Создание активных веб-страниц

5-е издание
Включает Ajax и DOM

JavaScript

Подробное руководство

Символ®

O'REILLY®

Дэвид Флэнаган

JavaScript

The Definitive Guide

Fifth Edition

David Flanagan

O'REILLY®

JavaScript

Подробное руководство

Пятое издание

Дэвид Флэнаган

Санкт-Петербург — Москва
2008

Дэвид Флэнаган

JavaScript. Подробное руководство, 5-е издание

Перевод А. Киселева

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Научный редактор	*О. Цилюрик*
Редактор	*А. Жданов*
Корректор	*С. Минин*
Верстка	*Д. Орлова*

Флэнаган Д.

JavaScript. Подробное руководство. – Пер. с англ. – СПб: Символ-Плюс, 2008. – 992 с., ил.

ISBN-10: 5-93286-103-7
ISBN-13: 978-5-93286-103-5

Пятое издание бестселлера «JavaScript. Подробное руководство» полностью обновлено. Рассматриваются взаимодействие с протоколом HTTP и применение технологии Ajax, обработка XML-документов, создание графики на стороне клиента с помощью тега <canvas>, пространства имен в JavaScript, необходимые для разработки сложных программ, классы, замыкания, Flash и встраивание сценариев JavaScript в Java-приложения.

Часть I знакомит с основами JavaScript. В части II описывается среда разработки сценариев, предоставляемая веб-броузерами. Многочисленные примеры демонстрируют, как генерировать оглавление HTML-документа, отображать анимированные изображения DHTML, автоматизировать проверку правильности заполнения форм, создавать всплывающие подсказки с использованием Ajax, как применять XPath и XSLT для обработки XML-документов, загруженных с помощью Ajax. Часть III – обширный справочник по базовому JavaScript (классы, объекты, конструкторы, методы, функции, свойства и константы, определенные в JavaScript 1.5 и ECMAScript v3). Часть IV – справочник по клиентскому JavaScript (API веб-броузеров, стандарт DOM API Level 2 и недавно появившиеся стандарты: объект XMLHttpRequest и тег <canvas>).

ISBN-10: 5-93286-103-7
ISBN-13: 978-5-93286-103-5
ISBN 0-596-10199-6 (англ)

Издательство «Символ-Плюс». 199034, Санкт-Петербург, 16 линия, 7,
тел. (812) 324-5353, www.symbol.ru. Лицензия ЛП N 000054 от 25.12.98.
Налоговая льгота – общероссийский классификатор продукции
ОК 005-93, том 2; 953000 – книги и брошюры.

Подписано в печать 14.02.2008. Формат 70×100 1/16 . Печать офсетная.
Объем 62 печ. л. Тираж 2000 экз. Заказ N
Отпечатано с готовых диапозитивов в ГУП «Типография «Наука»
199034, Санкт-Петербург, 9 линия, 12.

Эта книга посвящается всем,
кто учит жить мирно и противостоит насилию.

Оглавление

Предисловие

После выхода из печати четвертого издания книги «JavaScript. Подробное руководство» объектная модель документов (Document Object Model, DOM), представляющая собой основу прикладного программного интерфейса (Application Programming Interface, API) для сценариев на языке JavaScript™, исполняющихся на стороне клиента, была реализована достаточно полно, если не полностью, в веб-броузерах. Это означает, что разработчики веб-приложений получили в свое распоряжение универсальный прикладной программный интерфейс для работы с содержимым веб-страниц на стороне клиента и зрелый язык (JavaScript 1.5), остававшийся стабильным на протяжении последующих лет.

Сейчас интерес к JavaScript опять начинает расти. Теперь разработчики используют JavaScript для создания сценариев, работающих по протоколу HTTP, управляющих XML-данными и даже динамически создающих графические изображения в веб-броузере. Многие программисты с помощью JavaScript создают большие программы и применяют достаточно сложные технологии программирования, такие как замыкания и пространства имен. Пятое издание полностью пересмотрено с позиций вновь появившихся технологий Ajax и Web 2.0.

Что нового в пятом издании

В первой части книги, «Основы JavaScript», была расширена глава 8, описывающая функции; в нее включен материал, охватывающий замыкания и вложенные функции. Информация о порядке создания собственных классов была дополнена и выделена в отдельную главу 9. Глава 10 – это еще одна новая глава, которая содержит сведения о пространствах имен, являющихся основой для разработки модульного программного кода многократного использования. Наконец, глава 12 демонстрирует, как применять JavaScript при разработке сценариев на языке Java. Здесь показано, как встраивать интерпретатор JavaScript в приложения на Java 6, как использовать JavaScript для создания Java-объектов и как вызывать методы этих объектов.

Во второй части книги, «Клиентский язык JavaScript», описываются прежняя (уровня 0) объектная модель документа и стандарт DOM консорциума W3C. Поскольку в настоящее время модель DOM имеет универсальные реализации, отпала необходимость в двух отдельных главах, где в предыдущем издании описывались приемы работы с документами. Вторая часть книги подверглась самым существенным изменениям; в нее включен следующий новый материал:

- Глава 19 «Cookies и механизм сохранения данных на стороне клиента» дополнена новой информацией о cookies и сведениями о методиках программирования, применяемых на стороне клиента.

- Глава 20 «Работа с протоколом HTTP» описывает, как выполнять HTTP-запросы с помощью такого мощного инструмента, как объект `XMLHttpRequest`, который делает возможным создание Ajax-подобных веб-приложений.
- Глава 21 «JavaScript и XML» демонстрирует, как средствами JavaScript организовать создание, загрузку, синтаксический разбор, преобразование, выборку, сериализацию и извлечение данных из XML-документов. Кроме того, рассматривается расширение языка JavaScript, получившее название E4X.
- Глава 22 «Работа с графикой на стороне клиента» описывает графические возможности языка JavaScript. Здесь рассматриваются как простейшие способы создания анимированных изображений, так и достаточно сложные приемы работы с графикой с использованием ультрасовременного тега `<canvas>`. Кроме того, здесь говорится о создании графики на стороне клиента средствами подключаемых SVG-, VML-, Flash- и Java-модулей.
- Глава 23 «Сценарии с Java-апплетами и Flash-роликами» рассказывает о подключаемых Flash- и Java-модулях. В этой главе объясняется, как создавать Flash-ролики и Java-апплеты.

Часть III книги представляет собой справочник по прикладному интерфейсу базового языка JavaScript. Изменения в этой части по сравнению с предыдущим изданием весьма незначительные, что обусловлено стабильностью API. Если вы читали 4-е издание, вы найдете эту часть книги удивительно знакомой.

Существенные изменения коснулись организации справочного материала, описывающего прикладной интерфейс объектной модели документа (DOM API), который ранее был выделен в самостоятельную часть отдельно от описания клиентского языка JavaScript. Теперь же оставлена единственная часть со справочной информацией, относящейся к клиентскому языку JavaScript. Благодаря этому отпала необходимость читать описание объекта `Document` в одной части, а затем искать описание объекта `HTMLDocument` в другой. Справочный материал об интерфейсах модели DOM, которые так и не были достаточно полно реализованы в броузерах, попросту убран. Так, интерфейс `NodeIterator` не поддерживается в броузерах, поэтому его описание из этой книги исключено. Кроме того, акцент смещен от сложных формальных определений DOM-интерфейсов к JavaScript-объектам, которые являются фактической реализацией этих интерфейсов. Например, метод `getComputedStyle()` теперь описывается не как метод интерфейса `AbstractView`, а как метод объекта `Window`, что логичнее. Для JavaScript-программистов, создающих клиентские сценарии, нет серьезных оснований вникать в особенности интерфейса `AbstractView`, поэтому его описание было убрано из справочника. Все эти изменения сделали справочную часть книги, посвященную клиентскому языку JavaScript, более простой и удобной.

Порядок работы с книгой

Глава 1 представляет собой введение в язык JavaScript. Остальная часть книги делится на четыре части. Первая часть, которая непосредственно следует за главой 1, описывает основы языка JavaScript. Главы со 2 по 6 содержат достаточно скучный материал, тем не менее прочитать его совершенно необходимо, т. к. он охватывает самые основы, без знания которых невозможно начать изучение нового языка программирования:

- Глава 2 «Лексическая структура» описывает основные языковые конструкции.
- Глава 3 «Типы данных и значения» рассказывает о типах данных, поддерживаемых языком JavaScript.
- Глава 4 «Переменные» охватывает темы переменных, областей видимости переменных и всего, что с этим связано.
- Глава 5 «Выражения и операторы» описывает выражения языка JavaScript и документирует каждый оператор, поддерживаемый этим языком программирования. Поскольку синтаксис JavaScript основан на синтаксисе языка Java, который, в свою очередь, очень многое заимствовал из языков C и C++, программисты, имеющие опыт работы с этими языками, могут лишь вкратце ознакомиться с содержимым этой главы.
- Глава 6 «Инструкции» описывает синтаксис и порядок использования каждой JavaScript-инструкции. Программисты, имеющие опыт работы с языками C, C++ и Java, могут пропустить не все, но некоторые разделы этой главы.

Последующие шесть глав первой части содержат куда более интересные сведения. Они также описывают основы языка JavaScript, но охватывают те его части, которые едва ли вам знакомы, даже если вам приходилось писать на языке C или Java. Если вам требуется настоящее понимание JavaScript, к изучению материала этих глав следует подходить с особой тщательностью.

- Глава 7 «Объекты и массивы» описывает объекты и массивы языка JavaScript.
- Глава 8 «Функции» рассказывает о том, как определяются функции, как они вызываются, каковы их отличительные особенности в языке JavaScript.
- Глава 9 «Классы, конструкторы и прототипы» касается вопросов объектно-ориентированного программирования на языке JavaScript. Рассказывается о том, как определяются функции-конструкторы для новых классов объектов и как работает механизм наследования на основе прототипов. Кроме того, продемонстрирована возможность эмулирования традиционных идиом объектно-ориентированного программирования на языке JavaScript.
- Глава 10 «Модули и пространства имен» показывает, как определяются пространства имен в JavaScript-объектах, и описывает некоторые практические приемы, позволяющие избежать конфликтов имен в модулях.
- Глава 11 «Шаблоны и регулярные выражения» рассказывает о том, как использовать регулярные выражения в языке JavaScript для выполнения операций поиска и замены по шаблону.
- Глава 12 «Разработка сценариев для Java-приложений» демонстрирует возможность встраивания интерпретатора JavaScript в Java-приложения и рассказывает, как JavaScript-программы, работающие внутри Java-приложений, могут обращаться к Java-объектам. Эта глава представляет интерес только для тех, кто программирует на языке Java.

Часть II книги описывает реализацию JavaScript в веб-броузерах. Первые шесть глав рассказывают об основных характеристиках клиентского JavaScript:

- Глава 13 «JavaScript в веб-броузерах» рассказывает об интеграции JavaScript в веб-броузеры. Здесь броузеры рассматриваются как среда программирования и описываются различные варианты встраивания программного JavaScript-кода в веб-страницы для исполнения его на стороне клиента.

- Глава 14 «Работа с окнами броузера» описывает центральный элемент клиентского языка JavaScript – объект `Window` и рассказывает, как использовать этот объект для управления окнами броузера.
- Глава 15 «Работа с документами» описывает объект `Document` и рассказывает, как из JavaScript управлять содержимым, отображаемым в окне броузера. Эта глава является наиболее важной во второй части.
- Глава 16 «CSS и DHTML» рассказывает о порядке взаимодействия между JavaScript-кодом и таблицами CSS-стилей. Здесь показано, как средствами JavaScript изменять стили, вид и положение элементов HTML-документа, создавая визуальные эффекты, известные как DHTML.
- Глава 17 «События и обработка событий» описывает события и порядок их обработки, что является немаловажным для программ, ориентированных на взаимодействие с пользователем.
- Глава 18 «Формы и элементы форм» посвящена тому, как работать с HTML-формами и отдельными элементами форм. Данная глава является логическим продолжением главы 15, но обсуждаемая тема настолько важна, что была выделена в самостоятельную главу.

Вслед за этими шестью главами следуют пять глав, содержащих более узкоспециализированный материал:

- Глава 19 «Cookies и механизм сохранения данных на стороне клиента» охватывает вопросы хранения данных на стороне клиента для последующего использования. В этой главе показано, как средствами HTTP манипулировать cookies и как сохранять их с помощью соответствующих инструментов Internet Explorer и подключаемого Flash-модуля.
- Глава 20 «Работа с протоколом HTTP» демонстрирует, как управлять протоколом HTTP из JavaScript-сценариев, как с помощью объекта `XMLHttpRequest` отправлять запросы веб-серверам и получать от них ответы. Данная возможность является краеугольным камнем архитектуры веб-приложений, известной под названием Ajax.
- Глава 21 «JavaScript и XML» описывает, как средствами JavaScript создавать, загружать, анализировать, преобразовывать и сериализовать XML-документы, а также как извлекать из них данные.
- Глава 22 «Работа с графикой на стороне клиента» рассказывает о средствах JavaScript, ориентированных на работу с графикой. Здесь рассматриваются как простейшие способы создания анимированных изображений, так и достаточно сложные приемы работы с графикой с использованием форматов SVG (Scalable Vector Graphics – масштабируемая векторная графика) и VML (Vector Markup Language – векторный язык разметки), тега `<canvas>` и подключаемых Flash- и Java-модулей.
- Глава 23 «Сценарии с Java-апплетами и Flash-роликами» показывает, как организовать взаимодействие JavaScript-кода с Java-апплетами и Flash-роликами. Кроме того, в ней рассказывается, как обращаться к JavaScript-коду из Java-апплетов и Flash-роликов.

Третья и четвертая части содержат справочный материал соответственно по базовому и клиентскому языкам JavaScript. Здесь приводятся описания объектов, методов и свойств в алфавитном порядке.

Типографские соглашения

В этой книге приняты следующие соглашения:

Курсив

Обозначает первое появление термина. Курсив также применяется для выделения адресов электронной почты, веб-сайтов, FTP-сайтов, имен файлов и каталогов, групп новостей.

Моноширинный шрифт

Применяется для форматирования программного кода на языке JavaScript, HTML-листингов и вообще всего, что непосредственно набирается на клавиатуре при программировании.

Моноширинный жирный

Используется для выделения текста командной строки, который должен быть введен пользователем.

Моноширинный курсив

Обозначает аргументы функций и элементы, которые в программе необходимо заменить реальными значениями.

Клавиши и элементы пользовательского интерфейса, такие как кнопка Назад или меню Сервис, выделены шрифтом OfficinaSansC.

Использование программного кода примеров

Данная книга призвана оказать помощь в решении ваших задач. Вы можете свободно использовать примеры программного кода из этой книги в своих приложениях и в документации. Вам не нужно обращаться в издательство за разрешением, если вы не собираетесь воспроизводить существенные части программного кода. Например, если вы разрабатываете программу и задействуете в ней несколько отрывков программного кода из книги, вам не нужно обращаться за разрешением. Однако в случае продажи или распространения компакт-дисков с примерами из этой книги вам необходимо получить разрешение от издательства O'Reilly. При цитировании данной книги или примеров из нее и при ответе на вопросы получение разрешения не требуется. При включении существенных объемов программного кода примеров из этой книги в вашу документацию вам *необходимо* получить разрешение издательства.

Мы приветствуем, но не требуем добавлять ссылку на первоисточник при цитировании. Под ссылкой на первоисточник мы подразумеваем указание авторов, издательства и ISBN. Например: «JavaScript: The Definitive Guide, by David Flanagan. Copyright 2006 O'Reilly Media, Inc., 978-0-596-10199-2».

За получением разрешения на использование значительных объемов программного кода примеров из этой книги обращайтесь по адресу *permissions@oreilly.com*.

Отзывы и предложения

С вопросами и предложениями, касающимися этой книги, обращайтесь в издательство:

O'Reilly Media
1005 Gravenstein Highway North
Sebastopol, CA 95472
(800) 998-9938 (в Соединенных Штатах Америки или в Канаде)
(707) 829-0515 (международный)
(707) 829-0104 (факс)

Список опечаток, файлы с примерами и другую дополнительную информацию вы найдете на сайте книги по адресу:

http://www.oreilly.com/catalog/jscript5

Кроме того, все примеры, описываемые в книге, можно загрузить с сайта авторов:

http://www.davidflanagan.com/javascript5

Свои пожелания и вопросы технического характера отправляйте по адресу:

bookquestions@oreilly.com

Дополнительную информацию о книгах, обсуждения, центр ресурсов издательства O'Reilly вы найдете на сайте:

http://www.oreilly.com

Safari® Enabled

Если на обложке технической книги есть пиктограмма «Safari® Enabled», это означает, что книга доступна в Сети через O'Reilly Network Safari Bookshelf.

Safari предлагает намного лучшее решение, чем электронные книги. Это виртуальная библиотека, позволяющая без труда находить тысячи лучших технических книг, вырезать и вставлять примеры кода, загружать главы и находить быстрые ответы, когда требуется наиболее верная и свежая информация. Она свободно доступна по адресу *http://safari.oreilly.com*.

Благодарности

Брендан Эйх (Brendan Eich) из Mozilla – один из создателей и главный новатор JavaScript. Я и другие JavaScript-программисты в неоплатном долгу перед ним за разработку JavaScript и за то, что в его сумасшедшем графике нашлось время для ответов на наши вопросы, причем он даже требовал еще вопросов. Брендан не только терпеливо отвечал на мои многочисленные вопросы, но и прочитал первое и третье издания этой книги и дал очень полезные комментарии к ним.

Это руководство получило благословение первоклассных технических рецензентов, комментарии которых весьма способствовали улучшению этой книги. Аристотель Пагальцис (Aristotle Pagatzis) (*http://plasmasturm.org*) рецензировал новый материал о функциях, а также главы, в которых в этом издании приводится описание классов и пространств имен. Он с особым тщанием просмотрел программный код примеров и дал весьма ценные комментарии. Дуглас Крокфорд (Douglas Crockford) (*http://www.crockford.com*) рецензировал новый материал о функциях и классах. Норрис Бойд (Norris Boyd), создатель интерпретатора Rhino для JavaScript, рецензировал главу, в которой описывается механизм

встраивания JavaScript-кода в Java-приложения. Питер-Пауль Кох (Peter-Paul Koch) (*http://www.quirksmode.org*), Кристиан Хейльманн (Christian Heilmann) (*http://www.wait-till-i.com*) и Кен Купер (Ken Cooper) рецензировали главы этой книги, связанные с технологией Ajax. Тодд Дихендорф (Todd Ditchendorf) (*http://www.ditchnet.org*) и Джефф Штернс (Geoff Stearns) (*http://blog.deconcept.com*) рецензировали главу, описывающую приемы работы с графикой на стороне клиента. Тодд был настолько любезен, что занялся поиском и исправлением ошибок в программном коде примеров, а Джефф помог разобраться в технологиях Flash и ActionScript. Наконец, Сандерс Клейнфельд (Sanders Kleinfeld) рецензировал всю книгу, уделяя особое внимание деталям. Его предложения и исправления сделали эту книгу более точной и понятной. Приношу слова искренней признательности каждому, кто занимался рецензированием книги. Любые ошибки, которые вам встретятся в книге, лежат на моей совести.

Я очень признателен рецензентам четвертого издания книги. Валдемар Хорват (Waldermar Horwat) из Netscape рецензировал новый материал по JavaScript 1.5. Материал по W3C DOM проверен Филиппом Ле Хегаре (Philippe Le Hegaret) из W3C, Питером-Паулем Кохом (Peter-Paul Koch), Диланом Шиманом (Dylan Schiemann) и Джеффом Ятсом (Jeff Yates). Джозеф Кесселман (Joseph Kesselman) из IBM Research ничего не рецензировал, но очень помог мне, отвечая на вопросы по W3C DOM.

Третье издание книги рецензировалось Бренданом Эйхом, Валдемаром Хорватом и Вайдуром Аппарао (Vidur Apparao) из Netscape, Германом Вентером (Herman Venter) из Microsoft, двумя независимыми JavaScript-разработчиками – Джеем Ходжесом (Jay Hodges) и Анжело Сиригосом (Angelo Sirigos). Дэн Шейфер (Dan Shafer) из CNET Builder.com выполнил некоторую предварительную работу по третьему изданию. Его материал не нашел применения в этом издании, но принадлежащие ему идеи и общие принципы принесли большую пользу. Норрис Бойд (Norris Boyd) и Скотт Фурман (Scott Furman) из Netscape, а также Скотт Айзекс (Scott Issacs) из Microsoft нашли время, чтобы поговорить со мной о будущем стандарте DOM. И наконец, доктор Танкред Хиршманн (Dr. Tankred Hirschmann) показал глубокое понимание хитросплетений JavaScript 1.2.

Второе издание много выиграло от помощи и комментариев Ника Томпсона (Nick Thompson) и Ричарда Якера (Richard Yaker) из Netscape, доктора Шона Каценбергера (Dr. Shon Katzenberger), Ларри Салливана (Larry Sullivan) и Дэйва С. Митчелла (Dave C. Mitchell) из Microsoft, Линн Роллинс (Lynn Rollins) из R&B Communications. Первое издание рецензировалось Нилом Беркманом (Neil Berkman) из Bay Networks, Эндрю Шульманом (Andrew Schulman) и Терри Алленом (Terry Allen) из O'Reilly & Associates.

Эта книга стала лучше еще и благодаря многочисленным редакторам, которые над ней работали. Деб Камерон (Deb Cameron) – редактор этого издания, он особое внимание уделял обновлению и удалению устаревшего материала. Паула Фергюсон (Paula Ferguson) – редактор третьего и четвертого изданий. Френк Уиллисон (Frank Willison) редактировал второе издание, а Эндрю Шульман – первое.

И наконец, мои благодарности Кристи – как всегда и за очень многое.

Дэвид Флэнаган
http://www.davidflanagan.com
апрель 2006

II

Клиентский JavaScript

В данной части книги в главах с 13 по 23 язык JavaScript описан в том виде, в котором он реализован в веб-броузерах. В этих главах вводится много новых JavaScript-объектов, представляющих веб-броузер, а также содержимое HTML- и XML-документов.

- Глава 13 «JavaScript в веб-броузерах»
- Глава 14 «Работа с окнами броузера»
- Глава 15 «Работа с документами»
- Глава 16 «CSS и DHTML»
- Глава 17 «События и обработка событий»
- Глава 18 «Формы и элементы форм»
- Глава 19 «Cookies и механизм сохранения данных на стороне клиента»
- Глава 20 «Работа с протоколом HTTP»
- Глава 21 «JavaScript и XML»
- Глава 22 «Работа с графикой на стороне клиента»
- Глава 23 «Сценарии с Java-апплетами и Flash-роликами»

20

Работа с протоколом HTTP

Протокол передачи гипертекста (Hypertext Transfer Protocol, HTTP) определяет, как веб-броузеры должны запрашивать документы, как они должны передавать информацию веб-серверам и как веб-серверы должны отвечать на эти запросы и передачи. Вполне очевидно, что веб-броузеры очень много работают с протоколом HTTP. Тем не менее, как правило, сценарии не работают с протоколом HTTP, когда пользователь щелкает по ссылке, отправляет форму или вводит URL в адресной строке. В то же время обычно, хотя и не всегда, JavaScript-код *способен* работать с протоколом HTTP.

HTTP-запросы могут инициироваться, когда сценарий устанавливает значение свойства `location` объекта `Window` или вызывает метод `submit()` объекта `Form`. В обоих случаях броузер загружает в окно новую страницу, затирая любой исполнявшийся сценарий. Такого рода взаимодействие с протоколом HTTP может быть вполне оправданным в веб-страницах, состоящих из нескольких фреймов, но в этой главе мы будем говорить совсем о другом. Здесь мы рассмотрим такое взаимодействие JavaScript-кода с веб-сервером, при котором веб-броузер не перезагружает текущую веб-страницу.

Теги `<img>`, `<frame>` и `<script>` имеют свойство `src`. Когда сценарий записывает в это свойство URL-адрес, инициируется HTTP-запрос GET и выполняется загрузка содержимого с этого URL-адреса. Таким образом, сценарий может отправлять информацию веб-серверу, добавляя ее в виде строки запроса в URL-адрес изображения и устанавливая свойство `src` элемента `<img>`. В ответ на этот запрос веб-сервер должен вернуть некоторое изображение, которое, например, может быть невидимым: прозрачным и размером 1×1 пиксел.[1]

---

[1] Такие изображения иногда называют *веб-жучками* (*web bugs*). Они пользуются дурной славой из-за проблем с безопасностью, т. к. могут применяться для обмена информацией (подсчета числа посещений и анализа трафика) со сторонним сервером (не тем, откуда была загружена страница). Если же веб-страница устанавливает свойство `src` изображения для передачи информации обратно на тот сервер, с которого она была загружена, особых проблем с безопасностью не возникает.

Теги <iframe> достаточно недавно появились в HTML и они более универсальны, чем теги <img>, т. к. позволяют веб-серверу вернуть результат не в виде двоичного файла с изображением, а в удобочитаемом виде, который может быть проверен сценарием. При использовании тега <iframe> сценарий сначала добавляет в URL-адрес информацию, предназначенную для веб-сервера, а затем записывает этот URL-адрес в свойство src тега <iframe>. Сервер создает HTML-документ, содержащий ответ на запрос, и отправляет его обратно веб-броузеру, который выводит ответ в теге <iframe>. При этом элемент <iframe> необязательно должен быть видимым для пользователя – он может быть сокрыт, например средствами таблиц стилей. Сценарий может проанализировать ответ сервера, выполнив обход документа в элементе <iframe>. Обратите внимание: взаимодействие с документом ограничивается политикой общего происхождения, о которой рассказывается в разделе 13.8.2.

Даже изменение свойства src тега <script> может использоваться для инициирования динамического HTTP-запроса. Использование тегов <script> для взаимодействия с протоколом HTTP выглядит особенно привлекательно, потому что когда ответ сервера принимает форму JavaScript-кода, он не требует дополнительного анализа – интерпретатор JavaScript просто исполняет его.

Несмотря на то, что существует возможность использования тегов <img>, <iframe> и <script> для взаимодействия с протоколом HTTP, реализовать такую возможность на практике переносимым образом гораздо сложнее, чем это выглядит на словах, и в этой главе мы сосредоточимся на другом, более мощном способе достижения тех же результатов. Объект XMLHttpRequest прекрасно поддерживается всеми современными броузерами и предоставляет полный доступ к протоколу HTTP, включая возможность отправлять запросы методами POST и HEAD в дополнение к обычному запросу методом GET. Объект XMLHttpRequest может возвращать ответ веб-сервера синхронно или асинхронно, в виде простого текста или в виде DOM-документа. Несмотря на свое название объект XMLHttpRequest не ограничивается использованием XML-документов – он в состоянии принимать любые текстовые документы. Объект XMLHttpRequest является ключевым элементом архитектуры веб-приложений, известной как Ajax[1]. Об Ajax-приложениях мы поговорим после ознакомления с тем, как работает объект XMLHttpRequest.

В конце главы мы вернемся к теме использования тега <script> для организации взаимодействия с протоколом HTTP, и там я продемонстрирую, как можно применить эту методику, когда объект XMLHttpRequest недоступен.

20.1. Использование объекта XMLHttpRequest

Процесс взаимодействия с протоколом HTTP с использованием объекта XMLHttpRequest делится на три этапа:

- Создание объекта XMLHttpRequest.

---

[1] Эта глава – лишь введение в предмет; исчерпывающее описание архитектуры Ajax с детальными примерами реализации можно найти, например, в книгах:
Закас, Мак-Пик, Фосетт «Ajax для профессионалов». – Пер. с англ. – СПб.: Символ-Плюс, 2007;
Дари, Бринзаре, Черчез-Тоза, Бусика «AJAX и PHP. Разработка динамических веб-приложений». – Пер. с англ. – СПб.: Символ-Плюс, 2007. – *Примеч. науч. ред.*

- Определение и передача HTTP-запроса на веб-сервер.
- Синхронный или асинхронный прием ответа сервера.

Каждый из этих этапов более подробно рассматривается в следующих подразделах.

Все примеры этой главы представляют собой часть одного большого модуля. В них определяются вспомогательные функции, входящие в пространство имен `HTTP` (см. главу 10). Однако в приведенных здесь примерах вы не найдете программный код, фактически создающий пространство имен. В пакет с примерами, который можно загрузить с сайта издательства, входит файл с именем `http.js`, который включает в себя программный код создания пространства имен, но вы можете в рассматриваемые здесь примеры просто добавить одну строку:

```
var HTTP = {};
```

20.1.1. Создание объекта запроса

Объект `XMLHttpRequest` никогда не стандартизировался, и процесс его создания в Internet Explorer отличается от такового в других платформах. (К счастью, прикладной интерфейс для работы с объектом `XMLHttpRequest` после его создания одинаков для всех платформ.)

В большинстве броузеров объект `XMLHttpRequest` создается простым вызовом конструктора:

```
var request = new XMLHttpRequest();
```

В IE до появления версии 7 конструктор `XMLHttpRequest()` попросту отсутствовал. В IE 5 и 6 `XMLHttpRequest` представляет собой объект ActiveX и должен создаваться обращением к конструктору `ActiveXObject()`, которому передается имя создаваемого объекта:

```
var request = new ActiveXObject("Msxml2.XMLHTTP");
```

К сожалению, в разных версиях библиотеки XML HTTP компании Microsoft объект имеет различные имена. В зависимости от версии библиотеки, установленной у клиента, иногда приходится использовать следующий программный код для создания объекта:

```
var request = new ActiveXObject("Microsoft.XMLHTTP");
```

Пример 20.1 представляет собой платформонезависимую вспомогательную функцию с именем `HTTP.newRequest()`, которая создает объекты `XMLHttpRequest`.

Пример 20.1. Вспомогательная функция HTTP.newRequest( )

```
// Попробуем использовать следующие функции, создающие объект XMLHttpRequest.
HTTP._factories = [
    function() { return new XMLHttpRequest(); },
    function() { return new ActiveXObject("Msxml2.XMLHTTP"); },
    function() { return new ActiveXObject("Microsoft.XMLHTTP"); }
];

// Когда будет обнаружена работоспособная функция, она будет сохранена здесь.
HTTP._factory = null;
```

```
// Создает и возвращает новый объект XMLHttpRequest.
//
// При первом обращении к функции опробываются все функции из списка, пока
// не будет найдена та, что вернет непустое значение и не возбудит исключение.
// После того как будет обнаружена работоспособная функция, ссылка на нее
// запоминается для последующего использования.
//
HTTP.newRequest = function() {
    if (HTTP._factory != null) return HTTP._factory();

    for(var i = 0; i < HTTP._factories.length; i++) {
        try {
            var factory = HTTP._factories[i];
            var request = factory();
            if (request != null) {
                HTTP._factory = factory;
                return request;
            }
        }
        catch(e) {
            continue;
        }
    }

    // Если попав сюда, сценарию не удалось обнаружить подходящую функцию для создания
    // объекта, необходимо возбудить исключение в этом и всех последующих вызовах.
    HTTP._factory = function() {
        throw new Error("Объект XMLHttpRequest не поддерживается");
    }
    HTTP._factory(); // Возбудить исключение
}
```

20.1.2. Отправка запроса

После того как объект `XMLHttpRequest` создан, начинается следующий этап – отправка запроса веб-серверу. Этот процесс сам по себе также состоит из нескольких этапов. В первую очередь нужно вызвать метод `open()`, которому передается URL запроса и *метод* выполнения HTTP-запроса. Большая часть HTTP-запросов выполняется методом GET, который просто загружает содержимое по заданному URL-адресу. Другой, не менее полезный метод – POST; этот метод используется в основном HTML-формами: он позволяет включать в текст запроса имена и значения переменных. Еще один интересный метод – HEAD: он просто запрашивает у сервера заголовки, соответствующие заданному URL-адресу. Это позволяет сценариям проверять, например, время последнего изменения документа без загрузки содержимого этого документа. Указать метод и URL-адрес запроса можно следующим образом:

```
request.open("GET", url, false);
```

По умолчанию метод `open()` настраивает объект `XMLHttpRequest` на выполнение асинхронного запроса. Если передать ему в третьем аргументе значение `false`, запрос будет выполнен синхронно. Вообще, предпочтительнее использовать

асинхронные запросы, но синхронные запросы выполняются проще, поэтому наше рассмотрение мы начнем с них.

Помимо третьего необязательного аргумента метод `open()` может принимать имя и пароль в четвертом и пятом аргументах. Они используются для выполнения запроса к серверу, требующему авторизации.

Метод `open()` не отправляет запрос, он просто сохраняет свои аргументы для последующего использования, когда будет производиться фактическая отправка запроса. Прежде чем отправить запрос, необходимо настроить некоторые заголовки запроса. Вот несколько примеров:[1]

```
request.setRequestHeader("User-Agent", "XMLHttpRequest");
request.setRequestHeader("Accept-Language", "en");
request.setRequestHeader("If-Modified-Since", lastRequestTime.toString());
```

Обратите внимание: веб-броузер автоматически добавляет к создаваемому запросу все необходимые cookies. Явно настраивать заголовок `"Cookie"` может потребоваться только при необходимости отправить на сервер подложный cookie.

Наконец, после создания объекта запроса вызовом метода `open()` и установки необходимых заголовков можно выполнить отправку запроса:

```
request.send(null);
```

В качестве аргумента функции `send()` передается тело запроса. Для HTTP-запросов GET всегда используется значение `null`. Однако для запросов POST аргумент должен содержать данные формы, отправляемой на сервер (см. пример 20.5). Пока мы просто будем передавать значение `null`. (Обратите внимание: значение `null` должно передаваться обязательно. Объект `XMLHttpRequest` является клиентским, по крайней мере в броузере Firefox, его методы не допускают отсутствия аргументов, что вполне допустимо в обычных JavaScript-функциях.)

20.1.3. Получение синхронного ответа

Объект `XMLHttpRequest` хранит не только информацию о HTTP-запросе, но и ответ сервера. Если методу `open()` третьим аргументом передается значение `false`, метод `send()` выполнит запрос синхронно: он не вернет управление до тех пор, пока не будет получен ответ сервера.[2]

Метод `send()` не возвращает код состояния. После того как он вернет управление, можно проверить код состояния HTTP, возвращаемый сервером в свойстве `status` объекта запроса. Возможные значения кода состояния определяются протоколом HTTP. Код состояния 200 означает успешное завершение запроса и доступность ответа. В то же время код состояния 404 означает ошибку «не найдено», которая возникает в случае, когда указанный URL-адрес не существует.

---

[1] Подробное описание протокола HTTP выходит за рамки темы этой книги. За дополнительной информацией об этих и других заголовках, используемых при выполнении HTTP-запросов, обращайтесь к техническому описанию протокола HTTP.

[2] Объект `XMLHttpRequest` обладает поистине удивительными возможностями, но его прикладной программный интерфейс продуман недостаточно. Например, логическое значение, определяющее синхронное или асинхронное поведение, в действительности должно было бы быть аргументом метода `send()`.

Объект XMLHttpRequest возвращает ответ сервера в виде строки, доступной через свойство responseText объекта запроса. Если ответ представляет собой XML-документ, к нему можно обращаться как к DOM-объекту Document через свойство responseXML. Обратите внимание: чтобы объект XMLHttpRequest воспринял и преобразовал ответ сервера в объект Document, сервер должен явно идентифицировать его как XML-документ, указав MIME-тип "text/xml".

Когда запрос выполняется синхронно, программный код, следующий за вызовом метода send(), обычно выглядит как-то так:

```
if (request.status == 200) {
    // Ответ сервера получен. Отобразить текст ответа.
    alert(request.responseText);
}
else {
    // Что-то пошло не так. Отобразить код ошибки и сообщение.
    alert("Error " + request.status + ": " + request.statusText);
}
```

Помимо кода состояния и ответа сервера в виде текста или документа объект XMLHttpRequest предоставляет доступ к HTTP-заголовкам, полученным от сервера. Метод getAllResponseHeaders() возвращает заголовки ответа в виде одного сплошного блока текста, а метод getResponseHeader() возвращает значение заголовка по его имени. Например:

```
if (request.status == 200) { // Убедиться в отсутствии ошибок
    // Убедиться, что ответ - это XML-документ
    if (request.getResponseHeader("Content-Type") == "text/xml") {
        var doc = request.responseXML;
        // Теперь обработать полученный документ
    }
}
```

При использовании объекта XMLHttpRequest в синхронном режиме существует одна серьезная проблема: если веб-сервер не ответит на запрос, метод send() окажется заблокированным на достаточно продолжительное время. Исполнение JavaScript-сценария прекратится, создавая ощущение, что веб-броузер «повис» (разумеется, это во многом зависит от типа платформы). Когда сервер прекращает процесс передачи обычной страницы, пользователь может просто щелкнуть на кнопке Остановить и попробовать перейти по другой ссылке или ввести другой URL-адрес. Однако на объект XMLHttpRequest кнопка Остановить никакого воздействия не оказывает. Метод send() не предоставляет возможности определить максимальное время ожидания, а однопоточная модель исполнения сценариев в JavaScript не позволяет прервать работу объекта XMLHttpRequest в синхронном режиме после того как запрос отправлен.

Решение этой проблемы заключается в использовании объекта XMLHttpRequest в асинхронном режиме.

20.1.4. Обработка асинхронного ответа

Чтобы использовать объект XMLHttpRequest в асинхронном режиме, необходимо передать методу open() в третьем аргументе значение true (или просто опустить

третий аргумент, поскольку значение `true` используется по умолчанию). В этом случае метод `send()` отправит запрос серверу и сразу же вернет управление. Когда придет ответ от сервера, он будет доступен через те же свойства объекта `XMLHttpRequest`, которые были описаны выше.

Асинхронный ответ от сервера – это как асинхронный щелчок мыши, сделанный пользователем: вам потребуется извещение, сообщающее об этом. Роль такого извещения может выполнить обработчик события. В случае объекта `XMLHttpRequest` такой обработчик события устанавливается в свойство `onreadystatechange`. Как следует из имени свойства, функция-обработчик вызывается при изменении значения свойства `readyState`. Свойство `readyState` – это целое число, которое определяет код состояния HTTP-запроса, а его возможные значения перечислены в табл. 20.1. Объект `XMLHttpRequest` не определят символических констант ни для одного из пяти значений, перечисленных в таблице.

Таблица 20.1. Значения свойства readyState объекта XMLHttpRequest

readyState	Значение
0	Метод `open()` еще не вызывался
1	Метод `open()` уже был вызван, но метод `send()` еще не вызывался
2	Метод `send()` был вызван, но ответ от сервера еще не получен
3	Идет прием данных от сервера. Значение 3 свойства `readyState` в Firefox и Internet Explorer отличаются; подробности см. в разделе 20.1.4.1
4	Ответ сервера получен полностью[a]

[a] Запрос успешно завершен. – *Примеч. науч. ред.*

Поскольку объект `XMLHttpRequest` имеет всего один обработчик события, он вызывается для обработки всех возможных событий. Обычно обработчик `onreadystatechange` вызывается один раз после вызова метода `open()` и один раз после вызова метода `send()`. Еще раз он вызывается, когда от сервера начинает поступать ответ, и последний раз – когда ответ сервера полностью принят. В отличие от большинства событий в клиентском JavaScript, обработчику `onreadystatechange` не передается объект события. Чтобы определить причину вызова обработчика, необходимо проверить свойство `readyState` объекта `XMLHttpRequest`. К сожалению, обработчику не передается даже сам объект `XMLHttpRequest`, поэтому необходимо определять функцию-обработчик в той области видимости, откуда ей будет доступен объект запроса. Типичный обработчик асинхронного запроса выглядит примерно следующим образом:

```
// Создать XMLHttpRequest с помощью описанной ранее функции
var request = HTTP.newRequest();

// Зарегистрировать обработчик события для приема асинхронных извещений.
// Этот код выполняет обработку ответа и размещается во вложенной функции
// еще до того, как будет отправлен запрос.
request.onreadystatechange = function() {
    if (request.readyState == 4) {       // Если прием запроса завершился
        if (request.status == 200)       // Если запрос увенчался успехом
            alert(request.responseText); // отобразить ответ сервера
```

```
    }
}

// Создать запрос GET для заданного URL-адреса. Третий аргумент опущен,
// поэтому запрос будет выполнен асинхронно
request.open("GET", url);

// Здесь в случае необходимости можно было бы определить дополнительные
// заголовки в запросе.

// Передать запрос. Поскольку это запрос GET, в качестве тела запроса
// передается значение null. Так как это асинхронный запрос, метод send()
// не блокируется и сразу же возвращает управление.
request.send(null);
```

20.1.4.1. Дополнительные замечания о значении 3 свойства readyState

Объект `XMLHttpRequest` еще не стандартизован, поэтому броузеры по-разному обрабатывают значение 3 свойства `readyState`. Например, при загрузке достаточно длинного ответа броузер Firefox несколько раз вызывает обработчик события `onreadystatechange` для значения 3 в свойстве `readyState` с целью обеспечить обратную связь в процессе загрузки. Сценарии могут использовать это обстоятельство для демонстрации пользователю процесса загрузки. С другой стороны, Internet Explorer очень точно интерпретирует имя обработчика события и вызывает его только в случае фактического изменения значения свойства `readyState`. Это означает, что в Internet Explorer обработчик вызывается всего один раз для значения 3 в свойстве `readyState` независимо от того, как долго продолжается загрузка документа.

Броузеры также по-разному реагируют на значение 3 в свойстве `readyState`. Несмотря на то, что значение 3 означает, что какая-то часть ответа уже принята, тем не менее в документации компании Microsoft к объекту `XMLHttpRequest` явно указывается, что в этом состоянии обращение к свойству `responseText` рассматривается как ошибка. В других броузерах, похоже, свойство `responseText` возвращает ту часть ответа, которая уже доступна.

К сожалению, ни один из основных производителей броузеров не предоставил адекватную документацию к своему объекту `XMLHttpRequest`. До тех пор пока `XMLHttpRequest` не будет стандартизован или хотя бы достаточно ясно документирован, лучше всего игнорировать любые значения `readyState`, отличные от 4.

20.1.5. Безопасность объекта XMLHttpRequest

Будучи субъектом политики общего происхождения (см. раздел 13.8.2), объект `XMLHttpRequest` может отправлять HTTP-запросы только тому серверу, откуда был получен документ, использующий этот объект. Это вполне разумное ограничение, но его можно преодолеть, если на стороне сервера разместить сценарий, выполняющий функции прокси, который будет получать содержимое URL-адресов, расположенных за пределами сайта.

Это ограничение безопасности `XMLHttpRequest` имеет одно очень важное следствие: объект `XMLHttpRequest` выполняет HTTP-запросы и не может работать с другими схемами URL-адресации. Например, он не в состоянии работать с такими префиксами URL-адреса, как `file://`. Это значит, что нет никакой возможности прове-

рить работоспособность сценария, использующего объект XMLHttpRequest в локальной файловой системе. Вам придется загрузить тестовый сценарий на веб-сервер (или запустить веб-сервер на своем локальном компьютере). Чтобы сценарий мог выполнить HTTP-запрос, он должен быть загружен броузером через HTTP.

20.2. Примеры и утилиты с объектом XMLHttpRequest

В начале этой главы был представлен пример вспомогательной функции HTTP.newRequest(), которая позволяет получить объект XMLHttpRequest в любом броузере. Аналогичным образом с помощью других вспомогательных функций можно существенно упростить работу с объектом XMLHttpRequest. В следующих подразделах приводятся примеры таких вспомогательных функций.

20.2.1. Основные утилиты для работы с запросами GET

В примере 20.2 приводится очень простая функция, обрабатывающая наиболее общий случай использования объекта XMLHttpRequest: просто передайте ей требуемый URL-адрес и функцию, которая примет текст ответа.

Пример 20.2. Вспомогательная функция HTTP.getText( )

```
/**
 * Использует объект XMLHttpRequest для получения содержимого по заданному
 * URL-адресу методом GET. Получив ответ, передает его
 * (в виде простого текста) указанной функции обратного вызова.
 *
 * Эта функция не блокируется и не имеет возвращаемого значения.
 */
HTTP.getText = function(url, callback) {
    var request = HTTP.newRequest();
    request.onreadystatechange = function() {
        if (request.readyState == 4 && request.status == 200)
            callback(request.responseText);
    }
    request.open("GET", url);
    request.send(null);
};
```

В примере 20.3 приводится тривиальный вариант функции, которая принимает XML-документ и передает его функции обратного вызова.

Пример 20.3. Вспомогательная функция HTTP.getXML( )

```
HTTP.getXML = function(url, callback) {
    var request = HTTP.newRequest();
    request.onreadystatechange = function() {
        if (request.readyState == 4 && request.status == 200)
            callback(request.responseXML);
    }
    request.open("GET", url);
    request.send(null);
};
```

20.2.2. Получение только заголовков

Одна из особенностей объекта `XMLHttpRequest` заключается в том, что он позволяет определить используемый HTTP-метод. HTTP-метод HEAD запрашивает у сервера только заголовки для заданного URL-адреса без содержимого, расположенного по этому адресу. Эта возможность может использоваться, например, для проверки даты последнего изменения ресурса, прежде чем загружать его.

В примере 20.4 демонстрируется, как можно выполнить запрос HEAD. Он включает функцию, которая выполняет анализ пар имя–значение в HTTP-заголовке и сохраняет их в виде свойств JavaScript-объекта. Здесь также имеется функция обработки ошибок, которая вызывается в случае получения от сервера кода состояния 404 и других кодов ошибок.

Пример 20.4. Вспомогательная функция HTTP.getHeaders( )

```
/**
 * Использует HTTP-запрос HEAD для получения заголовков с указанного
 * URL-адреса. После получения заголовков анализирует их с помощью функции
 * HTTP.parseHeaders() и передает получившийся объект указанной функции
 * обратного вызова. Если сервер вернет код ошибки, вызывает указанную
 * функцию errorHandler. Если обработчик ошибок не задан, передает значение
 * null функции обратного вызова.
 */
HTTP.getHeaders = function(url, callback, errorHandler) {
    var request = HTTP.newRequest();
    request.onreadystatechange = function() {
        if (request.readyState == 4) {
            if (request.status == 200) {
                callback(HTTP.parseHeaders(request));
            }
            else {
                if (errorHandler) errorHandler(request.status,
                                               request.statusText);
                else callback(null);
            }
        }
    }
    request.open("HEAD", url);
    request.send(null);
};

// Анализирует заголовки ответа, полученные в XMLHttpRequest, и возвращает
// имена и значения в виде свойств нового объекта.
HTTP.parseHeaders = function(request) {
    var headerText = request.getAllResponseHeaders(); // Текст от сервера
    var headers = {}; // Это возвращаемое значение
    var ls = /^\s*/;  // Регулярное выражение, удаляющее начальные пробелы
    var ts = /\s*$/;  // Регулярное выражение, удаляющее конечные пробелы

    // Разбить заголовки на строки
    var lines = headerText.split("\n");
    // Цикл по всем строкам
    for(var i = 0; i < lines.length; i++) {
        var line = lines[i];
```

```
            if (line.length == 0) continue; // Пропустить пустые строки
            // Разбить каждую строку по первому двоеточию и удалить лишние пробелы
            var pos = line.indexOf(':');
            var name = line.substring(0, pos).replace(ls, "").replace(ts, "");
            var value = line.substring(pos+1).replace(ls, "").replace(ts, "");
            // Сохранить пару имя-значение в виде свойства JavaScript-объекта
            headers[name] = value;
        }
        return headers;
    };
```

20.2.3. HTTP-метод POST

HTML-формы по умолчанию отправляются на сервер методом POST. При выполнении запроса POST данные передаются на сервер в теле запроса, а не в строке URL-адреса. Поскольку параметры запроса в методе GET приходится вставлять в URL, метод GET пригоден только для случаев, когда запрос не вызывает побочных эффектов на стороне сервера, т, е. когда повторные запросы GET с тем же самым URL-адресом и с теми же параметрами приводят к получению тех же самых результатов. Если запрос сопровождается побочными эффектами (например, сервер сохраняет некоторые из параметров в базе данных), должен использоваться запрос POST.

Пример 20.5 демонстрирует порядок выполнения запросов POST с помощью объекта `XMLHttpRequest`. Метод `HTTP.post()` вызывает функцию `HTTP.encodeFormData()` для преобразования свойств объекта в строковую форму, которая может использоваться в качестве тела запроса POST. Затем полученная строка передается методу `XMLHttpRequest.send()` и становится телом запроса. (Кроме того, строка, созданная с помощью функции `HTTP.encodeFormData()`, может добавляться в URL-адрес метода GET; достаточно лишь отделить URL-адрес и данные символом вопросительного знака.) Помимо этого в примере 20.5 используется метод `HTTP._getResponse()`. Данный метод анализирует ответ сервера на основе его типа. Реализация этого метода приводится в следующем разделе.

Пример 20.5. Вспомогательная функция HTTP.post( )

```
/**
 * Отправляет HTTP-запрос POST по указанному URL-адресу,
 * используя имена и значения свойств объекта в качестве тела запроса.
 * Анализирует ответ сервера на основе его типа и передает
 * полученное значение функции обратного вызова.
 * В случае появления HTTP-ошибки вызывает заданную
 * функцию errorHandler или передает значение null
 * функции обратного вызова, если обработчик ошибок не определен.
 **/
HTTP.post = function(url, values, callback, errorHandler) {
    var request = HTTP.newRequest();
    request.onreadystatechange = function() {
        if (request.readyState == 4) {
            if (request.status == 200) {
                callback(HTTP._getResponse(request));
            }
            else {
```

```
                if (errorHandler) errorHandler(request.status,
                                               request.statusText);
                else callback(null);
            }
        }
    }
    request.open("POST", url);

    // Этот заголовок сообщает серверу, как интерпретировать тело запроса.
    request.setRequestHeader("Content-Type",
                             "application/x-www-form-urlencoded");
    // Вставить в тело запроса имена и значения свойств объекта
    // и отправить их в теле запроса.
    request.send(HTTP.encodeFormData(values));
};

/**
 * Интерпретирует имена и значения свойств объекта, как если бы они были
 * значениями элементов формы, использует формат application/x-www-form-urlencoded
 */
HTTP.encodeFormData = function(data) {
    var pairs = [];
    var regexp = /%20/g; // Регулярное выражение, соответствующее закодированному пробелу
    for(var name in data) {
        var value = data[name].toString();
        // Создать пару имя/значение, но сначала имя и значение закодировать.
        // Практически все, что нам требуется, выполняет глобальная функция
        // encodeURIComponent, но она превращает пробелы в виде %20 вместо
        // требуемого нам "+". Исправить это можно с помощью String.replace()
        var pair = encodeURIComponent(name).replace(regexp,"+") + '=' +
                   encodeURIComponent(value).replace(regexp,"+");
        pairs.push(pair);
    }

    // Объединить все пары в строку, разделяя их символами &
    return pairs.join('&');
};
```

Еще один вариант выполнения запроса POST с помощью объекта `XMLHttpRequest` приводится в примере 21.14. Код в этом примере вызывает веб-службу, но вместо значений элементов формы в теле запроса передает XML-документ.

20.2.4. Ответы в форматах HTML, XML и JSON

В большинстве примеров, продемонстрированных выше, ответ сервера на HTTP-запрос интерпретировался как простой текст. Это вполне законно, и никто не может заявить, что веб-серверы не могут возвращать документы с содержимым типа «text/plain». Анализировать такой ответ из JavaScript-сценария можно с помощью разнообразных строковых методов и делать с ним все, что потребуется.

Ответ сервера всегда можно интерпретировать как простой текст, даже если его содержимое имеет другой тип. Если сервер, например, возвращает HTML-документ, можно извлечь содержимое этого документа с помощью свойства `responseText` и затем вставить его в какой-либо элемент документа с помощью свойства `innerHTML`.

Однако существуют и другие способы обработки ответа, полученного от сервера. Как уже отмечалось в начале главы, если сервер отправляет ответ с содержимым типа «text/xml», можно получить преобразованное представление XML-документа из свойства `responseXML`. Значением этого свойства является DOM-объект `Document`, поэтому для работы с таким документом допускается использовать DOM-методы.

Однако следует заметить, что использование формата XML для передачи данных может оказаться далеко не лучшим выбором. Если сервер передает данные, которые будут обрабатываться на стороне клиента JavaScript-сценарием, весьма неэффективным будет сначала преобразовать эти данные в формат XML на стороне сервера, затем с помощью объекта `XMLHttpRequest` преобразовать эти данные в DOM-дерево узлов, а потом в сценарии выполнять обход этого дерева для извлечения данных. Более короткий путь заключается в том, чтобы на стороне сервера преобразовать данные в литералы объектов и массивов, а затем передать полученный исходный текст на языке JavaScript веб-броузеру. После этого сценарий сможет «проанализировать» ответ, просто передав его методу `eval()`.

Преобразование данных в форму JavaScript-объектов известно под названием JSON (JavaScript Object Notation – нотация JavaScript-объектов).[1] Вот примеры данных в форматах XML и JSON:

```
<!-- Формат XML -->
<author>
  <name>Wendell Berry</name>
  <books>
    <book>The Unsettling of America</book>
    <book>What are People For?</book>
  </books>
</author>

// Формат JSON
{
    "name": "Wendell Berry",
    "books": [
        "The Unsettling of America",
        "What are People For?"
    ]
}
```

Функция `HTTP.post()`, приводившаяся в примере 20.5, вызывает функцию `HTTP._getResponse()`. Эта функция отыскивает заголовок `Content-Type` и с его помощью определяет форму представления ответа. В примере 20.6 приводится реализация `HTTP_getResponse()`, которая возвращает XML-документ в виде объекта `Document`, интерпретирует с помощью `eval()` содержимое JavaScript- или JSON-документов, а документы любых других типов возвращает в виде обычного текста.

---

[1] Подробнее узнать о JSON можно на сайте *http://json.org*. Идея принадлежит Дугласу Крокфорду (Douglas Crockford); на его веб-сайте можно найти ссылки на утилиты преобразования данных в/из формата JSON, написанные на различных языках программирования. Такой способ кодирования данных может оказаться полезным даже для тех, кто не пользуется JavaScript.

Пример 20.6. HTTP._getResponse( )

```
HTTP._getResponse = function(request) {
    // Проверить тип содержимого, полученного от сервера
    switch(request.getResponseHeader("Content-Type")) {
    case "text/xml":
        // Если это XML-документ, вернуть объект Document.
        return request.responseXML;
    case "text/json":
    case "text/javascript":
    case "application/javascript":
    case "application/x-javascript":
        // Если это JavaScript-код или документ в формате JSON, вызвать eval(),
        // чтобы выполнить преобразование текста в JavaScript-значение.
        // Обратите внимание: делать это следует только в том случае,
        // если добропорядочность сервера не вызывает сомнений!
        return eval(request.responseText);
    default:
        // В противном случае интерпретировать ответ как простой текст
        // и вернуть его как строку.
        return request.responseText;
    }
};
```

Не используйте метод `eval()` для обработки данных в формате JSON, как это делается в примере 20.6, если не уверены в том, что сервер никогда не пришлет злонамеренный программный код вместо данных в формате JSON. Более безопасная альтернатива методу `eval()` – выполнить разбор объектов-литералов JavaScript «вручную», без вызова `eval()`.

20.2.5. Ограничение времени ожидания запроса

Недостатком объекта `XMLHttpRequest` является отсутствие возможности ограничить время ожидания исполнения запроса. Этот недостаток особенно критичен для синхронных запросов. Если связь с сервером пропадет, веб-броузер окажется заблокированным в методе `send()` и не будет реагировать на действия пользователя. В случае асинхронных запросов такого не происходит, поскольку метод `send()` не блокируется, и веб-броузер может продолжать реагировать на действия пользователя. Однако и здесь существует проблема ограничения времени выполнения запроса. Предположим, что приложение с помощью объекта `XMLHttpRequest` запустило HTTP-запрос, когда пользователь щелкнул на кнопке. Чтобы предотвратить возможность отправки нескольких запросов, нелишне сделать кнопку неактивной до того момента, как прибудет ответ сервера. Но что если сервер остановится или произойдет нечто, что помешает получению ответа на запрос? Броузер не будет заблокирован, но возможности приложения из-за неактивной кнопки окажутся ограниченными.

Чтобы избежать проблем подобного рода, было бы удобно иметь возможность устанавливать собственные тайм-ауты с помощью функции `Windows.setTimeout()` при выполнении HTTP-запросов. В обычной ситуации ответ приходит до того, как будет вызван обработчик события таймера, – в этом случае можно просто вызвать функцию `Window.clearTimeout()`, чтобы отменить срабатывание таймера. С другой стороны, если обработчик события от таймера будет вызван раньше,

чем свойство readyState получит значение 4, исполнение запроса можно будет отменить с помощью метода XMLHttpRequest.abort(). После этого обычно следует известить пользователя о том, что попытка выполнения запроса потерпела неудачу (например, методом Window.alert()). Если в этом гипотетическом примере перед запуском запроса кнопка была деактивирована, ее можно будет повторно активировать по истечении предельного времени ожидания.

В примере 20.7 определяется функция HTTP.get(), которая демонстрирует только что описанный прием организации тайм-аута. Она представляет собой усовершенствованную версию функции HTTP.getText() из примера 20.2 и поддерживает многие из возможностей, демонстрировавшихся в предыдущих примерах, включая обработку ошибок, параметры запроса и метод HTTP._getResponse(), описанный выше. Кроме того, она допускает возможность указать необязательную функцию обратного вызова, которая будет вызываться всякий раз, когда произойдет событие onreadystatechange со значением свойства readyState, отличным от 4. В таких броузерах, как Firefox, обработчик этого события может вызываться неоднократно со значением 3, и данная функция обратного вызова позволяет сценарию демонстрировать пользователю индикатор процесса загрузки.

Пример 20.7. Вспомогательная функция HTTP.get( )

```
/**
 * Отправляет HTTP-запрос GET с заданным URL. В случае успешного
 * получения ответа он преобразуется в объект на основе заголовка
 * Content-Type и передается указанной функции обратного вызова.
 * Дополнительные аргументы могут быть переданы в виде свойств объекта options.
 *
 * Если получен ответ с сообщением об ошибке (например, сообщение
 * 404 Not Found), код состояния и сообщение передаются функции
 * options.errorHandler. Если обработчик ошибок не определен, вызывается
 * функция обратного вызова со значением null в аргументе.
 *
 * Если объект options.parameters определен, его свойства интерпретируются
 * как имена и значения параметров запроса. С помощью HTTP.encodeFormData()
 * они преобразуются в строку, которую можно вставить в URL, после чего эта
 * строка добавляется в конец URL вслед за символом '?'.
 *
 * Если определена функция options.progressHandler, она будет вызываться
 * всякий раз, когда свойство readyState обретает новое значение, меньшее 4.
 * Каждый раз этой функции будет передаваться количество вызовов этой функции.
 *
 * Если указано значение options.timeout, работа объекта XMLHttpRequest будет
 * прервана, если запрос не будет исполнен до истечения заданного числа миллисекунд.
 * Если предельное время ожидания истекло и определена функция
 * options.timeoutHandler, она будет вызвана со строкой
 * URL запроса в виде аргумента.
 *
 **/
HTTP.get = function(url, callback, options) {
    var request = HTTP.newRequest();
    var n = 0;
    var timer;
    if (options.timeout)
```

```
            timer = setTimeout(function() {
                                    request.abort();
                                    if (options.timeoutHandler)
                                        options.timeoutHandler(url);
                                },
                                options.timeout);
        request.onreadystatechange = function() {
            if (request.readyState == 4) {
                if (timer) clearTimeout(timer);
                if (request.status == 200) {
                    callback(HTTP._getResponse(request));
                }
                else {
                    if (options.errorHandler)
                        options.errorHandler(request.status,
                                             request.statusText);
                    else callback(null);
                }
            }
            else if (options.progressHandler) {
                options.progressHandler(++n);
            }
        }
        var target = url;
        if (options.parameters)
            target += "?" + HTTP.encodeFormData(options.parameters)
        request.open("GET", target);
        request.send(null);
    };
```

20.3. Ajax и динамические сценарии

Термин *Ajax* обозначает архитектуру веб-приложений, которая основана на взаимодействии с протоколом HTTP и объектом `XMLHttpRequest`. (В действительности для многих объект `XMLHttpRequest` и Ajax являются синонимами.) Ajax – это акроним от «Asynchronous JavaScript and XML» (асинхронный JavaScript и XML).[1] Термин был придуман Джессом Джеймсом Гарреттом (Jesse James Garrett) и впервые появился в феврале 2005 года в его статье «Ajax: A New Approach to Web Applications» (Ajax: новый подход к разработке веб-приложений), которую можно найти по адресу *http://www.adaptivepath.com/publications/essays/archives/000385.php*.

[1] Значимость архитектуры Ajax сложно переоценить, и наличие простого названия лишь послужило катализатором начала революции в разработке веб-приложений. Однако, как оказывается, этот акроним недостаточно полно описывает технологии, используемые Ajax-приложениями. Все клиентские JavaScript-сценарии используют механизм обработки событий и потому являются асинхронными. Кроме того, применение XML в веб-приложениях, разработанных в стиле Ajax, часто бывает удобным, но это совершенно не обязательно. Главная особенность Ajax-приложений – взаимодействие с протоколом HTTP, но это никак не отражено в акрониме.

Объект `XMLHttpRequest`, на котором основана технология Ajax, был доступен в броузерах Microsoft и Netscape/Mozilla еще за четыре года до появления статьи Гарретта, но он никогда до этого момента не привлекал такого внимания.[1] Все изменилось в 2004 году, когда компания Google выпустила новую версию почтового веб-приложения Gmail, использующего объект `XMLHttpRequest`. Сочетание этого высококлассного приложения, выполненного на профессиональном уровне, и статьи Гарретта, вышедшей в начале 2005 года, открыло шлюзы для бурного интереса к Ajax.

Ключевой особенностью любого Ajax-приложения является взаимодействие с веб-сервером по протоколу HTTP без необходимости полной перезагрузки страницы. Поскольку объемы передаваемых данных невелики и броузер не тратит время на анализ и отображение целого документа (а также связанных с ним таблиц стилей и сценариев), время отклика таких приложений оказывается очень небольшим. В результате веб-приложения стали напоминать традиционные настольные приложения.

Необязательной особенностью Ajax-приложений является использование формата XML для представления данных во время обмена между клиентом и сервером. В главе 21 демонстрируется, как можно манипулировать XML-данными из JavaScript-сценариев, включая исполнение XPath-запросов и выполнение XSL-преобразований XML-документов в формат HTML. В некоторых Ajax-приложениях для отделения содержимого (данные в формате XML) от представления (HTML-форматирование, выполненное с помощью таблиц стилей XSL) используется язык XSLT. Такой подход дает дополнительные преимущества, уменьшая объемы данных, передаваемых от сервера клиенту, и перенося выполнение необходимых преобразований на сторону клиента.

Существует возможность формализовать Ajax в терминах RPC-механизма[2]. В такой формулировке веб-разработчики используют низкоуровневые Ajax-библиотеки как на стороне сервера, так и на стороне клиента, чтобы облегчить высокоуровневое взаимодействие между клиентом и сервером. В данной главе не описываются никакие библиотеки, реализующие RPC средствами Ajax, потому что основное внимание здесь уделяется низкоуровневым технологиям, обеспечивающим работу архитектуры Ajax.

Ajax – достаточно молодая прикладная архитектура, и описывающая ее статья Гарретта заканчивается призывом к действию, который стоит того, чтобы привести его здесь:

> Самые большие сложности в разработке Ajax-приложений лежат вовсе не в технической плоскости. Технологии, составляющие основу Ajax, достаточно

---

[1] Я сожалею, что не взялся за описание объекта `XMLHttpRequest` в четвертом издании этой книги. То издание было во многом основано на стандартах, и объект `XMLHttpRequest` не был включен в него просто потому, что он никогда не был стандартизован. Если бы в то время я осознавал мощные возможности, которые предоставляет работа с протоколом HTTP, я нарушил бы свои правила и включил бы описание объекта в книгу.

[2] Аббревиатура RPC происходит от Remote Procedure Call (вызов удаленных процедур) и описывает стратегию, используемую в распределенных вычислениях для упрощения взаимодействий между клиентом и сервером.

зрелые, устоявшиеся и понятные. Главные проблемы заключаются в том, что разработчики таких приложений забывают думать о существующих ограничениях Всемирной паутины и начинают воображать себе более широкий, более богатый диапазон возможностей.

Это будет забавно.

20.3.1. Пример применения Ajax

Примеры, приводившиеся до сих пор в этой главе, представляли собой вспомогательные функции, демонстрирующие порядок использования объекта `XMLHttpRequest`. Они не показывали, *зачем* может потребоваться этот объект или *какие* выгоды он дает. Как отмечалось в цитате из статьи Гарретта, архитектура Ajax открывает массу новых возможностей, которые только начали исследоваться. Следующий пример достаточно прост, но в нем демонстрируются некоторые вспомогательные функции и ряд возможностей, предоставляемых архитектурой Ajax.

Пример 20.8 представляет собой ненавязчивый сценарий, регистрирующий обработчики событий в ссылках документа, чтобы с их помощью отображать всплывающие подсказки при наведении на них указателя мыши. Для ссылок, указывающих на тот же сервер, откуда был загружен сам документ, сценарий выполняет HTTP-запрос HEAD с помощью объекта `XMLHttpRequest`. Из возвращаемых сервером заголовков извлекаются тип содержимого, размер и дата последнего изменения документа, на который указывает ссылка, и эта информация отображается в виде всплывающей подсказки (рис. 20.1). Тем самым всплывающие подсказки предоставляют своего рода средство предварительной оценки целевого документа, что может помочь пользователям в принятии решения о том, стоит ли щелкать на этой ссылке или нет.

В основе реализации лежит класс `Tooltip`, разработанный в примере 16.4 (здесь не требуется расширенная версия класса, которая приводилась в примере 17.3). Кроме того, здесь используется модуль `Geometry` из примера 14.2 и вспомогательная функция `HTTP.getHeaders()` из примера 20.4. В программный код заложено несколько уровней асинхронности: в форме это обработчик события `onload`, обработчик события `onmouseover`, таймер и функция обратного вызова для объекта `XMLHttpRequest`. Все это приводит к созданию глубоко вложенных функций.

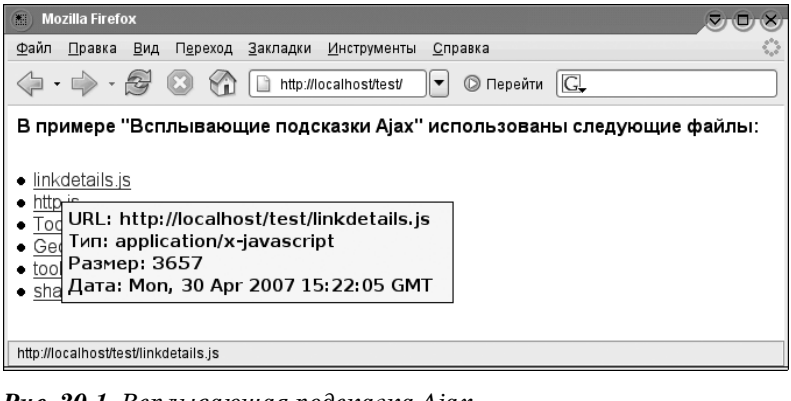

Рис. 20.1. *Всплывающая подсказка Ajax*

Пример 20.8. Всплывающие подсказки Ajax

```
/**
 * linkdetails.js
 *
 * Данный модуль добавляет обработчики событий к ссылкам в документе,
 * с помощью которых отображаются всплывающие подсказки при задержке
 * указателя мыши над этими ссылками в течение полусекунды. Если ссылка
 * указывает на документ на том же сервере, что и исходный документ, всплывающая
 * подсказка будет включать в себя информацию о типе, размере и дате, которая
 * извлекается с помощью HTTP-запроса HEAD, выполняемого объектом XMLHttpRequest.
 *
 * Данный модуль требует наличия модулей Tooltip.js, HTTP.js и Geometry.js
 */
(function() { // Анонимная функция, которая содержит все необходимые имена
    // Создает объект подсказки
    var tooltip = new Tooltip();

    // Настроить вызов функции init() после загрузки документа
    if (window.addEventListener) window.addEventListener("load", init, false);
    else if (window.attachEvent) window.attachEvent("onload", init);

    // Вызывается после загрузки документа
    function init() {
        var links = document.getElementsByTagName('a');

    // Цикл по всем ссылкам и добавление обработчиков событий
    for(var i = 0; i < links.length; i++)
        if (links[i].href) addTooltipToLink(links[i]);
    }

    // Эта функция добавляет обработчики событий
    function addTooltipToLink(link) {
        // Добавить обработчики событий
        if (link.addEventListener) { // Стандартный прием
            link.addEventListener("mouseover", mouseover, false);
            link.addEventListener("mouseout", mouseout, false);
        }
        else if (link.attachEvent) { // Для IE
            link.attachEvent("onmouseover", mouseover);
            link.attachEvent("onmouseout", mouseout);
        }

        var timer; // Используется в вызовах функций setTimeout/clearTimeout

        function mouseover(event) {
            var e = event || window.event;
            // Получить положение указателя мыши, преобразовать
            // в координаты документа и добавить смещение
            var x = e.clientX + Geometry.getHorizontalScroll() + 25;
            var y = e.clientY + Geometry.getVerticalScroll() + 15;

            // Если запланирован вывод подсказки, отменить его
            if (timer) window.clearTimeout(timer);

            // Запланировать вывод подсказки через полсекунды
            timer = window.setTimeout(showTooltip, 500);
```

```
            function showTooltip() {
                // Если HTTP-ссылка указывает на тот же хост, откуда был
                // загружен этот сценарий, использовать объект XMLHttpRequest
                // для получения дополнительной информации.
                if (link.protocol == "http:" && link.host == location.host) {
                    // Выполнить запрос заголовков по ссылке
                    HTTP.getHeaders(link.href, function(headers) {
                        // Собрать строку из заголовков
                        var tip = "URL: " + link.href + "<br>" +
                            "Тип: " + headers["Content-Type"] + "<br>" +
                            "Размер: " + headers["Content-Length"] + "<br>" +
                            "Дата: " + headers["Last-Modified"];
                        // И отобразить ее в виде всплывающей подсказки
                        tooltip.show(tip, x, y);
                    });

                }
                else {
                    // Иначе, если это ссылка на другой сайт,
                    // отобразить в подсказке лишь URL-адрес ссылки
                    tooltip.show("URL: " + link.href, x, y);
                }
            }
        }

        function mouseout(e) {
            // Когда указатель мыши смещается со ссылки, отменить отображение
            // запланированной подсказки или скрыть ее, если она уже отображается.
            if (timer) window.clearTimeout(timer);
            timer = null;
            tooltip.hide();
        }
    }
})();
```

20.3.2. Одностраничные приложения

Под термином *одностраничное приложение* (*single-page application*) понимается именно то, что он обозначает: управляемое JavaScript-сценарием веб-приложение, которое требует загрузки единственной страницы. Некоторые одностраничные приложения после загрузки вообще не взаимодействуют с сервером. Примером таких приложений могут служить DHTML-игры, где взаимодействие с пользователем приводит лишь к модификации загруженного документа.

Объект `XMLHttpRequest` и архитектура Ajax открывают массу дополнительных возможностей. Веб-приложения могут использовать эти технологии для обмена данными с сервером и оставаться одностраничными приложениями. Веб-приложение, разработанное в соответствии с этими положениями, может содержать небольшой объем JavaScript-кода, выполняющего начальную загрузку, и «экранную заставку» в формате HTML, которая отображается в процессе инициализации приложения. После вывода экранной заставки запускающий JavaScript-код мог бы с помощью объекта `XMLHttpRequest` загрузить фактический JavaScript-код приложения, который можно было бы запустить методом `eval()`. Этот Java-

Script-код мог бы взять на себя обязанности по загрузке требуемых данных с помощью `XMLHttpRequest` и с использованием DHTML отобразить эти данные перед пользователем.

20.3.3. Удаленное взаимодействие

Термин *удаленное взаимодействие* (*remote scripting*) появился более чем за четыре года до термина *Ajax* и представляет собой всего лишь менее броское название одной и той же идеи: использование протокола HTTP для обеспечения тесной интеграции (и уменьшения времени отклика) клиента и сервера. В статье компании Apple, вышедшей в 2002 году и получившей широкую известность, описывается, как с помощью тега `<iframe>` отправлять веб-серверу HTTP-запросы (*http://developer.apple.com/internet/webcontent/iframe.html*). В данной статье отмечается, что если веб-сервер отправляет обратно HTML-файл, содержащий тег `<script>`, то JavaScript-код из этого тега будет исполнен броузером и сможет вызывать методы, определенные в окне, содержащем этот тег `<iframe>`. Таким способом сервер может посылать клиенту прямые команды в форме JavaScript-инструкций.

20.3.4. Предостережения по использованию архитектуры Ajax

Подобно любой другой архитектуре, Ajax имеет свои ловушки. В этом разделе описываются три основные проблемы, о которых следует знать при разработке Ajax-приложений.

Первая проблема – визуальная обратная связь. Когда пользователь щелкает на традиционной гиперссылке, веб-броузер обеспечивает индикацию процесса загрузки содержимого ссылки. Такая обратная связь предоставляется еще до того, как содержимое будет готово к отображению, поэтому пользователь явно видит, что броузер работает над выполнением его запроса. Однако, когда HTTP-запрос запускается из объекта `XMLHttpRequest`, броузер не предоставляет никакой обратной связи. Даже при подключении к магистральным линиям инерционность сети часто вызывает заметные задержки между посылкой HTTP-запроса и получением ответа. Поэтому для Ajax-приложений особенно важно обеспечивать визуальную обратную связь (например, в виде простой DHTML-анимации; подробнее об этом см. в главе 16), пока приложение ожидает получение ответа от `XMLHttpRequest`.

Обратите внимание: в примере 20.8 не учитывается совет по обеспечению визуальной обратной связи просто потому, что в данном примере для запуска HTTP-запроса пользователь не предпринимает никаких активных действий. Запрос выполняется, когда пользователь (пассивно) наводит указатель мыши на ссылку. Пользователь явно не требует от приложения выполнить какое-либо действие и потому не нуждается в обратной связи.

Вторая проблема связана с URL. В традиционных веб-приложениях переход из одного состояния в другое сопровождается загрузкой новой страницы, причем каждая страница имеет свой уникальный URL-адрес. Это не относится к Ajax-приложениям: когда Ajax-приложения используют протокол HTTP для загрузки и отображения нового содержимого, URL в адресной строке не изменяется. У пользователя может появиться желание сделать закладку на приложение

в конкретном состоянии, но с помощью механизма закладок броузера сделать это будет невозможно. Более того, пользователю не поможет даже копирование URL из адресной строки броузера.

Эта проблема и ее решение прекрасно иллюстрирует приложение Google Maps (*http://local.google.com*). При изменении масштаба карты или ее прокрутке между клиентом и сервером передаются огромные объемы информации, но URL-адрес, отображаемый в адресной строке броузера, не изменяется. Компания Google решила проблему установки закладок, добавив в каждую страницу ссылку «link to this page» (ссылка на эту страницу). Щелчок на этой ссылке генерирует URL-адрес на отображаемую в данный момент карту и вызывает перезагрузку страницы с этим URL-адресом. После того как загрузка завершится, ссылка на карту в текущем состоянии может быть помещена в закладки, отправлена по электронной почте и тому подобное. Главное, что должны извлечь разработчики из этого урока: все, что имеет существенное значение для описания состояния веб-приложения, должно инкапсулироваться в URL-адресе и этот URL-адрес должен быть доступен пользователю в случае необходимости.

Третья проблема, которая часто упоминается при обсуждении Ajax, связана с кнопкой броузера Назад. Отобрав у броузера контроль над протоколом HTTP, сценарии, использующие объект `XMLHttpRequest`, обходят механизм хранения истории броузера. Пользователи привыкли применять кнопки Назад и Вперед для навигации по Всемирной паутине. Если Ajax-приложение средствами HTTP получает и отображает существенные объемы содержимого документа, пользователи могут попробовать с помощью этих кнопок перемещаться между различными состояниями приложения. Но когда они попробуют щелкнуть на кнопке Назад, то с удивлением обнаружат, что эта кнопка отправляет их за пределы приложения, вместо того чтобы вернуть к его предыдущему состоянию.

В свое время неоднократно предпринимались попытки решить проблему кнопки Назад за счет добавления URL-адресов в историю броузера. Однако, как правило, эти попытки увязали в трясине программного кода, учитывающего специфические особенности каждого броузера, и не давали достаточно удовлетворительных результатов. И даже когда удавалось добиться положительных результатов, найденные решения противоречили основной парадигме Ajax и вынуждали пользователя полностью перезагружать страницы, вместо того чтобы обеспечить прозрачное взаимодействие с сервером по протоколу HTTP.

На мой взгляд, проблема кнопки Назад не настолько серьезна, как ее пытаются представить, и ее отрицательное влияние может быть минимизировано за счет тщательного обдумывания дизайна веб-приложения. Элементы приложения, похожие на гиперссылки, должны вести себя как гиперссылки и действительно должны вызывать перезагрузку страницы. Это сделает их субъектами механизма истории броузера, как того и ожидает пользователь. Те же элементы приложения, которые инициируют взаимодействия с протоколом HTTP в обход механизма истории броузера, наоборот, не должны напоминать гиперссылки. Вернемся к приложению Google Maps еще раз. Когда пользователь прокручивает карту в окне броузера, он не ожидает, что кнопка Назад сможет отменить операцию прокрутки, точно так же, как он не ожидает, что кнопка Назад отменит операцию прокрутки обычной веб-страницы в окне броузера.

Следует с особой осторожностью подходить к использованию слов «вперед» и «назад» в Ajax-приложениях для обозначения внутренних элементов управления навигацией. Например, если интерфейс приложения реализован в стиле многостраничного мастера с кнопками Вперед и Назад для отображения следующего или предыдущего экрана, оно должно поддерживать традиционный способ загрузки страниц (вместо `XMLHttpRequest`), потому что в такой ситуации пользователь вполне оправданно ожидает, что кнопка броузера Назад будет вызывать точно такой же эффект, что и кнопка Назад в приложении.

В более широком смысле кнопка броузера Назад не должна восприниматься как кнопка Отмена в приложении. Ajax-приложения могут предусматривать собственную реализацию операций повторить/отменить, если они будут востребованы пользователем, но они должны совершенно четко отличаться от функций, выполняемых кнопками Назад и Вперед броузера.

20.4. Взаимодействие с протоколом HTTP с помощью тега <script>

В броузерах Internet Explorer версий 5 и 6 объект `XMLHttpRequest` представляет собой ActiveX-объект. Иногда из соображений безопасности пользователи запрещают применение ActiveX-объектов в Internet Explorer, и в такой ситуации сценарии лишены возможности создавать объекты `XMLHttpRequest`. В случае необходимости можно будет выполнять простые HTTP-запросы GET с помощью тегов `<iframe>` и `<script>`. Несмотря на то, что таким способом невозможно реализовать все функциональные возможности объекта `XMLHttpRequest`,[1] тем не менее удастся реализовать, по меньшей мере, вспомогательную функцию `HTTP.getText()`, которая работает без привлечения ActiveX.

Сгенерировать HTTP-запрос достаточно просто с помощью свойства `src` тегов `<script>` и `<iframe>`. Но гораздо сложнее извлечь данные из этих элементов без изменения этих данных броузером. Тег `<iframe>` ожидает, что в него будет загружен HTML-документ. Если попытаться загрузить в плавающий фрейм простой текст, вы обнаружите, что текст будет преобразован в формат HTML. Кроме того, некоторые версии Internet Explorer некорректно реализуют обработку событий `onload` и `onreadystatechange` в теге `<iframe>`, что еще больше осложняет ситуацию.

Подход, который здесь рассматривается, основан на использовании тега `<script>` и сценария на стороне сервера. В этом случае серверному сценарию сообщается URL-адрес, содержимое которого требуется получить, и имя функции на стороне клиента, которой это содержимое должно быть передано. Серверный сценарий берет содержимое с требуемого URL-адреса, преобразует его в JavaScript-строку (возможно, достаточно длинную) и возвращает клиентский сценарий, который передает эту строку указанной функции. Поскольку данный клиентский сценарий загружается в тег `<script>`, по окончании загрузки указанная функция вызывается автоматически.

В примере 20.9 приводится реализация серверного сценария на языке PHP.

---

[1] Полная замена объекта `XMLHttpRequest`, вероятно, потребует применения Java-апплета.

Пример 20.9. jsquoter.php

```
<?php
// Указать броузеру, что выполняется передача сценария
header("Content-Type: text/javascript");
// Извлечь аргументы из URL
$func = $_GET["func"];       // Функция вызова нашего JavaSript-кода
$filename = $_GET["url"];    // Файл или URL для передачи функции func
$lines = file($filename);    // Получить строки содержимого файла
$text = implode("", $lines); // Объединить их в одну строку
// Экранировать кавычки и символы перевода строки
$escaped = str_replace(array("'", "\"", "\n", "\r"),
                       array("\\'", "\\\"", "\\n", "\\r"),
                       $text);
// Отправить все это в виде одиночного вызова JavaScript-функции
echo "$func('$escaped');"
?>
```

Клиентская функция в примере 20.10 использует серверный сценарий *jsquoter.php* из примера 20.9 и работает на манер функции `HTTP.getText()` из примера 20.2.

Пример 20.10. Вспомогательная функция HTTP.getTextWithScript( )

```
HTTP.getTextWithScript = function(url, callback) {
    // Создать новый элемент-сценарий и добавить его в документ.
    var script = document.createElement("script");
    document.body.appendChild(script);

    // Получить уникальное имя для функции.
    var funcname = "func" + HTTP.getTextWithScript.counter++;

    // Определить функцию с этим именем, используя данную функцию как удобное
    // пространство имен. Сценарий, созданный на стороне сервера,
    // будет вызывать эту функцию.
    HTTP.getTextWithScript[funcname] = function(text) {
        // Передать текст функции обратного вызова
        callback(text);

        // Удалить тег script и созданную функцию.
        document.body.removeChild(script);
        delete HTTP.getTextWithScript[funcname];
    }

    // Создать URL-адрес, содержимое которого требуется получить, и имя функции
    // в качестве аргументов серверного сценария jsquoter.php. Установить
    // значение свойства src тега <script>, чтобы получить требуемый URL-адрес.
    script.src = "jsquoter.php" +
                 "?url=" + encodeURIComponent(url) + "&func=" +
                 encodeURIComponent("HTTP.getTextWithScript." + funcname);
}

// Этот счетчик используется для генерации уникальных имен функций обратного
// вызова на случай, если потребуется запланировать выполнение нескольких
// запросов одновременно.
HTTP.getTextWithScript.counter = 0;
```