в браузер можно, например, поместить анимацию. Налише контроля над браузером позволяет выполнять другие действия, так что ш можем быть уверенными, что пользователи приложения знают о том, что ЕОНСК выполняется.

Далее мы реализовали XSLT-обработку, превращающую наш документ i.ML в отформатированный HTML-код, который мы передаем свойству inегнтмь элемента div. Это позволяет отказаться от использования JavaScript ,ля динамической обработки узлов XML и создания большой строки, примеяемой к документу. Вместо этого в вопросе получения документа из XML-ода мы можем положиться на XSLT.

То, что мы использовали анимацию в процессе обработки и XSLT при реализации "живого" поиска, совсем не означает, что эти концепции нельзя применить к другим проектам. Описанные возможности добавляются и к обычным транзакциям с сервером. Пользователи часто говорят, что какой-то процесс требует нескольких минут. Можем ли мы как-то показать подобное сообщение? Например, с помощью свойства innerHTML из данного проекта мы можем легко добавить на страницу сообщение или изображение, указывающее пользователю, что поиск выполняется. Вообще, практически в любом приложении Ajax следует указывать, что затребованное действие выполняется, чтобы пользователь не щелкал повторно на кнопке отправки или обновления.

С помощью XSLT можно задавать стиль сообщений RSS или изменять любой другой описанный в книге проект, используя на стороне клиента вместо циклического прохода DOM XML возможности XSLT.

В четырех примерах, приведенных в этой и предыдущих главах, мы реализовали собственные серверные процессы обслуживания наших клиентов Ajax. В последней главе мы изучим клиент Ajax, сообщающийся непосредственно со стандартной Web-службой: подачей новостей RSS.



В этой главе...

- Обработка RSS-лент
- Использование Ajax для непосредственного доступа к Web-сайтам
- Ajax без сервера
- Эффекты перехода

В четырех предыдущих главах мы разработали широкий спектр аккуратных приложений Ajax и привели соответствующие коды клиентов и серверов. В данной главе мы не будем представлять код серверной части приложения, поскольку его просто не будет. Возможно, для многих это будет сюрпризом, но приложения Ajax можно запускать для обработки потока данных, формат которых совпадает с форматом ответа Web-сервера. В частности, для Web-страницы мы можем создать богатый пользовательский интерфейс, действующий как клиентское приложение.

Многие Web-сайты предлагают ленты сообщений XML в форматах RSS (Really Simple Syndication — очень простой синдикат), RDF (Resource Descriptor Framework — каркас дескриптора ресурсов) и Atom — трех наиболее популярных лентах сообщений. Информация, содержащаяся в подобных наборах лент, может представлять собой ежедневные новости, комиксы, блоги, шутки, прогноз погоды и т.д. Благодаря Ajax мы можем получать объединенную информацию, не посещая все эти сайты и не покупая клиентское приложение, считывающее XML-ленты на наш компьютер.

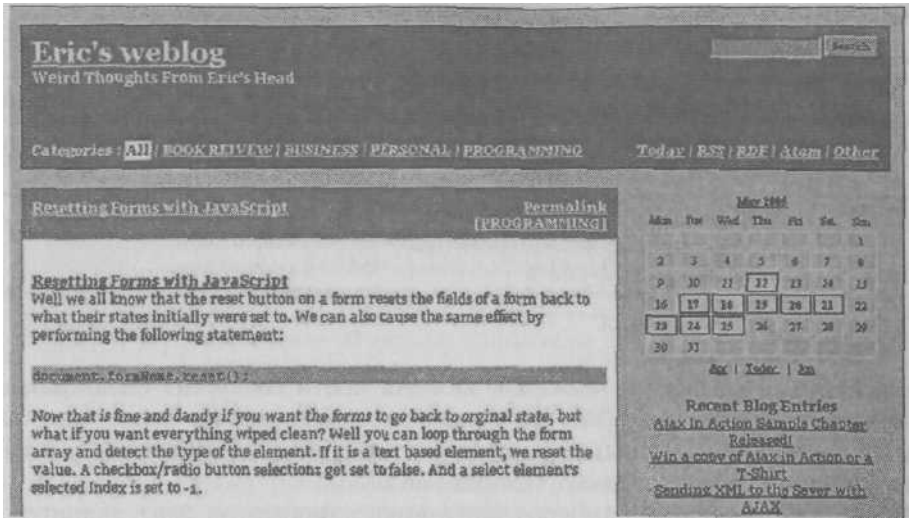
В данной главе мы создадим собственное клиентское приложение чтения XML-лент, получающее ленты с различных Web-сайтов. Его можно будет запускать в браузере на любом компьютере, подключенном к Интернету.

13.1. Считывание информации из внешнего мира

Объединенная лента XML состоит из статей, которые можно свободно читать и отображать на своих Web-сайтах. Хорошим примером объединенной ленты XML является блог. Следует отметить, что данные XML-ленты отличаются от старых добрых статей на Web-сайте, поскольку их можно совместно использовать и отображать во многих форматах. Это похоже на поиск газеты или журнала, точно соответствующих размерам стола, за которым вы завтракаете. Благодаря этому вы можете есть и читать, ничего не передвигая и не рискуя разлить утренний кофе. Объединенные ленты можно отформатировать в любом стиле, удовлетворяющем вашим потребностям, что позволяет получать только ту информацию, которая вам действительно требуется.

Содержимое объединенных лент XML можно просматривать несколькими способами. Например, если данную задачу нужно решить для одного блога, мы можем зайти на Web-сайт, где расположена XML-лента. Чтобы одновременно просматривать несколько лент без обращения к нескольким сайтам, можно, например, воспользоваться узлами, подобными JavaCrawl.com. Кроме того, RSS-ленту можно просматривать в необработанном виде, непосредственно открывая исходный XML-файл в Web-браузере, или использовать специальное программное обеспечение для структурирования и форматирования лент, которые вам требуются.

Помимо перечисленных вариантов, для просмотра лент можно использовать Ajax. Без Ajax нам потребовалось бы загрузить клиентское приложение или зайти на Web-сайт, чтобы получить необходимую информацию, но благодаря Ajax нам этого делать не придется. Вместо этого мы разработаем RSS-клиент на основе JavaScript, который можно будет запустить прямо на



Рис, 13.1. Блог Эрика Паскарелло, предлагающий RSS-ленту

рабочем столе, используя только браузер. В данной главе мы создадим приложение, получающее несколько лент RSS из нескольких источников и позволяющее упорядоченно просматривать поступающую информацию. Кроме того, для улучшения привлекательности пользовательского интерфейса мы интегрируем в приложение эффекты переходов. Однако, прежде чем мы приступим к разработке приложения Ajax, необходимо разобраться в формате XML-ленты и понять, где эти ленты можно взять.

13.1.1. Поиск XML-лент

Одним из наиболее популярных мест для поиска XML-лент являются блоги. Нынешняя популярность блогов объясняется множеством причин. Они могут содержать различную информацию: личные записи, мысли, новости, шутки или обсуждение технологий. Рассмотрим один из примеров — блог Эрика Паскарелло, показанный на рис. 13.1.

Данный блог является доступным, т.е. зайти на него может кто угодно. Хотя многие пользователи просто читают информацию этого блога, находящегося на сайте JavaRanch, блог доступен и в виде XML-ленты. Для чтения этой ленты можно обратиться к ее "родной" XML-форме либо зайти на другой Web-сайт (например, JavaLobby), который получает объединенную ленту и отображает ее в собственном формате.

На рис. 13.1 в правом верхнем углу экрана видны ссылки на RSS, RDF и Atom. Как отмечалось ранее, данные форматы XML-лент являются наиболее распространенными.

Все указанные ленты имеют различные спецификации XML. Это означает, что, изучив соответствующие XML-файлы, мы обнаружим различные схе-

Таблица 13.1. Необходимые элементы канала

Элемент	Описание	Пример
description	Фраза, описывающая канал	Странные мысли из головы Эрика
link	URL на Web-сайт HTML, с которым установлен канал связи	http://radio.javaranch.com/pascarello
title	Имя канала, а также название службы. Должно ассоциироваться с названием Web-сайта	Блог Эрика на JavaRanch.com

мы именования элементов. Вообще, каждая лента содержит спецификацию, уточняющую, какая информация должна определяться в ленте. Поскольку мы имеем дело с различными форматами, проще всего выбрать один из них и разработать для него клиент чтения лент.

Поскольку одним из наиболее популярных форматов лент является RSS (Really Simple Syndication — очень простой синдикат), далее мы будем изучать только RSS-ленты.

13.1.2. Изучение структуры RSS

Прежде чем приступить к разработке XML-клиента Ajax для чтения лент RSS, рассмотрим структуру RSS-файла. Знание структуры позволит более эффективно исследовать DOM XML и получить информацию, которую нам необходимо отобразить. Итак, документ RSS состоит из двух частей: канала и статей. Канал предоставляет информацию о том, откуда поступает лента; назначение статей понятно из названия.

Составляющие элементы канала

В определенном смысле канал можно рассматривать как заголовок RSS-ленты. Элементы канала сообщают пользователю, откуда поступает лента RSS, как она называется, когда обновлялась последний раз, и т.д. Как показано в табл. 13.1, спецификация RSS требует задания крайне малого числа элементов.

Данные три необходимых элемента предоставляют основную информацию о RSS-ленте. Они сообщают, откуда идет **RSS-канал,** как он называется и о чем он. Если нам понадобится другая информация о ленте, потребуется обратиться к дополнительным элементам.

Лента RSS может содержать любое число из указанных в табл. 13.2 дополнительных элементов канала. Разработчик ленты может не включать их в свой продукт, включить один, два или все шестнадцать.

Некоторые опции включают адреса электронной почты, которые потребуются при возникновении проблем, касающихся содержимого или структуры. Кроме того, имеется информация, объясняющая, когда обновляется лента.

Таблица 13.2. Дополнительные элементы канала

Элемент	Описание	Пример
category	Задает, к какой категории принадлежит канал	Программирование
cloud	Позволяет регистрировать процессы с атрибутом cloud, чтобы они уведомлялись об обновлении канала. Таким образом, реализуется облегченный протокол публикации-подписки	
copyright	Уведомление об авторских правах на содержимое канала	Эрик Паскарелло
docs	URL, указывающий на документацию по лентам RSS	http://backend.userland.com/rss
generator	Строка, указывающая, какая программа использовалась для генерации протокола	Pebble
image	Задает изображение, которое можно отображать вместе с лентой	http://pebble.soundforge.net/common/images/powered-by-pebble.gif
language	Язык, на котором написан канал	En
lastBuildDate	Время последнего изменения содержимого	
managing editor	Электронный адрес редактора, отвечающего за содержимое	pascarello@javaranch.com
pubDate	Дата публикации содержимого	
rating	Рейтинг PICS данного канала. См. http://www.w3.org/PICS/	
skipDays	Информирует агрегаторы (программы сбора и чтения RSS-лент), в какие дни они могут не заниматься поиском обновлений	
skipHours	Информирует агрегаторы, в какие часы они могут не заниматься поиском обновлений	
textInput	Задает поле текстового ввода, которое может отображаться	
ttl	Указывает время жизни (Time to Live— TTL), или число минут, в течение которых данные канала могут котироваться, не требуя обновления	
webmaster	Адрес электронной почты администратора, отвечающего за технические вопросы	webmaster@javaranch.com

Необходимые и дополнительные элементы канала описывают содержимое RSS-ленты, позволяя нам определять ее характеристики. Подобно составляющим канала, составляющие статей также делятся на необходимые и необязательные.

Элементы статей

Как газета состоит из множества статей, так и лента RSS может содержать множество составляющих элементов. Каждый элемент статьи должен иметь хотя бы один из следующих компонентов: заголовок или описание. Согласно спецификации RSS необходимым является только один из них, хотя допускается использование обоих.

Кроме того, существует восемь дополнительных элементов, которые могут добавляться к статье. Возвращаясь к аналогии с печатной прессой, можно сказать, что в газетной статье обычно можно выделить такие элементы; материал, фамилия автора, источник и заголовок. Точно так же каждый элемент статьи в ленте RSS может содержать свои заголовки, имена авторов, источники и т.д. Все дополнительные элементы, которые может содержать статья, показаны в табл. 13.3.

Сердцем и душой ленты RSS являются заголовок и описание. Заголовок (titile) позволяет понять, о чем идет речь в статье, а описание (description) может представлять собой одно из двух: краткий конспект статьи или всю статью. В настоящее время не существует стандарта, определяющего, как должен использоваться элемент description. Чтобы определить, что же с ним делать, необходимо изучить каждую ленту, а лишь затем приступать к написанию RSS-клиента. Если это конспект — его можно сравнить с аннотацией на обложке журнала, в которой сказано: "подробности см. на с. 10". Вот именно здесь нам потребуется элемент link, представляющий собой URL всей статьи на сайте автора.

Большинство RSS-лент пытается использовать максимальное число дополнительных элементов, поскольку это помогает разработчикам (например, нам) создавать максимально надежные RSS-приложения. Чем больше данных предоставлено, тем лучше можно отобразить содержимое RSS-ленты. Подробнее о спецификации RSS рассказывается на сайте http://backend.userland.com/rss.

Разобравшись с основными элементами документа RSS, можно приступать к разработке собственного RSS-клиента на основе Ajax.

13.2. Богатый пользовательский интерфейс

В данной главе мы создадим программу просмотра RSS-лент, получающую ленты XML с Web-сайта без использования серверных технологий, подобных •NET или JSP, или клиентского приложения. Ajax позволяет просматривать информацию в виде Web-страницы, записанной на рабочем столе. Данный пример показывает, что инфраструктура Ajax не обязательно требует Web-сервера, использующего технологию, подобную .NET или JSP. Если компью-

Таблица 13.3. Элементы статьи

Элемент	**Описание**	**Пример**
author	Адрес электронной почты автора позиции	Pascarello@javaranch. com
category	Включает позиции в одну или несколько категорий	Программирование
comments	URL страницы с комментариями, касающимися данной позиции	http://radio.javaranch.com/pascarello/2005/05/25/1117043999998.htmltcoiranents
description	Резюме	Ajax позволяет разработчикам улучшать пользовательский интерфейс, делая Web-приложение похожим на клиентское приложение
enclosure	Описывает медиа-объект, присоединенный к статье	<enclosure url="http://radio. javaranch.com/pascarello/media/TheAj axInActionSong.mp3 "length="5908124"type="audio/mpeg"/>
guid	Строка, представляющая собой уникальный идентификатор	http://radio.javaranch.com/ pascarello/2005/05/25/1117043999998.html
link	URL статьи	http://radio.javaranch.com/pascarello/
pubDate	Дата опубликования статьи	Среда, 25 мая 2005 г., 17:59:59 GMT
source	Канал RSS, по которому поступает статья	<source url="http://radio. javaranch.com/pascarello/blog-xmls-Eric's Blog</source>
title	Заголовок элемента	Ajax улучшает разработку пользовательского интерфейса

тер подключен к Интернету, мы можем обращаться к RSS-лентам любых выбранных сайтов. (Если вы пользуетесь браузером Mozilla, см. раздел 13.6.1. Прежде чем вы сможете использовать код, представленный в данном проекте, вам потребуется обойти ограничения безопасности Mozilla, рассмотренные в главе 7.)

13.2.1. Чтение лент

Если вам приходится ежедневно просматривать несколько Web-сайтов, с помощью нашего приложения вы сможете обойтись без этого утомительного процесса. Используя предлагаемую программу просмотра, вы получите необходимое число лент на одной странице.

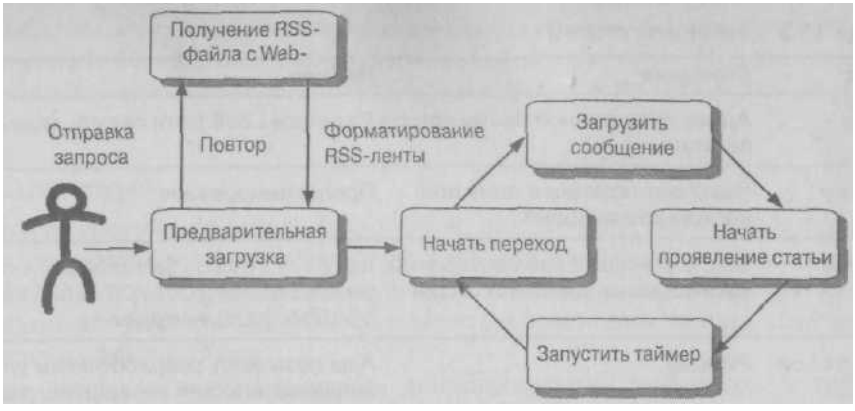


Рис. 13.2. Последовательность действий программы чтения RSS-пент

Уникальность нашего приложения заключается в том, что оно не использует никакого серверного кода; мы только получаем XML-документы RSS, создаваемые другими Web-сайтами. Все приложение размещается на Web-странице, записанной в среде рабочего стола или загружаемой как часть нашего Web-сайта.

Прежде всего необходимо понять, на какие этапы разбивается разработка требуемого приложения. Мы разрабатываем программу чтения RSS-лент, которая настраивает слайд-шоу, использующее два слоя. Каждый слой будет содержать одну ленту, которая будет показываться заданный период времени, после чего будет проявляться вторая лента. Алгоритм действий данного приложения показан на рис. 13.2.

Процесс чтения лент состоит из нескольких этапов Первый этап — это загрузка нескольких выбранных лент. Для хранения информации, необходимой различным ист:очникам лент, мы будем применять общий массив. Кроме того, мы не будем использовать дополнительные элементы статей, приведенные в табл. 13.2.

После загрузки файлов нам придется создать упомянутый выше эффект перехода (постепенное проявление и затухание). В данном случае мы используем для этого классы CSS. Переключение между сообщениями и циклический показ сообщений будет реализован с помощью таймера.

Мы также создадим в приложении кнопки "Вперед", "Назад" и "Пауза". Кроме того, можно добавить возможность вставки дополнительных потоков из предлагаемого списка. А начнем мы с создания на стороне клиента формы и слоев.

13.2.2. HTML-структура без таблиц

Самой объемной частью данного проекта является представление. Для создания структуры в форме таблицы, содержащей заголовок, тело и нижний колонтитул, мы используем ряд элементов div и span. Результат, который мы собираемся получить, показан на рис. 13.3.

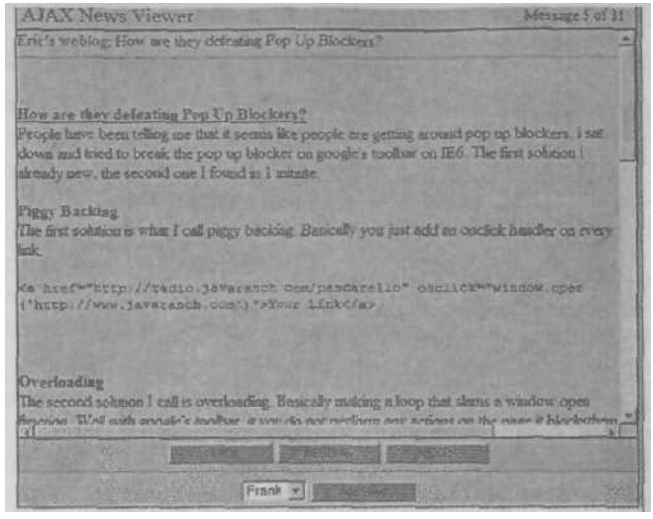


Рис. 13.3. Окончательный вид программы чтения RSS-лент

Для формирования структуры можно было бы использовать и таблицы, но такое решение было бы допустимым до начала эпохи CSS (см. главу 2). В настоящее время использование таблиц нежелательно, поскольку их визуализация требует больше времени, а изменять их сложнее, чем структуру на основе CSS. В листинге 13.1 показана разметка, на которой будет основываться структура нашей программы чтения XML-сообщений.

Листинг 13.1. Структура HTML-формы, используемой приложением

```
 <form name="Forml">
<!— O Внешний элемент div —>
  <div id="divNewsHolder">
<!— © Элемент div заголовка —>
    <div id="divNewsTitle">
<!— © Счетчик лент —>
      <span id="spanCount">Loading</span>
<!— O Название —>
       Ajax News Viewer
    </div>
<!— © Контейнер для лент новостей —>
    <div id="divAd">
<!— 0 Слой первой ленты новостей —>
      <div id="divNewsl">
        Loading Content...
      </div>
<!—  O Нижний колонтитул структуры —>
      <div id="divNews2">
        Loading Content...
      </div>
    </div>
<i—  0 —>
    <div id="divControls">
<!—  © Кнопки действия —>
      <input type="button" name="btnPrev"
```

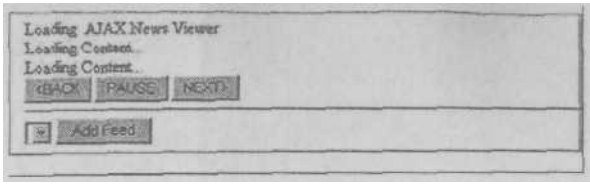


Рис. 13.4. Элементы HTML без стилей CSS

```
        id="btnPrev" value="<BACK" />
      <input type="button" name="btnPause"
        id="btnPause" value="PAUSE" />

      <input type="button" name="btnNext"
        id="btnNext" value="NEXT>" />
      <hr/>
<!—   © Дополнительный элемент лент —>
      <select name="ddlRSS">
      </select>
      <input type="button" name="btnAdd"
        value="Add Feed" />
    </div>
  </div>
</form>
```

Первым элементом div является divNewsHolder O — контейнер, используемый для задания общего размера окна, в котором будет отображаться результат выполнения приложения. Следующий элемент div — divNewsTitle © — вмещает заголовок нашей структуры. Внутри этого элемента div мы добавляем элемент span ©, содержащий заполнитель-счетчик лент. Следующая строка текста O — это название нашего приложения. В данной строке мы можем написать все, что угодно.

Далее идет элемент divAd ©. Это строка представляет собой заполнитель, содержащий информацию о RSS-ленте, которую мы извлечем позже. В элемент divAd мы помещаем еще два элемента div, divNewsl © и divNews2 ©, которые используются для хранения информации RSS-ленты. Позже мы изменим свойства CSS данных элементов с помощью JavaScript, чтобы создать эффект затухания.

Строка нижнего колонтитула создается с помощью элемента div divControls ©. Она содержит средства навигации и функции управления лентой. В данный элемент div добавляются кнопки "Вперед", "Назад" и "Пауза" ©. Чтобы пользователь мог выбирать дополнительные XML-ленты, добавляются элемент формы выбора и еще одна кнопка ©. Таким образом, мы получаем каркас приложения, показанный на рис. 13.4.

Рис. 13.4 выглядит не очень привлекательно, поскольку мы еще не отформатировали элементы HTML, но с введением правил CSS это изменится. Изучая рис. 13.3, видим, что наши элементы div (divNewsl и divNews2) должны накладываться друг на друга, чтобы мы смогли правильно реализовать эффект затухания.

13.2.3. Гибкое CSS-форматироеание

Без CSS наши Web-страницы выглядели бы так, как показано на рис. 13.4: серыми и неприглядными. Чтобы улучшить данный пример, мы применим к элементам правила стилевого оформления CSS. Стили позволяют легко редактировать свойства в будущем, не требуя редактирования HTML-файла. Первым элементом, для которого мы разработаем стиль, будет внешний контейнер div и строка заголовка.

Применение стиля к контейнеру и заголовку

Упомянутый ранее элемент divNewsHolder можно считать контейнером для нашего приложения. Он позволяет размещать программу чтения RSS-лент на странице и задавать ее ширину. Поскольку для представления остальных строк мы используем элементы div, они будут занимать 100% доступной им ширины страницы. Задавая ширину в контейнере, мы можем жестко задать размер остальных элементов, что облегчит их последующие обновления. Реализация сказанного с помощью CSS показана в листинге 13.2.

Листинг 13.2. Правила CSS для контейнера и заголовка

```
//О Элемент div контейнера
#divNewsHolder{
  width: 6ООpx;
  border: 2px solid black;
  padding: Opx;
)
// © Элемент div заголовка
#divNewsTitle{
   font-size: 20px;
  height: 26px;
  background-color: #BACCD9;
// © Высота строки
  line-height: 26px;
J
// О Счетчик
#spanCount{
// © Структура основана на "плавающих" элементах

  float: right;
  font-size: 15px;
  padding-right: 10px;
>
```

Для определения стиля элементов формы вызываются идентификаторы этих элементов со знаком # О. Таким образом указывается, что стиль необходимо связать только с элементом div, имеющим идентификатор divNewsHolder. В данном случае для элемента divNewsHolder задаются ширина и граница, а область заполнения устанавливается равной 0.

Задав стиль контейнера, можно форматировать его первую строку. Для элемента divNewsTitle нам хотелось бы установить высоту, цвет фона и размер шрифта ©. Значение свойства line-height © задается равным высоте

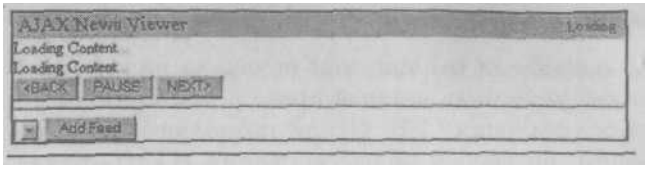


Рис. 13.5. К элементам div контейнера и заголовка применены правила CSS

элемента div. Таким образом, мы гарантируем, что наша строка текста высотой 20 пикселей будет правильно вертикально центрирована в элементе div. Если не задавать высоту текста, он будет размещен возле верхней границы элемента div.

Последним этапом форматирования строки заголовка будет перемещение элемента spanCount О к правому краю заголовка {изначально он находится перед заголовком). Для этого используется свойство float ©, значение которого устанавливается равным right. Вследствие этого наш текст выравнивается по правому краю, независимо от ширины элемента-контейнера (причем на заголовок данное правило не влияет). Размер шрифта основного текста можно сделать немного меньше, чтобы он не так бросался в глаза, как заголовок. Свойство padding-right определяет смещение текста от правого края, чтобы он не соприкасался с границей контейнера. Теперь стилевое оформление контейнера и заголовка завершено; результат его применения показан на рис. 13.5.

На рис. 13.5 видно, что строка заголовка очень отличается от других строк, стиль которых мы еще не определили. Слово Loading располагается в правой части элемента div, а текст внутри элемента div центрирован по вертикали. Граница элемента div, представляющего контейнер приложения, окружает все остальные элементы.

Далее мы займемся стилевым оформлением оставшихся элементов div.

Стилевое оформление содержимого окна

Следующий этап — определение стиля среднего участка, или тела, нашего приложения (листинг 13.3). Раздел тела будет содержать отформатированную информацию RSS-ленты. Элементы div divNewsl и divNews2 наложатся друг на друга; это **нужно** для создания эффекта перехода. Для реализации этого эффекта мы будем так увеличивать прозрачность слоя, чтобы постепенно становился виден слой, расположенный под ним.

Листинг 13.3. Правила CSS для элементов div тела

```
// О Отформатировать divAd
#divAd{
// © Задать ширину-
  width: 100%;
// © Задать высоту
  height: 400px;
// О Скрыть полосу прокрутки
  overflow: hidden;
// © Отформатировать границы
  border-top: 2px solid black;
```

```
  border-bottom: 2px solid black;
}
// © Задать стиль обоих элементов div новостей

#divNews1, #divNews2{
// в Задать ширину и высоту
  width: 100%;
  height: 40Opx;
  background: #D9CCBA;
// О Задать относительное расположение
  position: relative;
// © Если необходимо, показать полосы прокрутки

  overflow: auto;
// © Придвинуть элементы div к краю
  left: Opx;
1
//О Разместить первый элемент div
#divNewsl{top: Opx;}
// © Разместить второй элемент div
#divNews2{top: -4Q0px;}
```

Прежде всего нам требуется определить стиль элемента divAd О, представляющего собой контейнер элементов span RSS-лент. Ширина © устанавливается равной 100%, а высота © — 400px. Нам не нужно, чтобы в данной строке использовалась полоса прокрутки, поэтому значение свойства overflow О устанавливается равным hidden. Это означает, что если ширина какого-то элемента будет больше 400 пикселей, он будет частично не видим. Другие элементы div внутри данного контейнера позволяют использовать прокрутку, поэтому содержимое мы не потерям. После этого мы задаем стиль верхней и нижней границ © в виде черных линий шириной 2 пикселя. Боковые границы не требуются, поскольку для общего контейнера задана ненулевая границы. Если бы мы задали границу вокруг всей строки, ее ширина по бокам была бы равна 4 пикселям, а сверху и снизу — всего 2 пикселям, т.е. выглядела довольно несуразной.

Далее необходимо отформатировать два элемента div, вмещающих содержимое, — divNewsl и divNews2 ©. Их свойства можно задать одновременно, разделив их идентификаторы запятой. Значения ширины и высоты © задаются такими же, как у элемента div внешнего контейнера. Задание относительного (relative) положения элементов div © позволяет размещать их относительно родительского контейнера divAd, в отличие от абсолютного (absolute) размещения, привязанного к левому верхнему углу окна браузера. Свойство overflow элементов div © задается равным auto, что позволяет при необходимости отображать полосы прокрутки. Последний этап — установка левой позиции элементов © равной 0 пикселей, что позволяет выравнивать элементы div, чтобы между их краями не было промежутков.

Нам требуется, чтобы два элемента div, содержащих новости, располагались один поверх другого. Поскольку используется относительное позиционирование, с этими двумя элементами необходимо соотнести различные

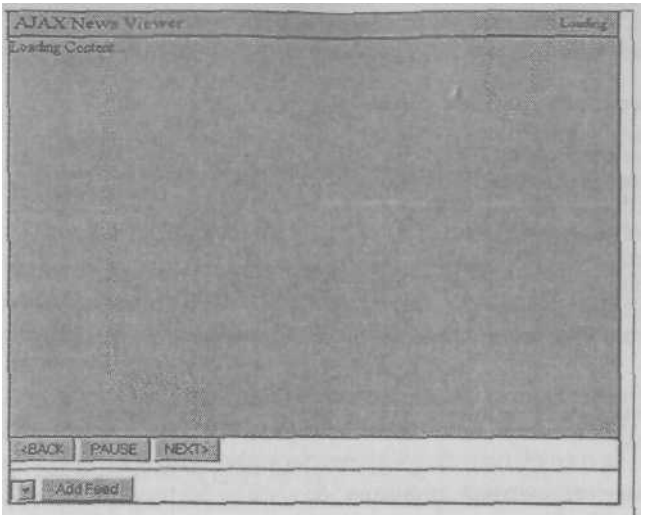


Рис. 13.6. Применение правил CSS к элементам div тела

свойства положения. Поэтому вертикальное положение элемента divNewsl О устанавливается равным 0 пикселям. Таким образом, этот элемент будет прилипать к верхней границе родительского элемента div. Положение элемента divNews2 © устанавливается равным -400px. Почему мы используем отрицательное значение? Дело в том, что, как показано на рис. 13.4 и 13.5, второй элемент div располагается ниже по странице, чем первый элемент div. Поскольку высоту контейнера мы установили равной 400 пикселям, элемент divNews2 необходимо поднять на эти 400 пикселей, чтобы он выровнялся по верху родительского элемента div (так же, как divNewsl). Обращаясь к рис. 13.6, мы видим, что, в отличие от рис. 13.5, два элемента div теперь накладываются друг на друга.

Поскольку два элемента div располагаются один поверх другого, мы видим только содержимое одного из них. В данном случае уровень прозрачности установлен равным 100%, поэтому содержимое нижнего контейнера не видно. К такой конфигурации мы должны прийти после завершения перехода, однако, прежде, чем мы рассмотрим этот вопрос, давайте закончим разработку стилей нашего приложения.

Настройка нижнего колонтитула

Последним блоком, для которого мы определим правила CSS, является нижний колонтитул. Для данного раздела мы установим цвет фона и стандартизуем элементы формы, чтобы структура раздела была более отчетливой. Для этого мы задаем цвета, размер шрифта и размер кнопок (листинг 13.4).

Листинг 13.4. Правила CSS для элемента div нижнего колонтитула

```
// О Определить стиль элемента div нижнего колонтитула
// © Задать цвет фона
#divControls{
  background-color:  #BACCD9;
```

```
// © Центрировать текст
  text-align: center;
//O Добавить заполнение
  padding-top: 4px;
  padding-bottom: 4px;
}
// © Определить стиль элементов формы
#divControls input{
// © Установить размер и цвета
  width: lOOpx;
  background-color: #8C8C8C;
  color: #000000;
  font-size: Юpx;
}
```

С элементом div нижнего колонтитула divControls мы соотносим такие правила CSS O, чтобы этот элемент согласовывался со строкой заголовка. Цвет фона © данного элемента выбран таким же, как у заголовка. Текст горизонтально выравнивается по центру элемента ©. Далее добавляется заполнение сверху и снизу O, чтобы содержимое не прилипало к границе. Добавлять границу к данному элементу div не обязательно, поскольку для средней строки определена верхняя граница, а три остальные границы определены для внешнего контейнера.

Последним этапом CSS-форматирования нижнего колонтитула будет применение стилей к элементам формы, чтобы они согласовывались с общим стилем приложения. Для обращения к элементам кнопок ©, расположенным в divControl, используется структура, сформированная следующим образом: имя элемента div, пробел и имя дескриптора. Это означает, что указанные свойства будут сопоставлены только с элементами, расположенными в данном дескрипторе div. Все остальные элементы страницы с таким же именем дескриптора этих свойств не получат.

Поскольку текст всех кнопок имеет разную длину, то свойство width © для кнопок задается так, что теперь все они будут одинаковой ширины, т.е. выглядеть более однородно. Мы изменили цвет фона, и теперь он не совпадает с принятым по умолчанию в системе пользователя. Кроме того, можно задать цвет текста и размер шрифта текста элемента. На рис. 13.7 показан окончательный вид нижнего колонтитула — теперь он лучше согласуется с внешним видом всего приложения.

На рис. 13.7 видны все свойства, соотнесенные с элементами div. Мы определили размеры, цвета, кегли, границы, заполнение и многое другое. Свойства CSS этих элементов можно настроить так, чтобы они соответствовали любой заданной теме Web-сайта или личным предпочтениям. Далее нам необходимо получить текст для нашего RSS-клиента!

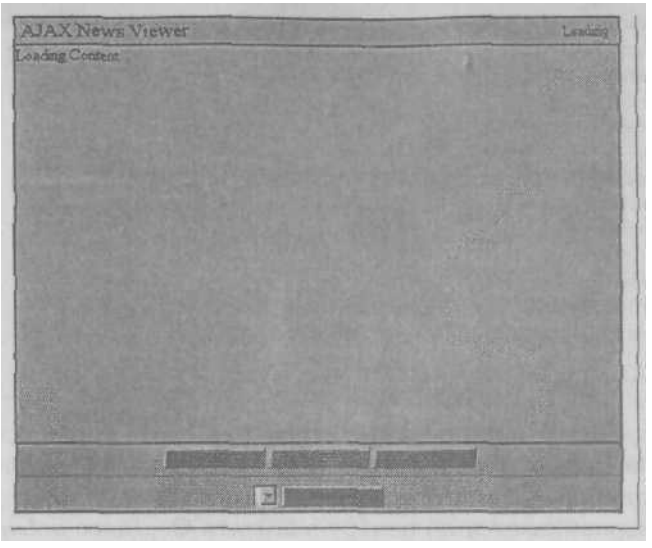


Рис. 13.7. Применение правил CSS к нижнему колонтитулу

13.3. Загрузка RSS-лент

В данном примере мы будем загружать файлы из нескольких лент. Для этого будет задействован объект ContentLoader (как и во всех остальных примерах книги). В первой версии разрабатываемого приложения просмотра RSS-лент мы используем ряд глобальных переменных.

13.3.1. Глобальный уровень

Глобальные переменные позволяют легко настраивать сценарий, чтобы нам не приходилось менять функциональные возможности внутри циклов for и таймеров. Возможность настройки содержимого глобальных переменных позволит вносить глобальные изменения в сценарий и осуществлять связь различных функций. Сейчас нам удобнее использовать именно глобальные, а не локальные переменные, поскольку они "открыты" для всех функций и можно не думать о возможности передачи информации из одной функции в другую. Позже в данной главе мы реструктуризируем сценарий и представим решение без использования глобальных переменных; сейчас же они просто облегчают нам разработку.

Одним из недостатков JavaScript является отсутствие типов переменных Для констант, поэтому мы имитируем их с помощью глобальных переменных. Однако следует помнить, что значения глобальных переменных могут перезаписываться, поэтому использовать их следует очень осторожно. Переменные, приведенные в листинге 13.5, в процессе работы нашего приложения никак меняться не будут.

Листинг 13.5. Глобальные переменные с постоянным значением

```
var flipLength = 5000; var timeColor = 100;
var strErrors = "<hr/>Error List: <br/>"
var arrayRSSFeeds = new Array(
   "http://radio.javaranch.com/news/rss.xml",
   "http://radio.3avaranch.com/pascarell0/rss.xml",
   "http://radio.]avaranch.com/bear/rss.xml",
   "http://radio.3avaranch.c0m/lasse/rss.xml");
```

В листинге 13.5 мы определяем глобальные переменные, задающие функционирование нашего приложения. Первая глобальная переменная, flipLength, определяет, сколько миллисекунд будет отображаться текущее сообщение до того, как оно заменится следующим. В данном случае время, проходящее между сменой сообщений, составляет 5000 миллисекунд. Следующая переменная-таймер — timeColor — задает время в миллисекундах между этапами "раскрашивания" нашего сценария. Чем больше ее значение, тем дольше переход.

Далее определяется глобальная переменная strErrors. Здесь формируется заголовок сообщений об ошибках, возникающих в процессе загрузки Стиль данного сообщения можно изменить; кроме того, с ним можно связать параметры стиля. Последняя глобальная переменная, влияющая на выполнение сценария, представляет собой массив arrayRSSFeeds. В данный массив мы добавляем URL лент RSS, которые нас интересуют. В данном случае мы обращаемся к четырем различным лентам RSS из радиоблогов JavaRanch.com.

Следующий набор глобальных переменных, представленный в листинге 13.6, используется для передачи данных между отдельными функциями Эти переменные хранят состояние приложения чтения RSS-лент Их значения изменяются согласно выполняемому действию.

Листинг 13.6. Глобальные переменные, содержащие информацию о состоянии

```
var currentMessage => 0;
var layerFront = 2;
var timerSwitch;
var bPaused = false;
var arrayMessage = new Arrayt);
var intLoadFile = 0;
var bLoadedOnce = false;                                    ^ —
```

Первая глобальная переменная, значение которой задается в листинге 13.6, — currentMessage. Она отслеживает сообщение, просматриваемое *в* текущий момент. В каком-то смысле ее можно считать счетчиком, который обнуляется при достижении максимального числа записей. Следующая глобальная переменная layerFront содержит состояние наших слоев. Разрабатывая структуру RSS-клиента, мы наложили друг на друга два слоя. Состояние этих слоев отслеживается *в* переменной layerFront.

Переменная timerSwitch содержит таймер, определяющий, когда приходит время загрузки следующего фрейма. Если пользователь щелкает на кнопке паузы, таймер сбрасывается и изменяется состояние переменной bPaused. Булево значение этой переменной позволяет определять состояние таймера: true — пауза, false — запущен.

Глобальная переменная arrayMessage содержит отформатированные сообщения, извлекаемые из элементов RSS. Этот многомерный массив хранит всю информацию, которую мы желаем отобразить на экране. Как отмечалось ранее, элементы статей содержат больше информации, чем нам может потребоваться, поэтому мы выбираем только несколько интересующих нас позиций и отображаем их в переменной arrayMessage.

Последняя переменная, intLoadFile, приводит нас к следующему блоку кода. Она представляет собой счетчик, содержащий число текущих файлов, извлекаемых из массива arrayRSSFeeds в процессе предварительной загрузки.

Объявив все глобальные переменные, мы уже видим, к чему идет данный проект. Мы предварительно загружаем RSS-ленты из массива URL. Для отслеживания состояния этого процесса используется счетчик. В ходе предварительной загрузки из каждого XML-файла мы выбираем только необходимую информацию. После завершения процесса сообщения отображаются попеременно, плавно переходя друг в друга, — создается слайд-шоу, которым мы можем управлять (в частности, приостанавливать). Используя объявленные глобальные переменные, мы можем контролировать функциональные возможности сценария. Таким образом, мы логически подходим к разработке функции предварительной загрузки RSS-лент.

13.3.2. Предварительная загрузка средствами Ajax

Одной из проблем, с которой сталкиваются разработчики Ajax-приложений, является предварительная загрузка нескольких файлов, не требующая отправки слишком большого числа запросов к внешним Web-сайтам и решающая проблему приоритетов. Одним из решений является реализация очереди — и именно за это отвечает наш объект Ajax ContentLoader.

Формирование запросов

Объект ContentLoader позволяет механизму очереди упорядоченно обрабатывать запросы. В листинге 13.7 показано, что мы берем массив (заполненный в момент загрузки страницы; см. листинг 13.5), который содержит URL лент, и подготавливаем их для объекта ContentLoader.

Листинг 13.7. Функция JavaScript предварительной загрузки

```
window.onload = function(){ var loader= new Array()
  for(i=0;i<arrayRSSFeeds.length;i++){
    loader[loader.length] = new net.ContentLoader(arrayRSSFeeds[i],
      BuildXMLResults,BuildError);}}
```

Код, приведенный в листинге 13.7, срабатывает согласно обработчику событий onload. При загрузке страницы мы подготавливаем переменную массива loader, которая будет содержать все запросы к серверу. Мы циклически проходим по массиву arrayRSSFeeds и получаем все указатели URL RSS-лент, из которых желаем получать информацию. При каждой итерации мы увеличиваем массив loader, включая в него новый запрос ContentLoader. Затем передаем массиву URL ленты, функцию BuildXMLResults(), форматирующую содержимое, и функцию BuildError(), которая будет вызываться в случае ошибки. Разобравшись с запуском процесса загрузки, мы можем переходить к форматированию возвращаемых XML-лент.

Адаптация функции

В процессе запроса мы должны вызвать либо функцию BuildXMLResults() (если запрос прошел успешно), либо BuildError() (если возникли какие-либо проблемы). Функция BuildXMLResults О принимает XML-ленту и форматирует ее в удобный формат. Функция BuildError() записывает ошибку в список ошибок. Обе функции обновляют состояние, чтобы мы могли видеть ход процесса загрузки. Реализация описанной логики приведена в листинге 13.8.

Листинг 13.8. Форматирование XML-результатов в форме массива JavaScript

```
f-unction BuildXMLResults () {
  var xmlDoc = this.req.responseXML.documentElement;
  var RSSTitle =
  xmlDoc.getElementsByTagName('title')[0].firstChild.nodeValue;
  var xRows = xmlDoc.getElementsByTagName('item') ;
  for (iOO; iCXxRows . length; iC++) {
     intMessage = arrayMessage.length;
     arrayMessage[intMessage] = new Array(
     RSSTitle,
     xRowsfiC] .getElementsByTagName{'title') {0]
.firstChild.nodeValue,
     xRows[iC].getElementsByTagName('link') [0]
.firstChild,nodeValue,
     xRowsfiC].getElementsByTagName('description')[0]
.firstChild.nodeValue);
  }
  UpdateStatusO;
```

Функция BuildXMLResults (), приведенная в листинге 13.8, извлекает документ XML, обращаясь к свойству responseXML объекта запроса. Сохранив документ XML в локальной переменной xmlDoc, мы можем получить для ленты информацию о названии RSS-ленты. Для этого мы обращаемся к дескриптору элемента названия и берем значение первого дочернего узла.

Далее мы получаем элементы статей и подготавливаем цикл для прохода по полученному массиву, записанному в xRows. Последовательно проходя данный массив, мы можем создать многомерный массив, сохранив его в следующей ячейке нашего глобального массива arrayMessage. Этот глобальный массив содержит название RSS-ленты, а также заголовок, ссылку и описание

статьи. Указанный многомерный массив создается для каждого элемента статьи, записанного в xRows. Завершив обход документа, мы вызываем функцию UpdateStatus () (листинг 13 9) для показа пользователю текущего состояния процесса .

Листинг 13,9. Функция, сообщающая о ходе предварительной загрузки

```
function UpdateStatus(){
  intLoadFile++;
  if{intLoadFile < arrayRSSFeeds.length){
  document.get£lementById("divNews2").innerHTML -
    "Loaded File " + intLoadFile + " of "
    + arrayRSSFeeds-length + strErrors;
  Jelse if(intLoadFile >= arrayRSSFeeds.length && 'bLoadedOnce){
    document.getEleraentByldf"divNews2").innerHTML =
      "Loading Completed" + strErrors;
    if{arrayMessage.length == 0){
      alert("No RSS information was collected."); return false;
    }
    bLoadedOnce « true;
    var timerX = setTimeout("ChangeView()",1500);
  }
}
```

Как показано в листинге 13.9, функция UpdateStatus () выполняет следующее. Во-первых, отображает пользователю состояние объекта предварительной загрузки; во-вторых, определяет, требуется ли начать слайд-шоу. Итак, прежде всего мы увеличиваем на единицу значение глобальной переменной intLoadFile, обновляя счетчик файлов. Если значение intLoadFile меньше общего числа загружаемых файлов, для отображения состояния загрузки мы задаем свойство innerHTML верхнего слоя элемента divNews2 равным выходной строке.

Если счетчик файлов больше или равен числу файлов в массиве (а слайд-шоу еще не запущено), тогда мы можем запускать переходы между лентами. Однако перед началом слайд-шоу необходимо проверить, есть ли у нас вообще данные для отображения. Для этого проверяется длина отформатированного массива сообщений arrayMessage. Если сообщений нет, мы сообщаем об этом пользователю и завершаем выполнение функции, возвращая значение false.

Если имеются данные для отображения, значение bLoadedOnce устанавливается равным true и после небольшой паузы вызывается функция Changeview(). Пауза нужна для того, чтобы пользователь успел прочесть возможные сообщения об ошибках. Как отмечалось ранее, если функция-загрузчик столкнется с проблемами в процессе загрузки XML-документа, она вызовет функцию BuildError() (листинг 13.10).

Листинг 13.10. Обработка ошибок, сгенерированных XMLHttpRequest

```
function  BuildError(){
  strErrors += "Error:" + "" + "<br/>"; UpdateStatus();
}
```

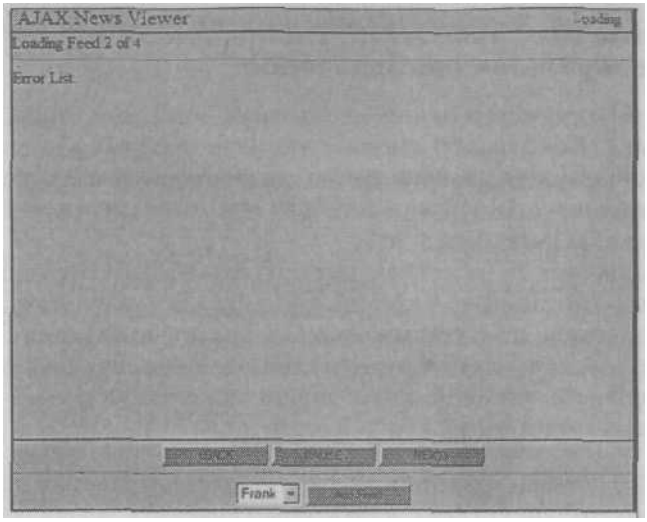


Рис. 13.8. Функция предварительной загрузки загружает второй файл из четырех, что видно по сообщению о состоянии и отсутствию ошибок

Функция BuildError() позволяет отображать на экране ошибку. Таким образом, пользователь узнает, что были загружены не все файлы. Возникающая ошибка просто добавляется к глобальной переменной strErrors, после чего вызывает функция UpdateStatus {), информирующая пользователя о текущем состоянии загрузки. Чтобы проверить, как работает функция предварительной загрузки, мы можем записать документ и открыть Web-страницу в браузере (рис. 13.8).

Тестируя приложение, мы должны наблюдать на экране процесс обновления состояния. На рис. 13.8 показан момент, когда функция предварительной загрузки загружает второй файл (из четырех) при отсутствии сообщений об ошибках. После загрузки всех файлов мы должны увидеть, что все они загружены успешно, а список ошибок пуст. Однако в строке состояния мы наблюдаем ошибку JavaScript, поскольку еще не создали функцию ChangeView{). Мы займемся этим в следующем разделе, однако вначале создадим эффект затухающего перехода, работающий во всех браузерах.

73.4. *Богатый эффект перехода*

Итак, мы написали код, загружающий файлы в массив. Теперь можно взять данные, хранящиеся в массиве, и создать слайд-шоу, основанное на DHTML. Изменяя содержимое элементов div с помощью inner HTML, можно отображать различные статьи, загруженные функцией предварительной загрузки. Изменяя CSS-классы элементов и значение параметра z-Index слоев, можно создавать затухающие переходы между элементами div. Объединив все это, мы сможем создать динамическое слайд-шоу с затухающими переходами.

13.4.1. Правила прозрачности, учитывающие индивидуальность браузеров

При создании эффекта затенения необходимо изменять прозрачность верхнего слоя. Таким образом, мы сможем увидеть содержимое слоя, находящегося под текущим. Когда степень непрозрачности равна 0%, будет видно все содержимое нижнего слоя. При 100%-ной непрозрачности нижний слой совершенно не просматривается.

Далее мы, как всегда, должны рассмотреть вопрос совместимости нашего кода с браузерами Internet Explorer и Mozilla. В указанных браузера! прозрачность реализована по-разному, поэтому мы должны учесть эти различия. В Mozilla применяется параметр непрозрачности (opacity), тогда как в Internet Explorer используется фильтр, задающий прозрачность через значение параметра альфа (листинг 13.11).

Листинг 13.11. Классы CSS фильтра прозрачности

```
.opac0{opacity: .0;filter: alpha(opacity=0);}
.opac1{opacity: .2;filter: alpha{opacity=20);}
.opac2{opacity: .4;filter: alpha(opacity=40);}
.opac3{opacity: .6;filter: alpha(opacity=60);}
.opac4{opacity: .8;filter: alpha(opacity=80);}
```

В листинге 13.11 показано, что мы создали ряд правил, определяющих стили с различными уровнями прозрачности. Использование для изменения значений правил CSS, а не JavaScript, —- вопрос личных предпочтений. Применяя CSS, мы можем изменять и другие свойства; возможно, вместе с модификацией степени прозрачности нам захочется изменить цвета или увеличить размер шрифта. Благодаря классам CSS мы можем обойтись без дополнительного JavaScript-кода, а также свести в одном месте весь код, учитывающий отличия браузеров.

Завершив разработку классов, мы можем начинать процесс загрузки информации RSS-ленты в элементы div.

13.4.2. Реализация затухающего перехода

Тестируя код в разделе 13.3.2, мы получили сообщение об ошибке, поскольку не создали функцию ChangeView(). Эта функция инициирует процесс затухающих переходов элементов div с текстами статей друг в друга. Чтобы процесс затухания проходил верно, мы изменяем классы CSS и размещаем элементы div на уровнях с разными значениями параметра z-lndex. Реализация сказанного показана в листинге 13.12.

Листинг 13.12. Функция ChangeView()

```
// О Объявить ChangeView()
function ChangeView(){
// © Отобразить название RSS-ленты
  strDisplay - "<span class='RSSFeed'>" +
    arrayMessage[currentMessage][0] + "</span>: "
// © Показать заголовок элемента
```

```
  strDxsplay += "<span class='itemTitle'V +
    arrayMessage[currentMessage][1] + "</spanxhr>";
// O Вставить описание статьи
  strDisplay += arrayMessage[currentMessage][3]
    + "<hr>";
// © Выдать URL ленты
  strDisplay += "<a href=1" +
    arrayMessage[currentMessage][2] +
    111 title='View Page'>View Page</a>";
// 0 Изменить состояние ленты
  document.getElementById("spanCount").innerHTML =
    "Message " + (currentMessage+1) +
    " of " + arrayMessage.length;
  var objDivl — document.getElementById("divNewsl");
var objDiv2 = document.getElementByldf"divNews2");
// © Подготовить переход
if(layerFront == 1){
    objDiv2.className = "opacO";
    objDivl.style.zlndex = 1;
    objDiv2.style.zlndex = 2;
    objDiv2.innerHTML = strDisplay;
    layerFront = 2;
  }
  else{
    objDivl.className = "opacO";
    objDiv2.style.zlndex = 1;
    objDivl.style.zlndex = 2;
    objDivl.innerHTML = strDisplay;
    layerFront = 1;
  }
// © Начать переход
  SetClass(O);
```

Функция ChangeView() O служит двум основным целям. Во-первых, она создает HTML-документ для отображения данных, полученных из RSS-лент; во-вторых, подготавливает элементы div к проявлению. Создание HTML-доку мента выполняется просто, поскольку мы используем стандартную структуру. Сложнее всего в этом деле гарантировать, что мы правильно отследили все кавычки и апострофы и не допустили ошибок.

Первой строкой текста, которую мы собираемся отобразить, является название канала RSS ®, которое хранится в первом элементе массива arrayMessage. Это название необходимо поместить в элемент span и соотнести с этим элементом имя класса CSS RSSFeed. Далее требуется отобразить заголовок статьи ©, обратившись ко второму элементу массива. Помещая заголовок в элемент span и сопоставляя с этим элементом имя класса CSS itemTitle, мы допускаем для заголовков определение различных стилей. Чтобы разделить заголовок и тело сообщения, между ними проводится горизонтальная черта.

Описание статьи O было записано в четвертом элементе массива arrayMessage. Мы отделяем это описание от следующего раздела, в котором

находится последний собранный элемент статьи — ссылка ©; атрибуту HREF данной ссылки присваивается значение элемента URL. Вследствие этого пользователь видит текст "View Page" ("Посмотреть страницу"), на котором он может щелкнуть мышью. Ссылка, привязанная к этому тексту, направит пользователя на Web-сайт RSS-ленты.

Далее мы хотим обновить отображаемый счетчик текущего сообщения, встроенный в наше RSS-приложение. Для этого мы изменяем свойство innerHTML 0 элемента spanCount, используя длину arrayMessage и счетчик текущего сообщения. Затем требуется подготовить элементы div ® к демонстрации переходов. Для инициализации элемента div значение zlndex задается так, чтобы данный элемент располагался поверх текущего, а класс соответствовал первому правилу CSS, определяющему степень прозрачности.

Загрузив текущее сообщение в элемент div, мы начинаем процесс проявления этого элемента. Для этого необходимо создать функцию, последовательно загружающую классы CSS; следовательно, мы вызываем функцию SetClassO ©.

13.4.3, Интеграция таймеров JavaScript

Процесс загрузки элемента div в поле зрения создает не резкую смену сообщений, а плавный переход между ними. Данный эффект достигается посредством изменения степени прозрачности слоя с помощью созданных ранее классов CSS. Благодаря этому мы можем наблюдать слой, располагающийся ниже данного (мы как бы смотрим сквозь тонированное стекло). Чтобы убрать из поля зрения все содержимое нижнего слоя, мы уменьшаем степень прозрачности верхнего.

Как отмечалось в разделе 13.4.1, для обработки затухания/проявления слоев мы применяем пять классов CSS. Использование классов позволит нам впоследствии добавить к переходу изменение цвета или любую другую модификацию стиля. В данном случае, как показано в листинге 13.13, мы просто циклически проходим по классам.

Листинг 13.13. Задание класса CSS и эффекта перехода

```
// О Объявить SetClassO
function  SetClass(xClass){
// © Проверить шаг перехода
  if(xClass<5){
// © Установить следующее значение className
    document.getEleraentById("divNews" +
// О Инициализировать таймер перехода
layerFront).className = "opac" + xClass;
    timerAmt = setTimeout("SetClass{" +
    (xClass+1) + ")",timeColor);
  }
  else{
// © Убрать класс CSS
    document.getElementByld("divNews"
    + layerFront).className = "";
// О Увеличить счетчик
```

```
      currentMessage++;
    // © Проверить следующее сообщение
      if(currentMessage>=arrayMessage.length)
        currentMessage — 0;
    // © Запустить таймер
      if(IbPaused)
        timerSwitch = setTimeout(
          "ChangeView()",flipLength);
    }
```

В листинге 13.13 показана функция Setciass() О, которой передается параметр xClass. Этот параметр позволяет отслеживать текущее состояние перехода, не используя никаких других глобальных переменных. Указанная функция вызывается на каждом шаге перехода и обновляет состояние, пока переход не будет завершен.

Поскольку мы работаем с пятью классами CSS, необходимо убедиться, что текущий шаг процесса перехода © не является пятым. Если это так, значит, для завершения перехода требуется применить еще один или несколько классов CSS, и мы применяем к элементу следующий класс CSS, обращаясь к атрибуту className ©.

Установив новый класс, необходимо создать таймер для вызова следующего шага. Используемый для этого метод setTimeout О имеет два параметра. Первый — это функция или выражение JavaScript, которое требуется выполнить, второй — время в миллисекундах до выполнения функции/выражения. В данном случае мы планируем вызвать функцию SetClass (), в качестве аргумента которой передается параметр состояния, значение которого увеличено на 1. Время ожидания задается согласно значению глобальной переменной flipLength, объявленной в разделе 13.3.1.

В ветке else сценария обрабатывается ситуация, когда мы уже прошли по всем пяти классам CSS, последовательно применив их к элементу div. Здесь мы вначале убираем из элемента div класс CSS ©. Параметр непрозрачности элемента устанавливается равным 100% (значение по умолчанию); таким образом, мы разрешаем элементу div полностью закрывать другой блок, расположенный на нижележащем слое.

Далее мы увеличиваем на 1 значение переменной currentMessage ©, позволив загружать следующее сообщение. Мы проверяем, чтобы номер этого сообщения © был больше числа сообщений, содержащихся в массиве arrayMessage. Если это так, мы запускаем с начала текущее сообщение. Таймер перезапускается, чтобы загрузить следующее сообщение по прошествии заданного промежутка времени ©. За вызов функции ChangeView() отвечает метод setTimeout, а время ожидания определяется глобальной переменной flipLength. Чтобы все сказанное было возможным, мы должны проверить, что значение глобальной переменной bPaused не равно true. Подробнее кодирование паузы будет рассмотрено в разделе 13.5.2.

Теперь переход между сообщениями готов, и мы можем проверить созданный сценарий. Если все работает нормально, мы должны увидеть счетчик

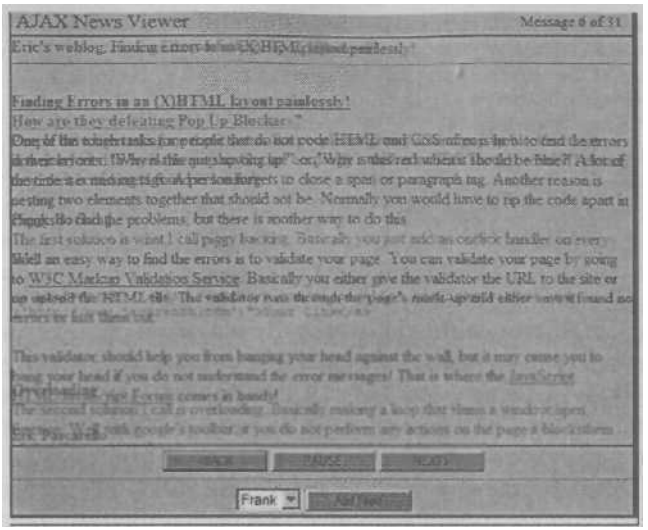


Рис. 13.9. Затухающий переход между сообщениями 5 и 6

загруженных страниц, плавно увеличивающийся по мере загрузки файлов в сценарий, а первое загруженное сообщение должно начать проявляться.

Как видно на рис. 13.9, сценарий содержит два сообщения, поскольку одно из них немного прозрачно. Текущее сообщение (6) отображается в заголовке, и мы видим, что всего загружено 31 сообщение. Все, что нам осталось сделать, — это добавить кнопки паузы, перехода назад и вперед, а также связать с ними соответствующие функциональные возможности.

13.5. Дополнительные возможности

Написанный нами код уже можно использовать, все дополнительные функции не являются обязательными, хотя они и могут сделать сценарий более гибким для пользователей (да и для нас тоже). Итак, прежде всего мы добавим функцию, которая позволит импортировать дополнительные RSS-ленты. не включенные в функцию предварительной загрузки. Возможно, мы хотим время от времени проверять какой-то сайт на наличие нового содержимого или добавить в приложение RSS-ленту с прогнозом погоды. Указанная возможность позволит получать ленту тогда, когда она нам потребуется. Кроме того, мы добавим возможность пропускать сообщения и приостанавливать их, если для прочтения требуется больше времени.

13.5.1. Введение дополнительных лент

Добавить дополнительные сообщения в ленту легче, чем вы могли подумать. Вернемся к рис. 13.9. Список содержит имена и URL дополнительных лент, которые нам требуется проверять эпизодически; потом мы просто выбираем

имя и щелкаем на кнопке Add Feed ("Добавить ленту"). Большинство необходимых функциональных возможностей было реализовано в разделе 13.3; все, что от нас требуется сейчас (листинг 13.14), — запустить объект ContentLoader, который и добавит ленту, выбранную в элементе select.

Листинг 13.14. Функция JavaScript, загружающая дополнительные ленты RSS

```
function LoadNews(){
  var sel = document.Forml.ddlRSS;
  if(sel,options[sel.selectedlndex].className!="added"){
    var url = sel.options[sel.selectedlndex].value;

    sel.options[sel.selectedlndex].className="added";
    var loaderl = new net.ContentLoader(url,
                          BuildXMLResults);
  }
```

В приведенном листинге показана функция LoadNews (), которая запускается щелчком на кнопке btnAdd. Поскольку указатели URL дополнительных лент RSS мы получаем из элемента select, нам необходимо обращаться к значениям элемента ddlRSS.

Если мы собираемся добавлять RSS-ленты из элемента select, необходимо учесть возможность того, что эта лента уже добавлена. Одним из возможных решений здесь является добавление к опции класса CSS. Таким образом, в приложение необходимо включить проверку, гарантирующую, что лента RSS не была добавлена ранее. Если лента действительно является новой, мы берем значение выбранной опции и изменяем значение className на added.

При выполнении объекта ContentLoader мы передаем ему в качестве аргументов URL ленты и функцию BuildXMLResults (). Если в процессе произойдет какой-либо сбой, мы можем использовать принятое по умолчанию сообщение об ошибке объекта ContentLoader. Таким образом, сейчас мы можем загружать документ из списка, но, кроме того, нам требуется предусмотреть введение в этот список новых RSS-лент и связать с кнопкой обработчик событий, приведенный в листинге 13.15.

```
<select name="ddlRSS">
  <option
    value="http://radio.javaranch.com/frank/rss.xml">
    Frank</option>
  <option
    value="http://radio.javaranch.com/gthought/rss.xml">
    Gregg</option>
</select>
<mput type="button" name="btnAdd"
  value="Add Feed" onclick="LoadNews()" />
```

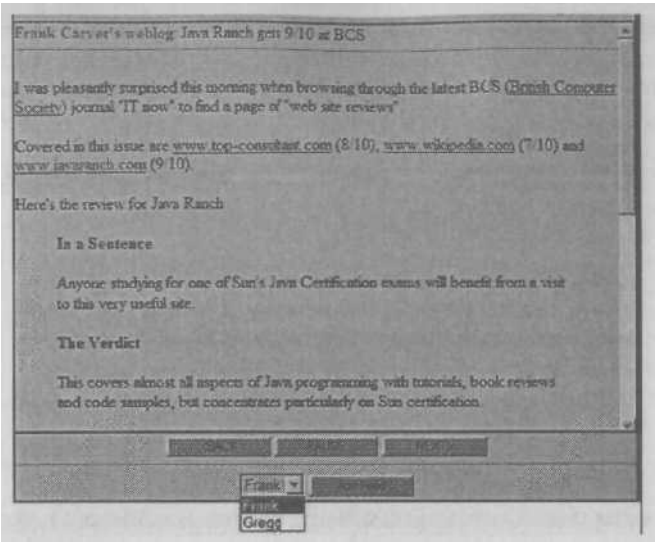


Рис. 13.10. В приложение чтения RSS-лент добавлена новая лента

С помощью данного списка мы добавляем указатели URL на ленты RSS, не содержащиеся в предварительно загруженном массиве RSS-лент. В данном случае мы добавили две RSS-ленты из радиоблога JavaRanch. К кнопке btnAdd был добавлен обработчик событий onclick, позволяющий выполнить функцию LoadNews (). Последним этапом загрузки дополнительных лент является создание класса CSS, добавляемого в таблицу стилей. Благодаря этому пользователь может увидеть, что лента загружена.

```
.added{background-color: silver;}
```

В приведенный класс CSS мы можем добавить любое правило CSS, позволяющее отличать добавленные ленты от всех остальных. В данном случае мы изменили цвет фона опции на серебристый, так что теперь эта опция визуально выделяется из списка. Добавив класс, мы снова можем протестировать полученное приложение.

Как показано на рис. 13.10, мы добавили RSS-ленту с пометкой Frank (она обозначена серебристым цветом). Лента, помеченная как Gregg, еще не добавлена, поскольку ее цвет является принятым по умолчанию белым цветом фона. Кроме того, после добавления ленты число сообщений в приложении увеличилось с 31 до 54. Уже почти все готово. Осталось добавить кнопки "Вперед", "Назад" и "Пауза".

13.5.2. Интеграция функций пропуска и паузы

Одной из наиболее полезных дополнительных возможностей является пропуск сообщений. Если нам попадается сообщение, которое нас не интересует, мы щелкаем на кнопке пропуска и смотрим следующее, не ожидая, пока до

него дойдет очередь. Функция паузы позволяет приостановить работу приложения, чтобы мы могли не торопясь прочесть заинтересовавшее нас сообщение. Поскольку для представления таймеров, паузы и счетчика currentMessage используются глобальные переменные, мы очень просто можем влиять на текущее состояние приложения. В листинге 13 16 показан код, позволяющий пользователю быстро проходить по лентам.

Листинг 13.16. Функция JavaScript, отвечающая за остановку и пропуск лент

```
//O Создать функцию MoveFeed()
function MoveFeed {xOption){
// @ Определить, требуется ли приостановка или возобновление работы
  if(xOption == 0){
// © Приостановить работу
    if(IbPaused){
      bPaused • true;
      if(timerSwitch)
        clearTimeout(timerSwitch);
      document.getElementById{"btnPause").value = "RESUME";
    }
// // O Возобновить работу
    else(
      bPaused = false;
      setTimeout("ChangeView()",300);
      document.getElementById("btnPause").value • "PAUSE";
    }
  }
// 0 Изменить текущее сообщение
  else{
  if(timerSwitch)
    clearTimeout(timerSwitch);
  if(xOption "™ -1)currentMessage += -2;
  if(currentMessage < 0)
  currentMessage = arrayMessage.length
    + currentMessage;
  ChangeView();
  }
```

Создав функцию MoveFeed () O и позволив ей принимать в качестве аргумента единственный параметр, мы сможем обрабатывать три ситуации: паузу, перемотку вперед и перемотку назад. Чтобы различать действия мы можем использовать целочисленное значение. Чтобы приостановить работу приложения, мы будем передавать функции MoveFeed () значение 0, чтобы перейти на следующее сообщение — значение 1, а чтобы перейти не предыдущее сообщение — значение —1.

В приведенном коде мы вначале проверяем условие паузы — равенство нулю аргумента функции © Кнопка паузы отвечает за два действия — приостановку работы приложения (прекращение выполнения переходов) и возобновление работы (продолжение слайд-шоу с переходами между сообщениями).

Если в текущий момент лента не приостановлена ©, тогда значение переменной bPaused необходимо установить равным `true` и проверить, запущен ли

таймер timerSwitch. Если да, то мы должны его сбросить, используя метод clearTimeout. Далее текст кнопки изменяется на "RESUME". Если на кнопке щелкают для того, чтобы возобновить работу, мы должны выполнить противоположные действия О. Значение переменной bPaused устанавливается равным false; мы вызываем функцию ChangeView() с небольшой задержкой во времени и изменяем текст кнопки на "PAUSE".

Все Работу с паузой мы завершили.

Теперь необходимо создать функции пропуска сообщений и возврата к предыдущим сообщениям ©. Поскольку при этом мы будем изменять отображаемые статьи, необходимо убрать таймер, чтобы избежать проблем с пропуском нескольких сообщений. После этого требуется проверить, не равен ли параметр действия — 1. Если мц возвращаемся к предыдущим сообщениям, из значения переменной currentMessage необходимо вычесть 2. Это происходит из-за того, что переменная currentMessage в действительности содержит номер следующего сообщения, поскольку ее значение уже было увеличено на 1 Вычитая 1 из значения переменной, мы получаем то же самое сообщение. Следовательно, поскольку в переменной currentMessage уже хранится номер следующего сообщения, при реализации кнопки перемотки вперед нам ничего не *нужно* делать.

Затем нам необходимо убедиться, что значение переменной не меньше нуля. Если оно меньше нуля, значение переменной нужно установить согласно последнему сообщению в массиве. Изменив значение currentMessage, мы можем вызвать функцию ChangeView{) для загрузки сообщения. Все, что от нас требуется, — *добавить* обработчики событий к кнопкам (листинг 13.17), чтобы мы могли выполнить функцию MoveFeed ( ).

Листинг 13.17. Обработчики событий onclick для действий с кнопками

```
value=" <BACK " onclick="MoveFeed{-1)">
value=" PAUSE " onclick="MoveFeed(0)">
value="NEXT>" onclick="MoveFeed(1)">                                    —
```

Для инициализации функции мы добавляем к кнопкам обработчики событий onclick. Эти обработчики вызывают функцию MoveFeed(), которая принимает в качестве аргументов такие значения: -1 — вернуться к предыдущему сообщению; 0 — приостановить работу, 1 — перейти на следующее сообщение. Чтобы проверить работу функций, мы сохраняем документ и открываем данную страницу в браузере.

Теперь, когда мы можем пропускать сообщения, можно легко перейти к сообщениям в середине списка RSS-ленты. На рис. 13.11 показано, как выглядит приложение при нажатой паузе: на кнопке btnAdd отображается надпись RESUME ("Возобновить"). Таким образом, используя описанные в разделе дополнительные функции, мы получаем приложение, позволяющее читать RSS-ленты на рабочем столе, не посещая Web-сайты, поставляющие эти ленты.

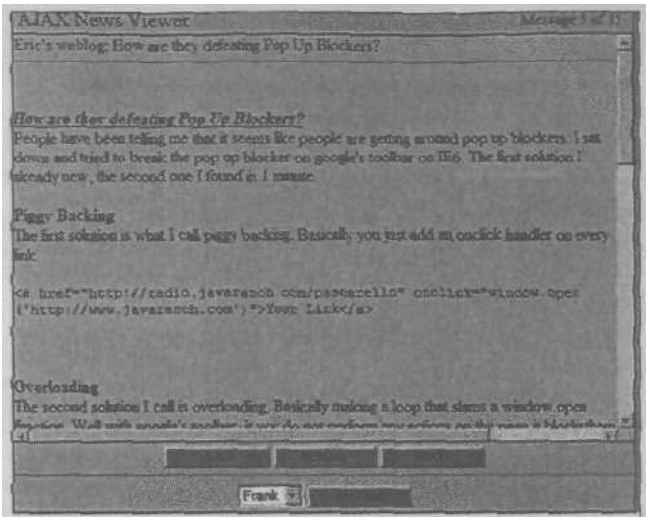


Рис. 13.11. Работа показанного в окне приложения чтения RSS-лент приостановлена, поскольку центральная кнопка имеет название RESUME

13.6. Как избежать ограничений проекта

С помощью основанного на Ajax приложения мы можем читать RSS-ленты из HTML-файла, записанного на рабочем столе, не требуя серверного кода. Данное приложение можно использовать для захвата RSS-лент без посещения соответствующих Web-сайтов. Мы можем предложить посетителям нашего Web-сайта загрузить данную HTML-страницу, настроив ее на чтение RSS-лент этого сайта. Поскольку представленный сценарий можно запустить и на самом Web-сайте, его можно использовать и для других целей. Например, это может быть переключение рекламы в баннере, баннер новостей компании или все, что мы можем придумать. Однако представленный сценарий обладает определенными ограничениями; кроме того, при его запуске на рабочем столе с помощью браузера Mozilla могут возникнуть определенные проблемы.

13.6.1. Обход системы безопасности браузеров Mozilla

В отличие от Microsoft Internet Explorer, браузеры Firefox и Mozilla не позволяют запускать созданное приложение с рабочего стола из-за встроенных ограничений безопасности. Данные ограничения не дают Ajax общаться с другими Web-сайтами, поскольку направлены на то, чтобы код не мог отправлять информацию без нашего ведома.

Чтобы убедиться, что проблема работы сценария связана именно с этим, необходимо исследовать сообщение об ошибке. В браузере Mozilla мы должны открыть консоль JavaScript, вызываемую из меню Tools =?• Web Development =Ф JavaScript Console (Инструменты => Web-разработка =Ф Консоль JavaScript) (рис. 13.12).

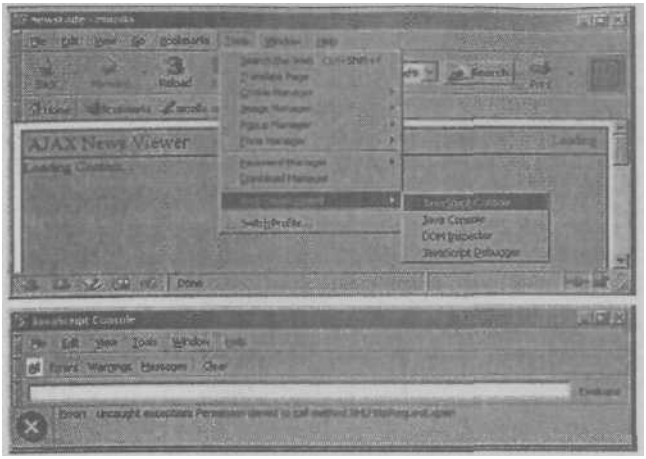


Рис. 13.12. В браузере Mozilla выберите Tools => Web Development => JavaScript Console (Инструменты => Web-разработка => Консоль JavaScript)

Рис. 13.13. Сообщение об ошибке "недостаточно полномочий", порожденной объектом XMLHttpRequest

Выбрав команду JavaScript Console (консоль JavaScript), мы открываем дополнительное окно (рис. 13-13).

На рис. 13.13 показана ошибка "permission denied" ("недостаточно полномочий"), порожденная объектом XMLHttpRequest. Исправить ее можно двумя способами. Например, можно открыть конфигурационный файл браузера Mozilla и изменить настройки разрешений, позволив объекту XMLHttpRequest выполнить то, что он хочет. Для этого в адресной строке браузера следует ввести about: conf ig и изменить необходимые настройки, однако данные действия небезопасны.

Небезопасность изменения настроек связана с тем, что наши разрешения касаются всего, что происходит на компьютере. Это означает, что любой сценарий, желающий связаться с внешним миром, сможет это сделать. Как этого можно избежать, разрешив данное общение только нашему сценарию? В качестве решения мы можем установить безопасность с помощью JavaScript. О том, как это сделать, при условии, что браузер настроен на обработку запросов к PrivilegeManager, мы рассказывали в главе 7. Напомним кратко, о чем идет речь. В листинге 13.18 показан общий код, который предоставляет дополнительные привилегии, требуемые для чтения внешних ресурсов.

Листинг 13.18. Управление привилегиями Security Privilege Manager

```
if(window.netscape &&
   window.netscape.security.PrivilegeManager.enablePrivilege)
   netscape.security.PrivilegeManager.enablePrivilege(
      'UniversalBrowserRead');
```

В приведенном коде проверяется, можем ли мы обращаться к PrivilegeManager. Если да, тогда мы получаем привилегию UniversalBrowserRead. Данный код необходимо добавить в два различных места объекта ContentLoader, отвечающего за функциональные возможности Ajax.

Вначале мы должны вставить указанный код сразу после объявления loadXMLDoc, как показано в листинге 13.19.

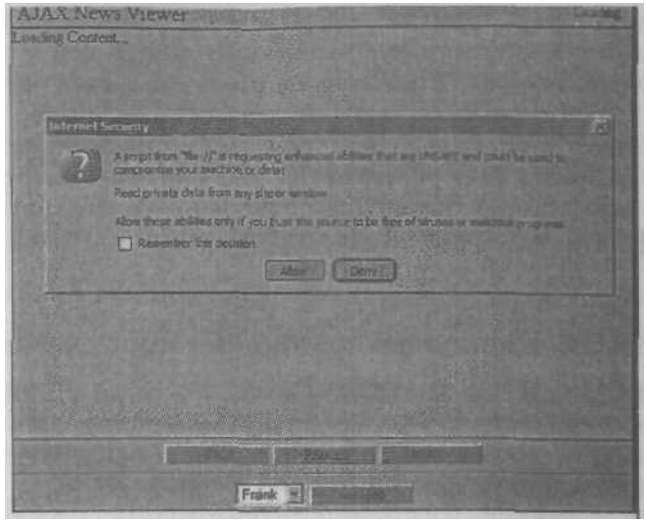


Рис. 13.14. Запрос, требующий от пользователя подтвердить изменение прав доступа

Листинг 13.19. Вставка "кода *m* функцию loadXMLDoc

```
net.ContentLoader.prototype.loadXMLDoc = function(
url,method,params,contentType){
  if(window.netscape &&
  window.netscape.security.PrivilegeManager.enablePrivilege)
  netscape.security.PrivilegeManager.enablePrivilege(
    'UniversalBrowserRead');
```

Кроме того, код необходимо вставить в функцию onReadyState (листинг 13.20).

Либтинг 13.20, Вставка кода в функцию onReadyState И М Я

```
net.ContentLoader.onReadyState=function(){

  if(window.netscape &&
    window.netscape.security
    .PrivilegeManager.enablePrivilege)
  netscape.security.PrivilegeManager
    .enablePrivilege('
UniversalBrowserRead');
```

Обе указанные функции взаимодействуют с данными из внешнего мира. Именно поэтому мы должны ввести обозначенные фрагменты в обоих местах. При выполнении сценария мы получаем сообщение-запрос, требующее подтверждения изменений настроек безопасности (рис. 13.14).

Если просто щелкнуть на кнопке Allow (Разрешить), запрос будет открываться при каждом вызове функции. Чтобы избежать этого, необходимо

установить флажок Remember this decision ("Запомнить мой выбор"). Таким образом, браузер запоминает ваш выбор и позволяет выполнять объект XMLHttpRequest без запроса подтверждения.

Изменив настройки безопасности браузера, мы можем запустить приложение на рабочем столе, используя браузеры Mozilla, Firefox и Netscape. Используя данное приложение, мы можем обращаться к XM1^лентам с любого сайта, не открывая несколько закладок или окон. Кроме того, приложение можно изменить, чтобы оно получало из Web другую информацию, например, прогноз погоды и комиксы.

13.6.2. Изменение масштаба приложения

Наше приложение может не только читать XML-ленты, извлекаемые с различных сайтов. Мы легко можем использовать его как переключатель рекламных объявлений в баннере, сценарий обновления новостей компании, календарь событий и многое другое.

Например, мы можем хранить баннерную рекламу в документе XML. Таким образом в XML-файл легко ввести новую рекламу, не затрагивая HTML-файлы или серверный код. Мы можем предварительно загрузить баннерную рекламу, а затем отобразить ее в нашем приложении. Вместо того чтобы на экране находился один рекламный ролик, мы можем циклически менять их, пока пользователь изучает сайт.

Мы можем поместить в XML-документ новости компании, чтобы отображать текущие статьи сотрудникам или клиентам. Все, что от нас требуется, — это заполнить стандартные элементы XML-ленты. Кроме того, можно отображать информацию об обновлении сайта или любые другие данные. Как видите, возможности приложения не ограничиваются простым чтением XML-лент.

13.7. Реструктуризация

Итак, мы получили готовый сценарий для чтения RSS-лент и можем, как обычно, заняться его улучшением. Как отмечалось ранее, существуют богатые возможности расширения сценария с точки зрения представления различных типов содержимого. В данном разделе мы сфокусируем внимание на реорганизации сценария согласно архитектуре "модель-представление-контроллер" (Model-View-Controller — MVC). Как объяснялось в главах 3 и 4, MVC широко распространена и позволяет разделять обязанности между различными фрагментами кода. Свое обсуждение мы начнем с определения типов моделей, затем создадим представление модели и разработаем контроллер, связывающий в единое целое все компоненты.

13.7.1. Модель приложения

Разработанное выше приложение чтения RSS-лент очевидно только выиграет от наличия нескольких типов модели. Благодаря этому программное обеспечение будет концептуально понятнее, его станет легче использовать и модифицировать. Поскольку приложения Ajax возлагают более интенсивную нагрузку на DHTML-код клиента, чем традиционные Web-приложения, важность понятного управляемого кода становится первостепенной. Классы модели, которые мы собираемся разработать, должны быть в общем случае применимы и к другим приложениями, работающим с RSS-лентами. Чтобы немного упростить синтаксис кода, для определения типов мы, как и в главе 10, будем использовать библиотеку Prototype.

Начнем с определения класса модели для RSS-ленты. В данном случае RSS-лента представляет собой XML-документ, обладающий предопределенной структурой и имеющий URL, задающий путь доступа к этому документу. Основными атрибутами структуры являются название, ссылка и описание, кроме того, как обсуждалось выше, имеется множество дополнительных атрибутов. Кроме того, лента содержит несколько элементов item, которые можно рассматривать как набор статей. Для начала давайте соберем все, что мы знаем о ленте, в такой форме, как показано в листинге 13.21.

Листинг 13.21. Класс модели RSSFeed • . ^ДВИЯДЦ^И^^ИЭД^^^^Р

```
RSSFeed = Class.create();
RSSFeed.prototype = {
  initialize: function( title, link, description ) {
    this.title - title;
    this.link = link;
    this.description = description;
    this.items = [];
  Ь
  addltem: function(anltem) {
    this.iterns.push(anltem); }
```

В приведенном коде тип RSSFeed определяется посредством функции Class.create() библиотеки Prototype. Напомним, что, используя эту идиому, мы с помощью сгенерированного конструктора вызываем метод itnitialize. Таким образом, при данном определении класса модели RSS-ленты ленту можно создать с помощью следующей строки кода:

```
var rssFeed = new RSSFeed{  'JavaRanch News',
                            'http://radio.javaranch.com/news/',
                            'Stories from around the ranch'  );
```

По сути, это все, что требуется для определения объекта модели RSSFeed. Обратите внимание на то, что RSSFeed имеет API addltem, разрешающий добавление статей к исходному массиву элементов. Каждая статья должна представлять собой объект модели, который инкапсулирует атрибуты каждого элемента ленты. Вспомнив все, что мы знаем о статьях RSS, можно определить класс модели статьи, как показано в листинге 13.22.

Листинг 13.22. Класс модели RSSItem

```
RSSItem = Class.create();
RSSltem.prototype = {
initialize: function( rssFeed, title, link, description ) {
  this.rssFeed = rssFeed,-
  this.title = title;
  this.link = link;
  this.description = description;
 ⌋
};
```

Ничего особенного. В элемент статьи инкапсулированы атрибуты `title`, `link` и `description`, кроме того, имеется ссылка на объект RSSFeed, к которому относится статья. Изучая два описанных класса, видим, что статью и одну из лент можно сформировать следующим образом:

```
var rssFeed = new RSSFeed( 'JavaRanch News',
  'http://radio.javaranch.com/news/',
  'Stories from around the ranch' );
var feedl = new RSSItem( rssFeed,
  'Win a copy of JBoss',
  'http://radio.javaranch.eom/news/05/07/20/9.html',
  'Text of Article' );
rssFeed.addltern(feedl);
```

Пока что все хорошо. Модель является довольно прямолинейной инкапсуляцией атрибутов RSS-ленты и ее статей. Данные концепции инкапсулируются в двух классах модели —- RSSFeed и RSSltem. Далее мы рассмотрим собственно построение модели. Мы знаем, что эти объекты будут обработаны в результате загрузки XML-данных клиенту в ответ на запрос Ajax. Поэтому мы определим программный интерфейс, вызывая который наш обработчик Ajax сможет преобразовать XML-отклик в экземпляр класса модели RSSFeed. Для этого мы определим вначале контракт процесса, формирующего модель.

```
var rssFeed = RSSFeed.parseXML{ rssXML );
```

Данный контракт подразумевает, что мы передаем XML-отклик, возвращаемый обработчиком Ajax, методу синтаксического разбора нашего приложения RSSFeed; в результате мы должны получить экземпляр RSSFeed. Будем придерживаться сказанного и реализуем метод parseXML(), как показано в листинге 13.23.

```
RSSFeed.parseXML = function(xmlDoc) {
 var rssFeed = new RSSFeed(
   RSSFeed.getFirstValue(xmlDoc, 'title'),
   RSSFeed.getFirstValue(xmlDoc, 'link' ),
   RSSFeed.getFirstValue(xmlDoc, 'description'));
 var feedltems = xmlDoc.getElementsByTagName('item');
 for ( var i = 0 ; i < feedltems.length ; i++ ) {
   rssFeed.addltern(new RSSltem(rssFeed,
     RSSFeed.getFirstValue(feedltems[i], 'title'),
     RSSFeed.getFirstValue(feedltems[i], 'link' ),
     RSSFeed.getFirstValue(feedltems[i], 'description'))
```

```
  }
  return rssFeed;
}
```

Приведенный метод выполняет классический разбор XML-отклика, с которым вы уже неоднократно сталкивались. Метод принимает значения элементов title, link и description, используя их для создания RSSFeed. Затем он последовательно проходит по всем элементам item, выполняя эти же действия и создавая для каждого элемента экземпляр RSSItem. На каждой итерации используется метод addItem(), добавляющий статью к родительской RSS-ленте. Обратите внимание на использование вспомогательного метода для получения значения узла первого дочернего элемента с данным именем дескриптора. Данный вспомогательный метод getFirstValue показан в листинге 13.24.

Листинг 13.24. Вспомогательный метод синтаксического разбора

```
RSSFeed.getFirstValue = function(element, tagName) {
  var children = element.getElementsByTagName(tagName);
  if ( children == null || children.length == 0 )
    return "";
  if ( children[0].firstChild &&
       children[0].firstChild.nodeValue )
    return children[0].firstChild.nodeValue;
  return "";
}
```

Это все, что нам требуется с точки зрения модели. Очевидно, мы можем добавить атрибуты, представляющие все необязательные элементы RSS-ленты, и присвоить им значения, если эти элементы присутствуют в ленте. В данном случае нам это делать не нужно, поскольку приложение чтения RSS-лент не использует дополнительные атрибуты (и не требует их предоставления). Тем не менее возможность предоставления дополнительных метаданных существует. Кроме того, мы можем определить методы, предлагающие более формальный контракт доступа к атрибутам. Например, мы можем написать пару getTitle()/setTitle(), предназначенную для доступа к атрибуту заголовка. Поскольку JavaScript, в отличие от других объектно-ориентированных языков, не поддерживает визуальную семантику (например, ключевые слова private/protected в Java), мы этим вопросом заниматься не будем. Итак, с моделью мы закончили, переходим к представлению.

13.7.2. Представление приложения

Разобравшись с классами модели, мы можем рассматривать класс представления. Мы можем разработать один класс представления для RSSFeed, а другой — для RSSItem, но поскольку приложение RSSReader в действительности не представляет ленту независимо, мы определим один класс представления RSSItemView, который инкапсулирует представление RSSItem в контексте родительского элемента RSSFeed. Поскольку в данном случае представ-

ление — это HTML-код, наш класс представления отвечает за генерацию HTML-страницы. Итак, рассмотрим для начала конструктор, представленный в листинге 13.25-

Листинг 13.25. Класс представления RSSItemView

```
RSSItemView = Class.create();
RSSItemView.prototype = {
  initialize: function(rssItem, feedIndex, itemIndex, numFeeds) {
    this.rssItem = rssItem;
    this.feedIndex • feedIndex + 1;
    this.itemIndex = itemIndex + 1;
    this.numFeeds = numFeeds;
  b
}
```

Давайте внимательно рассмотрим параметры. Первый параметр — экземпляр RSSItem. С его помощью мы сообщаем представлению, для какого экземпляра модели обеспечивается это представление. Обратите внимание на то, что в общем случае классам модели не разумно знать что-либо о представлении, однако представление по необходимости имеет детальные знания о модели. Другие параметры обеспечивают некоторый дополнительный контекст для представления. Параметр feedIndex сообщает представлению номер ленты. Параметр itemIndex указывает представлению, где размещается данная статья в родительском массиве статей RSSFeed. Параметр numFeeds сообщает представлению, сколько всего лент. Все указанные параметры необходимы для получения информации о месте данного представления в мире. Представление может пожелать отобразить область содержимого, указывающую, например, "это лента 1 из 7 и статья 3 из 5". Данные атрибуты можно внедрить в модель, однако в действительности они не являются атрибутами, за которые в общем случае должна отвечать модель, поэтому этот контекст (требуемые представлению) передается клиентом в конструктор представления.

Как отмечалось выше, представление отвечает за генерацию HTML-кода. Поэтому нашему классу представления понадобится еще один метод. Посмотрим, как это он выглядеть (листинг 13.26).

Листинг 13.26. Метод генерации HTML-кода

```
toHTML: function () {
  var out = ""
  out += '<span class="rssFeedTitlePrompt">RSS Feed '
  out += '(' + this.feedIndex + • of ' + this.numFeeds +  ') : ';
  out += '</span>';
  out += '<span class="rssFeedTitle">';
  out += '<a href="' + this.rssItem.rssFeed.link + '">' +
  this.rssItem.rssFeed.title + '</a>';
  out += '</span>';
  out += '<br/>';
  out += '<span class="rssFeedItemTitlePrompt">Article ';
  out += '(' + this.itemIndex + ' of * +
  this.rssItem.rssFeed.items.length + ') : ' ;
  out += '</span>';
```

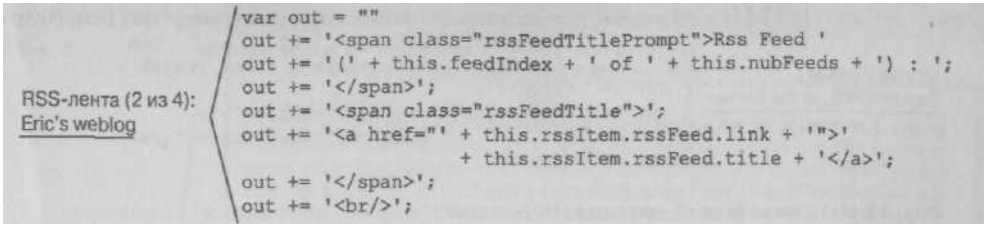


Рис. 13.15. RSS-лента (x из y): название RSS-ленты

```
out += '<span class="rssFeedItemTitle">';
out += '<a href="' + this.rssItem.link + '">' +
this.rssItem.title + '</a>';
out += '</span>';
out += '<div class="rssItemContent">';
out += this.rssItem.description;
out += '</div>';
return out;
}
```

Метод toHTML создает на экране контекстные элементы, за которыми следует текст статьи. Первый фрагмент кода отображает текст RSS-лента (x из y) : Заголовок RSS-ленты. Атрибут link родительского элемента rssFeed используется для генерации атрибута HREF анкера, а атрибут title — для генерации текста анкера. Для каждого элемента span генерируется имя класса CSS (одно для запроса, второе для анкера), что позволяет независимо определять стили этих элементов. Сказанное иллюстрируется на рис. 13.15.

Следующий фрагмент кода генерирует текст Статья (x из y) : Заголовок RSS-статьи. Для генерации атрибута HREF анкера используется атрибут link RSS-статьи, а с помощью атрибута title статьи генерируется текст анкера. Кроме того, как показано на рис. 13.16, код предоставляет имена классов CSS для запроса и заголовка. Несколько последних строк метода toHTML генерируют элемент div, который будет вмещать содержимое статьи RSSItem (атрибут description). Соответствующий код выглядит следующим образом:

```
out += '<div class="rssItemContent">';
out += this.rssItem.description;
out += '</div>';
```

Для текста статьи генерируется имя класса CSS rssItemContent. Чтобы текст не подходил вплотную к границе блока, в соответствующем классе необходимо задать небольшие поля и заполнение. Кроме того, данный блок должен иметь фиксированную высоту и значение auto параметра overflow, чтобы его содержимое можно было при необходимости прокручивать независимо от показанной ранее контекстной информации. Типичное определение класса CSS для блока текста статьи приводится ниже.

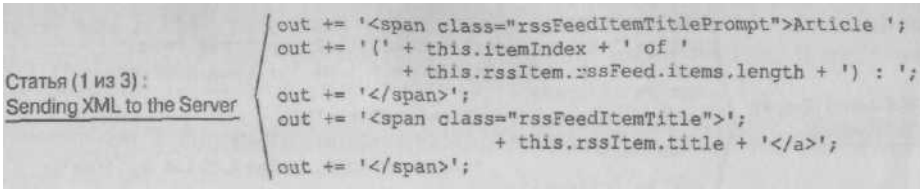


Рис. 13.16. Статья (x из y) заголовок RSS-статьи

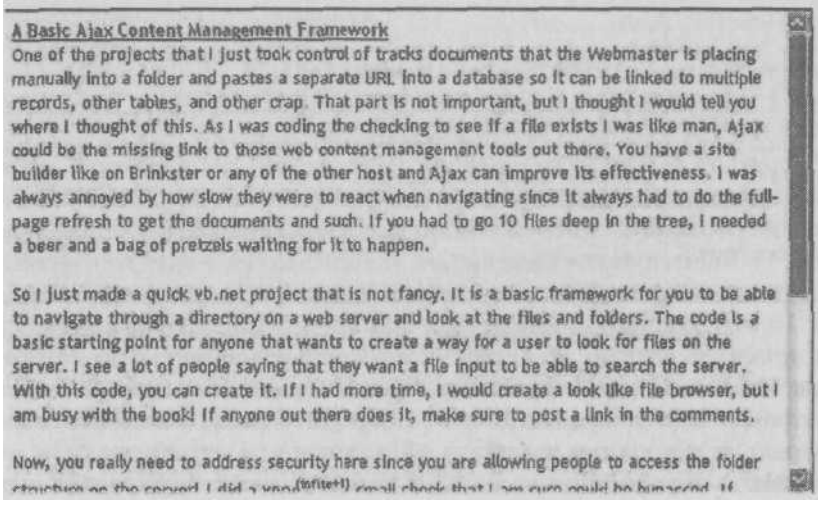


Рис. 13.17. Статья (x из y) заголовок RSS-статьи

```
.rssItemContent {
  border-top : 1px solid black;
  width : 590px;
  height : 350px;
  overflow : auto;
  padding : 5px;
  margin-top : 2px;
}
```

При таком стилевом оформлении область содержимого должна выглядеть примерно так, как показано на рис. 13.17. Собирая вместе все составляющие, получаем показанное на рис. 13.18 представление, генерируемое классом RSSitemView. Прежде чем завершить рассмотрение представления, добавим в него еще один маленький метод, который немного облегчит его использование.

```
toString: function() {
  return this.toHTMLO;
J
```

RSSItemView Result

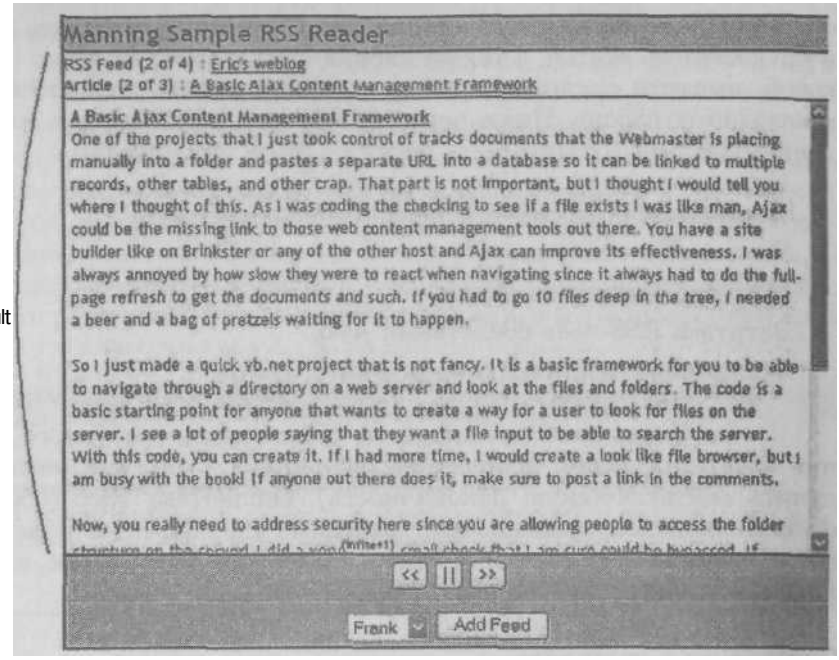


Рис. 13.18. Окончательный вид класса RSSItemView

Причина, по которой мы дали представлению метод tostring, заключается в том, что это позволяет нам взаимозаменяемо использовать экземпляр представления и генерируемую им строку HTML. Например, мы можем присвоить значение атрибуту innerHTML элемента, равное классу представления, и далее будет использоваться строковое представление (сгенерированный HTML-файл). Например, ниже приводится код, присваивающий сгенерированный HTML-документ представления атрибуту innerHTML элемента div с идентификатором contentDiv.

```
var rssItemView - new RSSItemView( anRSSFeed, 0, 0, 5 );
$('contentDiv'). innerHTML = rssItemView;
```

(Помните, $() — это функция из библиотеки Prototype, позволяющая извлекать элементы DOM по их идентификаторам.) Теперь, когда у нас есть хороший набор абстракций для классов модели и представления, рассмотрим контроллер приложения, который свяжет воедино все эти компоненты.

13.7.3. Контроллер приложения

Класс RSSReader будет объединять функции, связанные с манипуляцией классами модели и представления, координируя все действия, соотнесенные с приложением чтения RSS-лент. Напомним, что интерфейс приложения реализован в виде слайд-шоу лент, когда каждая статья будет отображаться на определенный период времени, а затем постепенно переходить в следующую

статью. Приложение имеет кнопки, позволяющие переходить на следующую и предыдущую статьи, а также кнопки паузы/возобновления слайд-шоу. Наконец, имеются раскрывающийся список и кнопка добавления новых лент к исходному набору. Ниже перечислены пять необходимых категорий поведения, которые должны быть реализованы в приложении.

1. Построение объектов и первоначальная настройка.
2. Реализация слайд-шоу.
3. Создание эффектов перехода.
4. Загрузка RSS-лент средствами Ajax.
5. Обновление пользовательского интерфейса.

Чтобы уменьшить сложность и объем кода, требуемого для выполнения всех этих задач, используем библиотеку Prototype (позволяющую получить синтаксическую лаконичность), библиотеку Rico (обеспечивающую функциональные возможности, требуемые для эффектов перехода) и объект net.ContentLoader (предлагающий поддержку Ajax). Итак, начнем с первоначального построения и настройки.

Построение и настройка

Фактически разработка всех наших предыдущих компонентов начиналась с конструкторов. Будем верны традиции и сейчас — начнем с определения конструктора, которым в данном случае является простой метод, задающий первоначальные значения по умолчанию для какого-то состояния компонента и, как в других примерах, устанавливающего конфигурационные опции и выполняющего инициализацию. Учитывая это, конструктор RSSReader определяется так, как показано в листинге 13.27.

Листинг 13.27.* Конструктор RSSReader ; ^ X ^ B

```
RSSReader = Class.create*);
RSSReader.prototype = {
  initialize: function( readerld, options ) {
// О Установить значения по умолчанию
    this.id = readerld;
    this.transitionTimer = null;
    this.paused = false;
    this.visibleLayer = 0;
// © Настроить опции конфигурации
    this.setOptions(options);
// © Инициализировать поведение
    this.start{);
  ),
```

Приведенный выше конструктор принимает два аргумента: идентификатор и объект опций. Идентификатор используется как уникальная метка приложения и добавляется как префикс к идентификаторам кнопок, кото-

рые он должен будет определить в структуре DOM. Реализация сказанного будет показана ниже, в методе applyButtonBehaviors. Первое, что делает конструктор О, — это устанавливает значения по умолчанию для состояния. Далее (как и в большинстве других проектов) используется объект опций ©, с помощью которого задаются конфигурационные опции компонента. Для этого применяется метод setoptions. Наконец, в методе start © происходит все, что требуется для активизации компонента. Далее мы сначала рассмотрим конфигурацию, а затем перейдем к поведению.

За конфигурацию отвечает идиома setoptions, показанная в листинге 13.28. Давайте разберемся с реализацией setoptions и обсудим опции конфигурации.

Листинг 13.28. Метод setoptions jjjl

```
setOptions: function(options) {
  this.options = {
    slideTransitionDelay: 7000,
    fadeDuration : 300,
    errorHTML : '<hr/>Error retrieving content.<br/>'
  }.extend(options);
```

В приведенном методе setOptions указаны свойства приложения, которые мы решили сделать настраиваемыми. Свойство slideTransitionDelay задает число миллисекунд, в течение которых один "слайд" статьи виден до перехода во второй. Свойство fadeDuration задает время в миллисекундах, требуемое для полного затухания "слайда" (а следовательно, для "проявления" следующего "слайда"). Наконец, при возникновении ошибки в процессе загрузки RSS-ленты используется свойство errorHTML, задающее HTML-документ, отображаемый в качестве сообщения об ошибке. В приведенном коде также показаны значения данных свойств по умолчанию, которые сохраняются, если пользователь явно не укажет другие. Здесь стоит отметить, что компонент ожидает, что ему в качестве исходного набора лент для обработки будет передано свойство rssFeeds объекта опций. Поскольку мы в действительности не можем предложить разумное значение по умолчанию для этой величины, в методе setoptions она не определяется. Подразумевается, что приложение будет создано с объектом опций, подобным показанному ниже.

```
var options = {
  rssFeeds:  [ "http://radio.javaranch.com/news/rss.xml",
               "http://radio.j avaranch.com/pascarello/rss.xml",
               "http://radio.javaranch.com/bear/rss.xml",
               "http://radio.javaranch.com/lasse/rss.xml" ]  };
var rssReader = new RSSReader{'rssReader', options );
```

Итак, мы довольно быстро закодировали создание и настройку конфигурации. Теперь можно переходить к магическому методу start, который и запустит весь процесс. В следующих разделах мы кратко рассмотрим этот метод и посмотрим, к чему приводит его использование. Начнем, как обычно, с реализации, показанной в листинге 13.29.

Листинг 13.29. Метод start : ЯННННИННННННННН|

```
start: function() {
  this.applyButtonBehaviors();
  new Effect.FadeTo( this.getLayer(1), 0.0, 1, 1, {} );
  this.loadRSSFeed(0,true);
  this.startSlideShow{false);
},
```

Метод applyButtonBehaviors настраивает обработчики событий onclick для кнопок перехода к предыдущей/последующей статье, паузы/возобновления, а также кнопки добавления новой ленты. Этот метод мы рассмотрим следующим. Эффект затенения, представленный во второй строке, вызывает постепенное исчезновение с экрана видимого элемента div, чтобы его можно было скрыть при загрузке первого слайда. Обратите внимание на то, что в данной реализации мы не кодируем эффект самостоятельно, а используем готовое решение из библиотеки Rico, что уменьшает объем кода, который требуется писать, отлаживать и сопровождать. Метод loadRSSFeed инициализирует запрос Ajax на загрузку в первую ленту, а метод startSlideShow запускает таймер со значением slideTransitionDelay, инициируя таким образом слайд-шоу. Подробнее метод loadRSSFeed будет рассмотрен в разделе "'Загрузка RSS-лент с помощью Ajax", а метод startSlideShow — в разделе 'Реализация слайд-шоу". На этом мы, как и обещали, завершаем обсуждение построения и настройки, рассмотрев упомянутый в листинге 13.29 метод applyButtonBehaviors.

Как отмечалось ранее, метод applyButtonBehaviors подключает кнопки к методам, реализующим их поведение. Реализация этого метода показана в листинге 13.30.

Листинг 13.30. Метод applyButtonBehaviors .

```
applyButtonBehaviors: function!) {
  S(this.id + '__prevBtn') .onclick =    this.previous.bind(this) ;
  $(this.id + '_nextBtn').onclick • this.next.bind(this);
  $(this.id + '_pauseBtn').onclick - this.pause.bind(this);
  $(this.id + '_addBtn')-onclick = this.addFeed.bind(this);
```

Здесь, пожалуй, стоит напомнить, какой синтаксис и какие идиомы мы используем. В частности, мы задействовали пару синтаксических элементов библиотеки Prototype. Во-первых, это метод $, который можно рассматривать как вызов document .getElementByld. Во-вторых, метод bind неявно создает для нас замыкание, так что обработчик событий onclick для каждой кнопки может вызывать методы первого класса нашего компонента. Рассмотрим теперь детали реализации.

Реализация открывает неявный контракт между компонентом и HTML-разметкой приложения. Компонент создается с идентификатором, который хранится в атрибуте this.id. Позже этот идентификатор используется как префикс для нахождения различных элементов в разметке. В данном случае предполагается, что идентификаторы кнопок -— это метки, переданные в

Таблица 13.4. Атрибуты контроллера

Атрибут	Описание
this.currentFeed	Экземпляр RSSFeed, в текущий момент загруженный в память
this.feedIndex	Номер ленты, показываемой в текущий момент. Является индексом массива this.options.rssFeeds
this.itemIndex	Номер статьи, показываемой в текущий момент Является индексом массива статей объекта RSSFeed, показываемого в текущий момент

конструктор, к которым добавлено _prevBtn, _nextBtn, _pauseBtn и addBtn. Чтобы проиллюстрировать сказанное, рассмотрим приведенный выше код. В качестве идентификатора здесь используется rssReader, а компонент ожидает, что кнопки будут заданы следующим образом:

```
<input type="button" id="rssReader_prevBtn" value=" « " />
<input type="button" id="rssReader_pauseBtn" value=" I I " />
<input type="button" id="rssReader_nextBtn" value=" » " />
<input type="button" id="rssReader_addBtn" value="Add Feed" />
```

Видно, что контроллер RSSReader начинает приобретать форму, и теперь мы можем рассмотреть детали реализации слайд-шоу.

Реализация слайд-шоу

Теперь самое время поговорить об изменении семантики по сравнению с предыдущей версией сценария. Изначально мы загружали все RSS-ленты в память в момент запуска, а затем просто формировали переходы между внутренними представлениями лент. Это было просто, но масштабируемость такого решения вызывала определенные сомнения. Если мы регулярно прочитываем десятки или сотни RSS-лент, содержащих десятки статей, предварительная загрузка их всех была бы чересчур сильной нагрузкой на браузер. Поэтому в разделе реструктуризации мы попытаемся добиться нормальной масштабируемости и производительности приложения, так изменив семантику, чтобы за раз в память загружалась единственная RSS-лента. Все статьи RSSItems одной ленты побывают в памяти, однако в каждый отдельный момент времени в памяти будет присутствовать только один элемент RSSFeed. За то, на каком этапе находится слайд-шоу в процессе отображения содержимого, отвечают три атрибута контроллера, описанных в табл. 13.4.

Разобравшись с семантикой, переходим к навигации. Если мы планируем реализовать навигацию по всем статьям (элементам item) всех RSS-лент, то должны изучить несколько методов. Рассмотрим для начала пару методов перехода к предыдущей/следующей статье. Механизм перехода требуется не только для реализации явных событий, связанных с кнопками, но и для пассивного чтения при просмотре автоматизированного слайд-шоу.

Итак, рассмотрим пару булевых методов, сообщающих читателю, может ли он переходить вперед или назад. Реализация этих двух методов hasPrevious и hasNext показана в листинге 13.31.

... Листинг 13.31. Пара методов has Preyious/has Next : ,

```
hasPrevious: function() {
  return !(this.feedlndex == 0 && this.iteinlndex == 0);
},
hasNext: function() {
  return !(this.feedlndex == this.options.rssFeeds.length - 1 &&
           this.itemlndex == this.currentFeed.items.length - 1);
},
```

С помощью представленных методов определяется, доступен ли предыдущий или последующий слайд. При указанной реализации предыдущий "слайд" будет доступен в том случае, если в текущий момент приложение демонстрирует не первую статью первой ленты, а следующие "слайд" доступен, если мы читаем не последнюю статью последней ленты.

Рассмотрим теперь, что такое переход на предыдущую и последующую статью ленты. Начнем с метода previous (), показанного в листинге 13.32.

I Листинг 13.32. Метод previous ()[r] ^ 1 Д | Д

```
previous: function() {
  if ( !this.hasPrevious() ) return;
  var requiresLoad = this.itemlndex == 0;
  this.fadeOut( this.visibleLayer, Prototype.emptyFunction );
  this.visibleLayer = {this.visibleLayer + 1 ) % 2;
  if { requiresLoad )
    this.loadRSSFeedf this.feedlndex - 1, false );
  else
    setTimeout( this.previousPartTwo.bind(this),
                parselnt (this. options. fadeDuration/4) ). ;

Ь
previousPartTwo: function() {
  this.itemlndex'—; this.updateView();
Ь
```

Первое, что мы сделали при написании метода previous (), — поместили в самом его начале защитное условие. Если предыдущей статьи не существует, метод previous () ничего не делает. Если значение requiresLoad равно true, тогда содержимое RSS-статьи, на которую планируется переход, еще не загружено. Если мы находимся на первой статье ленты и переходим назад, требуется загрузка предыдущей ленты. Метод затухания, подробно рассмотренный в разделе "Эффекты перехода", постепенно ослабляет видимый слой. Дальнейшие действия этого метода зависят от того, требуется ли перед отображением загрузка какого-либо содержимого. Если да, то мы инициируем загрузку необходимых данных посредством метода loadRSSFeed(). Первым параметром данного метода является номер загружаемой ленты, вторым — булево значение, указывающее направление: true — вперед, false (как в данном случае) — назад. Если же содержимое статьи уже загружено, то мы вызываем previousPartTwo() после паузы, равной одной четвертой общей длительности fadeDuration. Во "второй части" данного метода про-

сто обновляется свойство itemlndex и вызывается функция updateView(), приводящая к затуханию соответствующего слайда.

Ну что, запутались? Ладно, объясняем нормальным языком: если текст, который нужно отобразить, не загружен, его загрузка начинается немедленно, что приводит к обновлению пользовательского интерфейса сразу же после поступления данных. Время, которое требуется на прием данных, используется для естественной реализации затухания! С другой стороны, если содержимое уже загружено {т.е. мы переходим на другую статью загруженной ленты), то мы вводим искусственную задержку (четверть времени затухания) перед проявлением следующей статьи. Довольно хитро, правда?

Метод next(), показанный в листинге 13.33, представляет собой реализацию алгоритма, обратного к приведенному выше.

```
next: function{} {
  if ( ithis.hasNext() ) return;
  var requiresLoad =
    this.itemlndex == (this.currentFeed.items.length - 1);
    this.fadeOut( this.visibleLayer, Prototype.emptyFunction );
    this.visibleLayer = (this.visibleLayer + 1 ) % 2;
  if ( requiresLoad >
    this.loadRSSFeed{ this.feedlndex + 1, true );
   else
    setTimeout( this.nextPartTwo.bind(this),
                parselnt(this.options.fadeDuration/4) );

Ь
nextPartTwo: function() {
  this.itemlndex++; this.updateView();
Ь
```

Выглядит знакомо? Метод next() использует противоположную логику индексирования, а во всем остальном идентичен приведенному выше алгоритму. Обратите внимание на то, что при каждом переходе пара методов previous()/next{) переключает видимый слой с одного слайда на другой с помощью такого выражения:

```
this.visibleLayer = (this.visibleLayer + 1) % 2;
```

Таким образом, мы сообщаем коду, который в конечном счете обновит пользовательский интерфейс (после загрузки содержимого или явного вызова функции updateView()), на какой слой помещать результат. Напомним, что контекстная область приложения содержит HTML-разметку, которые выглядит примерно следующим образом:

```
<!— Контекстная область —>
<div class="content" id="rssReader_content">
  <div class="layerl">Layer 0</div>
  <div class="layer2">Layer K/div>
</div>
```

Здесь visibleLayer — целочисленное свойство, отслеживающее, в какой элемент div требуется помещать содержимое. Индекс 0 указывает, что при об-

новлении пользовательского интерфейса содержимое должно помещаться на слой 0. Значение 1 приводит к вставке данных на слой 1.

Подведем итог. У нас есть методы, с помощью которых реализуется переход вперед/назад по набору статей, и мы можем их использовать для создания методов слайд-шоу. Рассмотрим их В листинге 13.34 показаны метод startSlideShow, который, как вы помните, вызывался из метода start(), и дополняющий его метод nextSlide().

ЛистинМ3.34. Методы навигации по слайд-шоу

```
startSlideShow: function(resume) {
  var delay = resume ? 1  : this.options.slideTransitionDelay;
  this.transitionTimer = setTimeout{this.nextSlide.bind(this),
                                    delay);
K
nextSlide:  function{)  {
  if ( this.hasNextO  )
    this.next();
  else
    this.loadRSSFeed(0, true);
  this.transitionTimer = setTimeout{
                                    this.nextSlide.bind(this),
                                    this.options.slideTransitionDelay ) ;
),
```

Приведенный метод startSlideShow вызывает nextSlide после заданной задержки. В зависимости от того, возобновляем ли мы слайд-шоу после паузы или нет, задержка равна либо slideTransitionDelay, либо одной миллисекунде (практически мгновенно). Таким же простым является и метод nextSlide, вызывающий метод next() при наличии следующего слайда. Если текущий слайд является последним в данной ленте, вызывается loadRSS-Feed(0,true) и мы переходим на начало ленты. После этого просто устанавливается таймер и процесс повторяется. Все очень легко!

Мы уже говорили, что слайд-шоу можно приостановить с помощью кнопки паузы, однако соответствующий метод еще не был реализован. Восполним этот пробел и создадим метод, показанный в листинге 13.35.

riv*i к in i i O . O J . iv (C 1чЛЦ u a U o c

```
pause: function() {
  if ( this.paused )
    this.startSlideShow(true);
  else
    clearTimeout( this.transitionTimer );
  this.paused = !this.paused;
- ±
```

Метод pause переключает состояние паузы нашего слайд-шоу. Текущее состояние отслеживается с помощью булева атрибута this.paused. Если слайд-шоу уже приостановлено, метод pause вызывает функцию start-SlideShow, в качестве значения свойства resume передавая значение true;

в противном случае метод обнуляет атрибут transitionTimer, подавляющий все переходы между слайдами до тех пор, пока кнопка паузы не будет нажата повторно.

Чтобы завершить реализацию слайд-шоу, нужно реализовать функцию, которая бы позволила добавлять в этот процесс новые RSS-ленты. Изучая код функции applyButtonBehaviors(), мы видели, что кнопка добавления новой ленты вызывает метод addFeed. Давайте реализуем этот метод (см. листинг 13.36) и будем считать, что мы сделали все для получения управляемого слайд-шоу.

Листинг 13.36. Метод addFeed

```
addFeed: function() {
  var selectBox = $(this.id + '_newFeeds');
  var feedToAdd = selectBox.options[
                    selectBox.selectedIndex ].value;
  this.options.rssFeeds.push(feedToAdd);
},
```

Приведенный выше метод также зависит от неявного контракта с HTML-разметкой с позиции договоренности об именовании списка дополнительных RSS-лент. Идентификатор данного списка должен быть идентификатором приложения с суффиксом _newsFeeds. Метод addFeed просто принимает выбранную RSS-ленту в списке и добавляет ее в конец массива this.options.rssFeeds. Больше ничего не требуется! Ну как вам это нравится — добавление новых функциональных возможностей требует всего пары строк кода!

На этом мы завершаем разработку методов, связанных со слайд-шоу. Теперь рассмотрим кратко методы, поддерживающие эффекты перехода.

Эффекты перехода

В нашем коде упоминалось несколько методов, поддерживающих затухающие переходы между слайдами. Давайте рассмотрим эти переходы подробнее. Итак, прежде всего мы определим пару методов fadeIn() и fadeOutO, как показано в листинге 13 37.

Листинг 13.37. Пара методов fadeIn<)/fadeOut()

```
fadeIn: function( layer, onComplete ) {
  this.fadeTo( 0.9999, layer, onComplete );

fadeOut: function( layer, onComplete ) {
  this.fadeTo( 0.0001, layer, onComplete );
```

Оба указанных метода делегируют свои полномочия методу fadeToO (он разобран ниже). Они передают методу fadeToO параметр прозрачности — значение между 0 и 1, где 0 соответствует полностью невидимому слою, а 1 — полностью видимому. В некоторых браузерах значение, очень

Таблица 13.5. Параметры Effect.FadeTo

Параметр	Описание
this.getLayer(layer)	Затухающий элемент DOM
n	Степень прозрачности (значение между 0 и 1)
this.options.fadeDuration	Время затухания
12	Число промежуточных этапов
{complete: onComplete}	Обратный вызов, запускаемый после завершения

близкое к 1 (но не точно 1), похоже, немного уменьшает мерцание изображения, поэтому вместо 1 мы будем использовать величину 0,9999. Чтобы указать, какой слой должен затухать, мы передаем функции номер слоя (0 или 1). Наконец, последний аргумент — это функция, обеспечивающая перехват обратного вызова сразу после завершения перехода. Реализация метода fadeTo() показана в листинге 13.38.

```
fadeTo: function( n, layer, onComplete ) {
  new Effect.FadeTo( this.getLayer(layer),
                     n,
                     this.options.fadeDuration,
                     12,
                     {complete: onCompleteJ );
Ь
```

Из-за внезапного приступа лени или в качестве продуманного методологического приема мы решили не изобретать заново эффект перехода. Вместо этого мы используем метод Effect. FadeTo из библиотеки Rico, который и выполнит всю работу. Параметры Effect.FadeTo приведены в табл. 3.5.

Чтобы получить элемент div, соответствующий затухающему слою содержимого, мы использовали вспомогательный метод getLayerO, показанный в листинге 13.39.

Листинг 13.39. Метод aetLaverO _ | Ц ^ Н ^ ^ ^ ^ Д | Н ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ И в

```
getLayer: function(n) {
  var contentArea - $(this.id+'_content');
  var children = contentArea.childNodes; var j = 0;
  for ( var i = 0 ; i < children.length ; i++ ) {
    if ( children[i].tagName &&
         children[i].tagName.toLowerCase() == 'div' ) {
         if ( j -~ n ) return children[i];

    }
  }
  return null;
```

Данный метод находит область содержимого, предполагая, что ее идентификатор построен как идентификатор приложения с дописанным в конец значением _content. Найдя элемент содержимого, метод прослеживает его потомков и находит n-й дочерний элемент div.

Все, закончили и с переходами. Теперь рассмотрим тему загрузки RSS-лент посредством магии Ajax.

Загрузка RSS-лент с помощью Ajax

Мы уделили довольно много внимания темам создания компонента и обеспечения богатой семантики слайд-шоу, а также необычных технологий DHTML для переходов между слайдами. Однако если все это не создано на основе инфраструктуры Ajax, гордиться совершенно нечем. Все дело в том, что идеальный пользовательский интерфейс может обеспечить только Ajax со своей масштабируемостью и индивидуальным извлечением данных *плюс* сложный DHTML со своими богатыми возможностями и эффектами. Говорить об этом можно долго, так что лучше все лее рассмотрим требуемую инфраструктуру Ajax, начав с метода, приведенного в листинге 13.40 и выполняющего загрузку RSS-ленты в память.

```
loadRSSFeed: function(feedIndex, forward) {
  this.feedIndex = feedIndex;
  this.iteralndex = forward ? 0 : "last";
  new net.ContentLoader(this,
                        this.options.rssFeeds[feedIndex],
                        "GET", [] ).sendRequest();

},
```

Данный метод с помощью давно знакомого объекта net.ContentLoader формирует запрос Ajax, передавая в качестве параметра URL RSS-ленты, заданный в массиве this.options.rssFeeds. Параметр forward представляет собой булеву величину, указывающую, что мы загружаем (или не загружаем) новое содержимое вследствие перехода на следующую статью. Далее, согласно этой информации, обновляется свойство itemIndex. Обратите внимание на то, что, если мы переходим назад, itemIndex получает значение, равное last, а не целочисленной величине. Такое решение объясняется тем, что свойство itemIndex должно указывать индекс последней статьи предыдущей RSS-ленты. Единственная проблема заключается в том, что мы не знаем, сколько статей в ленте, поскольку она еще не загружена.

Напомним, что согласно неявному контракту с net.ContentLoader нам требуются методы ajaxUpdate и handleError. Разберем вначале метод ajaxUpdate, показанный в листинге 13.41, и посмотрим, как с помощью данной реализации разрешить нашу дилемму индексирования.

Листинг 13.41. Метод *1a ^Аз ^ШШШШШЯИШШШШШШШШШШШ

```
aj axUpdate: function(request) {
if ( window.netscape Sfi
  window.netscape.security.PrivilegeManager.enablePrivilege)
    netscape.security.PrivilegeManager.enablePrivilege(
       'UniversalBrowserRead');
  this.currentFeed =
    RSSFeed.parseXML(request.responseXML.documentElement);
  if ( this-itemlndex == "last" )
    this.itemlndex = this.currentFeed.items.length - 1;
    this.updateView();
```

Ш

Метод ajaxUpdate начинает с проверки того, что он работает в среде, предлагающей PrivilegeManager. Если да, тогда он затребует привилегию UniversalBrowserRead. Как отмечалось выше, это сделано для того, чтобы наше приложение могло запускаться локально в браузерах Mozilla.

В приведенном коде this.currentFeed — это экземпляр объекта модели RSSFeed, определенный в разделе "Модель". Он соответствует одной ленте RSSFeed, загруженной в память после отклика Ajax. Если значение this.itemlndex равно last (такое значение устанавливается методом loadRSSFeed при возврате к предыдущей статье), свойство itemlndex обновляется в соответствии с реальным числом статей в новой загруженной ленте RSSFeed. Наконец, в ответ на вызов updateView() обновляется пользовательский интерфейс.

Не забывайте, что нам еще нужно определить метод handleError (не только потому, что это подразумевает контракт с net.ContentLoader, но и потому, что нам действительно нужно какое-то средство обработки ошибок). Если ленту RSS загрузить не удается, мы предоставляем сообщение, показанное в реализации handleError. Разумеется, возможны (и желательны) более сложные реализации.

```
handleError: function(request) {
  this.getLayer(this.visibleLayer).innerHTML =
    this.options.errorHTML;
Ь
```

Итак, мы реализовали все необходимые возможности Ajax в приложении RSSReader, остался последний штрих — написать пару методов, обновляющих пользовательских интерфейс.

Обработка пользовательского интерфейса

Напомним, что классы модели и представления мы создавали так, чтобы облегчить себе реструктуризацию кода приложения. Теперь, переходя к контроллеру, отвечающему за обновление пользовательского интерфейса, мы можем считать, что сделана почти все работа. Это действительно так. Рассмотрим, например, метод updateView () (листинг 13.42), на который мы неоднократно ссылались в ходе предыдущей работы.

ЛИСТИНГ 13.42. Метод updateViewf)

```
updateView: function() {
  var rssItemView • new RSSItemView(
    this.currentFeed.items[this.itemlndex],
    this.feedlndex,
    this.itemlndex,
    this.options.rssFeeds.length )j
    this.getLayer(this.visibleLayer).innerHTML = rssItemView;
    this.fadeln( this.visibleLayer,
    this.bringVisibleLayerToTop.bind(this) );
```

— ±

Как видите, метод updateview() делегирует всю тяжелую работу классу представления, вызывая экземпляр этого класса и используя его как значение свойства innerHTML видимого слоя (в конечном счете это приводит к постепенному проявлению данного слоя). Три строчки кода. Совсем немного. Обратите внимание на то, что как только слой попадает в поле зрения, мы вызываем метод завершения работы bringVisibleLayerToTop. Этот метод обновляет стилевое свойство слоя z Index, вследствие чего данный слой становится выше другого, постепенно его затеняя. Реализация функции bringVisibleLayerToTop () выглядит следующим образом:

```
bringVisibleLayerToTop: function() {
  this.getLayer(this.visibleLayer).style.zlndex = 10;
  this.getLayer((this.visibleLayer+1)%2).style.zlndex = 5;
}
```

Вот и все, что можно сделать с точки зрения работы с пользовательским интерфейсом. Разделяя обязанности между классами модели, представления и контроллера, мы смогли получить простую управляемую архитектуру.

13.7.4. Выводы

Реструктуризация заключается в таком изменении кода, чтобы мы получили MVC-представление нашего приложения для чтения RSS-лент. Мы создали класс модели RSSFeed, инкапсулирующий концепцию RSS-ленты, кроме того, создали класс RSSItem. Для инкапсуляции концепции представления элемента RSSItem в контексте родительского элемента RSSFeed был создан класс представления RSSItemView. Наконец, мы связали классы модели и представления с помощью класса контроллера RSSReader, объединяющего управление событиями и реализацию слайд-шоу с эффектами перехода.

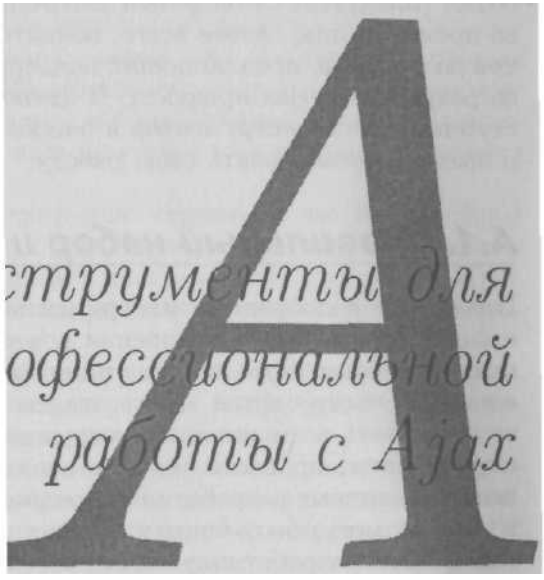
13.8. Резюме

В данной главе инфраструктура Ajax позволила получать информацию непосредственно с рабочего стола, не требуя коммерческого клиентского приложения, экономя наши деньги и позволяя настраивать решение согласно насущным потребностям. Нам удалось загрузить несколько XML-файлов и получить при этом только ту информацию, которая нас интересует. Мы разра-

ютали HTML-каркас приложения и применили правила CSS, что позволило сегко настраивать внешний вид программы. Используя DHTML, мы смогли >азработать богатый пользовательский интерфейс, позволяющий пользовате-им пропускать сообщения, приостанавливать их поток, а также добавлять ю мере необходимости новые ленты. Все это стало возможным благодаря ин-фраструктуре Ajax, разрешившей получать RSS-ленты с Web-сайтов. Изме-шв несколько операторов, мы можем легко настроить приложения на чтение побой XML-ленты. Кроме того, можно разрабатывать собственные форматы •CML для отображения новостей, рекламы и всего остального, что обычно раз-летается на Web-сайтах. Наконец, мы провели реструктуризацию сценария ;огласно архитектуре "модель-представление-контроллер" (MVC), повысив штаемость и удобство эксплуатации кода.

Часть V

Приложения


А
струменты для
офессиональной
работы с Ajax

Работать с технологией Ajax очень просто. Для выполнения всей работы вам потребуется лишь небольшой набор инструментов. Вообще, сложные приложения Ajax можно создавать, используя только Web-браузер, текстовый редактор и имея доступ к Web-серверу (который располагается либо на вашем компьютере, либо на каком-то Web-сайте, к которому вы имеет доступ). Тем не менее хороший набор инструментов очень важен для программиста, поэтому в настоящее время существует несколько таких довольно сложных наборов.

В настоящее время специализированных интегрированных сред разработки (Integrated Development Environment — IDE) Ajax не существует, хотя со временем они, скорее всего, появятся. Однако есть несколько инструментов разработки, предлагающих поддержку некоторых составляющих процесса разработки Ajax-продукта. В данном приложении представлен обзор доступных типов инструментов и рассказывается, как с их помощью разумнее и быстрее организовать свою работу.

АЛ. Правильный набор инструментов

Правильно подобранные инструменты разработки могут оказаться бесценными с точки зрения ускорения повторяющихся или сложных процессов и оказать огромнейшее влияние на производительность разработчика. "Неправильные" инструменты могут отвлекать внимание от необходимой работы, ограничивать возможности и вынуждать разработчика использовать только определенные процессы или совмещать в действительности несовместимые вещи. Различные разработчики предпочитают разные инструменты, которые, к тому же могут быть более или менее удачными для различных типов проектов. Иногда разработчику стоит потратить немного своего времени и выбрать набор инструментов, лучше всего подходящий для поставленной задачи. Пожалуй, лучше всего данную мысль сформулировал Авраам Линкольн:

"Если мне дадут шесть часов на то, чтобы срубить дерево, первые четыре из них я потрачу на затачивание топора".

Если вы будете поступать по такому же принципу, отдача может быть огромной. Разумеется, важно соблюдать и разумное соотношение между отладкой инструментов и их использованием, особенно если речь идет о нестандартной промежуточной ситуации, например, как в случае с существующими инструментами Ajax.

А.1.1. Получение совместимых инструментов

В настоящее время доступно множество инструментов в виде бесплатных пакетов, проектов с открытым исходным кодом и коммерческих продуктов. Мощных специализированных инструментов, предназначенных исключительно для Ajax, пока что не существует, однако имеются средства разработки Web-приложений, многие из которых поддерживают JavaScript, HTML и CSS.

Как уже говорилось в главе 1, Ajax использует те лее технологии, что и классическое Web-приложение, правда, несколько непривычным способом. Приложение Ajax построено не как набор небольших последовательных страниц, а как несколько страниц (чаще всего одна), выполняющих в ответ на действия пользователя разнообразные программные переходы и в фоновом режиме асинхронно общающихся с сервером. Кроме того, поскольку в ходе описанного процессе генерируется довольно много JavaScript-кода, программист Ajax, скорее всего, будет опираться на структуры JavaScript (наиболее популярные из них описаны в главе 3).

Вследствие подобных отличий классического Web-приложения и приложения Ajax возникают два момента. Во-первых, инструмент может предполагать наличие основанных на страницах процессов, применение которых в Ajax-приложении нецелесообразно. Во-вторых, поддержка JavaScript может опираться на использование определенного набора функций или схем кодирования, несовместимых с выбранными пользователем структурами сторонних производителей.

Таким образом, отличия преимущественно связаны с высокоуровневой структурой приложения, а не с его деталями. Кроме того, два обозначенных момента скорее всего относятся к сложным инструментам (например, IDE), а не к более простым средствам, подобным текстовому редактору с поддержкой JavaScript.

В общем, выбирая инструменты для своего проекта Ajax, помните о том, что мы сказали. К этому вопросу мы еще вернемся, когда будем рассматривать различные инструменты, представленные на рынке.

Наконец, стоит отметить, что многие существующие инструменты характеризуются возможностью расширения с помощью различных включаемых модулей (plug-in). Сложные инструменты, например универсальные IDE и Web-браузеры, по-разному используются различными категориями пользователей. Расширения позволяют модифицировать стандартное приложение, оснащая его функциями, требуемыми конкретным пользователям, при этом стандартный набор возможностей не затрагивается, приложение просто становится более мощным, включая функциональные возможности, не реализованные первоначальной командой разработчиков. Существуют два достойных упоминания класса расширений — Eclipse IDE, который, хотя и является преимущественно инструментом Java-разработчика, поддерживает благодаря расширениям богатый диапазон функциональных возможностей Ajax, и браузер Firefox, сообщество пользователей которого разрабатывает модули (или расширения), в частности, предназначенные для Web-разработчиков.

Сообщества разработчиков модулей для Eclipse и Firefox достаточно велики, так что вы вполне можете найти готовое расширение, более-менее отвечающее вашим требованиям. Кроме того, в среде Web-разработки и вычислительных технологий существует традиция создания индивидуальных инструментов, поддержанию которой способствует и технология расширений. Рассмотрим эту традицию более подробно.

A.1.2. Создание собственных инструментов

В качестве альтернативы покупному или загружаемому готовому инструменту вы всегда можете создать свой. После сказанного выше об IDE и о полномасштабных инструментах это может звучать устрашающе и нереалистично. Разумеется, мы не считаем, что проект Ajax следует начинать с написания собственной IDE!

В культуре Unix имеется давняя традиция разработки небольших инструментов, выполняющих одну небольшую работу. Инструменты подобного типа легко создать, потратив совсем немного времени, и сопровождение данных средств также не представляет особых проблем. В качестве примеров таких инструментов можно привести классы секундомера, которые мы разработали для профилирования JavaScript-кода в главе 7, а также консоль вывода информации, которую мы продемонстрируем в разделе A.3.4.

Инструменты, написанные на JavaScript, и другие технологии Ajax хороши тем, что их можно переносить на любой браузер. Однако возможности, доступные в браузере, существенно ограничены "благодаря" модели безопасности JavaScript (см. главу 6). Иногда имеет смысл создать инструмент в виде самодостаточной программы, используя .NET, Java или любой другой язык программирования. Если вы решили поступить именно так, вам помогут отладчики HTTP, описанные в разделе A.3.3.

Промежуточным подходом между созданием инструментов в браузере и написанием самодостаточной программы является разработка модуля. Многие современные большие средства Web-разработки поддерживают создание модулей, причем некоторые позволяют делать это довольно легко. Как отмечалось в предыдущем разделе, здесь выделяются IDE Eclipse и Web-браузер Firefox. Eclipse даже предлагает набор модулей, связанных со стандартным загружаемым приложением, которые облегчают написание новых модулей. Тем не менее разрабатывать модули сложнее, чем инструменты, встроенные в браузер, и, пожалуй, это оправдано только тогда, когда речь идет о фрагменте большого проекта.

При создании Ajax-приложения полезными могут оказаться множество инструментов. В настоящее время они разрознены; и процесс их активного сопровождения требует постоянного внимания. Остановимся на этом моменте подробнее, а затем рассмотрим несколько заслуживающих внимания инструментов

A.1.3. Сопровождение набора инструментов

Как уже отмечалось выше, инструменты Ajax обрывочны, что несколько непривычно для программистов Java и NET, привыкших опираться на удобный набор Eclipse, NetBeans или Visual Studio.

Разработчики сами должны заботиться о том, чтобы их наборы инструментов не устаревали. Поскольку центра притяжения в виде фактического стандарта IDE не существует, данная задача является более проблематичной, чем хотелось бы, поэтому в общем случае приходится полагаться на знакомых, рассылки, информационные порталы, блоги и другие распределенные средства передачи информации, существующие в Интернете.

Наиболее важным инструментом любого разработчика является редактор, в котором он набирает свой код. Рассмотрим, что собой представляет несколько популярных редакторов.

A.2. Редакторы и IDE

Сложность инструментов, используемых для редактирования кода, варьируется в очень широких пределах — от простой программы Notepad (Блокнот) для сложных интегрированных сред разработки, различным образом моделирующих объекты кода. JavaScript, HTML и CSS поддерживаются не так хорошо, как промышленные языки, подобные C#, Visual Basic и Java, однако диапазон предлагаемых функциональных возможностей достаточно широк, чтобы было из чего выбирать. Давайте посмотрим, какие типы функций могут нам потребоваться, а затем изучим доступные на сегодняшний день продукты.

A.2.1. Что требуется от редактора кода

Редакторы кода могут многое, возможно, даже слишком многое. Значительная часть этих возможностей предназначена для удовлетворения индивидуальных запросов пользователей. Из-за этого одни разработчики предпочитают простой инструмент обработки текста; другие любят визуальные подсказки и ключи полномасштабной интегрированной среды разработки. Относительно поддержки кода Ajax (т.е. кода HTML, CSS и JavaScript) можно отметить, что помощь редактора может выражаться несколькими способами. Многие из них могут показаться имитацией содержимого классического Web-приложения, однако поскольку код классического приложения Ajax обычно больше и имеет более четкую структуру, поддержка редактором структурирования приобретает первостепенное значение. Итак, перечислим полезные элементы, которые стоит иметь хорошему редактору кода.

Работа с несколькими файлами

Данное требование является стандартным, однако в любом случае упомянуть о нем стоит. Проекты Ajax обычно составлены из большого числа файлов, поэтому редактор, не позволяющий одновременно работать с несколькими файлами или буферами (как Windows Notepad), очень быстро станет раздражать. Практически все редакторы кода позволяют открывать несколько файлов, доступ к которым осуществляется через расположенные рядом закладки, панель выбора или другое похожее средство.

Подсветка синтаксиса

Как и ранее, визуальное выделение синтаксиса в настоящее время является одной из стандартных возможностей, которую поддерживает большинство редакторов программного кода. Под данной возможностью понимается выделение цветом, курсивом или обозначение любым другим образом ключевых слов языка, символов, строк в кавычках и комментариев, облегчающее чтение кода.

Большинство редакторов поддерживает выделение синтаксиса различных языков с помощью цветов; часто файлы-определения синтаксиса можно подключать как модули. Одной из характерных особенностей программирования Ajax является использование нескольких языков. На стороне клиента задействованы HTML. CSS, XML и JavaScript (все они только выигрывают от визуального выделения синтаксиса), кроме того, применяется часть (или все) языков из следующего набора: Java, C#, VB и более сложные ASP, PHP и JSP (в которых чередуются блоки "родного" кода и HTML-разметки). Отметим, что выделение синтаксиса полного набора языков, используемых в проектах Ajax, поддерживается не во всех редакторах.

Поддержка кода высокого уровня

Расцвечивание кода предлагает хорошие визуальные подсказки, однако некоторые редакторы на этом не останавливаются и моделируют код на более высоком уровне объявлений объектов, функций и методов. Такой подход позволяет создавать множество новых инструментов, например, планировщиков, резюмирующих содержимое файла, средств навигации (карт иерархии объектов), кроме того, с помощью средств поиска можно определять, когда используется конкретное свойство либо вызывается некоторый метод или функция. Когда код начинает разрастаться, инструменты подобного типа просто незаменимы.

Инструменты организации проекта

Некоторые интегрированные среды разработки идут дальше моделирования определений отдельных объектов и позволяют интегрировано управлять всем кодом проекта, распознавая связи между различными компонентами и ресурсами, объединение которых и дает конечный продукт. Основным преимуществом такого подхода в интегрированной среде разработки для компилируемого языка является возможность создания всего проекта в исполняемой форме, однако это не принципиально, когда речь идет об Ajax, где все ресурсы клиентской части приложения разворачиваются в форме, удобной для восприятия человеком. Тем не менее такая возможность может быть полезна при работе с кодом серверной части приложения.

Кроме того, с помощью средств организации проектов можно развернуть проект на Web-сервере и даже управлять самим Web-сервером (либо контролируя внешний сервер посредством удаленных вызовов процедур, либо внедрив простой сервер в интегрированную среду разработки). Инструмент, поддерживающий код на уровне проекта, может освободить разработчика от задачи сопровождения системы создания-развертывания.

Управление версиями

Управление версиями является необходимостью в больших проектах и "хорошим тоном" в проектах любого размера. Сами по себе системы управления версиями обычно работают с текстовыми и бинарными файлами, не особо вникая в их высокоуровневую семантику, поэтому назвать продукт, который

лучше других подойдет именно для **приложений Ajax,** весьма затруднител! но. В любом случае наличие в вашем наборе инструментов хорошего средств управления версии не помешает.

Разработка на нескольких языках: интеграция клиента и сервера

Как отмечалось ранее, многие проекты Ajax требуют помимо развертывЕ ния множества технологий Web-браузеров наличия серверного компонент? Вы можете написать серверную часть на JavaScript, но это не принято, та как обычно разработчики Ajax-приложений используют на стороне клиент и сервера различные языки. Для кодирования клиентской и серверной част приложения можно использовать два абсолютно разных набора инструмен тов, но некоторые типичные задачи взаимодействия включают быстрое пе реключение уровней, поэтому редактор, поддерживающий все используемы языки, может оказаться весьма кстати.

Таким образом, мы указали основные критерии, определяющие выбо] редактора кода, представляющего собой либо простой тестовый редактор либо интегрированную среду разработки. В следующем разделе рассмотре! ряд инструментов, существовавших на момент написания книги.

Двусторонний визуальный дизайн

Многие инструменты Web-дизайна предлагают WYSIWYG-средства визуаль ной разработки Web-страниц. Данные средства хорошо подходят для создания прототипов, но обычно они плохо реагируют на более динамический подход Ajax к переупорядочению пользовательского интерфейса путем манипуляций с DOM. Большинство визуальных редакторов также позволяет программисту переключаться на просмотр HTML-кода в текстовом виде. Если при работе над Ajax-приложением вы используете подобные средства, обратите внимание на то, чтобы они сохраняли элементы, которые не понимают. в частности, комментарии и специальные дескрипторы и атрибуты, которые могут использоваться кодом JavaScript для переупорядочения дерева DOM.

А.2.2. Существующие продукты

Технологии Ajax поддерживаются несколькими текстовыми редакторами. Наше рассмотрение начнется с редакторов для программистов, затем будут изучены более сложные интегрированные среды разработки.

Текстовые редакторы

В настоящее время существует множество текстовых редакторов с открытым исходным кодом, бесплатных или условно-бесплатных, предназначенных для различных операционных систем. В качестве бесплатно распространяемых инструментов можно назвать TextPad, Notepad2, EditPlus, ветеранов Unix Vim и Emacs, а также расширяемое межплатформенное средство jEdit, модульная система которого позволяет создавать что-то, подобное интегрированным средам разработки. Несколько распространенных текстовых редакторов с поддержкой JavaScript показано на рис. АЛ.

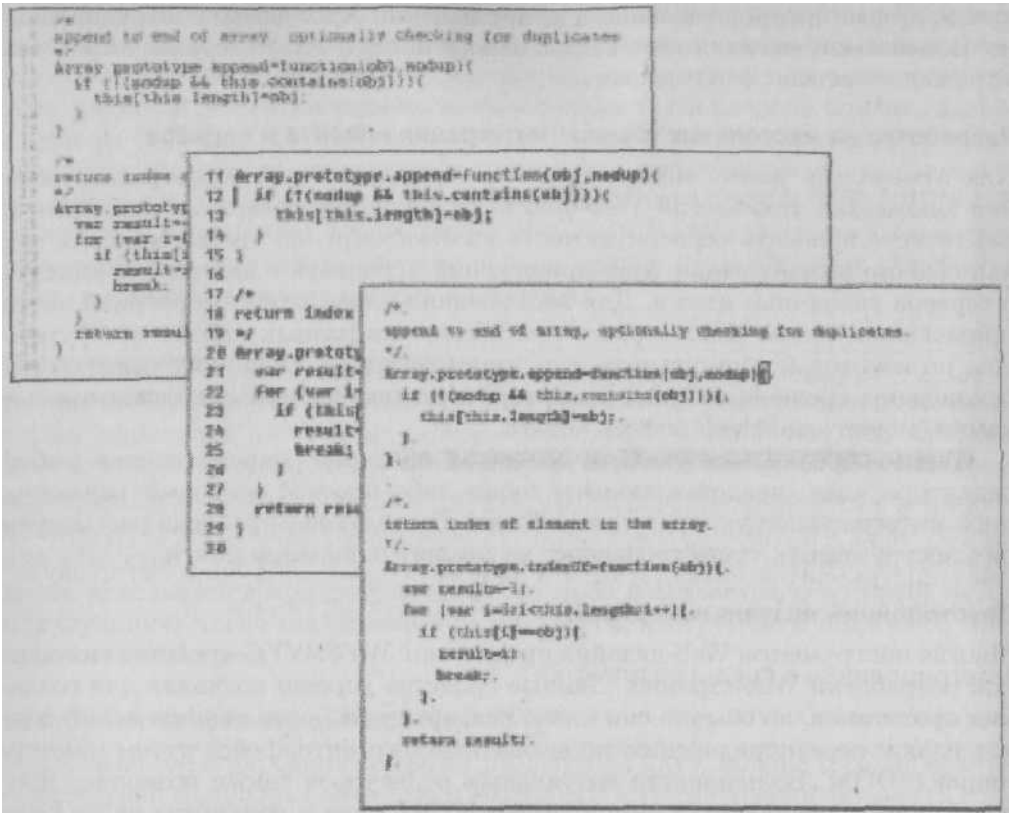


Рис. А.1. Текстовые редакторы, предлагающие подсветку синтаксиса JavaScript (слева направо): *TextPad, Gvim и* jEdit

TextPad **предлагает** потрясающе разнообразную баз}' файлов с определениями синтаксиса, включающую несколько файлов для CSS, JavaScript, XML и HTML, а также наиболее популярных серверных языков. Имеется минимальная поддержка запуска в текущем файле определенных пользователем команд (например, компиляторов). TextPad работает только в операционной системе Microsoft Windows. Сходные возможности предлагают и средства NotePad2 и EditPlus.

Инструмент jEdit создан на основе Java; его можно запускать на любой платформе, поддерживающей этот язык. Данное средство позволяет визуально выделять синтаксис более 100 языков, включая все основные языки, используемые в Ajax-проектах. Использование системы модулей позволяет получать дополнительные возможности, хорошо интегрирующиеся в систему. Модули допускают автоматическую навигацию, загрузку и установку непосредственно из jEdit. Существуют полезные модули, предлагающие проверку синтаксиса, поддержку отладчиков, компиляторов и интерфейсов управления версиями, а также специальную поддержку CSS и XSLT.

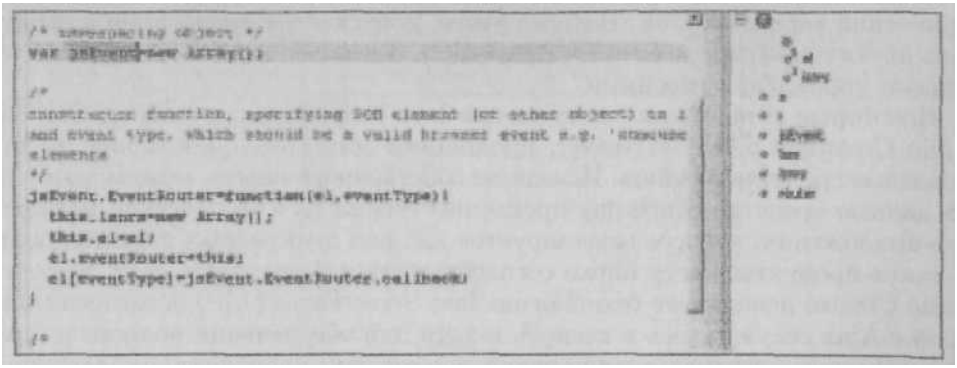


Рис. А.2. Модуль JavaScript-редактора для Eclipse предлагает элементарную поддержку контурного представления объектов JavaScript, но не позволяет в полной мере обрабатывать объектно-ориентированный синтаксис

Vim и Emacs представляют собой мощные расширяемые текстовые редакторы, поддерживающие традиции операционных систем Unix (хотя для обоих инструментов существуют и Windows-версии). Оба средства включают доскональный режим поддержки JavaScript-кодирования.

Интегрированные среды разработки

Корпоративные языки программирования, подобные .NET и Java, имеют интегрированные среды разработки с долгой историей. Соответствующий рынок достаточно развит, а в последние годы на нем появилось несколько богатых бесплатных интегрированных сред разработки с открытым исходным кодом. Среды, предназначенные для серверных языков, часто достаточно расширяемы, чтобы поддерживать и разработку клиентских частей Ajax-приложений.

Основным представителем технологий Microsoft является средство Visual Studio, поддерживающее Web-разработки с помощью компонента Visual InterDev, понимающего языки JavaScript и CSS. Не так давно стали бесплатно распространяться урезанные версии данного продукта (Visual Studio Express), в том числе версия, предназначенная для Web-разработчиков.

Наиболее известной IDE Java-приложений является средство Eclipse, поддерживаемое IBM, — инструмент, предназначенный преимущественно для разработки Java-приложений и снабженный сложным набором модулей, которые написаны конкретно для Java-разработчиков. Семейство данных модулей развивается довольно бурно и включает несколько относительно простых JavaScript-модулей, обеспечивающих визуальное выделение синтаксиса и контурное представление методов и классов (рис. А.2).

В Eclipse 3.1 (часть проекта Web Tools Platform) был разработан богатый набор модулей для Web-разработчиков; данный проект предлагает также поддержку серверных технологий J2EE, редакторы для JavaScript, XML, HTML и CSS. Кроме того, Eclipse предлагает богатые возможности управления кодом на уровне проекта и *полную* интеграцию с новейшим средством

управления версиями CVS. Помимо этого, допускается интеграция с продуктами других **производителей** — Subversion. Visual SourceSafe и другими системами управления версиями.

Некоторые корпоративные средства Java-разработки, подобные Sun Java Studio Creator и SAP NetWeaver. предлагают для Web-приложений высокоуровневые средства дизайна. Исходя из собственного опыта, можем заметить, что данные средства основаны преимущественно на классической метафоре Web-приложения, которое моделируется как ряд дискретных страниц, поэтому такие продукты могут плохо согласовываться с подходом Ajax. Впрочем, Studio Creator использует технологию Java ServerFaces (JSF), взаимосвязь которой с Ajax обсуждалась в главе 5, и хотя для обеспечения полного взаимодействия двух указанных технологий необходимо разрешить определенные проблемы, может случиться так, что в ближайшем будущем инструменты, основанные на JSF, обзаведутся лучшей поддержкой Ajax.

Если подножие Ajax располагается в лагере корпоративных разработок, то корни данной технологии уходят в сообщество Web-дизайна, использующее совершенно иной набор инструментов. В данной среде выделяются продукты Macromedia Dreamweaver и Microsoft FrontPage, к тому же поддерживающие стандартные клиентские технологии, которые используются Ajax. Dreamweaver предлагает хорошую поддержку стандартных средств редактирования JavaScript- и CSS-кода (рис. А.3) и двусторонний HTML-редактор с визуальным и текстовым режимом, однако если речь идет о гармоническом WYSIWYG-сочетании сложных пользовательских интерфейсов JavaScript, то приложение поддерживает только собственную библиотеку кода. Интеграция в проект Dreamweaver или FrontPage библиотек других производителей (например, x. Prototype и Rico) потребует кропотливого изучения сценариев, использования возможностей текстового редактора и кое-какой мелкой работы.

В завершение стоит еще упомянуть инструмент Komodo (ActiveState) — поддерживающую несколько языков (Perl, Python, PHP, Tel, JavaScript и XSLT) интегрированную среду разработки для подготовки сценариев. Поддержка Komodo навигации по коду JavaScript просто великолепна; имеется сложный режим структуры, распознающий классы, функции и методы JavaScript (рис. А.4). Поскольку данная среда является универсальной, она поддерживает только общий язык JavaScript, но не его реализации в браузерах. Поэтому инструмент наиболее полезен при разработке моделей предметной области для Ajax. Komodo — это коммерческий продукт с бесплатной пробной версией. Кстати, интересный факт: пользовательский интерфейс Komodo создан с использованием основанного на XML набора инструментов XUL, который применялся для разработки Web-браузера Firefox.

В следующем разделе мы рассмотрим отладчик исходного кода — еще один ключевой инструмент из арсенала разработчика.

А3. *Отладчики*

Поведение простых компьютерных программ часто можно понять, изучая их код. однако большие и более сложные программы часто **слишком** велики, чтобы их можно было охватить сразу С помощью инструментов отладки вы

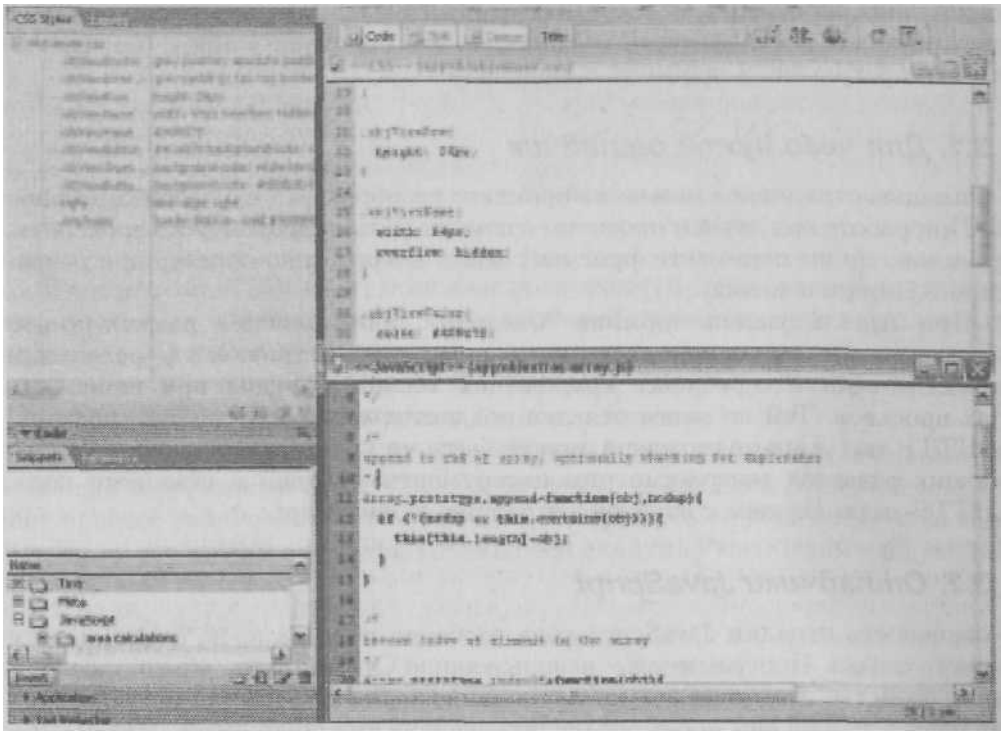


Рис. А.3. Редактор приложения Dreamweaver поддерживает JavaScript и CSS На правой верхней панели расположен редактируемый файл CSS; на левой верней панели тот же документ представлен в режиме структуры

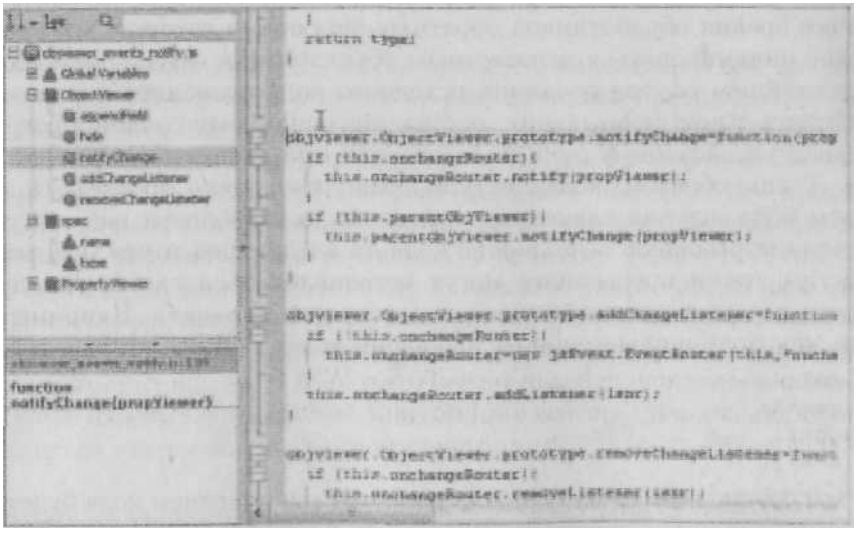


Рис. А.4. Интегрированная среда разработки Komodo предлагает высококачественные средства представления структуры для объектов JavaScript и может понимать множество идиом кодирования На рисунке показано, что данное средство распознало ряд функций, принадлежащих к прототипу O b j e c t V i e w e r

можете контролировать поток выполнения одного фрагмента запущенного кода, позволяя останавливать и запускать его вручную, а также изучая состояние программы в процессе выполнения.

A 3.1 *Для чего нужен отладчик*

С помощью отладчиков можно на практике разобраться, что делает программа. При работе над любым проектом с помощью отладчика можно проверить, правильно ли вы понимаете фрагмент кода, что особенно полезно при разработке Ajax-приложений.

При использовании термина '"отладчик" большинство разработчиков подразумевают отладчики исходного кода, и действительно, отладчики JavaScript-кода и серверных приложений весьма полезны при написании Ajax-проектов. Тем не менее отладке поддается и сетевой трафик, поскольку HTTP-код Ajax-приложений может быть на удивление сложным. В следующих разделах мы рассмотрим инструменты отладки и исходного кода, и HTTP-кода. Начнем с изучения отладчиков JavaScript.

A.3.2. Отладчики JavaScript

Возможность отладки JavaScript-кода особенно полезна из-за изменчивости данного языка. Программисты, использующие C# или Java, обычно узнают, какие свойства и методы доступны на данном объекте, изучая его определение класса, также они могут определить число и тип аргументов метода. При работе же с JavaScript-кодом не всегда можно выяснить, с каким числом аргументов должна вызываться функция и в какие переменные превратятся эти аргументы внутри функции. Последний момент представляет особые проблемы с точки зрения обработчиков обратных вызовов, в которых вызов функции может инициироваться неизвестным объектом или самим браузером.

В простейшем случае отладчик исходного кода позволяет пользователю устанавливать точки прерывания, останавливающие выполнение программы и передавая управление в руки пользователя при выполнении данной строки кода. Таким образом, пользователь может построчно проследить за выполнением кода, изучая значения переменных в их области видимости или возобновляя нормальное выполнение кода до следующей точки прерывания. В JavaScript точки прерывания могут устанавливаться самим отладчиком или программистом, добавляющим в код оператор debugger. Например, рассмотрим, как браузер выполняет следующий код:

```
var x=3;
var y=x*7,-
debugger;
var z=x+y;
```

На третьей строке кода (рис. A.5) контроль над выполнением кода будет передан любому отладчику, зарегистрированному в браузере. В это время можно эудет увидеть значения переменных x и y. Поскольку переменная z к эточ моменту еще не объявлена, ее значение можно будет исследовать только тосле того, как пользователь "перешагнет" через четвертую строку кода.

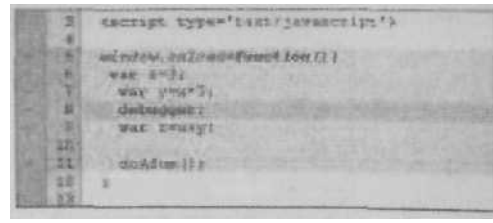


Рис. А.5. Использование оператора debugger (JavaScript) позволяет программно инициировать точку прерывания

В общем, так и выглядят основные возможности средства отладки исходного кода. Как показано ниже, более сложный отладчик может предлагать более богатый набор возможностей.

Навигация по стеку вызовов

В JavaScript при выполнении функции создается новый контекст выполнения, имеющий собственный набор локальных переменных. Если отладчик прерывает процесс выполнения функции, он может видеть ее локальные переменные, но не переменные функции, вызвавшей данную. Рассмотрим следующий пример:

```
function doASum(){
  var a=3;
  var b=4;
  var c=multiply(a-2,b+6);
  return (a+b)/c; }
  function multiply(varl,var2){
  var nl=parseFloat(varl);
  var n2=parseFloat(var2);
  debugger;
  return nl*n2; }
```

В момент остановки отладчиком процесса выполнения мы видим переменные nl, n2, varl и var2. Изучая проблемы нашей программы, мы можем подумать, что они связаны с аргументами, переданными функции. Следовательно, нам нужно узнать, какие значения имеют переменные a и b во внешнем методе doASum(). Мы можем добавить новую точку прерывания в doASum() и снова запустить программу, но в сложном приложении возврат к данному состоянию может потребовать времени. Если отладчик поддерживает навигацию по стеку вызовов, мы можем просто подняться по этому стеку до функции doASumO и исследовать ее состояние точно так же, как если бы мы установили точку прерывания в третьей строке на вызове функции multiply() (рис. А.6). В сложной программе стек вызовов может быть довольно глубоким, поэтому здесь особенно полезна способность отладчика перемещаться вверх-вниз по всем уровням.

Наблюдающие выражения

Некоторые инструменты отладки могут вычислять значения выражений на лету и позволяют пользователю предопределить кодовые выражения, которые будут пересчитываться по мере прохода отладчика по коду. Эти выраже-

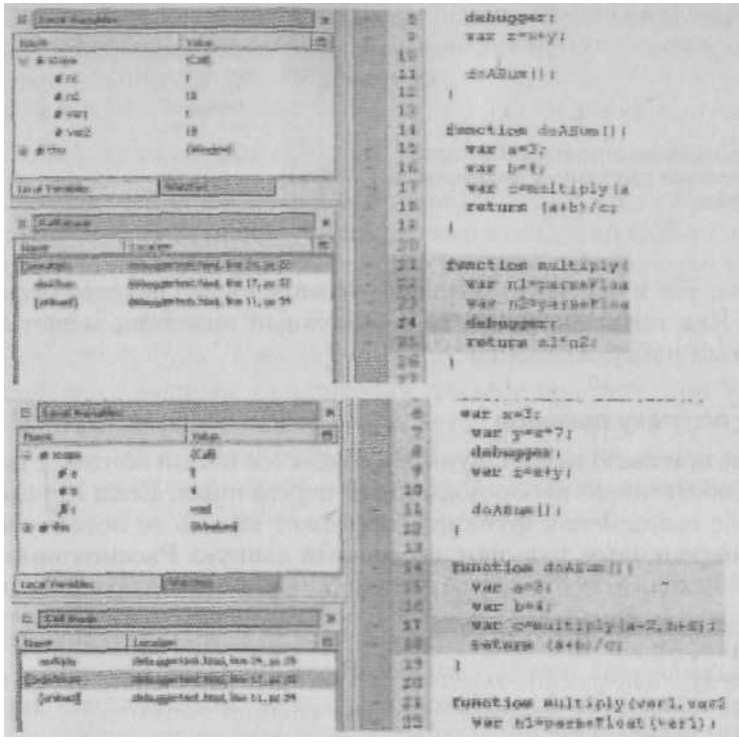


Рис. А.6. Отладчик Mozilla Venkman позволяет исследовать локальные переменные в функциях, находящихся в стеке вызовов выше текущей точки выполнения

гаем, что проблема связана с тем, что на каком-то шаге делитель становится равен нулю, однако не знаем, когда это происходит.

Мы можем установить точку прерывания внутри цикла.

```
for (var i=0;i<100;i++){
  var divisor=i-57;
  debugger;
  var val=42/divisor;
  plotOnGraphfi,val);
}
```

Однако в таком случае нам придется многократно щелкать на кнопке "Продолжить", пока отладчик не доберется до итерации, на которой появляется ошибка. Можно поступить умнее и проверить в коде наличие ошибки.

```
for (var i=0;i<100;i++){
  var divisor=i-57;
  if (divisor^=0){ debugger; }
  var val=42/divisor;
  plotOnGraph(i,val);
}
```

Так мы сразу перейдем на пятьдесят седьмую итерацию цикла — ту, на которой происходит сбой. Данное решение можно назвать условным прерыванием, т.е. поток выполнения программы прерывается только при выполнении определенного условия.

Условия подобного типа можно устанавливать, только модифицируя код. Тем не менее, если присваивать точки прерывания с помощью IDE отладчика, условия на эти точки можно задать независимо от реального кода (рис. А.7). Данная возможность поддерживается некоторыми отладчиками, которые позволяют пользователю привязывать к точке прерывания выражение и останавливать поток, только если значение этого выражения равно `true`.

Изменение значений переменных

При появлении ошибки выполнение программы прекращается. Предположим, что в ходе сеанса отладки мы поняли, как решить данную проблему, но хотим продолжить выполнение программы и проверить следующие фрагменты кода при текущем наборе условий.

Это возможно с некоторыми отладчиками, позволяющими считывать и записывать значения локальных переменных (рис. А.8). Допустим, что в приведенном выше примере мы разобрались, что проблема связана с делением на нуль в цикле, но желаем продолжить выполнение программы, чтобы изучить код, находящийся после цикла. В таком случае мы можем временно присвоить делителю значение 1, разрешив дальнейшее выполнение программы.

Существует множество отладчиков для JavaScript-кода. В число бесплатных продуктов входит модуль Venkman для Mozilla Firefox и Microsoft Script Debugger {подробнее о нем — ниже, в разделе "Ресурсы") для Internet Explorer. Модуль Venkman поддерживает все описанные выше дополнительные функции, в него также встроено средство профилирования, описанное в главе 7. Модуль Microsoft Script Debugger поддерживает навигацию по стеку вызовов

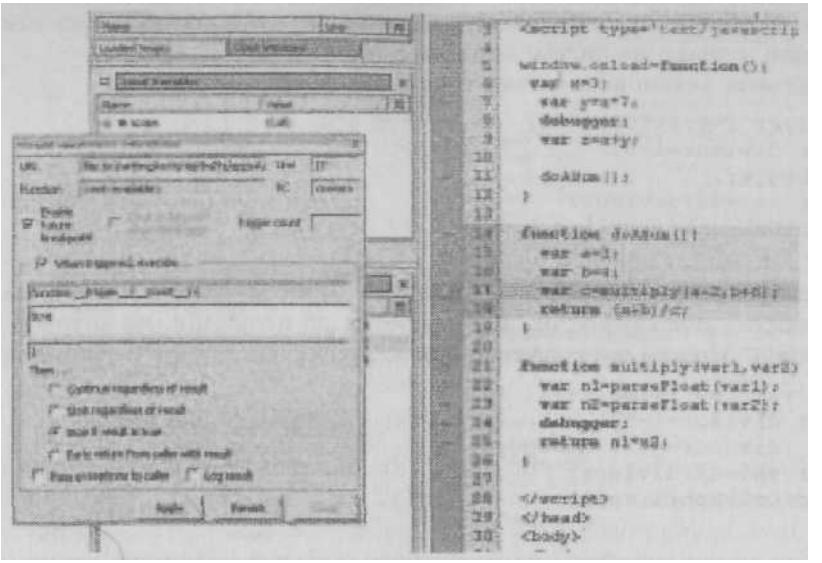


Рис. А.7. Задание условного прерывания в отладчике Mozilla Venkman

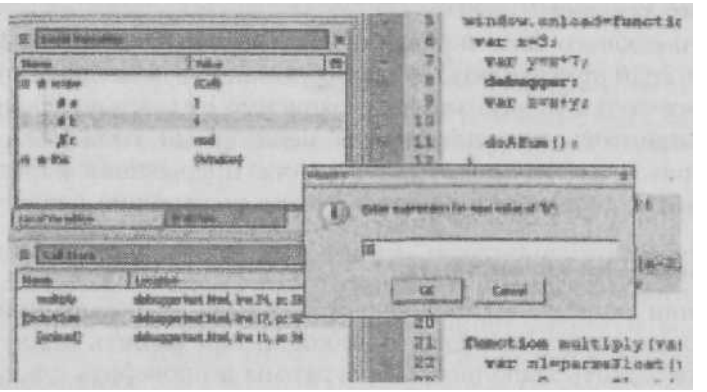


Рис. А.8. Изменение значения переменной в работающей программе с помощью отладчика Mozilla Venkman

и "окно немедленного действия" для выполнения JavaScript-выражений на лету (в частности, организацию запросов и присвоение значений локальным переменным).

Интегрированные среды разработки Visual Studio и Komodo также поддерживают отладчики JavaScript с богатым набором возможностей.

Отладка серверных приложений

Помимо отладки JavaScript-кода на стороне клиента, часто бывает прлезна отладка и серверной части кода. Интегрированные среды разработки приложений Java и .NET обычно имеют высококачественные отладчики. Инстру-

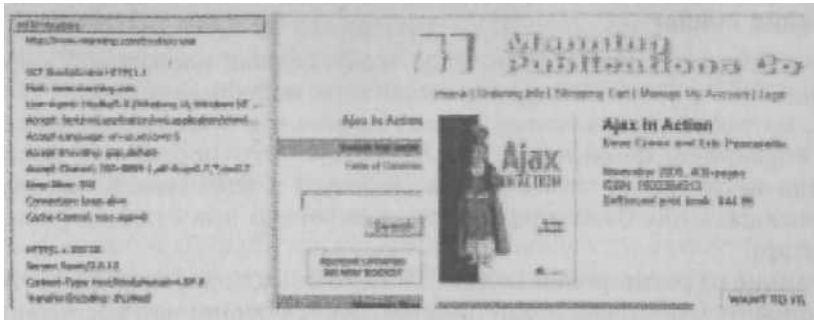


Рис. А.Э. Расширение Mozilla LiveHTTPHeaders может регистрировать HTTP-трафик и представлять подробную информацию о заголовках запросов и ответов

менты Eclipse и Visual Studio (как и большинство других IDE) предлагают прекрасные возможности отладки. Для отладки Web-приложений, основанных на Java, можно использовать сервер приложений JBoss и модули Eclipse, предлагающие простую систему развертывания и отладки Web-приложений. Версии Visual Studio, предназначенные для Web-разработки, имеют встроенный Web-сервер с поддержкой ASP.NET. По нашему мнению, Visual Studio — это единственная среда разработки, поддерживающая отладку клиентского и серверного кода с общим интерфейсом.

Отметим, что часто бывает полезна еще и отладка сетевого трафика. Как и предыдущих случаях, существует множество бесплатных и коммерческих продуктов, которые подходят для этой цели. Рассмотрим их.

A.3.3. Отладчики HTTP

Взаимодействие клиента Ajax и Web-сервера осуществляется с помощью HTTP. Сама по себе реализация этого взаимодействия может оказаться сложной и стать источником ошибок. Иногда стоит перестраховаться и изучить HTTP-трафик, проверив заголовки, строки запроса, содержимое запроса и ответа, а также ход обмена данными.

Расширение Live HTTP Header

Mozilla Firefox поддерживает расширение LiveHTTPHeaders, которое может регистрировать HTTP-трафик, проходящий через браузер (рис. А.9). Заголовки запросов и ответов записываются и отображаются, также их можно экспортировать в виде текстовых файлов, что позволяет хранить информацию о сеансе Ajax. Кроме названных данных, записываются строки запросов из методов GET и POST (однако содержимое ответа не сохраняется).

LiveHTTPHeaders поддерживает только чтение заголовков. Существуют и другие расширения Firefox (например, Modify Headers), позволяющие модифицировать заголовки проходящих пакетов.

Расширение Fiddler

Компания Microsoft Research недавно выпустила приложение Fiddler, основанное на .NET и предлагающее возможности, подобные аналогам LiveHTTP-Headers, но также допускающее запись заголовков в процессе передачи с помощью сценариев, написанных на JavaScript. Это похоже на возможность изменения некоторыми отладчиками значений в ходе сеанса, которую можно использовать для быстрого пропуска дефектов при отладке работающего приложения.

В отличие от расширения LiveHTTPHeaders, которое интегрируется в браузер, Fiddler — это независимый процесс, действующий как посредник между клиентом и сервером. В таком качестве данный продукт **можно** использовать при любой комбинации браузера и Web-сервера.

Приложение Charles

Charles — это условно-бесплатный инструмент, написанный на Java. Подобно Fiddler, он действует как посредник между браузером и сервером. Он может регистрировать данные запроса и ответа (не только заголовки, но и содержимое), экспортировать сеансы в виде файлов электронных таблиц. Кроме того, это приложение предлагает встроенный инструмент формирования полосы пропускания со множеством настроек, который позволяет легко сымитировать очень медленное соединение, когда клиент и сервер развернуты в очень быстрой локальной сети или даже на одном компьютере.

В данной категории существует множество других полезных инструментов, которые мы описывать не будем. Если Charles и Fiddler — это не то, что вам нужно, вы довольно быстро найдете в Интернете информацию о других полезных продуктах, например Ethereal или Apache TCPMon.

На этом мы завершаем обзор готовых инструментов отладки. Объединяя отладчик серверного кода, отладчик JavaScript-кода на стороне клиента и отладчик HTTP между ними, можно перехватить выполнение приложения в любой точке его жизненного цикла и разобраться, что же оно на самом деле делает.

По своей природе отладчики назойливы. Хотя они представляют собой довольно мощные инструменты, иногда предпочтительнее регистрация действий системы в фоновом режиме. Существует множество хороших структур регистрации действий на сервере, например log4j для Java производства Apache, но, повторимся, набор инструментов JavaScript является назойливым. Далее мы рассмотрим простое средство регистрации, написанное на JavaScript, которое можно интегрировать в код браузера и использовать для записи невидимой деятельности.

A.3.4. Создание консоли вывода, встроенной в браузер

Отладчик позволяет разработчику очень подробно рассмотреть работающий код, однако он прерывает естественный поток событий. При отслеживании действий пользователя для тестирования практичности приложения и/или наблюдения за выполнением кода в интегрированном цикле иногда полезнее регистрировать активность приложения, не прерывая поток.

Web-браузер JavaScript не предлагает встроенное средство регистрации. (При первом взгляде на консоль JavaScript в браузере **Mozilla** может показаться, что это не так, но записывать данные в нее могут только браузер и расширения.)

В данном разделе описано, как написать собственную систему регистрации, и продемонстрировано ее использование в одном из примеров приложений. Итак, сформулируем для начала наши требования. Мы не можем вести запись в локальный файл из-за модели безопасности JavaScript, поэтому предпочитаем записывать информацию в консольный элемент, расположенный на экране. Кроме того, нам требуется возможность добавления в консоль сообщений. В идеальном случае при регистрации было бы неплохо использовать HTML-разметку и обычный текст. Кроме того, нам нужно очистить консоль от существующих сообщений.

Не будем все усложнять — передадим элемент DOM в качестве аргумента конструктору объекта. Благодаря этому расположение консоли можно будет определять в зависимости от страницы. Конструктор просто задает двустороннюю связь между элементом DOM и самим объектом консоли.

```
Console=ftmction(el){
  this.el=document.getElementByld(el);
  this.el.className='console' ;
  this.el.consoleModel=this;
  this.clear();
I
```

Чтобы добавить информацию на консоль, мы передаем указанной функции аргумент, который может быть текстовой строкой или элементом DOM, при необходимости передавая также имя класса CSS.

```
Console.prototype.append=function(obj,style){
  var domEl=styling.toDOMElement(obj);
  if (style) {
  domEl.classKame=style;
  }
  this.el.appendChild(domEl);
}
```

Метод toDOMElement () вызывает общую функцию стилевого оформления, которая обеспечивает представление сообщения в виде элемента DOM. Если аргумент уже является элементом DOM, функция ничего не делает, если он является строкой, то последняя заключается в элемент div.

```
styling.toDOMElement=function(obj){
  var result=null;
  if (obj instanceof Element){
    result=obj;
  }else{
    var txtNode=document.createTextNode(String(obj));
      var wrapper=document.createElement('div') ;
      wrapper.appendChild(txtNode);
      result=wrapper;
  }
  return result;
}
```

Чтобы очистить консоль, мы последовательно удаляем все ее дочерние элементы.

```
Console.prototype.clear=function{)f
  while(this.el.firstChild){
    this.el.removeChild{this.el.firstChild);
  }
}
```

Таким образом мы получаем простую реализацию регистрирующей консоли, встроенной в браузер. Посмотрим, как она используется на примере проекта ObjectViewer (см. главы 4 и 5). Прежде всего определим на странице элемент DOM, который будет содержать регистрирующую консоль.

```
<div  id='console'x/div>
```

Далее зададим класс CSS, отвечающий за расположение консоли на экране.

```
div.console {
  position .-absolute;
  top:32px;
  left:600px;
  width:3OOpx;
  height:500px;
  overflow:auto;
  border: lpx solid black;
  background-color:  #eefOff;
}
```

Здесь мы использовали абсолютное позиционирование; в общем случае *можно применять любую* технологию пользовательского интерфейса Ajax. Затем необходимо создать регистрирующий объект. Для удобства в данном примере мы определим его в виде глобальной переменной.

```
var logger=null;
window.onload=function(){
  logger=new Console{"console");
  logger.append{"starting planets app");

}
```

Мы инициализируем регистратор в событии window.onload, чтобы требуемый им элемент DOM гарантированно был создан. Предположим теперь, что мы хотим регистрировать сообщение при создании объектов планет в нашей модели предметной области. Для этого нам нужно вызвать logger .append()•

```
planets.Planet=function
   (id,system,name,distance,diameter,image){
   this.id=id;

   logger.append("created planet object  '"+this.name+"'");
}
```

Подобным образом мы можем добавить в код ObjectViewer (объект ContentLoader) команды регистрации, которые вызываются при изменении зна-

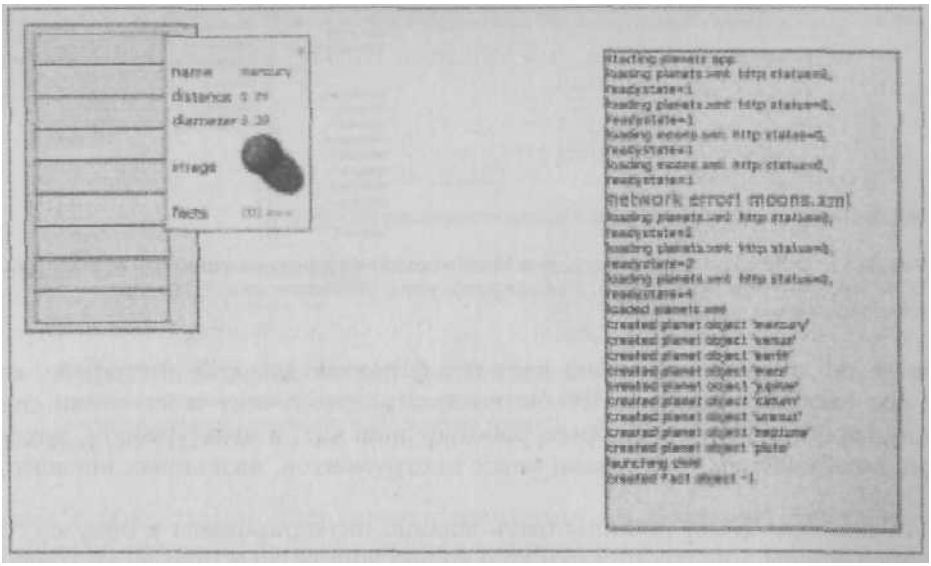


Рис. А.10. Консоль регистрации в действии — отслеживает создание объектов, сетевую активность, редактирование пользователем значений и т.д. Чтобы продемонстрировать отображение сообщений, выделенных специальным стилем, мы добавили запрос к несуществующему серверному ресурсу moons . xml

чений и открытии всплывающих окон, при загрузке сетевых ресурсов и т.д., что позволит отслеживать поведение работающего кода. Важные сообщения, например, информацию о сетевых сбоях, можно выделять особым стилем.

```
net.ContentLoader.prototype.defaultError=function(){
  logger.append("network error! "+this.url, "urgent");
}
```

На рис. АЛО показана регистрирующая консоль как часть модифицированного приложения Object Viewer.

На рисунке видно, насколько простым является введение регистрации в приложение Ajax. Описанная система существенно проще серверной структуры регистрации, подобной Iog4] (Apache). В качестве упражнения читателю предлагается реализовать различные категории растрируемой информации, которые можно включать и отключать независимо.

Рассмотрим теперь следующий тип инструментов — инспекторы DOM.

А4. *Инспекторы POM*

В приложении Ajax пользовательский интерфейс часто модифицируется путем программного изменения DOM. Используя отладчик JavaScript-кода, мы можем пошагово пройти все этапы работы с DOM. проверив, что они дают действительно то, что нам нужно.

Тем не менее DOM — это все же не то изображение, которое видит пользователь. Возможно, мы уверены, что код меняет DOM так, как мы того

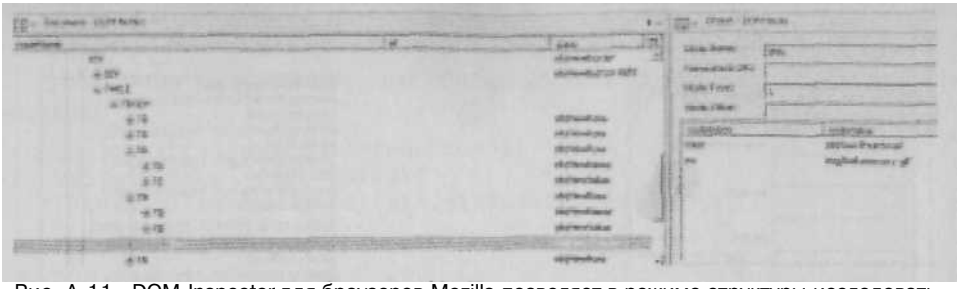


Рис. А.11. DOM Inspector для браузеров Mozilla позволяет в режиме структуры исследовать DOM Web-страницы, в частности, отображаются узлы, объявленные в HTML-коде и сгенерированные программно

желаем, но это не оонзательни даст тот ииль,зиьа,телы;кии интерфейс, который мы ожидаем увидеть Чтобы помочь разработчику в изучении связей между деревом DOM, с которым работает наш код, и интерфейсом, который видит пользователь, был создан класс инструментов, названных инспекторами DOM.

Инспекторы DOM должны быть хорошо интегрированы в браузер, причем инспекторы конструируются только под конкретные браузеры. Наиболее популярными являются инструменты, созданные для браузера Mozilla Firefox, поэтому вначале мы рассмотрим их. а затем альтернативы, доступные для браузера Internet Explorer.

А.4.1. Использование DOM Inspector для браузеров Mozilla

Инструмент DOM Inspector поставляется вместе с Firefox, но активизируется только при выборе соответствующей опции в процессе установки браузера. Если вы его установили, в меню Tools (Инструменты) появится опция DOM Inspector. При первом запуске DOM Inspector состоит из двух расположенных рядом панелей (рис. А.11)- Левая представляет собой древовидный элемент управления, вначале показывающий только документ и единственный узел HTML. Узел можно раскрыть, чтобы выявить заголовок и тело документа, а затем открыть в теле набор узлов, представляющих HTML-разметку страницы, а также все элементы, созданные программными средствами. Если узлам был присвоен идентификатор или атрибут класса CSS, соответствующая информация появится в дополнительных столбцах древовидного элемента.

Данный древовидный элемент управления синхронизирован со страницей, отображаемой в основном окне браузера. Если выбрать узел дерева с помощью мыши, соответствующий элемент в разделе структуры страницы будет выделен красной рамкой (данная связь является двусторонней). Вызывая команду Search => Select Element из меню Click DOM Inspector, пользователь может щелкнуть в окне Web-браузера и выделить элемент дерева, соответствующий отмеченному элементу. (То же можно сделать с помощью кнопки панели инструментов.)

На правой панели информация о текущем узле указывается в одном из нескольких доступных форматов, включая формат отображения узлов DOM, стилевых правил CSS, а также объектов JavaScript (рис. А 12). В последнем

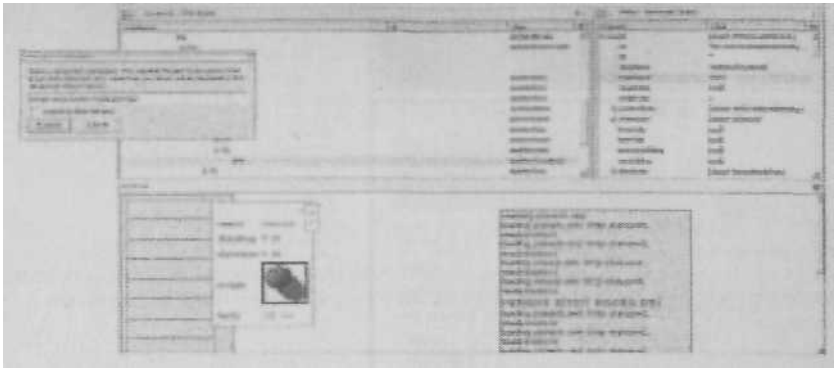


Рис. А.12. DOM Inspector (Mozilla) позволяет связывать с элементами DOM сценарии Переменная t a r g e t определяет узел DOM, выбранный в текущий момент; в данном случае это изображение планеты, границу которой мы хотим изменить

режиме с объектом можно связать программный сценарий, щелкнув правой кнопкой мыши на правой панели и выбрав кнопку Evaluate JavaScript. В качестве цели сценария (target) можно указать выбранный в текущий момент элемент DOM, поэтому, например, чтобы окружить этот элемент тонкой синей рамкой, можно ввести в поле команду `target.style.border='4px solid blue'`.

DOM Inspector также содержит третью панель, расположенную ниже двух описанных. В данной панели можно визуализировать видимое содержимое документа (рис. А. 13). Данная панель появится, если пользователь наберет в поле URL адрес страницы и щелкнет на кнопке inspect, после чего можно будет изучать расположенные рядом абстрактную DOM и видимый документ.

А.4.2. Инспекторы DOM для браузера Internet Explorer

Самой большой проблемой наборов инструментов для браузеров Mozilla является то, что инспекторы DOM не могут использоваться для выявления проблем в Internet Explorer. Впрочем, существует несколько продуктов, предлагающих подобные возможности и для этого браузера. Многие из них являются коммерческими или условно бесплатными, хотя имеется и несколько рабочих бесплатных утилит, например IEDocMon (подробнее о ней речь пойдет ниже — в разделе "Ресурсы[1]").

Подобно Mozilla DOM Inspector, панель инструментов IEDocMon предлагает простое двухпанельное представление DOM, с деревом слева и подробным представлением узлов справа (рис. А.13).

На этом мы завершаем обсуждение типов инструментов, используемых для разработки Ajax-проектов. Богатым поставщиком инструментов для работы с Ajax является сообщество, сформировавшееся в среде разработчиков расширений для браузера Firefox. В следующем разделе мы кратко опишем, как находить и устанавливать расширения Firefox.

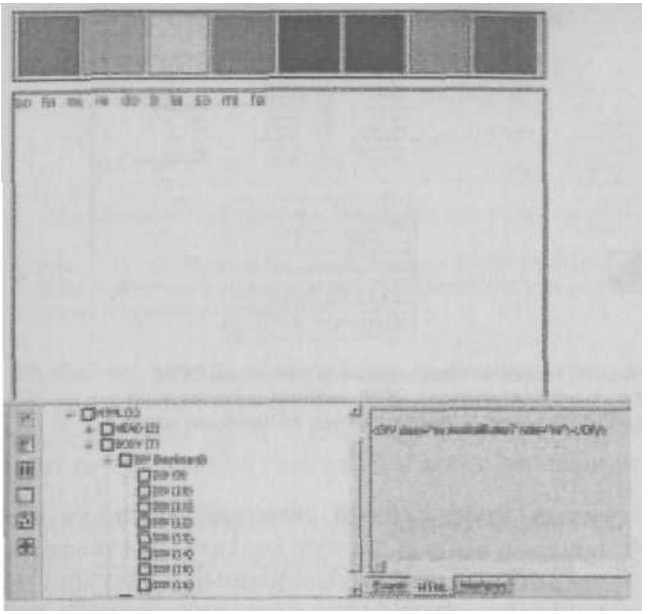


Рис. А.13. Панель инструментов IEDocMon для Internet Explorer предлагает возможности, подобные DOM Inspector (Firefox) и позволяет быстро решать вопросы визуализации с помощью программных пользовательских интерфейсов

14.3. Средство Safari DOM Inspector для Mac OS X

>раузер Safari для Mac OS X содержит встроенный инспектор DOM, который ызывается через меню отладки. По умолчанию данное меню не активизи-'Овано. Чтобы активизировать его, запустите приложение Terminal и введи-е строку defaults write com.apple.Safari IncludeDebugMenu 1. В зави-имости от ваших прав вам, возможно, придется использовать команду sudo. Госле ее выполнения перезапустите браузер Safari, и меню отладки должно удет появиться.

1.5. *Установив расширений Firefox*

1ы уже рассматривали два очень полезных расширения Firefox: отладчик ^enkman и сетевой отладчик LiveHTTPHeaders. Вообще, существует множе-гво расширений для Firefox. некоторые из которых предназначены для Web-азработчиков В данном разделе мы кратко рассмотрим процесс установки годуля Firefox, используя в качестве примера расширение Modify Headers.

Расширения Firefox устанавливаются из самого браузера. Прежде все-о необходимо найти страницу загрузки расширения; в данном случае это ttps://addons .mozilla.org/extensions. На рис. А.14 показано, как со границы загрузки расширения Modify Headers можно перейти на основную границу сайта Mozilla Update.

В данном случае требуемая гиперссылка представляет собой большую нопку Install Now. После щелчка откроется диалоговое окно с предупре-ждением об опасности установки неподписанных расширений (рис А.15).

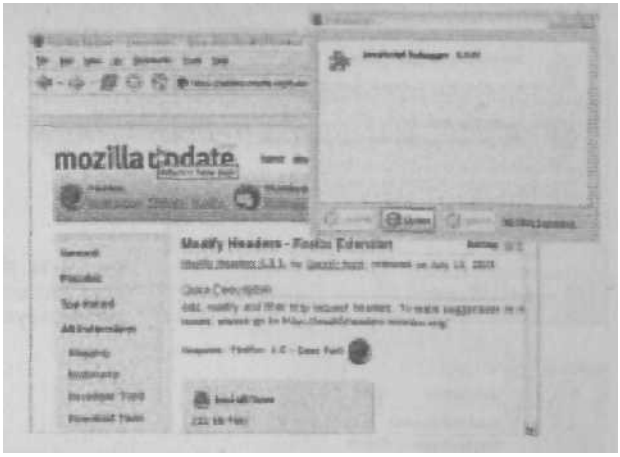


Рис. А.14. Все установленные расширения в браузере Firefox отображаются во всплывак диалоговом окне

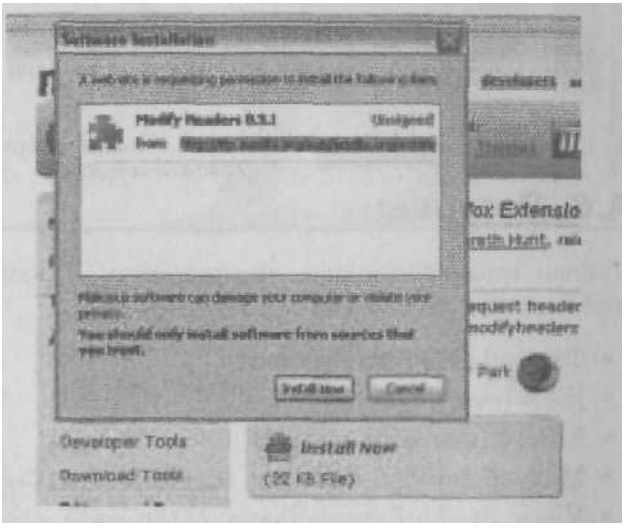


Рис. А.15. Расширения Firefox можно устанавливать из Интернета, используя специальный загружаемый архивный формат

В отличие от обычного JavaScript-кода расширения имеют полный доступ к локальной файловой системе. Кроме того, следует сказать о подписывании расширений. Теоретически это должно защитить автора от подделок, однако на практике подписываются не все расширения. После установки расширение регистрируется во всплывающем диалоговом окне Extensions (рис. А.16).

Нам осталось только закрыть все открытые окна Firefox (включая инспекторы DOM, отладчики и т.п.) и перезапустить Firefox. Теперь расширение готово к работе и отображается в меню Tools (рис. А.17).

Браузер Firefox поддерживает множество расширений, многие из которых предназначены для Web-разработчиков. Не все расширения можно найти на сайте addons.mozilla.org, но начинать поиск нужного инструмента стоит именно с этого сайта. Процедура установки расширения (в том числе рассмотренного ранее отладчика Venkman) обычно похожа на описанную выше.

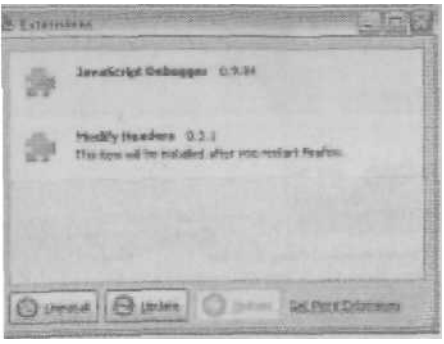


Рис. А.16. Расширения появляются в диалоговом окне Extensions сразу после установки но активизируются только после перезапуска браузера

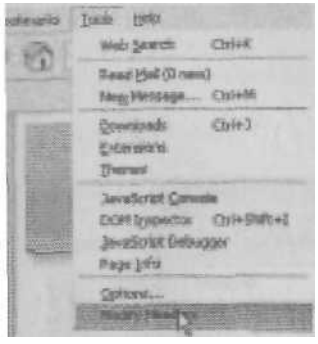
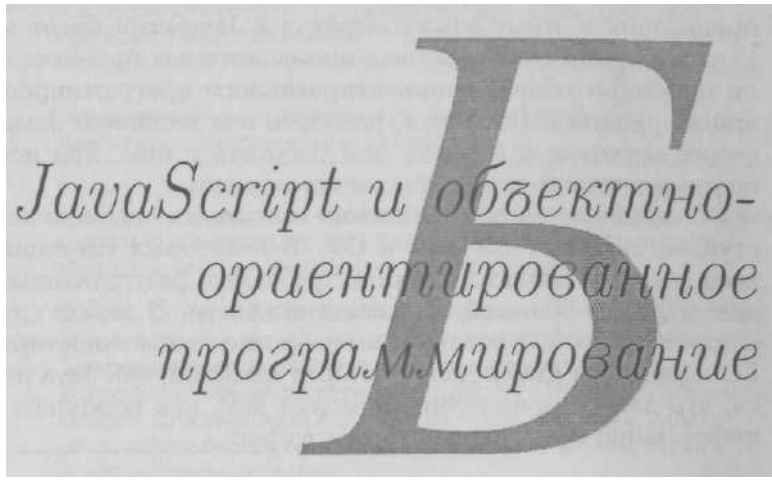


Рис. А.17. Расширение будет готово к использованию после перезапуска браузера

1.6. *Ресурсы*

Данная глава посвящена инструментам. Продукты, упоминавшиеся в ней, ожно найти на следующих Web-сайтах:

- Textpad (www.textpad.com);
- jEdit (www.jedit.org);
- Eclipse (www.eclipse.org);
- Модули JavaScript Eclipse (http://jseditor.sourceforge.net/);
- Visual Studio Express (http://lab.msdn.microsoft.com/express/);
- Dreamweaver (www.macromedia.com/software/dreamweaver/);
- Komodo (www.activestate.com/Products/Komodo/);
- Venkman Debugger (www.mozilla.org/projects/venkman/);
- Отладчик Microsoft Script (www. microsoft, com/downloads/details . aspx? Famil_yiD=2f4 65be0-94fd-45 69-b3c4-dffdfl9ccd99&displaylang=en);
- Charles (www.xk72 .com/charles/);
- Fiddler (www.fiddlertool.com);
- LiveHttpHeaders (http://livehttpheaders.mozdev.org/);
- Расширение Modify Headers (https://addons, mozilla.org/extensions/ moreinfo.php?id=967&vid=4243),
- Инспектор DOM для Internet Explorer IEDocMon (www.cheztabor.com/ IEDocMon/index. htm).


Б
JavaScript и объектно-
ориентированное
программирование

К использованию языка JavaScript приходят различными путями. Одним адо реализовать дополнительные элементы графического интерфейса, дру-им — написать сложные программы.

Данное приложение — не руководство по JavaScript; существуют многие орошие статьи и книги: которые помогут вам освоить данный язык. Мы ишь излагаем некоторые основные понятия, которые помогут специалистам, ладеющим Java и C#, в написании программ на JavaScript. (Сказанное отно-ится и к программистам, использующим C++. Однако необходимо заметить, то C++ унаследовал от C излишнюю гибкость, поэтому для специалистов, ривыкших к этому языку, переход к JavaScript будет менее болезненным.) lсли вы принимали участие в промышленных проектах и полностью овладе-и приемами объектно-ориентированного программирования, то на первых гапах работы с JavaScript, находясь под влиянием Java или C#, вы будете корее бороться с языком, чем работать с ним. Мы испытали это на себе, оэтому считаем своим долгом помочь вам.

С помощью JavaScript можно выполнять специфические действия, недо-гупные посредством Java и C#. В некоторых ситуациях особенности дан-ого языка позволят получить требуемые результаты более простым путем ли добиться большей производительности. В любом случае решение о том, тедует ли воспользоваться имеющимися возможностями, остается за вами. ели ранее вы имели дело с такими языками, как Java или C++, мы надеем-I, что данное приложение поможет вам, как несколько лет назад подобная ^формация помогла некоторым из нас.

11. JavaScript — это не Java

акую нагрузку несет на себе имя? В случае Java и JavaScript оно направлено а обеспечение рыночного успеха и мало отражает суть. Язык JavaScript по-учил это имя буквально в последнюю минуту перед тем, как быть представ-знным широкой публике. Сотрудники отдела маркетинга компании Netscape эедложили его вместо планировавшегося ранее имени livescript. Вопреки южившемуся мнению, JavaScript не принадлежит к семейству языков C. н больше похож на такие функциональные языки, как Scheme и Self; его 1кже можно сравнить с языком Python. Однако в имени JavaScript при-тствует слово Java, и синтаксические правила для этих языков в чем-то зхожи. В некоторых случаях JavaScript ведет себя подобно Java, а в других гучаях демонстрирует совершенно противоположное поведение.

Эти различия позволяют использовать данные языки разными спосо-1ми. а в некоторых случаях дают возможность проделывать в JavaScript-юграммах такие головокружительные трюки, которым позавидует и умуд-^нный опытом сторонник Lisp.

Если вы опытный программист, то можете воспользоваться предостав-гемыми возможностями и получить блестящие результаты, написав лишь **сколько** сотен строк исходного текста. Если же вы только считаете себя [ытным, а на самом деле ваша квалификация далека от совершенства, вы •гсгро "завязнете!" в созданном вами коде.

Таблица Б.1. Основные различия между Java и JavaScript

Характеристика	Описание
Переменные не типизированы	Переменные объявляются как переменные, а не как целые числа, строки или экземпляры определенных классов. В JavaScript можно присваивать одной и той же переменной значения различных типов
Код интерпретируется динамически	Код программы представляет собой исходный текст. В процессе ее выполнения интерпретатор читает команды и выполняет соответствующие действия. Этим JavaScript принципиально отличается от таких языков, как Java, C и C#. Исходный код Ajax-приложения доступен пользователю. Более того, в процессе выполнения программы код может быть сгенерирован динамически, при этом нет необходимости прибегать к услугам компилятора
Функции JavaScript представляют собой независимые элементы	В языке Java методы связаны с содержащими их объектами и не существуют за их пределами. В JavaScript функцию можно связать с объектом, но при этом ее можно вызвать в другом контексте или присоединить к другому объекту
Объекты JavaScript базируются на прототипах	В Java, C++ или C# типы объектов определяются посредством суперклассов, виртуальных суперклассов или интерфейсов. Тип определяет функциональные возможности объекта. В JavaScript объект по сути представляет собой ассоциативный массив. Для эмуляции в JavaScript типов, подобных типам Java, используются прототипы, но этим достигается лишь поверхностное сходство

Весь наш предыдущий опыт подтверждает, что в работе следует стремиться к простым решениям. Если вы входите в состав рабочей группы, то стандарты кодирования способствуют таким решениям (конечно, в том случае, когда руководство не возражает против их использования).

Представлять себе различия между языками необходимо также по следующей причине: некоторые из особенностей реализуются браузером, поэтому, понимая происходящее, можно сэкономить много времени и усилий, отлаживая приложение. Зная различия между JavaScript и Java, становится понятно, что сходство между этими языками лишь кажущееся.

Итак, читайте данное приложение и сами составьте представление о том, как в действительности выглядят объекты JavaScript, как они формируются и какие возможности предоставляют разработчику функции JavaScript.

Б.2. Объекты в JavaScript

Создавая программу на JavaScript, не обязательно использовать объекты и даже функции. Можно написать программу в виде набора команд, которые будут выполняться в процессе чтения их интерпретатором. Однако по мере усложнения программы обходиться без функций и объектов становится все труднее; не используя их, крайне сложно организовать должным образом свой код.

Создать объект JavaScript проще всего, вызвав встроенный конструктор класса Object.

```
var myObject=new Object();
```

В разделе Б.2.2 мы рассмотрим другие подходы и поговорим о том, что >значает ключевое слово new. Первоначально объект myObject — "пустой", :.е. он не содержит свойств и методов. Добавить их несложно; рассмотрим, сак это можно сделать.

5.2.1 *Формирование объектов*

<ак было сказано ранее, объект JavaScript представляет собой ассоциатив-lый массив, содержащий поля и имена. Ключевыми значениями для доступа i элементам массива являются имена. Синтаксис, подобный С, является свое-)бразной "надстройкой" над внутренней реализацией объекта. Существуют г альтернативные средства доступа к свойствам и функциям. **Сложный** объ-•кг можно создавать шаг за шагом, добавляя по мере необходимости новые **переменные** и функции.

Существуют два способа формирования объекта. Первый из них — это >бычное использование JavaScript. Второй способ предполагает применение пециальной системы обозначений, называемой JSON. Начнем рассмотрение подхода, использующего средства JavaScript.

Трименение выражений JavaScript

Три выполнении фрагмента кода может возникнуть необходимость присво-[ть свойству объекта некоторое значение. Свойства объектов JavaScript до-текают чтение и запись, поэтому для присваивания значения можно исполь-овать обычный оператор =. Добавим свойство к созданному нами простому 'бъекту.

```
myObject.shoeSize="12";
```

В обычном объектном языке нам необходимо определить класс и объявить нем свойство shoesize. в противном случае на этапе компиляции будет генерировано сообщение об ошибке. В JavaScript все обстоит по-другому. Лы можем ссылаться на свойства, используя выражения, предназначенные ,ля работы с массивом. В данном случае поступим так лишь для того, чтобы одчеркнуть внутреннюю реализацию объекта.

```
myObject['shoeSize']="12";
```

Такая запись плохо подходит для повседневного использования, одна-о в некоторых случаях она может обеспечить определенные преимущества. > частности, работая с объектом как с массивом, мы можем реализовать тражение на этапе выполнения программы. Подробнее этот вопрос мы рас-мотрим в разделе Б.2.4.

Мы также можем динамически добавить новую функцию к объекту.

```
myObject.speakYourShoeSize=function(){
  alert{"shoe size : "+this.shoeSize);
}
```

Кроме того, существует возможность использовать в своих целях уже существующую функцию.

```
function sayHello(){
  alert ('hello, my shoeSize is '-f-this. shoeSize);
1

myObject.sayHello=sayHello;
```

Заметьте, что, связывая с объектом определенную ранее функцию, мы не ставим круглые скобки. Если мы выполним присваивание так, как показано ниже, то выполним функцию sayHello() и присвоим свойству myObject значение, возвращаемое ею, в данном случае — null.

```
myObject.sayHello=sayHello();
```

Для того чтобы создать сложную модель данных, можно присваивать объекты другим объектам.

```
var myLibrary=new Object();
myLibrary.books=new Arrayf);
myLibrary.books[0]=new Object();
myLibrary.bookstO].title="Turnip Cultivation through the Ages";
myLibrary.books[0].authors=new Array();
var jim=new ObjectO;
jim.name="Jim Brown";
jim.age=9;
myLibrary.books[0].authors[0]=jim;
```

Такая рутинная работа быстро утомляет программиста. К счастью, в JavaScript предусмотрена система записи, которая называется JSON и позволяет быстро формировать графы объектов. Рассмотрим ее более подробно.

Использование JSON

JavaScript Object Notation (JSON) — одно из базовых средств данного языка, позволяющее быстро создавать массивы и графы объектов. Для того чтобы лучше понять JSON, нам надо разобраться в работе массивов JavaScript, поэтому рассмотрим данный вопрос подробнее.

В JavaScript предусмотрен встроенный класс Array. Его экземпляр можно создать с помощью ключевого слова new.

myLibrary.books=new Array();

При присвоении значений на элементы массива можно ссылаться с помощью числовых индексов, как в языках C или Java.

```
myLibrary.books[4]=somePredefinedBook;
```

Элемент массива можно связать с ключевым значением, как в Java Map-Python Dictionary и. конечно же, в любом объекте JavaScript.

```
myLibrary.books["Bestseller"]=somePredefinedBook;
```

Для некоторых задач такой синтаксис вполне подходит, но создание подобным способом объектов или массивов значительного объема требует неоправданно больших затрат времени. Для массивов с числовыми индекса-

ми существует сокращенная запись: в квадратных скобках указывается список значений, разделенных запятыми.

```
myLibrary.books=[predefinedBookl,predefinedBook2,predefinedBook3]  ;
```

При формировании объекта JavaScript мы используем фигурные скобки; каждый элемент массива записывается как пара *ключ:значение.*

```
myLibrary.books={
  bestseller : predefinedBookl,
  cookbook : predefinedBook2,
  spaceFiller : predefinedBook3
};
```

При любой записи лишние пробелы игнорируются, что позволяет выравнивать идентификаторы так, чтобы упростить восприятие программы. Ключевое значение может содержать пробелы. В этом случае, согласно правилам JSON, оно помещается в кавычки. Например:

```
"Best Seller" : predefinedBookl,
```

JSON-записи допускают вложенность. Это позволяет создавать сложные иерархии объектов (ограничением в данном случае является лишь максимально допустимая длина строки).

```
var myLibrary={
  location : "my house",
  keywords : [ "root vegetables", "turnip", "tedium" ],
    books: [
    {
      title : "Turnip Cultivation through the Ages",
      authors : [
       { name: "Jim Brown", age: 9 },
       { name: "Dick Turnip", age: 312 }
      h
      publicationDate : "long ago"
    Ь
    {
      title : "Turnip Cultivation through the Ages, vol. 2",
      authors : [
        { name: "Jim Brown", age: 35 }
      ],
      publicationDate : new Date(1605,11,05)
    }
  ]
);
```

В данном случае для объекта myLibrary заданы три свойства: location представляет собой обычную строку; keywords — это список строк с числовыми индексами; books — список объектов также с числовыми индексами. При этом каждый объект, содержащийся в books, содержит заголовок (строку), дату публикации (в одном случае это JavaScript-объект Date, а в другом — строка) и список авторов (массив). Каждый автор представлен параметрами name и age. JSON предоставляет механизм для формирования требуемой информации за один проход. В других случаях для этого потребовалось бы несколько строк кода (и большая пропускная способность).

Внимательные читатели, вероятно, заметили, что для второй книги мы

задаем дату публикации с помощью JavaScript-объекта Date. При присвоении значения можно использовать любой код JavaScript, даже определенную нами функцию.

```
function gunpowderPlot(){
  return new Date(1605,11,05);
}
var volNum=2;
var turnipVol2={
  title : "Turnip Cultivation through the Ages, vol. " +volNum,
  authors : [
  { name: "Jim Brown", age: 35 }
    h
    publicationDate : gunpowderPlot()
  }
  I
};
```

Здесь название книги вычисляется динамически с помощью включенного выражения, а свойству publicationDate присваивается значение, возвращаемое функцией.

В предыдущем примере мы определили функцию gunpowderPlot(), которая выполнялась при создании объекта. Мы также можем определить функцию-член для объекта, создаваемого средствами JSON, так, чтобы она выполнялась при непосредственном обращении.

```
var turnipVol2={
  title : "Turnip Cultivation through the Ages, vol. "+volNum,
  authors : [
    { name: "Jim Brown", age: 35 }
      L
    publicationDate : gunpowderPlot()
    }
  ],
  summarize:function(len)(
    if (!len){ len=7; }
    var summary=this.title+" by " +this.authors[0].name
      +" and his cronies is very boring. Z";
    for (var i=0;i<len;i++){
      summary+="z";
    }
    alert(summary);
  }
};
turnipVol2.summarize{6);
```

Функция summarize*) обладает всеми характеристиками стандартной функции JavaScript; для нее определены параметры и объект контекста, идентифицируемый с помощью ключевого слова this. Созданный объект ничем не отличается от любого другого JavaScript-объекта, поэтому по мере необходимости мы можем использовать для *его* формирования и запись JSON, и обычные выражения JavaScript. В частности, средствами JavaScript можно изменить объект, созданный с использованием JSON.

```
var numbers={ one:1, two:2, three:3

numbers.five=5;
```

Мы сначала определили объект, применив запись JSON, а затем добавили к нему свойство, используя обычное выражение JavaScript. Точно так же мы можем расширить посредством JSON объект, созданный с помощью выражений JavaScript.

```
var cookbook=new Object();
cookbook.pageCount=321;
cookbook.author={
  firstName: "Harry",
  secondName: "Christmas",
  birthdate: new Date(1900,2,29),
  interests:  ["cheese","whistling",
    "history of lighthouse keeping"]
};
```

Используя только встроенные JavaScript-классы Object и Array и средства JSON, можно сформировать сколь угодно сложную иерархию объектов. JavaScript также предоставляет средства, позволяющие выполнять действия, тодобные определению классов. Рассмотрим их и выясним, чем они могут томочь в работе над приложениями.

S.2.2. Функции-конструкторы, классы и прототипы

3 объектно-ориентированном программировании при создании объекта обычю указывается класс, экземпляром которого он является. И в Java, *i* в JavaScript присутствует ключевое слово new, позволяющее создавать объ-:кты требуемых типов. Этим данные языки похожи друг на друга.

В языке Java нет ничего, кроме объектов (исключением являются простые "ипы), причем каждый объект является потомком класса java.lang.Object. Тоддержка классов, полей и методов реализована в виртуальной машине lava. Когда мы записываем приведенное ниже выражение, то сначала задаем ^ип переменной, а затем создаем экземпляр класса, используя конструктор.

MyObject myObj=new MyObject{argl,arg2);

Данное выражение корректно только в том случае, если определен класс lyObject и в нем предусмотрен соответствующий конструктор.

В JavaScript также используются объекты и классы, но понятие наследования в этом языке отсутствует. Каждый объект JavaScript — это экземпlяр одного и того же базового класса, который позволяет формировать поля i функции в процессе выполнения программы. Пример определения свойства фи веден ниже.

```
MyJavaScriptObject.completelyNewProperty="something";
```

Для организации .объектов в таких условиях могут быть использованы [рототипы, с помощью которых объявляются свойства и функции. Эти свойтва и функции автоматически присоединяются к объекту при его создании. Ложно написать объектную программу на JavaScript, и не применяя прото-

типов, однако прототипы позволяют упорядочить код и привести его в соответствие с объектным подходом. Вряд ли нужно доказывать, что объектный подход стоит применять при работе над сложным приложением, в составе которого реализуется богатый клиент.

Мы можем написать на JavaScript нечто, напоминающее Java-определение.

```
var myObj=new MyObject();
```

Однако вместо класса MyObject возможно определить лишь функцию с таким именем. Ниже приведен пример простого конструктора.

```
function MyObject(name,size){
  this.name=naine;
  this.size=size;
}
```

Вызвать ее можно следующим образом:

```
var myObj=new MyObject("tiddles","7.5 meters");
alert("size of "+myObj.name+" is "+myObj.size);
```

Все свойства this, установленные в конструкторе, впоследствии доступны как свойства объекта. При необходимости можно включить в состав конструктора и вызов alert(), в результате tiddles будет сообщать свой размер. Часто в составе конструкторов встречаются определения функций.

```
function MyObject{name,size){
  this.name=name;
  this.size=size;
  this.tellSize=function(){
    alert("size of "+this.name+" is "+this.size);
  }
}
var myObj=new Object("tiddles","7.5 meters");
myObj.tellSizet);
```

Такое решение допустимо, но оно далеко от идеала по двум причинам. Во-первых, при формировании каждого экземпляра MyObject создается новая функция. Мы, как и все грамотные программисты, следим за использованием памяти, и если нам надо много таких объектов, то данный подход становится для нас неприемлемым. Во-вторых, мы непредумышленно создали то, что принято называть *замыканием* (closure). В данном случае оно вполне безобидно, но как только мы включим в конструктор выражения, работающие с узлами DOM, оно может стать причиной серьезных проблем. С замыканиями вы ознакомитесь далее в этом приложении. А сейчас рассмотрим альтернативный подход, основанный на использовании *прототипов*.

Прототип — это свойство JavaScript-объектов, аналогов которому в обычных объектных языках нет. С прототипом конструктора связываются функции и свойства. После этого прототип и ключевое слово new начинают работать совместно, и при вызове функции посредством new к объекту присоединяются все свойства и метода прототипа. Это звучит несколько непривычно, но использовать прототип на практике достаточно просто.

```
function MyObject{name,size){
  this.name=name;
  this.size=size;
```

```
>
MyObject.prototype.tellSize=function(){
  alert("size of "+this.name+" is "+this.size);
J
var myObj=new MyObject{"tiddles","7.5 meters");
myObj.tellSize();
```

Сначала мы объявляем конструктор, а затем добавляем функции к прототипу. Функции присоединяются к объекту при его создании. Посредством :лючевого слова this в процессе выполнения программы можно ссылаться га экземпляр объекта.

Обратите внимание на порядок действий. Обращаться к прототипу мы южем только после того, как функция конструктора уже объявлена. Объекы же могут наследовать от прототипа только те свойства и методы, которые же были добавлены к нему на момент вызова конструктора. Между вызоами конструктора допустимо изменять прототип. Добавлять к прототипу южно не только функции, но и другие элементы.

```
MyObject.prototype.color="red";
var objl=new MyObject();
MyObject.prototype.color="blue";
MyObject.prototype.soundEffect="boOOOoing!!";
var obj2=new MyObject{);
```

В данном примере для obj 1 задается красный цвет, а звуковые эффекты тсутствуют. Для ob j 2 цвет уже синий и, кроме того, с нш Cl связан довольно неприятный звук. Изменять прототип во время работы программы вряд и имеет смысл. Необходимо знать, что подобное возможно, но в своих программах лучше применять прототипы для того, чтобы приблизить поведение avaScript-объектов к обычным классам.

Заметьте, что прототипы *встроенных классов,* т.е. классов реализуемых раузером и доступных посредством JavaScript, также можно расширять встроенные классы принято также называть *рабочими объектами* (host bject)). Рассмотрим, как можно сделать это на практике.

к2.3. Расширение встроенных классов

Разработчики JavaScript планировали, что сценарии на данном языке будут ыполняться под управлением программ, предоставляющих доступ к собгвенным объектам, написанным, например, на C++ или Java. Эти объекты азывают *встроенными* или *рабочими.* Они несколько отличаются от объеков, определяемых разработчиками, которые мы рассматривали выше. Одна- о механизм прототипов дает возможность работать и со встроенными класами. Браузер Internet Explorer не позволяет расширять DOM-узлы; другие <е основные классы доступны для расширения во всех основных браузерах, 'ассмотрим в качестве примера класс Array и определим несколько вспомоательных функций.

```
Array.prototype.indexOf=function(obj){
  var result=-l;
  for (var i=O;i<this.length;i++){
    if (this[i]==obj){
```

```
        result=i;
        break;
      }
    }
    return result;
}
```

Мы определили для объекта Array дополнительную функцию, которая возвращает числовой индекс объекта в данном массиве или значение -1, если этот объект отсутствует. Далее мы можем написать метод, проверяющий, содержит ли массив указанный объект.

```
Array.prototype.contains=function{obj){
  return  (this.indexOf(obj)>=0);
}
```

Ниже приведен пример функции, включающей в массив новые элементы, предварительно убедившись в том, что данный элемент отсутствует.

```
Array.prototype.append=function(obj,nodup){
  if (!(nodup && this.contains(obj)))f
    this[this.length]=obj;
  }
}
```

После того как новые функции определены, они будут присутствовать в каждом новом объекте Array, независимо от того, применялся ли для его создания оператор new или выражение JSON.

```
var  numbers=[1,2,3,4,5];
var  got8=numbers.contains(8);
numbers.append("cheese",true);
```

Как и для объектов, определяемых разработчиком, прототипы встроенных функций можно изменять в процессе работы программы, в результате объекты, созданные в разное время, будут иметь различные наборы функций и свойств. Несмотря на наличие подобной возможности, мы рекомендуем модифицировать прототип только один раз в начале выполнения программы. Поступая так, вы избежите недоразумений; в особенности это важно в том случае, если в написании программы берет участие рабочая группа.

Прототипы могут оказаться очень полезными при разработке модели для клиентской части Ajax-приложения. Не исключено, что в этом случае потребуется не только определить объекты различных типов, но и использовать наследование. В отличие от C++, Java и C#, язык JavaScript непосредственно не обеспечивает наследование, но его можно реализовать посредством прототипов. Рассмотрим этот вопрос подробнее.

Б.2.4. Наследование прототипов

Объектный подход предполагает не только определение классов, но и поддержку иерархии наследования. В качестве классического примера часто приводят объект Shape, в котором определены методы для вычисления периметра и площади геометрической фигуры. На его основе формируются конкретные реализации для прямоугольников, треугольников и кругов.

С наследованием тесно связано понятие области видимости. Область видимости метода или свойства определяет, кто может пользоваться ими. Обычно в языках программирования **предусмотрены** области видимости public, private и protected.

Область видимости и наследование удобно применять при реализации модели предметной области. К сожалению, базовые средства JavaScript не обеспечивают их поддержку. Несмотря на это, разработчики пытаются реализовать элегантные решения имеющимися в наличии средствами, и эти попытки часто приводят к успеху.

Дуг Крокфорд (ссылка на его работу приведена в конце данного приложения) предложил оригинальное решение, позволяющее имитировать в объектах JavaScript и наследование, и область видимости. Его работа заслуживает внимания, но, к сожалению, мы не имеем возможности обсуждать ее здесь. Для того чтобы воспользоваться результатами работы Крокфорда, необходимо потратить достаточно длительное время на их изучение, поэтому если вы ие собираетесь пользоваться данной технологией постоянно, а проект надо выполнить в сжатые сроки, тратить время на знакомство с ней не следует. В аналогичном положении оказываются те разработчики, которые решают, стоит ли им применять Java-библиотеку Struts или Tapestry. Подобные средства надо либо использовать постоянно, либо не пользоваться ими вовсе. Те, кого заинтересовал данный вопрос, могут найти дополнительную информацию на Web-узле Крокфорда.

В объектном программировании наблюдается постепенный переход от наследования к композиции. В этом случае сложные функции реализуются в составе вспомогательных классов, которыми могут воспользоваться любые объекты. Во многих случаях композиция не уступает наследованию, причем JavaScript обеспечивает ее поддержку.

Продолжая обсуждение объектов JavaScript, необходимо обратить внимание на отражение.

Б.2.5. Отражение в JavaScript-объектах

Обычно при написании кода программист представляет себе структуру объекта, с которым он собирается работать. Однако в некоторых случаях приходится использовать незнакомые объекты. Перед тем как выполнять с такими объектами какие-либо действия, надо получить сведения о их свойствах и методах. Например, при написании системы отладки или средств протоколирования необходимо обеспечить поддержку произвольных объектов, поставляемых извне. Процесс получения сведений о структуре объекта называется *отражением*. Он знаком программистам, использующим Java и .NET.

Если нам надо выяснить, существует ли в составе JavaScript-объекта некоторое свойство или метод, мы можем выполнить несложную проверку.

```
if (MyObject.someProperty){

}
```

Однако, если MyObject.someProperty содержит логическое значение false, или число 0, или специальное значение null, тело оператора if не будет выполнено. Более строгая проверка выглядит следующим образом:

```
if (typeof(MyObject.someProperty) != "undefined"){
```

Если нас интересует тип свойства, мы можем использовать оператор instanceof, который позволяет определить основные встроенные типы,

```
if (myObj instanceof Array){

}else if (myObj instanceof Object){

}
```

Он также дает возможность распознавать классы, которые мы определили посредством конструкторов.

```
if (myObj instanceof MyObject){

}
```

Если вы хотите использовать instanceof для проверки своих классов, вам необходимо учитывать некоторые особенности. Во-первых, любой объект, созданный посредством JSON, — это либо Object, либо Array. Во-вторых, между встроенными объектами могут существовать отношения "родительский-дочерний". Например, Function и Array являются потомками Object, поэтому порядок проверки на соответствие тому или иному типу имеет значение. Предположим, что мы написали следующий фрагмент кода и передали функции объект Array:

```
function testType(myObj){
  if {myObj instanceof Array){
    alert{"it's an array");
  }else if {myObj instanceof Object){
    alert("it's an object");
  }
}
testType([1,2,3,4J);
```

В результате мы получим сообщение о том, что объект имеет тип Array. Эта информация вполне корректна. Внесем небольшие изменения в тот же код.

```
function testType(myObj){
  if (myObj instanceof Object){
    alert("it's an object");
  }else if (myObj instanceof Array){
    alert("it's an array");
  }
}
testType([1,2,3,4]);
```

Теперь при тех же условиях мы получим ответ, что объект имеет тип Object. Эта информация также формально правильна.

Бывают случаи, когда необходимо выявить все свойства и функции объекта. Сделать это мы можем с помощью простого цикла for.

```
function MyObject(){
  this.color='red';
  this.flavor='strawberry';
  this.azimuth='45 degrees';
  this.favoriteDog='collie';
}
var my0bj=new MyObject();
var debug="discovering...\n";
for (var i in myObj){
  debug+=i+" -> "+myObj[i]+"\n";
}
alert(debug);
```

В данном случае тело цикла выполняется четыре раза, предоставляя все значения, заданные в конструкторе. Такой подход применим и для встроенных объектов, однако для узла D0M размер окна с сообщением оказывается слишком большим. Более профессиональный способ подобной проверки рассматривался в главах 5 и 6 при создании пользовательского интерфейса ObjectViewer.

В объектно-ориентированных языках существует еще одна возможность, которую нам необходимо обсудить, — виртуальные классы или интерфейсы.

B.2.6. Интерфейсы и виртуальные типы

При разработке программ часто возникают ситуации, когда необходимо описать поведение некоторого элемента, не реализуя его. Выше в данном приложении мы рассматривали объект Shape, подклассами которого являются прямоугольники, круги и т.д. В этом случае мы можем точно сказать, что никогда не сможем получить абстрактную геометрическую фигуру саму по себе в отрыве от ее конкретной реализации. В объекте Shape определены общие абстрактные свойства всех геометрических фигур. В реальном мире не существует аналогов подобных объектов.

Виртуальные классы C++ и интерфейсы Java предоставляют разработчикам средства описания подобных данных в программе. Термин "интерфейс" мы часто используем для описания взаимодействия различных компонентов программного обеспечения. Чтобы обеспечить подобное взаимодействие, автор библиотеки для работы с геометрическими фигурами не должен принимать во внимание конкретные реализации этих фигур. С другой стороны, программист, реализующий интерфейс Shape, не должен учитывать особенности библиотечного кода или существующие реализации данного интерфейса.

Интерфейсы обеспечивают разделение понятий, кроме того, они широко используются в описании образов разработки. Применять их желательно и при написании Ajax-приложений. Однако в JavaScript понятие интерфейса отсутствует, поэтому нам необходимо найти ему замену.

Проще всего неформально определить взаимодействие и считать, что разработчики каждого объекта знают, как поддерживать его. Дейв Томас (Dave Thomas) назвал данный подход *фиктивными типами* (duck typing): если нечто описано как тип, следовательно, оно и является типом. Возвращаясь

к нашему примеру интерфейса Shape, можно сказать, что если для некоторого элемента можно вычислить периметр и площадь, то этот элемент является геометрической фигурой.

Предположим, что нам надо сложить площади двух геометрических фигур. Используя язык Java, мы можем написать следующий метод:

```
public double addAreas{Shape si, Shape s2){
  return si.getArea()+s2.getArea();
}
```

Заголовок метода запрещает передавать ему что-либо кроме геометрической фигуры, поэтому, занимаясь реализацией тела метода, мы знаем, что учитываются соглашения о взаимодействии. В JavaScript параметры метода могут иметь любой тип, поэтому нельзя говорить об автоматическом выполнении соглашений.

```
function addAreas(sl,s2) {
  return si.getArea()+s2.getArea{);
}
```

Если в объекте, переданном в качестве параметра, отсутствует функция getArea (), интерпретатор JavaScript сгенерирует сообщение об ошибке. Перед вызовом функции можно проверить, присутствует ли она.

```
function hasArea(obj){
  return obj && obj.getArea && obj.getArea instanceof Function;
}
```

Код нашей функции необходимо модифицировать с учетом данной проверки.

```
function addAreas(si,s2){
  var total=null;
  if (hasArea(sl) && hasArea(s2)){
    total-sl.getArea()+s2.getArea();
  }
  return total;
}
```

Используя отражение JavaScript, мы можем написать универсальную функцию, проверяющую, имеется ли в составе объекта функция с конкретным именем.

```
function implements(obj,funcName){
  return obj && obj[funcName] && obj[funcName] instanceof Function;
}
```

Мы можем также присоединить эту функцию к прототипу класса Object.

```
Object.prototype.implements=function{funcName){
  return this && this[funcName] && this[funcName] instanceof Function,
}
```

Это позволяет организовать проверку функции по имени.

```
function hasArea(obj){
  return obj.implements("getArea");
}
```

Можно даже проверить соответствие параметра интерфейсу.

```
function isShape(obj){
  return obj.implements("getArea") && obj.implements("getPerimeter");
}
```

Данный подход обеспечивает определенную степень надежности, однако не такую, как в программах на Java. Нам может встретиться объект, в котором имеется функция getAreaO, возвращающая, например, вместо числового значения строку. Тип значения, возвращаемого JavaScript-функцией, неизвестен до тех пор, пока мы не вызовем ее. (Мы можем написать, например, функцию, которая будет возвращать в рабочие дни числовое значение, а в выходные — строку символов.) Создать набор функций для проверки возвращаемых значений достаточно просто. Пример функции из подобного набора приведен ниже.

```
function isNum(arg){
  return parseFloat(arg)!=NaN;
}
```

где NaN — сокращение от "not a number" ("не число"). Это специальная переменная JavaScript, предназначенная для обработки ошибок числовых форматов. Приведенная выше функция возвращает значение true для строки, начинающейся с цифр. Функции parseFloat O и parselntO пытаются извлечь числовое значение из переданного им параметра. Однако выражение parseFloat("64 hectares") вернет значение 64, а не NaN.

Продолжим работу над функцией addAreas {).

```
function addAreas(si,s2){
  var total=null;
  if (hasArea(sl) && hasArea(s2)){
    var al=sl.getAreaf); var a2=s2.getArea();
    if {isNum(al) && isNum(a2)){total=parseFloat(al)+parseFloat(a2);
    }
  }
  return total;
}
```

В данном случае для обоих параметров вызывается функция parseFloat(). Это сделано для того, чтобы обеспечить по возможности обработку любых строк, которые могут быть случайно переданы функции. Если si содержит значение 32, а s2 — значение 64 hectares, то функция addAreas() вернет число 96. Если бы функция parseFloat() не использовалась, то возвращаемым значением была бы строка 3264 hectares!

Подводя итоги, можно сказать следующее: использовать фиктивные типы достаточно просто, но при этом приходится полностью доверять другим участникам рабочей группы и надеяться, что они корректно реализуют все детали кода. Такое доверие оправдано в том случае, если численность рабочей группы мала. При работе над большим проектом, когда, группа разделяется на подгруппы, доверие неизбежно ослабевает. Ели вы захотите реализовать проверки "поверх" фиктивных типов, информация, приведенная в этом разДеле, поможет вам осуществить задуманное.

Мы обсудили язык JavaScript с точки зрения объектов. Теперь перейдем на уровень функций и рассмотрим их особенности.

Б.3. Методы и функции

В предыдущем разделе, как и в большинстве глав данной книги, мы определяли функции и вызывали их. Программисты, имеющие опыт работы на Java или C#, могут посчитать, что функции JavaScript — это нечто вроде знакомых им методов, отличающиеся правилами записи. В этом разделе мы уделим функциям больше внимания и покажем, каких результатов можно добиться с их помощью.

Б.3.1. Функции как независимые элементы

Функции похожи на Java-методы тем, что для них определены параметры и возвращаемое значение. В тоже время между функциями JavaScript и Java-методами есть существенные различия. Java-метод связан с классом, в котором он определен, и не может существовать вне этого класса. Функция JavaScript — это независимый элемент, функционирующий по собственным правилам. (Статические Java-методы занимают промежуточное положение — они принадлежат лишь классу и не связаны ни с одним его экземпляром.)

Программисты, использовавшие языки семейства C, скажут, что функции JavaScript — это то же самое, что указатели на функции в C++. Они будут правы, но не совсем.

В JavaScript Function — это встроенный объект. Конечно же, он содержит исполняемый код и допускает вызов, но он также является потомком Object и может делать то же, что и любой объект JavaScript, в том числе и хранить значения свойств. К объекту Function можно присоединить в качестве методов другие объекты Function (многие разработчики поступают так).

Вы уже знаете, как получить ссылку на объект Function. Чаще всего требуется одновременно и сослаться на функцию, и вызвать ее.

```
var
result=MyObject.doSomething(x,y, z)
```

Однако Function — это также независимый объект, и к нему можно обратиться с помощью метода call() (а также метода apply(), предоставляющего аналогичные возможности).

```
var result=MyObj ect.doSomething.call(MyOtherObject, x, y, z)
```

Допустимо даже следующее выражение:

```
var result=MyObject['doSomething'].call(MyOtherObject,x,y,z)
```

Первый параметр метода Function.call{) — это объект, выполняющий роль контекста функции при ее вызове. Последующие параметры рассматриваются как аргументы самой функции. Метод apply() действует несколько по-другому, в частности, второй параметр представляет собой массив аргументов, передаваемых функции. Такой подход обеспечивает несколько большую гибкость.

Следует заметить, что число параметров функции JavaScript не фиксировано. В Java или C# при попытке вызывать метод, указав количество параметров больше или меньше объявленного, на этапе компиляции генерируется сообщение об ошибке. JavaScript лишь игнорирует лишние параметры и при-

звaивает недостающим значение undefined. Искусно реализованные функции чогут даже запрашивать список параметров посредством свойства arguments и присваивать недостающим параметрам осмысленные значения по умолчанию. Они также могут генерировать исключения и предпринимать другие черы по обеспечению работоспособности программы. Например, можно реализовать в рамках одной функции get- и set-метод.

```
function area(value){
  if (value){
    this.area=value;
  }
  return this.area;
}
```

Если мы вызовем area{) без параметров, значение value будет не определено, присвоение не будет иметь смысла и функция выступит в роли getлетода. Указав параметр, можно превратить функцию в set-метод. Данный годход широко использовал Майк Фостер в своей библиотеке x (см. ссылку в сонце данного приложения и главу 3). Если вы поработаете с библиотекой x, данный подход вскоре станет привычен вам.

Тот факт, что функции могут быть независимыми от конкретных объекюв, открывает ряд интересных возможностей.

5.3.2. Присоединение функций к объектам

Тоскольку JavaScript — функциональный язык, он позволяет определять функции в отсутствии объектов, например:

```
function doSomething(x,y,z){ ... }
```

Тело функции также можно определить непосредственно при вызове.

```
var doSomething=function(x,y,z){ ... }
```

Чтобы отдать должное объектному подходу, разработчики JavaScript [редусмотрели возможность присоединения функций к объектам. При этом |ни становятся похожими на методы Java или C#. Связать функцию с объктом можно несколькими способами.

Можно присоединить существующую функцию к существующему объеку (в этом случае вызвать функцию может только сам объект, но не другой, врожденный с использованием того же прототипа).

```
myObj.doSomethingNew=doSomething;
myObj.doSomethingNew(x,y,z);
```

Мы также можем присоединить функцию таким образом, что к ней бу-,ет иметь доступ любой экземпляр класса. Для этого надо добавить функ-,ию (либо предопределенную, либо объявленную в процессе присоединения) конструкторе (см. раздел Б.2.2) или связать ее с помощью прототипа.

Несмотря на возможность присоединения функции к объекту, связь меж-у ними нельзя считать очень прочной.

Б.3.3. Заимствование функций из других объектов

Способность функций выступать в роли независимых элементов существенно влияет на возможности языка. Очень важно представлять себе это влияние при программировании обработчиков событий, связанных с элементами внутреннего интерфейса; именно этим часто приходится заниматься программистам, разрабатывающим Ajax-приложения.

В чем же состоят эти новые возможности? Каждый объект может заимствовать функцию другого объекта и вызывать ее. Определим класс, описывающий некоторое дерево.

```
function Tree(name, leaf, bark){
  this.name=name;
  this.leaf=leaf;
  this.bark=bark;
}
```

Свяжем с этим классом функцию, предоставляющую описание дерева.

```
Tree.prototype.describe=function(){
 return this.name+": leaf="+this.leaf+", bark="+this.bark;
}
```

Если мы создадим экземпляр объекта Tree и запросим у него описание, то получим вполне предсказуемый результат.

```
var Beech=new Tree("green, serrated edge","smooth");
alert{Beech.describe());
```

В окне отображается текст Beech: leaf=green, serrated edge, bark=-smooth. Пока никаких неожиданностей не предвидится. Теперь определим класс, представляющий собаку.

```
function Dog(name,bark){
  this.name=name;
  this.bark=bark;
}
```

Создадим экземпляр класса Dog.

```
var Snowy=new Dog("snowy","wau! wau!");
```

Объект мог бы продемонстрировать описание звуков, издаваемых собакой (к тому же мы определили их), однако функция, с помощью которой можно было бы сделать это, отсутствует. Тем не менее можно воспользоваться функцией класса Tree.

```
var tmpFunc=Beech.describe;
tmpFunc.call(Snowy);
```

Как вы помните, первый параметр, передаваемый function.call(), — это объект контекста, на который ссылается специальная переменная this. Приведенный выше фрагмент кода генерирует окно с сообщением, в котором отображается текст Snowy: leaf=undefined, bark=wau! wau!. Формат сообщения нельзя назвать идеальным, но все же это лучше, чем ничего.

Но как это происходит? Почему собака получает возможность вызвать функцию, которая на самом деле принадлежит дереву? Ответ несколько неожиданный: на самом деле функция не принадлежит дереву. При при-

оединении функции к прототипу Tree связывание осуществляется лишь гастолько, насколько это необходимо для поддержки выражений типа lyTree.describe (). Если рассмотреть внутреннюю реализацию программы, •о можно увидеть, что функция хранится как фрагмент текста и выполняется при каждом вызове. При этом конкретное значение this изменяется от .бращения к обращению.

Заимствование функций — интересный прием; освоив, вы можете применять его при написания своего кода. Однако для повышения надежности мы ,се же рекомендуем реализовать для Snowy собственный метод bark(). Мы бсуждаем данный вопрос лишь потому что при написании кода для поддержки событий, вам необходимо знать, какие действия браузер выполняет ез вашего участия.

13.4. Обработка событий в Ajax-программах и контексты функций

)бработчики событий в Ajax-приложениях похожи на обработчики, реализумые инструментальными средствами создания пользовательских интерфейов для многих языков. Как было показано в главе 4, события, связанные с [ышью и клавиатурой, выделяются в отдельные категории. В рассмотренном ами примере использовался обработчик onclick; это событие генерируется о щелчку мышью на видимом элементе. Подробное обсуждение обработки обытий в DHTML не входит в круг задач, рассматриваемых в данной книге; [ы ограничимся лишь одной особенностью, которая может стать ловушкой ля неопытных программистов. Обработчик события может определяться в [TML-дескрипторе, например:

```
<div id='rayDiv' onclick='alert:alert(this.id)'x/div>
```

Его также можно задать из программы.

```
function clickHandler(){ alert(this.id); }
myDiv.onclick=clickHandler;
```

Заметьте, что во втором случае мы передаем ссылку на объект Function (после clickHandler круглые скобки не указываются). При объявлении >ункции в HTML-дескрипторе создается безымянная функция. Эта запись <вивалентна приведенной ниже.

```
myDiv.onclick=function{){ alert(this.id); }
```

Заметьте, что в обоих случаях параметры функции не предусмотрены: е существует способа задать их при щелчке мышью. Если же мы щелкатf на элементе DOM, при вызове функции передается в качестве параметра 5ъект Event, а сам элемент выполняет роль объекта контекста. Зная это, ожно избежать многих неприятностей, в особенности при написании объстного кода. Источником проблем может стать тот факт, что узел DOM -егда используется в качестве контекста, даже если функция присоединена ^средством прототипа к другому объекту.

В приведенном ниже примере мы определяем простой объект с обработаком события для видимого интерфейсного элемента. В данном случае объ-

ект можно рассматривать как модель, обработчик события выполняет роль контроллера, а элемент DOM является представлением.

```
function  myObj(id,div){
this.id=id;
this.div=div;
this.div.onclick=this.clickHandler;
}
```

Конструктору передается идентификатор и элемент DOM, с которым связывается обработчик события. Сам обработчик определен следующим образом:

```
myObj.prototype.clickHandler=function(event){
  alert(this.id);
}
```

Таким образом, щелкнув мышью на интерфейсном элементе, мы должны увидеть идентификатор объекта, не так ли? На самом деле этого не произойдет, так как функция myObj .clickHandler () заимствуется браузером (точно так же, как в предыдущем разделе собака заимствовала метод дерева), в результате чего она выполняется в контексте элемента, а не объекта модели. Поскольку элемент имеет встроенный идентификатор, это значение будет отображено и, в зависимости от соглашений об именовании, может даже совпасть с идентификатором объекта модели. В результате у разработчика сформируется ложное представление о происходящем.

Если мы хотим, чтобы обработчик события ссылался на объект модели, мы должны передать ссылку несколько по-иному. Существуют два способа решения этой задачи. Один из них, на наш взгляд, наверняка предпочтительнее другого, по крайней мере, он давно доказал свою работоспособность. Однако одна из задач данной книги — давать имена различным программным решениям, поэтому мы рассмотрим оба способа.

Ссылка на модель по имени

Согласно описываемому подходу, каждому экземпляру объекта модели ставится в соответствие уникальный идентификатор и поддерживается глобальный массив объектов. Имея ссылку на элемент DOM, мы можем обратиться к объекту модели, используя в качестве ключа для поиска в массиве часть идентификатора. Данный подход иллюстрируется на рис. Б.1.

Необходимость генерации уникального идентификатора для каждого элемента, несомненно, следует отнести к накладным расходам, однако эта задача может быть выполнена автоматически. Частью ключа может стать, например, длина массива или, при генерации кода на Web-сервере, ключ для базы данных.

В качестве примера рассмотрим объект типа myObj, который содержит строку заголовка, реагирующую на щелчок мышью путем вызова функции myObj.foo{).

Ниже представлен глобальный массив.

```
var MyObjects=new Array();
```

Функция конструктора, регистрирующая объект модели в этом массиве, выглядит следующим **образом:**

Рис. Б.18. Ссылка на модель из обработчика события по имени. Производится разбор идентификатора, и результаты разбора используются в качестве ключевого значения для поиска в глобальном массиве

```
function myObj(id){
  this.uid=id;
  MyObjects[this.uid]=this;

  this.render();
}
```

Ниже приведен метод объекта myObj. Мы вызываем его щелчком на строке-заголовка.

```
myObj.prototype.f00=function(){
  alert('foooo!M'+this.uid);
}
```

Метод render() объекта создает узлы DOM.

```
myObj.prototype.render-function(){

  this.body=docuraent.createElement("div");
  this.body.id=this.uid+"_body";

  this.titleBar=document.createElement("div");
  this.titleBar.id=this.uid+"_titleBar";
  this.titleBar.onclick=fooEventHandler;

}
```

Если мы создаем в представлении DOM-узлы для модели, то связываем ними идентификаторы, содержащие идентификаторы модели.

Заметьте, что ссылка на функцию f ooEventHandler () присваивается юйству onclick элемента DOM, соответствующего строке-заголовка.

```
function fooEventHandler(event){
  var modelObj=getMyObj(this.id);
    if (modelObj){ modelObj.foo(); } }}
```

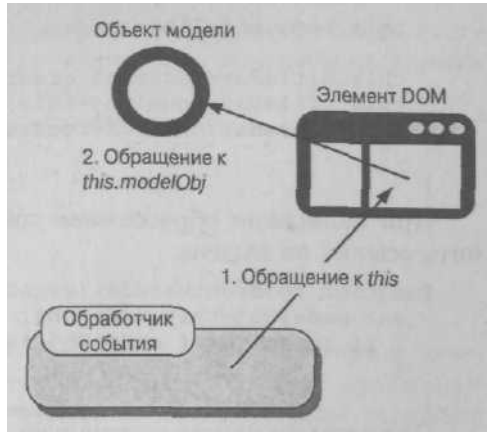


Рис. Б.19. Присоединение ссылки на модель непосредственно к узлу DOM упрощает поиск модели функцией обработки события

Чтобы вызвать метод f oo (), функция обработчика события должна найти экземпляр myob j. Для этой цели мы создаем специальный метод.

```
function  getMyObj(id){
  var  key-id.split(•_")[0];
  return  MyObjectsfkey];
}
```

Этот метод использует свойство id DOM-узла для того, чтобы извлечь ключ, с помощью которого можно получить объект модели из глобального массива.

Метод ссылки на модель по имени хорошо зарекомендовал себя на практике, однако существует более простой путь решения той же задачи, при котором нет необходимости снабжать узлы DOM длинными идентификаторами. (Трудно сказать, хорош этот способ или нет. С одной стороны, он приводит к излишнему использованию памяти, а с другой стороны — упрощает отладку в Mozilla DOM Inspector.)

Присоединение модели к узлу DOM

Согласно второму подходу к обработке событий DOM, все необходимые действия выполняются посредством ссылки на объекты, а дополнительные строки и поиск в глобальном массиве не требуются. Именно этот подход в основном используется в данной книге (рис. Б.2).

Данный подход существенно упрощает обработку события. Функция конструктора для объекта модели не выполняет специальных действий с идентификатором, а метод f oo () определяется так же, как и ранее. Создавая узлы DOM, мы используем особенность JavaScript, позволяющую присоединять к любому объекту произвольные атрибуты, и связываем объект модели непосредственно с узлом, получающим событие.

```
myObj.prototype.createView=function(){

  this.body=document.createElement("div");
```

```
    this.body.modelObj=this;

    this.titleBar=document.createElement("div");
    this.titleBar.modelObj=this;
    this.titleBar.onclick=fooEventHandler;

}
```

При написании обработчика события мы можем непосредственно получить ссылку на модель.

```
function fooEventHandler(event){
  var modelObj=this.modelObj;
    if (modelObj){ modelObj.foo(); }
  }
}
```

При этом не нужно специально выполнять поиск.

Необходимо, однако, учитывать одну важную особенность. При использовании данного подхода создается циклическая ссылка между DOM-узлом и переменной, не имеющей отношения к DOM. В некоторых браузерах это может вызвать проблемы в работе системы "сборки мусора". Если корректно реализовать данное решение, можно избежать неоправданного расходования памяти, но мы настоятельно рекомендуем вам изучить главу 7, перед тем как выполнять связывание модели с DOM-узлом.

Понимая, как функции JavaScript определяют свой контекст, можно реализовывать элегантные обработчики событий, пригодные для повторного использования. Возможность переключаться между контекстами может поначалу приводить к недоразумениям, но если хорошо освоить данный механизм, то им очень удобно пользоваться в работе.

Последняя особенность функций JavaScript, которую нам необходимо рассмотреть, — это способность создавать замыкания. В языках Java и C# понятие замыкания отсутствует, однако в некоторых языках сценариев, например Groovy и Boo, оно используется. С замыканиями также приходится сталкиваться в языке C# 2.0. Рассмотрим, что такое замыкания и как их использовать.

S.3.5. Замыкания в JavaScript

Сам по себе объект Function — неполный, и чтобы вызвать функцию, нам надо передать ей объект контекста и набор параметров (этот набор может быть пустым). Замыкание можно рассматривать как объект Function, связанный со всеми ресурсами, необходимыми для выполнения функции.

В JavaScript замыкания создаются неявно. Не существует конструктора, вызываемого с помощью выражения new ClosureO, и нет возможности получить ссылку на объект замыкания. Для создания замыкания достаточно объявить функцию в блоке кода (например, в теле другой функции) и обеспечить доступ к ней из-за пределов блока.

Вышесказанное может показаться несколько непривычным, поэтому стоит рассмотреть простой пример. Предположим, что мы определили объект,

представляющий робота, и фиксируем показания системного таймера, соответствующие времени создания конкретного робота. Для этой цели мы можем написать конструктор, подобный приведенному ниже.

```
function Robot(){
  var createTime=new Date();
  this.getAge=function{){
    var now=new Date();
    var age=now-createTime;
    return age;
  }
}
```

(Все роботы идентичны, поэтому мы не передаем конструктору в качестве параметра имя или какие-либо другие сведения.) Конечно, предпочтительнее бьуто бы оформить createTime в виде свойства так, как показано ниже, однако мы намеренно объявили локальную переменную, область видимости которой ограничивается блоком, где она определена (в данном случае конструктором).

```
this.createTime=new Date{);
```

Следующей строкой конструктора является определение функции getAge (Заметьте, что мы определили внутреннюю функцию в составе другой функции и что внутренняя функция использует локальную переменную createTime, принадлежащую области видимости внешней функции. Поступив таким образом, мы создали замыкание. Если мы определим робота и запросим его возраст, то получим значение в пределах 10-50 миллисекунд — интервал между первым выполнением сценария и загрузкой страницы.

```
var robbie=new Robot();
window.onload=function(){
  alert(robbie.getAge());
}
```

Несмотря на то что мы объявили createTime как локальную переменную для функции конструктора, процедура "сборки мусора" не затрагивает ее до тех пор, пока переменная robbie ссылается на объект Robot. Таким образом, произошло связывание посредством замыкания.

Замыкание присутствует, только если внутренняя функция создана в составе внешней. Предположим, что мы реструктуризировали код и вынесли функцию getAge () за пределы конструктора так, что ее могут совместно использовать все экземпляры класса Robot.

```
function Robot(){
  var createTime=new Date();
  this.getAge=roboAge;
}
function roboAge{){
  var now=new Date{);
  var age=now-createTime;
  return age;
};
```

В этом случае замыкание не будет создано и мы получим сообщение о том, что переменная createTime не определена.

Создать замыкание очень легко, и, к сожалению это можно сделать епреднамеренно. В результате локальные переменные будут связаны и про-,едура "сборки мусора" не затронет их. Если под действие случайно создан-ого замыкания попадут узлы DOM, это приведет к значительному расходо-аию памяти.

Чаще всего замыкания создаются при связывании функций обратного ызова, представляющих собой обработчики событий, с источниками этих обытий. Как было сказано в разделе Б.3.4, контекст и набор параметров, оторые получает функция обратного вызова, не всегда устраивают разра-отчика. Мы рассмотрели способ создания для элемента DOM дополнитель-ой ссылки на объект модели. В результате модель становится доступной осредством DOM-элемента. Замыкания обеспечивают альтернативный спо-об решения данной задачи.

```
myOb j.prototype.createView=function(){

  this.titleBar=docuroent.createElement("div"};
  var modelObj=this;
  this.titleBar.onclick=function(){
    fooEventHandler.call(modelObj);
  }
}
```

В определенном нами анонимном обработчике формируется ссылка на окальную переменную *modelObj*. В результате создается ссылка и доступ modelObj становится возможным в процессе выполнения функции. Заметь-е, что замыкания создаются только для локальных переменных; ссылка по-редством this не приводит к возникновению подобного эффекта.

Мы использовали данный подход в главе 2 при создании объекта ontentLoader. Функция обратного вызова onreadystatechange, предостав-яемая браузером Internet Explorer, возвращает в качестве контекста объект indow. Поскольку window определен глобально, мы не знаем, для какого бъекта ContentLoader изменилось свойство readyState. Чтобы решить эту адачу, мы передаем ссылку на соответствующий объект посредством замы-ания.

Мы настоятельно рекомендуем программистам, работающим над Ajax-риложениями, по возможности избегать замыканий и принимать альтер-ативные решения. Если вы будете использовать прототип для добавления эункций к объекту, то избежите дублирования функций и не создадите замы-ания. Перепишем наш класс Robot в соответствии с данной рекомендацией.

```
function  Robot(){
  this.createTirae=new Date{);
}
Robot.prototype.getAge=function(){
  var now=new Date();
  var  age=now-this.createTime;
  return age;
```

Функция getAgeO определяется только один раз, и, поскольку она свя-ана с прототипом, каждый созданный объект Robot имеет доступ к ней.

Замыкания находят свое применение, но мы не будем подробно рассматривать данный вопрос. Если вас заинтересовала эта тема и вы хотите изучить замыкания более подробно, вам следует прочитать статью Джима Лей (Jim Ley), ссылка на которую приведена в конце данного приложения.

Б.4. Выводы

В данном приложении мы рассмотрим ряд интересных особенностей языка JavaScript. При этом мы преследовали две цели. Во-первых, хотели продемонстрировать выразительные средства данного языка. Во-вторых, намеревались рассказать о некоторых решениях, часто принимаемых неопытными программистами и способных привести к возникновению проблем. Специалист, овладевший объектным мышлением, сразу назовет подобные решения неоптимальными и даже опасными.

Мы рассмотрели вопросы поддержки объектов в JavaScript и сходство между классами Object и Array, а также обсудили различные способы создания экземпляров объектов JavaScript: использование JSON, функций конструкторов и прототипов. Кроме того, вы узнали, как обеспечить в JavaScript такие средства объектно-ориентированных языков, как наследование и интерфейсы. Чтобы сделать это, приходится не следовать принципам языка, а скорее противодействовать им.

Говоря о JavaScript-объекте Function, мы показали, что возможно существование функций независимо от объектов, с которыми они связаны. Более того, объекты даже могут заимствовать функции друг у друга. Зная эту особенность, можно лучше понять модель событий JavaScript. И наконец, мы рассмотрели замыкания и показали, как общепринятые программные решения могут привести к непреднамеренному созданию замыкания, что в конечном итоге чревато неоправданным расходом памяти.

По сравнению с Java или C#, язык JavaScript обеспечивает гораздо большую степень гибкости и простор для применения различных стилей программирования. Подобная свобода, предоставляемая программисту, хороша лишь тогда, когда он знает, что должен сделать. Тем не менее при работе группы разработчиков особенности JavaScript могут привести к возникновению проблем. Чтобы избежать их, надо придерживаться соглашений о кодировании.

Если вы хорошо представляете себе работу средств JavaScript, то использование данного языка доставит вам немало приятных минут. Тем, кто до использования Ajax применял объектные языки, эта глава поможет побыстрее включиться в работу и избежать ряда ошибок.

Б.5. Ресурсы

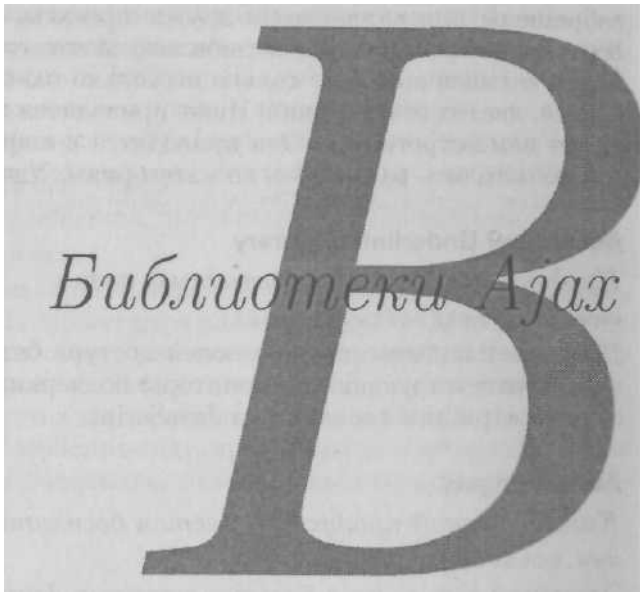
Книг, посвященных самому языку JavaScript, а не программированию в среде Web-браузера, сравнительно немного. Книга Дэвида Фланагана (David Flanagan) *JavaScript: The Definitive Guide* (O'Reilly, 2001) — безусловно, очень

серьезная работа. Она несколько устарела, но скоро должно выйти новое издание. Более новая книга Николаса Закаса (Nicholas Zakas) *Professional JavaScript for Web Developers* (Wrox, 2004) представляет собой хороший обзор возможностей языка, в том числе средств, появившихся недавно. Еще одной хорошей книгой является *Полный справочник по JavaScript, 2-е издание*. Томас Пауэлл, Фриц Шнайдер.

Полезные материалы можно найти в Web. Дуг Крокфорд (Doug Crockford) обсуждает вопросы применения объектного подхода при программировании на JavaScript, в частности, создание закрытых свойств и методов для классов (http://www.crockford.com/javascript/private.html) и реализацию наследования (http://www.crockford.com/javascript/private.html). На узле Питера-Пола Коха (Peter-Paul Koch) по адресу http://quirksmode. огд также рассматриваются многие особенности языка.

Материал Джима Лея (Jim Ley) о замыканиях в JavaScript можно найти в документе http://jibbering.com/faq/faq_notes/closures.html.

Библиотеках Майка Фостера расположена на сервере http: //www. cross-browser .com.


Библиотеки Ajax

В последнее время библиотеки Ajax и JavaScript прошли большой путь от небольших интерфейсных утилит до полномасштабных клиентских и серверных решений. В данном приложении мы попытаемся охватить весь диапазон предлагаемых продуктов и заранее просим прощения у тех компаний, чью продукцию мы не упомянули.

Авторы данной книги протестировали не все библиотеки и наборы инструментов; во многих случаях описание основано на информации от автора или поставщика. Если вы читаете данный текст через год после его публикации, многие описания будут неточными или устаревшими, а сами продукты — заброшены или включены в другие проекты. По нашему мнению, текущее состояние отрасли очень нестабильно, поэтому мы считаем, что через двенадцать месяцев выживет только несколько наиболее удачных библиотек.

Ну все, хватит отступлений! Ниже приводится обзор библиотек Ajax, которые могут вам встретиться. Они приводятся в алфавитном порядке; кроме того, мы попытались разбить их по категориям. Удачного кодирования!

Accesskey Underlining Library

Продукт с открытым исходным кодом

www.gerv.net/software/aul/

Добавляет подчеркивание ключей доступа без использования дескрипторов <и>. Соответствующие дескрипторы подчеркивания в DOM создаются с помощью атрибута accesskey и JavaScript.

Act iveWid gets

Коммерческий продукт; имеется бесплатная версия

www.activewidgets.com

Элементы управления богатых клиентов JavaScript; в настоящее время основной продукт предлагает полномасштабную поддержку действий с сеткой.

Ajax JavaServer Faces Framework

Продукт с открытым исходным кодом (Apache)

http://smirnov.org,ru/en/ajax-jsf.html

Предназначена для введения в любое существующее приложение JavaServer Faces инфраструктуры Ajax. Большинство существующих компонентов можно использовать без изменений или после простого преобразования в формат с поддержкой Ajax. Предлагает внедрение продукта в проект MyFaces. Отличия от спецификаций JSF минимальны.

Ajax JSP Tag Library

Продукт с открытым исходным кодом

http://ajaxtags.sourceforge.net/

Представляет собой набор дескрипторов JSP, упрощающих использование технологии Asynchronous JavaScript and XML (Ajax) на платформе JavaServer Pages. Данная библиотека дескрипторов облегчает разработку за счет того,

что разработчикам J2EE не приходится писать необходимый JavaScript-код для реализации Web-формы с поддержкой Ajax.
Функция Autocomplete извлекает список значений, согласующихся со строкой, введенной в текстовую форму, по мере ввода этой строки пользователем. Функция Callout отображает выносной или всплывающий блок, привязанный к элементу HTML с помощью события onclick. Функция Select заполняет значениями поле второго списка, основываясь на значении, выбранном из первого списка. Функция Toggle переключает значение скрытого поля формы между true и false, в то же время переключаясь между двумя изображениями. Дескриптор update Field обновляет одно или несколько значений поля формы, основываясь на отклике, порожденном вводом текста в другое поле.

Ajax.NET

Майкл Шварц (Michael Schwarz), 2005
Статус продукта не оговаривается, использование бесплатное
http://weblogs.asp.net/mschwarz/

Библиотека, допускающая различные варианты доступа JavaScript-кода к серверному приложению .NET. Может передавать вызовы от JavaScript к методам .NET и возвращаться к обратным вызовам JavaScript. Может обращаться к информации о сеансе из JavaScript-кода. Кэширует результаты. Не требует изменения исходного кода серверной части сценария. Значениям, возвращаемым клиентскому JavaScript-коду, предлагается полная поддержка классов (в том числе классов DataTable, DataSet, DataView, Arrays и Collections).

AjaxAC

Продукт с открытым исходным кодом (Apache 2.0)
http://ajax.zervaas.com.au

Инкапсулирует целое приложение в один класс PHP, содержащий помимо кода приложения дополнительные библиотеки JavaScript (если они требуются). Вызов файла PHP (HTML-страницы) довольно прост. Вы создаете класс приложения, затем ссылаетесь на JavaScript-код приложения и присоединяете к приложению все необходимые HTML-элементы. Вызывающий HTML-код не засоряется JavaScript-кодом, все события присоединяются динамически. AjaxAC легко интегрировать с процессором шаблонов и вставить в существующие классы PHP или базу данных MySQL для возврата данных от подзапросов. Расширяемая структура элементов управления позволяет легко создавать дополнительные JavaScript-объекты (правда, по мнению автора, это требует определенных усилий).

\jaxAspects

Бесплатное использование с указанием первоисточника
ittp://ajaxaspects.blogspot.com
Представляет собой процессор, использующий модули доступа JavaScript для зызова серверных методов Web-служб. Для связи клиента и сервера применяются стандартные средства SOAP и WSDL. В качестве параметров и возвращаемых значений допускается использование простых типов и объектов *KML*. Кроме того, продукт поддерживает кэширование и очередь действий.

^jaxCaller

Майкл Мехемофф (Michael Mahemoff), 2005
Продукт с открытым исходным кодом
ittp://ajaxify.com/run/testAjaxCaller
Интерфейсный объект с многопоточной поддержкой XMLHttpRequest. Предназначен в основном для начинающих пользователей Ajax. В настоящее время продукт находится в стадии разработки; доступна альфа-версия; поставляется только с демоверсией приложения "живого" поиска AjaxPatterns. Разработан согласно принципам REST.

AJaxFaces

Продукт с открытым исходным кодом (ASF)
http://myfaces.apache,org/
Экспериментальная реализация JavaServer Faces с поддержкой Ajax от Apache.

BackBase

Коммерческий продукт с общедоступной версией
http://www.backbase.com
Комплексная структура на основе браузера, поддерживающая богатые функциональные возможности браузера, а также интеграцию с .NET- и Java-кодом. BackBase предлагает приложение ША (Rich Internet Application), радикально повышающее практичность и эффективность интерактивных приложений, а также производительность разработчика. Используя BackBase, вы можете создавать Web-приложения с богатым и дружественным пользовательским интерфейсом. BackBase предлагает разделение представления и логики с помощью специального пространства имен XHTML.

Behaviour

Вен Нолан (Ben Nolan), 2005
Продукт с открытым исходным кодом
www.ripcord.co.nz/behaviour/ \
Действие Behaviour основано на использовании селекторов CSS для добавления JavaScript-кода к элементам DOM. Вы создаете хэш селекторов и функ-

ций CSS, принимающих элемент и добавляющих к нему обработчики событий JavaScript (например, onclick). Затем вы регистрируете эти правила на странице и сравниваете их с соответствующими элементами DOM, после чего добавляете в документ код JavaScript. Этот код разработан таким образом, что вы можете рассматривать файлы правил как таблицы стилей (т.е. все, что нужно для их использования, — это включить их на страницу). Библиотека Behaviour предназначена для того, чтобы убрать громоздкую механику атрибутов onclick и узлов сценария со страниц, чтобы они не запутывали ее содержимое. Благодаря своей организации библиотека достаточно удобна и может облегчить повторное использование вашего JavaScript-кода.

Bindows

Коммерческий продукт

www.bindows.net

Набор инструментальных средств разработки программного обеспечения (Software Development Kit — SDK), позволяющий генерировать настолько богатые интерактивные Интернет-приложения, что они могут соперничать с современными настольными приложениями, реализованными с интенсивным использованием DHTML, JavaScript, CSS и XML. Приложения Bindows не требуют загрузки и установки на стороне пользователя, необходим только браузер (средства Java, Flash или ActiveX не используются). Bindows предлагает богатый набор элементов управления окнами, а также встроенную поддержку XML, SOAP и XML-RPC.

BlueShoes

Коммерческий продукт; имеется бесплатная версия

www.blueshoes.org

Богатый набор компонентов, включающий текстовый редактор WYSIWYG и элемент управления электронными таблицами.

CakePHP

Продукт с открытым исходным кодом

http://cakephp.org/

Полный перенос продукта Ruby on Rails на платформу PHP с прекрасной поддержкой Ajax.

CL-Ajax

Ричард Ньюмен (Richard Newman), 2005

Продукт с открытым исходным кодом

http://cliki.net/cl-ajax

Направляет вызовы JavaScript непосредственно серверным функциям Lisp; генерирует суррогат JavaScript; может обращаться к функциям JavaScript или объектам DOM; может интегрироваться в SAJAX.

omfortASP.NET

Тредварителъный выпуск коммерческого продукта; имеется бесплатая версия

ww.daniel-zeiss.de/ComfortASP/

[редставляет подход, позволяющий разработчикам использовать чистое прораммирование ASP.NET для получения возможностей в стиле Ajax. Для еализации этих возможностей ComfortASP.NET использует Ajax (DHTML, avaScript, XML, HTTP), но Web-разработчики пишут только серверный код .SP.NET.

oolest DHTML Calendar

Тродукт с открытым исходным кодом с коммерческой поддержкой

ww.dynarch.com/projects/calendar/

[встраиваемый компонент-календарь JavaScript; может применяться для юрмирования раскрывающихся списков, стиль которых определяется с поющью CSS.

;PAINT

3ross-Platform Asynchronous Interface Toolkit)

Тродукт с открытым исходным кодом (GPL и LGPL)

ttp://cpaint.sourceforge.net

"трогая реализация Ajax и JSRS (JavaScript Remote Scripting), поддержива->щая PHP и ASP/VBScript. CPAINT предлагает код, требуемый для реалиации Ajax и JSRS на сервере, когда возвращаемые данные обрабатываются, Форматируются и отображаются на стороне клиента с помощью. JavaScript. >лагодаря этому можно написать приложение, предлагающее пользователю чень быструю обратную связь.

)ojo

\лекс Рассел (Alex Russell), 2004

Тродукт с открытым исходным кодом

ttp://dojotoolkit.org

1редлагает несколько библиотек для использования с Ajax, включая элемены управления окнами, модель событий и передачу сообщений с использова-[ием XMLHttpRequest и другие технологии. Поддерживает JavaScript в Webраузере.

)WR (Direct Web Remoting)

Тродукт с открытым исходным кодом (Apache)

'ww.getahead.ltd.uk/dwr

)болочка для вызова методов Java непосредственно из JavaScript-кода. Подобно SAJAX, она может передавать вызовы от JavaScript-кода методам Java, і затем возвращаться к обратным вызовам JavaScript Ее можно использо-

вать с любой Web-структурой (например, Struts или lapestry), поддерживающей философию KISS/POJO в стиле Spring. Данный продукт планировалось включить в следующую версию структуры OpenSymphony Web Works.

Echo 2

Продукт с открытым исходным кодом (MPL или GPL)
www.nextapp-com/products/echo2

Echo 2 позволяет кодировать приложения Ajax на Java, автоматически генерировать HTML- и JavaScript-код, а также координировать поток сообщений между браузером и сервером. Предлагает передачу сообщений в формате XML. Если требуется, разработчик может вручную писать JavaScript-компоненты.

f(m)

Продукт с открытым исходным кодом
http://fm.dept-z.com/

Представляет собой основанную на .NET библиотеку базового класса ECMAScript. Предполагается, что данный продукт станет основой нового поколения Web-приложений, основанных на браузере.

FCKEditor

Продукт с открытым исходным кодом
www.fckeditor.net

Богатый WYSIWYG-редактор; может загружаться в текстовую область HTML с помощью одной строки JavaScript-кода, что позволяет легко интегрировать его в существующие Web-приложения, системы CMS, энциклопедии и т.п. По своим функциональным возможностям очень похож на TinyMCE.

Flash JavaScript Integration Kit

Продукт с открытым исходным кодом
www.osflash.org/doku.php?id=flashjs

Данный продукт разрешает смешивание кода JavaScript и Flash; вызов функций ActionScript из JavaScript-кода и наоборот. Между двумя средами допускается передача данных всех основных типов.

Google AjaxSLT

Продукт с открытым исходным кодом (BSD)
http://goog-aj axslt.sourceforge.net

Продукт создан инновационной компанией-производителем приложений поиска Google. Представляет собой JavaScript-оболочку для выполнения XSLT-преобразований и запросов XPath. Разработан в ходе работы над Google Map.

3uise

Коммерческий продукт; имеется бесплатная версия

• *mw*.javaguise.com

Серверная компонентная модель на основе Java (чем-то похожа на JSF, но троще). В настоящее время для улучшения оперативности в продукт включается поддержка Ajax.

HTMLHttpRequest

Ангус Тернбул (Angus Tumbull), 2005

Продукт с открытым исходным кодом (LGPL)

tfww.twinhelix.com/JavaScript/htmlhttprequest/

Простая оболочка для удаленных сценариев. Для улучшения совместимости дспользует XMLHttpRequest и IFrames.

nteractive Website Framework

Продукт с открытым исходным кодом

ittp://sourceforge.net/projects/iwf/

Проект, цель которого — поддержка в браузере различных аспектов инфраструктуры Ajax. Описывается авторами как каркас для создания интерактивных Web-сайтов с использованием JavaScript, CSS, XML и HTML. Включает специальный компонент XML-разбора, позволяющий получать в удобном вице читаемый JavaScript-код. Содержит все необходимое для разработки Web-:айтов на основе Ajax и традиционных сценариев. Предлагает многопоточную реализацию XMLHttpRequest и интерфейсную оболочку для DOM, повышающую читаемость кода.

Jackbe

Коммерческий продукт

*ww.jackbe.com/solutions/development.html

Набор элементов управления окнами богатого клиента Ajax; может встраиваться в любое промежуточное программное обеспечение, например ASP-, Java-, .NET- или PHP-код.

JPSpan

Продукт с открытым исходным кодом (PHP)

uttp://jpspan.sourceforge.net/wiki/doku.php

JPSpan передает вызовы JavaScript непосредственно функциям PHP. В настоящее время выполняется интенсивное тестирование компонентов продукта.

jsolait

Продукт с открытым исходным кодом (LGPL)

http://jsolait.net

Набор библиотек JavaScript с открытым исходным кодом, поддерживающий криптографию, сериализацию, десериализацию, XML-RPC и JSON-RPC.

JSON

Продукт с открытым исходным кодом (большинство реализаций — LGPL)

www.json-rpc.org/

JSON — "обезжиренная XML-альтернатива"; JSON-RPC • - протокол удаленного вызова процедур, подобный XML-RPC, с хорошей поддержкой JavaScript-клиентов. Существуют реализации для нескольких серверных языков и платформ (в том числе Java, Python, Ruby и Perl).

JSRS (JavaScript Remote Scripting)

Брент Эшли (Brent Ashley), 2000

Продукт с открытым исходным кодом

www.ashleyit.com/rs/jsrs/test.htm

Перенаправляет вызовы из JavaScript-кода в код, написанный на серверном языке, и обратно. Поддерживаемые браузеры: IE 4+, Netscape 4.x, Netscape 6.x, Mozilla, Opera 7 и Galeon. Поддерживаемые серверные языки: ASP, ColdFusion, PerlCGI, PHP, Python и JSP (сервлеты).

LibXMLHttpRequest

Стефан В. Кот (Stephen W. Coate), 2003

Доступен исходный код; продукт защищен авторскими правами

www.whitefrost.com/servlet/connector?file=reference/2003/06/17/libXmlRequest.html

"Тонкая" интерфейсная оболочка объекта XMLHttpRequest.

Mochikit

Продукт с открытым исходным кодом (MIT)

www.mochikit.com/

Набор библиотек с акцентом на входе в систему, визуальных эффектах, асинхронном управлении задачами, форматировании даты/времени, включающий интерфейс "безболезненной" работы с DOM. Для представления DOM использует встроенные объекты Array JavaScript и форму записи в стиле JSON.

netWindows

Продукт с открытым исходным кодом

www.netwindows.org

Реализация в браузере полной среды рабочего стола/окон DHTML. Код в основном написан согласно существующим стандартам, специфические особенности браузеров не используются. Содержит реализацию обмена сообщениями с использованием сигналов и слотов, смоделированную на основе элементов управления Qt (Trolltech) и языка Smalltalk. Существует в виде отдельной библиотеки.

Oddpost

Коммерческий продукт

www.oddpost.com

Набор элементов управления окнами JavaScript; включает богатый полнофункциональный клиент электронной почты. В настоящий момент является частью Yahoo!.

OpenRico

Билл Скотт (Bill Scott), Даррен Джеймс (Darren James), 2005

Продукт с открытым исходным кодом

http://openrico.org

Многоцелевая структура с поддержкой Ajax. Основной акцент делается на поддержке таких элементов пользовательского интерфейса, как анимация, отделение содержимого от логики с помощью линий поведения, перетаскивание; имеется ряд встроенных элементов управления окнами. Разрабатывается при поддержке Sabre Airline Solutions на основе Prototype.

Pragmatic Objects

Продукт с открытым исходным кодом

tittp://pragmaticobjects.com/products.html

Входящий в продукт компонент WebControls представляет собой набор библиотек дескрипторов JSP, направленных на обогащение Web-приложений на эснове Java. В противоположность богатым, но "толстым" Web-приложениям, 'тонкое" Web-приложение состоит только из набора HTML-страниц с кодом JavaScript и CSS, которые визуализируются браузерами. Текущая реализация состоит из панели просмотра, элемента управления древовидной структурой *л* панели управления.

Prototype

Сэм Стефенсон (Sam Stephenson), 2004

Продукт с открытым исходным кодом

ittp://prototype.conio.net/

структура JavaScript, предназначенная для RIA-разработки. Включает фун-

даментальную библиотеку Ajax и набор инструментов для упрощения ее использования. Представляет собой процессор JavaScript для Ruby on Rails, Rico и Scriptaculous. Код JavaScript генерируется с помощью Ruby on Rails, однако потом его можно использовать и в других средах.

Qooxdoo

Продукт с открытым исходным кодом (LGPL)
http://qooxdoo.sourceforge.net
Библиотека пользовательского интерфейса Ajax с богатым набором встроенных компонентов и хорошо продуманной структурой. Включает элементы управления окнами и структурой. В качестве поддержки разработки предлагает таймеры для профилей и отладчик.

RSLite

Бренга Эшли (Brent Ashley), до 2000
www.ashleyit.com/rs/main.htm
Простой компонент, выпущенный как часть более сложной работы Брента Эшли Remote Scripting (см. выше раздел, посвященный JSRS).

Ruby on Rails

Дэвид Хейнемиер Хэнссон (David Heinemeier Hansson), 2004
Продукт с открытым исходным кодом (MIT)
www.rubyonrails.org
Общая основа для Web-разработки с хорошей поддержкой Ajax. На время начала Ajax-бума продукт находился в стадии разработки, поэтому становление Ruby on Rails происходило под сильным влиянием технологий Ajax. Генерирует большую часть (если не весь) JavaScript-кода для элементов управления окнами и анимации в браузере. Поддерживает планирование задач. Чтобы получить понятный и прямолинейный подход, авторы отказались от избыточного и всеобъемлющего продукта. Результат стал любимым средством многих Java-разработчиков. В данной книге Ruby on Rails использовался в основном из-за хорошей поддержки Ajax. В настоящий момент в продукт входят Prototype и Scriptaculous.

Sack

Продукт с открытым исходным кодом'(модифицированный MIT/X11)
http://twilightuniverse.com/2 005/05/sack-of-ajax
"Тонкая" интерфейсная оболочка объекта XMLHttpRequest. Позволяет вызывающей стороне задавать функцию или объект DOM обратного вызова. С помощью D0M обратного вызова текст ответа помещается непосредственно в D0M.

SAJAX

Продукт с открытым исходным кодом

www.modernmethod.com/saj ax

3AJAX направляет вызовы из JavaScript-кода в код, написанный на серверном языке, и обратно. Например, вызов метода JavaScript x__calculateBudget () направится на сервер и вызовет метод Java calculateBudget (), затем вернет в x_calculateBudget__cb() значение в виде JavaScript-кода. Продукт облегчает отображение суррогатной функции JavaScript в действие на сервере. Может направлять вызовы на различные серверные платформы: ASP, ColdFusion, Io, Lua, Perl, PHP, Python и Ruby.

Sarissa

Продукт с открытым исходным кодом (GPL и LGPL)

http://sarissa.sf.net

API JavaScript, инкапсулирующий функциональные возможности XML в вызовы, которые зависят от браузера. Поддерживает различные технологии XML, включая запросы XPath, XSLT и сериализацию объектов JavaScript в XML независимо от браузера.

Scriptaculous

Томас Фукс (Thomas Fuchs), 2004

Продукт с открытым исходным кодом

http://script.aculo.us

Библиотека визуальных эффектов с хорошей документацией, созданная с помощью JavaScript на основе Prototype. Включает демонстрации, примеры приложений и переносную библиотеку.

SWATO

Продукт с открытым исходным кодом (ASF)

http://swato.dev.java.net

SWATO (Shift Web Application TO...) представляет собой набор повторно используемых интегрированных библиотек Java/JavaScript, позволяющих легко переносить взаимодействие Web-приложений на фундамент Ajax. Серверная библиотека Java может развертываться во всех контейнерах, совместимых с Servlet 2.3+. Клиентская библиотека JavaScript работает в браузерах, поддерживающих XMLHttpRequest. Для упорядочения данных POJO на стороне сервера используется JSON, поэтому для удаленного доступа к данным в любой JavaScript-среде (HTML, XUL, SVG) требуется всего лишь закодировать нужные действия или интегрировать их с готовыми библиотеками JavaScript. Продукт поставляется с несколькими автономными компонентами (Auto-complete Textbox, Live Form, Live List и т.д.), которые помогают быстро разрабатывать Web-приложения.

Tibet

Коммерческий продукт
www.technicalpursuit.com

Предназначен для получения переносимого и понятного API JavaScript. Поддерживает стандарты Web-служб (SOAP и XML-RPC); имеет встроенную поддержку нескольких популярных Web-служб (Google, Amazon и Jabber). Включает интегрированную среду разработки, написанную на JavaScript.

TinyMCE

Продукт с открытым исходным кодом, коммерческой поддержкой и несколькими запатентованными модулями
http://tinymce.moxiecode.com/

Богатый WYSIWYG-редактор; может загружаться в текстовую область HTML с помощью одной строки JavaScript-кода, что позволяет легко интегрировать его в существующие Web-приложения, системы CMS, энциклопедии и т.п. По своим функциональным возможностям очень похож на FCKEditor.

TrimPath Templates

Продукт с открытым исходным кодом
http://trimpath.com/project/wiki/JavaScriptTemplates

Процессор образов JavaScript для соединения в браузере данных и представления.

Walter Zorn's DHTML Libraries

Продукт с открытым исходным кодом
www.walterzorn.com/index.htm

Предлагает поддержку перетаскивания и векторной графики (линии и кривые), используя в качестве пикселей элементы div.

WebORB for .NET

Коммерческий продукт; имеется бесплатная версия
www.themidnightcoders.com/weborb/aboutWeborb.htm

Платформа для разработки богатых клиентских приложений Ajax и Flash с возможностью их последующего объединения с объектами .NET и Web-службами XML.

WebORB for Java

Коммерческий продукт; имеется бесплатная/общедоступная версия
www.themidnightcoders.com/weborb/aboutWeborb.htm

Платформа для разработки богатых клиентских приложений Ajax и Flash с возможностью их последующего объединения с объектами Java и Web-службами XML. Включает клиентскую библиотеку Rich Client System

(www. themidnightcoders, сот/гсэ/index.htm), которая предлагает простой (в одну строку) API, позволяющий связываться с любым методом любого объекта Java, Web-службы XML или Enterprise JavaBean. Предлагает специальный API для обработки результатов запросов базы данных; серверный код может возвращать элементы DataSets или DataTables, а клиент представит их как специальный объект JavaScript Recordset. Данный объект позволяет извлекать названия столбцов и построчные данные.

x

Майте *Фостер (Mike Foster), 2005*
Продукт с открытым исходным кодом
www.cross-browser.com

Классическая библиотека DHTML, предлагающая поддержку в различных браузерах анимации, стилевого оформления, событий и других распространенных функциональных возможностей.

XAJAX

Дою. Макс Уилсон (J. Max Wilson), 2005 /
Продукт с открытым исходным кодом
http://xajax.sf.net

Передает вызовы JavaScript непосредственно функциям PHP. Для вызова сценария PHP использует суррогат JavaScript.

x-Desktop

Продукт с открытым исходным кодом (GPL)
www.x-desktop.org/

Проект включает библиотеку для разработки "тонких" клиентских приложений с использованием браузера. Помогает разработчикам создавать графические интерфейсы для приложений глобальных, внутренних и внешних сетей. Для работы продукта не требуются никакие модули, необходим только браузер. Поддерживает все операционные системы с браузером DOM 2/JavaScript; предлагает простой интерфейс с хорошей документацией; имеет настраиваемые сменные оболочки.

XHConn

Вред Фульц (Brad Fults), 2005
http://xkr.us/code/JavaScript/XHConn
"Тонкая" интерфейсная оболочка объекта XMLHttpRequest.

Предметный указатель

Научно-популярное издание

Дейв Крейн, Эрик Паскарелло, Даррен Джеймс

Ajax в действии

Литературный редактор *И.А. Попова*
Верстка *А.В. Назаренко*
Художественные редакторы *Т.А. Тараброва, С.А. Чернокозинскш*
Корректор *О.В. Мишутина*

Издательский дом "Вильяме"
101509, Москва, ул. Лесная, д. 43, стр. 1

Подписано в печать 21.04.2006. Формат *70X100/16.*
Гарнитура Times. Печать офсетная.
Усл. печ. л. 51,6. Уч.-изд. л. 36,86.
Тираж 2000 экз. Заказ № *1385,*

Отпечатано с диапозитивов
в ОАО "Печатный двор" им. А, М. Горького.
197110, Санкт-Петербург, Чкаловский пр., 15.

K79
681.3.07

Издательский дом "Вильяме"

Зав. редакцией *С.Н. Тригуб*

Перевод с английского и редакция *В.В. Вейтмшш* и *А.В. Назаренко*

По общим вопросам обращайтесь в Издательский дом "Вильяме" по адресу:
info@williamspiiblishing.com. http://ww\v.wi]Hamspubl is hing.com
115419, Москва, а/я 783; 03150. Киев, а/я 152

in, Дейв, **Паскарслло,** Эрик, Джеймс, Даррен.

Ajax в действии. : Пер. с англ. —- М. : Издательский дом "Вильяме", 2006. — 640 с. : ил. — Парал. тит. англ.

ISBN 5-8459-1034-X (рус.)

В этой книге описан новый подход к созданию Web-приложений, известный как Ajax. Авторы осматривают составные части Ajax: JavaScript, CSS, DOM и объект XMLHttpRequest. Кроме iro, в книге нашли отражение вопросы управления кодом, взаимодействия клиента и сервера применения архитектуры "модель-представленис-коцтроллер" на разных уровнях приложения, ататель также найдет сведения о защите и производительности — важных характеристиках, суественно влияющих на популярность любого продукта. Рассматриваемые вопросы иллюстрируся примерами практического использования Ajax. В приложениях содержится дополнительная формация об инструментальных средствах, о языке JavaScript и библиотеках. Материал книги ложен на высоком уровне и будет полезен спеииалистам высокой и средней квалификации.

ББК 32.973.26-018.2.75

5-8459-1034-X (рус.)
1-9323-9461-3 Ошгл.)