Learning JavaScript

第2版
涵盖 Ajax 与 DOM

JavaScript

学习指南

[美] *Shelley Powers* 著
李荣青 吴兰陟 申来安 译

O'REILLY®
人民邮电出版社
POSTS & TELECOM PRESS

JavaScript 学习指南（第 2 版）

[美]Shelley Powers 著

李荣青 吴兰陟 申来安 译

人民邮电出版社

北京

图书在版编目（CIP）数据

JavaScript学习指南 : 第2版 / (美) 鲍尔斯(Powers,S.) 著 ; 李荣青, 吴兰陟, 申来安译. -- 北京 : 人民邮电出版社, 2009.10
ISBN 978-7-115-21404-1

Ⅰ. ①J… Ⅱ. ①鲍… Ⅲ. ①JAVA语言－程序设计－指南 Ⅳ. ①TP312-62

中国版本图书馆CIP数据核字(2009)第162728号

JavaScript 学习指南(第 2 版)

◆ 著 [美]Shelley Powers
译 李荣青 吴兰陟 申来安
责任编辑 刘映欣

◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街 14 号
邮编 100061 电子函件 315@ ptpress. com. cn
网址 http://www. ptpress. com. cn
北京铭成印刷有限公司印刷

◆ 开本：800 × 1000 1/16
印张：22. 25
字数：409 千字 2009 年 10 月第 1 版
印数：1 - 3 000 册 2009 年 10 月北京第 1 次印刷
著作权合同登记号 图字：01-2009-1824 号
ISBN 978-7-115-21404-1

定价：45. 00 元

读者服务热线：(010)67132705 印装质量热线：(010)67129223
反盗版热线：(010)67171154

计算机精品学习资料大放送

软考官方指定教材及同步辅导书下载 | 软考历年真是解析与答案

软考视频 | 考试机构 | 考试时间安排

Java 一览无余：**Java** 视频教程 | **Java SE** | **Java EE**

.Net 技术精品资料下载汇总：**ASP.NET** 篇

.Net 技术精品资料下载汇总：**C#**语言篇

.Net 技术精品资料下载汇总：**VB.NET** 篇

撼世出击：**C/C++**编程语言学习资料尽收眼底 电子书+视频教程

Visual C++(VC/MFC)学习电子书及开发工具下载

Perl/CGI 脚本语言编程学习资源下载地址大全

Python 语言编程学习资料(电子书+视频教程)下载汇总

最新最全 **Ruby**、**Ruby on Rails** 精品电子书等学习资料下载

数据库精品学习资源汇总：**MySQL** 篇 | **SQL Server** 篇 | **Oracle** 篇

最强 **HTML/xHTML**、**CSS** 精品学习资料下载汇总

最新 **JavaScript**、**Ajax** 典藏级学习资料下载分类汇总

网络最强 **PHP** 开发工具+电子书+视频教程等资料下载汇总

UML 学习电子资下载汇总 软件设计与开发人员必备

经典 **LinuxCBT** 视频教程系列 **Linux** 快速学习视频教程一帖通

天罗地网：精品 **Linux** 学习资料大收集(电子书+视频教程) **Linux** 参考资源大系

Linux 系统管理员必备参考资料下载汇总

Linux shell、内核及系统编程精品资料下载汇总

UNIX 操作系统精品学习资料<电子书+视频>分类总汇

FreeBSD/OpenBSD/NetBSD 精品学习资源索引 含书籍+视频

Solaris/OpenSolaris 电子书、视频等精华资料下载索引

内 容 提 要

本书系统地介绍了 JavaScript 的基本语法、基本对象、调试工具与排错技术、事件处理机制、浏览器对象模型/文档对象模型（BOM/DOM）等方面的知识，并通过一个复杂的示例深入探讨了 Ajax 应用。本书提供了许多简单易懂、主题鲜明的示例，介绍了大量最佳实践和良好编程习惯，对提高代码可读性、可维护性均有很高的价值，并且对很多跨浏览器兼容问题进行了详细说明，追踪了新规范的发展。

本书适合于希望通过 JavaScript 为自己的网页/网站添加活力的读者，不管你是否有编程经验，通过阅读本书都能够很快地掌握这一技术。在阅读本书之前，最好对 CSS、HTML/XHTML 有所了解。

O'Reilly Media, Inc.介绍

为了满足读者对网络和软件技术知识的迫切需求，世界著名计算机图书出版机构 O'Reilly Media, Inc.授权人民邮电出版社，翻译出版一批该公司久负盛名的英文经典技术专著。

O'Reilly Media, Inc.是世界上在 UNIX、X、Internet 和其他开放系统图书领域具有领导地位的出版公司，同时也是联机出版的先锋。

从最畅销的 *The Whole Internet User's Guide & Catalog*（被纽约公共图书馆评为 20 世纪最重要的 50 本书之一）到 GNN（最早的 Internet 门户和商业网站），再到 WebSite（第一个桌面 PC 的 Web 服务器软件），O'Reilly Media, Inc.一直处于 Internet 发展的最前沿。

许多书店的反馈表明，O'Reilly Media, Inc.是最稳定的计算机图书出版商——每一本书都一版再版。与大多数计算机图书出版商相比，O'Reilly Media, Inc.具有深厚的计算机专业背景，这使得 O'Reilly Media, Inc.形成了一个非常不同于其他出版商的出版方针。O'Reilly Media, Inc.所有的编辑人员以前都是程序员，或者是顶尖级的技术专家。O'Reilly Media, Inc.还有许多固定的作者群体——他们本身是相关领域的技术专家、咨询专家，而现在编写著作，O'Reilly Media,Inc.依靠他们及时地推出图书。因为 O'Reilly Media, Inc.紧密地与计算机业界联系着，所以 O'Reilly Media, Inc.知道市场上真正需要什么图书。

前言

JavaScript 最初的设计意图是为了在浏览器端（当时就是 Netscape Navigator）载入的 Web 页面和位于服务器端的应用程序之间提供脚本化的接口。由于它早在 1995 年就出现了，因此已经发展成为 Web 开发的关键组件，当然你也能看到它在其他领域的应用。

本书的主题就是 JavaScript 语言，内容包括从最基本的数据类型（它是所有语言的基础）到最复杂的特性（包括在 Ajax 和动态页面效果中使用的）的所有方面。当你读完本书之后，你将掌握应用精妙的程序库和 Web 应用程序的知识基础。

本书读者

本书假设读者熟悉 Web 页面技术，包括 CSS 和 HTML/XHTML。你可以没有任何编程经验，但对于没有编程经验的你，有些小节可能需要多读几遍。

本书对以下读者将有帮助：

- 希望或需要在个人网站中集成 JavaScript 的人；
- 那些使用如 weblogging 之类的内容管理工具，并且希望深入理解这些工具所提供的各种模板中的脚本化组件的开发人员；
- 希望将 JavaScript 及一些动态 Web 页面、Ajax 功能集成到自己网站中的 Web 开发人员；
- 希望针对新的客户端市场开发 Web Service 的开发人员；
- 关注 Web 技术或者教授 Web 技术课程的老师；
- 想更好地理解如何才能为自己的设计添加交互性或动态效果的 Web 页面设计师；
- 对 Web 技术感兴趣的任何人。

假设与方法

前面说过，本书将假设你在 HTML 和 CSS 方面有一定经验，同时对 Web 应用程序的工作机制有基本的理解。编程经验并不是必要的，不过由于本书涉及 JavaScript 的方方面面，因此有些部分可能会比较复杂难解。虽然很难的内容并不多，但如果要使用新的 Ajax 程序库，还是需要对 JavaScript 有足够的理解。

开发环境

使用 JavaScript 是件很有挑战性的事情，因为你的应用程序不仅将运行在不同类型的计算机上，还将运行在不同的浏览器上。如果仔细研究一个 Web 服务器记录的网站日志文件，就会发现有使用诸如 Firefox 3.0 或 IE 8.0 之类的新浏览器的用户，但也有使用如 IE 5.0 之类的早期浏览器的用户。

当然也可以尝试创建能够应对不同操作系统和浏览器的 JavaScript 程序，不过更好的选择是只针对一部分目标浏览器，根据访问你的 Web 页面的用户最常用的浏览器来选择，然后再用它们来测试你的应用程序。你可能会发现你的应用程序无法在旧版本浏览器上正常工作，不过也不需要对所有环境都提供支持。

在本书中，当谈及某个 JavaScript 程序块时可能会提及“目标浏览器”。对于本书而言，我针对的目标浏览器是 Firefox 3.x、Opera 9.x、Safari 3.x（包括最新的 WebKit 版本，它是 Safari 的基础），以及 IE 8.0 预览版（IE 下一个要发布的版本）。绝大多数的示例在 IE 7.x 和 IE 6.x 中也都能够正常运行，如果不能正常运行，我会做一些说明。以下是下载这些浏览器的 URL 列表。

- 在 http://www.mozilla.com/en-US/firefox/上可以下载到 Firefox。
- 在 Mac OS X 操作系统上已经安装了 Safari，不过 Mac 和 Window 操作系统的用户都可以在 http://www.apple.com/safari 上下载到相应的安装版本。Safari 是基于开源 WebKit 项目的，该项目在 http://webkit.org/上提供了用于测试的每日构建版本。
- 在 http://www.opera.com/上可以下载到 Opera。
- IE 是内建在 Windows 操作系统中的，如果需要下载 IE 8.0 beta 版本，可以访问 http://www.microsoft.com/windows/internet-explorer/beta/default.aspx。

JavaScript 和浏览器的发展过程是动态的，这也为编写一本关于 JavaScript 的书带来了特殊的挑战。虽然我尽力在本书中包含了 JavaScript 的最新更新，但 JavaScript 规范（更准确地说应该是 ECMAScript 规范）和浏览器都在不断地发生重大变化。例如，在本书处于编辑阶段时，ECMAScript 工作组宣布了不再继续研发 JavaScript 2 项目，转而关注新的 ECMAScript 3.1 规范。不过，新的 ECMAScript 规范中所带来的变化在各种目标浏览器上不会马上得以实现。当然我确信这一规范中引入的功能将会在浏览器的后续版本中陆续实现，因此我在本书中添加了一些“提示”，简要说明这些即将来临的变化。

另外，浏览器开发商也在一直发布新版本。本书中用来测试示例的目标浏览器所展现的状态是我写书时的表现情况，当你阅读本书时运行结果应该不会受到太大影响。

不过，我关注的绝大多数素材都是“传统”（classic）的 JavaScript，它不仅稳定，而且是所有浏览器和脚本语言修改时所依托的平台。本书中的绝大多数（虽然不是所有）

示例都能够在老版本和后续版本的浏览器中正常工作，就像用来测试这些示例的目标浏览器一样。

祝你好运。

本书的组织形式

本书是由 6 个相对独立的部分组成的。

第 1～3 章概述了 JavaScript 应用程序的结构，包括其支持的简单数据类型，以及基本语句和程序控制结构。这部分将帮助你对后续章节中使用的语言建立最基本的理解。

第 4 章和第 5 章介绍了 JavaScript 中的主要对象，即 String、Number 和 Boolean，以及其他内建的对象，如 Math、RegExp（针对正则表达式）、Array，还有所有重要的函数。

第 6 章的讨论暂时离开了语言本身，介绍了一些浏览器调试工具和排错技术，为阅读本书后续章节中更复杂的脚本做些准备。

第 7 章介绍了事件处理机制，第 8 章详细描述了带表单的 JavaScript 应用程序中可能涉及的表单事件。

第 9～11 章深入研究了 Web 页面开发中更复杂难解的部分。这些章节介绍了浏览器对象模型（BOM）和比其更新的文档对象模型（DOM），同时说明了如何创建自定义对象。如果你要创建一个新窗口，或者要访问、修改甚至动态创建页面上的元素，理解这些模型是十分重要的。另外，通过使用自定义对象，可以实现语言或浏览器内建功能之外的功能。这些章节中还介绍了浏览器端的 cookie 以及更新的客户端存储技术。

第 12～15 章研究了 JavaScript 的高级应用，包括动态页面效果和 Ajax，并且详细地介绍了 XML 和 JavaScript 对象表示法（JSON）在 Ajax 应用程序中的应用。

虽然我希望在介绍 JavaScript 时遵循一条合理的线索，但有时仍然会在示例中用到一些必须在后续章节中才会详细介绍的功能。当出现这种情况时，我会注明在哪些章节中会提供更进一步的说明。

章节划分

以下是本书中各章节的详细说明，对每一章涉及的内容分别做简要的描述。

- 第 1 章“Hello JavaScript!” 本章简要介绍了 JavaScript 语言，同时分析了一个简单的 Web 页面应用程序。本章还讲解了与使用 JavaScript 相关的要点，包括针对 JavaScript 应用程序的良好编程实践。
- 第 2 章“JavaScript 数据类型和变量” 本章概述了 JavaScript 中的基本数据类型，同时介绍了其变量、标识符以及 JavaScript 语句的结构。

- 第 3 章“操作符和语句” 本章介绍 JavaScript 中的基本语句，包括赋值语句、条件分支语句及控制语句，同时还介绍了所需的各类操作符。
- 第 4 章“JavaScript 对象” 本章介绍了 3 种基本的 JavaScript 对象，包括 Number、String 和 Boolean，同时也介绍了 Date 和 Math 对象。本章还介绍了 RegExp 对象，它是用来实现模式匹配的关键对象。
- 第 5 章“函数” 本章重点介绍了 JavaScript 中内建的函数这一重要的对象。函数是创建自定义对象的关键，也是将 JavaScript 程序块封装成可复用功能的基础，这样在应用程序中就可以多次调用这些程序块。
- 第 6 章“排错、调试及跨浏览器问题” 本章简要介绍了本书中各种目标浏览器（IE、Safari、Firefox 和 Opera）所提供的调试环境，同时探讨了跨浏览器开发方面的问题。
- 第 7 章“捕获事件” 本章关注于事件处理，内容包括最初的基于表单的事件处理（它在许多应用程序中仍然很常用），以及更新的基于 DOM 的事件处理。
- 第 8 章“表单、表单事件和校验” 本章介绍 JavaScript 在表单和表单域方面的应用，包括如何访问各种类型的表单域，如文本输入域、下拉列表等，以及如何对表单域的数据进行校验。在表单提交给 Web 服务器之前，对其数据进行校验有助于避免出现不必要的服务器通信，从而节省时间和资源。本章还简要地说明一些与安全相关的话题。
- 第 9 章“浏览器就像是个难题箱” 本章将介绍通过 JavaScript 访问页面时需要使用的对象模型。首先是浏览器对象模型（BOM），它是一个层次化的对象结构，包括 window、document、forms、history、location 等对象。通过 BOM，JavaScript 能够打开新窗口，访问如表单、链接、图像等页面元素，甚至还能够创建一些基本的动态效果。
- 第 10 章“cookie 和其他客户端存储技术” 本章介绍基于脚本的、用来在客户端计算机上存储少量数据的 cookie。通过 cookie 可以保存用户名、密码及其他相关信息，这样用户就不必每次输入相同的数据。另外，本章还概要地介绍了一些新出现的客户端存储技术，如 Google 的 Gears 和 HTML 5.0 中的本地存储，它们都提供了比 cookie 更强的功能。本章还回顾了 JavaScript 沙箱方面的知识。
- 第 11 章“DOM 或以树形展示的 Web 页面” 本章关注于 DOM，它是一个直观、重要的对象模型，提供了访问所有文档元素和属性的机制。虽然该模型十分全面，也十分直观易懂，但本章对于新程序员而言仍然存在很多挑战。
- 第 12 章“动态页面” 本章全面介绍了动态修改 Web 页面的方法，包括修改某个元素的样式，以及在页面上添加或删除元素。本章中还研究了如拖放、页面区域的展开/折叠、可见性、动画等效果。阅读本章之前，需要对 CSS 有基本的理解。

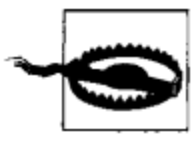
- 第 13 章“创建自定义 JavaScript 对象” 本章展示在 JavaScript 中创建自定义对象的方法，介绍了 JavaScript 中实现自定义对象支持的原型（prototype）机制。在此还将介绍一些编程语言的概念，如继承、封装，不过即使没有理解这些概念，也可以从本章中获益。
- 第 14 章“使用 Ajax” 本章介绍了 Ajax，虽然它能够创建令人兴奋的效果，但实际上并不是一个很复杂的 JavaScript 应用。本章剖析一个复杂的示例，该示例中带有服务器端程序代码。
- 第 15 章“Ajax 数据：XML 或 JSON” 在第 14 章中用来演示 Ajax 功能的示例是基于 HTML 片段完成数据交换的，在本章中将说明如何在 Ajax 应用程序中生成和处理 XML，然后再说明如何用 JSON 完成相同的任务。我们会介绍这些技术的优点，并说明何时应该使用哪种技术。

阅读须知

本书中使用了以下排版约定：

- 等宽字体表示是命令行的输入、要逐字输入的选项、C#关键字或代码示例；
- 等宽的斜体字在语法或代码中表示可替换的项目，如变量、可选元素等；
- 等宽的粗体字用于强调程序中的一段代码；
- 斜体字用来表示路径名、文件名、因特网地址（如域名、URL）以及新定义的术语。

提示
表示技巧、建议或提示。



警告
表示警告和告诫。

使用代码示例

本书的目标是帮你完成自己的工作。一般来说，你可以自由地在自己的程序和文档中使用本书中的代码，无须向我申请授权，除非要在新的商品中使用本书的一部分代码。例如，在编写程序时使用了本书中的几段代码是无须申请授权的。但销售或发布从本书中获取的示例就需要申请授权了。在回答别人问题时，引用本书中的文字和代码也无须申请授权。将本书中的示例代码放到你的产品文档中就需要申请授权了。

我们重视但不强制使用归属说明。归属说明通常包括书名、作者、出版社和 ISBN，例如：*Learning JavaScript*, Second Edition, by Shelley Powers. Copyright 2009 Shelley

Powers, 978-0-596-52187-5。

如果你感觉在使用这些示例代码时超出了正当权限，或者超出了这里的许可，可以发邮件到 permissions@oreilly.com 与我们联系。

如何联系我们

对于本书中的信息，我们已经尽最大努力进行了测试和校验，但你仍然可能会发现一些特性已经发生了变化（甚至是我们自身的错误）。如果你发现任何不妥之处，或者对后续版本有任何建议，请按以下联系方式与我们联系。

美国：

O'Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

中国：

100080 北京市西城区西直门成铭大厦 C 座 807 室

奥莱利技术咨询（北京）有限公司

如果想询问一些技术问题，或者对本书做些评论，请发邮件到：

bookquestions@oreilly.com

本书还有一个专门的 Web 页面，上面有本书的示例代码以及后续版本的写作计划。可以通过以下网址访问这些信息：

http://www.oreilly.com/catalog/9780596521875

关于本书、相关会议、资源中心以及 O'Reilly Network 相关的信息，请访问 O'Reilly 网站（http://www.oreilly.com）。

致谢

我想感谢我的编辑和技术审校团队，包括负责本书技术审校的技术编辑 Tony Ruscoe、Jeni Tennison、Matthew Russell 和 Trey Holdener，还有和我长期合作的编辑 Simon St.Laurent，你们的努力使本书变得更好。另外，我还得感谢出版团队的其他成员：Rachel Monaghan、Sumita Mukherji、Joe Wizda 和 Jessamyn Read。

计算机精品学习资料大放送

软考官方指定教材及同步辅导书下载 | 软考历年真是解析与答案

软考视频 | 考试机构 | 考试时间安排

Java 一览无余：**Java** 视频教程 | **Java SE** | **Java EE**

.Net 技术精品资料下载汇总：**ASP.NET** 篇

.Net 技术精品资料下载汇总：**C#**语言篇

.Net 技术精品资料下载汇总：**VB.NET** 篇

撼世出击：**C/C++**编程语言学习资料尽收眼底 电子书+视频教程

Visual C++(VC/MFC)学习电子书及开发工具下载

Perl/CGI 脚本语言编程学习资源下载地址大全

Python 语言编程学习资料(电子书+视频教程)下载汇总

最新最全 **Ruby**、**Ruby on Rails** 精品电子书等学习资料下载

数据库精品学习资源汇总：**MySQL** 篇 | **SQL Server** 篇 | **Oracle** 篇

最强 **HTML/xHTML**、**CSS** 精品学习资料下载汇总

最新 **JavaScript**、**Ajax** 典藏级学习资料下载分类汇总

网络最强 **PHP** 开发工具+电子书+视频教程等资料下载汇总

UML 学习电子资下载汇总 软件设计与开发人员必备

经典 **LinuxCBT** 视频教程系列 **Linux** 快速学习视频教程一帖通

天罗地网：精品 **Linux** 学习资料大收集(电子书+视频教程) **Linux** 参考资源大系

Linux 系统管理员必备参考资料下载汇总

Linux shell、内核及系统编程精品资料下载汇总

UNIX 操作系统精品学习资料<电子书+视频>分类总汇

FreeBSD/OpenBSD/NetBSD 精品学习资源索引 含书籍+视频

Solaris/OpenSolaris 电子书、视频等精华资料下载索引

目录

计算机精品学习资料大放送

软考官方指定教材及同步辅导书下载 | 软考历年真是解析与答案

软考视频 | 考试机构 | 考试时间安排

Java 一览无余：**Java** 视频教程 | **Java SE** | **Java EE**

.Net 技术精品资料下载汇总：**ASP.NET** 篇

.Net 技术精品资料下载汇总：**C#**语言篇

.Net 技术精品资料下载汇总：**VB.NET** 篇

撼世出击：**C/C++**编程语言学习资料尽收眼底 电子书+视频教程

Visual C++(VC/MFC)学习电子书及开发工具下载

Perl/CGI 脚本语言编程学习资源下载地址大全

Python 语言编程学习资料(电子书+视频教程)下载汇总

最新最全 **Ruby**、**Ruby on Rails** 精品电子书等学习资料下载

数据库精品学习资源汇总：**MySQL** 篇 | **SQL Server** 篇 | **Oracle** 篇

最强 **HTML/xHTML**、**CSS** 精品学习资料下载汇总

最新 **JavaScript**、**Ajax** 典藏级学习资料下载分类汇总

网络最强 **PHP** 开发工具+电子书+视频教程等资料下载汇总

UML 学习电子资下载汇总 软件设计与开发人员必备

经典 **LinuxCBT** 视频教程系列 **Linux** 快速学习视频教程一帖通

天罗地网：精品 **Linux** 学习资料大收集(电子书+视频教程) **Linux** 参考资源大系

Linux 系统管理员必备参考资料下载汇总

Linux shell、内核及系统编程精品资料下载汇总

UNIX 操作系统精品学习资料<电子书+视频>分类总汇

FreeBSD/OpenBSD/NetBSD 精品学习资源索引 含书籍+视频

Solaris/OpenSolaris 电子书、视频等精华资料下载索引

第 1 章

Hello JavaScript!

JavaScript 之所以如此流行，一个重要的原因是 JavaScript 能与网页完美集成。编写 JavaScript 代码，只需要在网页中添加一个 script 元素，将 type 属性指定为“text/javascript”，然后写入你的 JavaScript 代码：

```
<script type="text/javascript">
...some JavaScript
</script>
```

使用 JavaScript 不需要安装任何额外组件，也没有配置组件的路径等烦琐的工作，可以直接在大多数浏览器上运行，如 Firefox、IE、Opera 及 Safari。要做的只是添加一个 script 块，写入代码，仅此而已。

大多数教科书都会要求将 JavaScript 代码放在 head 元素中，而事实上，JavaScript 代码也可以放在 body 元素中，甚至可以同时在 head 元素和 body 元素中添加 JavaScript 代码。然而，在 body 元素中添加 JavaScript 代码并不是最佳实践，这种做法往往会给阅读、修改 JavaScript 带来麻烦。唯一的好处是可以解决性能问题，本书将在第 6 章中介绍如何解决这种问题。在本书的所有示例中，都将把 JavaScript 代码添加在 head 元素中。

1.1 “Hello World!”程序

在学习一门新语言时，第一个示例通常都是编写“Hello World”程序，也就是一个输出字符串“Hello, World!”的程序。对于 JavaScript 而言，“Hello World”程序则是要在网页中显示“Hello, World!”。如示例 1.1 所示，网页中 JavaScript 代码块只有一行代码，弹出一个 alert 对话框，并显示“Hello, World!”。

示例 1.1　最简单的 JavaScript 程序“Hello, World!”

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0
```

```
Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
<title>Hello, World!</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />

<script type="text/javascript">

alert("Hello, World!");

</script>
</head>
<body>
</body>
</html>
```

将示例 1.1 的代码复制到文件中，并在浏览器中打开，就能看到显示“Hello, World!”的 alert 对话框。如果没看到该对话框，请检查浏览器是否启用了 JavaScript。

提示

老版本的 IE 浏览器在这种情况下会自动禁用 JavaScript，请尝试使用访问网页地址的方式，而不是采用打开文件的方式，例如 http://<somedomain.com>/index.html。

尽管这个程序的功能非常简单，然而麻雀虽小，五脏俱全，通过该程序可以了解构成 JavaScript 程序的最小内容集：网页、script 元素及 JavaScript 代码。尝试自己修改 JavaScript 代码，将“World”替换成你的名字。

如果希望在浏览器中输出静态的信息，那么请跟我看第二个示例。

1.2 第二个“Hello World!”程序

和第一个“Hello World!”程序不同，第二个“Hello World!”程序是在网页中输出信息。该程序中用到了 JavaScript 程序的 4 个重要组成部分：浏览器内置的 document 对象、JavaScript 变量、JavaScript 函数及事件句柄，如示例 1.2 所示。

示例 1.2 第二个“Hello World!”程序

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>Hello, World!</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />

<script type="text/javascript">
function hello() {
```

```
    // 向世界问好
    var msg = "Hello, World!";
    document.open();
    document.write(msg);
    document.close();
}
</script>
</head>
<body onload="hello()">
<p>Hi</p>
</body>
</html>
```

尽管第二个“Hello World!”程序还是很简单，但是已经涵盖了大多数 JavaScript 应用程序的基本要素。在本章余下的内容中，我们将详细地介绍 JavaScript 程序中的每一个基本要素。

提示

本章并不打算介绍在示例 1.1 和示例 1.2 中使用的文档类型声明（DOCTYPE），它会对浏览器处理 JavaScript 代码的方式有些影响。我们将在第 6 章介绍 DOCTYPE 的影响。

1.2.1 script 标签

JavaScript 经常在其他语言的上下文中使用，例如在 HTML 及 XHTML 等标记语言中使用。然而，JavaScript 并不能简单地直接添加到这些标记语言中。

在示例 1.2 中，JavaScript 代码被封装在 script 元素中，所以当浏览器读取到 script 元素时，就不会以 HTML 或 XHTML 的方式处理其内容，而是通知浏览器的脚本引擎来接管 script 元素中的内容。

嵌入网页中的脚本并不一定都是 JavaScript，script 元素的 type 属性定义了脚本类型。在本示例中的脚本类型是 text/javascript，其他可能的属性值是：

- text/ecmascript
- text/jscript
- text/vbscript
- text/vbs

第一种可选值 ecmascript 表示以 ECMAScript 方式（基于 ECMA-262 脚本标准）解释这段脚本。第二种可选值 jscript 表示以 JScript 方式（微软在 IE 浏览器中所实现的 ECMAScript 语言的一种变种）解释这段脚本。最后两种可选值表示以微软的 VBScript 方式处理，这是一种完全不同的脚本语言。

所有这些可选类型值都表示内容是基于 MIME 编码的。MIME 是一种内容编码的方式。基于 MIME 编码，浏览器就可以处理这种类型，或者跳过该段代码，从而确保只有处理该脚本的应用程序才能访问脚本。

早期的 script 标签中还有 language 属性，用来表示语言的版本，如 javascript 1.1 或 javascript 1.2。然而，在 HTML 4.01 标准中已经弃用了 language 属性，不过在许多 JavaScript 示例中还是能够看到 language 属性，这也是一种早期的跨浏览器技术。

提示

跨浏览器表示 JavaScript 可以在不同的浏览器上运行，或者消除不同浏览器之间的差异，本书称为 JavaScript 程序可以跨浏览器使用。

几年前，在解决跨浏览器兼容问题时有一种常见的做法，那就是为不同的浏览器创建相应的 script 元素，并引用不同的文件或代码，然后使用 language 属性确保只有兼容的浏览器才能够访问该段代码。下面是一个在 1997 年编写的示例：

```
<script src="ns4_obj.js" language="javascript1.2">
</script>
<script src="ie4_obj.js" language="jscript">
</script>
```

这种方法的工作原理是，如果浏览器支持 JavaScript1.2（如 Netscape Navigator 4.x），那么就会执行 ns4_obj.js 文件中的代码；而如果浏览器支持 Jscript（如 IE 4.0），那么将执行 ie4_obj.js 文件中的代码。你可能觉得这种做法很诡异，的确是这样，不过这在当时的确解决了动态脚本的跨浏览器兼容问题。

script 的其他属性包括 src、defer 及 charset。charset 属性定义了脚本的字符编码集。一般情况下不需要设置 charset，除非脚本需要采用与文档完全不同的字符编码集。

另外一个有用的属性是 defer。如果将 defer 属性设置为“defer”，那么表示该脚本不会生成任何文档内容，于是浏览器可以提前处理页面的剩余部分，在页面处理结束并做好显示准备时才处理脚本部分。

```
<script type="text/javascript" defer="defer">
...no content being generated
</script>
```

defer 属性可以提高页面载入的速度，特别是那些引用了大量的 JavaScript 代码或者庞大的 JavaScript 程序库的页面。

最后一个 src 属性用来表示所需要载入的外部 JavaScript 文件。不过我们首先将介绍 text/javascript 类型属性，以及它在不同浏览器中的含义。

为 body 添加脚本

前面我们曾经说过，因为对 script 元素进行集中管理有利于网页的可维护性，所以 script 元素通常将添加在网页上的 head 元素中。然而，在 body 元素中添加脚本的原因往往是出于性能的考虑。

因为浏览器从同一个域名并发载入的资源是有限制的，所以，当把脚本添加到 head 元素中时，首先载入的将是脚本，其次才是文档的剩余部分。此外，浏览器可能会延迟页面剩余部分的显示，因为脚本中可能会调用 document.write 方法修改 document 对象。如果 JavaScript 文件很庞大，那么网页中的图片以及其他重要的信息将会被延迟显示，这所带来的问题远比可维护性更加重要。

即使在 script 元素中使用 defer 属性也不一定能完全解决该问题，特别是并发资源访问和页面显示的限制。

在 *High Performance Web Sites*（中译版《高性能网站建设指南》一书中，作者推荐将 script 元素放在文档的最末尾处，这样网页的其他部分就可以优先载入。大多数复杂网站的开发人员更倾向于这种方法。这种方法带来的负面影响是脚本不容易查找，网页的可维护性也较差。

那么什么才是最佳方法呢？我发现大多数网站并不引用庞大的 JavaScript 程序库，在保证较好性能的前提下，将脚本放在 head 元素中，也确保网页可维护性的优势。不过，如果的确需要使用庞大的 JavaScript 程序库，那么可以考虑将脚本放在页面的最末尾处。

不论采用何种方法，请确保脚本位置的一致性，要么全部放在 head 元素中，要么全部在 body 元素的最末尾处。

1.2.2 JavaScript、ECMAScript 和 JScript 的比较

示例 1.2 中的 script 元素的类型是 text/javascript，该程序可以在 Firefox、IE、Opera 以及 Safari 中正常运行。然而，并不是所有的浏览器都支持 JavaScript。

尽管“JavaScript”已经成为浏览器客户端脚本的代名词，但是只有 Mozilla 和流行的 Mozilla 浏览器 Firefox 实现了 JavaScript 语言，其实际名称是广泛应用的客户端脚本语言标准 ECMAScript，它的最新的版本是 ECMA-262 第 3 版。

然而，大多数浏览器都支持 text/javascript，除此之外还兼容 text/ecmascript，只不过不同浏览器对这两种类型的支持存在着一些差异。

提示

ECMAScript 并不局限于浏览器，Adobe 的 ActionScript（Flash 中的脚本语言）也是基于 ECMA-262 第 3 版的。

本书中所有程序都在 Firefox 3.x、Safari 3.x、Opera 9.x 以及 IE 8.0 中做过测试，虽然它们对 ECMAScript-262 第 3 版提供了基本支持，但并不是全部，而且 ECMAScript 也已经有了下一代——ECMAScript 3.1。本书还将介绍各种浏览器之间的差异以及相应解决方案。同时也将使用大家最为熟悉的 text/javascript 作为 script 元素的类型属性，如示例 1.2 所示。

1.2.3 在 JavaScript 中定义函数

示例 1.2 中的 JavaScript 代码创建了用来显示“Hello, World!”的函数 hello。函数是复用脚本的一种方式，它可以重复运行多次。函数也可以用来控制何时执行所引用的脚本。例如示例 1.2 只在页面载入后调用该函数。

下面是创建函数的典型语法：

```
function functionname(params) {
    ...
}
```

首先是关键字 function，后面紧跟着函数名、圆括号，圆括号中包含零个或多个参数（函数参数）。示例 1.2 中的 JavaScript 函数没有参数，然而在本书中的其他示例里就可以看到大量带参数的函数。函数的主体脚本则放在大括号中。

之所以说这是函数的“典型”语法，是因为它并不是创建函数的唯一语法。本书将在第 3 章中对 JavaScript 函数做更详细的介绍。

当然，有了函数后，还需要调用函数才能执行函数内的脚本，这样也就引出了事件句柄。

1.2.4 事件句柄

在示例 1.2 中的 body 元素里面，我们将 hello 函数赋值给一个名为 onload 的 HTML 属性。onload 属性就是众所周知的事件句柄。像这样的事件句柄，是浏览器提供的底层对象模型的一部分。

事件句柄可以将函数与某个特定事件关联起来，当事件触发时，就会执行该函数中的脚本。最常见的一种事件句柄就是 body 元素的 onload 事件，当网页载入结束时就会触发该事件，事件句柄也就将调用相应的函数。

下列是一些常见的事件句柄：

- onclick　当鼠标单击某元素时触发；
- onmouseover　当鼠标移到某元素上时触发；
- onmouseout　当鼠标离开某元素时触发；
- onfocus　当某元素获得焦点时触发（通过键盘或鼠标）；

- onblur　当某元素失去焦点时触发。

这些只是事件句柄中的一小部分而已，并且不是所有元素都支持所有的事件句柄。例如，只有一部分 HTML 元素支持 onload 事件句柄，如 body 和 img 元素；不要大惊小怪，因为事件是与某种资源的载入相关联的。

在 HTML 元素中直接添加是添加事件句柄的一种方法。还有一种方法是直接在 JavaScript 中添加如下代码：

```
<script type="text/javascript">
window.onload=hello;

function hello() {

   // 向世界问好
   var msg = "Hello, World!";
   document.open();
   document.writeln(msg);
   document.close();
}
</script>
```

onload 事件句柄是浏览器内置对象 window 的一个属性。在脚本的第一行中，我们将函数 hello 赋给了 window 对象的 onload 事件句柄。

提示

在 JavaScript 语法中，JavaScript 函数也是对象，所以可以通过名字或直接将函数赋给一个变量或另一个对象的属性。

基于对象属性的方法，就无须将事件句柄作为属性添加到元素中，而是可以在 JavaScript 代码中直接添加。第 7 章将更加详细地介绍事件句柄及其高级用法。现在，我们先简单看看 document 对象。

1.2.5　浏览器对象 document

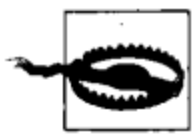
示例 1.2 中使用了浏览器中功能最强大的对象之一——document 对象。document 对象用来呈现整个页面，包括页面中的所有元素。通过 document 对象可以访问页面中的所有内容，正如你所见，基于 document 对象也可以修改页面的内容。

document 对象中还包含对应于页面元素的集合，如页面中所有的图像或窗体元素。document 对象中还提供了访问及修改页面的方法，如示例 1.2 中所使用的 open、writeln 及 close 方法。

open 方法可以打开要修改的 HTML 页面。在示例 1.2 中，JavaScript 脚本打开了包含脚本的同一个页面。writeln 方法是 write 方法的变种，可以输出文本到页面中。write 和 writeln 方法之间的唯一区别是 writeln 在输出文本之后会自动添加换行符。close 方法用

来关闭页面，并强制浏览器立即刷新页面内容。

当页面载入后，向现有页面中写入新内容会使页面之前的内容被覆盖。这也就是为什么打开页面后只看到“Hello, World!”，而看不到“Hi”。



警告

在 IE 浏览器（IE8 beta 版）中，如果向现有页面中写入新内容会引发另一个问题，即导致“后退”按钮丧失其功能。

在示例 1.2 中，open 和 close 方法并不是必需的方法，因为在页面载入之后，浏览器在调用 writeln 方法时会自动打开、关闭页面。如果是在页面载入过程中使用该脚本，那么就需要显式地调用 open 方法。

document 对象和之前所提及的 window 对象一样，都是浏览器内置对象结构（也就是浏览器对象模型，即 BOM）的一部分。BOM 是现在大多数浏览器实现的基本对象集合。第 9 章将详细介绍 document 对象以及其他 BOM 对象。

提示

BOM 是更为正式的文档对象模型（DOM）的前身，俗称为 DOM 0。

1.2.6 属性操作符

在示例 1.2 中，通过 JavaScript 语言中的属性操作符（.）就可以访问 document 对象中的方法。

JavaScript 中能支持的操作符有许多，包括算术运算符（+和-）、条件表达式（<和>）以及稍后会介绍到的其他运算符。在这些操作符中，最重要的一种类型就是属性操作符。数据元素、事件句柄以及对象方法在 JavaScript 语言中都被看做是对象的属性，都可以通过属性操作符访问。

属性操作符还可以通过俗称方法链的方式进行调用，在同一个语句中，可以一个连着一个地连续调用多个方法。在本书中你将会看到这样的示例：

```
var tstValue = document.getElementById("test").style.backgroundColor
="#ffffff";
```

在该示例中，通过 document 的 getElementById 方法访问 page 元素，然后通过访问 style 对象来设置元素的背景色。backgroundColor 是 style 对象的一个属性，style 对象是 page 元素的一个属性，而 page 元素则可以通过 getElementById 方法访问，getElementById 又是 document 对象的一个属性。

在后续章节中还会介绍这些方法和对象，现在我们只是了解一下方法链的调用方式。如果要返回对象的所有属性，则不能使用方法链，只有返回单个对象时才可以。

提示

在目前最流行的一个 Ajax 程序库 jQuery 中就大量应用了方法链。在后续的章节中将会简要地介绍 jQuery 程序库。

1.2.7 var 关键字和作用域

在示例 1.2 中，字符串“Hello, World!”赋给了 msg 对象，这是个 JavaScript 变量的示例。变量是对一个数据的命名引用。该数据可以是示例 1.2 中的这样一个字符串，也可以是一个数字或者布尔值 true 或 false，还可以是函数引用、数组或者另一个对象。

关键字 var 用于定义变量。如果使用 var 定义变量，那么该变量就是一个局部变量，也就是只能在定义变量的函数内使用。如果不使用 var，那么 msg 变量就将是全局变量，你可以在任意地方访问该变量。在局部上下文使用全局变量不一定是坏事，在某些情况下也的确需要这样用，然而这并不是一种好的实践，应当尽可能避免这样的用法。

之所以要避免使用全局变量，是因为如果应用程序中有许多个 JavaScript 脚本，那么就可能在某个文件中的某段代码里使用到 msg 变量，这样就会覆盖原有的数值。或者，如果已经有了全局变量 msg，而某些 JavaScript 脚本程序库错误地定义变量（漏了 var 关键字）就会导致其数据丢失。

变量的作用域是非常重要的，特别是当全局变量与局部变量的名字相同的时候。虽然在示例 1.2 中并没有使用全局变量，但是从一开始就保持良好的 JavaScript 编程规范是非常重要的。

变量作用域的规则如下：

- 如果在函数或代码块中使用 var 关键字定义一个变量，那么这是一个属于这个函数或代码块的局部变量；
- 如果使用一个没有用 var 关键字定义的变量，并且存在同名的全局变量，那么该局部变量将等同于已经存在的全局变量；
- 如果使用 var 关键字定义一个变量，但是没有对变量进行初始化（例如为变量赋值），那么它是个局部变量，但是它是未定义的；
- 如果定义变量时没有使用 var 关键字，或者显式地定义为全局变量，但是没有进行初始化，那么这是一个可以全局访问的变量，但是也是未定义的。

只要在函数内使用 var 关键字定义变量，就可以避免全局变量和局部变量同名的问题。特别是使用如 Dojo、jQuery 以及 Prototype 之类的 JavaScript 程序库时，因为程序员并不想要了解其他 JavaScript 代码所使用的变量名。

1.2.8 JavaScript 语句

JavaScript 语言中还提供了处理不同指令的不同语句。在示例 1.2 中使用了最基本的一

种 JavaScript 语句——赋值语句，它用来为变量赋值。其他的语句包括：for 循环语句，基于给定的计数值循环地处理代码块；if-else 条件语句，根据条件判断是否执行某段代码块；switch 条件判断语句，在给定的值当中进行判断，执行与值相关联的代码段。

每种类型的语句都有特定的语法要求。在示例 1.2 中，赋值语句的结尾是个分号。JavaScript 并不强制要求用分号作为语句的结束符，如果一行当中包含许多语句，那么就需要通过分号来区分不同的语句。

当在一行中写入一个完整的语句时，换行符也被认为是语句的结束符。然而，和坚持使用 var 一样，用分号作为语句的结束符也是良好的代码规范，这样的做法能够提高代码的可读性。第 3 章将介绍更多与分号、其他运算符和语句相关的内容。

1.2.9 注释

正如之前所介绍的那样，就像示例 1.2 这样的小型应用程序中也包含了许多 JavaScript 的细节。接下来我们将介绍 JavaScript 的注释。

注释往往是对代码的总结或解释。JavaScript 语言的注释，是解释代码段的含义以及代码依赖性的一种有用方法。注释往往能够提高代码的可读性和可维护性。

注释有两种不同的类型。第一种是使用双斜杠，所有的注释必须在同一行，例如：

```
// 本行是对代码的注释
var i = 1;  // 这是在同一行中使用的注释
```

第二种是使用 JavaScript 注释分隔符，/*和*/，用来注释一行或多行内容，例如：

```
/*  这是一个多行注释，
    它一共占用了三行。
    在注释一个函数时，多行注释特别有用 */
```

相对来说，单行注释比较安全，因为多行注释可能会在分隔符被误删除时引发问题。

一般来说，单行注释用于解释执行特定逻辑的代码块或者创建特定的对象，而多行注释则更常用于 JavaScript 文件的开始处。在每个 JavaScript 代码块、函数或者对象定义的地方写注释是一种良好的代码规范。此外，在所有 JavaScript 程序库文件的开头，都应该提供更为详细的注释块；注释块中应包含作者、日期、依赖的相关信息以及脚本用途等信息。

到目前为止我们已经对示例 1.2 中的 JavaScript 代码逐一做了详细的描述。现在我们就来看看在示例 1.2 中未涉及的其他细节。

1.2.10 你没看到的：HTML 注释和 CDATA 小节

10 年前，当大多数浏览器还处于发展早期阶段时，大家对 JavaScript 的支持情况并不一致，不同的浏览器中有着不同的实现。当浏览器（例如基于文本的 Lynx）遇到 script

脚本的时候，有时只是将脚本内容直接输出到页面上。

为了避免这样的问题，脚本内容通常会放在 HTML 注释（<!--和-->）中，这样即使是不支持 JavaScript 的浏览器，也会直接忽略该段脚本，但是新一代的浏览器则知道要执行该段脚本。

这是一种不正规的用法，但却广为流传。新一代的浏览器大多支持 JavaScript 特性，并且也能够识别位于 HTML 注释中的 JavaScript 代码段。然而，现在一些新的浏览器能够以 XHTML 方式解释网页，甚至是 XML 的方式，那么就会忽略所注释的代码段，在这些情况下，JavaScript 就会被忽略，而不会被执行。所以，使用 HTML 注释来“隐藏”JavaScript 代码的方式，已经不被推荐。

然而还有另一种“隐藏”代码的方式，那就是使用 XML CDATA 小节，这是现在所推荐的方法，特别是在 XHTML 文档中使用脚本代码时。示例 1.3 在示例 1.2 的基础上做了一些的修改，改为使用 CDATA 小节，修改的部分已加粗显示。

示例 1.3　对示例 1.2 的修改版本，通过 CDATA 小节来“隐藏”脚本代码

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>Hello, World!</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript">
//<![CDATA[

function hello() {
   var msg = "Hello, <em>World!</em>";
   document.open();
   document.write(msg);
   document.close();
}

//]]>
</script>
</head>
<body onload="hello()">
<p>Hi</p>
</body>
</html>
```

之所以使用 CDATA 小节，是因为 XHTML 处理器在解释标记型语言时会识别出所有的开始标签和结束标签，即使是位于 JavaScript 代码里面，如本例中的 em 元素。虽然脚本可能能够正确地处理，页面也能够正确地显示，但是如果试图去校验不使用 CDATA 小节的页面，就可能导致校验失败，如图 1.1 所示。

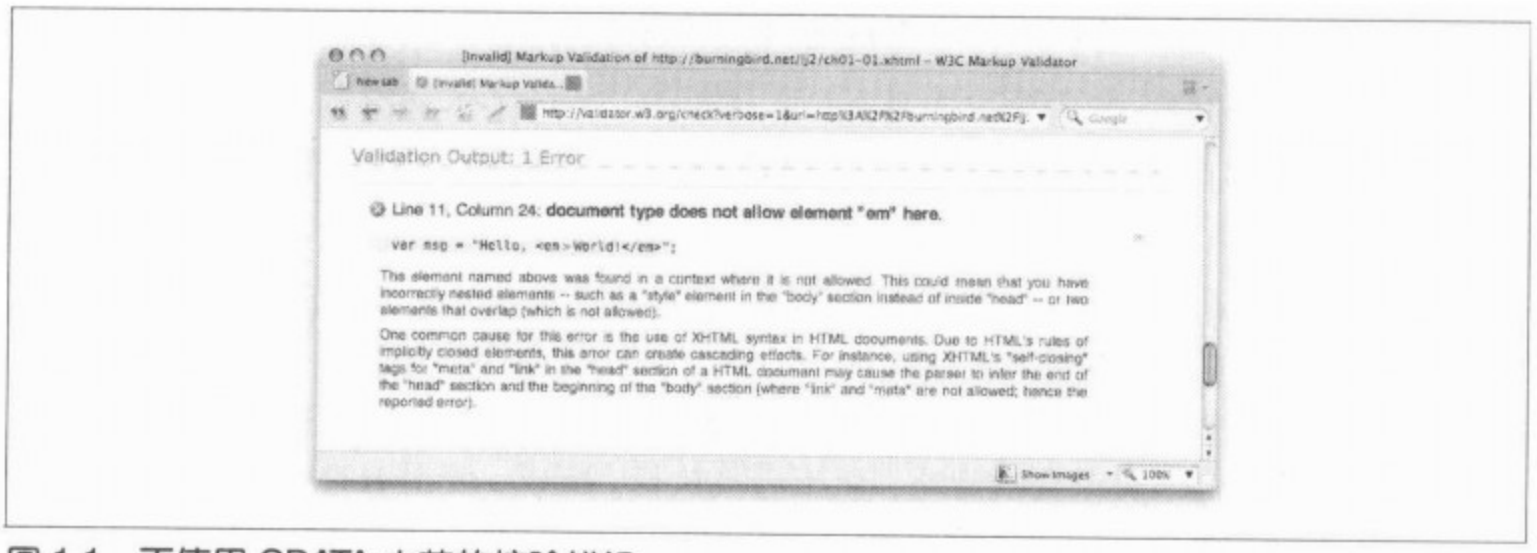


图 1.1　不使用 CDATA 小节的校验错误

在页面文件中通过 script 元素的 src 属性引用的 JavaScript，是 XHTML 标准所兼容的，并且不要求使用 CDATA 小节。如果是嵌入的 JavaScript 代码，那么就需要使用 CDATA 小节，特别是包含在 body 元素中的代码。对于大多数浏览器而言，还需要用 JavaScript 注释符（//）来隐藏 CDATA 小节，如示例 1.3 所示，否则将会出现 JavaScript 错误。

当然，保持页面整洁的最佳方式是将 JavaScript 代码从页面中彻底移去，改成使用链接 JavaScript 文件的方式。

在本书的大多数示例中，JavaScript 代码是直接嵌入在页面里的，这样可以提高代码的可读性且易于检查。然而，Mozilla 推荐将所有嵌入的 JavaScript 代码都从页面中移去，放在独立的 JavaScript 文件中。使用独立的 JavaScript 文件，可以避免校验以及文本解释错误等问题，而不用担心页面是以 HTML 还是 XHTML 的方式进行处理。

提示

使用 JavaScript 文件往往也能提高网页载入的效率，因为浏览器会在第一次载入文件的时候进行缓存，引用相同文件时则会从缓存中获取。

1.3　JavaScript 文件

JavaScript 的应用越来越基于面向对象的方式，而且更为复杂。为了简化开发工作、共享自己的 JavaScript 代码，许多 JavaScript 开发人员创建了可复用的 JavaScript 对象，以便在其他应用程序中复用。共享这些对象的唯一有效的方法，就是将这些对象放在独立的文件中，并在网页中提供每个文件的链接。所有开发人员所需做的只是将代码链接到自己的网页中。如果代码需要修改，那么只需要在一个地方进行修改。

现在，所有简单的 JavaScript 代码都将放在独立的脚本文件中。然而过度地引用 JavaScript 文件，所引发的问题也可能多于其带来的好处。在网页中引用 JavaScript 程序库或者 JavaScript 文件的语法如下所示：

```
<script type="text/javascript" src="somejavascript.js"></script>
```

script 元素中没有任何内容，但是需要以</script>作为结尾。

浏览器会依照 script 脚本在页面中出现的顺序，依次载入并处理每个脚本文件，当然标识为 defer 的脚本除外。处理脚本文件与嵌入在页面的代码的方式并不存在任何不同之处。

示例 1.4 是“Hello World!”程序的另一个版本，区别在于其将脚本从页面中移到了独立的文件中，该文件被命名为 helloworld.js。独立的 JavaScript 文件必须以 js 作为后缀，除非 Web 服务器能够将其他后缀名识别成 JavaScript MIME 类型。因为 js 后缀名已为开发人员所熟悉，所以没有必要特意去破坏这样的约束。

提示

当然，任何规则都有例外的情况，js 后缀名也是如此。如果 JavaScript 文件是由 PHP 服务端动态生成的，那么该文件会以不同的后缀名存在。

示例 1.4 是独立的 JavaScript 文件，而示例 1.5 则是引用该 JavaScript 文件的网页。

示例 1.4 “Hello World!”程序要使用的独立的 JavaScript 文件

```
/*
   函数名: hello
   作者: Shelley
   hello 函数将打印"Hello, World!"
*/

function hello() {

   // 向世界问好
   var msg = "Hello, <em>World!</em>";
   document.open();
   document.write(msg);
   document.close();
}
```

示例 1.5 引用外部 JavaScript 文件的网页

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>Hello World!</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />

<script type="text/javascript" src="helloworld.js">
</script>
</head>
<body onload="hello()">
<p>Hi</p>
```

```
</body>
</html>
```

现在的网页比之前干净整洁多了，并且应用程序的可维护性也更好。其他应用程序也可以复用这段 JavaScript 代码。虽然复用“Hello World!”这样的代码是不太可能的，但是在本书稍后介绍的示例中能够体现出复用性的重要。

在进入介绍变量和数据类型的第 2 章之前，现在还是先看看本章的最后一节。

1.4 可访问性和 JavaScript 最佳实践

如果有理想的世界，那么开发人员都希望访问自己网站的用户使用相同的操作系统、相同的浏览器，并且都启用了 JavaScript。用户不会使用移动电话或者其他奇怪的设备，视力不好的人也不需要使用屏幕朗读设备，听力不好的人也不需要语音导航设备。

然而理想总归是理想，现实往往与之不一样，许多 JavaScript 开发人员都需要考虑这些现实的情况。

许多最佳实践都是与 JavaScript 相关的，但最重要的实践有这么一条：任何 JavaScript 功能都不应该成为网站和用户之间的障碍。

什么是“网站和用户之间的障碍”呢？JavaScript 不应当阻碍那些没有启用 JavaScript 的用户正常访问网站。如果基于 JavaScript 创建了下拉框菜单，那么同时也应该为没有启用 JavaScript 的用户提供非基于 JavaScript 的替代选项。如果访问站点的用户视力不佳，那么当动态地往页面添加指令的时候，也应该考虑对语音浏览器的支持。

许多开发者并不遵循这些最佳实践，因为这需要做更多的工作。然而这并不应成为负担，因为这样能够增加站点的可访问性。此外，现在的许多公司都期望自己网站的可访问性达到一定程度。当然，应该从一开始就养成创建高可访问性页面的习惯，而不是每次去解决页面可访问性的问题。

1.4.1 可访问性指南

关于如何创建高可访问性的 JavaScript 应用，在 WebAIM 网站上能够找到一份很详尽的指南（http://www.webaim.org/techniques/javascript）。该指南介绍了在哪些情况下应该避免使用 JavaScript，如使用 JavaScript 创建菜单以及其他导航。然而，该指南也介绍了如何利用 JavaScript 来提高站点的可访问性。

本书的建议是判断这些事件是否能由鼠标触发。例如，与其只捕获鼠标点击事件，那么还不如捕获键盘或鼠标所触发的事件，如 onfocus 和 onblur 事件。如果是一个下拉框菜单，那么应该添加一个独立的页面，并提供静态的菜单。

阅读了 WebAIM 的指南之后，还可以再看看 W3C 网站中关于可访问性的介绍（http://www.

w3.org/WAI)，另外还可以浏览一下美国政府的 Section508 网站（http://www.section508.gov）。

当然，并不是所有可访问性的问题都跟浏览器禁用了 JavaScript 有关，如遇到有屏幕朗读设备的情况。许多人并不信任 JavaScript，或者无意中禁用了 JavaScript 功能。对于这些不喜欢使用 JavaScript 的用户，提供非基于 JavaScript 的替代选项是非常重要的，如 noscript。

1.4.2 noscript

当一些浏览器或应用程序并不支持 JavaScript 时，或者对 JavaScript 支持存在部分限制时，如果 JavaScript 不是用在导航或用户交互方面，那么浏览器忽略 JavaScript 不会带来任何问题。然而，如果网站需要 JavaScript 才能访问，而同时又不提供任何替代选项，那么用户就会离你而去。

很多年前，当 JavaScript 还是一个新事物的时候，一种流行的方法是提供另一份纯文本的页面，将链接放置在页面的上方。然而，这种维护两份页面的做法并不被推荐，开发人员经常会忽略同时对两份页面进行更新的工作。

另外一种更好的方法是为脚本生成的动态内容提供静态的替换选项。当使用 JavaScript 创建下拉框菜单时，同时也应该提供一个标准的层次关系的菜单；当用户交互基于脚本来修改窗体元素时，那么也应提供传统的页面来实现同样的功能。

所有这一切都要使用 noscript 元素。添加 noscript 元素还包括所需的静态内容。如果浏览器或其他应用程序不能处理脚本（因为某些原因禁用了 JavaScript），那么就会选择 noscript 的内容；如果浏览器支持 JavaScript，那么就会忽略 noscript 中的内容。

示例 1.6 是“Hello World!”程序的最后一个版本，它在之前版本的基础上添加了 noscript 元素。在启用 JavaScript 的浏览器中可以看到“Hello World!”，如果在禁用 JavaScript 功能的浏览器中，则会显示不一样的内容。

示例 1.6　使用 noscript 元素来支持禁用 JavaScript 的浏览器

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>Hello, World!</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />

<script type="text/javascript">
//<![CDATA[

function hello() {
   //向世界问好
   var msg = "Hello, <em>World!</em>";
   document.open();
```

```
    document.write(msg);
    document.close();
}

//]]>
</script>
</head>
<body onload="hello()">
<noscript>
<p>I'm still here, World!</p>
</noscript>
</body>
</html>
```

当然，示例 1.6 只是一个非常简单的示例，它介绍了 noscript 的用法；在本书稍后的章节中还会介绍更为复杂的示例，以及可选的脚本安全的方法。

要对示例 1.6 进行测试，可以使用 Firefox 中名为 Web Developer Toolbar（Web 开发工具条）的扩展。它的菜单中有个选项能够禁用 JavaScript 支持。当启用 JavaScript 时，会显示“Hello World!”信息。不过，如果禁用 JavaScript 支持，那么就会显示“I'm still here, World!”。虽然在浏览器中也可以禁用 JavaScript，不过使用类似 Web Developer Toolbar 的开发工具还是非常方便的。

浏览器决定了所能使用的工具。本书推荐基于 Firefox 进行开发，这样就可以使用 Web Developer Toolbar 以及 Firebug（一个神奇的调试工具）了。在后续的章节还将介绍如何调试和解决疑难杂症，以及如何充分利用这些工具。

第 2 章

JavaScript 数据类型和变量

JavaScript 中的变量是已命名的数据，是一种为数据创建引用的方式，无论数据是字符串、数字、布尔值、数组或者其他对象，都可以通过变量名多次访问该数据。更重要的是，变量可以用于在不同的过程中存储数据。例如，JavaScript 应用可以将窗体元素的值保存在变量中，然后就能够通过变量访问该值，而不必每次都从窗体元素中获取。

变量的数据类型是 JavaScript 脚本引擎对变量中所保存数据的类型解析。字符串变量用来保存字符串，数字变量用来保存数字。不过，JavaScript 和其他语言不同，在同一个应用程序中，相同的变量可以保存不同类型的数据。这就是俗称的松散类型（loose typing）或动态类型，这意味着 JavaScript 中的变量在不同时候可以根据不同的上下文保存不同的数据类型。

在支持松散类型的语言中，声明变量时无须指定变量的类型，因为变量的数据类型将是动态决定的。如果某个之前用来保存字符串的变量现在要作为数字来使用，只要其字符串中的内容的确能够表示为数字，而不是如电子邮件地址之类的内容，那么也是没有问题的。如果还想继续将其当成字符串来使用，也一切正常。

然而松散类型的特性也会引发一些麻烦。如果希望对两个数字执行加法操作，但是 JavaScript 引擎将保存数字的变量解释成了字符串类型，那么其结果就将是连接这两个字符串，而不是求这两个数字的和。在 JavaScript 语言中，上下文环境将决定变量的数据类型。

本章将介绍 JavaScript 的基本数据类型，包括字符串、数字和布尔值，以及这些数据类型的内建函数。此外，我们还将介绍 JavaScript 中两个特殊的数据类型 null 和 undefined（未定义）。最后还将说明字符串的转义以及 Unicode 编码。此外，本章还将深入介绍变量，以及如何让变量名称有效且有意义。

2.1 标识变量

JavaScript 变量涉及变量名、作用域以及特定的数据类型。因为 JavaScript 语言支持松散类型，所以可以对变量做任意的修改。

JavaScript 变量与其他编程语言大致相同，都是用来保存数值的，并且都能够在其他代码中调用。每个变量都有一个变量名，而且在同一范围内只能存在一个同名的变量。变量名由字母、数字、下划线和$符号组成。JavaScript 变量名对格式没有特定的要求，但要求以字母、$或下划线开头，例如：

```
_variableidentifier
_variableidentifier
variableIdentifier
$variable_identifier
var_ident
```

从 JavaScript 1.5 开始，也可以在变量名中使用 Unicode 编码中的字母（如 ü）和数字(\u0009)。下面几个也是合法的 JavaScript 变量名：

```
_üvalid
T\u0009
```

使用特殊字符时要特别注意，因为某些工具（如调试器）对特殊字符的支持可能会存在一些问题。

JavaScript 是区分大小写的，也就是说大写变量名和小写变量名是不一样的。例如，JavaScript 语言会将下面两个变量名识别为两个不同的变量：

```
stringVariable
stringvariable
```

对于变量名还有一个限制，它们不能是 JavaScript 关键字（如表 2.1 所示）。随着 JavaScript 新版本（如 ECMAScript）的推出，可能还会引入新的关键字。

表 2.1 JavaScript 关键字

break	else	new	var
case	false	null	void
	finally	return	
catch	for	switch	while
continue	function	this	with
default	if	throw	
delete	in	true	
		try	
do	instanceof	typeof	

根据 ECMA-262 标准的相关扩展，表 2.2 中列出的单词也被认为是关键字，因而也不能用于变量名。

表 2.2　ECMA-262 标准的关键字

abstract	enum	int	short
boolean	export	interface	static
byte	extends	long	super
char	final	native	synchronized
class	float	package	throws
const	goto	private	transient
debugger	implements	protected	volatile
double	import	public	

除了 ECMAScript 关键字之外，大多数浏览器所实现的与 JavaScript 相关的单词也被认为是关键字。许多关键字都是源于浏览器对象模型（BOM）的，如 document 和 window 对象。表 2.3 中列出了这种情况中最常见的单词，不过它并不完整。

表 2.3　浏览器定义的关键字

alert	eval	location	open
array	focus	math	outerHeight
blur	function	name	parent
boolean	history	navigator	parseFloat
date	image	number	regExp
document	isNaN	object	status
escape	length	onLoad	string

2.1.1　命名规范

讲完变量命名的限制之后，现在来谈谈与 JavaScript 变量相关的命名规范，这些规范有些源于 Java 或其他编程语言，有助于提高代码的可阅读性。

命名时应该使用有意义的单词，而不要随便命名。例如 interestRate 这样的变量名要比 intRt 或 ir 来得更清楚些。后两种变量名太过简化，容易造成阅读代码的困难。

当然也可以将数据类型作为变量名的一部分，例如一个用来保存姓名的字符串变量，可以命名为：

```
var strFirstName = "Shelley";
```

这种将数据类型作为变量名一部分的命名方法，也被称为匈牙利命名法，在 Windows 开发中非常流行。在过去的 Jscript 应用程序中，经常会看到这样的命名方式，然而这

种命名方式在现在的 JavaScript 开发中并不常见。

另一种命名规范是对集合型变量采用复数形式命名：

```
var customerNames = new Array();
```

一般来说，变量名的首字母不应采用大写字母，因为首字母大写通常用来表示类，如 String 类：

```
var firstName = String("Shelley");
```

对类名的首字母使用大写，有助于区分变量和类。

对于函数名和变量名，通常首字母都是小写的，而函数名将以动名词形式表示，可以让读者更容易通过函数名了解函数的功能，例如：

```
function validateNameInRegister(firstName,lastName) ...
```

在大多数情况下，变量名和函数名都是由一个或多个单词组成的，并遵循 CamelCase 命名法（每一个单词的首字母小写，后续单词的首字母大写），这也是其他编程语言所流行的命名法。下面就是一些遵循 CamelCase 命名法的变量名：

```
validateName
firstName
```

CamelCase 命名法有助于代码的读者理解变量的含义，当然横杠或下划线也能起到相同的作用，例如：

```
Validate_name
first_name
```

在最新的 JavaScript 程序库中也大量使用了 CamelCase 命名法，这也正是本书推荐这种命名法的原因。

尽管变量名的首字母可以是$、下划线、字母等，但是最好的选择还是用字母作为首字母。在变量名中使用不常见的符号，会使代码变得晦涩难懂，特别是对于那些 JavaScript 新手而言。然而，如果看过新的 JavaScript 程序库或者示例，就会发现在这些代码中也存在一些奇怪的变量名。流行的 Ajax 框架、JavaScript 程序库 Prototype 就是典型代表，所以我认为这可能会引发“Prototype 现象”的新命名规范。

下面就是一个使用这种命名规范的变量名示例：

```
var _break = someval;
```

在这些 JavaScript 程序库中，以下划线为首字母的变量表示它是对象的私有成员变量（第 3 章将详细介绍对象的成员变量）。Prototype 引人的另一个有意思的命名规范是，对于查找页面元素的函数使用$符号作为函数名称，例如：

```
$('test').invokeSomeMethod();
```

下划线或$符号的这类新用法，并不会改变变量的行为，它仅仅是一种命名方式。

提示

关于 JavaScript 程序库 Prototype 的详细介绍参见 http://www.prototypejs.org/。

在这一小节中，我们主要介绍了 JavaScript 的命名规范，这些命名规范并没有什么神奇的地方，它们仅仅是为了让 JavaScript 代码更容易阅读和调试。

2.2 基本类型

JavaScript 是一种清晰易懂的编程语言，它实现了脚本语言所需的功能，但又不显得臃肿。然而，JavaScript 在某些方面的确存在一些容易引发混淆的地方。

例如，JavaScript 中只有 3 种基本数据类型：字符串、数字以及布尔型。每种数据类型互不相同，它们分别对应于字符串值、数字值以及布尔值。然而，JavaScript 还提供了一些内建的对象，如 String、Number 和 Boolean。这些对象看起来与基本数据类型又截然不同，前 3 种是基本值的类型，而后 3 种则是对象，是拥有内建属性和方法的对象。

然而它们事实上又紧密相关。当以对象的方式操作基本类型时，String 对象会封装字符串基本类型，而 Number 和 Boolean 也同样会封装各自的基本类型。当在 JavaScript 中创建了简单的字符串变量并使用 String 对象的方法时，JavaScript 会隐式地通过 String 对象封装字符串基本类型，并且调用 String 对象的属性和方法，最后销毁该对象。在接下来的示例中，当字符串变量 firstName 调用 toUpperCase 方法的时候，JavaScript 会创建一个对象来封装这个字符串，然后调用 toUpperCase 方法，最后销毁这个临时对象：

```
var firstName = "Shelley";
var cappedName = firstName.toUpperCase();
```

firstName 变量虽然是一个基本类型，但是却表现得像一个对象，可以像对象一样调用 toUpperCase 的方法。如果调用 String 对象的其他方法，也同样会创建一个 String 对象，封装字符串基本类型，调用方法，最后销毁临时对象。所以，如果希望以对象的方式操作字符串，那么最好的方式是创建一个对象。如果只是需要用一个简单的字符串来输出信息，或者保存字符串值，而不需要对象提供的所有功能，那么字符串基本类型将是更好的选择。

所以，如果仍然糊涂地混用基本数据类型和对象，那么就违背了 JavaScript 区分基本数据类型和对象的初衷了。接下来的 3 个小节将分别介绍每种基本数据类型，说明它们是如何创建和操作的，以及如何进行数值转换。第 5 章将详细介绍数据对象，以及它们的方法和属性。

提示

本书所提及的“封装”一词，指的是简单地为数据项添加一层包装它的对象，如 String 对象封装了字符串基本类型。

2.3 String 数据类型

由于 JavaScript 是一门支持松散类型的编程语言，字符串变量或者数字型、布尔型变量在声明时并没有什么差别，只有把文本赋给 String（字符串）型变量，并且对变量进行初始化之后才定义了变量的上下文。

字符串文本是由单引号或双引号所引用的一系列字符，例如：

```
var strString = "This is a string";
var anotherString= 'But this is also a string';
```

JavaScript 并未限定必须用单引号或双引号来表示字符串，唯一的规则是前后的符号必须匹配。字符串中可以包括各式各样的字符，例如：

```
var thirdString = "This is 1 string.";
var stringFour = "This is--another string.";
var stringAsNumber = "543";
```

在最后一个字符串中引用的虽然是一个数字，但是由于该数字是放在双引号中的，所以 JavaScript 认为这是一个字符串变量。

字符串中还可以包括单引号或双引号，但这个时候封装整个字符时要使用另外一种引号，而且必须保持前后一致。如果字符串中包含单引号，那么就要使用双引号来表示字符串；如果需要包含双引号，那么就应使用单引号来表示字符串，例如：

```
var string_value = "This is a 'string' with a quote."
```

或者

```
var string_value = 'This is a "string" with a quote.'
```

空字符串是一种特殊的情况，一般用来初始化字符串变量。下面就是空字符串的示例：

```
var string_value = '';
var anotherStringValue = "";
```

JavaScript 引擎会以统一的方式来处理表示字符串的单引号和双引号。当然，在代码中，统一使用单引号或双引号，能够提高代码的可读性。

2.3.1 字符串转义符

在 JavaScript 语言中，并不是所有字符都可以直接放在字符串中的。字符串中还可以包含转义符，例如\n 表示换行符。转义符是通过特定方式将一些特殊字符放在字符串中的一种模式。

下列代码将包含换行符的字符串文本赋给某个变量。当在对话框中显示该字符串时，转义符\n 会被解释为换行符：

```
var string_value = "This is the first line\nThis is the second line";
```

这时对话框中显示的结果将是：

```
This is the first line
This is the second line
```

反斜杠同样可以用来在字符串中对引号进行转义，使用\"就表示引号是字符串的一部分，而不是字符串的结束符号，例如：

```
var string_value = "This is a \"string\" with a quote."
```

通过使用反斜杠，就可以在字符串中同时使用单引号和双引号。

如果要在字符串中使用反斜杠，那么应该写成双反斜杠（第一个反斜杠表示转义）：

```
var string_value = "This is a \\string\\ with a backslash."
```

该字符串中包含两个反斜杠，在 string 字符前后各有一个。

在字符串中还可以使用 Unicode 字符，表示方法是在\u 后面加上 4 位的十六进制数值。例如，以下示例定义的是中文简体的“爱”字：

```
document.writeln("\u7231");
```

当然 Unicode 字符的显示是由浏览器所决定的，但是，大多数流行的浏览器都支持标准的 Unicode 编码。

提示

关于 Unicode 编码的更多内容详见 http://www.unicode.org。

2.3.2 字符串编码

转义符、反斜杠是非常实用的，特别是在字符串中要使用表示控制字符的 ASCII 码时。然而，反斜杠对于非 ASCII 码字符是无能为力的，也无法确保某个字符串在 HTML 处理过程中是安全的，而这些情况却又是现在流行的 Ajax 程序经常会面临的。

encodeURI 和 encodeURIComponent 方法的用途就是对字符串进行转义，准确地说是对字符串进行编码，将其从 ASCII 码或非 ASCII 码字符转换成 URIencoding 的字符。

提示

URI 是统一资源标识符（Uniform Resource Identifier）的缩写，例如网页的 URL。ISO Latin-1（或者 ISO 8859-1）就是一种 URI 编码规范。

encodeURI 方法假设字符串是一个 URI，例如“http://oreilly.com”，并对下列字符进行编码：

```
; , / ? : @ & = + $
```

对于字母、数字以及下列字符不会进行编码：

```
- _ . ! ~ * ' ( )
```

对页面片段符号（#）也不会进行编码。

encodeURIComponent 方法会对所有的字符（除了字母和上面所列举的字符）进行编码。该方法将需要进行编码的字符串视为 URI 的一个参数，因此属于 URI 的字符也会被编码，例如下列这些字符：

```
# & + =
```

这两个函数有着相对应的解码函数：decodeURI，用来对 encodeURI 编码过的字符串进行解码；decodeURIComponent，用来对 encodeURIComponent 编码过的字符串进行解码。

示例 2.1 中的页面里使用了这 4 个函数，分别对两个字符串进行编码和解码，编码和解码后的字符串都会通过 document.writeln 输出到页面中，如图 2.1 所示。

示例 2.1　通过 JavaScript 的 encodeURI 和 encodeURIComponent 方法对两个字符串进行 URI 编码

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>URI Encoding</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript">
//<![CDATA[
function encodeStrings() {
    var sOne =
encodeURIComponent("http://burningbird.net/index.php?pagename=$1&page=$2");
    var sTwo = encodeURI("http://someapplication.com/?catsname=Zöe&URL=");
    var sOutput = "<p>Link is " + sTwo + sOne + "</p>";
    document.write(sOutput);

    var sOneDecoded = decodeURI(sTwo);
    var sTwoDecoded = decodeURIComponent(sOne);

    var sOutputDecoded = "<p>" + sOneDecoded + "</p><p>" + sTwoDecoded + "</p>";
    document.write(sOutputDecoded);

}
//]]>
</script>
</head>
<body onload="encodeStrings()">
   <p></p>
</body>
</html>
```

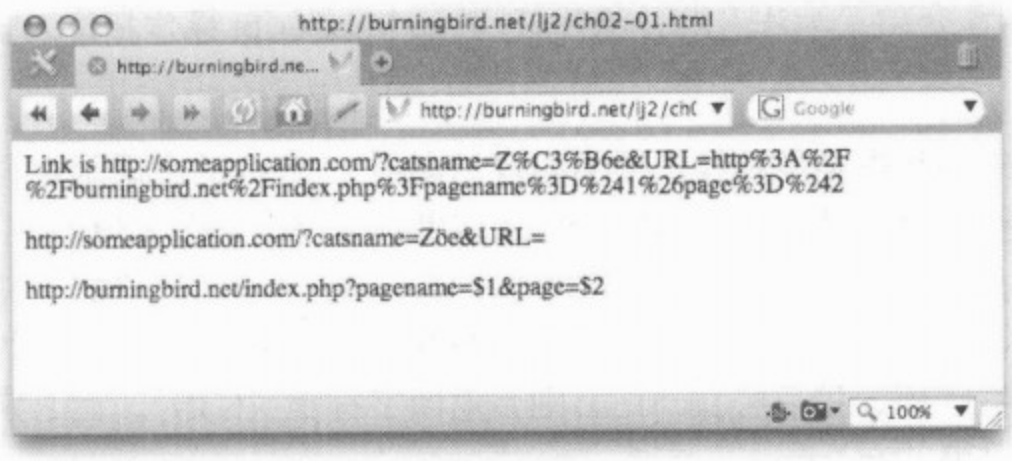


图 2.1　对 URI 编码和解码后的输出

这些示例介绍了如何显式地创建字符串变量，以及对包含特殊字符的字符串的处理。当然，也可以根据上下文环境，将其他数据类型转换成某个特定类型。

2.3.3　字符串转换

数字、布尔值等其他数据类型都可以转换成字符串；一般来说，脚本引擎将根据上下文自动完成这样的转换。例如，当把数字或布尔型变量传给希望接收字符串变量的函数时，就会先隐式地将该数值转换成字符串，再进行处理：

```
var num_value = 35.00;
alert(num_value); //预期为一个字符串
```

此外，如果在赋值语句中要对两个变量执行加法操作，其中一个是字符串变量，而另一个是数字变量，那么数字变量就会自动转换成字符串，接着再连接这两个字符串：

```
var num_value = 35.00;
var string_value = "This is a number:" + num_value;
```

什么时候将数字转换成字符串，取决于 JavaScript 脚本引擎在什么时候处理字符串。例如，如果字符串是所有数值的第一个，那么所有数值都会被当成字符串进行处理：

```
var strValue = "4" + 3 + 1;     // 结果是 "431"
var strValueTwo = 4 + 3 + "1";  // 结果是 71
```

然而，如果使用其他操作符号，那么会将字符串转换为数字：

```
var firstResult = "35" - 3;     // 减法操作，结果是 32
var secondResult = 30 / "3";    // 除法操作，结果是 10
var thirdResult = "3" * 3;      // 乘法操作，结果是 9
```

隐式转换取决于操作符和变量的位置，这更加充分地体现了松散类型的危险：数值会随着上下文发生变化，而这取决于引入新数据类型操作的顺序，以及所引用的操作符。

提示

本章介绍了加法和其他操作符，其他 JavaScript 操作符请阅读本书第 3 章。

与其依赖于隐式的数据类型转换，还不如自己调用 String 的全局函数显式地执行字符串转换。如果要转换的是一个布尔值，那么结果字符串则是布尔值的文本表示，“true”表示布尔真值，而“false”则表示布尔假值。对于数字而言，转成的字符串就是表示该数字的字符串，例如“-123.06”表示数字-123.06，当然这取决于数字的位数和精度。NaN 数值（非数字值，稍后将会介绍）则会返回字符串“NaN”，而未定义或空值的变量则会返回字符串“undefined”或者“null”。

表 2.4 是在不同数据类型上调用 String 函数的执行结果。

表 2.4　String 转换表

输　入	结　果
undefined	"undefined"
null	"null"
布尔值	"true"或"false"
数字	数字相应的字符串，NaN 表示非数字值
字符串	不转换
对象	对象默认的字符串

在表 2.4 中，最后一行描述了 ECMAScript 中针对对象调用 String 函数的执行规则，其生成的默认字符串形如：

```
"[object "+className+"]"
```

示例 2.2 显式地对不同变量和对象进行转换，创建数字和布尔型变量，并初始化相应数据类型的数值，接着调用 String 的函数显式地将其转换成字符串。该示例还为一个已创建但还没有初始化的变量，以及一个初始化为 null 的变量进行转换。最后，对 document 对象执行转换，并将结果输出到页面中。

示例 2.2　隐式和显式的字符串转换

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>Implicit and Explicit String Conversion</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript">
//<![CDATA[
```

```
function convertToString() {
   var newNumber = 34.56;
   var newBoolean = true;
   var nothing;
   var newNull = null;

   var strNumber = String(newNumber); var strBoolean = String(newBoolean);
   var strUndefined = String(nothing); var strNull = String(newNull);

   var strOutput = "<p>" + strNumber + " " + strBoolean + " " + strUndefined
+ " " + strNull + "</p>";
   document.writeln(strOutput);

   var strOutput2 = String(document);
   document.writeln(strOutput2);

}
//]]>
</script>
</head>
<body onload="convertToString()">
   <p></p>
</body>
</html>
```

在不同的浏览器（Firefox、Opera、IE 及 Safari）中，该示例输出的第一个字符串都是一样的：

```
34.56 true undefined null
```

然而，只有 Opera 和 Firefox 浏览器将输出 document 对象的 ECMAScript 特定表示法：

```
[object HTMLDocument]
```

IE 浏览器则仅仅显示[object]，而 Safari 和 WebKit 浏览器则根本不支持将 document 对象转换成字符串。

2.4 Boolean 数据类型

Boolean（布尔）数据类型只有两种可能值：true 和 false。布尔值不需要使用引号，所以"false"与 false 所表示的含义是完全不一样的。

```
var isMarried = true;
var hasChildren = false;
```

布尔值可以显式地赋给不同类型的变量，这取决于该变量是否已经赋值，以及该变量的当前值。例如，下列的条件表达式将把 testVariable 转换为布尔型变量，如果 testVariable 变量值为 true 则执行 if 语句内的程序，如果 testVariable 变量值为 false 则跳过这段程序：

```
if (testVariable) {
   ...
}
```

关于如何根据布尔值决定应用程序的控制流程，我们将在以后的章节中介绍。在此我们将不同类型的变量显式或隐式转换为布尔值的规则列在表 2.5 中。在这里所说的“显式”是指使用 Boolean 函数将其他数据类型（如 String）的值转换为布尔值：

```
var someValue = 0;
var someBool = Boolean(someValue)
```

表 2.5　toBoolean 转换表

输　入	结　果
Undefined	False
Null	False
布尔值	布尔值
数字	如果数字等于 0 或 NaN，那么结果是 false，否则为 true
字符串	如果是空字符串，那么结果为 false，否则为 true
对象	true

双重否定符（两个否定操作符“!”）可以用来显式地将数字或字符串转换为布尔值：

```
var strValue = "1";
var numValue = 0;
var boolValue = !!strValue;        // 字符串"1" 转换为 true
boolValue = !!numValue;            //  数字 0 转换为 false
```

2.5　Number 数据类型

在 JavaScript 中，Number（数字）数据类型是浮点数，可以包含小数部分，也可以没有小数部分。如果没有小数部分，那么该数字将被当做十进制整数，取值范围为-2^{53}～2^{53}。

以下变量是合法的整数：

```
var negativeNumber = -1000;
var zero = 0;
var fourDigits = 2534;
```

浮点型可以包含小数部分。数字也可以用指数形式（科学计数法）表示。下面列出的都是合法的浮点型数字：

```
var someFloat = 0.3555
var anotherNumber = 144.006;
```

```
var negDecimal = -2.3;
var lastNum = 19.5e-2   // 也就是 0.195
var zeroDecimal = 12.0;
```

尽管 JavaScript 能够支持较大的数字，但是某些函数只能够支持-2e31（-2,147,483,648）到 2e31（2,147,483,648）之间的数字；所以，请尽量将数字限制在这个范围内。

JavaScript 的 Number 数据类型有两个特殊的数字：正无穷大和负无穷大，分别用 Infinity 和-Infinity 表示。当 JavaScript 数字溢出的时候，就会返回正无穷大。而当所使用的数字比 JavaScript 所能够支持的最小数字还小时就将用到负无穷大。

除了十进制外，还可以使用八进制或十六进制来表示数字，不过对于一些老浏览器而言，八进制可能会与十六进制相互混淆。十六进制数字是以 0x 开头的，例如：

```
var firstHex = -0xCCFF;
```

八进制也是以 0 开头，但是没有 x，例如：

```
var firstOct = 0526;
```

我们之所以对八进制和十进制表达方式都感兴趣，是因为如果使用 String 函数将八进制或十进制数字转换为字符串，那么脚本引擎将首先会把数字转换为十进制数，然后再将结果转换为字符串。所以，firstOct 变量转换为字符串的结果是“342”，342 正是八进制数字 0526 转成十进制的结果。

在前面的两个小节中，我们介绍了如何将其他数据类型转换为字符串或者布尔型。本小节将继续介绍两个不同的函数，分别用于将字符串转换为数字：parseInt 和 parseFloat。其中 parseInt 函数只返回数字的整数部分，无论字符串是一个整数还是浮点数。而 parseFloat 函数则会返回浮点型数值。

示例 2.3 中有 3 个包含数字的字符串，分别调用 parseInt 和 parseFloat 函数对其进行转换，并将结果输出到页面上。

示例 2.3　调用 parseInt 和 parseFloat 将字符串转换为数字

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>Convert to Number</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript">
//<![CDATA[

function convertToNumber() {
   var sNum = "1.23e-2";
   document.writeln("<p>" + parseFloat(sNum) + "</p>");
   document.writeln("<p>" + parseInt(sNum) + "</p>");
```

```
    var fValue = parseFloat("1.45 inch");
    document.writeln("<p>" + fValue + "</p>");

    var iValue = parseInt("-33.50");
    document.writeln("<p>" + iValue + "</p>");

}
//]]>
</script>
</head>
<body onload="convertToNumber()">
 <p></p>
</body>
</html>
```

上述代码在 Firefox、Safari/WebKit、Opera 以及 IE8 中的运行结果如下所示：

```
0.0123
1
1.45
-33
```

请注意，在输出的结果中第一个数值是以十进制形式显示的，而不是原字符串中所采用的科学计数法。第二个数值是调用 parseInt 函数转换得到的数字，因此在应用指数方式之前先截去了整数后面的部分。同样，在第四个转换中 parseInt 截去了数字中的小数部分。

对第三个数字的转换过程是非常有趣的。parseFloat 函数会先从字符串中获取数字部分，直到遇到第一个非数字的部分为止。在该示例中，也就是“1.45”与“inch”之间的空格。接着将得到的字符串“1.45”转换成浮点数。

提示

示例 2.3 中的数字还需要转换成字符串，才能输出到页面中。

通过 parseInt 函数可以对数字在十进制、八进制或十六进制之间进行任意转换。该函数的第二个参数是数字的进制，默认值为 10。如果该参数值是 2～36 之间的数字，那么在生成字符串之前会先做进制转换。

```
var iValue = parseInt("266",16);
document.writeln("<p>" + iValue + "</p>");

var iValue = parseInt("55",8);
document.writeln("<p>" + iValue + "</p>");
```

上述 JavaScript 代码的执行结果如下所示：

```
550
45
```

除了 parseInt 和 parseFloat 函数之外，Number 函数也可以用来将其他数据类型转换为数字。转换后的返回类型取决于数字的具体内容：包含浮点数的字符串将返回浮点数，而包含整数的字符串将返回整数。表 2.6 是数字与其他数据类型之间的转换表。

表 2.6　其他数据类型与数字之间的转换表

输　入	结　果
undefined	NaN
null	0（在 IE 中将返回 NaN）
布尔值	如果是 true，那么返回 1，否则返回 0（在 IE 中将返回 NaN）
数字	数字值
字符串	整数或浮点数（取决于字符串内容）
对象	NaN

除了将字符串转换为数字外，还可以通过 IsFinite 函数来判断变量的数值是否为无穷大。如果数值是无穷大或者 NaN，那么该函数将返回 false，否则将返回 true。

第 4 章中还将介绍 Number 对象的其他函数，这些函数的作用也是对数字进行操作。在接下来的小节中将介绍 JavaScript 中两种特殊的基本类型：null 和 undefined。

2.6　null 和 undefined 变量

在 JavaScript 代码中，除了文本、简单数据类型、对象以外，还应该了解两个变量，它们分别是表示不存在的 null 变量和未定义的 undefined 变量。

null 变量是已定义的、值为 null 的变量。以下是一个 null 变量的示例：

```
var nullString = null;
```

如果变量已经声明但是还没有初始化，那么就是 undefined 变量：

```
var undefString;
```

如果声明了变量并且赋予了初始值，那么该变量就不是 null 或 undefined：

```
var sValue = "";
```

当你使用 JavaScript 程序库时，或者在一些复杂的代码中，某些变量有可能还没有初始化；如果尝试在表达式中使用这样的变量，那么就有可能得到出乎意料的结果，通常会导致 JavaScript 错误。如果不确定变量的状态，那么可以在条件表达式中测试该变量，例如：

```
if (sValue) ...    // 如果变量是 null 或 undefined，那么结果为 true，否则是 false
```

在第 3 章中将详细介绍条件语句，现在你只需要知道该表达式会判断变量 sValue 是否已经声明并初始化，如果已声明并初始化则该表达式的值为 true，否则，该表达式的值为 false。

```
if (unknownVariable)      // false，变量没有声明或赋值
if (undefinedString)      // false，变量没有赋值
if (nullString)           // false，变量已经声明并且赋值，但是所赋的值是 null
if (sValue)               // true，变量已经声明并且赋值（包括空字符串）
```

使用 null 关键字，可以判断数值是否为 null：

```
if (sValue == null)
```

在 JavaScript 中，即使变量已经声明，但只要还没有初始化就是 undefined 变量。如果变量已经声明并初始化，那么变量就不是 null 或 undefined。然而，在该示例中它是一个全局变量，正如之前的章节所述，没有以 var 关键字声明的变量可能会引起各种各样的问题。

提示

在本书中的某些代码片段中，以//开始的注释可能会由于页面宽度的原因移到下一行，而不是因为它本身就是多行注释。

尽管不一定存在，不过在此还是要介绍另一个与变量类型相关的数值：NaN。如果一个字符串或布尔值变量不能转换为数字，那么所返回的数值就将是 NaN：

```
var nValue = 1.0;
if (nValue == 'one' )     // false，第二个操作数是 NaN
```

isNaN 函数可以用来检查变量值是否为 NaN：

```
if (isNaN(sValue))        //如果字符串不能隐式转换为数字，那么就返回 true
```

null 值也是 NaN。

提示

在 O'Reilly 2006 ETech 会议上，著名的技术专家和撰稿人 Simon Willison 发表了题为“A (Re)-Introduction to JavaScript”的演讲。在他的网站上可以找到这份演讲稿（http://simon.incutio.com/slides/2006/etech/javascript/js-tutorial.001.html）。该演讲稿非常值得一读，以其中一句为例：“换言之，0、null、NaN 以及空字符串都是 false，而其他则都是 true。”

在大多数情况下，JavaScript 开发人员在编写代码的时候都会在声明变量的同时就进行初始化。所以，这些时候就无须显式地判断变量是否已经初始化。

不过，当使用复杂的大型 JavaScript 程序库或者具体的 Web 服务应答的程序时，判断变量是否初始化就将是非常重要的事，因为变量的声明和初始化并不在开发人员自己的控制范围之内。在访问应用程序的时候，了解 null 和 undefined 变量的行为也是非常重要的。

2.7 常量：已命名数值，但不是变量

在有些情况下，我们希望为某个数值定义一个名字，然后通过这个名字以只读方式访问它。这个时候，可以使用 const 关键字来创建 JavaScript 常量：

```
const CURRENT_MONTH = 3.5;
```

常量可以是任意值，因为常量不可以重新赋值，所以在声明常量的时候就必须将常量初始化为一个固定的值。

和变量一样，JavaScript 常量也可以是全局常量或局部常量。常量一般声明为全局常量，因为通常希望能够在所有 JavaScript 程序块中访问该常量。同样，请注意常量名称是全部大写的，这并不是强制要求的，但这是一种标准的命名规范，从而在代码中能够更加容易地识别出常量。

2.8 知识测验

1. 在下列变量名中，哪些是合法的变量名？哪些是非法的变量名？为什么？

```
$someVariable
_someVariable
1Variable
some_variable
sömëvariable
function
some*variable
```

2. 将下列标识符转成 CamelCase 命名法：

```
var some_month;
function theMonth          // 返回当前月份的函数
current-month              // 常量
var summer_month;          // 夏季月份的数组
MyLibrary-afunction        // JavaScript 程序库中的某个函数
```

3. 下列字符串是否合法？如果不合法，如何修改？

```
var someString = 'Who once said, "Only two things are infinite, the universe
and human stupidity, and I'm not sure about the former."'
```

4. 对于给定的数字 432.54，用什么 JavaScript 函数可以返回该数字的整数部分？如何转换成八进制和十进制？

5. 在 JavaScript 程序库中创建一个新的 JavaScript 函数，该函数包含一个名为 someMonth 的参数，如何判断该参数是否为 null 或者 undefined 变量？

2.9 测验答案

1. 下列变量名是合法的：

```
$someVariable
_someVariable
some_variable
somëvariable
```

下列变量名是非法的：

```
1Variable，因为首字母是非法的字母
function，因为它是关键字
some*variable，因为包含了非法字符*
```

2. 转换后的标识符如下所示：

```
var someMonth
function getCurrentMonth
CURRENT_MONTH
summerMonths
myLibraryFunction
```

3. 该字符串是非法的。修改方法是在双引号中只使用单引号，或者转义双引号：

```
var someString = "Who once said, 'Only two things are infinite, the
universe and human stupidity, and I'm not sure about the former'";
```

或者

```
var someString = 'Who once said, "Only two things are infinite, the
universe and human stupidity, and I\'m not sure about the former"';
```

4. 可以采用以下代码：

```
var fltNumber = 432.54;
var intNumber = parseInt(fltNumber);
var octNumber = intNumber.toString(8);
var hexNumber = intNumber.toString(16);
```

5. 如果将未声明或未定义的变量传给函数或对象方法，必将导致 JavaScript 错误，所以函数必须对该参数进行判断。

下列代码可以判断变量是否声明并已初始化：

```
if (a) {
   ...
}
```

第3章

操作符和语句

到目前为止，我们所展示的示例都非常简单，无非是变量定义、为变量赋值、在页面或对话框中输出变量值、执行加法或乘法操作、修改变量值等。所有这些示例都使用了 JavaScript 语句和操作符。

JavaScript 的语句包括以下几种类型：赋值语句、函数调用、条件分支语句和循环语句等。每种类型的语句都非常直观、简单易懂。然而，JavaScript 与其他编程语言一样，虽然这些语句学起来不难，但要将这些语句组合使用则需要一定的技巧。

3.1 JavaScript 语句的格式

JavaScript 语句通常是以分号作为结束符的，但是正如之前所说的那样，分号并不是必需的。如果处理 JavaScript 的应用程序判断语句是完整的，并且每行都以换行符结尾，那么就可以忽略分号：

```
var bValue = true
var sValue = "this is also true"
```

如果在同一行中包含多个语句，那么就必须使用分号来区分不同的语句：

```
var bValue = true; var sValue = "this is also true"
```

然而，隐式地结束 JavaScript 语句是一种坏的习惯，它可能会导致意料之外的结果。例如，当你使用工具去除代码中的空格，从而压缩 JavaScript 代码规模时，如果你的程序不是以分号显式地结束每条 JavaScript 语句，那么就可能会引发错误。

虽然 JavaScript 代码中的空格会影响代码的可读性以及 JavaScript 文件的大小，但是对代码的处理是没有丝毫影响的。例如，JavaScript 对下列两行代码的解释是完全一样的：

```
var firstName = 'Shelley'    ;
```

```
var firstName = 'Shelley';
```

与引号中用于分隔单词的空格、用于结束语句的空格不同，语句中多余的空格（制表符、空格或换行符）都将被忽略。在下列代码中，变量赋值语句是有效的，尽管该语句被两个换行符分开了：

```
var
firstName
= 'Shelley';
```

在该示例中，换行符不会被当做语句的结束符，根据 JavaScript 中赋值语句的格式，要求最左边是变量名，接着是赋值操作符（=），最后是数值。JavaScript 解释程序会一直处理 JavaScript 代码，直到遇到分号或者语句结束为止。

下面的代码也是有效的，因为当新的变量没有赋任何值时，默认的值是“undefined”。接着下一个赋值语句会通知 JavaScript 解释程序结束上一个语句的处理。

```
var firstName
var lastName;
```

下面的代码则会导致错误，因为 JavaScript 解释程序将识别到新语句的 var 关键字，然而这个时候前一个语句是无效的。

```
var firstName =
var lastName = 'Powers';
```

回到空格的话题，本书中示例代码都使用了代码缩进，从而提高代码的可读性，但没有任何函数要求使用制表符或空格。操作符的空格也是一样的，如赋值语句（=）或算术操作符（+）。空格不是必需的。空格、注释以及有意义的标识符，都能够提高代码的可读性，从而使代码更容易维护。

缩减 JavaScript 代码

空格能够提高代码的可读性，但同时也会增加文件的大小。通常情况下，这没什么影响，因为大多数 JavaScript 文件都非常小。然而，对于某些大型 Ajax 应用，或者复杂的 JavaScript 程序库来说，太大的 JavaScript 文件会影响载入 JavaScript 的速度。

如果要尽可能压缩 JavaScript 文件，那么可以选用一些免费的工具，例如 Dean Edward 的 Packer（http://dean.edwards.name/packer/），它能够在线压缩 JavaScript 代码。或者使用 Wikipedia 中“minify”条目（http://en.wikipedia.org/wiki/Minify）中所列举的工具，这些工具可以在桌面或者服务器上使用。

另一类工具是用来保护 JavaScript 代码版权的。这些工具不仅仅会压缩 JavaScript 代码，甚至还会对代码进行加密，从而使代码难以阅读。

3.2 赋值语句

JavaScript 中最常见的语句是赋值语句。赋值语句是由左边的变量、紧跟着的赋值操作符（=）以及右边的数值组成的。

右边的表达式可以是数值，或者是变量和数值所组成的表达式，例如：

```
nValue = 35.00;
nValue = nValue + 35.00;
```

语句右边的表达式也可以是函数调用：

```
nValue = someFunction();
```

多个赋值语句可以放在同一行，当然每个赋值语句之间要用分号隔开，例如：

```
var firstName = 'Shelley'; var lastName = 'Powers';
```

你还可以一次性将相同的数值赋给多个变量：

```
var firstName = lastName = middleName = "";
```

然而如果用逗号将变量隔开，如下面的代码所示，那么第一个变量将会被赋值，而第二个变量将是 undefined：

```
var nValue1,nValue2 = 3;    // nValue2 是 undefined
```

也可以通过逗号隔开多个变量的赋值，例如：

```
var nValue1=3, nValue2=4, nValue3=5;
```

为了提高代码可读性，并且避免可能的错误，最好的选择是通过分号隔开每个赋值语句。

3.2.1 算术操作符

数值计算常常需要使用算术操作符，它也可以用于赋值，或者作为函数或方法的参数。例如下面的代码将把两个变量的值相加，并将结果赋给第三个变量：

```
var theResult = varValue1 + varValue2;
```

该操作是二元算术表达式的示例，算术操作符的两边分别是一个操作数，并产生一个新结果。下面我们看看更为复杂的示例，例如多个数值的算术操作符，以及各种各样的操作数（数值或变量）：

```
nValue = nValue + 30.00 /  2 - nValue2 * 3;
```

操作数是算术运算符在执行时所需要的数值。

表达式中用到的操作符如下所示，这些操作符经常在计算器等地方可以看到：

- + 加法运算

- - 减法运算
- * 乘法运算
- / 除法运算
- % 取余运算

这些都是二元运算符，也就是需要两个操作数，在操作符旁边一边放一个。在一个语句中可以使用任意个二元运算符，并且将计算的结果赋值给变量，或者将其作为参数传给函数：

```
bigCalc = varA * 6.0 + 3.45 - varB / .05;
```

到目前为止的所有示例都是基于数字的二元运算符。那么如果是字符串呢?

在第 2 章中曾经介绍过如何通过加法符号将字符串连接起来，其操作符和执行数字相加的操作符一样：

```
var newString = "This is an old " + oldString;
```

当对数字进行相加时，加法符号执行的是加法操作，然而对字符串进行相加时，执行的将是连接操作。所有的二元操作符，都可以对字符串进行操作，但是字符串必须可以正确转换为数字，正如之前所说的那样，表达式会在执行前先将字符串转换为数字：

```
var newValue = 3.5 * 2.0;       // 结果是 7
var newValue = 3.5 * "2.0";     // 结果是 7
var newValue = "3.5" * "2.0";   // 结果还是 7
```

另一方面，如果将数字或变量与字符串相加，那么数字将会转换成字符串再执行连接操作。在下面的例子中，本来希望得到 5.5 的结果，然而实际的结果却是字符串“3.52.0”。

```
var newValue = 3.5 + "2.0";  // 结果是字符串 "3.52.0"
```

请特别注意可能存在隐式类型转换的混合操作，否则不小心的疏忽就会产生意外的结果。

警告

对于掺杂数字和字符串变量的操作都要格外小心，否则执行结果可能会出乎意料。下一章将介绍如何更为安全地转换数据。

3.2.2 一元操作符

除了前面介绍的二元算术操作符外，还有 3 种一元操作符。一元操作符与二元操作符的区别在于，一元操作符只能有一个操作数：

- ++ 自增
- -- 自减
- - 负数

下面是一些使用一元操作符的示例：

```
someValue = 34;
var iValue = -someValue;
iValue++;
document.writeln(iValue);      // 结果为-33
```

在第二行代码中，通过“负数”这个一元操作符将数字转成负数值。紧接着通过两个加号++使该数值自增 1，也就是将该变量的数值加 1，再赋值给同一个变量，它等价于：

```
iValue = iValue + 1;
```

该操作符称为自增操作符。自减操作符的用法类似，不同之处是它将数值减去 1。

自增和自减操作符还有另外一种有趣的用法。在表达式中，如果操作符出现在变量前面，那么将先计算（自增和自减）后赋值。但是，如果操作符出现在变量后面，那么将先赋值再计算：

```
var iValue = 3.0;
var iValue2 = ++iValue;     //iValue 的值为 4.0，而 iValue2 的值也是 4.0
var iValue3 = iValue++;     //iValue3 的值为 4.0，而 iValue 的值为 5.0
```

3.2.3 操作符的优先级

JavaScript 中的操作符同样也有优先级的关系，也就是说，JavaScript 中某些操作符的执行优先级要比其他操作符来得高。在 JavaScript 语句中，当所有操作符优先级一样的时候，那么表达式的执行顺序将是从左至右依次执行。如果语句中的某些操作符优先级更高时，那么执行的规则是，从左到右执行优先级高的，然后再执行低优先级的。

在算术操作符中，除法、乘法及取余等操作符的优先级比加法和减法来得高，以下面的表达式为例：

```
newValue = nValue + 30.00 /  2 - nValue2 * 3;
```

如果 nValue 的值为 3，而 nValue2 的值为 6，那么上述表达式的结果为 0。将表达式分解来看，它首先执行的是 30 除以 2，因为除法的优先级要比加法高。由于乘法的优先级与除法一样，但是乘法表达式在除法表达式的右边，所以接着会执行乘法运算。

这个乘法运算将把 nValue2 变量值乘以 3，得到的结果是 18。现在表达式只剩下加法和减法操作，而这两个操作符的优先级一样，所以从左至右执行：

```
newValue = nValue + 15 - 18;
```

赋值操作符的优先级是最低的，所以只有当算术运算符全部执行完成之后，才会将结果赋值给 newValue。

如果要控制优先级，那么可以使用括号，放在括号里的表达式将会优先执行。回到之前的示例中，使用括号可以得到不同的结果：

```
newValue = ((nValue + 30.00) / (2 - nValue2)) * 3;
```

在现在的表达式中，加法和减法操作将会优先执行，特别是优先于乘法和除法操作。表达式的执行结果将是-24.75。

到目前为止我们看到的示例都是基于算术操作符的，然而该规则同样适用于 JavaScript 的其他操作符。

提示

JavaScript 与其他编程语言有一些不同，除法得到的结果是浮点型数字，而不像其他编程语言那样得到取整后的整数。例如以下表达式的计算结果是 1.5，而不是 1：

```
iValue = 3 / 2;
```

前面这些表达式还有一些更简洁的写法，在接下来的小节中就将介绍它们。

3.2.4 带操作符的赋值符

如果赋值操作符的左右两边的变量是相同的，那么可以组合使用赋值操作符与算术操作符，例如：

```
nValue = nValue + 3.0;
```

可以使用更简洁的表示方法：

```
nValue += 3.0;
```

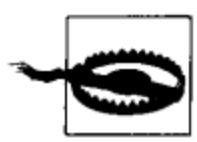
二元算术操作符都可以通过这种形式组合使用，这就是带操作符的赋值符：

```
nValue %= 3;
nValue -= 3;
nValue *= 4;
nvalue += 5;
nvalue /= 2;
```

带操作符的赋值符也可以用于位操作运算，在下一个小节中将介绍位操作运算符。

3.2.5 位操作

本小节将介绍 JavaScript 位操作，你需要对二进制运算有所了解。该功能在 JavaScript 中并不常见，如果是第一次接触 JavaScript 可以忽略本小节。



警告

如果你不熟悉二进制运算，又想阅读本小节，那么向你推荐一个优秀的关于二进制运算的资源，它们被收集在 BBC（英国广播公司）网站上（http://www.bbc.co.uk/dba/h2g2/A412642）。

位操作符将把操作数当做 32 位的二进制数（由 0 和 1 组成），然后按位执行运算，下面列举了几种类型的位操作：

- & 与操作，只有当两个操作数都为 1 的时候，结果才是 1；
- | 或操作，只要任意一个操作数为 1，结果就是 1；
- ^ 异或操作，只有当操作数不同的时候结果才是 1。如果两个操作数都是 1 或都是 0，那么结果为 0，否则结果为 1；
- ～ 非操作，按位返回相反的结果，例如 1 返回 0，0 返回 1。

在 JavaScript 语言中，位操作并不常用，然而这的确是一种比较方便的创建二进制标识的方法。二进制标识与变量类似，然而它占用的内存比较少。Mozilla Core JavaScript 1.5 手册（http://developer.mozilla.org/en/docs/Core_JavaScript_1.5_Reference/Operators/Bitwise_Operators）中的一个示例介绍了如何使用二进制标识。该示例中的变量表示了 4 个标识：

```
var flags = 0x5;
```

该变量等同于二进制值 0101：

```
flag A: false
flag B: true
flag C: false
flag D: true
```

每一位的标识如下：

```
var flag_A = 0x1;    // 0001
var flag_B = 0x2;    // 0010
var flag_C = 0x4;    // 0100
var flag_D = 0x8;    // 1000
```

如果要了解 flags 变量是否设置了 flag_C，那么可以使用与操作：

```
if (flags & flag_C) {      // 0101 & 0100 => 0100 => true
   do stuff
}
```

二进制标识可以减少内存的使用，然而这同时也会降低代码的可读性，而事实上并没有得到任何性能的提升。除非内存的使用是瓶颈，否则请不要考虑使用二进制标识。

Mozilla 手册中提供了与位操作运算相关的更多信息，位运算是一项有趣的技术，尽管 JavaScript 不需要开发人员关心内存的管理，然而开发人员还是可以在这之间找到一个平衡点。

位运算还包括 3 个位操作：左移位 (<<)、带符号的右移位 (>>)、补零的右移位 (>>>)。这些操作符根据第二个操作数，将第一个操作数左移或右移相应的位数：

```
newValue = oldValue >>> 3;
```

到目前为止介绍的都是简单的赋值语句，在接下来的小节中将介绍更为复杂的控制语句。

3.3 条件分支语句和程序流

在一般情况下，JavaScript 程序流是线性的：每个语句依次执行，每条语句紧接着上一条语句。如果希望改变程序流，那么可以将代码封装在函数中，并在特定的事件中调用该函数。另外也可以使用条件分支语句，当条件为真的时候才执行某一段代码。

在 JavaScript 中，改变程序流最通用的方法之一就是使用条件分支语句。条件分支语句的语法如下所示：

```
if (条件为真) {
      相应的语句
}
```

之所以称之为“条件分支”语句，是因为只有指定条件为真时才会执行相关联的代码块。例如，仅当某个值为真时执行下列代码，否则跳过该代码块并继续执行下一行代码。

if 关键字用来表示条件判断的开始，括号中的表达式就是条件表达式。在以下的示例中，我们将判断 tstValue 变量值是否等于 3，如果相等，就执行条件表达式后大括号中的代码。

```
if (tstValue === 3) {
     alert("value is 3");
}
```

对于上述示例而言，不一定要使用大括号，因为只有一行 JavaScript 语句需要执行。如果有多于一行的 JavaScript 语句，那么必须使用大括号引用这些代码。这也是俗称的 JavaScript 代码块，通过大括号，解析 JavaScript 的应用程序会在条件为真的情况下执行代码块中包含的所有 JavaScript 代码。

因为未来可能添加满足条件表达式时所需要执行的代码，所以即使在只有一行代码的情况下也使用大括号是一种更好的编码习惯，该编码习惯适用于所有流程控制语句。

如果要提高 JavaScript 代码的可读性，那么可以缩进大括号中所引用的代码块。如果代码块中还有其他条件分支语句，那么也同样要对代码进行缩进，从而可以清晰地阅读条件分支语句。

在示例 3.1 中介绍了 3 个嵌套的条件分支语句，每个条件分支语句中都包含一段代码，并进行代码缩进。每个条件分支语句都将判断不同的变量，你可以自行修改变量初始值，从而测试不同的条件表达式。

示例 3.1　3 个嵌套的条件语句（采用代码缩进，提高代码可阅读性）

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
```

```
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>Nested Conditional Statements</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript">
//<![CDATA[

function choices() {
   var prefChoice = 1;
   var stateChoice = 'OR';
   var genderChoice = 'F';

   if (prefChoice === 1) {
        alert("You've picked option 1. Here is what will happen...");

      if (stateChoice === 'OR') {
        alert ("You've picked 1 and you're from Oregon.");

        if (genderChoice === 'M') {
            alert("You've picked 1 and you're from Oregon and you're a man.");

        } // 最里面的代码块

      } // 第二层代码块

   } // 最外层代码块
}
//]]>
</script>
</head>
<body onload="choices()">
<p>Imagine a form with five fields and a button here...</p>
</body>
</html>
```

通常情况下，代码缩进 4 个空格，并且大括号应与条件分支语句对齐。当然也可以使用制表符来减少代码总大小（由于存在类似 minify 等压缩 JavaScript 文件的工具，所以采用制表符或空格显得并不重要）。代码缩进不存在绝对的规则，然而一定要遵守的是采用一致的缩进风格。

提示

在代码块结尾处的括号可以添加注释。如果代码太长、太复杂，并包含太多嵌套语句（如示例 3.1），那么可以在代码块结尾处的括号后添加注释，从而提高代码的可阅读性和可维护性。

示例 3.1 中也使用了条件操作符（在此例中使用的是相等判断）来判断变量值。在后续的内容中将陆续介绍其他条件操作符。不过我们还是首先了解一下各种条件语句。

3.3.1 if...else 条件分支语句

在许多应用程序中，首先根据条件判断的结果执行一段代码块，接着程序会继续执行余下的代码。然而，并不是所有程序逻辑都只有一次条件判断。即使在自然语言的文法中，也有着“如果……那么……否则”（if…then…else）的语法，以提供不同的选项，例如：如果出太阳了，那么去公园，否则就去看电影。

在 JavaScript 语言中，关键字 else 起着相同的作用，可以在 if 条件判断为假的时候执行另外的语句，例如：

```
if (expression) {
    …
} else {
    …
}
```

在下面的代码中，如果 stateCode 变量值为 MA，那么税收比例的值为 3.5，否则是 4.5：

```
if (stateCode === 'MA') {
     taxPercentage = 3.5;
} else {
     taxPercentage = 4.5;
}
```

stateCode 变量不论是不是 MA，都会设置税收比例（taxPercentage 变量）的值。

然而，条件判断还不仅如此。一些情况下，需要判断多个可能的条件，所以要进行一系列的判断，如 if then…else if then…else if then…这样的语法。JavaScript 中的语法是在 else 关键字后加上条件表达式，例如：

```
if (条件表达式) {
    代码块
} else if (其他条件表达式) {
    代码块
}
```

在示例 3.2 中，会根据 stateCode 变量值设置不同的变量值。该示例将判断 3 个不同的州，并赋予不同的税收比例。

示例 3.2　多条件判断的分支语句

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>if...else</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript">
//<![CDATA[
```

```
function choices() {
   var stateCode = 'MO';
   var taxPercentage = 0.0;

   if (stateCode === 'OR') {
        taxPercentage = 3.5;
   } else if (stateCode === 'CA') {
        taxPercentage = 5.0;
   } else if (stateCode === 'MO') {
        taxPercentage = 1.0;
   } else {
        taxPercentage = 2.0;
   }

   alert(taxPercentage);
}

//]]>
</script>

</head>
<body onload="choices()">
<p>Imagine a form with options to pick state code</p>
</body>
</html>
```

程序会依次执行每一个条件判断，直到某个条件返回真值为止。接着执行相关联的代码块，然后执行条件语句之后的代码。如果没有任何一个条件判断为真，那么将执行最后的 else 语句，并相应地设置税收比例。

提示

我一般会用单引号来引用简单的数值（如'OR'或'CA'），而用双引号来引用短语（如"this is a state"）。然而，单引号与双引号之间并没有本质的区别。本书将会交替使用这两种符号。

你可以不断添加 else if 语句来对同一个变量的值进行判断，不过这样的代码可读性较差。更好的方法是使用 switch 语句。

3.3.2 switch 条件语句

JavaScript 中的 switch 语句适用于要对同一个表达式的值进行多次判断的情况。JavaScript 引擎将执行这个表达式，根据返回的数值执行相应的代码块，例如：

```
switch (表达式) {
   case 第一个值:
      语句;
       break;
   case 第二个值:
```

```
    语句;
     break;
...
case 最后一个值:
    语句;
    break;
default:
    语句;
```

在 switch 语句中会计算表达式的数值，接着从上到下依次执行每一个 case 语句。如果 case 语句的数值与表达式的数值相等，那么将执行相应 case 语句中的代码块。如果代码块中有 break 语句，那么会跳过下列的 case 语句，并跳转到 switch 语句之后的语句中。否则，程序会继续执行其他 case 语句中的代码块。

如果没有符合的 case 语句，那么 JavaScript 引擎会查找 default 语句；如果存在 default 语句，那么就会执行 default 语句下的代码块。在下面的代码示例中，如果变量值为 3，那么将执行第一个代码块，如果变量值为 4，那么执行第二个代码块，其他数值则会执行 default 代码块。

```
switch(某变量) {
    case 3:
        代码块;
        break;
    case 4:
        代码块;
        break;
    default:
        代码块;
}
```

如果要对不同的数值执行一样的代码，那么可以在 case 语句中列出每个数值，并且将代码块放在最后一个 case 语句中。

```
case 数值 1:
case 数值 2:
case 数值 3:
    代码块;
    break;
```

在上面的代码中，当表达式数值为数值 1、数值 2 或数值 3 时，都会执行相同的代码块。

接着我们通过一个示例来介绍 switch 语句。示例 3.3 通过判断州名来根据不同州设置相应的税收比例，如果是 OR、MA 或 WI 州，那么税收比例为 3.5，州税率为 0.5；如果是 MO 州，那么税收比例为 1.0，州税率为 1.5；如果是 CA、NY 或 VT 州，那么税收比例为 4.5，州税率为 2.6；如果是 TX 州，那么税收比例为 3.0，而州税率为 0；否则，税收比例为 2.0，州税率为 2.3。

示例 3.3　switch 语句

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>switch</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript">
//<![CDATA[

function choices() {
   var stateCode = 'MO';
   var statePercentage = 0.0;
   var taxPercentage = 0.0;

   switch (stateCode) {
      case 'OR':
      case 'MA':
      case 'WI' :
         statePercentage = 0.5;
         taxPercentage = 3.5;
         break;
      case 'MO' :
         taxPercentage = 1.0;
         statePercentage = 1.5;
      case 'CA' :
      case 'NY' :
      case 'VT' :
         statePercentage = 2.6;
         taxPercentage = 4.5;
         break;
      case 'TX' :
         taxPercentage = 3.0;
         break;
      default :
         taxPercentage = 2.0;
         statePercentage = 2.3;
   }

   alert("tax is " + taxPercentage + " and state is "
         + statePercentage);
}

//]]>
</script>
</head>
<body onload="choices()">
<p>Imagine a form with options to pick state code</p>
```

```
</body>
</html>
```

在最前面的 switch 语句中，表达式只是 stateCode 变量。你还可以在 swtich 语句中的表达式里添加与/或等逻辑操作符（将在下一小节中讨论）。

case 语句会判断自己的数值是否与 stateCode 变量值相等，如果是 OR、MA 或 WI 州，那么税收比例是一样的，因为这些数值关联着相同的代码块。CA、NY 或 VT 州也是同样的逻辑。

如果是 TX 或 MO 州，那么会执行它们各自的代码块。然而，对 MO 州，taxPercentage 变量值会是 4.5，statePercentage 变量值是 2.6，而不是所预期的 1.0 和 1.5。这是因为 MO 州相关联的代码块中没有 break 语句，所以程序会接着执行下面的 case 语句，直到遇到第一个 break 语句才退出。所以在示例 3.3 中会继续执行 CA、NY 或 VT 的代码块，taxPercentage 和 statePercentage 变量值也就相应地设置为 4.5 和 2.6。

最后，如果没有相应的 case 语句就将执行 default 语句，接着是 switch 语句结构之后的代码。

请注意，示例只在 switch 控制块中使用了一对大括号。这是由于 switch 的流程取决于 break 语句，所以不需要大括号。当然，switch 语句也可以采用代码缩进，毕竟将 case 条件和相关联的代码放在同一行的做法并不常见：

```
case 'OR' : taxPercentage = 3.5; statePercentage = 2.0; break;
```

大多数通过条件控制语句判断的表达式都是简单的相等判断。当然也可以使用更加复杂的条件表达式，甚至是多条件操作符的表达式。

3.4 条件操作符

条件操作符可以用来执行特定的条件判断，如相同、相等、关系、逻辑等。尽管条件判断不尽相同，条件可能很简单也可能很复杂，但它们的结果必然要么为真，要么为假。

3.4.1 相同和相等操作符

条件表达式中最常见的操作符是相等操作符（==）。该操作符可以用来比较变量值是否相等，例如：

```
// 将 nValue 赋值为 3
var nValue = 3;
...
if (nValue == 3) ...
```

在该示例中，如果变量 nValue 的值等于 3，那么条件判断将返回 true，并执行 if 语句中的代码块。否则，就将跳过该段代码。

警告

请特别注意不要遗漏第二个等号。如果只有一个等号，那么表达式将变成赋值语句，而不是条件判断。在之前的代码中，如果遗漏了第二个等号，那么就会将 nValue 变量赋值为 3，而赋值之后“条件判断”必将返回 true。

相等操作符会自动转换变量的数据类型。如果一个数值为数字型，而另一个数值是字符串型，那么相等操作符会转换变量的数据类型，判断同类型的时候数值是否相等，例如：

```
var nValue = 3.0;
var sValue = "3.0";
If (nValue == sValue) ...
```

当然，这种隐式的类型转换（转型）可能会导致一些出乎意料的结果。

switch 语句中隐式地使用了相等操作符，下列两个 case 语句是一样的，都判断 switch 表达式是否等于“3.0”：

```
case 3.0: ...
case "3.0": ...
```

从 JavaScript 1.3 开始，JavaScript 就能够支持一种新的操作符，即相同操作符，它是一种更为严格的相等操作符，用来判断变量的数值和类型。相同操作符使用 3 个等号，以和相等操作符的两个等号相区别。与一般的相等判断有些不同，相同操作符只有在操作数的数值相同且类型相同的情况下才返回 true。

```
if (nValue === sValue)  // false
```

此外还可以通过不相等和不相同的操作符来进行判断，不相等的操作符是!=：

```
if (sName != "Smith") ...
```

而不相同的操作符是!==：

```
if (sName !== "Smith")
```

示例 3.4 对数字变量和字符串值进行判断，首先是相等性判断，接着是相同性判断。接着再对字符串变量和数字值进行同样的判断。

示例 3.4　相等和相同的判断

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>Equality</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript">
//<![CDATA[
```

```
function choices() {
   var sValue = "3.0";
   var nValue = 3.0;

   if (nValue == "3.0") alert("According to equality, value is 3.0");

   if (nValue === "3.0") alert("According to strict equality, value is 3.0");

   if (sValue != 3.0) alert ("According to equality, value is not 3.0");

   if (sValue !== 3.0) alert ("According to strict equality, value is not 3.0");
}

//]]>
</script>
</head>
<body onload="choices()">
<p>Some page content</p>
</body>
</html>
```

在第一种情况中，数字变量值为 3，与字符串“3.0”进行相等比较，结果为 true，因此将弹出对话窗口。然而，相同比较则会返回 false。

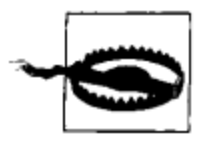
在第三种情况中，字符串变量“3.0”与数字 3.0 进行比较，不相等操作符将返回 false，因为该操作符认为这两个数值是相等的。而如果是不相同操作符则返回 true，因为该操作符认为这两个数值不是相同的（数据类型不同）。

提示

在示例 3.4 中，if 语句里没有使用大括号，而是直接将处理语句放在条件语句的后面，因为只有一条语句需要处理，所以代码也相当简洁。

正如示例 3.4 所示，相同比较操作符显得更为精确，但是为什么该操作符并没有被广泛使用呢?

相等和不相等操作符，从 JavaScript 出现以来就存在，并且所有 JavaScript 引擎都支持这两种操作符。而更为严格的相同和不相同操作符则是后来才添加到 JavaScript 标准中的。而且，在 ECMA-262 标准第一版中抛弃了相同操作符，却又在第三版中添加回来。另外老版本的浏览器也不支持这种新的操作符。不过，大多数现在使用的浏览器（包括 IE 6.0），都支持相同操作符，所以可以较为放心地使用该操作符。



警告

在不能保证变量类型一致的情况下，那么请使用相同和不相同操作符进行判断。

相等性的判断是非常实用的，然而有些时候也需要对数值的范围进行判断，而不仅仅是特定的数值，这时就需要大于、小于等关系操作符。

3.4.2 其他关系操作符

关系操作符是一种以某种方式比较操作数的方法。相等和相同操作符都是关系操作符，用来判断两个操作数是否相等。在某些情况下，关系操作符可以用于判断操作数之间的大小关系。

大于操作符（>）：如果左边的操作数大于右边的操作数（基于相同的数据类型），则返回 true。例如，数字 4 大于 1，而字符串“one”则比字符串“four”来得大。

```
var a = 1; var b = 4;
if (a > b)     // false
var a2 = "one"; var b2 = "four";
if (a2 > b2)   // true
```

大等于操作符（>=）：如果左边的操作数大于或等于右边的操作数，则返回 true。

```
var nValue = 1.0;
if (nValue > 3.0)    // false
...
if (nValue >= 1.0)   // true
...
if (nValue >= 0.5)   // true
...
```

小于操作符（<）：如果左边的操作数小于右边的操作数，则返回 true。

小等于操作符（<=）：当左边的操作数小于或等于右边的操作数的时候才返回 true。

```
var nValue = 1.0

if (nValue < 3.0)    // true
...
if (nValue <= 1.0)   // true
...
if (nValue <= 0.5)   // false
...
```

与相等操作符一样，小于操作符、大于操作符也会自动进行类型转换。所以，下列判断将返回 false：

```
sValue = "1.0";
if (sValue >= 2.0)   // false
```

只有当数值格式为数字的时候，才会进行字符串转换。例如，JavaScript 不会将“one”隐式转换成“1”或者“1.0”。

对于非数字值，大于或小于操作符将以字符顺序比较两个操作数：

```
var strValue = "apple";
if (strValue < "banana")      //true
if (strValue > "banana")      //false
```

判断数值之间的大小是非常实用的功能，然而有些时候需要判断变量或表达式的数值是否在某个范围之内。示例 3.5 将判断变量值是否在 0～100 之间（包括 0 和 100）。该示例还将判断变量值是否在 0～100 之间，但不包括 0 和 100。最后判断变量值是否大于 100 或者小于 0。该示例将根据不同的判断结果显示相应的信息。

示例 3.5　数值范围的判断

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>Ranges of values</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript">
//<![CDATA[

function values() {
   var nValue = 0;

   if (nValue >= 0 && nValue <= 100) {
       alert("value between 0 and 100 inclusive");
   }
   if (nValue > 0 && nValue < 100) {
       alert("value between 0 and 100 exclusive");
   }
   if (nValue > 100) {
       alert ("value over 100");
   }
   if (nValue < 0) {
       alert ("value is negative");
   }
}
//]]>
</script>
</head>
<body onload="values()">
<p>Some page content</p>
</body>
</html>
```

在该示例中，数值为 0，所以只有“数值在 0～100 之间，且包括 0 和 100”的判断返回 true，所以也只显示一条信息。

在前两个判断中，是通过额外的操作符来完成范围判断的：逻辑与操作符 &&。后续的章节还将对逻辑操作符做更多介绍，但是接下来我们还是先来了解一下 JavaScript 中

唯一的三元操作符。

3.4.3 JavaScript 中唯一的三元操作符

到目前为止，本章所介绍的操作符都是一元或二元操作符。JavaScript 语言还有一个三元操作符，也就是需要有 3 个操作数的条件操作。例如：

```
var nValue = 1.0;
var sResult = (nValue > 0.5) ? "value over 0.5" : "value not over 0.5";
```

在该示例中，sResult 的值为“value over 0.5”，因为条件判断为真，因此将返回第二个操作数的数值。这个条件操作符的格式为：

```
condition ? value if true : value if false;
```

这个条件操作符的功能与 if…else 语句类似，但语法显得更为简洁。如果有这样一个 If…else 语句：

```
var stateCode = 'OR';
var taxPercentage = 0.0;
if (stateCode == 'OR') {
      taxPercentage = 3.5;
} else {
      taxPercentage = 4.5;
}
```

如果要将该 if 语句转换为条件操作符表示，那么其代码为：

```
var stateCode = 'OR';
var taxPercentage = 0.0;
var taxPercentage = (stateCode == 'OR') ? 3.5 : 4.5;
```

这样的语法简洁且可读性高，应用也非常普遍。本书将在后面的章节使用这一操作符来解决浏览器之间的差异性问题。

3.5 逻辑操作符

本书目前为止所展示的示例所使用的条件表达式都是由一个操作符、两个操作数组成的，例如：

```
if (sValue == 'test')
```

然而，条件表达式很多时候是由多个不同的条件所组成的。每个条件都有自己的表达式，这些条件的结果将通过 JavaScript 的逻辑操作符计算出最终的结果。

逻辑操作符有三种：两个二元操作符和一个一元操作符。第一个操作符是逻辑与，由两个&符号所组成，即&&。当与操作符用于条件语句的时候，那么只有左右两个表达式都为 true，整个表达式的结果才为 true。

```
var nValue = 10;
if ((nValue > 10) && (nValue <=100))    //false，因为 nValue 值不大于 10
```

这个使用了与操作符的表达式的最终结果是 false，因为变量 nValue 等于 10，也就是第一个条件表达式（nValue > 10）不成立。如果第一个表达式为假，那么 JavaScript 引擎不会处理第二个表达式，因为无论第二个表达式的结果是什么，整个表达式的结果都将为假。

第二个逻辑操作符是或操作符，它是由两个 | 符号组成的，即 ||。当在条件语句中使用或操作符的时候，只要左右两个表达式中任意一个为 true，整个表达式的结果就为 true。

```
var nValue = 10;
if ((nValue > 10) || (nValue <= 100))     //ture，因为 nValue 值小于 100
```

上述代码的最终结果为 true，因为变量 nValue 值小于 100。或操作符的左右两边的表达式都会执行，因为第一个条件判断为 false，于是要执行第二个条件判断是否为 true。当介绍逻辑操作符的时候，通常会提及操作符左右两边的条件判断是否会执行。这是因为 JavaScript 引擎遵循最短路径的原理来判断表达式。如果是与操作符且第一个表达式为 false，那么就不会执行第二个表达式，因为整个表达式的结果一定为 false。如果是或操作符且第一个表达式为 true，那么也不会执行第二个表达式，因为任意一个表达式结果为 true，整个表达式结果就为 true。

基于最短路径的原理，第一个表达式往往应该是 CPU 或资源消耗比较少的那个表达式，从而提高程序的性能。特别是，如果需要检查变量是否为空或者未定义，那么先进行这样的检查，再进行其他的处理：

```
if ((nValue != null) && (nValue > 8))
```

这也可以预防由于变量未定义而导致的 JavaScript 错误。

提示

请充分利用最短路径的优势，把最不消耗资源的表达式作为逻辑与和逻辑或操作符的第一个表达式。

最后一个逻辑操作符是逻辑非操作符，形式为一个感叹号，即!。该操作符将返回与表达式的逻辑相反的结果。如果表达式结果为 true，那么就将返回 false，如果结果为 false，那么就将返回 true。

```
var nValue = 10;
if (!(nValue > 10))    // 返回 true
```

在上述的代码中，嵌套内的表达式（nValue > 10）的结果为 false，因为变量值不大于 10。然而，整个表达式的结果应该是 true。

请注意，尽管示例中的表达式是通过圆括号引用的，但是这个圆括号并不是必需的；条件操作符的优先级要比逻辑操作符更高，所以会先执行。示例 3.5 中就没有使用圆括

号。然而，通过圆括号引用表达式可以提高代码的可读性，特别是复杂的条件表达式。

提示

将逻辑与和逻辑或操作符左右两边的表达式放在圆括号中，这是一种很好的编码规范，它不仅仅能提高代码的可读性，还可以避免出现意外的结果。

3.6 高级语句：循环

循环语句与条件分支语句非常类似，都是基于条件表达式判断是否执行相应的代码块。然而在循环语句中，当表达式为 true 的时候，那么就会执行一次循环并回到这个条件表示式，而不是执行循环语句后面的代码。

3.6.1 while 循环

JavaScript 中最简单的循环是根据每次循环开始的条件表达式进行判断，表达式为 true 则继续循环。循环中的代码块会更改条件判断所涉及的操作数，从而使得条件判断为 false，并退出循环。关键字 while 就是为这种类型的循环设计的。

在示例 3.6 中将判断条件表达式中变量值是否大于 10。在循环代码块中，将把变量值加到字符串变量中，并递增该变量。当变量大于 10 时，循环结束，字符串变量则通过 alert 对话框显示出来。

示例 3.6 在 while 循环中判断条件值

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>while loop</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript">
//<![CDATA[

function loops() {

   var strValue = "";
   var nValue = 1;

   while (nValue <= 10) {
      strValue+=nValue;
      nValue++;
   }
   alert(strValue);
}
```

```
//]]>
</script>
</head>
<body onload="loops()">
<p>Some page content</p>
</body>
</html>
```

该程序运行后将弹出一个 alert 对话框，并显示 12345678910。

nValue 变量值将随着每次循环逐渐增加，并且在字符串相加的时候隐式地转换为字符串。

3.6.2 do...while 循环

在前一小节中介绍了 while 循环语句，它在每次循环执行前都要判断条件表达式。如果条件不成立，那么就不会执行循环代码块。在某些情况下，你可能希望无论条件表达式成立还是失败，都至少要执行一次代码，那么请使用 do…while 循环。

与 while 循环不同的是，do…while 循环不会根据条件表达式决定是否执行相应的代码块，而是至少会执行一次代码。示例 3.6 中的 while 循环也可以修改成 do…while 形式，如下：

```
do {
    strValue+=nValue;
    nValue++;
} while (nValue <= 10)
```

如果 nValue 的初始化为 11，那么循环内的代码仍然会执行一次。

while 循环和 do…while 循环一样，都是根据条件表达式决定是否继续循环。条件表达式可以是简单的判断，也可以是复杂的条件组合，例如：

```
while (nValue >= 1 && nValue < 10) ...
```

在上述代码中，只要 nValue 变量值大于等于 1，并且小于 10，就会继续执行循环的代码块。这是一种很好的方法，然而，如果要使循环执行一定的次数，更好的选择是 for 循环。

3.6.3 for 循环

比起未知的条件表达式，for 循环更倾向于让循环执行一定的次数。for 循环有不同的类型，但是所有浏览器只实现了一种版本。

在最常使用的 for 循环中，首先为某个变量设定初始数值，并在每次循环中修改数值，然后在每次循环前进行同一个条件的判断。当变量的数值不再符合条件时就退出循环。

```
for (设定初始值; 条件; 更新数值) {
    ...
}
```

for 循环一般由 3 个不同的语句组成：赋值语句（设定初始值）、条件判断语句以及更新语句，例如：

```
for (var i = 0; i < 10; i++) {
     document.writeln("hello");
}
```

变量 i 初始化为 0，每次循环前，都会判断条件表达式是否成立，也就是变量 i 是否小于 10；如果条件成立，那么则执行循环内的代码块，并且更新变量 i 的值。for 循环的 3 个要素如下。

- 赋值语句：var i = 0;
- 条件判断语句：i < 10
- 更新语句：i++

for 循环的第二种形式是对循环计数值做递减操作：

```
for (var i = 10; i > 0; i--)
```

示例 3.7 用了一个 for 循环，创建出与前面示例（while 循环）相同的字符串。程序接着将字符串变量重新赋值为空字符串（""），并且反向遍历这些数字。

示例 3.7　通过 for 循环对数据做正向和反向遍历

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>for loop</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript">
//<![CDATA[

function doFor() {

   var strValue = "";

   for (var i = 1; i <= 10; i++) {
        strValue+=i;
   }
   alert(strValue);

   strValue = "";
   for (var i = 10; i >= 0; i--) {
        strValue+=i;
   }
   alert(strValue);
}
//]]>
```

```
</script>
</head>
<body onload="doFor()">
<p>Some page content</p>
</body>
</html>
```

第一次循环的初始值为 1，所以 for 循环将从 1 开始进行循环。

无论是增加或减少数值，for 循环取决于如何使用循环变量，而不仅仅是满足条件判断，例如通过该循环变量访问数组元素。然而，访问数组元素的另一种方法是 for 循环的另一种形式。这种形式与数组相关，本书将在下一章中介绍数组时说明该形式。

for 循环的第三种形式是 for…in 循环，它可以遍历某个对象的所有属性。尽管本书还没有介绍如何创建自定义对象，我们还是先演示一下它的用途，在示例 3.8 中我们创建了一个自定义对象 MyText，它包含 3 个属性：one、two 和 three。每个属性都进行初始化，接着我们通过 for…in 循环将每个属性的数值输出到网页上。

示例 3.8　通过 for…in 循环访问对象属性

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>for...in loop</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript">
//<![CDATA[

function doFor() {

  var MyText = {
     one : "one",
     two : "two",
     three : "three"
  };

  for (var prop in MyText) {
    document.writeln(prop + "<br />");
  }

}
//]]>
</script>
</head>
<body onload="doFor()">
<p>Some page content</p>
</body>
</html>
```

该示例的执行结果如下所示：

```
one
two
three
```

for…in 循环也同样可以用于数组，然而循环的属性值是每个数组元素的索引，与一般的 for 循环较为不同。

```
var tsts = new Array('one','two','three');
for (indx in tsts) {
   alert(tsts[indx]);
}
```

因为传统形式的可读性较高，所以可以在处理数组时使用传统形式，将 for…in 形式应用于对象属性的处理。

3.7 知识测验

1. 请为下列表达式添加括号，从而使得表达式的结果等于 8。

```
var valA = 37;
var valB = 3;
var valC = 18;
var resultOfComp = valA - valB % 3 / 2 * 4 + valC - 3;
```

2. 请使用 switch 语句，判断表达式的数值是否为'one'、'two'或者'three'，当表达式为'one'或'two'的时候，将变量值设置为'OK'；当表达式为'three'的时候，将变量值设置为'OK2'；如果没有匹配的值，就将变量值设置为'NONE'。

3. 假设有 3 个变量 varOne、varTwo 和 varThree。那么如何对这两个变量进行判断，从而仅当 varOne 为 33，varTwo 小于等于 100，但是 varThree 大于 0 的时候执行某段代码？

4. 如果希望执行某段代码 6 次，有哪三种方式？对这三种方式如何进行取舍？

5. 下列条件表达式是否有问题？如果有，那么是什么问题？

```
if (valTest1 == valTest2) ...
```

3.8 测验答案

1. 其解决方案是：

```
var resultOfComp = (valA - valB) % 3 / 2 * (4 + valC) - 3;
```

2. 其解决方案是：

```
switch(val) {
   case 'one' :
   case 'two' :
      result = 'OK';
      break;
   case 'three' :
      result = 'OK2';
      break;
   default :
      result = 'NONE';
}
```

3. 其解决方案是：

```
if ((varOne == 33) && (varTwo <= 100) && (varThree > 0))
```

4. 其答案是：

```
for (var i = 0; i < 6; i++) {
      ...
}

i = 0;
while (i < 6) {
   ...
   i++;
}

i = 0;
do {
   i++;
   ...
   }
while (i < 6)
```

一般来说，do…while 更适用于无论条件是否成立都至少要执行一次代码的情况下。for 循环更适用于代码需要执行特定次数时。而 while 循环则适用于循环代码会影响条件表达式判断的情况。

5. 如果不确定变量的数据类型，那么请考虑使用相同操作符：

```
if (valTest1 === valTest2)
```

第 4 章

JavaScript 对象

JavaScript 对象是 JavaScript 语言中固有的组件，而且与 JavaScript 的执行环境无关。所以无论在什么环境下都可以访问 JavaScript 对象。

在 JavaScript 基本对象中，有一些是与数据类型平行的对象，例如表示字符串的 String、表示布尔值的 Boolean 以及表示数字的 Number。这些对象封装了基本类型，并在其基本功能上进行一定的扩展。

此外还有 3 种提供其他功能的内建对象，Math、Date 及 RegExp。Math 和 Date 就不需要做太多介绍了，它们分别提供了数学运算和日期的功能。RegExp 则是提供正则表达式功能的对象。正则表达式通过提供模式匹配的功能，可以对字符串进行精确的查找或匹配。

JavaScript 还有另外一个内建的集合型对象，即 Array。事实上，JavaScript 中的所有对象都是数组，只是在实际使用中并没有将对象当成数组来访问而已。在后续的章节中将会详细介绍数组。现在，我们先来回顾一下前面提到的针对基本数据类型的对象，以理解 JavaScript 语言中的对象究竟是什么。

4.1 基本数据类型对象

在本书后续的内容中将介绍 JavaScript 面向对象的特性。现在先来看看 JavaScript 内建对象的一些常见的面向对象功能。

JavaScript 对象也拥有方法和属性，可以通过对象属性操作符 (.) 进行访问。例如，String 对象的长度可以通过该对象的 length 属性访问：

```
var myName = "Shelley";
alert(myName.length);
```

对象的方法也可以通过属性操作符进行访问，因为在 JavaScript 中，方法也被认为是对

象的属性。下面的示例调用了 String 对象的 strike 方法，该方法将字符串中的内容输出到 HTML 标签中，例如：

```
var myName = "Shelley";
alert(myName.strike());      //返回<strike>Shelley</strike>
```

读者可能会对该示例存在一些困惑，因为该示例只是创建了一个字符串基本类型，而不是 String 对象。没错，在代码中的确只是创建了字符串基本类型的变量 myName。然而，当在该变量上调用与 String 相关的方法时，甚至是 String 的所有属性时（包括 length 和 strike 方法），那么该变量就会被当做是 String 对象实例。

在第 2 章中曾经说过，当在基本数据类型上调用对象方法时，JavaScript 将创建基本数据类型的对象实例，执行方法调用，并销毁该临时对象。该原则同样适用于数字和布尔等数据类型。

然而，除了隐式创建 String、Boolean 或 Number 对象之外，还可以显式地通过关键字 new 创建对象，语法如下：

```
var myName = new String("Shelley");
```

new 关键字是非常重要的，如果省略了该关键字，那么它将只是字符串基本类型，而不是 String 对象。也就是说，下面两行 JavaScript 代码都只是创建字符串基本类型：

```
var strName = "Shelley";
var strName2 = String("Shelley");
```

回到 String 对象，当创建了 String 对象实例后，可以通过 valueOf 方法访问相应基本类型的数值，例如：

```
var myName = new String("Shelley");
alert(myName.valueOf());
```

此外还可以直接访问基本类型的数值，例如：

```
var myName = new String("Shelley");
alert(myName);
```

这里再重复一次在第 2 章中曾经介绍过的字符串基本类型和字符串对象实例相互转换的过程。如果要访问对象属性，如 String 对象的 length 和 strike 方法，那么可以创建对象变量。如果创建的是字符串基本类型，却又以对象的行为访问，那么 JavaScript 会将该基本类型自动转换成对象，但是转换的 String 对象只是一个临时变量，并且在属性操作后销毁该对象，所以这种操作不够有效，多了一次转换的过程。

```
var strName ="Shelley";     //字符串基本类型
alert(strName.length);      //隐式创建 String 对象，数值与 strName 相同，并执行
                              length 方法
```

不过，如果不需要访问基本类型的对象属性，那么可以仅仅创建字符串基本类型的变量，例如：

```
var strName = "Shelley";
alert(strName);
```

因为本章主要介绍对象属性，所以在余下的内容中都将创建对象实例。

4.2 布尔值、数字和字符串

Number 和 String 对象实例都有各自特殊的属性，而唯独 Boolean 没有。不过，所有这 3 种对象都将继承基类 object 中定义的属性和方法。所有对象继承的方法包括 toString 和 valueOf。由于 Boolean 对象只有继承的属性，接下来就逐一介绍该对象类型的每个方法。

4.2.1 Boolean 对象

创建 Boolean 对象实例有不同的方法，这取决于其初始值是 false 还是 true。如果创建的对象不指定初始值，那么该 Boolean 对象的默认初始值为 false：

```
var boolFlag = new Boolean();
```

初始值还可以通过数字的方式设置，例如用 0 代表 false：

```
var boolFlag = new Boolean(0);
```

或者用 1 代表 true：

```
var boolFlag = new Boolean(1);
```

此外，还可以通过 true 和 false 来设定初始值，例如：

```
var boolFlag1 = new Boolean(false);
var boolFlag2 = new Boolean(true);
```

如果以空字符串创建 Boolean 对象实例，那么对象的初始值将为 false。如果以任何非空字符串创建 Boolean 对象实例，那么对象的初始值将为 true。

```
var boolFlag1 = new Boolean("");          // 初始值为 false
var boolFlag2 = new Boolean("false");     // 初始值为 true
```

无论字符串的内容是什么，即使是“false”，只要字符串非空，那么对象实例的初始值就是 true。下面的示例 4.1 通过 valueOf 方法来证明这点。

示例 4.1　通过 valueOf 方法判断 Boolean 对象实例的初始值

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>Boolean valueOf</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript">
```

```
//<![CDATA[

function newBool() {
   var boolFlag = new Boolean("false");
   alert(boolFlag.valueOf());
}

//]]>
</script>
</head>
<body onload="newBool()">
<p>Some page content</p>
</body>
</html>
```

弹出的 alert 对话框如图 4.1 所示，从中可以看到 Boolean 实例的值为 true。valueOf 方法将返回 Boolean 对象封装的基本类型的数值。

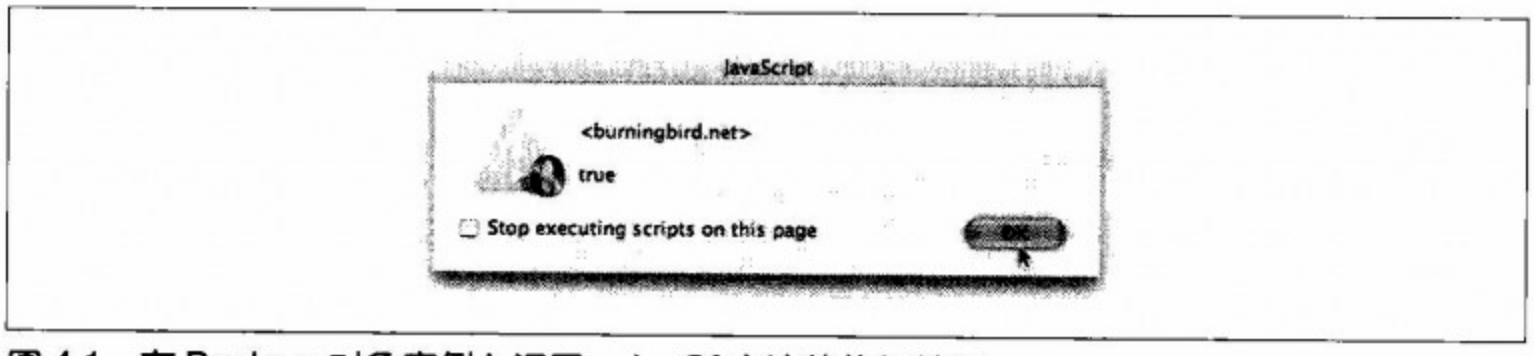


图 4.1　在 Boolean 对象实例上调用 valueOf 方法的执行结果

toString 方法的执行结果与 valueOf 相同，不同的是返回的结果是字符串形式，而不是布尔值，例如：

```
var boolFlag = new Boolean("false");
var strFlag = boolFlag.toString();      // 字符串"true"
```

4.2.2　Number 对象、静态属性及实例方法

在介绍 Number 对象之前，我们先了解一下实例对象的属性。

前一小节中介绍的 Boolean 对象提供的两个方法，俗称为实例方法，因为这些方法是基于对象实例 boolFlag 的，而不是基于对象类 Boolean 的。无论是 toString 或 valueOf 方法都必须基于对象实例，将对象实例的数值转换为字符串或者返回基本类型数值。

对象方法的另一种形式是静态方法。静态方法不是基于对象实例的，而是直接为对象类所调用的。所有 Math 对象的方法都是静态方法：

```
var newNum = Math.abs(oldNum);
```

实例而非对象的实现还适用于属性，而不是方法。例如，Number 类的 5 个属性只可以通过 Number 对象进行访问：

- Number.MAX_VALUE　JavaScript 所能表示的最大数值；
- Number.MIN_VALUE　JavaScript 所能表示的最小负数；
- Number.NaN　表示不是数字；
- Number.NEGATIVE_INFINITY　表示负无穷大；
- Number.POSITIVE_INFINITY　表示正无穷大。

无穷大属性通常只用来判断是否溢出。溢出表示数字太大或太小，超过了 MAX_VALUE 和 MIN_VALUE 属性：

```
var someValue = -1 * Number.MAX_VALUE  * 2;
alert(someValue);

someValue = Number.MAX_VALUE * 2;
alert(someValue);
```

alert 语句首先会显示 NEGATIVE_INFINITY 所表示的负无穷大，然后是 POSITIVE_INFINITY 所表示的无穷大。

我们在前面 4 个属性列表中使用 Number 对象，是强调访问这些属性是基于 Number 对象本身的，而不是 Number 对象实例。如果基于 Number 实例访问这些属性，那么将会返回 undefined。

```
var someNumber = new Number(3.0);
var maxValue = someNumber.MAX_VALUE; // undefined
```

Number 对象实例的方法可以用来将其转换为字符串、精确或固定位数的格式或者指数格式。Number 对象中有 3 种实例方法：

- toExponential　以字符串形式返回数值的指数格式；
- toFixed　以字符串形式返回数值的固定位数格式；
- toPrecision　以字符串形式返回数值的特定精确格式。

valueOf 和 toString 等全局方法同样适用于 Number 实例，此外还有 toLocaleString 方法。后者返回与环境相关的数值格式。与 Boolean 不同的是，Number 对象实例的 toString 方法可以有一个参数，该参数可以在十进制和十六进制，或者十进制和八进制之间进行转换。

示例 4.2 介绍了 Number 对象实例的不同方法。

示例 4.2　Number 对象的实例方法

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
```

```
<title>Number methods</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript">
//<![CDATA[

function numbers() {

   // Number方法
   var newNumber = new Number(34.8896);

   document.writeln(newNumber.toExponential(3) + "<br />");
   document.writeln(newNumber.toPrecision(3) + "<br />");
   document.writeln(newNumber.toFixed(6) + "<br />");

   var newValue = newNumber.valueOf();

   document.writeln(newValue.toString(2) + "<br />");
   document.writeln(newValue.toString(8) + "<br />");
   document.writeln(newValue.toString(10) + "<br />");
   document.writeln(newValue.toString(16) + "<br />");
}
//]]>
</script>
</head>
<body onload="numbers()">
<p>Some page content</p>
</body>
</html>
```

这个 JavaScript 应用程序在 Safari 中运行的结果如图 4.2 所示。

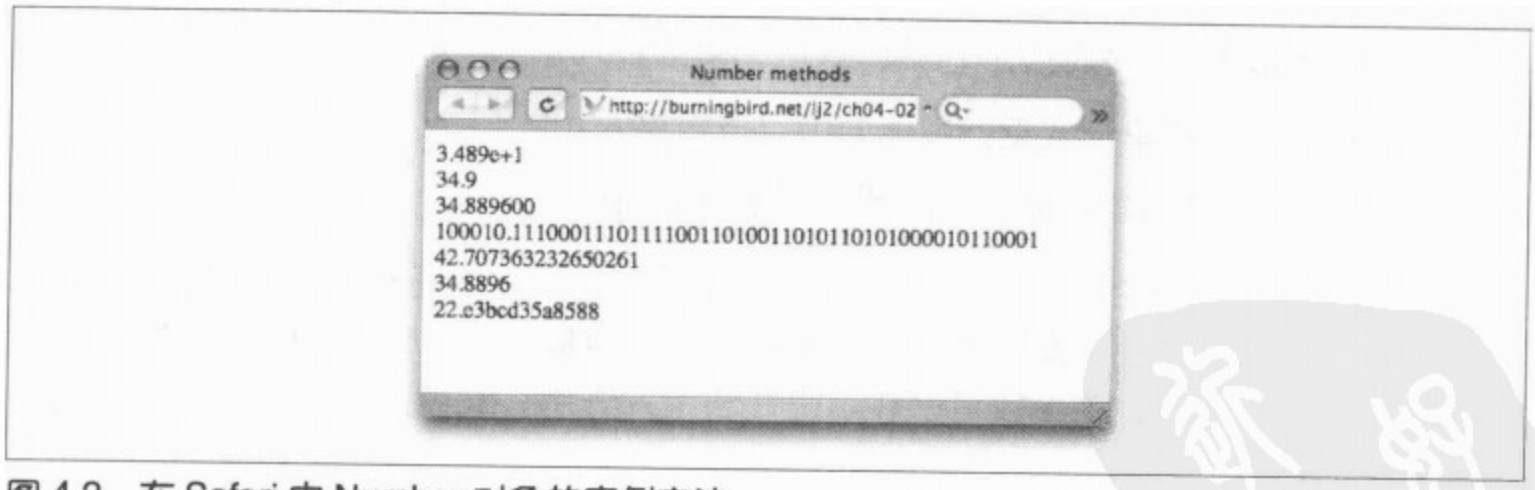


图 4.2　在 Safari 中 Number 对象的实例方法

这段代码中第一个调用的方法是 toExponential，它的参数是小数点的位数，在该示例中为 3。第二个调用的方法是 toPrecision，参数 3 表示字符串转换中数字的位数。第三个方法是 toFixed，表示十进制取整后的位数。

代码接着将 Number 对象的数值通过 valueOf 方法赋给新变量，接着调用新变量的

toString 方法。之所以这样做是因为如果要在 toString 方法中使用 base 参数，其前提是针对数字基本类型。变量 newNumber 是一个对象，而不是数字基本类型。所以只有在将真正数字赋给新的变量之后，才可以调用带参数的 toString 方法。

4.2.3 String 对象

String 对象是最常用的 JavaScript 内建对象，可以显式地通过 new String 构造函数创建新的 String 对象，并传入字符串作为参数：

```
var sObject = new String("Sample string");
```

String 对象有着不同的方法，有些方法用于 HTML，而有些方法则用于其他用途。表 4.1 中列出了 String 对象实例的属性和方法。

表 4.1 String 对象方法

方　法	描　述	参　数
valueOf	返回字符串对象封装的字符串数值	无
length	属性，字符串的文本长度	使用时不带括号
anchor	创建一个<a>元素	<a>标签中的字符串
big，blink，bold，italics，small，strike，sub，sup	以 HTML 格式化字符串对象中的值，并将其返回	无
charAt，charCodeAt	返回字符串中特定位置的字符（charAt）或字符代码（charCodeAt）	表示位置的数字，从 0 开始计数
indexOf	返回另一个字符串在该字符串中第一次出现的位置	所查找的子字符串
lastIndexOf	返回另一个字符串在该字符串中最后一次出现的位置	所查找的子字符串
link	返回针对链接的 HTML	href 属性的 URL 值
concat	连接字符串	所要连接的字符串
split	根据特定的分割符，分割字符串	分割符和分割的最大数量
slice	返回字符串的某个片段	该片段的起始和结束位置
substring，substr	返回子字符串	字符串的起始和结束位置
match，replace，search	正则表达式匹配、替换和查找	正则表达式
ToLowerCase，toUpperCase	大小写转换	无

上面有一个非针对 HTML 的方法，就是能将两个字符串连接起来的 concat。示例 4.3 介绍了创建 String 对象的方法，并使用了 concat 方法。

示例 4.3　创建 String 对象并调用 concat 方法

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>String concatenation</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript">
//<![CDATA[

function stringConcat() {

   var sObj = new String();
   var sTxt = sObj.concat("This is", " a ", "new string");
   alert(sTxt);
}
//]]>
</script>
</head>
<body onload="stringConcat()">
<p>Some page content</p>
</body>
</html>
```

concat 方法并没有限制要连接的字符串数量。然而，我并不鼓励使用 concat 方法，如果要构建一个字符串，我更倾向于使用字符串基本类型，并使用加号来连接字符串。当然，字符串连接的次数要尽可能的少。每次字符串连接都会创建一个新的 String 对象。在 JavaScript 中 String 对象是不可变的，也就是说，每次对字符串的修改也就意味着重新创建一个新的字符串，新旧字符串都会保留在内存中，除非该应用程序结束（一般就是函数执行结束）。

```
var newStr = "this is";
newStr+= " a new string";      // 丢弃旧的字符串，并创建新字符串
```

如果字符串连接操作过多，那么应用程序的性能就有可能受到影响。然而，大多数字符串连接是非常简单的，所以不必太担心性能问题。

如 anchor、link、big、blink、bold、italics、sub、sup、small 及 strike 等 HTML 格式化方法将生成表示 HTML 元素标签的字符串。示例 4.4 通过相同的字符串来介绍这些格式化方法，从而体现这些方法的不同之处。

示例 4.4　String 对象的格式化方法

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
```

```
<title>String HTML</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript">
//<![CDATA[

function stringHTML() {

var someString = new String("This is the test string");

document.writeln(someString.big() + "<br />");
document.writeln(someString.blink() + "<br />");
document.writeln(someString.sup() + "<br />");
document.writeln(someString.strike() + "<br />");
document.writeln(someString.bold() + "<br />");
document.writeln(someString.italics() + "<br />");
document.writeln(someString.small() + "<br />");
document.writeln(someString.link('http://www.oreilly.com'));
}
//]]>
</script>
</head>
<body onload="stringHTML()">
<p>Some page content</p>
</body>
</html>
```

该程序在 Firefox 中显示的结果如图 4.3 所示。注意，我们为每个字符串都添加 HTML 样式。在图片中不能显示的是通过 blink 格式化的字符串。blink 是不推荐使用的 HTML 格式化方法，因为新的 HTML 和 XHTML 标准已经不再支持该功能，浏览器也已经停止了对该功能的支持。并且如果 DOCTYPE 声明为 strict 类型的，那么该网页将无法通过校验。不过如果使用的是 document.writeln 方法的话是可以通过校验的，因为 XHTML 校验器能识别 JavaScript 语法，而无法识别执行后的结果。如果将执行后的结果复制到新文档中，再执行 XHTML 校验器，那么就会看到 blink 校验错误。

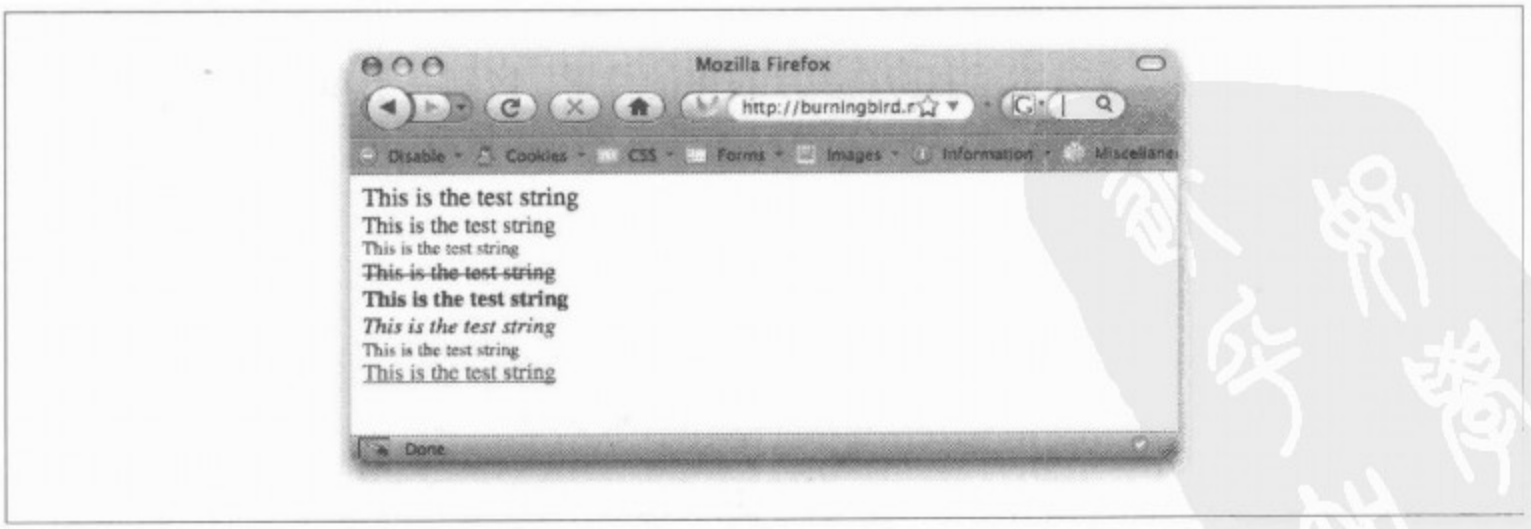


图 4.3　示例 4.4 所生成的页面

警告

即使没有任何错误，也应当尽量避免使用 HTML 格式化方法，主要是因为这些方法不支持 CSS 样式，并且在任何时候都应该避免使用 blink。

现在我们回到 String 对象的实例方法，charAt 和 charCodeAt 方法将根据给定的位置返回字符和 Unicode 字符编码。这些方法唯一的参数是字符的位置：

```
var sObj = new String("This is a test string");
var sTxt = sObj.charAt(3);
document.writeln(sTxt);
```

位置是从 0 开始计数的，例如如果要返回第 4 个字符，那么参数值应该为 3。

substr、substring 及 slice 方法，将根据相应的两个参数返回子字符串。substr 方法的参数分别是开始位置和子字符串的长度，而 substring 方法的参数则是子字符串的开始和结束位置。

```
var sTxt = "This is a test string";
var ssTxt = sTxt.substr(5,8);          // 返回"is a tes"
var ssTxt2 = sTxt.substring(5,8);      // 返回"is "
```

如果结束位置小于开始位置，那么 substring 方法将会自动交换这两个参数。如果省略表示结束位置的参数，那么子字符串将从开始位置开始，一直到字符串结束。如果第二个参数（位置或长度）大于整个字符串的位置或长度，那么返回的子字符串也将是一直到字符串结束为止。

slice 方法与 substring 类似，唯一的区别在于如果结束位置小于开始位置，那么不会自动交换这两个参数，而是返回空字符串：

```
var ssTxt2 = sTxt.substring(4,1);      // 返回 his
var ssTxt3 = sTxt.slice(4,1);          // 返回空字符串
```

和本章示例一样，字符串方法可以用于字符串基本类型，也可以是 String 对象。不过请记住，JavaScript 处理程序会将变量转换成对象，调用方法，再将对象转换成基本类型变量，最后销毁对象。

indexOf 和 lastIndexOf 方法将返回所查找字符串的位置，区别在于 indexOf 返回的是字符串第一次出现的位置，而 lastIndexOf 返回的是该字符串最后一次出现的位置：

```
var sTxt = "This is a test string";
var iVal = sTxt.indexOf("t");
document.writeln(iVal);
```

示例 4.3 中展示的是字符串连接。反之，如果要分割字符串则可以使用 split 方法。该方法有两个参数：第一个参数是分割符，第二个参数则是分割的数量。

示例 4.5 中介绍了通过逗号分割字符串的方法。在该示例中，分割后的结果将再根据等号做第二次分割。

示例 4.5　使用 split 方法分割字符串

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>Gotta Split</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript">
//<![CDATA[

function splitString() {

   var inputString = new
String('firstName=Shelley,lastName=Powers,state=Missouri,statement="This
is a test,of split"');
   var arrayTokens = inputString.split(',',3);

   // 按逗号分割字符串
   for (var i = 0; i < arrayTokens.length; i++) {
        document.writeln(arrayTokens[i] + "<br />");

      // now split on equals and write just value
      var newTokens = arrayTokens[i].split('=');
      document.writeln(newTokens[1] + "<br /><br />");
   }
}
//]]>
</script>
</head>
<body onload="splitString()">
<p>Some page content</p>
</body>
</html>
```

该示例的运行结果如下所示：

```
firstName=Shelley
Shelley

lastName=Powers
Powers

state=Missouri
Missouri
```

这是处理表单域的一种简便方法，在数据提交到服务器之前，从页面链接中获取单个数值，或者处理 Ajax 调用返回的结果。

我们再次回到 String 对象的实例方法上，toUpperCase 和 toLowerCase 方法用来将字符

串中的字母全部转换成大写或小写字符，并返回转换后的结果：

```
var someString = new String("Mix of upper and lower");
var newString = someString.toUpperCase();      // 将所有字符转换成大写
```

这两个方法特别适用于需要将字符串进行大小写转换的场合。

String 对象中还有另外一个静态方法，即 fromCharCode：

```
var s = String.fromCharCode(345,99,99,76);
document.writeln(s);
```

fromCharCode 方法的参数是 Unicode 数值，它将返回该编码对应的字符串。在前面的章节也介绍过，在字符串中可以直接写入 Unicode 编码的字符。

与字符串相关的最后一个方法是与正则表达式有关的。由于正则表达式属于 JavaScript 另外一个内建对象 RegExp，所以接下来的小节中将介绍与正则表达式相关的功能。

4.3 正则表达式和 RegExp

正则表达式是由字符串所组成的表达式，用于匹配、替换或者查找特定的字符串。大多数编程语言（包括 JavaScript 在内）都提供了正则表达式支持。

通过 RegExp 对象可以显式地创建正则表达式，当然也可以通过文字量方式创建正则表达式，下面这个是显式创建的例子：

```
var searchPattern = new RegExp('s+');
```

下面这行代码则是使用文字量方式创建的正则表达式：

```
var searchPattern = /s+/;
```

表达式中的加号表示字符 s 必须在字符串中出现 1 次以上。而字符串（/s+/）中的斜杠表示这是一个正则表达式，不是其他对象类型。

4.3.1 RegExp 方法：test 和 exec

RegExp 对象中只有两个实例方法：test 和 exec。test 方法将判断以参数传入的字符串是否与正则表达式相匹配。下面的示例将判断字符串是否匹配正则表达式/JavaScriptrules/：

```
var re = /JavaScript rules/;
var str = "JavaScript rules";
if (re.test(str)) document.writeln("I guess it does rule") ;
```

正则表达式的匹配过程是区分大小写的，如果表达式是/Javascriptrules/，那么结果将是不匹配的。如果希望不区分大小写进行匹配，那么可以在正则表达式后面添加字母 i，例如：

```
var re =/Javascript rules/i;
```

i 选项是正则表达式匹配的标识符，用来强制匹配过程忽略大小写。此外还有一些其他选项，包括表示全局匹配的 g 标识，表示多行匹配的 m 标识。

全局匹配是查找与正则表达式匹配的所有字符串，而忽略正则表达式的位置。如果不使用全局匹配选项 g，那么只会返回第一个匹配项。多行匹配选项 m 可以使用与行相关的字符，例如^表示一行的开始，$表示一行的结束，以便在多行的字符串中进行匹配。

如果是 RegExp 对象实例，那么第二个参数表示匹配选项，例如：

```
var searchPattern = new RegExp('s+', 'g');
```

在下面的代码中，RegExp 方法 exec 将根据正则表达式/JS*/在字符串中进行查找（全局匹配并忽略大小写）。返回的结果数组中，第一个元素就是匹配正则表达式的字符串，接着继续查找下一个匹配：

```
var re = new RegExp("JS*","ig");
var str = "cfdsJS *(&YJSjs 888JS";
var resultArray = re.exec(str);
while (resultArray) {
  document.writeln(resultArray[0]);
  document.writeln(" next match starts at " + re.lastIndex + "<br />");
       resultArray = re.exec(str);
}
```

正则表达式首先是字母 J，接着是任何数量的字母 S。由于使用了选项 i，所以匹配过程中将忽略大小写，因此也会查找到 js 字符串。并且由于设置了选项 g，RegExp 中的 lastIndex 属性会设置为上一次匹配的位置，所以每次 exec 调用都会查找下一个匹配。该示例中总共查找到了 4 次匹配，当没有匹配时，返回的数值将是空值。因为循环条件是基于数组变量的，所以当数组为空值时也会结束循环。该示例的执行结果如下所示：

```
JS next match starts at 6
JS next match starts at 13
js next match starts at 15
JS next match starts at 21
```

exec 方法将返回一个数组，但是数组元素并不是所有的匹配项，而是当前匹配项和所有带圆括号的子字符串。如果在表达式中使用圆括号引用正则表达式的某部分，那么匹配的时候，这些括号所匹配的字符串也会体现在返回的数组中。示例 4.6 在之前的示例基础上做了一些修改，数组其他元素体现带括号的正则表达式所匹配的内容。

示例 4.6　RegExp 所匹配的字符串和子字符串

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
```

```
<title>RegExp string matching</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript">
//<![CDATA[

window.onload=function() {
  var re = /(ds)+(j+s)/ig;
  var str = "cfdsJS *(&dsjjjsYJSjs 888dsdsJS";
  var resultArray = re.exec(str);
  while (resultArray) {
    document.writeln(resultArray[0]);
    document.writeln(" next match starts at " + re.lastIndex + "<br />");
    for (var i = 1; i < resultArray.length; i++) {
      document.writeln("substring of " + resultArray[i] + "<br />");
    }
    document.writeln("<br />");
    resultArray = re.exec(str);
  }
}
//]]>
</script>
</head>
<body>
<p></p>
</body>
</html>
```

该示例的执行结果如下所示：

```
dsJS next match starts at 6
substring of ds
substring of JS

dsjjjs next match starts at 16
substring of ds
substring of jjjs

dsdsJS next match starts at 31
substring of ds
substring of JS
```

这些示例介绍了一些应用于正则表达式的特殊字符。这些都是正则表达式的专有字符，例如示例 4.6 中的加号。有一些书和文章会将这些字符整理在一张表里，并且通过一个复杂的表达式来使用所有的这些字符，以展示这些字符的用法。然而，许多人还是对正则表达式的不同字符组合的使用存在困惑，正则表达式的匹配结果经常与他们预期的不同。展示正则表达式最好还是通过一系列示例来说明，从简单的到复杂的。如果曾经在其他语言中使用过正则表达式，那么可以跳过下一小节中对正则表达式应用的介绍。

尽管有 RegExp 对象，但是正则表达式更常用的方法是调用 String 对象中提供的正则表达式方法：replace、match 和 search。本节的其他示例都将通过这些方法来引入正则表达式。

4.3.2 正则表达式的应用

反斜杠，也称为转义符，用来对字符进行转义。在 JavaScript 的正则表达式中，转义符有两种用途。如果是一般的字符，例如字母 s，那么\s 表示特殊字符（空格、制表符、换行符等）。如果是特殊的字符，例如加号，那么\+表示是一般的加号，而不表示特殊字符。

示例 4.7 所示的 JavaScript 程序将查找紧接在空格之后的星号，并将其替换成横杠。正则表达式中的星号一般表示为字符匹配零次或多次，但是在本示例中只是当成一般的星号。

示例 4.7　正则表达式中的转义符

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>RegExp</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript">
//<![CDATA[

function matchString() {

var regExp = /\s\*/g;
var str = "This *is *a *test *string";
var resultString = str.replace(regExp,'-');
alert(resultString);

}
//]]>
</script>
</head>
<body onload="matchString()">
<p>Some page content</p>
</body>
</html>
```

应用正则表达式后的结果如下所示：

```
This-is-a-test-string
```

示例 4.7 中使用的正则表达式是非常简单的。如果希望将所有空格都替换成横杠，那么在 replace 方法中引用的正则表达式应该是/ \s+/g，替换字符则是横杠。

正则表达式中关于匹配字符出现次数的有 4 种字符：星号（*）表示字符出现零次或多次；加号（+）表示字符出现一次或多次；问号（?）表示字符出现零次或一次；点号（.）表示字符只出现一次。

匹配的模式有两种，贪婪匹配（.*）和延迟匹配（.*?）。前者由于点号可以表示任何字符，因此星号会尝试到最后一次匹配。如果希望查找引号内的任何字符，那么可以使用/".*"/。例如字符串是：

```
test='"one" or this is also a "test"'
```

那么匹配则会从第一个双引号开始，直到最后一个双引号，所以将得到：

```
"one" or this is also a "test"
```

而延迟匹配则强制在第二个双引号匹配的时候就结束，而不是最后一个双引号，所以将得到：

```
"one"
```

示例 4.8 中引入了一个较为复杂的正则表达式。该示例将查找特定格式的日期：月份、空格、日期、空格、年份。并且该日期以冒号作为开始符号。

示例 4.8　重复字符的正则表达式

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>Find Date</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript">
//<![CDATA[

function findDate() {

   var regExp = /:\D*\s\d+\s\d+/;
   var str = "This is a date: March 12 2009";
   var resultString = str.match(regExp);
   alert("Date" + resultString);

}
//]]>
</script>
</head>
<body onload="findDate()">
<p>Some page content</p>
</body>
</html>
```

该正则表达式的第一个字符是冒号，接着是反斜杠和大写字母 D，即\D，用来表示查

找任何非数字符号，接着的星号表示匹配任何数量的非数字符号。接下来的部分是空格符（\s），然后是\d。与\D 不同的是，小写的\d 表示只匹配数字。加号表示匹配一个或多个数字。接着是又一个空格符（\s），以及又一个\d+。

通过 String 的 match 方法匹配该字符串时，就会查找到符合表达式的日期，其结果如下：

```
Date: March 12 2009
```

在该示例中，\D 用来匹配任何非数字字符。另一种方法是使用方括号以及数字范围，并且利用脱字符（^）。如果想要匹配数字以外的任何字符，那么也可以使用：

```
[^0-9]
```

该方法也适用于\d，如果想要只匹配数字，那么可以使用：

```
[0-9]
```

如果希望匹配一种以上的字符类型，那么可以在方括号中列出这些字符。如下所示的正则表达式将匹配所有大写或小写字符：

```
[A-Za-z]
```

所以，示例 4.8 中的正则表达式也可以使用另一种形式表示，例如：

```
var regExp = /:[^0-9]*\s[0-9]+\s[0-9]+/;
```

脱字符（^）还有另一种用途，^和$可以表示一行的开始和结束。$匹配任何行结束符。在下列代码中，因为匹配字符不是出现在行的开始，所以匹配不成功。

```
var regExp = /^The/i;
var str = "This is the JavaScript example";
```

然而，以下的匹配是成功的：

```
var regExp = /^The/i;
var str = "The example";
```

如果使用多行匹配选项，脱字符也可以匹配换行符后的第一个字符：

```
var regExp = /^The/im;
var str = "This is\nthe end";
```

相同规则同样适用于行结束符。例如下面的示例将不匹配：

```
var regExp = /end$/;
var str = "The end is near";
```

而下面的示例则是匹配的：

```
var regExp = /end$/;
var str = "The end";
```

如果使用了多行匹配选项，那么也可以匹配换行符前的字符串，例如：

```
var regExp = /The$/im;
var str = "This is really the\nend";
```

在正则表达式匹配中，括号的使用是非常重要的。括号匹配字符串，并记住匹配结果。匹配结果则保存在结果数组中：

```
var rgExp = /(^\D*[0-9])/
var str = "This is fun 01 stuff";
var resultArray = str.match(rgExp);
document.writeln(resultArray);
```

该代码中，数组输出两次“This is fun 0”，表示返回数组有两条记录。第一条记录是正则表达式的匹配；第二条记录是括号中的数值。如果括号不是引用整个正则表达式，而是正则表达式的一部分，如/(^\D*)[0-9]/，那么执行结果则是：

```
This is fun 0,This is fun
```

只有括号引用的所匹配的字符串会保存。

括号同样可以用于进行所匹配字符串的交换。RegExp 能够识别特殊字符，例如$1、$2，直到$9，这些特殊字符保存括号所匹配的字符串。示例 4.9 将查找用横杠分割的字符串，并通过$1 和$2 进行交换。

示例 4.9　通过正则表达式交换字符串

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>swap</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript">
//<![CDATA[

function swapWords() {

   var rgExp = /(\w*)-*(\w*)/
   var str = "Java---Script";
   var resultStrng = str.replace(rgExp,"$2-$1");
   alert(resultString);
}
//]]>
</script>
</head>
<body onload="swapWords()">
<p>Some page content</p>
</body>
</html>
```

这个 JavaScript 脚本的执行结果如下所示：

```
Script-Java
```

请注意，横杠的数量也被缩减到了一个。该示例还使用了另一个很常见的正则表达式匹配字符，\w。该字符表示任何数字或字母，包括下划线，等效于[A-Za-z0-9_]。该符号的相反形式是\W，表示任何非数字或字母。

最后一个正则表达式字符是竖线|和大括号{}。竖线表示可选择性的匹配。例如，下面的表达式可以匹配字母 a 或字母 b：

```
a|b
```

竖线还可以用于多个字符，例如：

```
a|b|c
```

大括号表示字符重复一定次数。下面的表达式可以匹配两个 s：

```
s{2}
```

正则表达式特别适用于 Ajax 程序的数据处理、窗体内容验证等场合，后续的章节将会针对这些场合进行更深入的介绍。

4.4 Date 对象

在 JavaScript 中，Date 对象可以用来创建日期实例。当创建了日期对象实例后，就可以使用不同的方法访问或修改日期，例如年、日、月等信息。

当创建日期对象时，如果没有传入任何参数，那么所创建的日期是当前客户端计算机的日期：

```
var dtNow = new Date();
```

如果在 Date 构造函数中传入参数，那么将创建出具有指定日期值的对象。例如，传入的参数可以是自 1970 年以来所经过的毫秒数：

```
var dtMilliseconds = new Date(5999000920);
```

上述代码的执行结果为：

```
Wed, 11 Mar 1970 10:23:20 GMT
```

你还可以通过日期格式正确的字符串来创建日期对象，例如：

```
var newDate = new Date("March 12, 1980 12:20:25");
```

日期格式可以只包含日期部分而不要时间部分，例如：

```
var newDate = new Date('March 12, 2008');
```

还可以传入多个表示日期的整数值，依次为年份、月份（0～11）、天数、小时、分钟、秒，甚至毫秒，例如：

```
var newDt = new Date(1977,11,23);
var newDt = new Date(1977,11,24,19,30,30,30);
```

Date 对象的实例方法包括访问日期各个部分的 getter 和 setter 方法。下列方法将根据当前时间获取日期对象中相应的数值：

- getFullYear　4 位数的年份；
- getHours　日期中的小时值；
- getMilliseconds　日期中的毫秒值；
- getMinutes　日期中的分钟值；
- getMonth　日期中的月份值；
- getSeconds　日期中的秒值；
- getDay　日期按周计算的天数，从 0（星期天）到 6（星期六）；
- getDate　日期按月份计算的天数。

下面是与格林威治时间相对应的方法：

- getUTCFullYear
- getUTCHours
- getUTCMilliseconds
- getUTCMinutes
- getUTCMonth
- getUTCSeconds
- getUTCDay
- getUTCDate

大多数 get 方法都有与之相对的 set 方法。例如，setYear 可以设置年份，setSeconds 可以设置秒，setUTCMonth 可以设置 UTC 的月份等。唯一没有相对应的 set 方法的是 getDay。此外，getYear 和 setYear 是不推荐使用的方法，因为未来版本可能不再支持这两个方法。

还有一个名为 getTimezoneOffset 的方法，它的返回值是当前时间与格林威治时间所相差的分钟数（以+或−表示）。例如我所在的位置是 UTC-5，所以调用该方法的结果是 300。

如果要将日期转成格式化字符串，有 6 个可用的方法：

- toString　根据当前时间格式显示日期的字符串；
- toGMTString　以 GMT 标准格式化日期；

- toLocaleDateString 和 toLocaleTimeString　根据所给定的时区显示日期的字符串；
- toLocaleString　根据当前时区显示日期字符串；
- toUTCString　根据 UTC 标准格式化日期。

Date 对象还有 3 个静态方法。Date.now 可以返回当前日期和时间；Date.parse 能返回自 1970 年以来所经历过的毫秒数；Date.UTC 则可以将给定的长格式日期转成该日期到现在所经历过的毫秒数，例如：

```
var numMs = Date.UTC(1977,11,24,30,30,30);
```

提示

Date.now 并不是标准的创建日期的方法。正确的方法应该是通过 new 构造函数，它将返回当前日期：

```
var rightNow = Date();
```

示例 4.10 中介绍了 Date 的不同方法，包括如何设置日期、如何显示日期等。

示例 4.10　Date 对象的 set 方法和格式化方法

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>Date</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript">
//<![CDATA[

function testingDate() {

   // 新日期
   var dtNow = new Date();

   // set day, month, year
   dtNow.setDate(18);
   dtNow.setMonth(10);
   dtNow.setYear(1954);
   dtNow.setHours(7);
   dtNow.setMinutes(2);

   // 输出已格式化的日期
   document.writeln(dtNow.toString() + "<br />");
   document.writeln(dtNow.toLocaleString() + "<br />");
   document.writeln(dtNow.toLocaleDateString() + "<br />");
   document.writeln(dtNow.toLocaleTimeString() + "<br />");
   document.writeln(dtNow.toGMTString() + "<br />");
   document.writeln(dtNow.toUTCString());
```

```
}
//]]>
</script>
</head>
<body onload="testingDate()">
<p>Some page content</p>
</body>
</html>
```

示例 4.10 在 Firefox 3.1 中的执行结果如下所示：

```
Thu Nov 18 1954 07:02:19 GMT-0600 (CST)
Thu Nov 18 07:02:19 1954
11/18/1954
07:02:19
Thu, 18 Nov 1954 13:02:19 GMT
Thu, 18 Nov 1954 13:02:19 GMT
```

Date 对象的选项太多，你可能会对如何选择特定的时区感到困惑。这里有一条合适的规则供参考：如果是网上商店，那么可以显示用户本地时区所对应的时间；如果是为因特网用户服务，那么可以设定格林威治时间，从而保证所有时间的一致性。

4.5 Math 对象

算术并不是数学的全部，至少在 JavaScript 中是这样；除了之前介绍的基本运算符以外，Math 对象还提供了许多与数学相关的方法和属性，例如取 10 的对数的 Ln10 方法，或者取 x 的自然对数的 log(x)方法。Math 对象中并不包括如加法、减法之类的简单运算。

与其他 JavaScript 对象不同，Math 对象的所有属性和方法都是静态的。这表示访问 Math 对象时并不需要创建 Math 对象的实例；可以直接通过 Math 对象访问相关的方法和属性，例如：

```
var newValue = Math.SQRT1;
```

4.5.1 Math 的属性

Math 对象中提供了一些常量属性，例如众所周知的 pi、2 的平方根、10 的自然对数等。下面列出了 Math 的所有属性：

- E　表示常数 e 的值，它是自然对数的基础；
- LN10　10 的自然对数；
- LN2　2 的自然对数；
- LOG2E　以 2 为底的，常数 e（自然对数的底）的对数；
- LOG10E　以 10 为底的，常数 e（自然对数的底）的对数；

- PI　圆周率值；
- SQRT1_2　1/2 的平方根；
- SQRT2　2 的平方根。

数学运算在编程语言中的执行结果取决于其底层架构，以及浏览器中 JavaScript 引擎对数学函数的实现，甚至涉及操作系统、计算机硬件等。所以，执行三角函数的结果可能存在细微的差异，但是一般不会对计算结果产生太大的影响。

4.5.2 Math 的方法

Math 对象的方法相当简单。无论变量是什么类型，传入 Math 函数的参数都会先转换为数字类型，开发人员无须自己进行转换。

abs 函数的参数是数字值，并返回该数字的绝对值。如果传入的数字是负数，那么就会返回正数。在下面的代码中变量 pVal 的数值会被设置成 3.4。

```
var nVal = -3.45;
var pVal = Math.abs(nVal);
```

此外还有一些三角函数，如 sin、cos、tan、acos、asin、atan 和 atan2。这些函数分别表示正弦、余弦、正切、反余弦、反正弦、反正切，用来计算 x 点和圆心之间的角度。每个函数都有一个特定类型的数字型参数，并返回相应的数值。

- Math.sin(x)　特定的角度，单位为弧度；
- Math.cos(x)　特定的角度，单位为弧度；
- Math.tan(x)　特定的角度，单位为弧度；
- Math.acos(x)　−1 到 1 之间的数字；
- Math.asin(x)　−1 到 1 之间的数字；
- Math.atan(x)　任意数字；
- Math.atan2(py,px)　某一点的 x 和 y 坐标。

Math.ceil 方法对数字执行向上取整。下面的代码将对变量 pVal 执行取整操作，结果为 4：

```
var nVal = 3.45;
var pVal = Math.ceil(nVal);
```

在下面的代码中，pVal 的值将是−3：

```
var nVal = -3.45;
var pVal = Math.ceil(nVal);
```

Math.floor 方法则是用来对数字执行向下取整的。下面的代码将使 pVal 的值变为 3：

```
var nVal = 3.45;
var pVal = Math.floor(nVal);
```

下面的代码将使 pVal 的值变成-4：

```
var nVal = -3.45;
var pVal = Math.floor(nVal);
```

Math.round 方法对数字取整，得到和它最接近的整数；实际执行向上还是向下取整取决于该数值本身（采取的原则是四舍五入）。例如，3.45 会取整为 3，而 3.85 则取整为 4。中间数字 3.5 则会取整为 4。结果是最接近的整数，而不论是正数或负数。

Math.exp(x)则用来计算 e 的自然对数，结果取决于方法的参数：

```
var nVal = Math.exp(4)      //等于 e^4
```

Math.pow 将执行幂方操作，例如：

```
var nVal = Math.pow(3,2)    // 3^2 也就是 9
```

Math.min 和 Math.max 用来比较多个数字，返回其中的最小值或最大值：

```
var nVal = 1.45;
var nVal2 = 4.5;
var nVal3 = -3.33;
var nResult = Math.min(nVal, nVal2, nVal3)     // 结果为-3.33
var nResult2 = Math.max(nVal, nVal2, nVal3)    // 结果为 4.5
```

最后一个方法 Math.random 是用来生成 0～1 之间的随机数字的（包含 0，但不包含 1）：

```
var nValue = Math.random();
```

将该数值乘以 10 或 100，可以得到大于 1 的随机数。然而，随机数的生成并不能限定范围，不过可以自己模拟实现，如示例 4.11 所示。

示例 4.11　生成随机数

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>Random Quote</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript">
  //<![CDATA[

function getQuote() {

   var quoteArray = new Array(5);
   quoteArray[0] = "Quote one";
   quoteArray[1] = "Quote two";
   quoteArray[2] = "Quote three";
   quoteArray[3] = "Quote four";
   quoteArray[4] = "Quote five";
```

```
    iValue = Math.random();          // 0~1之间的随机数
    iValue *= 5;                     // 乘以5
    iValue = Math.floor(iValue);     // 取整
    alert(quoteArray[iValue]);
}

//]]>
</script>
</head>
<body onload="getQuote();">
  <p>some content</p>
</body>
</html>
```

在该示例中，我们创建了一个包含 5 个元素的数组。当页面载入时就会调用该函数，并根据 Number 和 Math 函数生成 0～4 之间的随机数，然后再通过该数字访问数组元素，并将结果输出到页面上。

4.6 JavaScript 数组

与 Math 或 String 类似，JavaScript 数组也是一个对象，你可以通过构造函数来创建一个数组，例如：

```
var newArray = new Array('one','two');
```

基本类型也可以创建数组，这时就不需要显式地调用 Array 对象：

```
var newArray = ['one','two'];
```

与 String、Number 对象不同的是，JavaScript 会立即将这样的基本类型转换为 Array 对象，并将结果赋给变量。而 String、Number 或者 Boolean 对象只在调用对象方法时才进行转换。

创建数组对象实例后，就可以通过下标访问数组元素，下标就是数组元素在数组中的位置：

```
alert(newArray[0]);
```

数组下标从 0 开始，一直到“元素总数-1”为止。所以对于拥有 5 个元素的数组而言，下标范围就是 0～4。

数组不一定是一维的，虽然多维数组并不常见。在 JavaScript 中管理多维数组的方法是为每个数组元素创建一个新的数组。以下代码将创建一个三维数组：

```
var threedPoints = new Array();
threedPoints[0] = new Array(1.2,3.33,2.0);
threedPoints[1] = new Array(5.3,5.5,5.5);
threedPoints[2] = new Array(6.4,2.2,1.9);
```

如果最内层的数组下标分别是 x、y、z，那么可以通过如下所示的代码访问 z 坐标，例如：

```
var newZPoint = threedPoints[2][2];    // 数组坐标从 0 开始
```

然后添加数组维数，也就是继续在元素中创建新数组：

```
threedPoints[2][2] = new Array(4.4,4.6,44)
var newthreedZPoint = threedPoints[2][2][1];
```

JavaScript 语言中不需要提前知道数组中元素的个数。如上述示例所示，可以根据固定元素个数创建数组，也可以任意添加新的元素。添加新元素就会改变数组的大小。

```
var testArray = new Array();
testArray[99] = 'some value';          // testArray 现在有 100 个元素
```

通过 Array 对象的属性 length 可以得知数组的长度（元素的个数）：

```
alert(testArray.length);               // 输出 100
```

如果访问多维数组的长度，那么只能获取特定维度的元素个数，例如：

```
alert(threedPoints[2][2].length);      //结果为 3
alert(threedPoints[2].length);         //结果为 3
alert(threedPoints.length);            //结果为 3
```

除了 length 外，数组对象还提供了其他的一些属性和方法，例如 splice，可以在数组中插入或移除元素。在下面所示的代码中，将从坐标 2 开始插入两个元素并移除两个元素：

```
var fruitArray = new Array('apple','peach','orange','lemon','lime','cherry');
var removed = fruitArray.splice(2,2,'melon','banana');
document.writeln(removed + "<br />");
document.writeln(fruitArray);
```

其执行结果如下所示：

```
orange,lemon
apple,peach,melon,banana,lime,cherry
```

splice 方法返回的结果是数组中被移除的元素。

数组的 slice 方法可以分割数组，并返回结果：

```
var newFruit = fruitArray.slice(2,4);   // 返回 3 个元素的数组: melon, banana,
                                           and lime
```

concat 方法可以连接不同的数组，返回的数组包含下列元素：apple、peach、melon、banana、lime、cherry、orange 和 lemon。

```
var newFruit = fruitArray.concat(removed);
```

concat 和 slice 都不会改变原有的数组，而只是创建一个新的数组以作为方法的返回值。

之前的示例曾经直接输出数组的内容。JavaScript 引擎会将数组转换为字符串，以默认分隔符（逗号）隔开每个数组元素。如果希望使用不同的分隔符，那么可以使用 join 方法来生成新的字符串，并传入所希望的分隔符，例如空格：

```
var string = fruitArray.join(" ")
```

通过 reverse 方法可以对数组执行倒序排列，例如：

```
fruitArray.reverse();
```

在大多数情况下，数组元素的顺序并不重要。不过也有些需要维护数组元素顺序的场景，例如队列。接下来我们就来看看一些将数组用于队列或列表的方法。

4.6.1 FIFO 队列

数组可以用来保存一系列元素，并以先进先出 FIFO 的方式访问。Array 对象有 4 个用于维护队列和列表的方法：push、pop、shift 和 unshift。

push 方法能将元素添加到数组最后面，而 unshift 方法则是将元素添加到数组的最前面。返回值都是数组新的长度值。

pop 方法用来移除数组的最后一个元素，而 shift 则是移除第一个元素。返回值都是从数组中所移除的元素。

这 4 个方法都会导致数组的变化，即添加或移除数组元素。示例 4.12 将介绍如何在 JavaScript 中管理 FIFO 队列。

示例 4.12 基于 Array 方法的 FIFO 队列

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>FIFO</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript">
    //<![CDATA[

function pushPop() {

   // 创建 FIFO 队列，并通过 push 方法添加元素
   var fifoArray = new Array();
   fifoArray.push("Apple");
   fifoArray.push("Banana");

   var ln = fifoArray.push("Cherry");

   // 输出数组长度和数组内容
   document.writeln("length is " + ln + " and array is " + fifoArray + "<br />");
```

```
    // 使用 shift 移除元素
    for (var i = 0; i < ln; i++) {
         document.writeln(fifoArray.shift() + "<br />");
    }

    // 输出数组长度
    document.writeln("length now is " + fifoArray.length + "<br /><br />");

    // 使用 unshift 方法添加元素
    var fifoNewArray = new Array();

    fifoNewArray.unshift("Learning");
    fifoNewArray.unshift("Java");
    ln = fifoNewArray.unshift("Script");

    document.writeln("length is " + ln + " and array is " + fifoNewArray + "<br/>");

    // 使用 pop 方法移除元素
    for (var i = 0; i < ln; i++) {
      document.writeln(fifoNewArray.pop() + "<br />");
    }
    document.writeln("new length is " + fifoNewArray.length );
}

//]]>
</script>
</head>
<body onload="pushPop();">
</body>
</html>
```

在该示例中，第一个值得注意的地方是在此混合使用了 shift、push、unshift 和 pop 操作。之所以这样做，是为了展示这些方法的作用。push 方法将向数组最末尾处添加元素，而每次添加新元素时，第一个元素将一直在数组最前面。而 pop 方法则是从数组最末尾处移除元素，从而创建了后进先出列表（LIFO），也就是一个合理的堆栈。然而这不是该示例的本意，该示例希望最早添加的元素也是最早获取的元素。那么从数组中移除最前面元素的 shift 方法恰恰符合先进先出的本意。

该规则同样适用于 unshift 和 pop 方法。unshift 方法往数组最前面添加新元素，每个新元素都将之前的元素推到后面，而 pop 方法从数组最末尾处移除元素。这一对组合同样维护了元素的顺序，这也正是本示例的初衷，不关心数组元素的顺序，而是关心元素添加的顺序。

该示例的执行结果（除了 IE 外，IE 的 unshift 方法不会返回数组长度）如下所示：

```
length is 3 and array is Apple,Banana,Cherry
Apple
```

```
Banana
Cherry
length now is 0

length is 3 and array is Script,Java,Learning
Learning
Java
Script
new length is 0
```

示例 4.12 中还介绍了如何通过循环语句来遍历数组。该示例并没有一次次地调用 shift 或 pop 方法，而是通过循环遍历数组并逐一调用。这只是一个简单的示例，但如果是较大的数组，那么可以想象这样的操作能够节省多少时间。

一般来说，通过 for 循环遍历数组，变量 i 将随着循环递增，并作为数组的下标：

```
for (var i = 0; i < someArray.length; i++) {
     alert(someArray[i]);
}
```

不过也不一定需要使用该下标，甚至可以通过递减的方式遍历数组：

```
for (var i = someArray.length-1; i >=  0; i--) ...
```

另一种遍历数组元素的方式是使用 for…in 循环。

```
var programLanguages = new Array
('C++','Pascal','FORTRAN','BASIC','C#','Java','Perl','JavaScript');
for (var itemIndex in programLanguages) {
     document.writeln(programLanguages[itemIndex] + "<br />");
}
```

数组的其他方法与函数回调有关，本书将在后续的章节中进行介绍。

4.7 知识测验

1. 以逗号分隔的字符串是常见的数据格式。如何根据逗号分隔的字符串创建数组呢？请为下面的字符串创建一个数组，并访问第三个元素：

   ```
   "cats,dogs,birds,horses"
   ```

2. 特殊字符\b 是表示分隔单词的符号，而\B 则是表示非分隔单词的符号。请定义一个正则表达式，在下面的字符串中查找单词 fun，并将其替换成 power：

   ```
   "The fun of functions is that they are functional."
   ```

3. 编写一段程序代码，获取今天的日期并计算下一周同一天的日期。

4. 编写一段程序代码，分别对数字 34.44 执行向上和向下取整操作。

5. 根据下面的字符串，利用正则表达式将所有标点符号替换成逗号，然后将其转换

成数组，并输出每个数值：

```
var str = "apple.orange-strawberry,lemon-.lime";
```

4.8 测验答案

1. 使用 String.split 方法，传入逗号作为分割符，例如：

```
var animalString = "cats,dogs,birds,horses";
var animalArray = String.split(animalString,",");
alert(animalArray[2]); // alert box displays birds
```

2. 其解决方案为：

```
var funPattern = /\bfun\b/;
var strToSearch = "The fun of functions is that they are functional";
var afterMatch = strToSearch.replace(funPattern,"power");
```

3. Date 对象中没有能够操作周的函数，但是我们知道一周是 7 天，一天是 24 小时，也就是一周是 168 小时。使用 getHours 方法获取当前日期的小时数，在此基础上增加 168，然后重新赋给 Data 对象的小时值，并打印日期即可：

```
var dtNow = new Date();
var hours = dtNow.getHours();
hours+=168;
dtNow.setHours(hours);
document.writeln(dtNow.toString());
```

4. Math.floor 可以对数字执行向下取整，而 Math.ceil 可以对数字执行向上取整。其解决方案为：

```
var baseNum = 34.44;
var numFloor = Math.floor(baseNum);   // 返回 34
var numCeil = Math.ceil(baseNum);     // 返回 35
```

5. 其解决方案为：

```
var strToAlter = "apple.orange-strawberry,lemon-.lime";
var puncPattern = /[\.|-]/g;
var afterMatch = strToAlter.replace(puncPattern,",");
var fruits = afterMatch.split(',');
for (var i = 0; i < fruits.length; i++)
      document.writeln(fruits[i]);
```

第5章

函数

函数是 JavaScript 语言中的关键部分，但和你的直觉可能有些差别。它看上去应该可以归入语句的范畴，不过它实际上却和在之前章节中介绍的其他对象相似。你可以定义一个函数，创建一个新的函数，甚至输出一个函数。

正是因为有了这个功能，所以你可以将一个函数赋给一个变量或数组元素，甚至可以将其作为参数传给另一个函数调用。这使得函数成为了一个十分灵活的、有用的对象，但它也最容易使 JavaScript 编程新手产生理解上的混淆。

在 JavaScript 中有 3 种创建函数的方法：声明式的/静态的、动态的/匿名的、字面量式的。在使用它们之前，理解各种方法的效果是十分重要的。

5.1 声明式的函数

使用声明式/静态形式定义函数是最常见的。采用这种方法定义，首先需要一个 function 关键字，接着是函数名称、放在圆括号内的零个或多个参数，然后就是函数体：

```
function functionname (param1, param2, ..., paramn) {
   function statements
}
```

你必须将函数内的语句放在一对花括号中，即使函数体中只有一条语句。

除非要创建一个带有新对象的新程序库，或者基于事件动态定义一个函数，否则这种语法是我最常用的，在本书中，在迄今为止的所有示例里使用的也是这种语法。

当页面载入之后，就会对这个声明式的函数进行解析，在每次调用该函数时将复用其解析结果。要从代码中找到函数定义是很容易的，它易于阅读、理解，但它也有一些负面效应，如内存泄漏。这些问题对于使用其他编程语言的开发人员也是经常遇到的。

下面所示的代码片段中，就使用了这种格式创建了一个函数，并且在声明之后就马上调用了这个函数：

```
function sayHi(toWhom) {
   alert("Hi " + toWhom);
}
sayHi("World!");
```

在前面的这段代码中，调用这个函数将弹出一个 alert 对话框，上面将显示“Hi World!”，除非 JavaScript 代码出错，否则不管你向这个函数传入了什么字符串，或者调用这个函数多少次，都将使用相同的函数对象，其结果也是相同的：弹出一个显示指定消息的 alert 对话框。

5.1.1 函数的命名规范和大小

函数都是用来执行特定操作的。因此你会希望在函数名中加入一个动词，尽可能概括出该函数所执行的活动。以下都是一些很不错的函数名：

- runQuote
- printDate
- processName
- addNumbers

函数名称通常以动词开头，然后跟上一个或多个名词，每个名词的首字符都是大写字母。在 JavaScript 中并不是必须采用这样的命名规范，也不是所有开发人员都喜欢这样的方法。不过，这样命名更易于阅读，并且能够起到一些自说明的效果。

如果在一个函数中需要执行多个任务，那么可能很难对函数进行命名，可以考虑将这个函数拆分成几个更小的单元，这样做也能够提高可复用性。与其在一个函数中完成表单值解析、表单域验证、使用该值执行一个 Ajax 服务器端调用等所有任务，还不如将验证电子邮件或邮政编码的功能抽出来，定义成一个专门的函数，这样还能够在其他函数中复用它。这样做的好处是你可以将验证的函数打包到一个单独的文件中，以便在多个 JavaScript 应用程序中使用它。

实际上，尽可能让函数短小、使其特定于某个任务、尽量保持通用是应该遵循的规则。

5.1.2 函数返回值和参数

函数与调用它的程序之间的通信是通过传入函数的参数以及函数返回的值完成的。

参数中的变量传给函数的实际上是原始值，如一个字符串、一个布尔值、一个数字等。这也就意味着如果你在函数中修改实际参数值，那么它是不会影响调用程序的。

另一方面，对于传给函数的对象而言传递的则是一个引用。在函数中对这个对象的修

改将会反映在调用程序中。

示例 5.1 向一个函数传递了两个参数：一个是字符串型变量，另一个是数组。函数中的代码将对这些参数进行修改，然后在调用程序中输出其内容。字符串变量的值没有变化，因为它传入的只是原始值，而数组对象的第二个成员的值就被修改了，而且还增加了第三个成员。

示例 5.1　函数参数，传值和传引用

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>Pass Me</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript">
//<![CDATA[

function alterArgs(strLiteral, aryObject) {

   // 覆盖原来的字符串
   strLiteral = "Override";
   aryObject[1] = "2";
   aryObject[aryObject.length] = "three";
}
function testParams() {
   var str = "Original Literal";
   var ary = new Array("one","two");

   document.writeln("string literal is " + str + "<br />");
   document.writeln("Array object is " + ary + "<br /><br />");

   alterArgs(str,ary);

   document.writeln("string literal is " + str + "<br /> ");
   document.writeln("Array object is " + ary);
}

//]]>
</script>
</head>
<body onload="testParams();">
<p>Some page content</p>
</body>
</html>
```

图 5.1 展示了示例 5.1 在 Firefox 浏览器中载入的效果。前两行是在函数调用之前字符串和数组对象的值，后两行是函数调用之后它们的值。

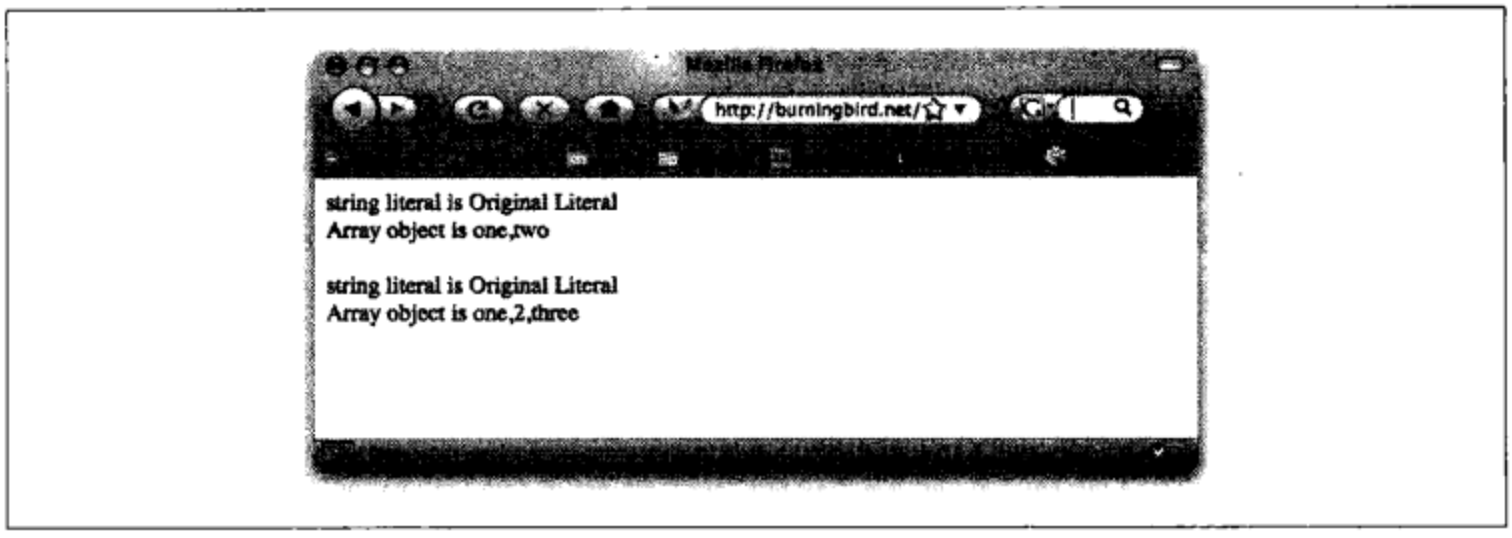


图 5.1　示例 5.1 应用程序在 Firefox 中的运行结果

一个函数可以返回一个值，也可以不返回。如果它返回了一个值，那么可以将 return 语句放在函数代码的任何位置上，甚至可以有多个 return 语句。如果 JavaScript 应用程序在运行时遇到一个 return 语句，那么就会停止函数代码的执行，并且将控制权返回调用该函数的语句。

想使用多个 return 语句的原因通常是需要在某个条件满足时停止并退出该函数。在下面这个代码片段中，如果条件不满足，那么该函数将立即停止运行；否则将继续执行：

```
function testValues(numValue) {
  if (isNaN(numValue)) {
       return "error -- not a number";
  }
  ...
  return ...
}
```

函数并不是必须有返回值的，虽然返回值对于错误处理而言是十分有用的，例如当函数未能成功执行时将返回一个 false 值（第 13 章将介绍一些更有用的错误处理方法）。

与声明式的函数相反的就是在接下来的小节中要介绍的动态/匿名函数。

5.2　匿名函数

函数就是一个对象。因此，你可以像创建字符串、数组那样通过一个构造器创建它们，并将该函数赋给一个变量。在下面的代码中，将通过函数构造器创建一个新的函数，其函数体将作为参数传给这个函数构造器：

```
var sayHi = new Function("toWhom","alert('Hi ' + toWhom);");
sayHi("World!");
```

这种类型的函数通常被称为匿名函数，因为这个函数本身并不是直接声明的，也没有对其进行命名。

当 JavaScript 解析它时，和声明式函数不一样，它将动态创建一个匿名函数；当其被调

用后，该函数就将被自动删除。如果该函数在一个循环语句中使用，那么将意味着每次循环将创建一次函数，而声明式/静态函数只会被创建一次。因此，你可能会认为匿名函数并不是很有用。不过，动态函数对于定义一个在运行时才能确定需求的函数而言是一个很好的方法。

以下是使用函数构造器创建一个匿名函数的语法格式：

```
var variable = new Function ("param1", "param2", ..., "paramn", "function body");
```

第一个参数是在定义声明式函数时指定的第一个参数。最后一个参数是函数体。整个函数将被赋给一个变量：

```
var func = new Function("x", "y", "return x * y")
```

它的效果与下面这个声明式/静态函数是等价的：

```
function func (x, y) {
   return x * y;
}
```

示例 5.2 将匿名函数的动态性发挥到了极致。用户将通过 alert 对话框来设置定义函数所需的函数体以及两个参数。然后调用该函数，将其生成的结果输出在页面上。

示例 5.2　一个动态/匿名函数

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>Build a Function</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript">
//<![CDATA[

function buildFunction() {

   // 提示输入函数体和参数
   var func= prompt("Enter function body:");
   var x = prompt("Enter value of x:");
   var y = prompt("Enter value of y:");

   // 调用这个匿名函数
   var op = new Function("x", "y", func);
   var theAnswer = op(x, y);

   // 输出函数执行结果
   alert("Function is: " + func);
   alert( "x is: " + x +
                        " y is: " + y);
   alert("The answer is: " + theAnswer);
}
```

```
//]]>
</script>
</head>
<body onload="buildFunction();">
<p>Some content</p>
</body>
</html>
```

由于 JavaScript 不是一个强类型语言，因此该函数能够接受数字型参数：

```
Function is: return x * y
x is: 33  y is: 11
The answer is: 363
```

也可以接受字符串型参数：

```
Function is: return x + y
x is: This is  y is: the string
The answer is: This is the string
```

唯一的要求是输入的操作符必须对于输入的数据类型有意义。即使没意义，也不会遇到 JavaScript 错误，因为浏览器看不到这样的错误；它将发生在运行时。你最终可能会看到类似于以下的输出：

```
Function is: return x * y
x is: this is  y is: the answers
The answer is: NaN
```

另外，你还可能会得到一个预期外的结果。如果你输入两个数据，并且定义的操作符是“+”号，那么将得到如下所示的结果（而不是两数相加的结果），因为 JavaScript 将根据上下文将其视为字符串，所以将执行字符串连接操作：

```
Function is: return x + y;
x is: 2  y is: 3
The answer is: 23
```

要想确保得到预期的结果，那么需要进行显式的类型转换：

```
Function is: return parseInt(x) + parseInt(y);
x is: 2  y is: 3
The answer is: 5
```

毫无疑问，在使用匿名函数时必须小心谨慎。我不建议你让 Web 页面访问者定义一个用于页面的函数。不过，用动态函数对用户输入进行处理是一种有趣的方法，只要你消除用户输入的可能带来问题的东西即可，例如嵌入的链接、混乱的 cookie、对服务端功能的调用、创建新函数等。

警告

如果在 IE 8.0 中运行示例 5.2 时遇到问题，那么你需要为 Windows 命令行设置相应的安全参数。

此外，还有一种混合型的函数创建方法，它将静态的声明式函数和动态的匿名函数整合在一起，它就是将在下一小节中介绍的函数字面量。

5.3 函数字面量

在介绍下一种（可能会带来混淆的）函数类型之前，简单复习一下对象和字面量可能是很有帮助的。正如我们在前面的章节中介绍的那样，JavaScript 对象可以以字面量形式定义。与其使用构造器和对象，不如直接使用表达式。你可以使用 String 构造器创建一个字符串，然后使用 String 的方法访问它：

```
var str = new String("Learning Java");
document.writeln(str.replace(/Java/,"JavaScript"));
```

你也可以使用原生的字符串类型定义，这时仍然可以使用 String 对象的方法，当调用 String 方法时，JavaScript 引擎会暗中将字面量转成一个对象：

```
var str2 = "Learning Java";
document.writeln(str2.replace(/Java/,"JavaScript"));
```

实际上，你甚至可以不定义变量：

```
document.writeln("Learning Java".replace(/Java/,"JavaScript"));
```

对字符串可以这样做，对于函数也不例外，也就是说当想创建一个函数时也可以不使用函数构造器，同样也可以将其赋给一个变量；这样的字面量就成了函数字面量：

```
var func = function (params) {
     statements;
}
```

函数字面量也被称为函数表达式，因为这样创建的函数将成为表达式的一部分，而不是一个特有类型的语句。它们像匿名函数一样也没有定义函数名称。不过，它和匿名函数之间也是有差异的，函数字面量只会被解析一次。实际上，除了该函数将赋给一个变量之外，函数字面量和声明式函数是类似的：

```
var func = function (x, y) {
     return x * y;
}
alert(func(3,3));
```

当你想实现如把一个函数作为另一个函数的参数之类的扩展时，函数字面量的特色就会显示出来。示例 5.3 中展示了这个有趣的特性。该示例中定义了一个名为 funcObject 的函数，它有 3 个参数，第三个参数就是一个用来处理前两个参数的函数。

示例 5.3　将一个函数作为参数传给另一个函数

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
```

```
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>Function Literal</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript">
//<![CDATA[

// 将第三个参数作为函数来调用
function funcObject(x,y,z) {
   alert(z(x,y));
}

function testFunction() {

// 第三个参数是一个函数
funcObject(3,4,function(x,y) { return x * y})
}

//]]>
</script>
</head>
<body onload="testFunction();">
<p>some content</p>
</body>
</html>
```

这个应用程序的执行结果是弹出一个显示“12”的 alert 对话框。下面是调用该函数的另一个例子：

```
funcObject(4,2,function(x,y) { return x - y})
```

该函数调用的结果是参数 x 与参数 y 的差，因此将弹出一个显示“2”的 alert 对话框。

函数字面量的第二种形式是非匿名形式，你可以指定带有名称的函数：

```
var func = function multiply(x,y) {
    return x * y;
}
```

不过，这个函数名称只能够在函数内部调用。因此它通常是没用的，除非你要定义一个递归函数。

5.3.1 函数和递归

一个将调用自身的函数称为递归函数。通常，当你需要多次执行一个过程，并且每次处理都是基于前一次的处理结果时，就应该考虑使用递归函数。在 JavaScript 中递归函数并不常用，不过当你要处理的是如 DOM 之类的 tree-line 结构数据时，递归函数就很有用。不过，它占用的内存和资源都比较多，同时也难以实现和维护。因此，应该保守地使用递归，使用时也要进行彻底的测试。

在前一小节中，我写了一个命名的函数字面量，这个带名称的函数只有函数内部的代码才能够通过其名称调用它。这是专门为递归函数设计的。

示例 5.4 使用了一个递归函数来遍历一个数字型数组，累加数组中的数字，然后将其添加到一个字符串上。

示例 5.4　JavaScript 递归函数

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>Recursion</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript">
//<![CDATA[

function runRecursion() {

  var addNumbers = function sumNumbers(numArray,indexVal,resultArray) {

     // 递归结束检查
     if (indexVal == numArray.length)
         return resultArray;

     // 完成数值累加
     resultArray[0] += Number(numArray[indexVal]);

     // 完成字符串连接
     if (resultArray[1].length > 0) {
         resultArray[1] += " and ";
     }
     resultArray[1] += numArray[indexVal].toString();

     // 令索引值自增 1
     indexVal++;

     // 再次返回该函数，并返回执行结果
     return sumNumbers(numArray,indexVal,resultArray);
}

// 创建数值型数组和结果数组
var numArray = ['1','35.4','-14','44','0.5'];
var resultArray = new Array(0,''); // necessary for the initial case

// 调用该函数
var result = addNumbers(numArray,0, resultArray);

// 输出结果
```

```
    document.writeln(result[0] + "<br />");
    document.writeln(result[1]);
}
//]]>
</script>
</head>
<body onload="runRecursion();">
<p>some content</p>
</body>
</html>
```

示例 5.4 的执行结果是在页面中输出以下内容：

```
66.9
1 and 35.4 and -14 and 44 and 0.5
```

在这个 JavaScript 应用程序中，函数字面量 sumNumbers 首先将检查作为参数传入的索引值是否与这个数字型数组的长度（它也是传入的参数之一）相等。如果不相等，函数将把下一个成员数值累加到总计数中，然后将这个数值添加到一个字符串，它们都是作为参数传入的双元素数组的成员之一。

当索引值与这个数字型数组的长度相等时，该函数将返回 resultArray。由于这个函数是递归调用的，因此每次传入都是上一次执行 sumNumbers 函数所返回的结果，直到最后一次执行为止，这时结果值将返回给第一次调用它的应用程序。

当然，对于这个示例而言，你也可以使用一个循环语句实现相同的结果。不过，正如我们之前所说的，当你处理 DOM 之类的树形结构数据时，递归是相当有价值的，而这时就需要使用函数字面量。不过，使用函数字面量并非在所有浏览器中都不会导致潜在的负面效应，其中一个风险就是嵌套的函数可能会因为闭包问题而导致内存泄漏。

5.3.2 嵌套的函数、函数闭包与内存泄漏

与 JavaScript 函数字面量相关的问题中还有一个重要的方面，那就是用于嵌套的函数。请考虑以下代码：

```
function outer (args) {
   function inner (args) {
      inner statements;
   }
}
```

对于一个嵌套的函数，内部函数将在外部函数内执行，包括对外部函数变量和参数的访问。而外部函数并不应访问内部函数的变量，也不应调用将访问内部函数的应用程序（是的，它不仅是以函数字面量形式定义的，它还将向调用它的应用程序返回值，这将会添加其自身的复杂度）。

示例 5.5 中创建了一个嵌套的、内部的函数字面量，它将向调用它的应用程序返回值（在

示例中以粗体字显示)。这个内部的函数将访问外部函数的参数和变量。它也有自己的参数，而外部函数不会访问它。

当内部函数通过外部函数传给应用程序时将直接调用它，它将连接当时作为参数传给外部函数的字符串，然后将这个字符串作为参数传给内部函数，并返回结果。将内部函数的参数改成这个字符串之后，同时还将用不同的参数调用外部函数，这时将再调用一次内部函数。

示例 5.5　嵌套函数和闭包

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>Nested</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript">
//<![CDATA[

// 外部函数
function outerFunc(base) {

   var punc = "!";

   // 返回内部函数
   return function (ext) {
      return base + ext + punc;
   }

}

function processNested() {

   // 创建对内部函数的访问
   var baseString = outerFunc("Hello ");

   // 内部函数仍然将访问外部函数的参数
   var newString = baseString("World!");
   alert(newString);

   // 再次调用
   var notherString = baseString("Reader!");
   alert(notherString);

   // 创建另一个内部函数实例
   var anotherBase = outerFunc("Hiya, Hey ");

   // 另一个本地字符串
   var lastString = anotherBase("you!");
   alert(lastString);
```

```
}
//]]>
</script>
</head>
<body onload="processNested();">
<p>Some content</p>
</body>
</html>
```

运行该应用程序后，将连续不断地弹出 3 个 alert 对话框，分别显示“Hello World!”、“Hello Reader!”和“Hiya, Hey you!”。

它的工作机制是怎么样的呢？它是否违反了范围规则？当函数中止时将处于什么状态？局部变量所占的内存是否能够通过自动垃圾回收机制释放吗？

答案是不确定。

每当 JavaScript 应用程序创建一个新的作用范围时，如果需要的话将创建相关的 scoping bubble（作用范围冒泡），以封装这个作用范围。这将应用在函数中，它将在自己的作用范围内执行。

通常，当函数结束时该作用范围也就会释放，因为它已经不再需要了。但是，当一个内部函数向外部应用程序返回一个值，并赋给一个外部变量时，内部函数的作用范围就将附加到外部函数上，然后再附加到调用它们的应用程序中，这样才能保证内部函数和外部函数参数和变量的完整性。返回一个在其他函数中以内部对象形式创建的函数字面量，然后将其赋值给调用它的应用程序中的一个变量，这就是 JavaScript 中的闭包。它将引入一个作用范围链的概念，它是确保应用程序在此时能够正常工作所需的数据。

这在 JavaScript 程序库中是一个很重要的概念，本书后面的内容将对闭包做更进一步的解释；不过这里引入了一个与闭包相关的问题，那就是你创建这个闭包是很偶然的。在示例 5.5 中，如果针对每个字符串创建一个新的内部函数引用，而不是复用内部函数中的变量引用，那么将在整个过程中引入很多对象实例。

当创建循环引用时，也会意外创建一个闭包，例如以下这段源于 Mozilla 文档网站的代码：

```
function leakMemory() {
    var el = document.getElementById('el');
    var newObj = { 'el': el };
    el.newObj = newObj;
}
```

在第 11 章中，我们将更深入地介绍 DOM，并且在本书的第二部分中也将更大量地使用 getElementById 方法，不过对于这个示例而言，我们只是访问一个标识符为 el 的 DOM 元素。然后使用该元素创建一个新的对象引用（newObj），它所使用的格式我们会在本书后面的内容中介绍。这个新的函数字面量将创建一个未命名的对象，并将其赋给查

找到的 DOM 对象中名为 el 的属性。

然后就有些突然了：我们将把 newObj 赋给最初从 DOM 中查找到的元素（在 JavaScript 中很常见的行为）的一个新增加的属性，从字面上看，这意味着我们将该对象本身当做一个属性赋给该对象。我并不鼓励这样做，不过绝大多数浏览器都能够有效管理闭包，能够中止它、回收内存，除了较早版本的 IE 之外。

针对 DOM 对象，老版本（但仍然很流行）的 IE（6.x 和 7.x）提供了自己的内存管理体系，除了针对 JavaScript 对象的内存管理之外。在这种情况下，如果是因为循环引用带来的意外闭包，并且这种引用是在 JavaScript 和 DOM 对象之间的，那么已分配的内存将永远不会释放，甚至在页面关闭时也不会释放。实际上，只有当浏览器关闭时才会释放。

这样的内存泄漏所产生的后果通常不严重，除非你将它放在一个循环语句中，那么这时内存消耗可能会很快。由于存在内存泄漏问题，而且老版本的 IE 又十分流行，因此在使用这里展示的闭包时应该很小心。所幸的是，这样的内存泄漏问题在 IE 8.0 中已经解决了。

提示

Jim Ley 在他的一篇文章（http://jibbering.com/faq/faq_notes/closures.html）中对闭包做了很精彩的概述。

5.3.3 回调函数

在第 4 章中的“JavaScript 数组”小节中，我编写了一些依赖函数的方法，它们根据一些事件自动调用。数组的方法包括 filter、forEach、every、map、some 以及用函数字面量定义的函数，不过当以这种形式使用时，它们通常被称为回调函数。

警告

数组对象的回调函数不是标准的 ECMAScript。虽然本小节中的示例能够在 Safari、Opera 和 Firefox 中正常运行，但却无法在 IE 中正常运行，包括 IE 8.0。

我们回顾一下数组的方法，filter 方法用来确保添加到数组中的元素能够满足特定的标准。不管你应用了什么样的 filter，都可以将任何东西扔到数组中，filter 将帮你完成一切工作，而无须逐一检查这些值然后分别将其添加到数组中。

forEach 方法的参数是一个函数，这个函数将逐一处理每个元素。和 filter 不同，这个函数不会对数组产生影响。map 方法将对数组中的所有元素执行回调函数，并根据其执行结果创建一个新数组；它和 forEach 十分类似，因为都将针对每个数组元素执行回调函数。不过，用 forEach 主要是针对数组中每个元素执行一个函数；而使用 map 则是根据针对数组中每个元素执行一个函数的结果生成一个新数组。

every 方法将针对数组中的每个元素执行回调函数，直到其中一个元素返回 false 为止。

some 方法和 every 相反，它将针对数组中的每个元素执行回调函数，直到其中一个元素返回 true 为止。

每个回调函数都有 3 个参数：element、index 和 array。有些会返回一个值，有些不返回值。所有的回调函数都不会对原始的数组产生影响。

示例 5.6 展示了在数组中使用回调函数的方法。在该示例中，原始数组所包含的元素也是一个数组，这些数组中是一些颜色值，不过有些“颜色”的值超过了允许的取值范围（即 0～255）。当该数组构建好之后，将添加一个名为 checkColor 的函数，它将检查每个数组元素，检查它们是否在允许的取值范围内，然后将不满足这个取值范围的元素过滤掉。

这将引发第二个问题，因为这时颜色数组中的元素个数不足 3 个，无法完成必需的红-蓝-绿设置。因此将为该数组应用第二个回调函数，以确保每个颜色数组中拥有 3 个 RGB 值。

示例 5.6　使用带有数组 filter 方法的回调函数

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>Callbacks</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript">
//<![CDATA[

// 用来检查颜色值范围的回调函数
function checkColor(element,index,array) {
  return (element >= 0 && element < 256);
}

// 检查并确保有 3 个 RGB 色
function checkCount(element,index,array) {
  return (element.length == 3);
}

function testingCallbacks() {

   // color 数组
   var colors = new Array();
   colors[0] = [0,262,255];
   colors[1] = [255,255,255];
   colors[2] = [255,0,0];
   colors[3] = [0,255,0];
   colors[4] = [0,0,255];
   colors[5] = [-5,999,255];
   colors[6] = [255,255,1204556];
```

```
    // 基于颜色值范围进行过滤
    var testedColors = new Array();
    for (var i in colors) {
        testedColors[i] = colors[i].filter(checkColor);
    }

    // 输出第一轮的结果
    document.writeln("<h3>First check</h3>");
    for (i in testedColors) {
        document.writeln(testedColors[i] + "<br />");
    }
    // 过滤出少于3个值的元素
    var newTested = testedColors.filter(checkCount);

    document.writeln("<br /><h3>Second</h3>");
    // 输出余下的元素
    for (i in newTested) {
        document.writeln(newTested[i] + "<br />");
    }
}
//]]>
</script>
</head>
<body onload="testingCallbacks();">
<p>Some content</p>
</body>
</html>
```

最后，只有 4 个颜色数组通过了测试，通过第二个函数的最后一个元素是数组中的第五个元素，正如程序输出所示：

```
First check
0,255
255,255,255
255,0,0
0,255,0
0,0,255
255
255,255

Second
255,255,255
255,0,0
0,255,0
0,0,255
```

虽然回调函数能够简化 JavaScript 应用程序，但它们只是可选的。你也可以使用一个循环语句来检查数组中的每个元素。如果希望在 IE 中也能够正常使用，就必须使用循环语句来完成。

5.4 函数类型小结

总而言之，函数只有 3 种类型：

- 声明式函数　一个拥有自己语句的函数，最开始是关键字 function。声明式函数只会被解析一次，它是静态的，并且提供了一个名称以便访问它；
- 匿名函数　一个使用构造器创建的函数。每次访问它时都将解析一次，而且没有指定一个特定的函数名称；
- 函数字面量或函数表达式　在其他语句或表达式中创建的函数。它只会被解析一次，它是静态的，可以指定也可以不指定一个特定的函数名称。如果它是已命名的，那么也只能够在其定义的函数体内访问它。

声明式函数在任何浏览器中的任何形式的 JavaScript 代码里都是有效的。基于函数构造器的匿名函数是动态的，它占用的内存也更大，只有较新版本的 JavaScript 中提供了支持；同样，在老版本的浏览器中也可能是不可用的。函数字面量是更后面的创新，在 JavaScript 1.5 之后才提供了支持，而且只在最新的浏览器中提供了支持，包括最流行的 Mozilla、Firefox、IE 和 Safari。不过，使用函数字面量会引发内存使用方面的复杂问题，正如本章中的“嵌套的函数、函数闭包与内存泄漏”小节所介绍的那样。

函数字面量也是一些较高级的 Ajax 程序库的基础，在后续章节中我们进一步说明 Ajax 和 JavaScript 高级应用时就会看到。另外，函数字面量是在需要回调函数的对象事件句柄中经常使用的，就像这些与数组对象相关的事件句柄那样。

谈到对象，正如之前所说，JavaScript 中的函数也是对象，只不过它是一个语句序列。在本章的最后一个小节中，我们将更完整地说明这一概念。不过，我们还是先来看看函数的作用域，以及它对全局变量及局部变量的影响。

5.5 函数作用域

在第 2 章中，我们曾经介绍过全局变量、局部变量及变量作用域的概念。作用域关系到一个变量是否能够在一个函数的作用域外访问。

在一个函数中，如果在变量定义时使用了 var 关键字，那么这将告诉 JavaScript 处理程序该变量的作用域声明为局部，这意味着它只在函数体的代码中可用。在函数外，这个变量是未定义的：

```
function test () {
  var myTest = "test";
  ...
  alert(myTest);  // 输出"test"
```

```
}
alert(myTest);      // 未定义
```

在函数外也可以使用 var 关键字，但不管你是否使用这个关键字，该变量的作用域都将是全局的，这是因为它并不包含在任何函数中。

另外，如果你忘了在函数中使用 var 关键字，那么 JavaScript 处理程序将会认为它是一个全局变量，以进行相应的处理：

```
function test() {
   myTest = "test";
   ...
   alert(myTest);  // 输出"test"
}
alert(myTest);     // 输出"test"
```

对于一个较小的应用程序，全局变量不会带来任何问题。不过，在一个大型的应用程序中，可能其他全局变量会使用相同的名字，这时你使用的全局变量就会覆盖重要的应用程序信息。同样，你的重要信息也可能被覆盖。

性能也是一个问题。当变量声明为局部变量时，当函数执行结束时变量所使用的资源将由垃圾回收机制释放，以便其他应用程序可以使用。但对于一个全局变量而言，在应用程序执行结束之前资源是不会被释放的。

简单来说，请记住在函数中要用 var 关键字来定义变量，避免使其成为全局变量。如果你需要一个全局可访问的数据，那么你可以创建一个针对自己应用程序的自定义对象，然后将所有所需的全局值都赋给它。这样做，至少能够控制应用程序需要多少个全局变量。在第 13 章中，我们将更详细地讲解自定义对象。

JavaScript 的新版本和后续版本还允许你定义一个作用域仅为一个程序控制块的变量，如一个循环体内。不过这时不是使用 var 关键字，而是使用 let 关键字，如下所示：

```
function myTest() {
   var someVar = 1;
   while(someTest) {
      let someVar = 2;  // 不同的变量
   }
}
```

不过，使用 let 关键字定义程序控制块级的作用域还是很新的做法，现在还没有被广泛支持，因此我们还不能依赖它。

5.6 函数就是一个对象

创建一个对象时通常会使用一个带属性和方法的构造器，函数也不例外。

函数对象看起来是一个最容易被修改的 JavaScript 对象。最初有一个提供参数数量的

arity 属性。后来它被函数名称的 length 方法所代替，当然你也可以通过参数数组的 length 属性来获得参数数量。它本身最好是通过函数名访问的，但现在又被改为在函数调用时作为参数访问。

对于函数对象而言，只能使用标准的属性和方法，它和针对所有对象的标准并不同，如 toSource、toString 及针对参数数量的 length，以及一些可调用的方法，我们将在第 13 章中详细说明。

另外，正如你在第 13 章中将看到的那样，当构建一个自定义对象时，函数能够通过特定的 this 关键字引用其自己的作用域，这对于构建针对新对象的类而言十分重要。

5.7 知识测验

1. 对一个数（n）执行阶乘操作，那么将得到 1～n 的数字乘积，通常写做 3!（1×2×3 或者 6）。编写一个 JavaScript 函数，使用递归的方法计算出指定数字的阶乘值。
2. 函数如何才能修改其作用域之外的变量？编写一个函数，传入一个由 1～5 的数字组成的数组作为参数，调用该函数后将把其中的数字项替换成相应的字符串（也就是“one”、“two”等）。
3. 创建一个函数，它的参数是一个数据对象和一个函数，它将对这个数据对象调用该函数。

5.8 测验答案

1. 下面是可能的解决方案之一：

```
function findFactorial(n) {

  if (n == 0) return 1;

  return (n * findFactorial(n-1));
}

var num = findFactorial(4);    // 将返回 24
```

2. 如果将一个对象（如一个数组）作为函数的参数传入，那么函数中对这个数组的修改就会反映到函数外。针对这个问题，其中一种解决方案是：

```
function makeArray() {
    var arr = [1,5,3];
    alert(arr);
    alterArray(arr);
    alert(arr);
```

```
}

function alterArray(arrOfNumbers) {
  for (var i = 0; i < arrOfNumbers.length; i++) {
    switch(arrOfNumbers[i]) {
      case 1 : arrOfNumbers[i] = "one"; break;
      case 2 : arrOfNumbers[i] = "two"; break;
      case 3 : arrOfNumbers[i] = "three"; break;
      case 4 : arrOfNumbers[i] = "four"; break;
      case 5 : arrOfNumbers[i] = "five"; break;
    }
  }
}
```

3. 用一个匿名函数就能够满足需求：

```
function invokeFunction(dataObject, functionToCall) {
   functionToCall(dataObject);
}
var funcCall = new Function('x','alert(x)');
invokeFunction ('hello', funcCall);
```

第 6 章

排错、调试及跨浏览器问题

直到现在为止，我们向你展示的 JavaScript 代码都很容易创建和调试，因为这些例子都和 HTML 代码在同一个页面中。但在本书下面的内容中，所举的例子将更加复杂，并且将引入内建的对象模型，包括文档对象模型（DOM）。复杂性增加、对象模型的应用都需要有一个用来详细控制 JavaScript 程序执行过程，并仔细分析代码中所访问对象的调试器。

另外，从第 7 章“捕获事件”开始，不同浏览器对 JavaScript 提供的不同支持将逐渐显露出来。虽然和多年前第一次使用 JavaScript 时相比，对 JavaScript 的支持有了很大的改进，但仍然不是所有浏览器都对 JavaScript 的所有功能提供了支持。另外，浏览器厂商还在不断地对 JavaScript 的新的和后续的增强进行测试，知道何时使用（以及何时不使用）一个新功能是很重要的。

6.1 调试的简单方法

拥有长期开发经验的程序员可能会不屑于这种方法，不过使用某种输出机制来检查变量在程序处理时的值仍然是最简单的调试方法。虽然也有一些比较复杂的调试工具，但这仍然是检查变量在应用程序运行过程中的状态和当前值的简单方法。

在 JavaScript 中，输出信息的最简单方法是使用 alert 对话框，也就是我们在前一章中所使用过的：

```
alert(someVariable);
```

如果你想快速地完成数据检查，而不想启动一个调试程序，那么可以添加一个 alert 对话框来输出你需要检查的值，然后马上将这个 alert 对话框去掉。

在下面的小节中，我们将介绍一些更加复杂的 JavaScript 应用程序调试方法，它们将使用针对每种浏览器的专有工具。

6.2 浏览器提供的开发和调试工具

在过去的几年中，我无法判断哪种工具对于 JavaScript 程序员会更好用些，因为绝大多数主流的浏览器都提供了类似的 JavaScript 支持，并为开发人员提供了调试 JavaScript 应用程序的工具，通过它们可以检查程序的执行过程和效果。

6.2.1 Firefox 和 Firebug

Firefox 是当前流行程度位居第二的 Web 浏览器，不过当提及更复杂的 Ajax 的开发时，它或许堪称最为流行。它在 JavaScript 的开发人员中获得如此高声望的原因之一就是名为 Firebug 的 Firefox 扩展。

提示

Firebug 可以从 http://getfirebug.com/中下载到。

Firebug 提供了一个带标签页的用户界面，通过它可以访问 JavaScript 控制台和各种消息，并且能够对 HTML 或 CSS 代码进行检查，细究在任何页面中可访问的 DOM 对象，检查特定页面上发出的所有网络调用，并且提供了如图 6.1 所示的 JavaScript 调试窗口，这对本章主题而言是最重要的。

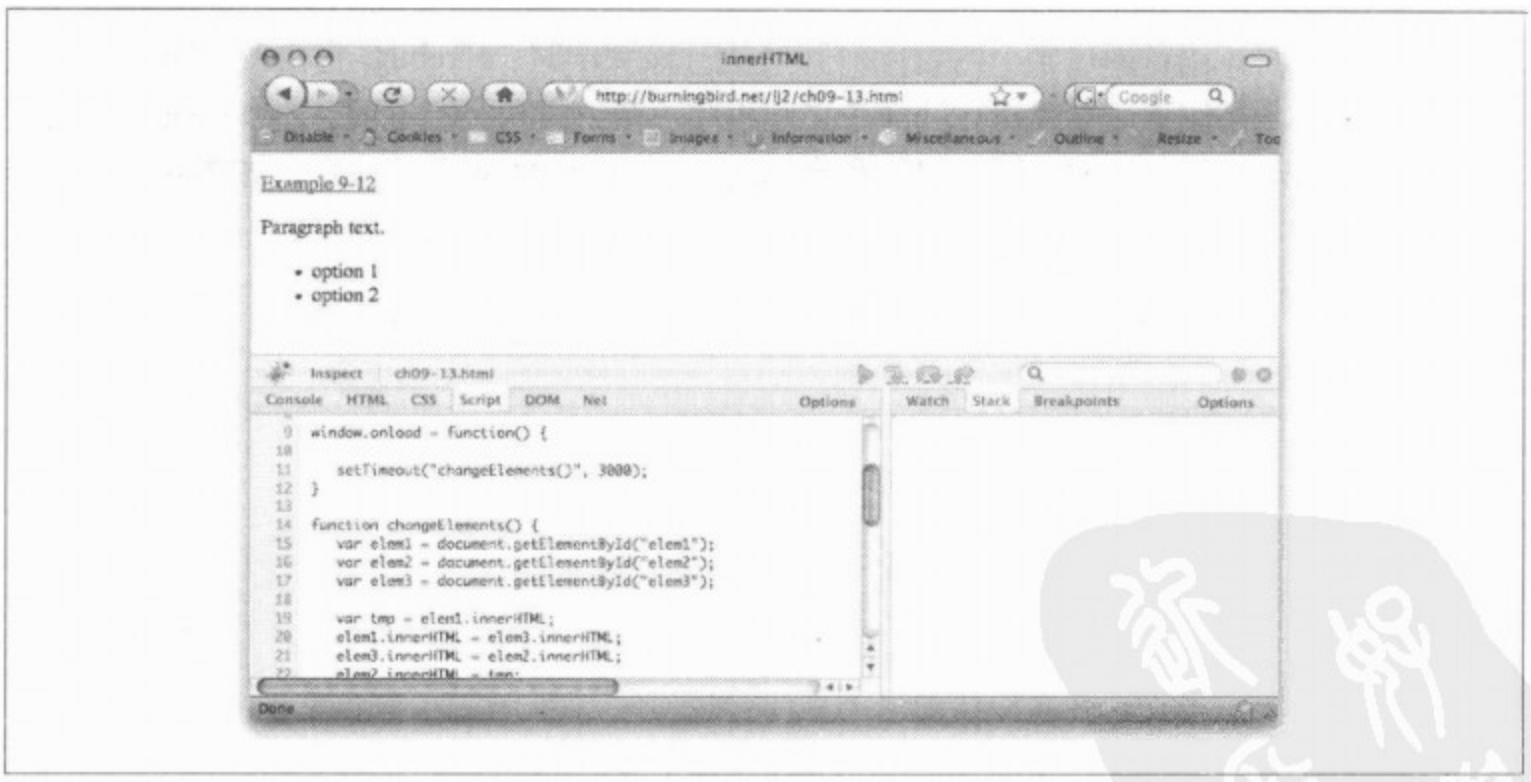


图 6.1 在 Firebug 扩展中打开的一个带脚本的 Web 页面

在 Firebug 的脚本窗口中，你可以从一个下拉列表（列出了该 Web 页面上可用的所有 JavaScript 文件）中选择想调试的 JavaScript 文件。当打开选择的脚本后，你可以复查该脚本，在代码中添加断点。使用断点可以让脚本引擎执行到这个位置时暂停执行，

这时你就可以检查所有局部、全局变量以及所有的 DOM 对象，以观察应用程序的执行结果，分析是否遇到问题。

在 Firebug 中，设置断点所需的操作就是找到想要设置断点的那行代码，然后单击鼠标右键。图 6.2 中展示了设置了两个断点之后的界面外观。

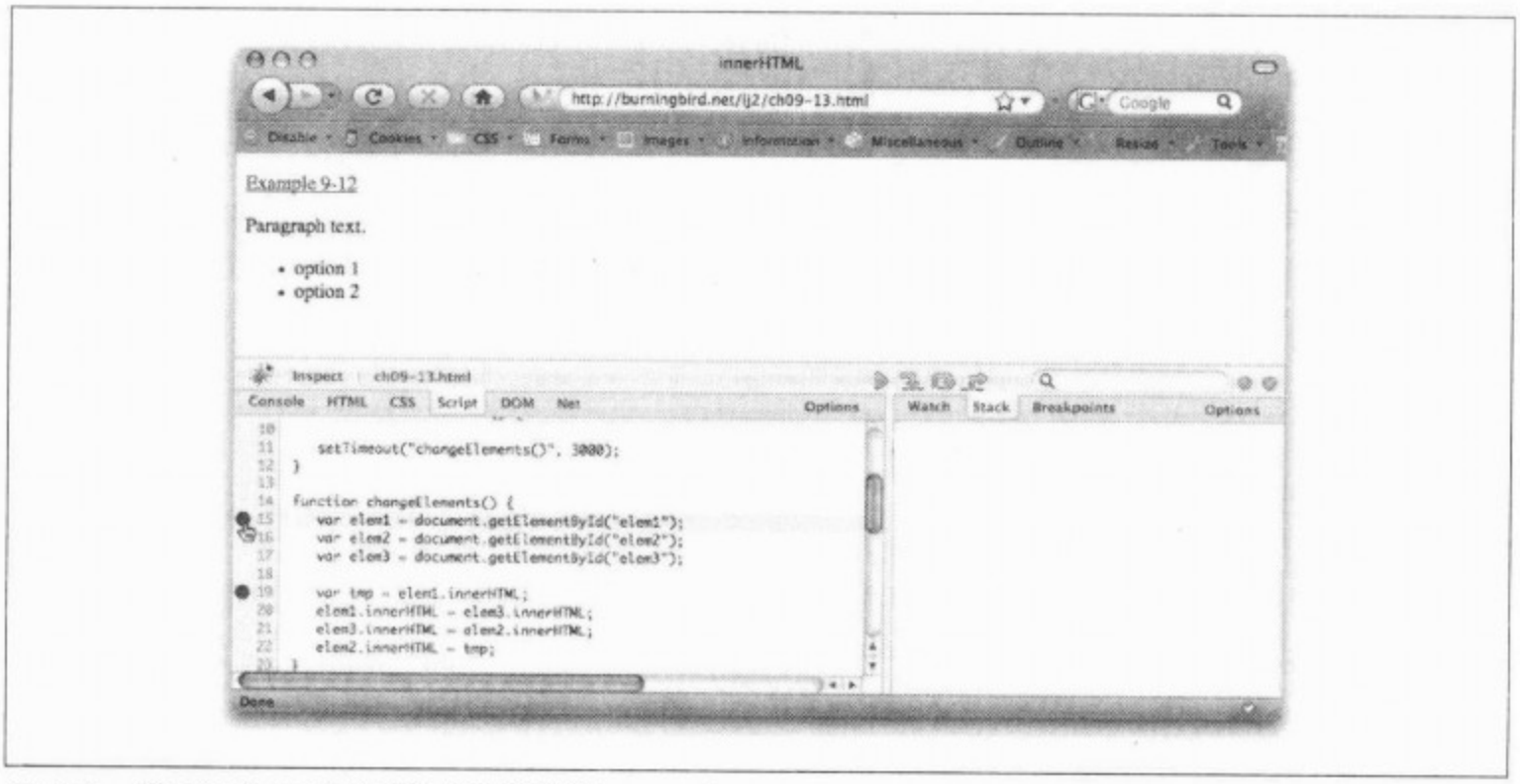


图 6.2　在 Firebug 中设置了两个断点

当你载入该页面，或者通过某个特定操作开始执行该脚本时，Firebug 将会使程序执行停止在断点位置上，在你下达继续执行的命令之前程序不会继续执行。这时，在 Firebug 右边的面板中将显示诸如脚本处理过程、局部和全局变量以及 DOM 对象等信息，如图 6.3 所示。

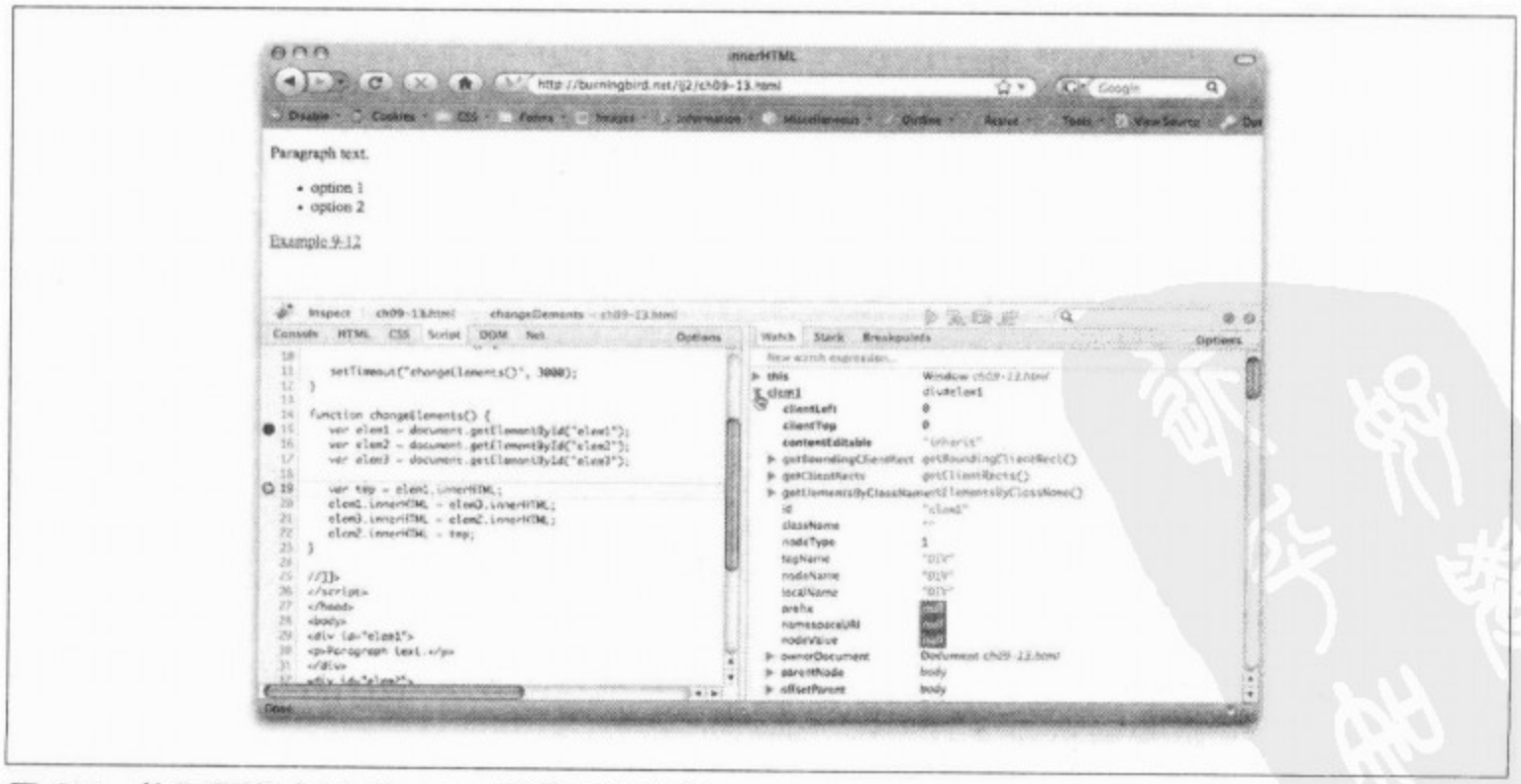


图 6.3　执行到断点时 Firebug 所显示的内容

在该图中所打开的应用程序（参见第 9 章中的示例 9.13）里有大量局部变量，这些变量都是 Web 页面中的元素。在图 6.3 中，可以看到变量 elem1 的值是一个页面中 id 为“elem1”的 div 元素的引用，还可以看到该对象可用的所有属性和方法。对于每个页面元素对象而言，拥有的属性和方法的数量令人惊讶。

在右边的顶部还有一些箭头按钮，它们分别用来让脚本调试器继续执行程序（程序将继续执行直到代码结束或遇到下一个断点）；以 step over[1]模式单步执行；以 step into[2]模式单步执行（当该语句是一个函数调用时很有用）；或者以 step out[3]模式单步执行（如果它位于一个函数中）。

当然，这里的脚本是个很简单的例子。图 6.4 中展示了在复杂的 JavaScript 应用程序中应用 Firebug 时的情况，这是我的网站中使用到的 Lightbox 程序库。这些 JavaScript 程序库和应用程序可能有几百行甚至几千行代码。

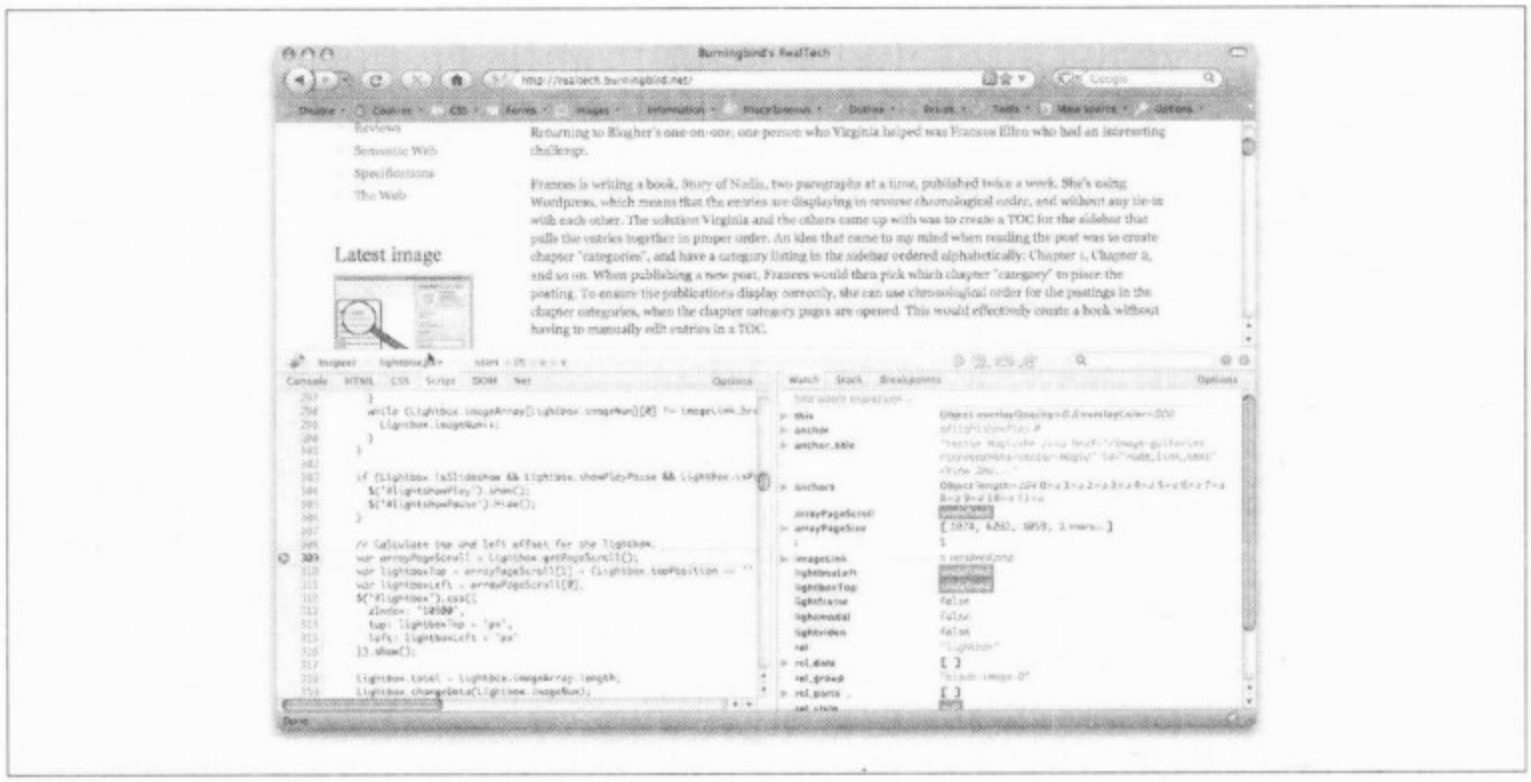


图 6.4　用 Firebug 调试一个复杂的 JavaScript 应用程序

提示

在 http://www.lokeshdhakar.com/projects/lightbox2/中可以找到 Lightbox 程序库，它用来实现在页面中显示照片。

在右边列出了一些标签页，分别用来显示当前的程序堆栈（Stack 标签页）和现有程序断点（Breakpoints 标签页，它在调试大型应用程序和拥有大量断点时有意义）。

---

1 译者注：step over 的意思是在单步执行时，如果在函数内遇到子函数，不会进入子函数内单步执行，而是将子函数整个执行完再停止，也就是把整个子函数作为一步。

2 译者注：step into 的意思是在单步执行时，遇到子函数就进入并且继续单步执行。

3 译者注：step out 的意思是在单步执行时，执行完该函数中的余下所有部分程序，并返回到上一层函数中。

在 Stack 标签页中展示的是程序执行到当前断点的脚本嵌套。在图 6.5 中，断点设置在一个名为 changeData 的对象方法中，它源于另外一个流行的、名为 jQuery 的 JavaScript 程序库中封装的一个对象，该方法是在某个特定事件（由 jQuery 管理的）被触发之后执行的。

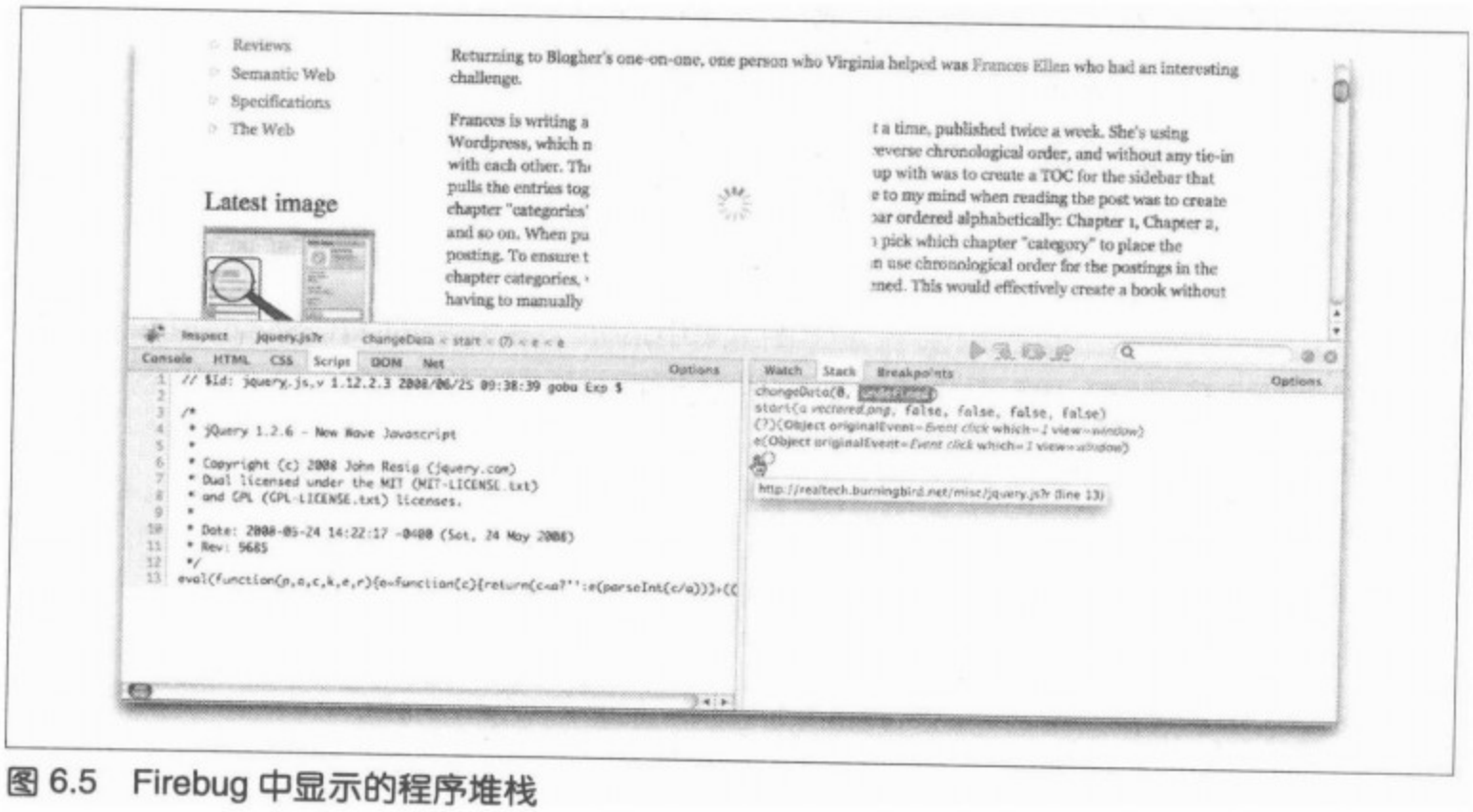


图 6.5　Firebug 中显示的程序堆栈

Firebug 还提供了一个 JavaScript 控件台，在此你不仅可以看到状态和错误，还可以输入 JavaScript 代码并执行它，然后将这些代码复制到你的 Web 页面或应用程序中。除了调试之外，你还可以用 Firebug 替代 document.writeln 及 alert 对话框，这些都是本书中用来传达 JavaScript 代码执行结果的工具。你无须在页面中写入结果值，可以改成将其输出在 Firebug 提供的控制台上。

6.2.2　使用 console.log

想要将结果写入 Firebug 控制台，则需要在你的程序中使用以下代码：

```
console.log("Here is the result: %s", theVariable);
```

对于要写入控制台的字符串而言，也可以接受来自变量的输入，这里需要使用占位符，例如针对字符串的%s 和针对数字值的%d：

```
console.log("My name is %s and I'm %d years old", myName,myAge);
```

示例 6.1 中展示了改用 Firebug 控制台之后的“Hello World!”示例。

示例 6.1　使用 Firebug 控制台来传达 JavaScript 执行结果

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
```

```
<head>
<title>Hello, World!</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />

<script type="text/javascript">
function hello() {

   // 向世界问好
  var msg = "Hello, World!";
  console.log("%s",msg);
}
</script>
</head>
<body onload="hello()">
<p>Hi</p>
</body>
</html>
```

图 6.6 中展示了示例 6.1 执行后 Firebug 控制台中显示的内容。

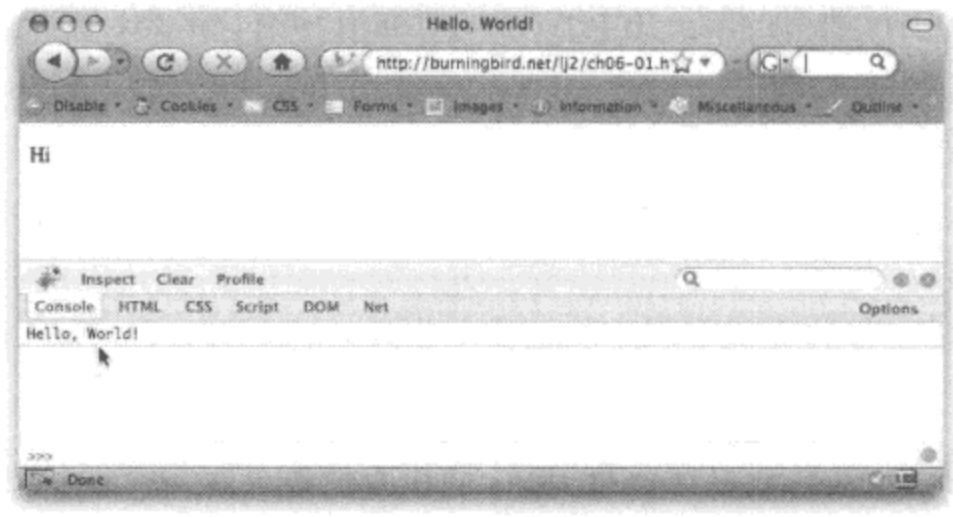


图 6.6　使用 Firebug 控制台输出 JavaScript 执行结果

当然，Firebug 控制台只有在安装了 Firebug 之后才可用。不过，浏览器开发人员使用的绝大多数调试程序都提供了该功能，包括本章后续介绍的所有工具，都可以通过 console.log 向调试程序控制台写入信息。如果是这样，可以马上用这个方法代替 document.write 和 alert 对话框，用来作为传达 JavaScript 执行结果的新方法。这样做的好处是使用 document.write 方法将会对 Web 页面产生修改，而 alert 对话框也并不是查看执行结果的最佳方法。

尽管如此，直到现在为止我还是坚持使用传统的、经过检验而且可靠的方法。

提示

在第 14 章中，将说明如何融合诸如 jQuery 之类的外部 JavaScript 程序库。

6.2.3 Firefox、Web Developer toolkit 和 NoScript

Firebug 是最常用的 Firefox 开发工具，但我还发现有许多 Firefox 扩展对于 JavaScript 及其他 Web 开发而言，具有很高的价值。

Firefox 中最早也是最流行的 Web 工具是 Web Developer toolkit（该工具可以在 http://addons.mozilla.org/en-US/firefox/addon/60 中下载到）。该工具集将在独立的工具条中添加几个下拉菜单，最重要的一键式功能包括页面 HTML 和 CSS 代码的校验、可访问性检查、查看 CSS 和 cookies、检查图像、查看 JavaScript 修改后的页面源代码（包括动态源代码）。最后一个最重要，当 JavaScript 搞乱页面之后，能够提供良好的查看功能的工具并不多。

该工具集还提供了禁用标准页面功能的手段，包括 cache、JavaScript 等。如果你希望 JavaScript 应用程序在禁用脚本的浏览器中运行时，能够提供有效的退而求其次的方案，那么这将是至关重要的。使用 Web Developer toolkit 提供的这一功能可以测试当脚本被禁用时 Web 页面的执行情况。

Web Developer toolkit 中的 Disable JavaScript（禁用 JavaScript）选项并不是测试脚本被禁用时网站执行情况的唯一方法。另一个很流行的 Firefox 扩展是 NoScript（在 https://addons.mozilla.org/en-US/firefox/addon/722 中可以下载到），它允许你控制哪些网站启用脚本，哪些网站禁用脚本。

实际上，Firefox 中还有许多其他针对 JavaScript 的扩展，除了 Firebug 提供的控制台之外还有很多不同版本的控制台，最流行的 Greasemonkey 扩展还提供了修改 Firefox 甚至是 JavaScript 编辑器行为的手段。在这些扩展中，有很多不仅能提供 JavaScript 调试功能，还能够对 JavaScript 代码的有效性进行校验。

警告

当然，安装过多的扩展会使 Firefox 变得很缓慢。如果我不是在开发 Web 应用程序，就会禁用安装在 Firefox 中的绝大多数针对开发人员的扩展，包括（特别是）Firebug。不过，我仍然会启用 Web Developer toolkit，因为我喜欢用它来分析其他开发人员的工作成果。

6.2.4 Opera 和 Dragonfly

Opera 发布 9.5 版本时，还同时发布了一个名字很怪异的开发工具：Dragonfly。它和 Firebug 很像，可以实现脚本载入、断点设置、变量值检查等功能。不过在写作本书时，它仍然是 alpha 版本，这些功能还不是很完善，甚至有些简陋。

提示

在 http://www.opera.com/products/dragonfly/上可以下载到 Dragonfly。

接下来，我们只关注于 Dragonfly 中的脚本调试组件。当你安装了 Dragonfly 以后，就

可以在 Opera 的主菜单中选择 Tools（工具）→Advanced（高级）→Developer Tools（开发人员工具）命令来启动该应用程序。Dragonfly 将在浏览器的底部打开一个多帧的调试窗口，和 Firebug 类似。你将看到多个标签页，但你可能最想做的是选择 Scripts 标签页，并选择一个网站或脚本，如图 6.7 所示。

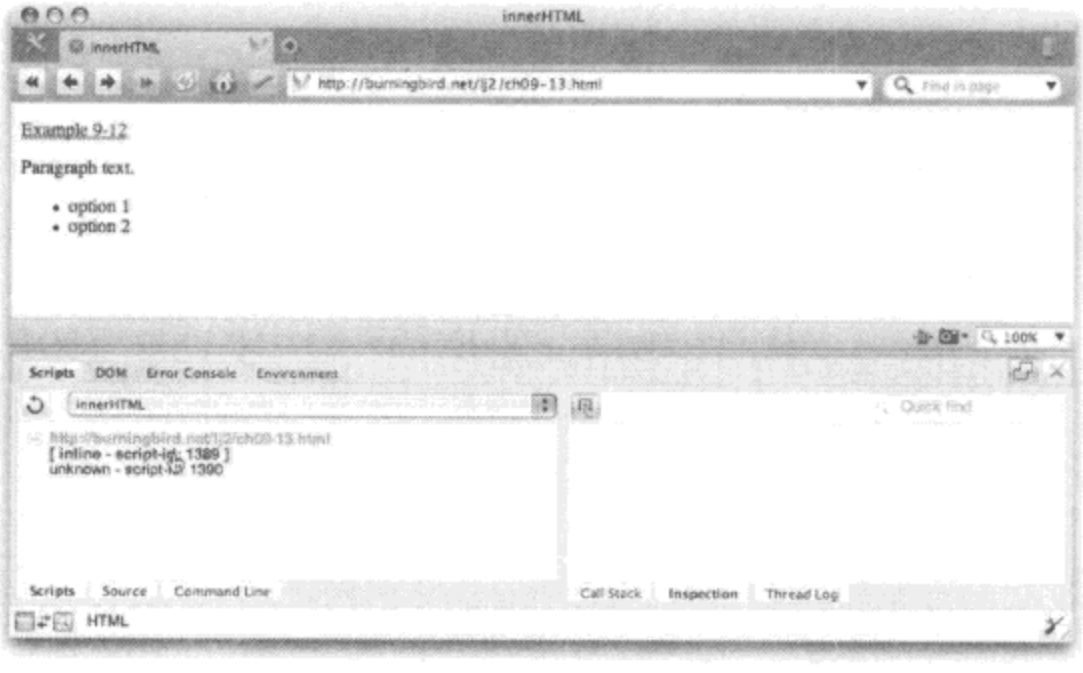


图 6.7　在 Dragonfly 中选择一个脚本

当载入了所选脚本之后，单击 Source（源代码）标签页，就可以查看其源代码、设置程序断点，如图 6.8 所示。

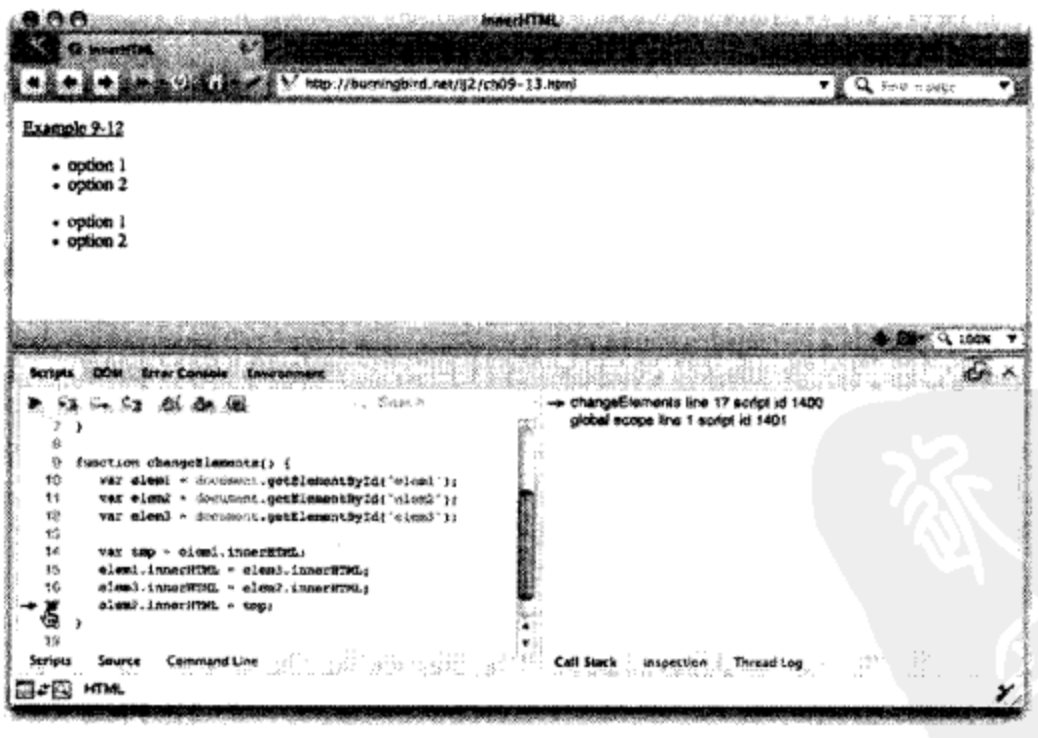


图 6.8　在 Dragonfly 中设置断点

最右边的帧将显示当前执行的线程，在 Call Stack（调用堆栈）标签页中单击某一项将

打开 Inspection（检查）帧，这里会显示出当前程序的所有变量（如图 6.9 所示）。将 Opera 中显示的页面元素值和 Firebug 中显示的结果进行比较是件有趣的事情。

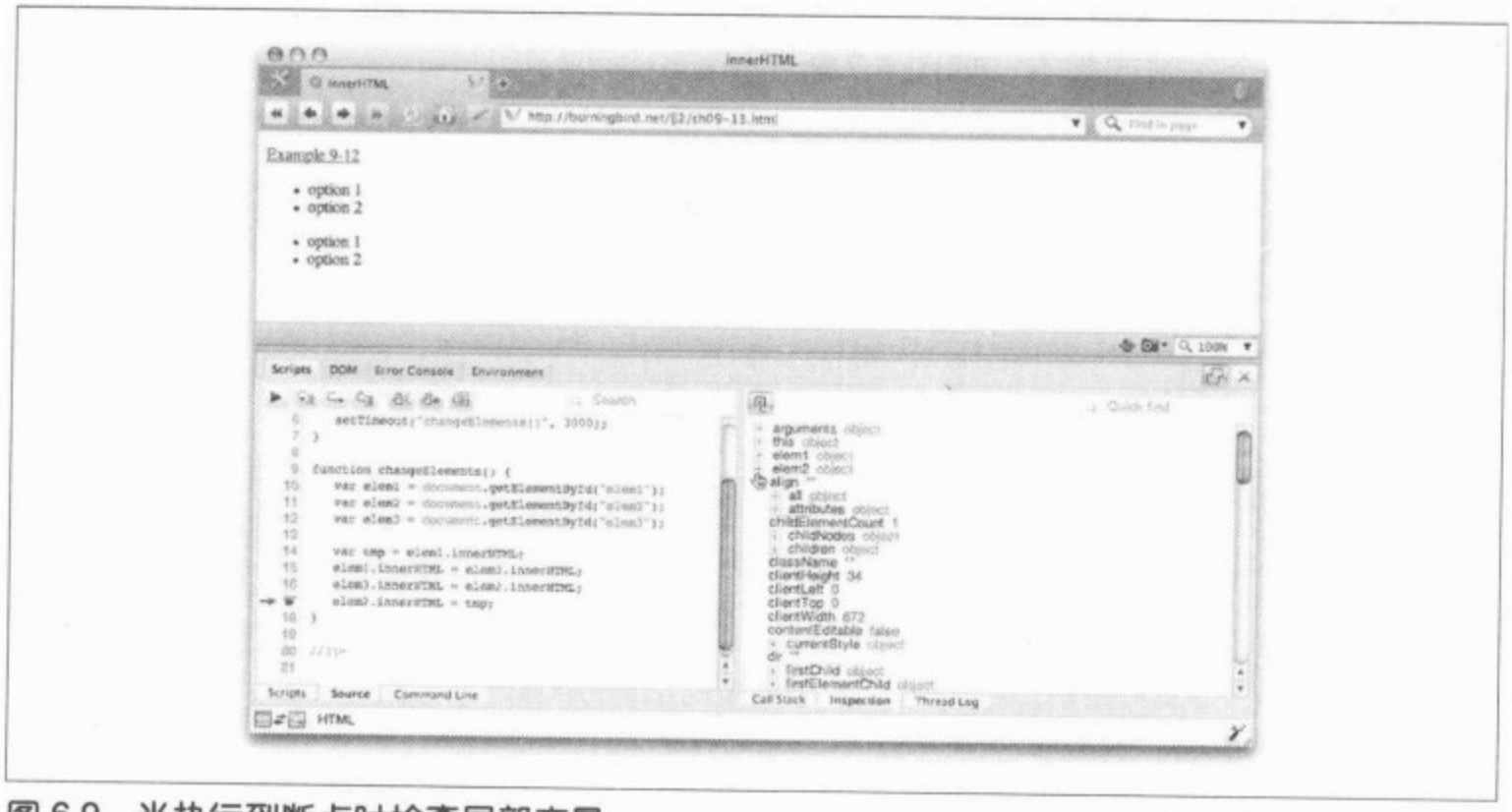


图 6.9　当执行到断点时检查局部变量

6.2.5　Safari/WebKit 和 Web Inspector

Safari 是基于开源 WebKit 开发的浏览器，在其菜单中有内建的、针对开发人员的工具。其中包括能够显示所有脚本错误的 Error console（错误控制台），用来检查 HTML、CSS 和 JavaScript 代码的 Web Inspector。

当 WebKit 发布新版本时，它就被集成到下一版本的 Safari 中，而针对 Web 开发人员的一些功能也随之继续增加，其中之一就是内建的脚本调试器，如图 6.10 所示。在编写本书时，它的功能还十分有限，包括为脚本设置程序断点等功能。不过 Safari 正在不断地提供新的功能，它必将发展成为功能完整的调试程序。

此外，Safari 和 Webkit 还在其 Develop（开发）菜单项中提供了其他调试类的功能，以下是其中的一些在 JavaScript 开发时很有用的功能：

- 关闭/打开缓存；
- 禁用 JavaScript；
- 禁用失控的 JavaScript 定时器（当你使用定时器时，它十分有用）；
- 禁用样式；
- 禁用图像；
- 禁用网站特定的 hacks。

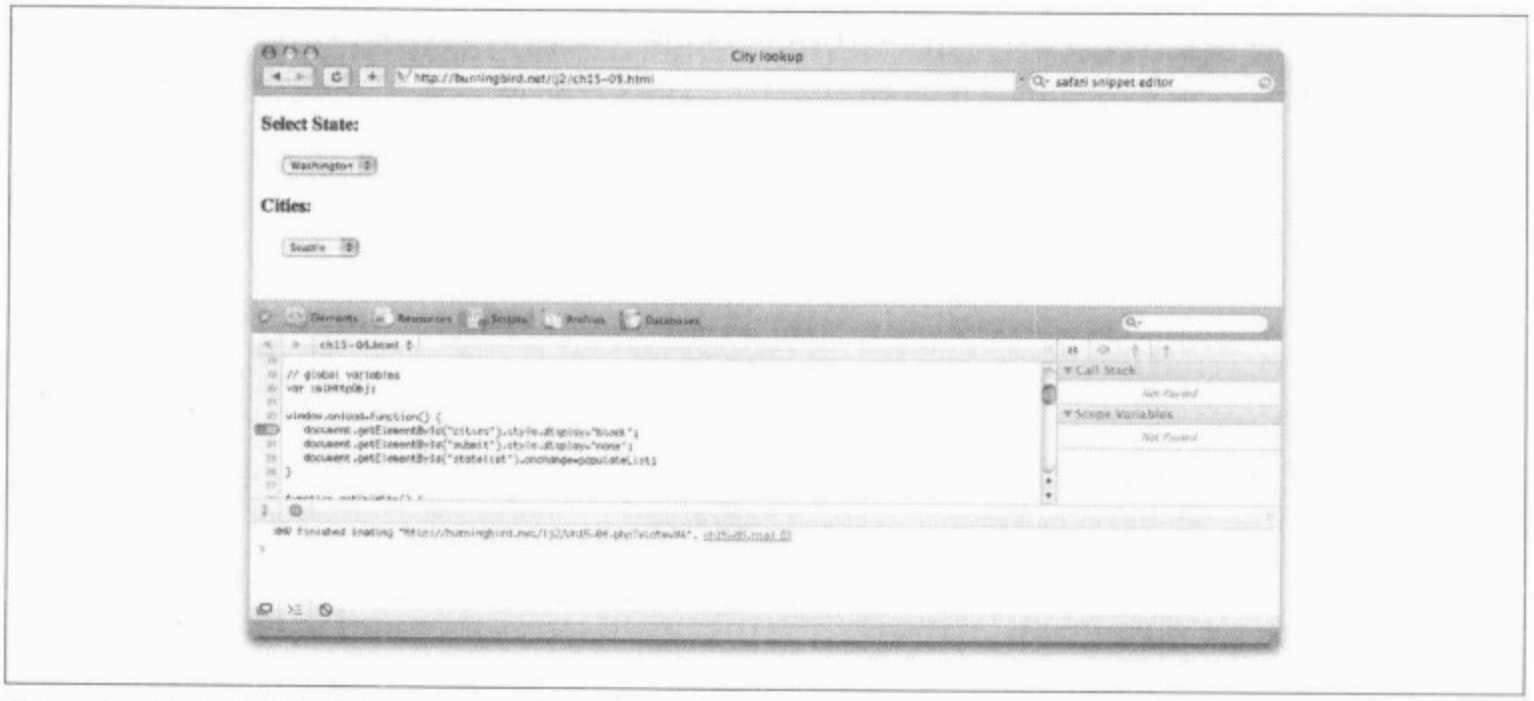


图 6.10　WebKit Web Inspector 中新的调试程序

6.2.6　Internet Explorer

随着 IE 8.0 的发布，微软还发布了一个相当好的调试工具，单击工具条上的 Development Tools（开发工具）项就可以启动它。这时将打开一个独立的窗口，呈现出 3 个分别标记为 HTML、CSS 和 Script 的面板。它们都十分有用，不过我们只研究 Script 面板。

当单击 Script（脚本）标签页时，将看到一个双面板的窗口，左边显示的是脚本的源代码，右边显示的是消息窗口，如图 6.11 所示。

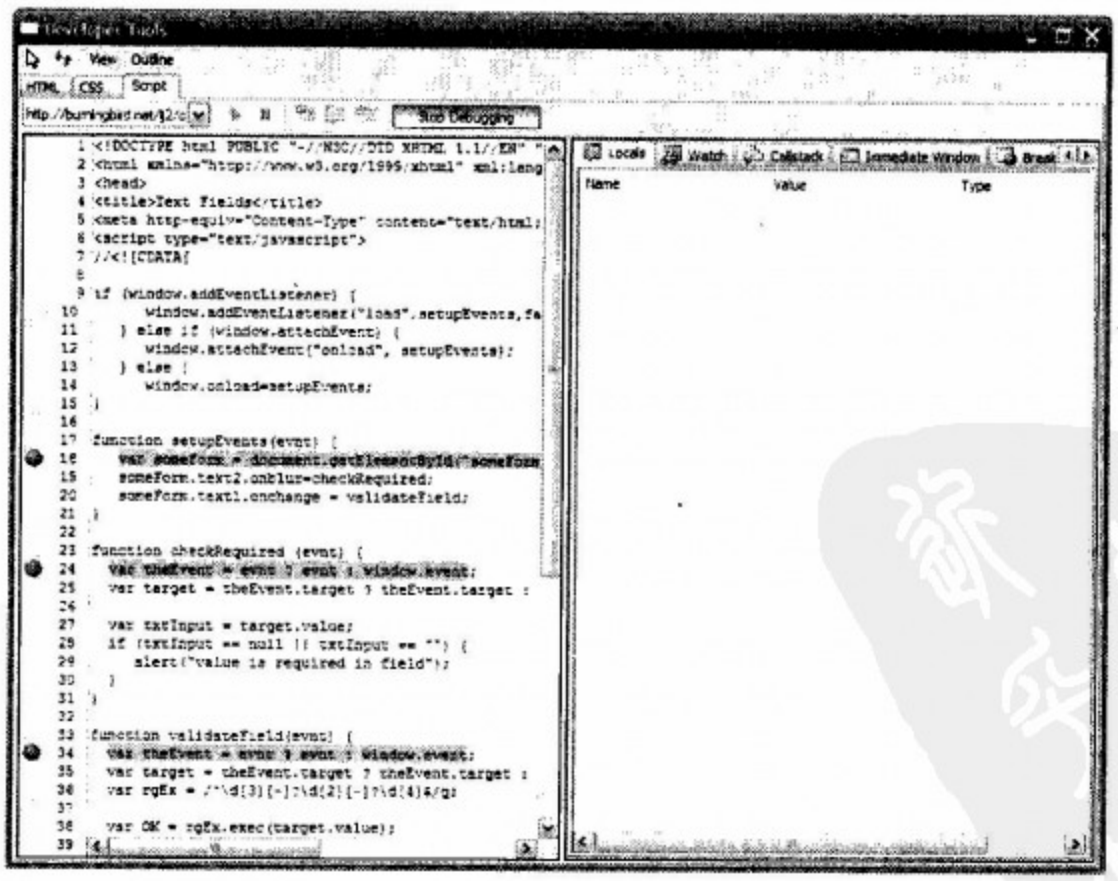


图 6.11　IE 8.0 中的脚本调试程序

当你要开始脚本调试时，只需单击 Start Debugging（开始调试）按钮，然后根据需要在代码中添加断点。和 Firebug、Dragonfly 类似，要创建断点只需单击需要暂停的代码行即可。当执行到断点时，它也提供了 step into、step over 等单步调试选项以及继续运行的选项，这和 Firebug、Dragonfly 是类似的。当程序暂停时，你也可以检查局部变量和全局变量的值，如图 6.12 所示。

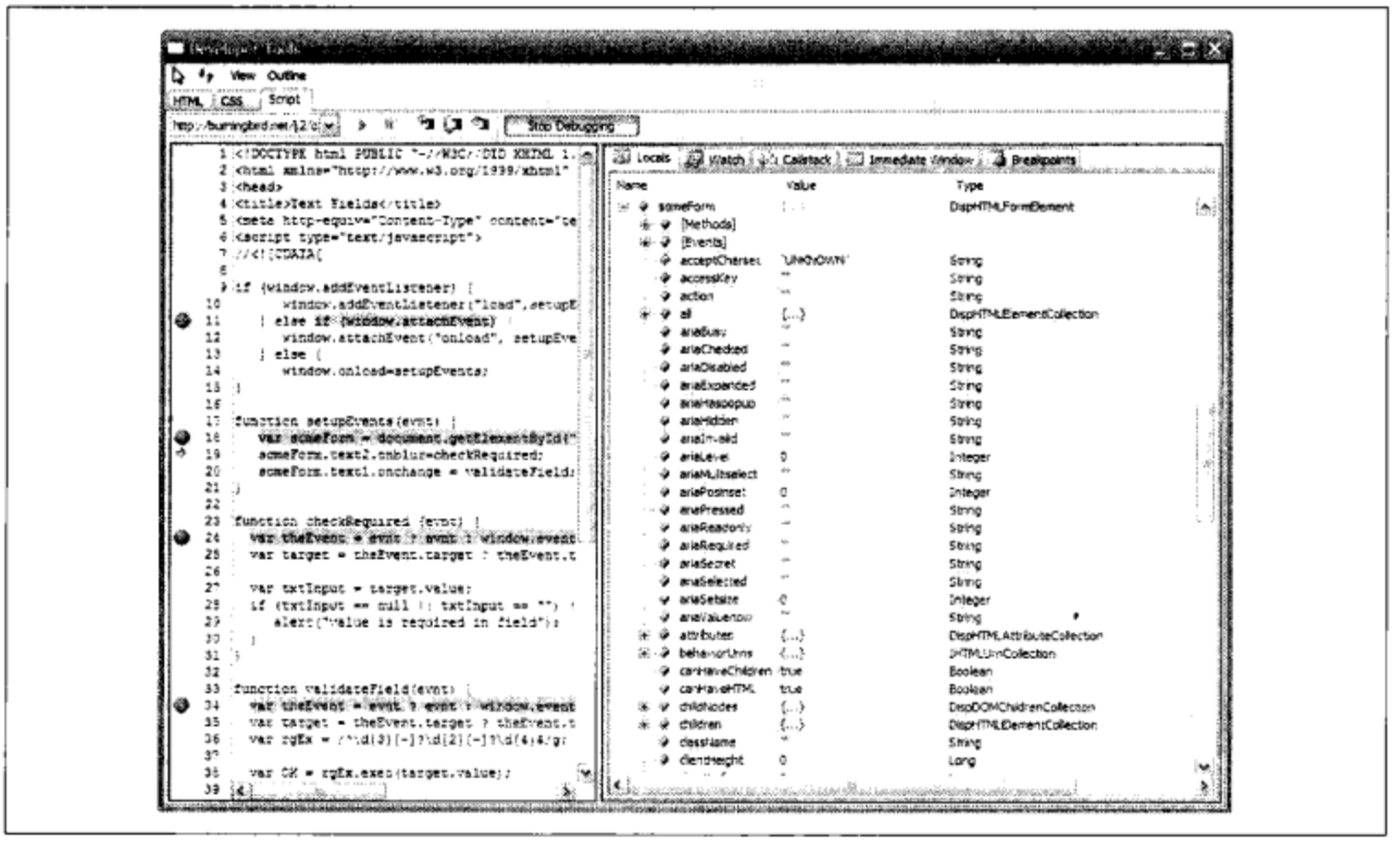


图 6.12　当执行到断点时，检查赋给局部变量的 Web 页面元素

6.3　处理浏览器之间的差异

10 年前，处理浏览器之间的差异，主要是当时两个具有统治性地位的 IE 和 Netscape 的 Navigator 之间的不同，由于 JavaScript 和 DOM 标准要么还很新，要么根本不存在，因此这是一个需要勇气的痛苦过程。虽然现在的浏览器更多了，但是工作却更加简单，因为绝大多数的浏览器都支持相同的规范，不过并不是全部浏览器都做到了这一点。

差异仍然存在，而且这些差异并不是只限于 Firefox-Opera-Safari 和 IE 之间，这 4 种浏览器仍然有一些不同。

6.3.1　对象检测

以前，我们通过 navigator 对象来检查 user agent 字符串，以确认浏览器类型、版本。以下就是一个实例（源于很早的一篇名为 *NetscapeWorld* 的文章，现在还能在 Wayback Machine 网站中找到它，其链接是 http://web.archive.org/web/19981205085025/

http://developer.netscape.com/docs/examples/javascript/browser_type.html)。

```
// *** 浏览器版本 ***
this.major = parseInt(navigator.appVersion)
this.minor = parseFloat(navigator.appVersion)

this.nav  = ((agt.indexOf('mozilla')!=-1) && ((agt.indexOf('spoofer')==-1)
              && (agt.indexOf('compatible') == -1)))
this.nav2 = (this.nav && (this.major == 2))
this.nav3 = (this.nav && (this.major == 3))
this.nav4 = (this.nav && (this.major == 4))
this.nav4up = this.nav && (this.major >= 4)
this.navonly      = (this.nav && (agt.indexOf(";nav") != -1))

this.ie   = (agt.indexOf("msie") != -1)
this.ie3  = (this.ie && (this.major == 2))
this.ie4  = (this.ie && (this.major == 4))
this.ie4up  = this.ie  && (this.major >= 4)

this.opera = (agt.indexOf("opera") != -1)
```

正如你所能想到的那样，这个过程是令人痛苦的，你很快就会失去使用对象检测方法的兴趣。

使用对象检测方法，JavaScript 应用程序可以访问通过条件分支语句检测出来的对象。如果该对象不存在，那么该分支语句将返回 false。例如，以下判断用来确保浏览器或其他用户代理能够提供最基本的对象模型支持：

```
if (document.getElementById)...
```

如果这个条件判断返回的是 false，那么所有动态 Web 网页效果都将失去。幸运的是，所有现代浏览器所支持的模型都很一致。我们支持的浏览器都支持 document.getElementById 方法，这是访问特定元素的关键方法；也都能够支持 style 属性，通过它可以修改元素的 CSS 样式属性。

不过直到现在差异仍然存在。例如，从第 7 章开始介绍的 JavaScript 事件处理机制就有一些差异。绝大多数浏览器都支持 DOM Level 2 事件处理机制，但微软的 IE 则不支持，甚至 IE 8.0 也不支持。基于这一原因，任何事件处理都需要进行对象检测。例如，如果你想确定是哪个元素接收了 Web 页面中的 click 事件，就需要采用如下所示的 JavaScript 代码，它源于第 12 章的示例 12.9：

```
function showBlock(evnt) {

  var theEvent = evnt ? evnt : window.event;
  var theSrc = theEvent.target ? theEvent.target : theEvent.srcElement;
  var itemId = "elements" + theSrc.id.substr(5,1);
  var item = document.getElementById(itemId);
  if (item.style.display=='none') {
```

```
            item.style.display='block';
        } else {
            item.style.display='none';
        }
    }
```

你只需关注最前面两行加粗的代码。在 DOM Level 2 中，事件对象可以作为参数传递给事件句柄函数。但在 IE 8.0 中无法自动传递这个事件对象，而是需要通过 window 对象来访问这个事件对象。

这里的对象检测使用了三元操作符（参见第 3 章），以判断参数是否是 evnt，它是一个非空的、已定义的对象。如果是，那么将赋给一个局部变量 theEvent。如果不是，则将 windows.event 属性赋给该局部变量：

```
var theElement = evnt ? evnt : window.event;
```

到此工作仍然没有完成；在此之后是对被点击的实际对象的处理。我们再一次遭遇了浏览器间的差异。

对于绝大多数浏览器而言，target 属性保存着一个指向接收到该事件的元素的引用。不过在 IE 中，我们需要通过 scrElement 来访问该元素：

```
var theSrc = theEvent.target ? theEvent.target : theEvent.srcElement;
```

示例 6.2 展示了使用该方法输出 click 事件所针对元素的页面。

示例 6.2　使用对象检测方法查找事件针对的对象

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>object detection</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<style type="text/css">
div {
        position: absolute;
        top: 30px;
        left: 50px;
   }
#div1 img {
filter:
progid:DXImageTransform.Microsoft.AlphaImageLoader(src=fig0902.png,
sizingMethod='scale');
}

</style>
<script type="text/javascript">
//<![CDATA[
```

```
window.onload=function() {
  document.getElementById("div1").onclick=getSrc;
}

function getSrc(evnt) {

   var theEvent = evnt ? evnt : window.event;
   var theSrc = theEvent.target ? theEvent.target : theEvent.srcElement;
   alert(theSrc.src);
}

//]]>
</script>
</head>
<body>
<div id="div1">
<img src="fig0902.png" alt="" />
</div>
</body>
</html>
```

多年来，对象检测手段已被证实是有效的。即使一个浏览器现在无法支持某个对象，未来还可以再试一试，但只要它有可能不支持，应用程序中就需要使用对象检测功能。

提示

在示例 6.2 中使用了一个以“filter”开头的、不常用的 CSS 设置，它是在老版本的 IE 中启用 PNG 透明显示功能的方法，因为这些浏览器无法支持 alpha 透明性。它对于这个 JavaScript 示例而言并不重要，它所体现的浏览器差异超出了 JavaScript 的范畴。

不过，对象检测虽然有效，但也不是绝对可靠的。

6.3.2 对象检测失败的场合

对象检测要获得成功，被检查的对象必须满足两个条件：它是已定义的，它是非空的。在前面的几节中，都是应用于事件对象的示例，在支持 DOM Level 2 事件处理机制的浏览器中都已经定义和设置了。

不过，即使对象已经定义但仍然未赋值，对象检测也会遇到问题。换句话说，该对象虽然属于浏览器支持的对象模型，但还没有赋值。如果对象已定义但未设置，那么像下面这样的条件语句将会执行失败：

```
if (someobject) {...}
```

当你检查浏览器支持哪种透明性标准时，就会遇到这类问题。

Opera、Firefox 和 Safari 的近期版本，都支持 CSS 2 中定义针对样式对象的 opacity 属性。使用以下语法，可以将元素的 opacity 属性设置成 50%：

```
document.getElementById("div1").style.opacity=0.5;
```

opacity 属性的取值范围是 0（透明）到 1（完全不透明）。

但是，过去 IE 只在特定的 filter 中实现了透明性机制：

```
document.getElementById("div1").style.filter='alpha(opacity=' + 50 + ')';
```

它并不是通过 opacity 属性来设置透明性，而是通过为 filter 赋值（在此指定的是 alpha opacity）的方法实现。另外，其透明性取值范围是 0%～100%。

要怎样才能测试浏览器的这一差异以便知道应该如何赋值呢？你可以测试浏览器是否支持 filter 属性：

```
if (div1.style.filter) { ... }
```

不过，如果浏览器支持 filter 属性但并没有设置其值会发生什么？正如我们在第 12 章中将会看到的那样，如果在样式表中设置了一个样式属性，那么即使它未被赋值，也可以通过 JavaScript 访问这个样式属性。首先，你必须在脚本中设置该属性（或者放在内联的 style 属性上）以确保该样式对象能够被访问。

在该示例中，如果在前面的代码中未对透明性 filter 赋值，那么 filter 对象虽然是已定义的，但它的值是 null。这个条件判断语句将失败，即使是在支持 filter 对象的 IE 浏览器中也是如此。

对象检测还有另外一种做法，它对于对象之前还未设置的情况尤为有用，它将检测该对象是否定义，而不管它的值是不是 null，即：

```
if (div1.style.filter != undefined) { ... }
```

这样的做法在 Firefox 和 IE 中都能够正常工作，但在 Opera 或 Safari 中则不行。其原因是不管是在 Opera 中还是在 Safari 中，当你直接访问未定义的属性时都将返回 undefined。

因此，我们还可以通过 typeof 函数来测试对象的类型，以判断该对象是否未定义，即：

```
if (typeof(div1.style.filter) != "undefined") { ... }
```

这样的代码可以在 Firefox、IE 和 Safari 中正常运行，但在 Opera 中仍然有问题，因为 Opera 中虽然提供了 filter 属性，但和 IE 不同，它看起来不是用来设置透明性的。在 Opera 中运行这行代码时，肯定会返回 false，这意味着在该浏览器上无法使用透明性。

此时的问题是我们没有对正确的对象进行测试。我们在此应该测试的是 style.opacity 的支持。示例 6.3 展示的是同时测试 style.opacity 和 style.filter 的方法。如果在被测试的浏览器中定义了与透明性相关的对象，那么它将返回 true。这样的检测可以成功地在 Opera、Firefox、Safari 中正常运行，但正如我们能够想到的那样，在 IE 中这将失败。

示例 6.3　测试 style.filter 和 style.opacity 对象是否存在

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
```

```
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>object detection</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<meta http-equiv="X-UA-Compatible" content="IE=EmulateIE7" />
<style type="text/css">
div {
          position: absclute;
          top: 30px;
          left: 50px;
    }
#div1 img {
filter:
progid:DXImageTransform.Microsoft.AlphaImageLoader(src=fig0902.png,
sizingMethod='scale');
}

</style>
<script type="text/javascript">
//<![CDATA[

window.onload=function() {

  var divElement = document.getElementById("div1");

  // 测试 filter 对象是否存在
  var tst1 = (divElement.style.filter) ? "filter exists" : "filter does not
exist";
  alert(tst1);

  // 测试 filter 对象是否存在
  var tst2 = (typeof(divElement.style.filter) !== "undefined") ? "filter
exists" : "filter does not exist";
  alert(tst2);

  // 测试 filter 对象是否存在
  var tst3 = (divElement.style.filter !== undefined) ? "filter exists" :
"filter does not exist";
  alert(tst3);

  // 测试 opacity 对象是否存在
  var tst4 = (divElement.style.opacity) ? "opacity exists" : "opacity does
not exist";
  alert(tst4);
  // 测试 opacity 对象是否存在
  var tst5 = (typeof(divElement.style.opacity) !== "undefined") ? "opacity
exists": "opacity does not exist";
  alert(tst5);
}
```

```
//]]>
</script>
</head>
<body>
<div id="div1">
<img src="fig0902.png" alt="" />
</div>
</body>
</html>
```

对象检测的技巧在于知道要检测哪个对象，以及知道如何“表示”要检测的对象（确保检测的是对象是否定义，而不是是否存在）。

当你在第 13 章中再次看到这个示例时，你会发现有时对象是否存在对于创建跨浏览器兼容的对象是没有意义的。不过，当浏览器不再支持通常会支持的属性时，它也并不是毫无价值的。

6.3.3 DOCTYPE、X-UA-Compatible 和 Quirks 模式

在第 1 章中的示例 1.2 里，Web 页面代码的第一行中有一个关于 XHTML 1.1 的 DOCTYPE 声明。虽然本书中所有示例都采用了.html 扩展名，但指定 XHTML DOCTYPE 会对 JavaScript 如何处理产生影响。这些都和名为 quirks 模式的概念有关。

当一些浏览器在改进它们对 HTML 标签和 CSS 标准的支持时，浏览器软件开发人员仍然希望能够保持向后兼容，使其能够支持基于老版本浏览器创建的 Web 页面。实现这一目标的其中一种方法是提供 quirks 模式（这个名字很恰当），以老版本浏览器的、非标准的、行为“奇怪”的方式呈现 Web 页面。浏览器是以 quirks 模式还是标准模式呈现页面内容，取决于 DOCTYPE 的设置。

该模式很重要，因为对 JavaScript 的支持就是“标准”之一，当页面以标准模式而非 quirks 模式呈现时，CSS 和实际页面中的标签都会以不同的形式解释。当从 quirks 模式改成标准模式时，IE 是受影响特别明显的浏览器。在 quirks 模式中，IE 假定 HTML 标签、CSS、JavaScript 都是基于老版本的，如 IE 5.x 或更早版本。

对于绝大多数浏览器而言，XHTML 转换和精确的 DOCTYPE 设置都会使其切换到标准模式下，无论 XHTML 代码是否添加了可选的 XML 声明：

```
<?xml version="1.0" encoding="UTF-8" ?>
```

Firefox、Safari 和 Opera 会把这个 XML 声明解释为标准模式，而 IE 会将这种声明解释为 quirks 模式。

对于绝大多数浏览器而言，指定 HTML 4.01 的 DOCTYPE 声明以及最新的 HTML 5 的 DOCTYPE 声明，都会使其切换到标准模式，但在本章的所有示例中，我们都将声明为 XMHTL 1.1。

DOCTYPE 不是页面中唯一的会改变浏览器处理 JavaScript 方式的非脚本元素。从 IE 8.0 的第一个 beta 版本开始，就引入了一个新的名为 X-UA-Comatible 的 meta 标签，它专门用来使 IE 8.0 切换成 quirks 模式、IE 7.0 标准模式或 IE 8.0 所支持的其他新标准。

6.3.4 阻止向后兼容：IE 8.0 中的 Meta 标签 http-equiv

使用对象检测，以及确保我们的代码是基于标准和产业规范开发的，就能够确保这些代码能够在将来和过去的浏览器中正常运行。不过在某些浏览器中是无法确保这一点的，那就是 IE，特别是新发布的 IE 8.0。

微软已经尽力对其浏览器进行更新，使其支持标准，但在这个过程中，对于一些专门针对 IE 开发的功能可能无法保证向后兼容。之前说过，要将标准模式切换成 quirks 模式，可以按前面说过的方法添加标准的 DOCTYPE 声明。不过，一旦你已经使用了这些声明，就无法再次使用它们重新声明。

微软引入了一个新的 HTML meta 标签 http-equiv，它可以告诉 IE 8.0 是按 IE 8.0 的模式呈现页面，还是按 IE 7.0 的模式：

```
<meta http-equiv="X-UA-Compatible" content="IE=IE7" />
```

但是该标签也存在一些问题，那就是会使 IE 8.0 按 IE 7.0 的模式运作，而浏览器实际上或多或少已经做到了。从 IE 8.0 的第二个 beta 版本开始，微软就引入了第二个 meta 标签 EmulateIE7，它和 DOCTYPE 协作完成 IE 7.0 行为的模拟。

```
<meta http-equiv="X-UA-Compatible" content="IE=EmulateIE7" />
```

在第 13 章中，你会看到这个 meta 标签的本质，尽管本书中的绝大多数示例都能够在 IE 7.0 和 IE 8.0 中运行，但透明性无法实现。IE 8.0（至少在其第一个 beta 版本中）已经去除了其私有的、用来设置透明性的 filter 属性，但仍然没有实现 CSS 2 中的透明性样式。要实现透明性，就必须在文档头中添加这个 meta 标签。

在 IE 8.0 中默认是支持标准模式的，但这只是针对因特网中的网页而言。对于内部网的页面，其默认设置是使用 IE 7.0 的标准，除非在浏览器选项中禁用了 compatibility 模式（微软的术语）。该功能还不稳定，在 IE 8.0 的最终版本中会怎么做还不得而知。

6.4 知识测验

1. 编写一段代码，用来检查某个变量的值，不得使用针对某种浏览器的调试程序。
2. CSS 属性 text-shadow 的历史很有趣。它曾经被添加到了 CSS 2 标准中，但却没有浏览器实现它，然后在 CSS 2.1 标准中又被删掉了。不过，这时却开始有浏览器实现了它，现在它又已经添加到了 CSS 3 标准中。在写作本书时，4 个主要浏览器中

已经有两个实现了该属性。你是否可以通过对象检测来测试你的浏览器是否支持这个样式属性？编写一个跨浏览器兼容的代码，对一个已有的 header 元素设置这个 CSS 属性。

6.5 测验答案

1. 要检查一个变量的值，最快的方法是使用 alert 对话框：

```
// 检查一些变量
alert(firstName);   // 其值要么已设置，要么为 null，要么为 undefined（未定义）
```

2. 你首先会想到的方法可能是检查是否存在 textShadow，这是该 CSS 属性脚本可访问的名称，它是由 style 对象实现的，但它和之前的 opacity/alpha 示例一样，不是永远可用的方法。

最简单的方法是直接设置 textShadow 的值，不管浏览器是否支持，如果浏览器支持它则会使该元素的字体改成带阴影的效果，如果浏览器不支持，那么仍然会设置该值，但它将被忽略：

```
var headerElement = document.getElementById("pageHeader");
headerElement.style.textShadow="#ff0000 2px 2px 3px";
```

第 7 章

捕获事件

在 Web 页面中出现特定行为时将触发事件，包括当页面载入完成时。在 JavaScript 中，你可以通过事件句柄这一“钩子”来捕获事件，然后可以通过它来调用一个函数或执行一些其他 JavaScript 代码。

JavaScript 中的事件同样是和 Web 页面元素关联的，而不是该语言本身所固有的。在前面的章节中，应用程序正是通过 onload 事件句柄实现在页面载入完成时调用一个函数。这时触发的事件就是 load，它是和浏览器的 window 对象关联的。当页面载入到浏览器窗口后，将触发 onload 事件句柄。

由于事件处理是 JavaScript 中在近几年变化最大的部分，而且也是现有浏览器中仍然存在差异的主要方面之一，这使得与事件相关的工作变得更有挑战性。幸运的是，这些差异很容易克服。不过在我们介绍跨浏览器兼容的事件处理之前，还是先快速了解一下事件，以及它们通常在什么时候被触发。

7.1 事件

事件相对而言还是比较容易理解的。W3C（World Wide Web Consortium）将事件分成了 3 种不同类型：用户界面事件（鼠标、键盘触发的）、逻辑事件（一个处理的结果）和变化事件（修改文档的操作）。表 7.1 列出了一些基本的事件，提供了它们的描述以及受影响的对象。

表 7.1 事件和其影响的对象

事　　件	描　　述	受影响的对象
abort	当图像被禁止载入时	图像元素
blur，focus	当对象失去或获得焦点时	适用于 window 和表单元素

续表

事　件	描　述	受影响的对象
change	当选择项发生变化时	适用于表单元素，当该元素失去焦点后，值发生了变化
click，doubleclick (dblclick)	单击鼠标，双击（快速连续点击两次）鼠标	绝大多数页面元素
contextmenu	单击鼠标右键（调出右键快捷菜单）	Web 页面文档
error	当页面或图像无法载入时	Web 页面文档和图像
keydown，keyup，keypress	按下一个键，松开一个键，按下后再松开一个键	Web 页面文档和特定的表单元素
load，unload	当图像或页面载入完成时，页面失去焦点	Web 页面文档和图像（仅限于载入）
mousedown，mouseup	按下鼠标键，松开鼠标键	绝大多数页面元素
mouseover，mouseout	将鼠标移到某个元素上面，将鼠标从某个元素上移开	绝大多数页面元素
mousemove	移动鼠标	绝大多数页面元素
reset	表单被重新设置	针对表单
resize	窗口或帧的大小被调整	窗口或帧
select	选中某个文本	表单文字输入框
scroll	当对象收到滚动操作时	窗口、帧，或者超出自动设置的高度或宽度范围的对象（这时将显示出滚动条）
submit	表单已提交	针对表单

在表 7.1 中，还有些私有的事件没有被列出，我们在后面还会讲到它们。

事件句柄的名称只是在事件名称的前面加上一个前缀“on”，如 onload、onblur、onsubmit 等。在较早的应用程序中，事件的命名规范通常采用的是 CamelCase 风格，也就是事件名称的第一个字母将改成大写字母表示（onLoad、onSubmit 等）。不过，由于许多事件句柄也是页面元素的属性，而 XHTML 是不允许元素属性采用大写字母的，因此早期的命名风格就不再使用了。

7.2　0 级事件处理

我们再重复一下，事件句柄将使用如下所示的语法：

```
onload
```

在此事件句柄的名称是以“on”开头的，后面跟着的是事件类型，load、click 等。

事件是和页面元素相关联的，你可以将它们以属性的形式添加到元素中。以属性的形式为 HTML 元素添加属性的方法，有时称之为内联模式（inline model）或内联注册模式。

你可以直接将句柄的 JavaScript 代码写在这里：

```
<body onload="var i = 23; i *= 3; alert(i);">
```

不过，更常见的做法是调用一个函数：

```
<body onload="calcNumber();">
```

注意，在这两个场景中 JavaScript 代码都是放在引号里面的。即使你使用的是 HTML，并且没有将其他属性放到引号中，也应该确保将 JavaScript 代码放到引号中。

你也可以像访问每个对象的属性那样直接访问事件句柄。下面这行代码将为 window 对象的 onload 事件属性指派一个函数：

```
window.onload=calcNumber;
```

如果想禁用事件处理，那么可以将事件句柄的值赋为 null。这种通过对象属性将一个函数指派为事件句柄的做法有时称为传统模式或传统注册模式。

示例 7.1 针对 window 的载入事件演示传统模式和内嵌事件模式的使用方法。它们是相同的事件，使用了相同的事件句柄，因此你将会看到两次 alert 对话框。

示例 7.1　使用传统和内嵌事件句柄捕获 load 事件

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>Event Handling</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript">
//<![CDATA[

function helloMsg() {
   var helloString = "hello there";
   alert(helloString);
}

function helloTwice() {
   var helloString = "hi again";
   alert(helloString);
}
```

```
window.onload=helloTwice;
//]]>
</script>
</head>
<body onload="helloMsg();">
<p>Some content</p>
</body>
</html>
```

第一个使用传统模式指定的事件句柄将会弹出一个 alert 对话框，上面显示着“hello there”。不过，使用内嵌模式指定的事件句柄并没有显示任何信息。

只显示了一个 alert 对话框的原因是在 DOM Level 0 事件模型中，任何对象只允许指定一个事件句柄；因此指定的函数不会附加上去。如果你想对针对某一特定对象的某个事件指定多个函数，则需要在事件句柄代码中列出它们，如果使用内嵌模式的话就是：

```
<body onload="helloMsg(); helloTwice()">
```

如果使用传统模式的话就是：

```
function helloMsg() {
    var helloString = "hello there";
    alert(helloString);
    helloTwice();}
```

内嵌事件可以在所有浏览器中正常应用，但仍然应该限制使用。如果你为 HTML 元素添加一个事件，并且需要修改 JavaScript 的函数名或行为，那么由于你当时指定事件是在不同页面中将某个函数指定成事件句柄的，因此需要找到每个页面、每个元素，手动完成这些修改。对于任何网站，即使是最简单的，这种做法都是应该避免的。更好的方法是使用传统模式，它也可以在所有现代浏览器中正常工作。

提示

最好的方法是使用最新的事件句柄，其理由在本章后面会详细说明。

对于如表单提交之类的事件，你可能希望在执行完事件句柄之后不再执行该事件的默认行为。例如，如果在某个表单域中输入的值是无效的，那么你将不希望继续将表单提交给服务器。对于这种类型的事件句柄行为，你可以在事件句柄函数中返回一个 false 值：

```
function doSomething() {
   // 完成某些操作
   return false;
}
```

从事件句柄函数中返回 false，意味着要求浏览器停止执行事件行为。在本章稍后介绍到表单处理时，你会看到这一过程。

对于许多事件而言，知道事件已触发就足够了，但对于如 click、mousedown 之类的事件而言，你还需要知道一些其他相关信息，包括该事件触发时鼠标的位置等。其问题是，这类信息如何获得呢？稍后你会看到这对于不同浏览器可能存在着不同，不过该问题是能解决的。

7.2.1 Event 对象

传统的事件处理也称为 DOM Level 0 事件处理，因为它基于浏览器中对 DOM 的早期实现，也就是众所周知的 Level 0。你可以将 DOM Level 0 中的事件分成两个阵营：早期的 Netscape（也是现在的 Mozilla/Firefox，以及 Opera 和 Safari）和 IE。在很大程度上，你可以获得能够正常处理各种类型事件的交互式页面，你需要一些技巧。其中一个技巧就是如何访问 Event 对象。

Event 对象是和所有事件相关的。它有一些用来提供事件相关信息的属性，如 Web 页面中鼠标点击的位置。Firefox 中的 Event 对象要比 IE 中的简单得多，虽然也提供了许多方法。不过，访问该对象的方法则有很大不同。

IE 将 Event 视为 window 对象的属性。当处理事件时，将通过程序访问 window 对象，其所包含的数据也会相应地进行填充。示例 7.2 将在用户点击鼠标键时访问 IE 中的 Event 对象，并且在屏幕的(x,y)位置上显示一个弹出窗口。

示例 7.2 访问 IE 中的 Event 对象

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
<head>
<title>X/Y Marks the Spot</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript">
//<![CDATA[

function mouseDown() {
  var locString = "X = " + window.event.screenX + " Y = " + window.event.screenY;
  alert(locString);
}

document.onmousedown=mouseDown;
//]]>
</script>

</head>
<body>
</body>
</html>
```

这种获取 Event 对象的方法一直可以沿用到 IE 8.0，在早版本的 IE 中也不例外。不过，

如果在 Firefox、Opera 或 Safari 中以这种方法访问，那么应用程序将出错。在基于 Netscape 的浏览器（如 Firefox、Mozilla、Opera 和 Safari）中，获取 Event 对象的方法是不同的：它将作为函数的一部分传入。在本例中，要在如 Firefox 之类的浏览器中正常运行，该函数应写为：

```
function mouseDown (theEvent) {
  var locString = "X = " + theEvent.screenX + " Y = " + theEvent.screenY;
  alert(locString);
}
```

处理这些跨浏览器差异的方法之一是检查传入函数的 Event 对象是否已经实例化。如果是，那么将这个 Event 对象赋给一个局部变量；否则，将假定 window.event 为该事件，并将其赋给这个局部变量。示例 7.3 展示了示例 7.2 的跨浏览器兼容版本。

示例 7.3　针对示例 7.2 的跨浏览器兼容版本

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>Event Handling</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript">
//<![CDATA[

function mouseDown(nsEvent) {
  var theEvent = nsEvent ? nsEvent : window.event;
  var locString = "X = " + theEvent.screenX + " Y = " + theEvent.screenY;
  alert(locString);
}

document.onmousedown=mouseDown;
//]]>
</script>
</head>
<body>
<p>Some content</p>
</body>
</html>
```

示例 7.3 还展示了很有用的三元操作符，它曾经在第 2 章中介绍过。在本例中，该操作符用来判断 nsEvent 对象是否已定义。如果已定义，那么就将其赋给应用程序中的变量；否则将把 window 对象的 event 属性值赋给应用程序中的变量。

当你拥有了 Event 对象之后，你就可能需要基于它完成一些功能。在 Event 提供的属性中，以下是跨浏览器兼容的：

- altkey　布尔值，用来表示事件触发时 Alt 键是否被按下；

- clientX　事件触发时客户端的当前 x 坐标；
- clientY　事件触发时客户端的当前 y 坐标；
- ctrlKey　布尔值，用来表示事件触发时 Ctrl 键是否被按下；
- keyCode　当前按键的代码（数字）；
- screenX　事件触发时屏幕的 x 坐标；
- screenY　事件触发时屏幕的 y 坐标；
- shiftKey　布尔值，用来表示事件触发时 Shift 键是否被按下；
- type　事件类型。

本书后面讲到动态页面的创建时，将更详细地介绍客户端（client）和屏幕（screen）两个体系。

对 Ctrl 键进行测试，是确定用户是否按下了一个特定键盘序列（用来触发不同的操作集）的好方法。另外，如果你想创建如幻灯片播放之类的应用时，如果想把按钮 N 和 P 解析成向前和向后翻页，那么按键代码就十分有用。

在非跨浏览器兼容的属性中有一个 fromElement，它只能用于 IE 中，而在 Mozilla/Firefox 的事件处理中对应的属性是 relatedTarget。这些属性用来获取鼠标移动到的对象。而用来获取鼠标从哪个对象移开的属性是 aretoElement 和 currentTarget（分别用于 IE 和 Mozilla）。当需要支持拖放操作时，这些属性是十分有用的。

要解决与事件处理相关的浏览器间的差异，可以使用另外一个三元操作符：

```
var oldElement = theEvent.fromElement ? theEvent.fromElement :
theEvent.relatedTarget;
```

IE 中的 srcElement 和 Mozilla 等浏览器中的 target 属性是用来表示接收到事件的对象的：

```
var theSrc = theEvent.target ? theEvent.target : theEvent.srcElement;
```

cancelBubble、preventDefault 和 stopPropagation 则是一些用于处理事件冒泡操作的非跨浏览器兼容属性，我们接下来就将介绍它们。

提示

关于 preventDefault 将在第 8 章中介绍。

7.2.2　事件冒泡

当你在一个 Web 页面中点击鼠标时，可能不是点击某个文档，而是点击一个链接或者一个 div 元素。在绝大多数情况下，你无须关心包含该链接的容器元素，因为你通常只会为一个元素设置一个事件句柄。但如果你为多个嵌套的元素设置了相同的事件句柄，

会发生什么呢？它们将以什么样的顺序触发？如果你想使得一次只影响一个元素，那么如何保存触发事件句柄的事件呢？

要管理元素堆栈中的事件，其中一个方法就是众所周知的事件冒泡。在事件冒泡中，最内部的元素将首先触发该事件，然后是堆栈内的下一个元素触发该事件，以此类推，直到最外面的元素。如果事件句柄被指定给所有的元素，那么这些事件将依次被触发。

在示例 7.4 中，该 Web 页面中有两个 div 元素，一个位于另一个的内部。它们和 document 对象都指定了针对 mousedown 事件的事件句柄。

示例 7.4　多元素的事件冒泡行为

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>Event Handling</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript">
//<![CDATA[

// 为 div 元素设置事件句柄
window.onload=setupEvents;

function setupEvents() {
  document.getElementById("first").onmousedown=function() {
    alert("first element event");
  }

  // 第二个事件句柄
  document.getElementById("second").onmousedown=function () {
     alert("second element event");
  }

  document.onmousedown=function() {
    alert("document event");
  }
}

//]]>
</script>
</head>
<body>
<div id="first" style="padding: 20px; background-color: #ff0; width: 150px;
">
  <div id="second" style="background-color: #f00; width: 100px; height:
100px;
border: 1px dashed #000">
```

```
   </div>
</div>
</body>
</html>
```

如果点击页面中最里面的 div 元素，不管是在哪种浏览器上都将弹出 3 个 Alert 对话框，顺序显示出以下消息：

1．Select element event

2．First element event

3．Document event

这些消息是以堆栈顺序显示的，首先是最里面的 div 元素，然后是外面的 div 元素，最后是 document 元素。该事件首先从堆栈最下面开始，然后不断冒泡，并按顺序触发其事件句柄。图 7.1 展示了这一处理过程。

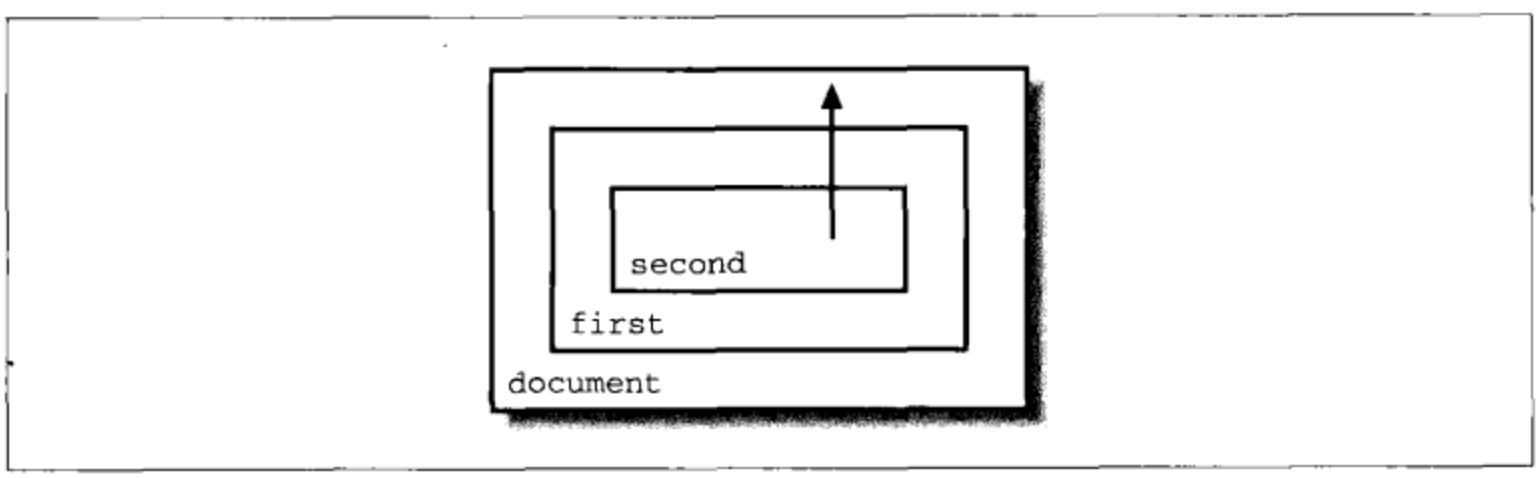


图 7.1　在页面元素中的事件冒泡过程

如果你有一个元素堆栈，并且只希望一个元素触发该事件句柄，那么你可以取消事件冒泡机制。如果在 IE 中要取消一个事件冒泡，可以使用 IE 中事件的 cancelBubble 属性；对于 Mozilla 而言，则应该使用事件的 stopPropagation 方法。你可以先检查 stopPropagation 方法是否存在，然后根据其结果确定使用哪种方法：

```
function stopEvent(evnt) {
   if (evnt.stopPropagation) {
     evnt.stopPropagation();
   } else {
     evnt.cancelBubble = true;
   }
}
```

注意，我们检查的是 stopPropagation 函数是否存在，而不检查 cancelBubble 函数是否存在，这是因为如果 cancelBubble 的值为 false 或该属性不存在，都将返回 false 值。

如果你添加一个对该函数的调用，以结束针对示例 7.4 中第一个 div 元素对 click 事件句柄的调用，那么前两个 alert 对话框仍然会显示，但是最后一个针对 document 事件的

alert 对话框将不会影响，因为在事件到达堆栈最顶部之前，事件已经被取消了：

```
document.getElementById("first").onmousedown=function(evnt) {
  var theEvent = evnt ? evnt : window.event;
  alert("first element event");
  stopEvent(theEvent);
}
```

现在我们已经访问了 event 对象，但我们如何访问该事件所针对的对象呢？这里关心的内容是 this。

7.2.3 事件句柄和 this

在一个事件句柄函数或对象方法中，在 JavaScript 中访问包含它的元素的属性时，可以使用 this 关键字。

this 关键字表示的是当前调用的函数或方法的所有者。对于一个全局函数而言，它表示的就是 window 对象。对于一个对象的方法而言，它表示的就是该对象实例。而在一个事件句柄中，它表示的就是接收到该事件的元素。

在下面这个针对 window 的 onload 事件的事件句柄函数中，就使用了 this 关键字来访问 window 对象的 status 属性：

```
window.onload=setupEvents;

function setupEvents() {
  alert(this.status);
}
```

在示例 7.4 中，出现在第一个 div 元素的 onclick 事件句柄函数中的 this 指向的就是 div 元素本身：

```
document.getElementById("first").onmousedown=function() {
  alert(this);  // 在 Firefox 中将输出"[object HTMLDivElement]"
  alert("first element event");
}
```

使用 this 是访问事件触发后表单值的简单方法，你不需要提供文档的路径、表单名称、字段名称等。示例 7.5 中为一个 form 元素的 blur 事件指定了一个 onblur 事件句柄，它就将使用 this 来访问 form 元素的 value 属性。

示例 7.5　在事件句柄中使用 this 关键字

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>this</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript">
```

```
//<![CDATA[

window.onload=setObjects;

function setObjects() {
   document.getElementById("personData").firstName.onblur=testValue;
}

function testValue() {
   alert("Hi " + this.value);  //表单域的值
}
//]]>
</script>
</head>
<body>
<form id="personData" action="">
<p>
First Name: <input type="text" id="firstName" /><br /><br />
Second Name: <input type="text" id="secondName" />
</p>
</form>
</body>
</html>
```

当你输入姓并按 Tab 键或在第二个字段上单击鼠标左键时，将弹出一个 alert 对话框，并显示出第一个表单域的值。

提示

这里对于 this 关键字的使用，只是针对 DOM 对象和自定义对象的一部分功能，在后续的章节中将对其做更完整的介绍。

到此为止，我们介绍的事件模型都聚焦于向上的事件冒泡，也就是从最里面的元素向最外面的元素冒泡。对于堆栈内元素的事件处理，还有一种被称为事件捕获（event capturing）或 cascade-down 的事件处理机制。对于前面这个包含 3 个元素的示例而言，事件将从最外面的元素开始触发：首先触发的是 document 元素的事件句柄，然后是第一个 div 元素，最后是第二个 div 元素。

在下一小节中介绍的 W3C 事件模型将支持这两种事件处理方式：事件捕获和事件冒泡。

7.3 DOM Level 2 事件模型

直到现在为止，我们展示的事件处理过程是老版本的、DOM 之前的事件模型。许多主流的浏览器仍然提供了这种老模型的支持，但 JavaScript 开发人员被鼓励采用新的、基于规范的事件模型，也就是被称为 DOM Level 2 的事件模型。

老事件模型和新的 DOM Level 2 事件模型之间，最主要的区别在于：

- 新事件模型并不依赖于特定的事件来处理属性；
- 你可以对任何一个对象的任何一种事件注册多个事件句柄函数。

用来代替事件句柄属性的是每个对象提供的 3 个方法：addEventListener、removeEventListener 和 dispatchEvent。第一个方法用来添加一个事件监听器，第二个用来删除一个事件监听器，第三个用来分发一个新的事件。

addEventListener 的语法是：

```
object.addEventListener('event',eventFunction,boolean);
```

如 click 或 load 之类的事件是其第一个参数；第二个参数是指定的事件句柄函数；第三个参数用来指定事件是以 cascade-down 模式或冒泡模式处理的。如果第三个参数的值是 false，那么这个事件监听器将以冒泡模式处理，就像老事件模型的示例那样；否则，将把这个事件监听器改成事件捕获模型。

在示例 7.6 中，使用的是一个包含一个元素的表单，并在页面中添加了一个提交按钮，针对这个提交按钮、表单以及 document 的 click 事件都将被捕获。为该事件指定的句柄函数支持 cascade-down（cascadeDown）和冒泡（bubbleUp）两种模式。在这个句柄函数中，将把 this 关键字传给一个 alert 对话框，以输出当前接收到事件的对象是哪个。当页面卸载时，将清理该事件处理。

示例 7.6　用 DOM Level 2 事件句柄捕获事件

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>Capture/Bubble</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript">
//<![CDATA[

function cascadeDown(evnt) {
   alert("Capturing: " + this);
}

function bubbleUp(evnt) {
   alert("Bubbling: " + this);
}

window.onload=setup;

function setup(evnt) {
```

```
    // 事件捕获
    document.addEventListener('click',cascadeDown,true);
    document.forms[0].addEventListener('click',cascadeDown,true);
    document.forms[0].elements[0].addEventListener('click',cascadeDown,true);

    // 事件冒泡
    document.addEventListener("click",bubbleUp,false);
    document.forms[0].addEventListener("click",bubbleUp,false);
    document.forms[0].elements[0].addEventListener("click",bubbleUp,
false);
}

//]]>
</script>
</head>
<body>
<form style="background-color: #f00; width: 100px; height: 100px; padding:
10px"
action="">
<p>
    <input type="submit" value="Submit" /><br />
</p>
</form>
</body>
</html>
```

单击该按钮将生成 6 个 alert 对话框。在 Firefox 中，将按顺序输出以下值：

```
Capturing: [object HTMLDocument]
Capturing: [object HTMLFormElement]
Capturing: [object HTMLInputElement]
Bubbling: [object HTMLInputElement]
Bubbling: [object HTMLFormElement]
Bubbling: [object HTMLDocument]
```

当应用程序捕获到第一次处理的事件时，将依次调用 document、表单和提交按钮的事件句柄。这在你想采用 cascade 模式时十分有用，它将首先执行元素堆栈中最外层的元素，然后依次处理里面的元素，直到处理到最里层的元素为止。

接下来将进入冒泡阶段，这时的处理顺序将从提交按钮开始，然后是表单，最后是 document 元素，也就是自下向上地处理。这种事件执行顺序在你需要从最里层的元素“冒泡”到最外层时十分有用。

如果你想停止堆栈中事件的执行过程将如何做？当你想停止事件执行时，可以在函数中调用 stopPropagation 方法：

```
function cascadeDown(evnt) {
    ...
    evnt.stopPropagation();
}
```

如果要彻底删除一个事件监听器，则可以使用 removeEventListener 方法：

```
document.forms[0].elements[0].removeEventListener("click",cascadeDown,true);
```

当事件已经开始处理时，该方法是无法正常运作的，不过当按钮下一次被单击时，针对这个表单内元素的事件句柄函数将不会被执行。不过针对表单和 document 的事件句柄函数仍然会执行。

addEventListener 和 removeEventListener 的概念和执行过程是令人恐怖的，有一件事情除外：微软只支持自己的事件处理模型。即使在最新的 IE 8.0 中，该公司的浏览器仍然只支持其自己创建的事件模型。

因为微软发布 IE 7.0 花了好多年的时间，而且发布 IE 8.0 也花费了近 3 年的时间，因此微软的浏览器可能不会很快提供对 W3C 事件模型的支持。实际上，我怀疑该公司永远不会提供支持，因此我们需要考虑 workaround（工作区）。

在 IE 中，与 addEventListener 和 removeEventListener 方法相似的是 attachEvent 和 detachEvent。attachEvent 的语法是：

```
object.attachEvent("eventhandler", function);
```

detachEvent 的语法和 attachEvent 相同：第一个参数是事件句柄，第二个是其函数。

虽然在 IE 中添加事件句柄的方法和 Firefox 之类的浏览器不同，但解决这种差异还是比较容易的。示例 7.7 演示了如何提供一种处理指定 document 元素的 click 事件的跨浏览器兼容方法。

示例 7.7　跨浏览器兼容的事件处理

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>Capture/Bubble</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript">
//<![CDATA[

function clickMe(evnt) {
    var eventTarget = evnt.target ? evnt.target : evnt.srcElement;
    alert(eventTarget + " " + evnt.type);
    var canBeCanceled = evnt.cancelable ? evnt.cancelable : "NA";
    alert("Can be canceled? " + canBeCanceled);
    var bubbleEvent = evnt.bubbles ? evnt.bubbles : "NA";
    alert("Bubbling? " + bubbleEvent);
    var theTime = evnt.timeStamp ? evnt.timeStamp : "NA";
    alert(theTime);
}
```

```
window.onload=setup;
window.onunload=cleanup;

function setup(evnt) {
   var evtObject = document.getElementById("clickme");

   // 检查对象模型
   if (evtObject.addEventListener) {
      document.addEventListener("click",clickMe,false);
   } else if (evtObject.attachEvent) {
      evtObject.attachEvent("onclick", clickMe);
   } else if (evtObject.onclick) {
      evtObject.onclick=clickMe;
   }
}

// 清理
function cleanup() {
   var evtObject = document.getElementById("clickme");
   if (evtObject.detachEvent) {
       evtObject.detachEvent("onclick",clickMe);
   }
}

//]]>
</script>
</head>
<body>
<div id="clickme" style="background-color: #ff0; width: 200px; height: 200px;
padding: 20px">
<p>Click Me</p>
</div>
</body>
</html>
```

在这段代码中，首先检查了浏览器是否支持 addEventListener 方法。如果支持，则使用它来添加事件句柄。如果不支持，而且提供了 attachEvent 方法的话就使用该方法。如果都不支持，那么将使用老版本的 0 级事件处理机制。

和老的、传统事件处理机制不同，我们将向 attachEvent 方法或 addEventListener 方法传入一个 Event 对象。不过，Event 对象所支持的属性在不同浏览器中是不同的。

在这个事件句柄函数中，将访问和输出其 cancelable（该事件是否能够被取消）、timestamp、bubbles 等属性。不过，这些属性在 IE 中大多都无法支持，因此分别使用了一个三元操作符来检查，并且如果发现无法使用则将其设置为字符串“NA”（也就是无法应用）。这些属性在 Firefox 和 Safari/WebKit 中都能够正常使用，在 Opera 中除了 timestamp 之外也都能够正常使用。在这里还访问了针对 IE 模型的 scrElement 属性。

该示例说明尽管浏览器之间的区别很明显，但也是可以解决的。

不幸的是，并不是所有差异都能很简单地解决的。在 IE 中，contextual（上下文）对象、this 都是针对 window 对象的，与对象和事件无关。而在 DOM Level 2 事件处理模型中，这些都是和接收到事件的对象相关联的。

对于微软的模型而言，另一个要考虑的因素是它会为每个事件句柄留出相应的内存，即使你刷新页面，也会为每次连续的页面载入留出相应的内容，这样一会儿就会消耗大量的内存。为了避免使用过多的内存，你需要跟踪 window 的 unload 事件，然后调用 detachEvent 方法清理每个事件。这将使内存管理系统在页面卸载时释放相应的内存空间。在示例 7.7 中，我们为 window.onunload 事件句柄指定了一个名为 cleanup 的函数，就是用来完成这一工作的。不过 window 的 unonload 事件中 addEventListener 方法使用的内存是无须清理的。

提示

IE 中的内存问题，在 IE 8.0 中可能被解决。不过，由于在老版本的 IE 中仍然存在这些内存方面的问题，因此最好保留 cleanup 代码。

正如前面介绍的 Event 对象一样，这在不同的事件模型实现中也是不同的。Event 对象和事件支持的属性中也存在着差异。

下面列出的是 Event 的属性列表；它们是否已设置，取决于事件的类型。并不是所有浏览器都支持所有的事件属性；如果某个属性不被支持，那么当访问该属性时将返回 undefined 值：

- altkey　Alt 键的状态（按下与否）；
- bubbles　是否会按 DOM 执行事件冒泡（非 IE 浏览器）；
- button　鼠标按键；
- cancelBubble　冒泡操作是否被取消；
- cancelable　事件是否能被取消；
- charCode　当前按下的字符键的 Unicode 值（对于 IE 而言，要使用 keyCode）；
- clientX　事件的水平位置；
- clientY　事件的垂直位置；
- ctrlKey　Ctrl 键的状态（按下与否）；
- currentTarget　事件针对元素的引用（对于 IE 而言，要使用 srcElement）；
- detail　与事件相关的细节信息；
- eventPhase　事件当前的处理阶段；

- isChar　事件是否生成一个字符；
- keyCode　当前按下的非字符串的 Unicode 值；
- layerX　如果元素采用的是绝对定位，那么该值为与当前层（元素）相关的 x 坐标；
- layerY　如果元素采用的是绝对定位，那么该值为与当前层（元素）相关的 y 坐标；
- metaKey　是否按下了 Meta 键[1]；
- pageX　相对于页面的 x 坐标；
- pageY　相对于页面的 y 坐标；
- screenX　相对于屏幕的 x 坐标；
- screenY　相对于屏幕的 y 坐标；
- shiftKey　Shift 键的状态；
- target　接收到该事件的原始对象；
- timeStamp　事件创建的时间戳；
- view　事件生成的 AbstractView（window 对象，取决于跨浏览器实现的 window 对象标准化方面的努力）；
- which　所按键的 Unicode 值，不管它是否为字符键。

要判断在每个事件中支持哪些属性，最简单的方法是使用脚本调试器分析 Event 对象。在第 6 章中，我们介绍了针对不同浏览器的不同脚本调试器。

在本小节前面介绍的事件，既是新事件系统所支持的，也是与 DOM 相关的，包括 keypress、click、鼠标相关事件、window 对象的 load 事件以及针对表单及表单元素的事件。

7.3.1　生成事件

当在 Web 页面执行某些操作时就会启动一个事件，可能是单击了某个按钮、单击了某个链接、做出了一个选择等。不过，有时你可能需要手动触发一个事件。

要触发针对 Web 页面或页面元素的事件，该事件必须是与该类元素相关的。例如，你可以触发表单中一个按钮的 click 事件，但无法触发表单文本输入框的 click 事件。要触发表单中一个按钮的 click 事件，可以使用 click 事件，并在方法中调用 click 对象：

```
<input type="button" id="someButton" value="Some Button" />
...
document.getElementById("someButton").click();
```

[1] 译者注：Meta 键是键盘上的特殊键，如 Window 窗口键等。

直接调用一个事情的理由之一是使一个文本框获得焦点，并将光标移到这个文本框上。在示例 7.8 中，当页面载入完成时，获得焦点的将是文本框 last-name（名字），而不是文本框 first-name（姓），而通常情况获得焦点的是后者。

示例 7.8　使一个文本框获得焦点，并将光标移到该文本框上

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>Focus</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript">
//<![CDATA[

window.onload=setObjects;

function setObjects() {
   document.getElementById("personData").lastName.focus();
}

//]]>
</script>
</head>
<body>
  <form id="personData" action="">
    <p>
First Name: <input type="text" name="firstName" /><br /><br />
Last Name: <input type="text" name="lastName" />
</p>
</form>
</body>
</html>
```

在下一章中，我们将关注创建动态表单验证的一些技巧，将光标移动到输入无效的字段上，并且直接在页面上强调显示出错误信息。

7.4　知识测验

1. 编写一段程序代码，使用 DOM Level 0 方法为 document 的 click 事件指定一个事件句柄函数。
2. 现在，使用更现代的 DOM Level 2 事件处理机制为 document 添加 click 事件句柄（不用考虑跨浏览器兼容问题）。
3. 如何使问题 2 中所写的代码能够安全地运行在所有浏览器上？

4. 对于为 document 对象指定的 onclick 事件句柄，如何知道是在屏幕的什么位置执行了单击操作？

5. 使用 DOM Level 2 事件系统，如何阻止从其他元素中冒泡上来的事件？

6. 将下面这段针对 DOM Level 0 的事件句柄改成能够跨浏览器兼容的 DOM Level 2 方法：

```
<body onload="functionCall();">
```

7. 编写一段 JavaScript 程序代码，捕获 document 对象的 keydown 事件，然后通过调用 document.writeln 函数打印出当前所按的键。

7.5 测验答案

1. 其解决方案是：

```
document.onclick=clickMe;
```

2. 其解决方案是：

```
document.addEventListener("click",clickMe,false);
```

3. 其解决方案是：

```
if (document.addEventListener) {
    document.addEventListener("click",clickMe,false);
} else if (document.attachEvent) {
    document.attachEvent("onclick", clickMe);
}
```

4. 如果你使用 DOM Level 0 事件处理系统，那么将无法使用 window 对象的 event 对象，也不能将其作为参数传给函数。对于 DOM Level 2 事件处理模型而言，event 对象将会传给事件句柄函数。你可以通过 event 对象访问其 screenX 和 screenY 属性。

5. IE 所支持的方法和绝大多数浏览器所支持的方法有所不同，因此你需要分别支持 IE 和其他浏览器。你可以检查 event 对象是否支持 stopPropagation 方法。如果支持，则调用它；否则就将 cancelBubble 属性的值设置为 true。

6. 其解决方案是：

```
if (window.addEventListener) {
   window.addEventListener("load",functioncall,false);
} else if (window.attachEvent) {
   window.attachEvent("onload", functioncall);
}
```

7. 虽然我们没有介绍过如何捕获键盘事件，但你应该知道在此只需捕获 keydown 事件，然后从该事件的 which 属性中就能够获得当前按键的 Unicode 格式的键盘码：

```
window.onload=function() {
    if (document.addEventListener) {
      document.addEventListener("keydown",getKey,false);
    } else if (document.attachEvent) {
      document.attachEvent("keydown", getKey);
    }
}

function getKey(evnt) {
  var theEvent = evnt ? evnt : window.event;
  alert(theEvent.which);
}
```

第 8 章

表单、表单事件及校验

在 JavaScript 中，你可以借助于 document 对象提供的一组不同的方法，通过 DOM 访问表单。第一种方法就是使用 document 对象的 forms 属性。表单是页面元素的一种，它是以数组形式组织的，附加在 document 对象上。表单将存放在表单数组的什么位置上，取决于该表单在页面中的显示位置，将该位置减 1 就是其在数组中的索引值，这是因为 JavaScript 中的数组是从 0 开始索引的。如果页面中包含两个表单，那么使用索引值 0 就可以访问第一个表单：

```
var theForm = document.forms[0];
```

使用表单数组的索引值来访问表单也存在一些问题，如果你修改了页面（例如添加或删除一个表单），那么 JavaScript 代码就可能出现错误，因为基于数组来访问是和其在页面上的位置直接相关的。更好的方法是为表单设置一个标识符，然后使用 document 对象的 getElementById 方法访问它：

```
<form id="someform" ...>
...
var theForm = document.getElementById("someform");
```

在第 7 章中，许多示例里都使用了 document.getElementById 方法，在本章中我们仍然将使用它。在第 9 章中，我们还将更深入地说明该方法；现在，你只需要知道该方法可以用来访问 Web 页面中具有指定标识符的元素。在前面的代码片段中，该方法将返回标识符为“someform”的表单元素。

同样，正如第 7 章中所介绍的那样，在你将表单提交到服务器之前，有 3 种拦截的方式：使用内嵌的事件句柄捕获 submit 事件；使用传统的事件句柄捕获该事件；使用更现代的 addEventListener/attachEvent 方法捕获该事件。无论使用哪一种，最重要的是当你需要对表单中输入的内容进行校验时，都需要在表单内容出错时取消该事件的执行。

8.1 为表单添加事件：不同方法

与表单关联的主要事件是 submit，其事件句柄是 onsubmit。下面就是用传统的方法为表单添加该事件句柄的方法：

```
document.getElementById("someform").onsubmit=formHandler;
```

当你通过内嵌方法为表单添加一个事件句柄时，需要加上一个 return 语句：

```
<form name="someForm" onsubmit="return formHandler();">
```

不管采用哪种方法，都要取消提交操作，这只需在事件句柄函数中返回 false 即可；如果返回 true 或不指定返回值都将使表单提交继续执行。

这些方法都能够正常工作，但它们都不易于维护；在此提及它们，是因为在现有的 JavaScript 应用程序中这些方法都能看到。事件处理的更好方法是使用更现代的 DOM Level 2 的事件监听器。不过，当你需要在要捕获的事件中添加相同的代码时，那么在第 7 章中介绍的事件句柄都将令你感到痛苦，特别是在你处理表单的事件时。这就是为什么可复用的、跨浏览器兼容的事件监听函数是你最常见的公共 JavaScript 代码的原因。

8.1.1 跨浏览器兼容的事件处理

在第 7 章的示例 7.7 中，演示了跨浏览器兼容的、用来捕获事件、添加事件句柄的函数：

```
// 检查对象模型
if (evtObject.addEventListener) {
   document.addEventListener("click",clickMe,false);
} else if (evtObject.attachEvent) {
   evtObject.attachEvent("onclick", clickMe);
} else if (evtObject.onclick) {
   evtObject.onclick=clickMe;
}
```

对于现在的 JavaScript 应用程序而言，你应该使用 attachEvent 和 addEventListener 来实现跨浏览器兼容，但你也不希望在每个事件句柄中都重复输入这些代码，特别是当你要对表单进行验证时，将会涉及很多事件。

更好的方法是创建一个可复用的事件处理函数，例如：

```
function catchEvent(eventObj, event, eventHandler) {
   if (eventObj.addEventListener) {
      eventObj.addEventListener(event, eventHandler,false);
   } else if (eventObj.attachEvent) {
      event = "on" + event;
      eventObj.attachEvent(event, eventHandler);
   }
}
```

在这个函数中，其参数包括一个页面元素、一个特定类型的事件，以及一个事件句柄函数的引用。我们首先将检查 event 对象的 addEventListener 属性是否可用。如果可用，那么就通过它将事件句柄赋给事件。如果不可用，那么将检查 attachEvent 属性是否可用，如果可用则使用它。不过，我们首先将修改事件的名称，使其带上前缀“on”，因为 attachEvent 函数希望传入的参数类似于“onclick”，而非“click”。

最后，如果 attachEvent 或 addEventListener 都不可用（不过这在今天的绝大多数浏览器中都不太可能出现），那么就使用传统的 DOM Level 0 事件处理方法，也就是将要添加的事件句柄直接赋给该对象的事件句柄属性。

提示

在本书中，有很多示例仍然使用了传统的事件处理方法，最主要的原因是它是最简单（但健壮性不足）的方法，这样能够将示例的代码限制到最少。

当验证表单的输入时，如果某个表单域无法通过验证，那么当然需要取消这个事件，因此这意味着你还需要一个通用的事件句柄函数。

8.1.2 取消一个事件

对于表单验证而言，在 submit 事件句柄函数中，可以通过 window 对象将 cancelBubble 属性设置为 true（针对微软的浏览器）、将 event 对象的 returnValue 属性设置为 false，或者组合使用 preventDefault 方法（针对其他浏览器）和传入事件句柄的 event 对象的 stopPropagation 方法，以停止表单提交操作：

```
function formFunction(evnt) {
    var event = evnt ? evnt : window.event;
...
      if (event.preventDefault) {
         event.preventDefault();
         event.stopPropagation();
    } else {
         event.returnValue = false;
         event.cancelBubble = true;
    }
```

在第 7 章中，我们介绍了 cancelBubble 和 stopPropagation 方法，不过 returnValue 和 preventDefault 还没有讲过。我们回顾一下，cancelBubble 用来阻止事件冒泡到 IE 浏览器还能够捕获该事件的其他元素上，而 stopPropagation 方法则用来在其他遵循 DOM Level 2 事件处理机制的浏览器上完成相同的工作。停止将事件冒泡到其他元素上，也是阻止执行默认行为所需工作的一部分。

returnValue 属性相当于在函数中显式地返回 false 值，而 preventDefault 函数则用来阻止基于该元素和事件的默认行为。例如，在提交按钮的 click 事件句柄中，调用 stopPropagate

方法并且将 returnValue 的值设置为 false，将阻止默认的表单提交行为。

正如前一小节中的事件句柄函数那样，取消一个事件的处理过程也可以考虑写成一个可复用的函数：

```
function cancelEvent(event) {
   if (event.preventDefault) {
      event.preventDefault();
      event.stopPropagation();
   } else {
      event.returnValue = false;
      event.cancelBubble = true;
   }
}
```

贯穿本章的这个示例应用程序将同时使用 cancelEvent 和 catchEvent 方法。

一个典型的验证过程是捕获 submit 事件，然后访问每个表单元素并验证其数据，然后向 Web 页面的读者提供一个消息，指出哪些字段缺少或无效。不过如果该表单有点大，可能会有多个字段的数据存在问题，那么列出所有的有问题字段并不是很友好的应答。

对于这种情况还有更加有效的方法，特别是针对大型表单而言。例如，你可以在用户输入一个数据或做出一个选择的同时验证该字段。接下来的每个小节将分别介绍不同类型的表单域，分别说明如何从中获取数据，以及其他可以通过 JavaScript 实现的功能。

8.2 选择列表框

select（选择列表框）元素和其相关的选项为用户提供了从一个列表中选择一项或多项的方法。要定义一个 select 元素，可以使用如下所示的语法：

```
<select name="theSelection" multiple="multiple">
<option value="Opt1">Option 1</option>
<option value="Opt2">Option 2</option>
...
<option value="Optn">Option n</option>
</select>
```

select 元素提供了以下属性，它们都可以通过 JavaScript 访问：

- disabled　该元素是否被禁用；
- form　其包含的表单；
- length　选项数组中包含的选项数；
- options　选项数组；

- selectedIndex　对于单选框元素，那么它就是当前选中的项目编号；对于多选框元素，那么它的值就是选中的第一个值；
- type　元素类型。

select 元素的选项被放在选项数组中。每个选项都是一个包含多个属性的对象。不过，对于表单验证而言，我们感兴趣的属性只有 selected、value 和 text。如果某个选项被选中，那么它的 selected 属性将被设置为 true；value 属性的值就是选项的值；而 text 属性则是它在 Web 页面上呈现的文本信息。

要从一个选择框中获取当前选中的元素，有两种方法，它取决于用户能够选择多个选项还是一个选项。如果用户一次只能选择一个选项，那么就可以使用 select 元素的 selectedIndex 在选项数组中找到相应的选项。

```
var slIdx= document.getElementById("formname").theSelection.selectedIndex;
var opt = document.getElementById("formname").theSelection.options[slIdx];
```

如果用户可以选择多个选项，那么程序就需要遍历整个选项数组，以检查哪些选项是被选中的。示例 8.1 创建了一个包含 3 个选项的多选框。当表单被提交时，每个已选中的选项的文本和值都将显示在一个弹出窗口中。

由于我们需要捕获两个事件，也就是 window 的 load 事件和表单的 submit 事件，因此在此仍然使用了可复用的 catchEvent（前面介绍过）方法，并传入对象、事件和事件句柄作为其参数。

示例 8.1　处理多选框的结果

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>Select</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript">
//<![CDATA[

function catchEvent(eventObj, event, eventHandler) {
   if (eventObj.addEventListener) {
      eventObj.addEventListener(event, eventHandler,false);
   } else if (eventObj.attachEvent) {
      event = "on" + event;
      eventObj.attachEvent(event, eventHandler);
   }
}

catchEvent(window,"load",setupEvents);

function setupEvents(evnt) {
```

```
   catchEvent(document.getElementById("someForm"), "submit", checkForm);
}

function checkForm(evnt) {

   var opts = document.getElementById("someForm").selectOpts.options;

   for (var i = 0; i < opts.length; i++) {
      if (opts[i].selected) {
         alert(opts[i].text + " " + opts[i].value);
      }
   }
   // 没有任何服务器端处理，取消 submit 事件
   return false;
}


//]]>
</script>
</head>
<body>
<form id="someForm" action="">
<p>
<select id="selectOpts" multiple="multiple">
<option value="Opt1">Option One</option>
<option value="Opt2">Option Two</option>
<option value="Opt3">Option Three</option>
</select>
<input type="submit" value="Submit" />
</p>
</form>
</body>
</html>
```

由于在此只输出其值，而不对其进行验证，因此将继续执行表单提交操作，所以在前面小节中介绍的自定义 cancelEvent 函数并没有被用到。

当你需要提供大量选择时（如美国的州名列表、中国的城市列表等），通常会使用选择列表框。由于存在潜在的性能问题，因此可能会限制用户一次只能选择一项，这样就可以直接使用 selectedIndex 来访问被选中的选项，否则就需要对一个很大的数组进行遍历。不过，在这里使用的数组仍然很短，选项数也很容易处理。

你也可以基于实时事件动态地构建一个选择列表框。

8.2.1 动态修改选择列表框

使用 JavaScript，你可以动态地创建、删除一个列表项，还可以基于其他的用户输入信息更新列表项。如果要在示例 8.1 所示的应用程序中添加一个新的选项，那么首先要创

建一个 option 元素，然后将其添加到选项数组中：

```
opts[opts.length] = new Option("Option Four", "Opt4");
```

由于数组的索引值是从 0 开始编号的，因此只要以数组的 length 属性作为索引值，就可以在数组的最后面添加一个新的数组元素。

如果要删除一个选项，那么只需将数组中该选项设置为 null。这样的操作会重新安排数组，而不会导致编号出现不连续现象。

```
opts[2] = null;
```

如果想删除所有的选项，那么只需将数组的 length 设置为 0：

```
opts.length = 0;
```

我对示例 8.1 做了一些小修改，使得当某个选项被选中之后就将其删除掉。修改后的结果如示例 8.2 所示，其中被修改的部分已经通过加粗显示标识出来了。

示例 8.2　删除当前选中的选项

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>Select</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript">
//<![CDATA[

function catchEvent(eventObj, event, eventHandler) {
   if (eventObj.addEventListener) {
      eventObj.addEventListener(event, eventHandler,false);
   } else if (eventObj.attachEvent) {
      event = "on" + event;
      eventObj.attachEvent(event, eventHandler);
   }
}

function cancelEvent(event) {
   if (event.preventDefault) {
      event.preventDefault();
      event.stopPropagation();
   } else {
      event.returnValue = false;
      event.cancelBubble = true;
   }
}

catchEvent(window,"load",setupEvents);
```

```
function setupEvents(evnt) {
    catchEvent(document.getElementById("someForm"), "submit", checkForm);
}

function checkForm(evnt) {
   var theEvent = evnt ? evnt : window.event;

   var opts = document.getElementById("someForm").selectOpts.options;

   for (var i = 0; i < opts.length; i++) {
      if (opts[i].selected) {
         opts[i] = null;
      }
   }

   cancelEvent(theEvent);
}

//]]>
</script>
</head>
<body>
<form id="someForm" action="">
<p>
<select id="selectOpts" multiple="multiple">
<option value="Opt1">Option One</option>
<option value="Opt2">Option Two</option>
<option value="Opt3">Option Three</option>
</select>
<input type="submit" value="Submit" />
</p>
</form>
</body>
</html>
```

我使用了 cancelEvent 方法来取消表单提交操作，因为当一个表单提交之后，该页面将被重载，那么仍然会填装上所有选项，这样我们在 JavaScript 中所做的修改就又被恢复了。

基于其他表单元素的内容修改选择列表框的情况并不多，特别是使用下拉列表框时更是这样，因为当用户点击该列表框上的箭头按钮之前是不会显示其选项的。不过，你可能想根据每个选项的长度自动水平伸缩选择列表框。要解决这个问题，可以通过 JavaScript 动态设置其外框的宽度。我将在第 12 章中介绍页面元素的显示属性的设置方法。

8.2.2 选择列表框和自动选择

除了在表单提交时处理其选项数组元素之外，你还可以捕获到修改选择项的事件，还

能够完成一些自动选择操作。自动选择是一种表单验证机制，它确保所选的选项组合是可以组合在一起的。要完成这一任务，你可以通过捕获表单域的 change 事件，检查其是否选择了相关的选项，如果没有则自动选择它。这对于填写该表单的人而言，可能会带来一些挫折；你无须等到用户填写完所有表单域之后才对整个表单进行验证，只需检查是否遗漏了对成组选项的选择。

我们对示例 8.1 做了一些修改，以展示自动选择功能。在示例 8.3 中，有两个选项是与其他两个选项嵌套的，如果你选择了父选项，那么将自动获得相应的嵌套选项；不过反之则不会，也就是选择嵌套的选项不会自动获得其父选项。

示例 8.3　对选择列表框实现即时验证

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>Select</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript">
//<![CDATA[

function catchEvent(eventObj, event, eventHandler) {
   if (eventObj.addEventListener) {
      eventObj.addEventListener(event, eventHandler,false);
   } else if (eventObj.attachEvent) {
      event = "on" + event;
      eventObj.attachEvent(event, eventHandler);
   }
}

function cancelEvent(event) {
   if (event.preventDefault) {
      event.preventDefault();
      event.stopPropagation();
   } else {
      event.returnValue = false;
      event.cancelBubble = true;
   }
}

catchEvent(window,"load",setupEvents);

function setupEvents(evnt) {
   catchEvent(document.getElementById("selectOpts"),"change",checkSelect);
}

function checkSelect(evnt) {
```

```
    var theEvent = evnt ? evnt : window.event;

    var opts = document.getElementById("someForm").selectOpts.options;
    for (var i = 0; i < opts.length; i++) {
      if (opts[i].selected) {
          switch(opts[i].value) {
          case "Opt1" : opts[i + 1].selected = true;
                        break;
          case "Opt3" : opts[i + 1].selected = true;
                        break;
          case "Opt4" : opts[i + 1].selected = true;
                        break;
        }
      }
    }
    // 没有任何服务器端处理，取消 submit 事件
    cancelEvent(theEvent);
}

//]]>
</script>
</head>
<body>
<form id="someForm" action="">
<p>
<select id="selectOpts" multiple="multiple">
<option value="Opt1">Option One</option>
<option value="Opt1a"> -- Option OneA</option>
<option value="Opt2">Option Two</option>
<option value="Opt3">Option Three</option>
<option value="Opt3a"> -- Option ThreeA</option>
<option value="Opt4">Option Four</option>
<option value="Opt4a"> -- Option FourA</option>
<option value="Opt5">Option Five</option>
</select>
<input type="submit" value="Submit" />
</p>
</form>
</body>
</html>
```

自动选择某些选项可以确保数据的正确性。而且它给人印象深刻，所需的努力也并不多。

自动选择也是即时验证的一种形式，虽然在绝大多数时候，验证给我们的印象是检查一个输入表单域中是否填入了正确的数据。在本章中，我们不打算展示后一种即时验证的形式，不过现在只需要知道选择逻辑上成组的选项，并且禁用不匹配的选项也是一种有用的变体。

你是否经常使用即时验证，取决于表单的复杂度和意图。如果对每个表单元素都使用这种方法，则会激怒用户而不是协助他，不过如果等到将表单提交给服务器之后再对数据进行验证，那么可能会给出一堆需要修改的项，这对用户也必然是一种打击。代码能够做的就这么多，你需要自己判断什么时候及如何使用它。

8.3 单选按钮和复选框

单选按钮（radio button）和复选框（checkbox）提供了一键式选择机制，它所提供的选项数通常要比选择列表框少。它们之间的区别是在单选按钮中一次只能选择一项，而在复选框中一次可以选择多项。

以下是使用单选按钮的语法：

```
<form id="someForm" action="">
<p>
<input type="radio" value="Opt 1" name="radiogroup" />Option 1<br />
<input type="radio" value="Opt 2" name="radiogroup" />Option 2<br />
</p>
</form>
```

注意，它们提供的所有选项的名称是一样的。它们实际上是一个成组的按钮。要访问这个单选按钮组，可以使用 document.getElementById 函数，其参数是这个单选按钮组的名称：

```
var buttons = document.getElementById("radioGroup");

for (var i = 0; i < buttons.length; i++) {
   if (buttons[i].checked) {
      alert(buttons[i].value);
   }
}
```

正如前面所述，复选框与单选按钮之间的区别是在复选框中可以选择多项；另外，如果 input 元素的类型被设置成 checkbox，那么你必须遍历每个 checkbox 元素，以检查哪些是被选中的。

你可以采用如下所示的方法定义一个复选框：

```
<form id="someForm" action="">
   <p>
    Option 1: <input type="checkbox" name="checkbox1" value="Opt1" /><br />
    Option 2: <input type="checkbox" name="checkbox2" value="Opt2" /><br />
   </p>
</form>
```

要判断哪个 checkbox 被选中，你需要遍历所有的 checkbox 项。完成该工作的其中一种方法是基于 checkbox 的名称创建一个元素数组，然后遍历这个数组，检查每个 checkbox

是否被选中。

如果想了解另一种方法，你可以先看看第 11 章中介绍的 DOM 方法。它需要使用一个名为 getElementsByTagName 的 document 方法。

getElementsByTagName 方法将返回指定元素类型的所有项目。你可以获取整个页面中指定类型的所有元素，你也可以使用 getElementById 方法获得一个元素，然后将其放到一个包含相应元素类型的所有元素的集合中。

在示例 8.4 中所示的应用程序里是一个由 4 个复选框组成的表单。我们并不关心选择了哪个，而是关心一共选择了几个。不过，至少需要选择一个，并且将通过 JavaScript 代码来验证是否至少选择了一个。

示例 8.4　检查复选框的选择情况

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>Checkbox</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript">
//<![CDATA[

function catchEvent(eventObj, event, eventHandler) {
   if (eventObj.addEventListener) {
      eventObj.addEventListener(event, eventHandler,false);
   } else if (eventObj.attachEvent) {
      event = "on" + event;
      eventObj.attachEvent(event, eventHandler);
   }
}

function cancelEvent(event) {
   if (event.preventDefault) {
      event.preventDefault();
      event.stopPropagation();
   } else {
      event.returnValue = false;
      event.cancelBubble = true;
   }
}

catchEvent(window,"load",setupEvents);

function setupEvents(evnt) {
  catchEvent(document.getElementById("someForm"), "submit",checkColors);
}
function checkColors(evnt) {
```

```
    var theEvent = evnt ? evnt : window.event;
    var colorOpts =
document.getElementById("someForm").getElementsByTagName("input");

    // 遍历复选框中的每个checkbox，检查是否被选中
    var isChecked = false;
    for (var i = 0; i < colorOpts.length; i++) {
      if ((colorOpts[i].type == "checkbox") && (colorOpts[i].checked)) {
        isChecked=true;
        break;
      }
    }

    // 没有一个checkbox被选中
    if (!isChecked) {
       alert("You must check one of the four color checkboxes");
       cancelEvent(theEvent);
    }
}

//]]>
</script>
</head>
<body>
<form id="someForm" action="">
  <p>
    <input type="checkbox" name="color1" value="red" /> : red<br />
    <input type="checkbox" name="color2" value="blue" /> : blue<br />
    <input type="checkbox" name="color3" value="green" /> : green<br />
    <input type="checkbox" name="color4" value="yellow" /> : yellow<br /><br />

    <input type="submit" value="Submit" />
  </p>
</form>
</body>
</html>
```

对于示例 8.4 而言，有两个地方需要说明。首先，访问该元素所使用的标识符是“input”而不是“checkbox”；因为并不存在“checkbox”元素。这段代码访问了 input 元素的 type 属性，以了解触发事件的是什么类型的元素。其次，我们需要检查 input 元素是不是 checkbox，并且如果 flag 变量（说明是否至少有一个 checkbox 被选中）还未设置，那么将检查它是否被选中。因为我们只关心是否至少选择了一个 checkbox，因此当第一次遇到已选中的 checkbox 就将跳出循环语句，显然已经无须再对剩余的 checkbox 进行判断了。

除了确保至少选择了一个 checkbox 之外，更麻烦的是用户搞乱了复选框或单选按钮。你可以使用其他表单选项使它与按钮的行为更相似，但如果你希望禁用一个或多个单选按钮或复选框，更好的做法是禁用这些选项，而不是在提交之后校验它。在 JavaScript

中可以使用以下语句来禁用一个选项：

```
document.getElementById("someForm").radiogroup[1].disabled=true;
```

如果你想基于一个单选按钮或复选框的选择来修改其他表单元素，那么可以捕获单选按钮组或复选框组的 click 事件。如果要添加一个事件句柄，可以将其添加给单选按钮组或复选框组：

```
document.getElementById("someForm").radiogroup.onclick=handleClick;
```

另外，你也可以创建一个新的复选框或单选按钮元素，只不过你需要确保创建的选项是一个 input 元素，并且设置了相应的类型。你也可以向表单添加一个新的元素：

```
var newCheckbox = document.createElement("input");
newCheckbox.type="checkbox";
newCheckbox.name="colors1";
newCheckbox.checked=true;
someForm.appendChild(newCheckBox);
```

你也可以使用动态 Web 效果（将在第 12 章中介绍）来隐藏或显示复选框及其他表单元素，不过更好的方法是在 JavaScript 中设置其 disabled 属性，以管理控件的动态属性。当一个表单元素被禁用时，它就无法接收事件，也无法设置其值。

虽然这几个小节示范了如何为各种表单元素类型提供即时验证功能，不过最经常需要校验的还是接下来介绍的、更加自由的文本框。

8.4 文本框、多行文本框、密码框和隐藏表单域元素

文本框（text）、多行文本框（textarea）、密码框（password）可能算是最经常使用的表单域元素了，同时这些 input 元素也是最需要验证的。隐藏表单域（hidden）通常不存在验证的问题，但它也是一个基于文本的表单域，因此我们也将它放在这个小节里，将相似的控件放在一起介绍。

在 HTML 中定义一个单行的、基于文字的 input 元素，可以采用以下代码：

```
<input type="text|hidden|password" name="fieldName" value="Some value" />
```

对 type 属性的设置定义了表单域的类型。文本框是常规的文本类表单域，而隐藏表单域则是用户在表单中看不到的，而密码框则会用圆点代替输入文本的回显，以防有人站在用户的背后偷看。

多行文本框与它们相似，只不过不像其他 input 字段，它有一个开始和结束标签，你可以设置其列和行的宽度。另外，多行文本框是独立的元素；它不是 input 元素的一种：

```
<textarea name="fieldName" rows="10" cols="10">Initial text</textarea>
```

在 JavaScript 中，可以通过表单元素的 value 属性来访问各种文本类型表单域的值。为了展示该属性的用法，示例 8.5 定义了一个表单，上面有 4 种文本类型的表单元素。当

该表单提交时，将使用 JavaScript 访问 3 种文本类型的 input 元素，并根据它们的内容构建一个字符串，然后将其添加到多行文本框中。

示例 8.5　从 JavaScript 中访问文本类型的 input 元素

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>Text Fields</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript">
//<![CDATA[

function catchEvent(eventObj, event, eventHandler) {
  if (eventObj.addEventListener) {
     eventObj.addEventListener(event, eventHandler,false);
  } else if (eventObj.attachEvent) {
     event = "on" + event;
     eventObj.attachEvent(event, eventHandler);
  }
}

function cancelEvent(event) {
  if (event.preventDefault) {
     event.preventDefault();
     event.stopPropagation();
  } else {
     event.returnValue = false;
     event.cancelBubble = true;
  }
}

catchEvent(window,"load",setupEvents);

function setupEvents(evnt) {
    catchEvent(document.getElementById("someForm"), "submit",validateForm);
}

function validateForm(evnt) {

   var theEvent = evnt ? evnt : window.event;

   var strResults = "";
   var textInputs =
document.getElementById("someForm").getElementsByTagName("input");
   for (var i = 0; i < textInputs.length; i++) {
    if (textInputs[i].type != "submit") {
      strResults += textInputs[i].value;
```

```
    }
  }
  document.getElementById("text4").value=strResults;

  // 不去除结果
  cancelEvent(theEvent);
}


//]]>
</script>
</head>
<body>
<form id="someForm" action="">
<p>
<input type="text" name="text1" /><br />
<input type="password" name="text2" /><br />
<input type="hidden" name="text3" value="hidden value" />
<textarea name="text4" cols="50" rows="10">The text area</textarea>
<input type="submit" value="Submit" />
</p>
</form>
</body>
</html>
```

在这个示例中，由于提交按钮也是一个 input 类型的表单元素，因此代码将检查当前处理的元素不是“submit”类型的。同样，你会注意到在代码中取消了表单提交操作，不管文本域中的值是什么。如果你不想取消提交操作，那么表单的字段将被重置，那么你也就将失去刚刚创建的字符串。图 8.1 展示了在 Google Chrome 浏览器中打开该应用程序的显示效果。

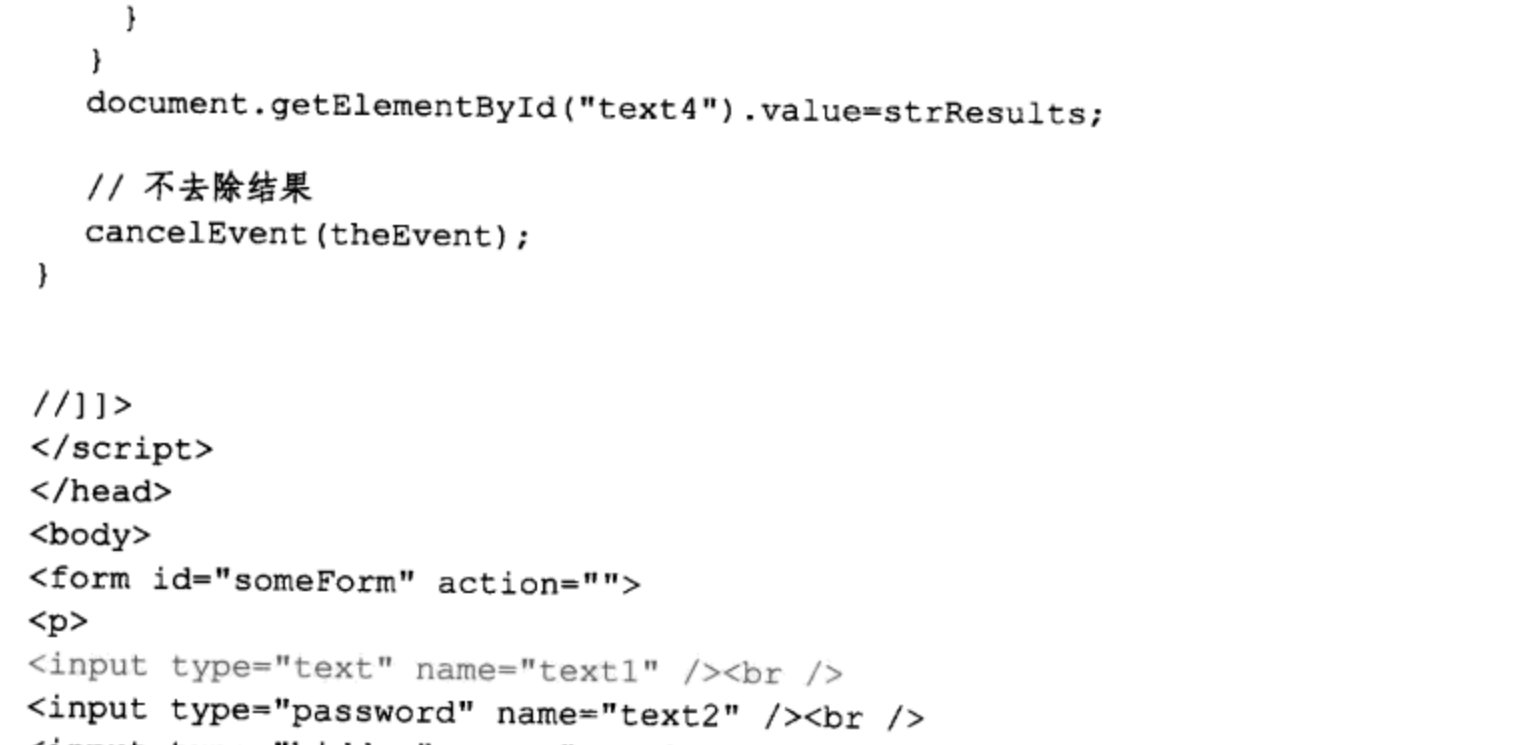


图 8.1　示例 8.5 在 Google Chrome 中打开的效果

8.4.1 文本验证

文本域是所有表单元素中最容易出现“坏数据”的，这些错误可能源于用户的误解，也可能是用户输入错误。因此，这些表单域是最需要验证的。

当执行即时验证时，我们最关心的事件包括 change、focus 和 blur。当鼠标移到一个文本类型的 input 域中时，将触发 focus 事件；当鼠标离开时，将触发 blur 事件。当它们的内容被修改时将触发 change 事件。因此你最可能通过捕获 change 事件来验证其内容，捕获 blur 事件来确保该表单域已经填入了相应值，当然前提是该表单域是必填的。

通过对示例 8.5 所示的应用程序的修改，示例 8.6 中所示的应用程序将捕获密码框的 blur 事件，并且检查它们的值以确保它们是相同的。另外，当第一个文本框的内容被修改时，将通过一个验证社保号码（SSN）的正则表达式（nnn-nn-nnnn）进行校验。

示例 8.6　对基于文本的 input 域应用即时验证

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
<head>
<title>JiT RegEx</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript">
//<![CDATA[

function catchEvent(eventObj, event, eventHandler) {
   if (eventObj.addEventListener) {
      eventObj.addEventListener(event, eventHandler,false);
   } else if (eventObj.attachEvent) {
      event = "on" + event;
      eventObj.attachEvent(event, eventHandler);
   }
}

catchEvent(window,"load",setupEvents);

function setupEvents(evnt) {
  catchEvent(document.getElementById("text2"), "blur",checkRequired);
  catchEvent(document.getElementById("text1"), "change", validateField);
}

function checkRequired (evnt) {
  var theEvent = evnt ? evnt : window.event;
  var target = theEvent.target ? theEvent.target : theEvent.srcElement;

  var txtInput = target.value;
```

```
  if (txtInput == null || txtInput == "") {
    alert("value is required in field");
  }
}

function validateField(evnt) {
  var theEvent = evnt ? evnt : window.event;
  var target = theEvent.target ? theEvent.target : theEvent.srcElement;
  var rgEx = /^\d{3}[-]?\d{2}[-]?\d{4}$/g;

  var OK = rgEx.exec(target.value);
  if (!OK) {
    alert("not an ssn");
  }

}
//]]>
</script>
</head>
<body>
<form name="someForm">
<input type="text" name="text1" id="text1" /><br />
<input type="password" name="text2" id="text2" /><br />
<input type="hidden" name="text3" value="hidden value" />
<textarea name="text4" cols=50 rows=10>The text area</textarea><br /><br />
<input type="submit" value="Submit" />
</form>
</body>
</html>
```

现在，如果用户输入的 SSN（社保号码）的格式不正确，那么当用户离开这个表单域时就会看到相应的错误提示信息。另外，如果用户没有提供密码，那么将弹出一个 alert 对话框。当然，alert 对话框总是令人讨厌的，在本书后续的内容中，你会看到更好的反馈机制。

对于必填的表单域，如果你决定使用 focus 方法来让光标回到这个表单域，那么堪称是极致的处理方法，特别是再加上一个弹出对话框给出相应的错误信息。在如 Opera 之类的浏览器中，这将导致一个无法结束的循环，它也将大大地激怒用户。底线是不要通过 focus 方法来强调必填表单域。

该示例还展示了正则表达式对于验证用户在表单上的输入是多么重要。在接下来的小节中，我们将更进一步了解如何将正则表达式应用到表单域上。

8.5 input 元素和基于正则表达式的验证

绝大多数表单域只需要一些文本，而不会对格式有要求。不过，有些表单域的输入是

有格式要求的。与其将数据发送给服务器，让它来判断数据是否有效，还不如先通过正则表达式验证一下数据的格式。

使用正则表达式验证 input 元素通常发生在以下场景中：

- 授权码或购买证明；
- 电子邮件地址；
- 电话号码；
- SSN 或其他形式的身份证明；
- 日期；
- 状态缩写；
- 信用卡号；
- Web 页面的 URL 或其他形式的 URI。

与其直接在代码中使用不同的正则表达式，还不如采用示例 8.7 中的方法，该示例使用了一个名为 JiT RegEx Machine 的小型应用程序，它将从一个表单域中获取一个正则表达式，再从另一个表单域中获取字符串，然后在表单提交时进行模式匹配。其结果将输出在第三个表单域中。

示例 8.7　JiT RegEx Machine 应用程序

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
<head>
<title>The JiT RegEx Machine</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript">
//<![CDATA[

if (window.addEventListener) {
       window.addEventListener("load",setupEvents,false);
    } else if (window.attachEvent) {
       window.attachEvent("onload", setupEvents);
    } else {
       window.onload=setupEvents;
}

function setupEvents(evnt) {
   document.someForm.onsubmit=validateField;
}

function validateField(evnt) {
```

```
  var rgEx = new RegExp(document.someForm.text1.value);
  var OK = rgEx.exec(document.someForm.text2.value);

  // 获得结果并打印
  if (!OK) {
     document.someForm.text3.value = "Not a match";
  } else {
     document.someForm.text3.value = "The Pattern matched!";
  }

  return false;

}
//]]>
</script>
</head>
<body>
<form name="someForm" style="padding: 10px">
Regular Expression: <input type="text" name="text1" /><br /><br />
<textarea name="text2" cols=50 rows=10></textarea><br />
Result: <input type="text" name="text3" /><br /><br />
<input type="submit" value="Check RegExp" />
</form>
</body>
</html>
```

授权码或购买凭证号的正则表达式模式是在特定的位置要有特定的字符或数据。例如，你可以有一个 13 个字符长的授权码，其中第 6～8 个字符是 BUS，其他的部分是文字和数字组成的字符，那么就可以使用如下所示的正则表达式：

```
^\w{5}BUS\w{5}
```

如果你想验证一个电子邮件，也就是必须有一个“@”字符，后面是一个域名，以及其他的一些限制，因此可以采用如下所示的正则表达式：

```
^.+@[^\.].*\.[a-z]{2,}
```

对于日期而言，如果你想要求输入的格式是 mm/dd/yyyy，那么可以使用：

```
^\d{2}\/\d{2}\/\d{4}
```

这些示例都太简单了？好吧，我们来看看用于检查 SSN 的正则表达式：

```
^(?!000)([0-6]\d{2}|7([0-6]\d|7[012]))([ -]?)(?!00)\d\d\3(?!0000)\d{4}$
```

前面列出的这些正则表达式都是从 Regular Expression 程序库中获得的，http://regexlib.com/中提供了价值无法估量的宝贵资源。这些正则表达式模式不仅验证格式，还验证数字组合，就像针对 SSN 的正则表达式模式那样。其他的正则表达式则更复杂，它甚至可以区分 Visa 信用卡和 MasterCard 信用卡。

提示

如果你想成为正则表达式的专家，可以花些时间研究 Regular Expression 程序库。或者你可以买一本 Jeffrey E.F. Friedl 所著的 *Mastering Regular Expressions*（O'Reilly 出版社出版）[1]。这是一本关于正则表达式的权威指南。

另外，你需要区分 Visa 和 MasterCard 信用卡吗？在使用正则表达式时，要紧记研究一个完美的验证模式将花费大量的时间，花费比验证更多的时间是值得。但你需要评价是否值得花费这么多时间来完成对将提交给服务器的数据进行验证的任务。

8.6 表单、沙箱和 XSS

当 JavaScript 第一次发布时，大家对于它能够在打开 Web 页面时执行一些运行在客户端的代码的特性表示担忧，这种担忧是可以理解的。如果 JavaScript 中包含一些有害的代码，如删除所有 Word 文档，或者更恶劣地将其发送给脚本编写者，那该怎么办？

为了避免这些问题的出现，同时也让浏览器用户更放心，JavaScript 将其操作限制在一个沙箱中。沙箱是一个受限的环境，使脚本无法访问浏览器所在计算机中的资源。另外，浏览器所实现的安全限制要远远多于 JavaScript 语言所实现的这些机制。它还定义了一个针对浏览器的安全策略，用来判断脚本是否能够执行。

在这样的安全策略中有一条规定，脚本不能与该表单所属域之外的页面上的脚本进行通信。绝大多数浏览器都提供了对这些策略做进一步自定义的手段，以便为脚本提供更严格或更宽松的限制。

不幸的是，即使用了 JavaScript 的沙箱和浏览器的安全策略等机制，JavaScript 仍然存在问题，黑客会寻找和发现 JavaScript 的错误，有些与浏览器相关，有些是无关的。其中最严重的一个就是众所周知的跨网站脚本（XSS）。它实际上是一个会影响安全的类（有些是针对 JavaScript 的，有些是浏览器的漏洞，还有一些是针对服务器的），它们可以偷取 cookie 信息，以暴露客户端或网站的数据，以及引发一些其他严重的问题。它还可以执行一些对服务器和数据库产生破坏的行为，并且使你的网站用户担受风险。

XSS 攻击的一个例子就是众所周知的 SQL 注入攻击。这种类型的攻击是通过在 input 表单域文本的最后添加实际的 SQL 代码来进行的。下面就是一个示例（节选自我写的另一本名为 *Adding Ajax* 的书，也是 O'Reilly 出版社出版的），它将完成发出一个帖子之类的任务：

```
x' where ID = '1'; DROP TABLE wp_users; update wp_posts set post_title='x
```

如果 input 表单域的内容不够“干净”，并且直接在如 PHP 代码中添加一个 SQL 语句，

[1] 译者注：本书的中译版已于 2007 年 9 月出版，书名为《精通正则表达式》，电子工业出版社。

那么最终将生成下面的语句：

```
update wp_posts set post_title = 'x' where ID = '1'; DROP TABLE wp_users; update
wp_posts set post_title = 'x' where ID = '$post';
```

附加上去的内容将成功完成一些更新操作，包括一些添加到 input 表单域的 SQL 语句代码。

要避免出现如 SQL 注入的 XSS 攻击，就必须检查文本输入，确保其中的内容是可接收的数据，不存在如 SQL 语句之类的内容。不过在哪些地方处理这些问题，仍然在大家的争论之中。

我相信所有此类安全清理和验证工作都应该放在服务器端应用程序上，因为这种应用程序至少有多种攻击入口点：通过前端的如 JavaScript 编写的应用程序，通过直接的服务器到服务器的应用程序调用。另外，我认为在 JavaScript 中揭示清理标准能够发现系统中还存在哪些弱点。

不过，有人认为前端开发人员也要像开发人员那样考虑安全问题。特别是，如果我们没有在后端进行控制，那么考虑安全问题要比期望用户避免低劣的输入更好一些。

另外，还有一些除 SQL 注入之外的 XSS 弱点，包括可能会在最终生成的文本（如评论）中嵌入不安全的 URL。针对这类问题，其中一种处理方法是使用编码后的 HTML（将 HTML 元素转成无害的文本），或者对所有 HTML 代码进行过滤。不过，这些过滤和编码任务出现在服务器端可能要比出现在客户端 JavaScript 代码中更好。

不管清理和预防性过滤功能放在哪里，最重要的是知道在表单中的 input 元素易受到什么样的攻击，以确保这些表单域中传递的是安全、有效的文本。

提示

CERT 是关于安全问题的权威网站，在 http://www.cert.org/advisories/CA-2000-02.html 中可以找到与 XSS 相关的介绍。在 CGISecurity.com 网站上有关于 XSS 的更深入的 FAQ; 你可以在 http://www.cgisecurity.com/articles/xss-faq.shtml 上找到它。

8.7 知识测验

1. 如果表单数据不完整或无效，如何停止表单提交操作？
2. 想在表单提交之前对文本框进行验证，那么应该捕获它的什么事件？
3. 如果要确保一个表单域输入的内容只包含字符和空白符，应该使用什么样的代码？
4. 编写一段 JavaScript 程序代码，捕获选中一个单选按钮时触发的事件，如果其中一个按钮被单击，则禁用文本框，如果另一个按钮被单击，则启用该文本框。

8.8 测验答案

1. 如果你使用 DOM Level 0 的事件，那么只要在事件句柄中返回 false 值，并且在事件句柄脚本中取消表单提交操作即可。如果使用的是 DOM Level 2 模型，那么就将 event 对象的 cancelBubble 属性设置为 true（针对 IE），并调用其 preventDefault 方法（针对其他浏览器）。

2. 当一个表单域失去焦点时将触发 blur 事件。这时是检查文本框的值并确保其为有效数据的最佳时机。

3. 以下是其中一种可选的方法：

```
var fieldPattern = /^[A-Za-z\s]*$/g;
var OK = fieldPattern.exec(document.forms[0].text1.value);
```

4. 首先，程序必须为每个单选按钮的 onclick 事件句柄指定一个函数：

```
document.forms[0].radiogroup[0].onclick=handleClick;
document.forms[0].radiogroup[1].onclick=handleClick;
```

如果有多个按钮，那么可以通过一个 for 循环实现。在 handleClick 函数中，检查单选按钮是否被选中，并根据其结果决定是否禁用该表单元素。例如禁用 submit 按钮：

```
function handleClick() {
   if (document.forms[0].radiogroup[1].checked) {
       document.forms[0].submit.disabled=true;
   } else {
       document.forms[0].submit.disabled=false;
   }
}
```

第 9 章

浏览器就像个难题箱

浏览器对象模型（BOM）是一组从浏览器上下文中继承而来的对象，这也是绝大多数 JavaScript 应用程序中函数运行的上下文。有时它也被称为 DOM Level 0，或者就叫 DOM，BOM 是一组有限的公共 Web 对象，在 Netscape Navigator 和微软的 IE 的早期版本中就能够通过 JavaScript 来访问它们。

前面的章节中已经介绍了一些这样的对象，如 window、document 和 form。本章将更进一步介绍这些对象，同时还将介绍 BOM 中的其他对象。

9.1 浏览器结构概述

BOM 是一个层次化的对象集，每个层次上的对象都可以通过它们的父对象来访问。你可以通过 window 对象访问 BOM 中的所有对象，它是最顶层的元素。下一层是 document，在之前的例子中最经常使用到它。与它同级的对象还有 navigator、frames、location、history 和 screen 对象。通过 document 对象，可以访问 form、anchor、link、cookie 和 image 对象集合。正如第 8 章中所演示的那样，form 对象也拥有作为其子元素的元素集合。

图 9.1 展示了 BOM 的主体结构，说明了这些元素和其他元素之间的关系。

一个对象的属性可以是独立的对象，该对象可以拥有自己的属性（包括方法），也可以没有，还可以是一个对象集合的引用，而且集合中每个对象还可以有自己的属性。在图 9.1 中，可以作为集合成员的属性都添加了阴影。

在一个集合中，要访问某个项目有两种方法。第一种方法是使用数组索引，例如如果要访问页面载入时包含的前两个 image 元素，那么分别可以使用 document.images[0] 和 document.images[1]。第二种访问元素的方法是使用其 name 属性：

```
var img = document.images("somename");
```

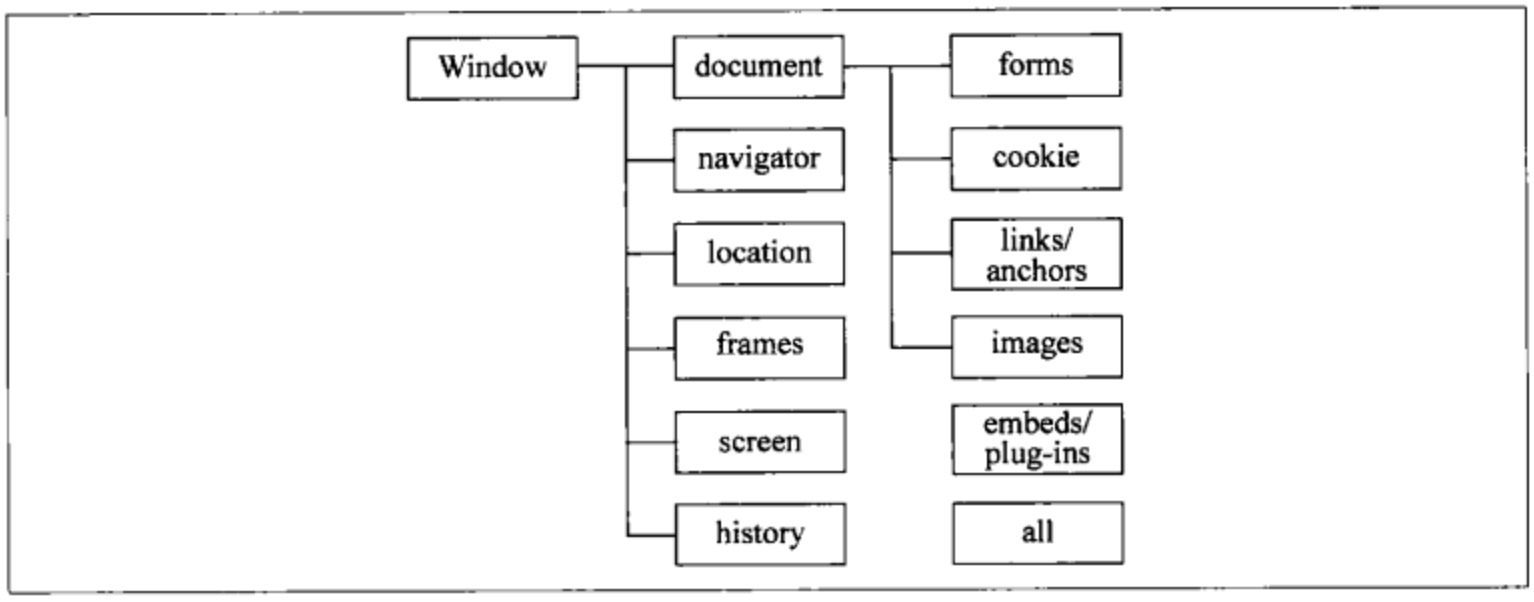


图 9.1　BOM 的层次结构

注意，通过 name 属性访问元素时使用的格式与函数调用相似（使用小括号），而不是像数组中那样使用方括号。另外，使用 name 属性访问的前提是这些元素定义的 name 属性符合 XHTML 强制的 DOCTYPE，正如本书中各个示例里使用的那样。对于绝大多数页面元素而言（包括 img 和 form 元素），定义其 name 属性可能会导致验证失败。这是因为在 XML 中（它是以 XHTML 作为基础的），只能使用 id 属性作为元素标识。

访问这些元素的第三种方法是使用元素的 id 属性，但这要求它是一个集合的一部分。和 name 属性不同，我们为元素指定了一个唯一标识符，并使用之前章节中展示的 getElementById 方法来访问：

```
var theImage = document.getElementById("someimageid");
```

在 XHTML 传统的 DOCTYPE 中，仍然允许使用 name 属性；而对于表单内的元素（不包括 form 本身），不管是 XHTML 传统还是强制 DOCTYPE，都能够使用 name 属性。你可以使用以下技术来访问 HTML 和 XHTML 中的 form 元素：

```
var theElement = document.forms[0].elements("someelement").value;
```

现在我们回到 BOM 上来，正如你在图 9.1 中所看到的那样，window 是所有对象的领头羊，我们先看看这个对象。

9.2　window 对象

浏览器的 window 对象封装了整个浏览器环境，包括 window 的“chrome”（浏览器的一部分，用来封装 document）、实际的 Web 页面，以及页面中的事件。

提示

Chrome 在这里表示的是组成浏览器窗体的通用组件，而不是 Google 开发的名为 Chrome 的 Web 浏览器。

window 是一个全局对象，它始终存在，即使是隐式的（而非显式的）。在前一章节中，你使用了一个名为 alert 的全局函数，该函数虽然看起来和所有对象都“不相关”，它实际上是 window 对象的一部分（隐式的），也是 document 等二级对象的一部分，还是所有全局变量和其他对象的一部分，只要没有指定与对象模型中其他对象相关联。

window 对象有趣之处不仅仅是所有元素的父元素。通过它可以手动设置浏览器状态条上的状态，执行打开一个新窗口、重新调整已显示窗口的大小、关闭窗口等操作。如果你通过一个独立的窗口显示帮助或其他辅助信息，那么这种打开新窗体的功能将是十分有用的，不过随着动态 Web 效果和 Ajax 的流行，这种弹出式窗口已经越来越不被喜欢，这种类型的通信现在通常会在一个 document 内进行，而不会放在一个独立的窗口中。

警告
在不同浏览器中的不同安全限制，会限制这些功能的使用。

window 对象的方法和属性可以分成 4 类：

- 创建新窗口，维护现有窗口的行为；
- 在窗口中创建带分区的文档（帧和 iframes）；
- 定时器的创建和控制；
- 用来控制浏览器窗体内特定元素的属性，如 document、navigator、screen 等。

9.3 窗口的创建和控件

第一类方法主要用来创建新窗口，包括用来创建弹出式窗口的 3 个方法（每个都有特定的用途），还有一个用来创建一个包含部分（或多或少）基础设施的窗口。

9.3.1 对话框：alert、confirm 和 prompt

有 3 个简单的、用来弹出窗口对象的方法，通过它们可以创建一个最简单的窗口，每个都有特定的用途。它们通常被称为对话框窗口。

你对于 alert 对话框窗口应该很熟悉，它是向访问当前页面的读者显示一条消息的最简单方法。它的参数只有一个，那就是存放该消息的字符串，它不返回任何值：

```
alert("This is the message");
```

confirm 方法将创建一个带有一个问题、两个按钮（Cancel 和 OK）的对话框。显示的消息就是传入该方法的参数，返回值将根据用户单击哪个按钮决定，如果用户单击 OK 按钮，那么将返回 true，如果单击 Cancel 按钮，则返回 false：

```
var result = confirm("Do you want fries with that?");
```

prompt 方法将弹出一个带有文本框的窗口，同时也提供了 Cancel 和 OK 按钮。它有两个参数，一个是提示信息，一个是文本框中的默认字符串：

```
var response = prompt("What's your name?", "Wouldn't you like to know...");
```

如果不想为第二个参数指定特定字符串，那么最好传入一个空字符串。否则，在如 IE 之类的某些浏览器中，prompt 对话框上的文本框可能会被输入“undefined”。

某些浏览器可能会出于安全方面的考虑而限制 prompt 对话框的使用。你可能不得不让 JavaScript 应用程序的用户启用对 prompt 对话框的支持。更好的选择是改为使用表单。

提示

注意，这些方法都是基于 window 对象的引用的，该对象是全局的，通常假定是存在的。

我们将这类窗口称为弹出窗口，因为它们都是弹出式的。不过，这一术语通常特指在访问 Web 时接管操作权的窗口。是的，你知道它是一种可能会被浏览器阻止的窗口。不过，我们不会像“打兔子”游戏一样，弹出一些让用户不断单击的窗口。打开一个新窗口显示额外信息是更有效的方式，这样做不会让用户离开当前页面。

9.3.2 创建自定义窗口

我们经常需要打开一个新窗口，例如访问帮助系统、提供额外信息、查看购物车或其他相关信息，当然还可能是为了显示抓人眼球的动画。

如果想打开一个新窗口，并且控制其内容、大小、位置等属性，可以使用 window 对象的 open 方法。该方法有几个参数，它们都是可选的：

- 第一个参数是要打开的 URL，如果需要的话；
- 第二个参数是为窗口指定的名称，通过它可以实现父窗口和子窗口之间的通信，如果打开了多个窗口，通过它还能够完成兄弟窗口之间的通信；
- 第三个参数是一组 window 对象选项，它们将整合在一个字符串中，每个选项之间用逗号分开，它们用来控制新窗口的外观和行为。

在下面这行代码中，创建一个名为“test”的新窗口。它包含了一个指向 O'Reilly 网站的链接，大小设置为 600×400 像素，隐去了地址栏和工具条：

```
window.open("http://oreilly.com","test","width=600,height=400,toolbar
=no,location=no");
```

你无法对所有选项进行设置。有一些与窗口中帧的特定组件以及 window 层的位置相关的选项是无法设置的，除非你的脚本拥有 UniversalBrowserWrite 权限才能够修改其默认值，这通常需要脚本签名支持。因为脚本签名支持并不普遍，所以最好避免依赖于这些选项。

表 9.1 中列出了绝大多数浏览器支持的公共选项，介绍了它们的用途和默认值。

表 9.1　跨浏览器兼容的 window.open 选项

选　项	用　途	默 认 值
alwaysLowered	将该窗口指定为“pop-under”类型，也就是将新窗口放在父窗口之下，除非父窗口被最小化了	默认值为 no；除非拥有 UniversalBrowserWrite 权限，否则自定义无效
alwaysRaised	打开一个位于父窗口顶部的窗口	默认值为 no；除非拥有 UniversalBrowserWrite 权限，否则自定义无效
dependent	打开一个依赖于父窗口的新窗口；当父窗口被关闭时，它也将被关闭	默认值为 no
directories	在浏览器上显示个人书签或链接工具条，具体取决于浏览器类型	默认值为 yes；在有些浏览器中可以修改它
height	内容区域的高度，单位为像素	最小值，100 像素
width	内容区域的宽度，单位为像素	最小值，100 像素
outerHeight	整个浏览器窗口的高度，单位为像素	最小值，100 像素
outerWidth	整个浏览器窗口的宽度，单位为像素	最小值，100 像素
top	浏览器窗口最上方的位置	必须在屏幕范围之内
left	浏览器窗口最左边的位置	必须在屏幕范围之内
menubar	如果设置为 yes，则显示菜单条	在有些浏览器中可以修改
toolbar	如果设置为 yes，则显示工具条	在有些浏览器中可以修改
location	如果设置为 yes，则显示地址栏	IE 7 中要求必须显示
status	如果设置为 yes，则在浏览器窗口底部显示状态栏	有些浏览器默认设置为 yes
resizable	如果设置为 yes，则该窗口大小可调整	在有些浏览器中可以修改
scrollbars	如果设置为 yes，则窗口将带有滚动条（如果一个页面显示不完）	在有些浏览器中可以修改
modal	在返回主窗口之前，所有打开的窗口必须先关闭	对话框窗口就是模式窗口；在某些浏览器中需要有 UniversalBrowserWrite 权限才能修改
dialog	打开一个对话框窗口，其外观和行为与 alert 窗口类似	
minimizable	当 dialog 选项设置成 yes 时，添加最小化窗口按钮	
titlebar	呈现或删除标题栏	默认值为 on；如果拥有 UniversalBrowserWrite 权限，则在某些浏览器中可以修改
close	呈现或删除关闭按钮（图标）	默认值为 on；如果拥有 UniversalBrowserWrite 权限，则在某些浏览器中可以修改

正如你所看到的那样，当使用弹出式窗口时，安全是最主要的关注点。

示例 9.1 中的应用程序将使用 prompt 方法获取打开一个新窗口的参数字符串。你可以尝试不同的选项字符串，然后看看它们之间的差异。以下就是一个选项字符串的例子：

```
menubar=yes,location=yes,resizable=yes
```

注意，你很可能需要修改浏览器的安全设置，以使得 prompt 对话框能够正常显示。另外，当你要显示新窗口时，需要清除该页面的弹出窗口。

示例 9.1　使用 prompt 对话框获取 window.open 选项

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>Windows</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript">
//<![CDATA[

window.onload = function() {
   var optionString = prompt("Enter your option string", "");
   optionString = optionString ? optionString : "";

   window.open("http://burningbird.net","test",optionString);
}
//]]>
</script>
</head>
<body>
<p>some content</p>
</body>
</html>
```

在程序中，我们通过检查返回值就能判断当前窗口是否被阻塞了。如果它的值不是 null，那么该窗口仍然是阻塞状态。每种浏览器对弹出窗口的控制方式有所不同，但都会通过一些提示告诉你弹出窗口阻塞了当前窗口，并且向访问者提供一种解除阻塞的方法。

如果在示例 9.1 中没有给出选项字符串，新打开的窗口可能会显示为一个标签页，这是绝大多数浏览器的默认行为，也可能以独立窗口的形式显示。当指定了某些选项时，其他选项（如 toolbar、location 和 menubar）可能默认为 off，或者根据你使用的浏览器决定。在打开一个窗口时还有其他可用的选项，不过使用它们将违背可访问性标准。fullscreen 就是其中一个例子，该选项将使浏览器以全屏显示，这是一个要小心使用的选项。

当你得到一个 window 对象之后，就可以对其父窗口或窗口本身进行调整。接下来，我们将介绍维护窗口的一些方法。

9.3.3 维护窗口

在这些维护窗口的方法中，其功能通常包括改变窗口的大小、焦点和位置。通过窗口引用就能够完成对该窗口的维护；要维护父窗口则使用关键字 opener；要维护包含当前运行脚本的窗口，则应该使用关键字 self。

在下列代码中，该窗口中包含的 JavaScript 代码将把窗口移到位置(0,0)上（也就是屏幕的左上角）：

```
self.moveTo(0,0);
```

如果你不是想在代码中引用打开新窗口的父窗口，而是想获得该窗口的引用，那么可以通过 window.open 的返回值获得：

```
var newWindow = window.open("http://somecompany.com","NewWindow",
"...options...");
newWindow.moveTo(0,0);
```

打开的窗口可以引用任何它创建的窗口。在新创建的窗口中，通过关键字 opener 也可以获得打开它的父窗口引用：

```
opener.moveTo(0,0);
```

每个窗口都能够调用其他窗口的方法，包括访问其 window、document、frames、location 等对象。对于跨窗口通信而言，也有一些限制，在绝大多数浏览器中，新打开的窗口是无法关闭原始窗口的。

警告

关闭原始窗口，将会导致用户的后退历史、打开的标签页、填写一半的字段等信息的丢失。那些允许新窗口关闭原始窗口的浏览器，在关闭之前也会弹出一个对话框要求用户进行确认。

当你拥有一个指向 window 对象的引用（通过 window 对象的 open 方法获得，或者通过 self 或 opener 方法获得）之后，就可以通过 focus 方法使它获得焦点，或者将焦点还给原始窗口。还有一个名为 blur 的方法，它将使得下一个窗口获得焦点：

```
newWindow.focus();
...
newWindow.blur();
```

使用 resizeBy 和 resizeTo 方法可以调整窗口的大小。resizeBy 方法指定的是相对于当前窗口宽度和高度的修改量，单位为像素：

```
newWindow.resizeBy(50,50);
```

resizeTo 方法则是重新设置窗口的宽度和高度，单位也是像素：

```
opener.resizeTo(100,100);
```

moveTo 是一个更有用的方法，它能够将窗口左上角移到(x,y)坐标指定的位置上：

```
    self.moveTo(x,y);
```

使用 moveTo 方法可以将上下文相关帮助窗口移到事件触发位置的边上。在示例 9.2 中，将打开一个仅有一个表单元素的页面；下面显示着红色的“Click for Help”（单击查看帮助信息）字符。在这个 script 元素中，添加了捕获 click 事件的事件监听器。当该页面打开时，用来输入名字的表单元素获得焦点。不过，在此没有提交按钮，因此用户肯定会单击“Click for Help”（单击查看帮助信息）来获取帮助。

示例 9.2　打开一个帮助窗口

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>Windows</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript">
//<![CDATA[

function catchEvent(eventObj, event, eventHandler) {
  if (eventObj.addEventListener) {
     eventObj.addEventListener(event, eventHandler,false);
   } else if (eventObj.attachEvent) {
     event = "on" + event;
     eventObj.attachEvent(event, eventHandler);
   }
}

catchEvent(window,"load",setupEvents);

function setupEvents(evnt) {

   document.forms[0].elements[0].focus();

   var evtObject = document.getElementById("panicbutton");
   catchEvent(evtObject,"click",openHelp);

}

function openHelp(x) {

   var optionString =
"width=200,height=150,menubar=no,toolbar=no,scrollbars=no,location=no,
resizeable=no";
   var helpWindow = window.open("help.html","test",optionString);
   helpWindow.focus();
   helpWindow.moveTo(x.screenX,x.screenY);
   return false;
}
```

```
//]]>
</script>
</head>
<body>
<form id="currentForm" action="">
<p>
Your name: <input type="text" size="50" />
</p>
</form>
<div id="panicbutton" style="width:100px;height:40px;background-color:#f00;
padding: 5px; margin-left: 50px">
<p>Click for Help</p>
</div>
</body>
</html>
```

这时将弹出一个最简单的窗口，其位置正好位于鼠标点击位置的右下角，如图 9.2 所示。它能够根据 click 事件的位置进行定位，那是因为该窗口打开时被移到了 click 事件触发的位置上。当窗体打开时，该窗口也就自然获得了焦点。

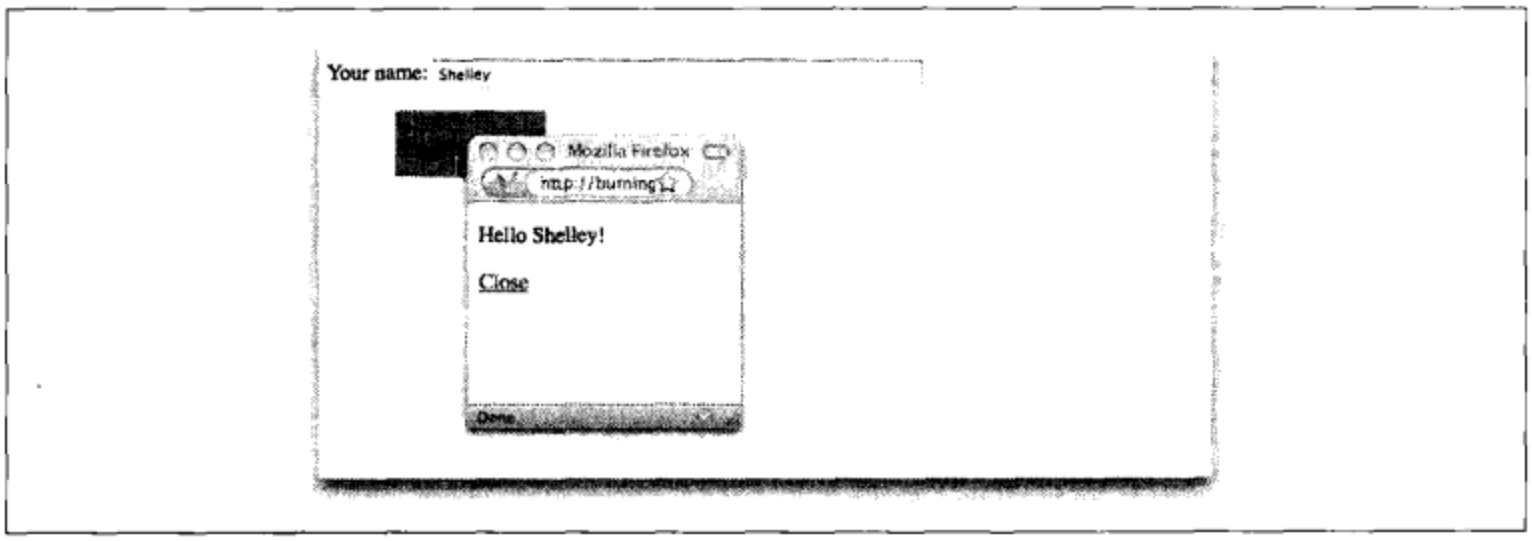


图 9.2　较小的、新打开的、位置合理的帮助窗口

示例 9.3 中列出的是该窗口显示的内容。它实际上将访问 opener 窗口（也就是打开它的窗口），查找表单元素，然后将其值复制过来。因此在该窗口中显示了相应的信息，并且提供了一个关闭该窗口的链接。

示例 9.3　已打开的窗口

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
    "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
    <html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
    <head>
    <title>Windows</title>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <script type="text/javascript">
    //<![CDATA[
```

```
window.onload=function() {

  var nmStr = opener.document.forms[0].elements[0].value;
  document.writeln("<p>Hello " +  nmStr + "!</p>");
  document.writeln("<p><a href='' onclick='self.close(); return
false'>Close</a></p>");
}

//]]>
</script>
</head>
<body>
<p>some content</p>
</body>
</html>
```

弹出式窗口通常用来向网站访问者推销一些东西，特别是那些页面没有很好链接上的信息。打开一个帮助窗口，使其显示在事件触发的位置上，然后与原始窗体进行信息通信是十分有用的。稍后，我们在第 12 章中将通过隐藏的页面元素来完成相同的效果，但现在，你也已经有了一种实现上下文相关帮助的手段。

到现在为止，我们还都只是和一个窗口打交道。不过，通过使用 frame（帧），就能够将浏览器窗口分成几块，然后为每一小块指定不同的 URL，承担不同的功能。

9.4 frame 对象

frame 对象中定义了以下属性：parent、length 和 name。对于跨帧通信而言，name 和 parent 属性特别重要。其父元素 frameset（帧集，包括帧的窗口）可以通过每个帧的名字访问所有子帧（也可以通过帧数组，以对象数量作为索引值）；每个帧都可以通过通用的关键字 parent 来访问该帧集。兄弟帧元素之间可以通过 parent 和对方的 name 属性来访问。

在示例 9.4 中，载入了一个包含两个帧的帧集。这两个帧分别被命名为 framea 和 frameb。

示例 9.4　载入两个帧的帧集

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Frameset//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-frameset.dtd">
    <html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
    <head>
<title>Frames</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
</head>
    <frameset cols="50%,*">
    <frame name="framea" src="framea.html" />
    <frame name="frameb" src="frameb.html" />
```

```
    </frameset>
    </html>
```

在 framea 中载入的文件是 framea.html。在该文档中有一个链接，点击该链接时将通过其 parent 属性访问其兄弟帧，然后将链接地址改成自身，如示例 9.5 所示。第二个帧载入的文件是 frameb.html，其报头和页面标题都不同，但其功能是一样的，只不过它将 framea 换成自身。

示例 9.5　每个帧都将载入自身

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>Frame A</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript">
//<![CDATA[

window.onload=function() {
  var theFrame = document.getElementById("thelink");
  theFrame.onclick = function () {
     parent.frameb.location.replace("framea.html");
  }
}

//]]>
</script>
</head>
<body>
<h1>Frame A</h1>
<p><a href="" id="thelink">Steal this page</a></p>
</body>
</html>
```

在每一个帧中，都使用了 location 的 replace 方法来载入自身的内容。图 9.3 展示了页面载入之后的情况；图 9.4 展示了单击 Frame B 中的链接之后的效果，它们两个正在竞争页面的主人。

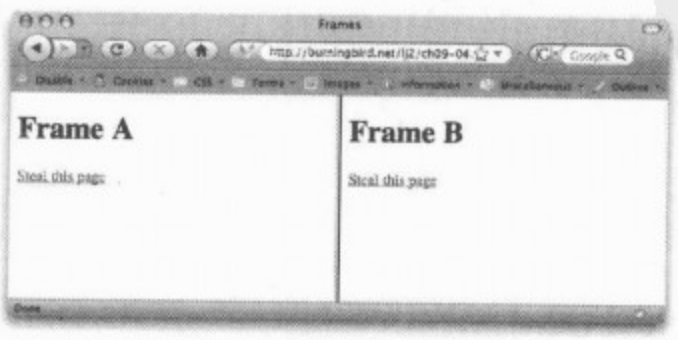


图 9.3　当 Frames 应用程序第一次载入时

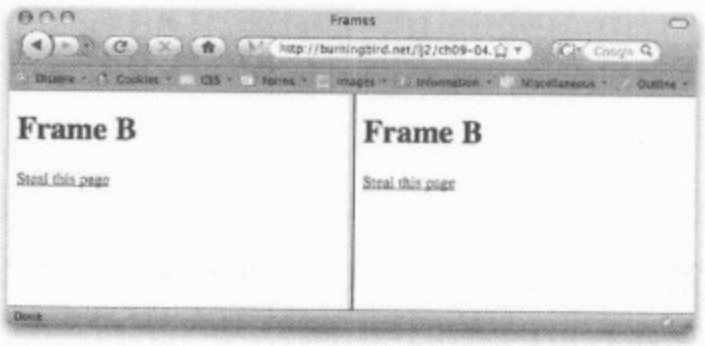


图 9.4　当单击 Frame B 中的链接之后

我并不太喜欢帧。是的，它们的确很有用，不过当左边窗口（或顶部窗口）会触发右边窗口（或底部窗口）的改变时，它仍然是一种很令人头疼的应用程序管理手段。每个帧都能够独立地进行滚动，我们无须付出什么工作。

不过，有太多公司曾经（或者现在仍然）喜欢在其网站中使用帧，而且通常会导致其他网站的内容嵌套显示在自己网站中。我们绝大多数人都不喜欢这样做。幸运的是，有了 JavaScript 之后，我们可以使用 window 的第二个对象 location 来代替它。

9.4.1　location 对象

location 对象的属性都是与页面位置相关的。在前面，你已经看到了它提供的 replace 功能，我们曾经用它来改变帧中所显示的页面。另一个是 reload 方法，它将使浏览器刷新文档。此外，还有一些与页面位置有关的属性，包括 host、port 和 protocol，如表 9.2 所示。

表 9.2　location 对象的属性

属　性	用　途
hash	形如 http://<some.com>/<somepage>#<somehash>格式的 URL，该属性中的“somehash”是#号后的值
host	URL 的主机名（域名）和端口号
hostname	主机名（域名）
href	完整的 URL（读取或写入）
pathname	域名之后的路径名
port	URL 中的端口号
protocol	URL 中使用的协议，如“http:”
search	查询字符串，如果存在则是从页面中派生的，它包含 URL 中第一个问号之后的内容
target	如果有的话，表示载入 URL 的目标元素名称

对于一个如“http://learningjavascript.info/ch09-01.htm?a=1”的 URL，其对应的 location

属性值为：

```
host/hostname: learningjavascript.info
protocol: http:
search: ?a=1
href: http://learningjavascript.info/ch09-01.htm?a=1
```

让我们回到与帧相关的第一个问题，帧在载入页面时是无须经你允许的，使用 location 对象（加上一些零碎的东西）可以解决该问题。

修改前面示例中的 frameb.html 文件，添加一个指向另一文件的链接，其内容如示例 9.6 所示。在该页面中有一个脚本块，负责检查该文档是否在最顶端的窗口中打开了，或者是否已经在帧集中打开了。在 JavaScript 中，如果某个窗口不是最顶端的（如果它是载入在一个帧中的），那么就将最顶端窗口的位置赋给自己，这样就能够有效地把帧集扔在一边。在不同的浏览器中，替换页面需要使用不同的功能，因此这段代码首先将检查有哪种替换方法可用，如果都没有，则使用备选的方法（直接设置 href 属性）。

示例 9.6　避免窗口在帧集中打开

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>Destroy all frames</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript">
//<![CDATA[

window.onload = function() {
if (self != top) {
   if (top.location.replace)
       top.location.replace(self.location.href);
    else
       top.location.href=self.document.href;
   }
}

//]]>
</script>
</head>
<body>
<p>Frames, no way</p>
</body>
</html>
```

这样的做法很简捷。不过，现在你已经很少会看到使用帧的页面了。帧不再像以前那样流行，绝大多数人也不希望为页面添加非必要的功能。

并不是所有帧都需要一个作为父元素的帧集。例如，你可以在页面中嵌入一个 iframe

而不是 frameset，我们接下来就将介绍它。

9.4.2 基于 iframe 的远程脚本

和标准的帧不一样，iframe 是内嵌在页面中的。你可以为其指定高度和宽度，如果将它们都设置成 0，那么它就将被隐藏起来。iframe 会把它嵌入到的页面视为自己的父元素，这也是它和更高层页面进行通信的方法。通常，你可以使用 document 的 getElementById 方法来访问它，也可以使用 target 属性载入其内容。

在客户端/服务器端开发领域，iframe 也拥有着战略性地位，因为 iframe 是实现远程脚本的重要参与者。基于远程脚本，客户端页面能够访问服务器端的远程服务，并且可以将这些服务的结果数据返回到客户端，而且无须重载客户端。现在，这一功能通常被称为 Ajax，绝大多数远程脚本都被改成通过一个特定的对象来实现，在第 14 章中你会看到更详细的信息。不过，你仍然可以通过 iframe 来完成远程脚本功能。

提示

使用 iframe 实现远程脚本的观点，最初是 Eric Costello 在 Apple Developer Network 发表的一篇文章中提出来的；该文章可以从 http://developer.apple.com/internet/webcontent/iframe.html 中下载到。

示例 9.7 中有一个 iframe，它是一个宽度值和高度值都被设置成 0 的隐藏 iframe。该页面中还有一个表单和两个单选按钮，每个单选按钮都指定了特定的颜色。单击任何一个按钮，就会显示与该颜色相关的文本字符串：红色的显示“Apple”，蓝色的显示“Sky”。

示例 9.7 基于内嵌的 iframe 进行通信

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
<title>Windows</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript">
//<![CDATA[

function catchEvent(eventObj, event, eventHandler) {
  if (eventObj.addEventListener) {
     eventObj.addEventListener(event, eventHandler,false);
  } else if (eventObj.attachEvent) {
     event = "on" + event;
     eventObj.attachEvent(event, eventHandler);
  }
}

catchEvent (window, "load", function() {
   catchEvent(document.forms[0], "click",colorChange);
```

```
   });

// 使用 iframe 的远程脚本
function colorChange() {
   var colors = document.forms[0].color;
   var result = document.getElementById('result');
   for (var i = 0; i < colors.length; i++) {
     if (colors[i].checked) {
        var myFrame = document.getElementById("myFrame");
        myFrame.contentWindow.location.href="service.php?color=" +
colors[i].value;
        myFrame.onload=function () {
           result.innerHTML =
myFrame.contentWindow.document.getElementById("val2").innerHTML;
        }
     }
   }
}

//]]>
</script>
</head>
<body>
<div>
  <form action="">
    <p>
      <input type="radio" name="color" value="red"/>Red<br />
      <input type="radio" name="color" value="blue" />Blue
    </p>
  </form>
</div>
<iframe
  name="myFrame"
  id="myFrame"
  style="width:0; height:0; border: 0"
  src="service.php"></iframe>
<div id="result">
</div>
</body>
</html>
```

当该页面载入时，iframe 的 source 属性被设置为一个名为 service.php 的 PHP 页面，其内容如示例 9.8 所示。当你单击任何一个单选按钮时，都会使 JavaScript 在 iframe 中重载这个 service.php 页面，不过这时还将把选中的单选按钮的值作为查询字符串传给它。根据该值将返回一个新的文档，如果选中的是红色的单选按钮，返回的是：

```
<div id='val1'><p>Rose</p></div>
<div id='val2'><p>Apple</p></div>
```

如果选中的是蓝色的单选按钮，那么将返回：

```
<div id='val1'><p>Berry</p></div>
<div id='val2'><p>Sky</p></div>
```

如果要访问与 iframe 关联的 window 对象，那么需要引入一个名为 contentWindow 的新属性，它用来表示 iframe 的 window 对象。你可以使用 contentWindow 方法捕获针对 iframe 的 onload 事件句柄。

当该帧载入完成时，另一个函数将访问新载入的标识符为 val2 的元素，然后生成针对该元素的 HTM 代码，再赋给父页面上的目标 div 元素。

示例 9.8　负责处理颜色请求的 PHP 应用程序页面

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
</head>
<body>
<?php

$color = $_GET['color'];
if ($color == 'red') {
   printf("<div id='val1'>Rose</div>");
   printf("<div id='val2'>Apple</div>");
} else if ($color == 'blue') {
   printf("<div id='val1'>Berry</div>");
   printf("<div id='val2'>Sky</div>");
}

?>
</body>
</html>
```

这样看起来，我们为了得到这点结果花费了大量的努力，不过该示例不仅演示了传统的远程脚本技术，还间接地演示了在 Ajax 应用程序中所采用的现代技术，它对如何维护动态页面效果和回退按钮之间的关联有兴趣。另外，该应用程序演示了一些当前 JavaScript 开发人员在切换不同版本 HTML 和 XHTML 时经常面临的问题。

首先，该示例的 DOCTYPE 是 XHTML transitional（传统 XHTML），因为 iframe 元素在 XHTML 1.0 strict（严格的 XHTML）中被抛弃了。因此，为什么还要使用该元素？如果你载入该页面，先单击蓝色按钮然后单击红色按钮，虽然父页面没有重载，你仍然会发现浏览器触发了 iframe 的 load 事件。如果你单击浏览器上的后退和前进按钮，那么页面将反映你的操作“历史”，虽然 iframe 所在的页面并未重新载入。如果你使用的是 Firefox 浏览器，那么只要在其 Firebug 中打开 Net communication（网络通信）窗

口就会更加清晰地看到这个过程，如图 9.5 所示。单击几次后退和前进按钮，将会看到每次都将生成一个网络调用，以及针对这次后退按钮操作的历史记录。

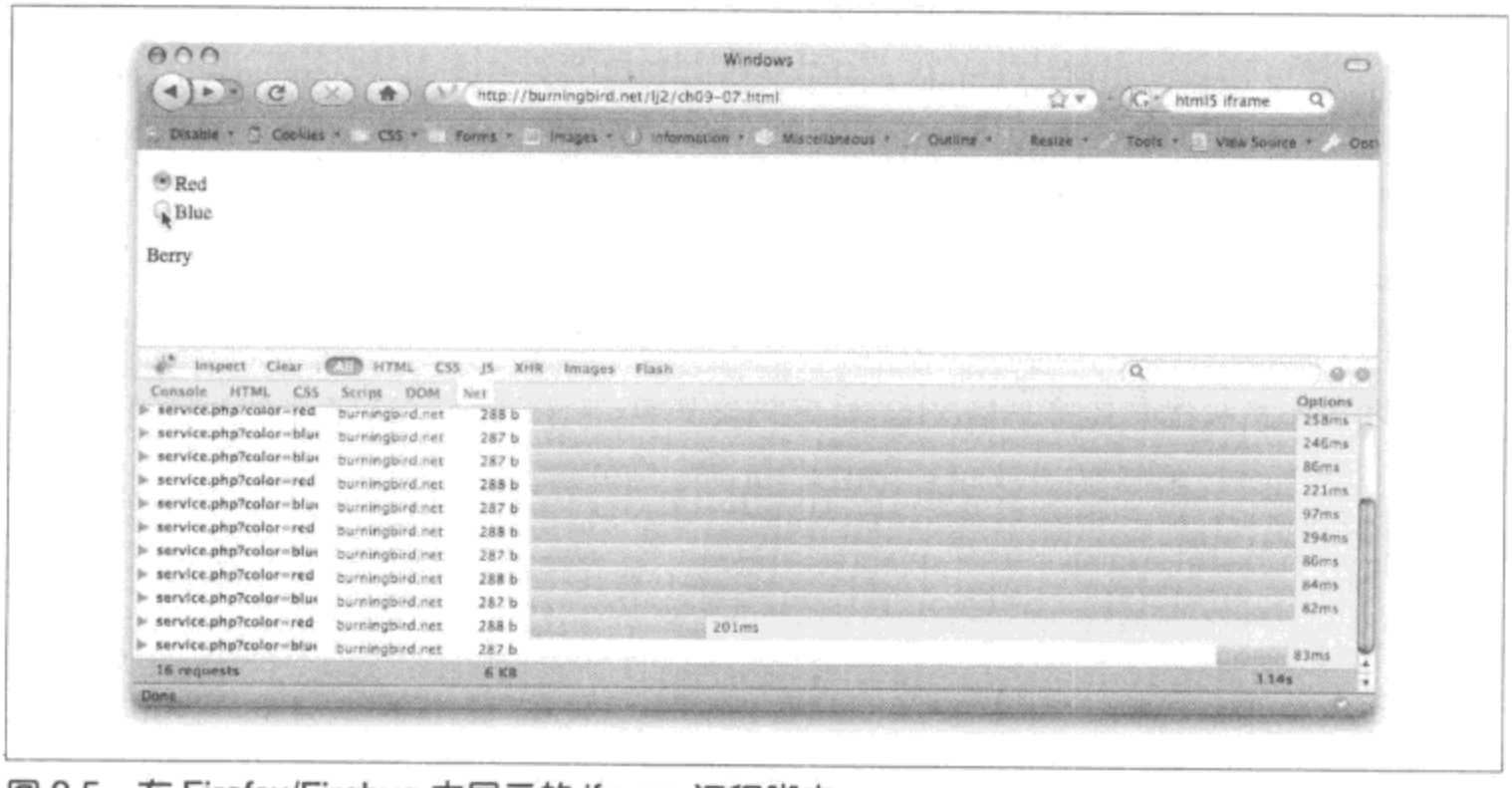


图 9.5　在 Firefox/Firebug 中展示的 iframe 远程脚本

载入一个新的 iframe，将在浏览器的后退按钮记录中添加一项。现代的 Ajax 应用正是发挥了它的优点，将 Ajax 应用程序的状态及时地保存为一项历史记录。然后我们更新 iframe，传入一个值，通常是一个数字，正如示例 9.7 所做的那样，只不过此时该值是用来表示其状态的。你可以从 iframe 中获取其状态值，就像示例 9.7 中的与颜色相关的值那样，然后通过它查询缓冲区中的状态，以便恢复相应的页面。

虽然通过 iframe 来维护 Ajax 应用程序状态的具体实现超出了本书讨论的范围，不过该示例还是说明了在今天的应用程序中为什么仍然有人使用 iframe，即使它已经被 XHTML 标准弃用了。另外，虽然该元素在 XHTML 1.0 和 1.1 中被弃用了，但 iframe 仍然是 X/HTML 5.0 标准的一部分。当然，HTML 5.0 中的其他修改使得记录操作历史已经不再依赖于 iframe，但我们在本书第 3 版时仍然会保留这方面的内容。

现在，我们还是先回到本书的主题，接下来将讨论下一组对象：timer（定时器）。

9.5　添加并控制定时器

location 对象的属性中有两类定时器：一类是一次性的，另一类是周期性使用的。两种定时器都能取消，一次性定时器方法只会被调用一次。

要想创建一个不重复触发的定时器，可以使用 setTimeout 方法。调用它时至少要指定两个参数：第一个是函数文字量或函数名称，该函数将在定时器指定的延迟时间到了

的时候执行；另一个是以毫秒为单位的延迟时间。如果需要向指定的函数传入参数，那么它们应该顺序列在此次调用的前两个参数后面，并用逗号分隔。该方法将返回该定时器的标识符：

```
var tmOut = setTimeout(func, 5000,"param1",param2,...,paramn);
```

如果想清除这个定时器，可以使用 clearTimeout 方法：

```
clearTimeout(tmOut);
```

如果你想周期性地使用这个定时器，那么应调用 setInterval 方法。调用它时也至少要指定两个参数：函数名称和定时器周期间隔。和 setTimeout 方法一样，它将返回一个标识符：

```
var tmOut = setInterval("functionName", 5000);
```

同样，如果想暂停或取消这个周期性定时器，可以使用 clearInterval 方法。如果你想实现一个周期性定时器，但又想在参数中指定一个函数文字量，那么你可以在每次定时器过期时再用 setTimeout 函数重新设置这个定时器。

警告

在 IE 浏览器中，setInterval 和 setTimeout 方法是不支持在最后添加函数调用所需参数的。示例 9.9 展示了解决这一限制的方法，使你能够向该函数传递所需的参数。

示例 9.9 使用了一个由定时器触发的函数，它将修改 div 元素的背景颜色，使其出现从白到黄的渐变效果。这与你在 Ajax 应用程序中经常看到的、用于提示用户页面上的修改或吸引用户注意时常用的“褪黄”效果类似。在下一章中，我们将更进一步地介绍动态 Web 页面机制。现在，我们仅关注定时器。

示例 9.9 使用定时器为 div 元素创建“闪动”效果

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>Timers</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<style type="text/css">
#block
{
   width: 400px;
}
</style>
<script type="text/javascript">
//<![CDATA[

window.onload=function () {
```

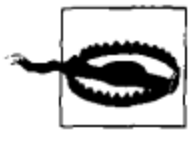
```
    document.getElementById("block").style.backgroundColor="#ffffff";
    setTimeout("colorFlash(255)",300);
}

function colorFlash(newColor) {
    var hexVal = newColor.toString(16);

    // 确保长度为 2
    if (hexVal.length < 2) {
      hexVal= "0" + hexVal;
    }

    var colorString = "#ffff" + hexVal;
    var blockDiv = document.getElementById("block");
    blockDiv.style.backgroundColor=colorString;
    if (newColor > 0) {
       newColor-=5;
       setTimeout("colorFlash(" + newColor + ")", 50);
    }
}

//]]>
</script>
</head>
<body>
<div id="block">
<h1>Hello</h1>
</div>
</body>
</html>
```

要实现颜色的淡出或闪烁效果，通常只需要修改一个参数，该参数将从红-蓝-绿色谱中选择一个颜色。在本例中，红色和绿色 channel（通道）都采用 255（十六进制表示是 ff），但需要对蓝色通道进行修改，使其值从 255 一直改变成 0。

提示

这里检查了十六进制的长度，看它是否超过两个字符。如果不超过，那么就补上 0，因为用十六进制表示的颜色值不能只有 5 位。

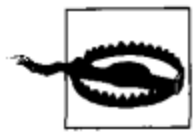
第一次调用 setTimeout 方法时，还调用了 colorFlash 方法，并将 255 作为参数传入，它将作为蓝色通道值。这样 div 元素的背景色就将被修改，接着蓝色通道值将逐渐递减，然后将其添加到函数中并传给 setTimeout 方法。这是一种在每次迭代中修改定时器的方法，这样就不需要使用一个全局变量来保存蓝色通道的当前值。



警告

本示例在 IE 8.0 beta 2 中载入时会呈现出不稳定的行为，因此我不得不切换到 Compatibility View（兼容视图）模式以获得可靠的执行结果。

该应用程序也有一个结束点，也就是当蓝色通道值小于 0 时。在那时，定时器将不会再被调用，因为已经到了最终的颜色。另一种方法是使用 setInterval 方法，然后在到达最终颜色时清除该定时器。



警告

如果你想避免使用任何类型的定时器操作，那么可以使用 document.write 或其他方法来修改 document 对象中的标签。这将使页面处于不稳定的状态。因为修改文档中的一部分总是要比修改整个文档好些。

9.6 history、screen 和 navigator 对象

我们将 BOM 对象中最重要的 document 对象放在最后，现在先来看看 history、screen 和 navigator 对象。

9.6.1 history 对象

history 对象负责维护浏览器中页面载入操作的历史记录。同样，其方法和属性能够完成通过浏览器的后退和前进按钮所能实现的导航操作。

你可以使用如 next 和 previous，或者 back 和 forward 之类的属性来遍历历史记录。使用 current 可以找到当前页面，并找到历史记录的长度（保存在历史记录缓冲区中的页面数）。你也可以通过 go 方法转到特定的页面，只需将页码作为参数传入，如果该值为负数则表示后退几页：

```
history.go(-3);
```

如果是正数则表示前进几页：

```
history.go(3);
```

正如大家所想的那样，history 对象会自己处理好自己的事；你作为一个页面开发人员，是无须关心它的具体工作机制的。只有当你使用 Ajax 页面效果时，它才成为你需要关心的东西，因为它们采用的页面载入模式是不一样的，正如本章前面的“基于 iframe 的远程脚本”小节中所说的那样。

9.6.2 screen 对象

screen 对象中所包含的信息是与屏幕显示有关的，包括其宽度、高度，以及颜色或像素浓淡。虽然它们不是很常用，但它对于那些需要修改浏览器窗口大小、创建需要特定调色板的带色彩对象等功能而言是个不错的引用。

它具体支持的属性会随着浏览器以及它们的版本改变而发生改变。不过，screen 对象至少能够支持以下属性：

- availTop（或 top） 窗口可以放置的最顶部位置，以像素表示；
- availLeft（或 left） 窗口可以放置的最左边位置，以像素表示；
- availWidth（或 width） 屏幕的宽度，单位为像素；
- availHeight（或 height） 屏幕的高度，单位为像素；
- colorDepth 屏幕的颜色深度；
- pixelDepth 屏幕上每个像素的位元数。

实际高度/宽度与可用高度/宽度之间存在一些差异，主要看工具条是放在顶部、底部还是边上。

在早期的动态页面实现上，开发人员会检查屏幕的色深，然后将图像修改成为与配置更吻合的低分辨率图像。不过现在很便宜的显示器所支持的色深都超过了 8 个像素，绝大多数都能够支持真彩色[1]。同样，花费一些额外的工作来实现根据屏幕情况返回图像的功能已经不值得了。不过，色深还会影响你在样式设置中使用的颜色，因此它还是很有用的信息，而当你在进行页面布局设计时，其 width 和 height 属性也是有用的。

9.6.3 navigator 对象

navigator 对象中提供的是和浏览器或其他访问该页面的用户代理相关的信息。通过它可以检查操作系统、浏览器或浏览器族、安全策略、语言以及 cookie 是否启用。有些浏览器还提供了一个数组，用来保存当前已安装的插件名称，此外还有一些针对特定用户代理的特殊属性。

navigator 对象通常支持以下属性：

- appCodeName 浏览器源代码基础的名称；
- appName 浏览器的名称；
- appMinorVersion 浏览器的子版本号（如 1.52 版本的子版本号就是 52）；
- appVersion 浏览器的主版本号（如 1.52 版本的主版本号就是 1.00）；
- cookieEnabled cookie 是否启用；
- mimeTypes 它是一个数组，用来保存浏览器所支持的 MIME 类型；
- onLine 用户是否联机；
- platform 浏览器运行的平台；

[1] 译者注：8 像素色深就是 8 位色，即 256 色；而真彩色通常是指 32 位色。

- plugins　它是一个数组，用来保存浏览器所安装的插件；
- userAgent　针对浏览器（或其他用户代理）的、完整的用户代理描述；
- userLanguage　浏览器支持的语言。

mimeType 集合是由 mimeType 对象组成的，它的属性包括 description、type 和 plugin。plugins 集合是由 plug-in 对象和它自己的 mimeType 数组属性组成：description、filename、mimeType 数组长度和 plug-in 名称。

有些浏览器还支持以下方法：用来检查浏览器是否启用了 Java 的 javaEnable；获取、设置浏览器偏好设置的 preference；检查数据污点检查（一种安全特性）是否启用的 taintEnabled。

你有可能永远都用不到这些集合。实际上，访问 navigator 对象通常是为了检查 cookie 是否启用，或者想了解用户是通过哪种浏览器访问页面的。

9.6.4　history、screen 和 navigator 属性的实际应用

在示例 9.10 所实现的页面中，将访问前面介绍的 3 个 window 属性（包括 history、screen 和 navigator），输出这些属性值并提供一组测试历史记录的选项。然后在不同浏览器、不同操作系统中尝试该页面，看它们是否被支持。

示例 9.10　探索 history、screen 和 navigator 对象

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>Timers</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript">
//<![CDATA[

window.onload=function () {
document.writeln("<h1>screen object</h1><p>");
document.writeln("screen.availTop: " + screen.availTop + "<br />");
document.writeln("screen.availLeft: " + screen.availLeft + "<br />");
document.writeln("screen.availWidth: " + screen.availWidth + "<br />");
document.writeln("screen.availHeight: " + screen.availHeight + "<br />");
document.writeln("screen.colorDepth: " + screen.colorDepth + "<br />");
document.writeln("screen.pixelDepth: " + screen.pixelDepth + "<br /></p>");

document.writeln("<h1>navigator object</h1><p>");

document.writeln("navigator.userAgent: " + navigator.userAgent + "<br />");
document.writeln("navigator.appName: " + navigator.appName + "<br />");
document.writeln("navigator.appCodeName: " + navigator.appCodeName + "<br />");
```

```
document.writeln("navigator.appVersion: " + navigator.appVersion + "<br />");
document.writeln("navigator.appMinorVersion: " + navigator.appMinorVersion + "<br/
>");
document.writeln("navigator.platform: " + navigator.platform + "<br />");
document.writeln("navigator.cookieEnabled: " + navigator.cookieEnabled + "<br
/>");
document.writeln("navigator.onLine: " + navigator.onLine + "<br />");
document.writeln("navigator.userLanguage:"+ navigator.userLanguage +"<br />");

if (navigator.mimeTypes) {
   document.writeln("navigator.mimeTypes[1].description: " +navigator.mimeTypes
[1].description + "<br />");
   document.writeln("navigator.mimeTypes[1].type: " + navigator.mimeTypes[1]. type
+"<br />");
}

if (navigator.plugins) {
   for (var i = 0; i < navigator.plugins.length; i++) {
      document.writeln("navigator.plugins[i].description: " +navigator.
plugins[i].description + "<br />");
   }
}

document.writeln("</p>");
document.onclick=function () {
   history.back();
   return false;
}
}

//]]>
</script>
</head>
<body>
<h1>Hello</h1>
</body>
</html>
```

你可能会对其中一些集合（如 plugins）的输出感到惊讶，如图 9.6 所示。我记得当时看到一个关于数字权限管理的插件时就感到惊讶，因为我不记得我什么时候安装过该插件。

对于 mimeType 对象而言，有些浏览器还支持带后缀的属性，如*.html 等。

刚才演示的这 3 个对象，是能够直接通过 window 对象访问的最后几个对象，此外还有一个对象。那就是在本章中要介绍的最后一个对象，它就是我们的老朋友 document 对象。从某种程度上，本书余下的绝大部分内容都将集中在 document 对象上。不过，在我们从更现代的 DOM Levels 1（及后续版本）角度介绍它之前，还是先从 BOM 角度谈谈它。

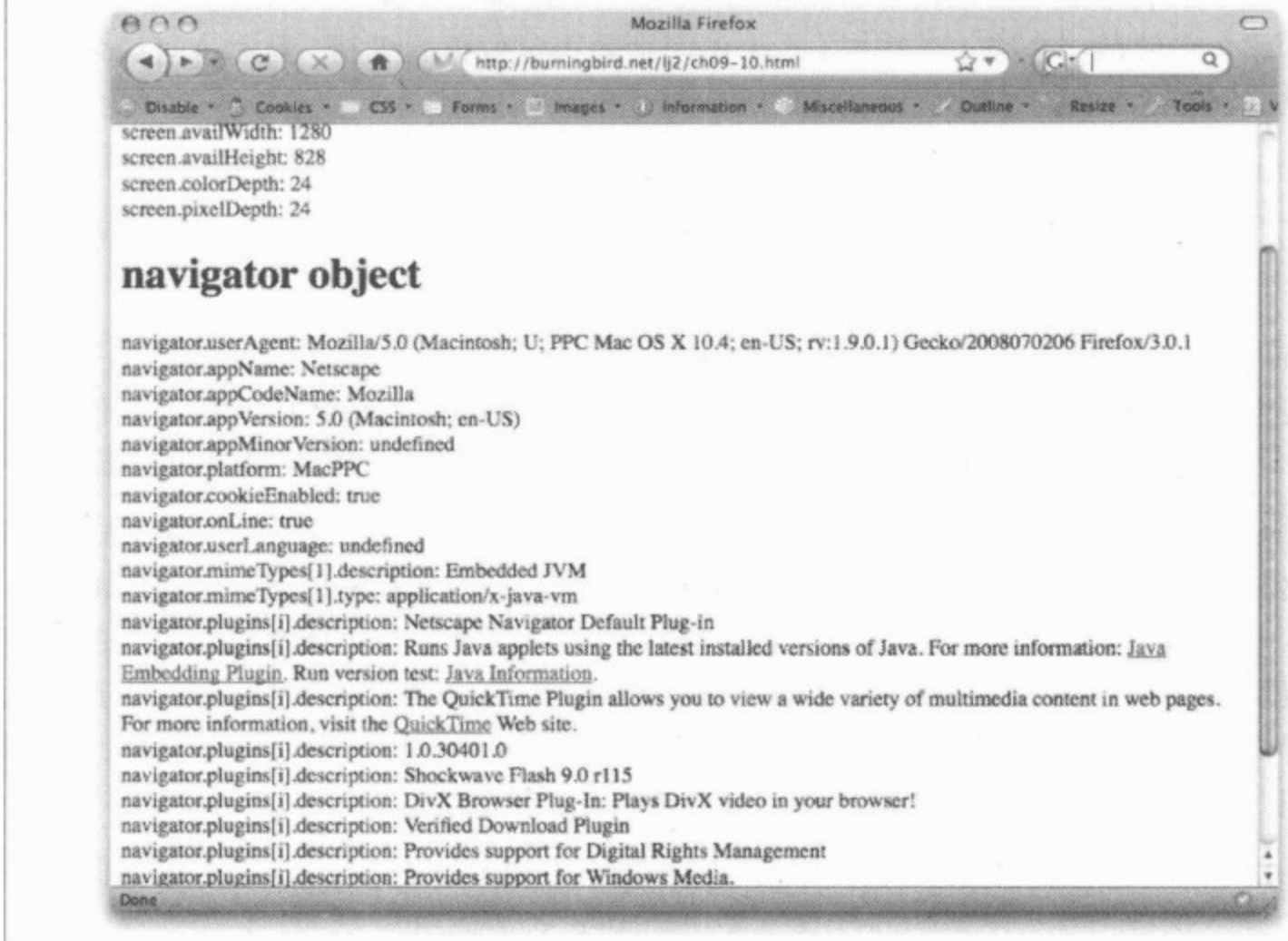


图 9.6　示例 9.10 实现的应用程序的输出

9.7　document 对象

我们首先回顾一下在本章最开始给出的图 9.1，从该图中你会看到 document 对象为访问浏览器页面中所包含的其他元素提供了一种途径。这些元素包括以前介绍过的 form 及表单元素，此外还包括页面中图像、链接、嵌入式对象的集合；实际上在页面边界之内的所有元素都将把 document 元素视为其父元素。较早的 document 元素变体还包括名为 layer 的其他集合，以及在各种浏览器的新近版本中所共有的 style 属性，不过这一切都会让你感受到 document 元素对于动态页面开发的重要性。

之前的章节中已经介绍过 document 对象的 getElementById、getElementByTagName 以及 writeln 方法；第 10 章还将介绍使用通用方法访问所有页面元素的方法。现在，我们将关注通过不同的 document 集合访问页面元素的老方法，主要介绍链接、图像等流行的元素。

9.7.1　链接

链接和锚（anchor）之间的差别在于使用了不同类型的属性。它们都是基于 anchor 标

签的（<a>）。但时，如果指定了 href 属性，那么它就是一个链接；如果指定了 thename 属性，那么它就是一个锚，它提供了一个访问页面指定位置的机制。当你访问某个 Web 页面时，可能会看到了如下所示的 URL：

```
http://<someco.com>/index.html#one
```

#号和“one”用来指定一个页面片段。当访问这个 URL 时，浏览器在载入相应 Web 页面的同时会自动滚动到一个 name 属性指定为“one”的链接上。

document 对象的 links 集合是由页面中所有超链接组成的，它的访问方法和数组一样，首先是页面上的第一个链接，然后是沿着右下方向所找到的后续链接。不过，你也可以为每个超链接添加一个标识符，这样就能通过这个标识符来访问数组中的链接。

在这个集合中的每一项都是一个 link 对象，它也有自己的属性。它的属性有些和 location 对象的属性类似：host、protocol、port、search 和 hash，每个都将返回超链接中的特定片段。你也可以通过 href 属性访问完整的链接，还可以通过 text 属性显示相关的链接对象（文本显示）。如果你想将一个 Web 页面上的所有链接放到一个工具条中，那么这是一个很有用的属性。只是你应该确保输出的链接不要写在同一个页面上，因为如果在处理原有链接的同时添加一个新链接将会把浏览器搞糊涂。

在示例 9.11 中，页面上有 3 个带文本信息的链接。在此将通过 document 对象访问这个链接集合，然后把每个链接的 href 属性输出到 alert 对话框中。

示例 9.11　从页面中获取链接

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>Links</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript">
//<![CDATA[

window.onload=function () {
   var links = "";
   for (var i = 0; i < document.links.length; i++) {
     links = links + document.links[i].href + "\n\n";
   }
   alert(links);
}

//]]>
</script>
</head>
<body>
<p>The <a
```

```
href="http://msdn.microsoft.com/workshop/author/dhtml/reference/objects/
link.asp">links</a>
collection off of the document
<a href="http://www.w3.org/TR/html4/struct/objects.html">object</a> consists of
all
hypertext links in the page, accessible as an array, starting with the first
link in
the page and moving down and to the right. However, you can also add an identifier
for each hypertext link and access it in the array through this identifier. </p>
<p>Each item in the collection is a
<a href="http://www.devguru.com/Technologies/ecmascript/quickref/link.html">
link</a>
object, which has properties of its own. Among these are those similar to what we
found with location: host, protocol, port, search, and hash, each of which returns
that specific piece of the hypertext link. You can also access the complete link
through the href property, and the associated linked object (text) through text. This
can be handy if you're pulling links from a document in a web page, into a handy
sidebar reference or other functionality such as this.
</p>
</body>
</html>
```

当在浏览器中打开该页面时，就会弹出一个显示其链接的 alert 对话框。

9.7.2 图像

最早的动态页面开发技术就是用来修改文档中的图像信息的。对于以幻灯片形式播放相片的应用程序而言，该技术仍然是十分流行的，它就是通过 document 对象的图像集合实现的。

和链接一样，图像也有其对应的对象，也可以直接设置它们的属性，如表示图像 URL 的 src 属性。你也可以使用 new 构造器创建一个新的图像实例：

```
var newImage = new Image();
newImage.src="someimage.png";
```

示例 9.12 创建了一个以幻灯片形式播放本书附图（图 9.1～图 9.5）的应用程序。在该程序中，我们用了两个 div 元素来实现“next”（下一幅）和“previous”（上一幅）按钮，以提供遍历附图功能。这些图像是预装载在一个数组中的，点击任何一个 div 元素都会改变当前 img 元素的 src 属性，将其替换成数组中的下一幅（或上一幅）。修改 img 元素的 src 属性将使 Web 页面上显示出新的图像。

示例 9.12　使用图像集合创建的幻灯片播放程序

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>Slideshow</title>
```

```
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript">
//<![CDATA[

var currentPhoto = 0;
var pics = new Array();

window.onload=function() {
   for (var i = 0; i < 5; i++) {
      pics[i] = new Image();
   }
   pics[0].src = "fig0901.png";
   pics[1].src = "fig0902.png";
   pics[2].src = "fig0903.png";
   pics[3].src = "fig0904.png";
   pics[4].src = "fig0905.png";

   document.getElementById("next").onclick=nextPic;
   document.getElementById("prev").onclick=prevPic;
}

function changePhoto(photo) {
   document.images[0].src = pics[photo].src;
}

function nextPic() {
  currentPhoto++;
  if (currentPhoto < pics.length) {
    changePhoto(currentPhoto);
  } else {
    alert("at the end of the photo list");
  }
}

function prevPic() {
  if (currentPhoto > 0) {
    currentPhoto--;
    changePhoto(currentPhoto);
  } else {
    alert("at the beginning of the photo list");
  }
}

//]]>
</script>
</head>
<body>
<div id="next"><span>Next</span></div>
<div id="prev"><span>Previous</span></div>
```

```
<img src="fig0901.png" />
</body>
</html>
```

和前一个关于链接的示例一样，该示例倾向于模糊 DOM Levels 0 和 Levels 1 之间的区别。不过，它仍然能够在绝大多数主流 Web 浏览器上正常运行，这才是我们最重要的意图。

另外，注意在示例 9.12 中我们可以修改图像的 src 属性。这和示例 9.11 不同，它仅是输出链接的属性。图像的 src 属性是可读写的，而链接的属性是只读的。不过，我们仍然有办法对所有页面元素进行调整，在接下来的小节中将介绍它。

9.8 innerHTML

在过去几年中，不同浏览器提供了许多不同的属性和方法，以便通过脚本修改 Web 页面。而 DOM 的出现基本取代了它们。不过，现代浏览器仍然对 document 对象中 innerHTML 属性提供了支持。

使用 innerHTML 属性可以修改页面中任何一个 HTML 元素。它之所以仍然流行，是因为通过它修改页面元素时无须构建整个页面的内容，你只需要创建一个 HTML 格式的字符串，然后通过 innerHTML 就可以添加到 Web 页面中。不过，使用 innerHTML 意味着无论向 Web 页面添加了什么，它们都无法融合到页面的 document 树上，因此如果你混合使用 innerHTML 和新的 DOM 方法将会带来很大的破坏。

为了展示 innerHTML 的使用方法，在示例 9.13 中实现了一个由 3 个 div 元素组成的 Web 页面，每个 div 元素中还有其他 HTML 标签。第一个 div 元素中是一个段落（<p> 标签）；第二 div 元素中是一个无序列表；第三个 div 元素中是一个超链接。当载入该页面时，我们将通过 document 对象的 getElemnetById 方法来访问这些元素，使用 innerHTML 来替换它们的内容。这些处理将由一个定时器事件触发，因此你可以看到页面修改之前的样子。

示例 9.13　访问已命名的元素并使用 innerHTML 属性修改它

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>innerHTML</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript">
//<![CDATA[

window.onload = function() {

   setTimeout("changeElements()", 3000);
}
```

```
function changeElements() {
   var elem1 = document.getElementById("elem1");
   var elem2 = document.getElementById("elem2");
   var elem3 = document.getElementById("elem3");

   var tmp = elem1.innerHTML;
   elem1.innerHTML = elem3.innerHTML;
   elem3.innerHTML = elem2.innerHTML;
   elem2.innerHTML = tmp;
}

//]]>
</script>
</head>
<body>
<div id="elem1">
<p>Paragraph text.</p>
</div>
<div id="elem2">
<ul>
<li>option 1</li>
<li>option 2</li>
</ul>
</div>
<div id="elem3">
<p><a href="ch09-12.html">Example 9-12</a></p>
</div>
</body>
</html>
```

所有已标识的 HTML 元素都具有 innerHTML 属性。它是一个可读写的属性，这就意味着可访问、可修改甚至完全替换。不过最令人喜欢的是它不会影响源文件。如果你查看页面的源代码，其显示的 HTML 元素将对应于动态修改之前的 Web 网页。不过，如果你使用 Firefox 中的 Web Developer toolkit 的话，那么可以通过它提供的 View Generated Source（查看生成的源代码）功能显示出对应于动态修改之后的页面源代码。

所有主流的浏览器都支持 innerHTML，虽然每种浏览器在实现上都有一些细节上的变化（这也就是为什么将其应用于生产环境之前必须做一些测试）。W3C 也弃用了 innerHTML，浏览器仍然支持的原因是它已经被广泛应用，另外和 DOM 方法相比，实现相同任务它更加简单。值得一提的是，在本书写作时，HTML 5.0 仍然是支持 innerHTML 的。

在 document 对象提供的各个集合中，本章尚未谈及的是 cookies 集合，我们将在第 10 章中更详细地介绍它，我们的重点是介绍不同的客户端存储技术。

9.9 知识测验

1. 如果你想得到一个文本输入，那么应该使用哪种对话框？在程序代码中使用这个对话框，让用户输入其姓氏。
2. 定义一个定时器，使其隔 3000 毫秒执行一次 callFunction 方法，并且传入两个参数：paramA 和 paramB。
3. 如果要检查 cookies 是否启用，应该使用什么样的程序代码？
4. 创建一个大小为 200×200 像素的、不带工具条或状态条的新窗口，并在该窗口中打开 Google 的搜索页面。
5. 访问一个 Web 页面上的 form 元素有哪些方法？

9.10 测验答案

1. 如果想获得一个文本输入应该使用 prompt 对话框。其解决方案为：

```
var firstName = prompt("Enter your first name", "");
```

2. 以下就是该定时器的定义：

```
setTimeout(callFunction, 3000, paramA, paramB);
```

3. 你可以通过 navigator 对象来检查 cookie 是否启用：

```
if (navigator.cookieEnabled) ...
```

4. 其解决方案为：

```
var newWindow =
window.open("http://www.google.com","","width=200,height=200,toolbar=
no,status=no");
```

5. 如果这个 form 元素有标识符，可以使用 document.getElementById 方法。你还可以通过 document 对象的 forms 集合来访问一个 form 元素。

第 10 章

cookie 和其他客户端存储技术

最初，JavaScript 的方向并不是提供一个用来创建大型、复杂应用程序的工具，也不打算与服务器端实现独立的通信。在客户端保存信息的方法，最初就是通过众所周知的 cookie 对象。

cookie 是保存在客户端的一小块数据，它是基于键值信息的，由服务器端提供，JavaScript 开发人员可以在会话期间（直到浏览器关闭为止）或不同会话之间（Web 访问）对一些信息做持久化存储。其最初的概念是只有 Web 页面所在域的相关页面能够获取或写入 cookie，因此这些信息应该是安全的。基于这一假设，JavaScript 曾经用它来持久化存储各种信息，包括个人的登录名、密码甚至是购物车中的内容。现在的商业网站很少不使用 cookie 的，不管你愿意与否。

随着时间的推移，cookie 的安全性受到了破坏，同时从隐私的角度考虑，JavaScript 的 cookie 的名声也在逐步下降。大家出于保护隐私的考虑，经常会在浏览器中关掉 cookie 支持。不过 cookie 到现在还是很流行的，如果没被滥用，它还是十分有用的技术。

不过，cookie 也有一些限制，还不仅仅只是安全方面的问题。cookie 的存储总量和访问速度都无法通过大型 Ajax 应用程序的测试。在过去几年中，新的客户端数据存储方法已经开发出来，在本章最后我们也将对它们做一些简要的介绍。

首先让我们回到 cookie 和其安全性问题上来。

10.1 JavaScript 沙箱与 cookie 安全

现在有一些浏览器专有的安全选项，如使用已签名的脚本，不过 JavaScript 开发人员通常更喜欢使用 JavaScript 固有的安全策略。其中之一就是 JavaScript 沙箱，它有许多规则，其中包括禁止文件访问，也就是在 JavaScript 中不能直接打开、创建或删除操作系统中的文件。另外，它只提供了很底层的网络功能，如 Web 页面载入。最后，基于

JavaScript 沙箱，JavaScript 应用程序初始化一个访问其他网站的连接是受限的，数据传输是禁止的，这也就是众所周知的同源安全策略（same-origin security policy）。

10.1.1 同源安全策略

同源安全策略确保了不同域名、协议或端口的页面之间不能够通过脚本进行通信。同源安全策略将应用于不同页面之间的通信，包括父窗口中的表单和内嵌窗口之间的通信，如帧及 iframe。

为什么这样的限制很重要呢？如果某个网站弹出了一个小型的窗口，而且一直隐藏在你的主页之后，那么当你继续访问其他网站（如你的银行网站）时，这个弹出窗口的 JavaScript 就可以监听到你在另一个页面上的活动。而同源安全策略通过禁止页面中的 JavaScript 代码访问另一个域名中的页面（以及该页面下的 DOM 或其他资源），从而避免了这种类型的监听。

下面就通过一个例子来帮助你建立对同源安全策略的了解，如果你打开了一个位于 http://www.somecompany.com 的页面，在该页面中尝试访问以下域名中的页面：

- http://othercompany.com　由于 somecompany.com 和 othercompany.com 是两个不同的域名，因此这样的访问将失败；
- https://www.somecompany.com　由于这里采用的是 https 协议，和之前的 http 协议不同，因此这样的访问将失败；
- http://www.somecompany.com:8080　由于在原先访问的 URL 中没有指定端口（那么它将使用默认端口，通常是 80），和这里的 8080 是不一样的，所以这样的访问将失败；
- http://other.somecompany.com　由于这里使用了不同的主机名，和前面的主机名不同，因此这样的访问也将失败。

如果你之前访问的是没有指定任何子域名的 http://somecompany.com，那么当你访问如 www.somecompany.com 和 other.somecompany.com 之类的子域名时将成功。

10.1.2 使用 document.domain

网站开发人员可以解决同源问题。它可以使用相同的域名、不同的主机名（也就是子域名）来避开这一限制，如使用 about.somecompany.com 和 help.somecompany.com，而且现在这样的做法也很流行，那么同源限制就会受到影响。另外还可以通过 document 对象中的一个名为 domain 的属性来解决该限制，它可以设置成只允许子域名中的页面之间进行通信，不过只允许的是子域名中的页面，也就是只当 document 对象的属性和原先的主机名匹配时。

如果包含 JavaScript 代码的页面是通过形如 http://admin.somecompany.com 的 URL 访问

的，那么可以将 document.domain 设置为 somecompany.com，它仍是最初访问的域名。你不能将其设置成 othercompany.com，因为它是一个不同的域名。

以下设置是有效的：

```
document.domain = "somecompany.com";
```

以下的设置是无效的：

```
document.domain = "othercompany.com";
```

当设置完成之后，在 admin.somecompany.com 中的页面上的 JavaScript 代码将能够与 help.somecompany.com 中的页面进行通信。

幸运的是，对于从其他域名中链接进来的脚本是不使用同源安全策略的。你可以从任何地方链接所需的脚本，虽然 JavaScript 是在该页面中执行的，但会被自动做一些处理，使其包含后续通信所需的相同域名。如果没有这种从其他域名中链接脚本的机制，那么就无法使用 Google Maps 来实现所需的功能：

```
<script src="http://maps.google.com/maps?file=api&v=2&key=yourkey"
    type="text/javascript"></script>
```

不过同源安全策略仍将会应用于 cookie 的实现。

10.2 cookie 全解

为什么叫 cookie？cookie 这一名字最早源于 magic cookie 一词，它是两个程序之间传递的令牌。虽然能够从 JavaScript 中访问 cookie，不过它的确不是基于脚本的：它是 HTTP 服务器提供的机制。同样，客户端和服务器端都能够访问它。

无论名字叫什么，cookie 的意义就是一个带有过期时间、域名、路径的小型的键/值对，之所以需要提供这些消息，是为了确保正确的服务器能够读取到正确的 cookie。这些信息将作为 Web 请求的一部分发送，因此在服务器端和浏览器都能够访问这些数据。

10.2.1 cookie 的保存和读取

和其他浏览器元素类似，cookie 也是可以通过 document 对象访问的。要创建一个 cookie，你需要提供 cookie 名称（或称为键）、关联的值、过期时间以及与该 cookie 相关的路径。要访问一个 cookie，可以通过 document.cookie 值获得，并将其转成 cookie 格式。

如果想创建一个 cookie，只需为 document.cookie 赋一个类似于以下格式的字符串值：

```
document.cookie="cookieName=cookieValue; expires=date; path=path";
```

cookie 的名称和值可以根据你的需要设置，只要是简单值即可。我们通常在命名 cookie 时会以“$”（$cookieName）或“_”（_cookieName）为前缀，然后跟上其他字符。

我也尝试过为 cookie 赋不同的值，能够为 cookie 赋什么值，取决于浏览器能够将哪些

值转成字符串格式，可以包括数字、数组或其他对象。不过，不同的浏览器之间会有明显的差异。为了确保得到一致的结果，我建议你只使用基本类型（字符串、布尔型和数字型），它们都能够准确无误地转成字符串。

提示

如果你想在 cookie 中保存和读取复杂对象，那么可以参考 O'Reilly 上的一篇文章（http://www.oreillynet.com/onlamp/blog/2008/05/dojo_goodness_part_8_jsonified.html），首先将其转成 JSON 格式再保存到 cookie 上。我们在第 15 章中会对 JSON 做更进一步的介绍。

在 document 对象的 cookie 设置字符串中，过期日期是一个重要部分，它需要以专门的 GMT（UTC）格式表示。它将创建一个 data 对象，然后使用 toGMTString 方法进行转换，以确保该日期值能够正常使用。如果没有提供日期，那么它会默认为只在当前会话有用，当浏览器会话结束时就会删除它。

cookie 的路径特别重要。这里的域名、路径将会与页面请求的域名、路径做比较，如果不同步，那么该 cookie 将无法被访问或设置。这样就避免了其他网站访问存放在浏览器端的其他或所有的 cookie，因此这在过去是一个很明智的安全策略。

如果路径设置为“path=/”，那么将使 cookie 所允许的路径是该域名的顶级目录。如果你访问的页面位于 http://somedomain.com，那么意味着这个 cookie 能够被位于 http://somedomain.com 下所有子目录中的 Web 页面访问。如果指定了一个特定的子目录，如“path=/image”，那么 cookie 就只能被该子目录下的 Web 页面访问。反过来说，如果你的网站有多个子域名，如 sub1.somedomain.com、sub2.somedomain.com 等，如果想让一个 cookie 能够被它们中所有的页面访问，那么就应该为其指定一个最高级的域名：path=somedomain.com。

提示

选择 cookie 能够访问的位置是十分重要的。将 path 设置为应用程序的最顶级的做法应该严格限制。

以下代码片段展示了一个 JavaScript 函数实例，它在 cookie 中设置一对特定的键/值对，并将其有效时间设置为 2010 年，路径设置为最顶级的子目录：

```
function setCookie(key,value) {
   var cookieDate = new Date(2010,11,10,19,30,30);
   document.cookie=key + "=" + encodeURI(value) + "; expires=" +
cookieDate.toGMTString() + "; path=/";
}
```

使用 encodeURI 函数可以将 cookie 值中的特殊元素做转义处理。这样可以使你的 cookie 更加安全，因为它可以抹掉实际的 HTML 代码，包括脚本元素。另外，encodeURI 可以使 cookie 格式中使用到的特殊字符用于 cookie 值中，包括“=”和“;”。

另外一个处理 cookie 的函数是用来对其日期和路径进行调整的。请注意在这个字符串中，每个分号背后都跟着一个空白符。

提示

对于一个 cookie 而言，第四个参数是个标签位，用来说明 cookie 是否安全。获取一个安全的 cookie 只能通过 SSL 协议（HTTPS 而非 HTTP）。

获取 cookie 并不像设置这样容易，因为所有的 cookie 都被连接成了一个字符串，字符串中是用分号分隔的各个 cookie 对象。下面就是一个 cookie 字符串的实例：

```
var1=somevalue; var2=3.55; var3=true
```

获得键值的方法有好几种。第一种方法就是使用字符串的 split 方法来按分号分割字符串；另外还可以使用不同的子字符串搜索方法。下面这个示例函数混用了这些技术：

```
function readCookie(key) {
  var cookie = document.cookie;

  var first = cookie.indexOf(key+"=");

  // 存在 cookie
  if (first >= 0) {
    var str = cookie.substring(first,cookie.length);
    var last = str.indexOf(";");

    // 如果是最后一个 cookie
    if (last < 0) last = str.length;

    // 获取 cookie 的值
    str = str.substring(0,last).split("=");
    return decodeURI(str[1]);
  } else {
    return null;
  }
}
```

在前面的代码中，键值和等号是相连的，因此可以通过在字符串中搜索它来获得所需的结果。如果找到，那么就以它作为分割位置，获得包含当前字符串其余部分的子字符串。然后在这个新生成的字符串中搜索分号，如果找到，则解析到这个分号为止（还有多个 cookie）或访问整个字符串（只有一个 cookie）。最后，不断地对字符串按“=”分割，就能够获得所需的键和值。返回的值还需要通过 thedecodeURI 函数解转义，这样才能返回原始值。

如果要删除一个 cookie，只需删除其值或者将其有效时间设置为一个已经过去的日期，或者同时做以上修改，如下面这个 JavaScript 函数所示：

```
function eraseCookie (key) {
```

```
    var cookieDate = new Date(2000,11,10,19,30,30);
    document.cookie=key + "=; expires=" + cookieDate.toGMTString() + ";
path=/";
}
```

那么当下次访问这个 document 中的 cookie 时，该 cookie 就将不再存在。

在你使用各种 cookie 相关的功能之前，最好还是先检查一下当前浏览器是否实现并启用了 cookie。虽然人们通常很少禁用 cookie，但最好还是在程序代码中解决这个问题。想要检查 cookie 是否启用，可以使用另一个内建的浏览器对象 navigator，并使用其 cookieEnabled 属性：

```
if (navigator.cookieEnabled) ...
```

注意，当检查 cookieEnabled 属性时，并不是所有浏览器都将返回正确的值。例如，IE 6.x 就没有正确地设置该属性。在这种情况下，除了设置一个 cookie 看看是否能够找到它之外，没有什么好办法。

在示例 10.1 中，我们将这些东西都整合在一起，得到了一个完整的应用程序，它为 cookie 设置了一个值，当页面每次载入时都将访问它并对其做自增操作。当其值大于 5 时，就将这个 cookie 删除掉，然后在下一次迭代（页面载入）时重新创建这个 cookie。

示例 10.1　设置、读取、删除 cookie

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>innerHTML</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript">
//<![CDATA[

// if cookie enabled
window.onload=function() {

   if (navigator.cookieEnabled) {

      var sum = readCookie("sum");
      if (sum) {
         var iSum = parseInt(sum) + 1;
         alert("cookie count is " + iSum);
         if (iSum > 5) {
            eraseCookie("sum");
         } else {
            setCookie("sum",iSum);
         }
      } else {
         alert("no cookie, setting now");
```

```
        setCookie("sum", 0);
      }
    }

}
// 将 cookie 有效期设置为 2010 年
function setCookie(key,value) {

   var cookieDate = new Date(2010,11,10,19,30,30);
   document.cookie=key + "=" + encodeURI(value) + "; expires=" +
cookieDate.toGMTString() + "; path=/";
}
// 在每个 cookie 之前用分号隔开
function readCookie(key) {
  var cookie = document.cookie;
  var first = cookie.indexOf(key+"=");

  // 存在 cookie
  if (first >= 0) {
    var str = cookie.substring(first,cookie.length);
    var last = str.indexOf(";");

    // 如果是最后一个 cookie
    if (last < 0) last = str.length;

    // 获取 cookie 的值
    str = str.substring(0,last).split("=");
    return decodeURI(str[1]);
  } else {
    return null;
  }
}
// 将 cookie 的有效期设置成过去，以达到删除 cookie 的目的
function eraseCookie (key) {

   var cookieDate = new Date(2000,11,10,19,30,30);
   document.cookie=key + "= ; expires="+cookieDate.toGMTString()+"; path=/";
}
//]]>
</script>
</head>
<body>
<p>Paragraph text.</p>
</body>
</html>
```

cookie 是个很方便使用的家伙，但它也存在一些限制，对于一个域名而言只能存储 20 个 cookie，而且总容量不能大于 4KB。当然，这对于绝大多数情况而言已经可以满足了；实际上，你也应该很节俭地使用这个小型的客户端存储机制。不过，你仍然有可

能会遇到需要存储更大量数据的时候。

10.3 Flash 共享对象、Google Gears 和 HTML5 DOM 存储

如果想存储更大的 cookie 或更复杂的对象，在早先的应用程序中也曾经使用过一些技巧，包括 JavaScript 和 Java 小程序之间的 LiveConnect 接口，或者是 ActiveX 控件。另一种方法是使用隐藏表单域来持久化存储表单提交返回的数据。

在过去的几年中，受到 Ajax 和富 Internet 应用（RIA）的影响，出现了一批新方法。这些方法涉及范围很广：有私有的对象，如 Flash Shared Object（Flash 共享对象）；还有代表 HTML 5.0 未来的本地存储能力。对于这些方法而言，最大的困难是它们没有提供跨所有浏览器的实现。

在 2008 年 5 月，Paul Duncan 对客户端存储方法进行了鉴别和确认。根据他提供的列表，这些方法包括：

- Flash 9.0 持久化存储
- Gears（源于 Google）
- HTML 5.0 的 localstorage 和 whatwg_db
- IE 的 userdata 行为
- cookie

你可以以插件的形式安装 Gears；在 Google 的 Web 浏览器 Chrome 中它是内置的功能。如果 Gears 可用，那么可以使用 localServer 对象来设置本地存储：

```
<script type="text/javascript" src="gears_init.js"></script>
<script type="text/javascript">
  var localServer = google.gears.factory.create('beta.localserver');
  var store = localServer.createManagedStore('test-store');
  store.manifestUrl = 'site-manifest.txt';
  store.checkForUpdate();
</script>
```

另外，你也可以使用功能完备的关系型数据库，只要你的系统提供了 SQLite 数据库实现的支持：

```
<script type="text/javascript" src="gears_init.js"></script>
<script type="text/javascript">
  var db = google.gears.factory.create('beta.database');
  db.open('database-test');
  db.execute('create table if not exists Test' +
```

```
            ' (Phrase text, Timestamp int)');
  db.execute('insert into Test values (?, ?)', ['Monkey!', new
Date().getTime()]);
  var rs = db.execute('select * from Test order by Timestamp desc');

  while (rs.isValidRow()) {
    alert(rs.field(0) + '@' + rs.field(1));
    rs.next();
  }
  rs.close();
</script>
```

Gears 的功能的确相当强大，但这强大的功能依赖于你的用户使用的是 Chrome 或者安装了 Gears 插件。这在一个开放的 Web 环境中是很受限的。

提示

在 Gears API（http://code.google.com/api/gear/design.html）中可以看到一些与 Gears 相关的示例。你可以从 http://gears.google.com/上下载到 Gears 插件。

Flash 共享对象（Flash Shared Object，SO）的运作模式与 HTTP 的 cookie 类似。它们也是基于域来存储和访问的，某个域名的页面不能够访问其他域名创建的 SO。从 Flash 7.0 之后，Adobe 公司就将沙箱保护作为了 SO 设计的一部分。

与 HTTP 的 cookie 不一样，它没有 4KB 的限制，SO 能够存储无限大的东西，当存储的内容不超过 100KB 时不会做任何提示。这也就意味着如果同一个域名的 Web 页面或 Web 应用程序尝试设置一个超过 100KB 的 SO，那么将会弹出一个消息对话框，让用户决定是否提供相应的空间。然后客户端为 SO 提供显式的权限。

提示

在 http://livedocs.adobe.com/flash/9.0/ActionScriptLangRefV3/flash/net/SharedObject.html 上有与 Flash 9.0 的 SO 相关的更多资料。

Firefox、WebKit 和 IE 8.0 都在不同程度上实现了 HTML 5.0 中定义的 localStorage 和 sessionStorage 对象，也就是大家所熟悉的 DOM 存储，不过 Opera 将在未来版本中才实现这一更先进的客户端存储方法。在未来，DOM 存储是除了 cookie 之外唯一能够内建在浏览器中的跨浏览器解决方案。

以下是一个使用 localStorage 的示例，它是从微软的文档中节选出来的：

```
<p>
  You have viewed this page
  <span id="count">an untold number of</span>
  time(s).
</p>
<script>
  var storage = localStorage[location.hostname];
```

```
    if (!storage.pageLoadCount) storage.pageLoadCount = 0;
    storage.pageLoadCount = parseInt(storage.pageLoadCount, 10) + 1;
    document.getElementById('count').innerHTML = storage.pageLoadCount;
</script>
```

以下是从 Mozilla 中节选的一个示例，它实现了 DOM 存储中的 sessionStorage 对象：

```
// 获取我们要追踪的文本框
var field = document.getElementById("field");

// 看它是否是自动保存的值
// （这仅当页面意外刷新时才会发生）
if ( sessionStorage.autosave ) {
    // Restore the contents of the text field
    field.value = sessionStorage.autosave;
}

// 每一秒钟检查一次该文本框的值
setInterval(function(){
    // 将结果保存到 sessionStorage 对象中
    sessionStorage.autosave = field.value;
}, 1000);
```

当然，在这些示例中保存的数据也可以通过 cookie 来保存，但 HTML 5.0 存储和 cookie 相比有一些明显的优势，首先是可以存储更多的数据，其次是可以处理更复杂的数据和数据访问，包括在前一个示例中提到的基于会话的恢复。

HTML 5.0 存储的实际限制是不同浏览器中的不同实现，由于该规范还在开发中，因此还没有一个完整的 DOM 存储实现。

提示

在 http://www.w3.org/html/wg/html5/中，你可以找到 HTML 5.0 规范，其中包含对 DOM 存储的描述。在 http://msdn.microsoft.com/en-us/library/cc197062(VS.85).aspx 中可以找到关于 IE 8.0 的文档。在 http://developer.mozilla.org/en/DOM/Storage 上可以找到 Mozilla 的文档。

这些方法可以分成几种，有的是基于插件的（Flash 和 Gears），有的是基于浏览器使用的（IE 的 userdata），还有一些是基于仍然在不断改变的规范的（HTML 5.0 DOM 存储规范）。此外，还有一个开源的、名为 Dojo Storage 的程序库，它的目标就是通过程序库来提供跨浏览器的解决方案，你可以在自己的应用程序中引用它。直到本书写作时，该程序库还没能够支持所有浏览器；不过这一情况可能在你读到这本书时已经发生改变了。最后的结论是跨浏览器兼容的是标准的 cookie，或者是 Paul Duncan 发布的名为 PersistJS 的跨浏览器兼容的程序库。

提示

在 http://pablotron.org/?cid=1557 中，你可以找到 Paul Duncan 写的关于 PersistJS 的文章和宣言。Dojo Offline Toolkit 可以在 http://dojotoolkit.org/offline 上下载到。

不过，有意思的是在这么多客户端存储技术中，只有 cookie 是所有浏览器默认支持的。除非你需要其他方法所提供的更大的存储能力，那么最好还是仍然使用 cookie，至少是在接下来的几年中。

10.4 知识测验

1. 说出几种在客户端计算机上存储信息的方法。
2. 脚本的 cookie 由几个部分组成?
3. 如何创建一个在浏览器关闭时自动删除的 cookie?
4. 对于用户输入的哪类数据需要进行清理?
5. 想想一个你已经创建或未来将创建的网站。请构思脚本 cookie 在你的网站中可能实现的 5 个用途。那么在这些应用场景中，你需要的空间是否比 cookie 所能提供的更大?

10.5 测验答案

1. 将日期或其他信息存储到客户端计算机上有以下几种方法：

- 使用 cookie；
- 使用第三方插件，如 Flash 或 Google Gears；
- 要求用户在一个链接的资源上单击鼠标右键，然后将其保存到本地计算机上；
- 插入一个作为链接的可下载文件；
- 创建一个用来保存数据的浏览器扩展；
- 使用 HTML 5.0 中的 localStorage。

2. 脚本的 cookie 是由 cookie 名称、值、过期时间、相关路径组成的。
3. 不提供任何有效日期值，并且/或者将值设置为空。
4. 任何可以在浏览器上调用的数据，或者能够用来监听客户端 cookie 的、能够运行服务器端进程的用户输入都将进行清理。特别是“javascript:”或脚本标签都会被从输入中清理掉。

不过，对输入进行清理并不是完全像你想的那样直接删除。对于内容管理工具而言，用户在一个特定的帖子或页面中输入脚本是可行的。但在一个多用户环境中，某个用户就可能通过脚本获得使用该系统的其他用户信息。

5. 这是一个没有标准答案的问题。以下就是使用 cookie 的常见场景：

- 维护某个人的名字、URL 和电子邮件；
- 为数据项提供实时反馈；
- 启用拼写检查；
- 保存登录信息；
- 维护购物车。

这些数据所需的存储空间都不超过 4KB。

第 11 章

DOM 或以树形展示的 Web 页面

万维网联盟的文档对象模型（W3C DOM）的第一个版本是 DOM Level 1，它于 1998 年作为建议规范（recommendation）发布。这次发布定义了 DOM 的基础设施，包括模式（schema）和 API，这些都是 DOM 后续版本可以使用的基础功能。它还包括有助于兼容 DOM 的用户代理（如浏览器）构建针对每个版本的建议规范的核心构件；对于所有其他分别发布的规范，则提供一些相关的、可选的模块。这种模块化方法有助于浏览器尽早采纳，同时也能和关键元素保持一致。

DOM Level 2 在 2000 年发布，它在早期的 Level 1 的基础上做了扩展，不过仍然保持着和早期版本的一致性。在第 7 章中就已经介绍过 Level 2 的事件处理机制，相信大家对这个方面已经有了些印象。DOM Level 2 增强了对 CSS 的支持，改进了对 document 元素的访问方法，以及对 XML 建议规范中的命名空间支持。

DOM Level 3 已经在 2004 年发布了。除了在先前版本的基础上进行扩展和改进之外，该版本还增加了支持 Web Service 的模块，同时增强了对 XML 的支持，包括对 XPath 的支持。DOM Level 3 是 W3C 最后发布的规范版本，至少是计划中最后的版本。

本章不打算介绍 DOM API 中所有的对象。我们只关注于最有代表性的对象，说明它们之间是如何交互的，以及它们是如何影响浏览器页面的。

11.1 两个接口的传说

当 W3C 发布 DOM 的第一个版本时，该组织还发布了两个不同的 API：Core（核心）API 和 HTML API。

DOM 的核心是语言无关（也是模型无关）的 API，你可以用任何语言实现，不仅仅是 JavaScript，也可以针对任何基于 XML 的模型，不仅仅是 XHTML。同样，从字面上理解，DOM 的核心就是提供修改、删除或创建 Web 页面内容所需的各种功能。

不过在 DOM 规范发布之前，浏览器已经以不同形式实现了 BOM（浏览器对象模型），有些是私有的，有些不是。为了与这些之前的工作保持向后兼容，W3C 还发布了一个 DOM API 的自定义子集：DOM HTML API。

DOM HTML API 是面向对象的、层次性的 Web 页面视图，它提供了一些与 HTML 元素相映射的对象：针对 document 的 HTMLDocumentElement、针对 body 的 HTMLBodyElement 等。使用这个 API 与第 9 章中使用的 BOM 类似。在 BOM API 和 DOM HTML API 之间也存在一些主要的差异，后者的目标是将之前的努力作为一个子集整合到一个API中，使其能够在所有浏览器上正常运行。W3C 还对 DOM HTML API 进行了扩展，使它能够与底层的 Core API 保持兼容。

正如前面所述，Core API 是一个通用的 API，能够应用于各种形式的 XML 标准，包括 Web 页面上的 XHTML，也包括 Ajax 调用返回的 XML（这方面的内容我们将在第 15 章中介绍）。该 API 由如 Node、NodeList、Attr、Element 和非常重要的 Document 等对象组成。Core API 还提供了浏览器必须支持的基本数据类型和预期行为，虽然这些支持在使用 JavaScript 时没有明显的感知。

HTML API 涵盖了 W3C 发布的前两个规范，但不包含第三次发布的规范。这是因为在 W3C 发布的 DOM Level 3 中，所有增加和修改的内容都是针对 Core API 的，HTML API 没有受到直接的影响。

由于需要向后兼容，因此在 DOM 中也有一些冗余。使用 DOM Core API 和 DOM HTML API 都能够实现相同的功能，如创建一个 HTML 表格行。选择哪个 API 基本上取决于你自己的喜好，虽然尽可能保持一致是十分重要的。DOM HTML API 具有很好的自描述性，因为它有专门的对象名称。不过，如果你想创建自己的可复用对象程序库，最好使用更通用的 DOM Core API。对于绝大多数 JavaScript 应用程序而言，这两个 API 中的所有功能都可能会用到，在本章接下来的内容中就将更详细地介绍它们。

提示

如果想对不同的 DOM 规范建立概要性了解，可以阅读 OASIS Cover Pages 上的文章（http://xml.coverpges.org/dom.html）。另外，在 David Flanagan 所著的 *JavaScript: The Definitive Guide*（O'Reilly 出版社出版）一书中，介绍了更多的 DOM 对象。

11.2 DOM HTML API

本小节将简介 DOM（HTML）Level 1 和 DOM（HTML）Level 2。之所以称为“简介”，是因为 DOM（HTML、Core 或者全部）实在太大了，在一章中是肯定讲不完的。不过在这个小节中，我希望能够足够详细地介绍 DOM（HTML）背后的基本概念，这样你就可以基于我介绍的这些对象，推知规范中文档化的其他所有对象。说到规范，如果你想下载

DOM（HTML）Level 1 规范，可以访问 http://www.w3c.org/TR/REC-DOM-Level-1/，想了解其相关的 ECMAScript 绑定，则可以访问 http://www.w3c.org/TR/REC-DOM-Level-1/ecma-script-language- binding.html。如果想下载 DOM（HTML）Level 2 规范，可以访问 http://www.w3c.org/TR/REC-DOM-Level-2/，想了解其相关的 ECMAScript 绑定，则可以访问 http://www.w3c.org/TR/ REC-DOM-Level-2/ecma-script -binding.html。

提示

ECMAScript 绑定是 JavaScript 对象和其等价的 DOM 规范对象之间的映射。如果你使用前面提到的规范，那么 DOM 规范将解释每个对象，而 ECMAScript 绑定则提供该对象在 ECMAScript（JavaScript）中如何实现的具体信息。

11.2.1 DOM HTML 对象及其属性

DOM HTML API 是一组接口对象集，而不是实际的对象类。通过这些接口可以访问原有的、新创建的页面对象，每个都与特定类型的页面对象相关联。

提示

在此我谈到了一个新术语——接口。从我们的意图来看，接口就是一个表示特定页面元素的对象。它与类不同，它没有构造函数；对象是通过其他函数创建的，而非直接创建的。

关于模型中必然有些冗余，之前我们说是为了向后兼容。另外一个造成冗余的原因是两个模型（Core 和 HTML）之间有交叠，而且 HTML API 对象将组合所有模型中的方法和属性。HTML API 不仅从基本的 HTML 元素中继承了属性和方法，还会从核心的 Node 对象（本章稍后将介绍）中继承。绝大多数 DOM（HTML）对象也是从 HTMLElement 中继承的，它拥有以下属性，都是基于其允许的元素属性的：

- id　元素标识符；
- title　元素内容的描述性标题；
- lang　元素及其内容的语言；
- dir　文字方向（从左到右、从右到左）；
- className　等价于 class 属性。

每个 HTML 对象的名字都是从 HTML 元素的名称中获取的，而元素的标签名并不是必需的。HTMLFormElement 是 HTML 中 form 元素的接口对象，而 HTMLParagraphElement 则是 p（段落）元素的接口对象。通过该对象可以访问该元素中的所有有效元素，如 HTMLDivElement 中的 align 属性、HTMLImageElement 中的 src 属性等。

绝大多数 DOM（HTML）属性都是可读写的，这也就意味着你可以通过 JavaScript 来

访问和修改它。为了展示其方法，在示例 11.1 中，我们将通过 document.image 集合来访问一个图像元素。在 HTML DOM Level 1 中，图像元素拥有 HTMLElement 中列出的所有属性，还拥有针对 HTMLImageElement 对象的属性，不过有一些已经被 HTML 4.0 弃用了。在 HTML 4.0 及后续规范中，height、width 和 src 都仍然是可用的。

本示例将把所有有效的图像属性连接成一个字符串，然后通过一个 alert 对话框显示给用户。在此之后，我们对图像元素中几个属性做了一些修改。

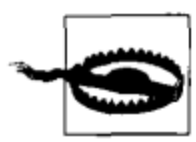
示例 11.1　读取并修改图像元素的属性

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>Modifying HTML Elements</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript">
//<![CDATA[

window.onload=function() {

   var img = document.images[0];

   // 获取现有的图像属性
   var imgAttr = img.id + " " + img.alt + " " + img.className;
   alert(imgAttr);
   // 修改属性
   img.src="osprey.jpg";
   img.width="800";
   img.height="498";
   img.alt="Alternative";
   img.align="left";
   img.title="Osprey";
}
//]]>
</script>
</head>
<body>
<p><img src="ospreythumb.jpg" alt="test image" class="testimage"
id="original"
/></p>
</body>
</html>
```

注意，我们在此是通过 DOM 中的 className 属性来访问 img 中的 class 属性的。其他属性的名称在 XHTML 和 DOM 中都是一样的。

这种通过 JavaScript 修改元素的技术，也是“解决”XHTML 严格要求的一种方法。在

本示例中，我们通过 JavaScript 修改了图像的宽度和高度，但是当页面的 DOCTYPE 被声明为 XHTML strict 时，这些属性是不允许出现在 img 元素中的。通过 JavaScript 可以为页面元素添加弃用的或者自定义的属性，并且能确保页面文档仍然有效。虽然对于该技术而言，这只是一个价值不大的示例，因为使用 style 属性也能够添加 width 和 height 值，但在开发更为复杂的 Ajax 应用时，该技巧是向页面元素添加自定义属性的好方法。



警告

严格的 DOCTYPE 是为了确保页面能够在所有浏览器中呈现相同的效果。因此在实践中要注意，使用 JavaScript 动态修改页面元素时，有可能会“改写”DOCTYPE。

还有一些 DOM HTML 接口对象提供了创建、删除或以其他方式修改相关页面元素的方法。特别是 table 元素，它就提供了一组特定的方法和特定的对象。不过，这个过程看起来有些复杂，其中的原因就像之前所说的，API 对象没有构造函数，不能独立于其他对象单独创建。要创建这些接口对象，你需要使用一个工厂方法。顾名思义，工厂方法扮演的就是类似于工厂的角色，负责生产出所需的对象。不过它和汽车、iPod 工厂不一样，这些工厂生产的是如表格单元格、表格行之类的对象。

在示例 11.2 中，HTML 文档中有一个图像和一个只有表头的 HTML 表格。当载入该文档时，将调用一个函数，它将使用 document.getElementById 方法来访问这个表格和图像，这是 BOM 和 DOM（HTML）Level 1 都支持的另一个对象/方法。该应用程序的目的是获取图像元素的 src 属性，然后将其显示在表格上。

为了在表格上添加内容，该应用程序使用了 table 元素的工厂方法 insertRow，传入的参数为-1，表示在表格最后添加一行。如果要在原有行中的某个位置上插入一行，则应该传入表示该行插入位置的值。

insertRow 方法将返回一个实现 HTMLTableRowElementinterface 接口的对象。在 HTMLTableRowElement 接口对象中还有另一个名为 insertCell 的工厂方法，它将创建一个 HTMLTableCellElement 的对象，用来表示特定的单元格。通过 insertCell 方法可以创建两种单元格，其中一种是表格中的表格数据元素（td）。

要向新创建的表格单元格添加文本内容，需要使用 document 对象中的一个名为 createTextNode 的工厂方法，它将根据传入的字符串创建一个 text 对象。然后使用 HTMLElement 中的 appendChild 方法将这个 text 对象添加到表格单元格上。

示例 11.2 使用 DOM HTML 接口将图像属性输出到表格中

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
```

```
<title>Modifying HTML Elements</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript">
//<![CDATA[

window.onload=function() {

   // 获取表格和图像
   var tbl = document.getElementById('table1');
   tbl.border="5px";
   tbl.cellPadding="5px";

   var img = document.getElementById("img1");
   img.vspace="10";

   // 为每个属性添加一个表格行
   var row1 = tbl.insertRow(-1);

   // 创建两个表格单元格
   var cell1 = row1.insertCell(0);
   var cell2 = row1.insertCell(1);

   // 创建文本值
   var txtAttr1 = document.createTextNode("src");
   var txtAttr1Val = document.createTextNode(img.src);

   // 将文本值添加到单元格上
   cell1.appendChild(txtAttr1);
   cell2.appendChild(txtAttr1Val);

}
//]]>
</script>
</head>
<body>
<p><img src="ospreythumb.jpg" alt="test image" class="testimage" id="img1"
/></p>
<table id="table1">
<tr><th>Attribute</th><th>Value</th></tr>
</table>
</body>
</html>
```

要删除一个元素很简单，在本例中，我们分别使用了 removeCell 和 removeRow 方法来删除表格单元格和表格行。正如你所看到的那样，使用 DOM HTML API 为 Web 页面添加元素并不复杂，但很乏味。示例 11.2 也展示了 DOM Core 和 HTML Level 1 对象之间的交叠，除了可以使用特定于 HTML 的 insertRow 方法之外，还可以使用更通用于不同 XML 文档的 document.createTextNode 和 object.appendChild 方法。

图 11.1 展示的 Web 页面中就有一个动态创建的 HTML 表格。

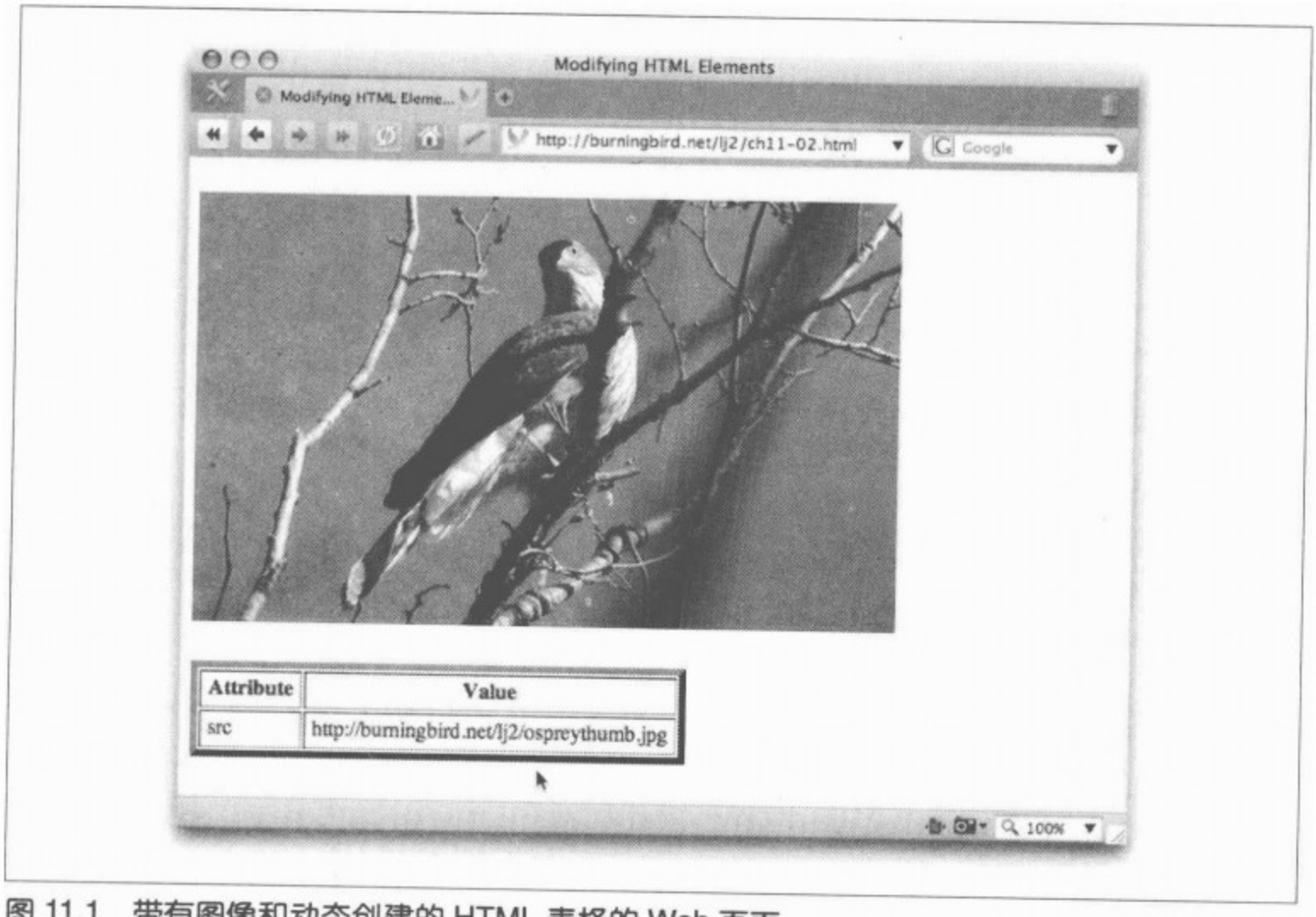


图 11.1　带有图像和动态创建的 HTML 表格的 Web 页面

11.2.2　DOM（HTML）集合

有些 DOM HTML 接口并不直接表示特定的 HTML 元素，而是表示对象的集合。

对象的集合可以通过其父元素访问，如 document 或 form 元素，它是用 HTMLCollection 接口表示的，是 DOM（HTML）Level 1 规范所引入的。HTMLCollection 接口只有一个名为 length 的属性，还有两个方法：一个是 item，参数为数字型的索引值；另一个是 namedItem，其参数是字符串。这两个方法都能够返回集合中的特定对象。

在示例 11.2 中，document 对象的图像集合是通过一个数组访问的。这是使用 BOM 访问图像的典型做法。使用 DOM，将改为使用数组索引值访问图像，你可以使用 item 方法，并传入相同的索引值：

```
var img = document.images.item(0);
```

这条代码的运行结果和示例 11.2 中的代码一样，将一个图像对象（HTMLImageElement）赋给一个局部变量。两种不同的代码之间的主要区别是一个使用的是方法调用，另一个是数组索引。使用 length 属性也可以知道当前页面上的图像数：

```
var numImages = document.images.length;
```

如果你想通过指定名字或标识符访问一个元素，那么可以使用 namedItem 方法：

```
var img = document.images.namedItem("original");
```

不过，要获得指定标识符的元素还有一种更通用也更简单的方法，那就是使用 document.getElementById 方法，主要的原因是 id 属性应用很广泛，反而是 name 属性可能会受到限制，这和页面声明的 DOCTYPE 相关。

DOM（HTML）Level 2 规范对 HTMLCollection 做了些扩展，在 ECMAScript 语言绑定中引入了名为 HTMLOptionsCollection 的特定类型集合对象。HTMLOptionsCollection 表示的是 select 元素的选项列表，而它本身则是用 HTMLSelectElement 表示的。通过后者访问 option 属性时将返回 HTMLOptionsCollection 对象，然后就可以使用 item 和 namedItem 方法来访问每一项。别的集合都没有自己专用的语言相关的对象，我认为这种情况是由浏览器实现造成的。由于该对象的方法和属性与 HTMLCollection 相同，你可以将它们视为同一个对象。

集合对于处理这种成组项目而言是最简单的方法。例如，如果你想寻找页面中所有的超链接，然后将它们输出到该页面最后的列表中，那么可以创建一个如示例 11.3 所示的应用程序。

在这个应用程序中，每个链接将作为 document.links 集合中独立的元素（HTMLLinkElement）来访问。对于 HTMLLinkElement 元素而言，可以应用的属性之一是 href，它用来指定该链接的 URI（统一资源标识符，URL 是 URI 的一种类型）。该值将输出到一个 p（段落）元素中，然后添加到页面最后的 div 元素中。

示例 11.3　使用链接集合输出页面的 URL 列表

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>Modifying HTML Elements</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript">
//<![CDATA[

window.onload=function() {

    // 获得链接和 div 元素
    var theLinks = document.links;
    var theHrefs = document.getElementById("hrefs");

    // 遍历每个链接元素
    for (var i = 0; i < theLinks.length; i++) {

      // 获取 href 属性
      var href = theLinks.item(i).href;
```

```
        // 基于href创建文本节点，然后将其添加到新的段落元素上
        var p = document.createElement("p");
        var txt = document.createTextNode(href);
        p.appendChild(txt);

        // 将这个新的段落元素添加到div元素上
        theHrefs.appendChild(p);
    }
}
//]]>
</script>
</head>
<body>
<p>A good reference for the DOM is the <a
href="http://xml.coverpages.org/dom.html">OASIS Technology Report
on the DOM</a>. In addition, the primary DOM location at the W3C is the <a
href="http://www.w3.org/DOM/">W3C Document Object Model page</a>, from which you
can then access each DOM level. A handy interactive guide to test your user
agent (browser) and its compliance to the DOM can be found at <a href="http://www.
w3.org/2003/02/06-dom-support.html">the W3C</a>, though I'm not sure how
up-to-date it is
since it's not showing the DOM Level 3 support browsers have implemented.</p>
<div id="hrefs">
<h3>The links</h3>
</div>
</body>
</html>
```

该页面在 Opera 中的执行效果如图 11.2 所示。

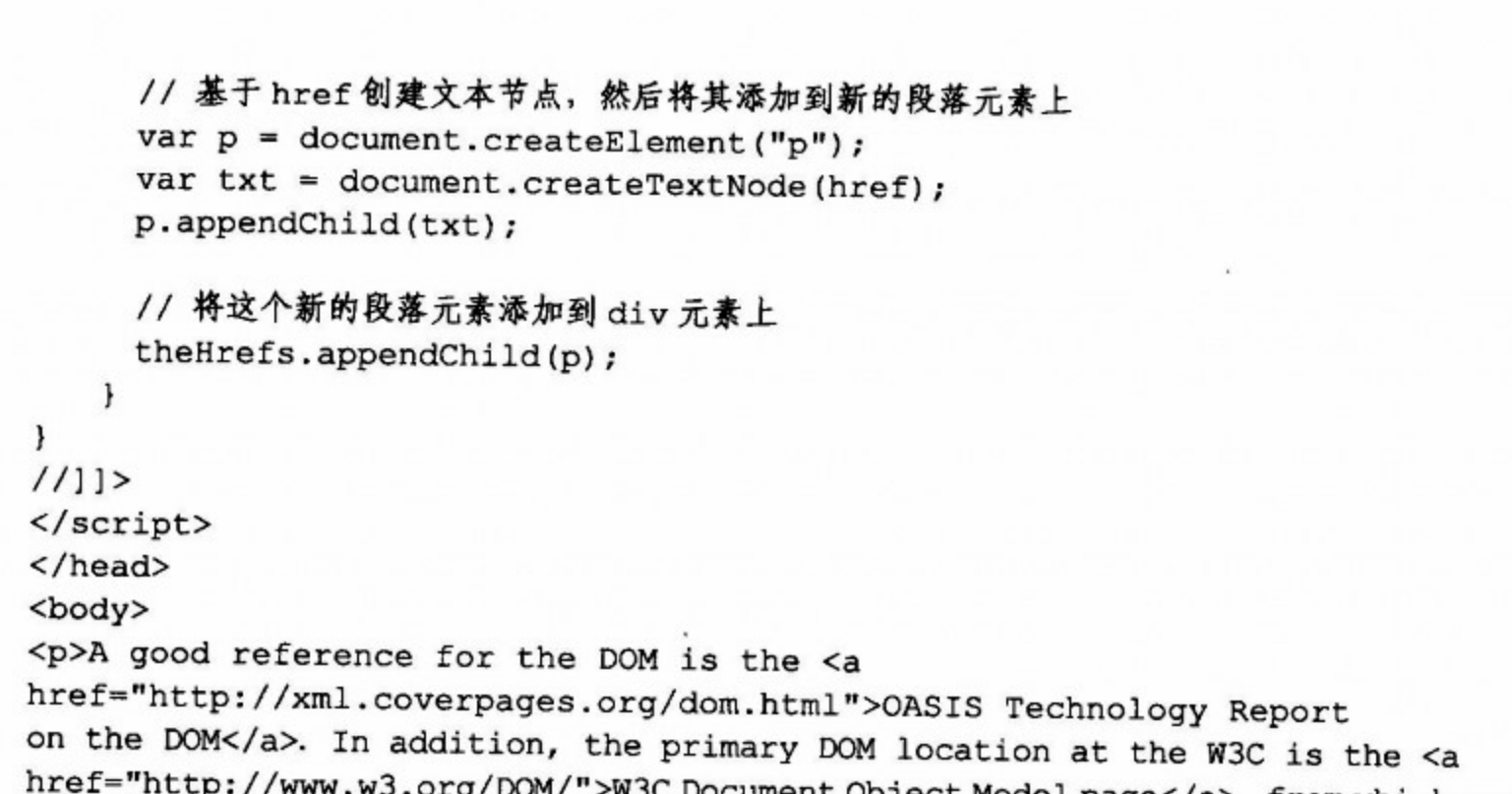


图 11.2　示例 11.3 输出的页面，演示了链接集合的应用

示例 11.3 再次展现了混用 DOM（HTML）API 和 DOM（Core）API 的场景。我们采用 HTML API 访问链接集合中的每个链接元素，并获取其 href 属性；然后采用 Core API 创建一个新的 p（段落）元素和文本节点，并将它们添加到页面上。

11.3 理解 DOM：Core API

创建 DOM（HTML）API 的目的主要就是为了引入跨浏览器存在的 BOM 实现。不过，在过去的几年中，产业界开始不再使用 HTML（以及各种私有的扩展），逐渐倾向于使用基于 XML 的 XHTML。对于 XHTML 而言，DOM HTML API 仍然有效，但名为 DOM Core API 的另一组接口逐渐在当前的 JavaScript 开发人员间流行起来。

提示

事物的发展总是来回反复的。现在正在进行的新变革是创建 HTML 5.0，它是 HTML 的下一代。HTML 5.0 中的一部分已经在绝大多数浏览器中得以实现。

在 W3C 关于 DOM 的规范中，将 document 元素描述为一个节点（node）集合，它们之间是以有层次的树型结构相连接的。如果你使用的浏览器是 Firefox 3.x，并且安装了 DOM Inspector 插件，那么可以打开该工具看看页面中的对象，这时你会看到整个页面的组织十分像一棵树。一个 Web 页面中有 head 和 body 标签，body 中有 h1 也有 div 元素，而在 div 元素中又包含段落元素，在段落元素中则包含文本等。该页面的结构类似于以下所示的结构，也类似于图 11.3 所示的结构。

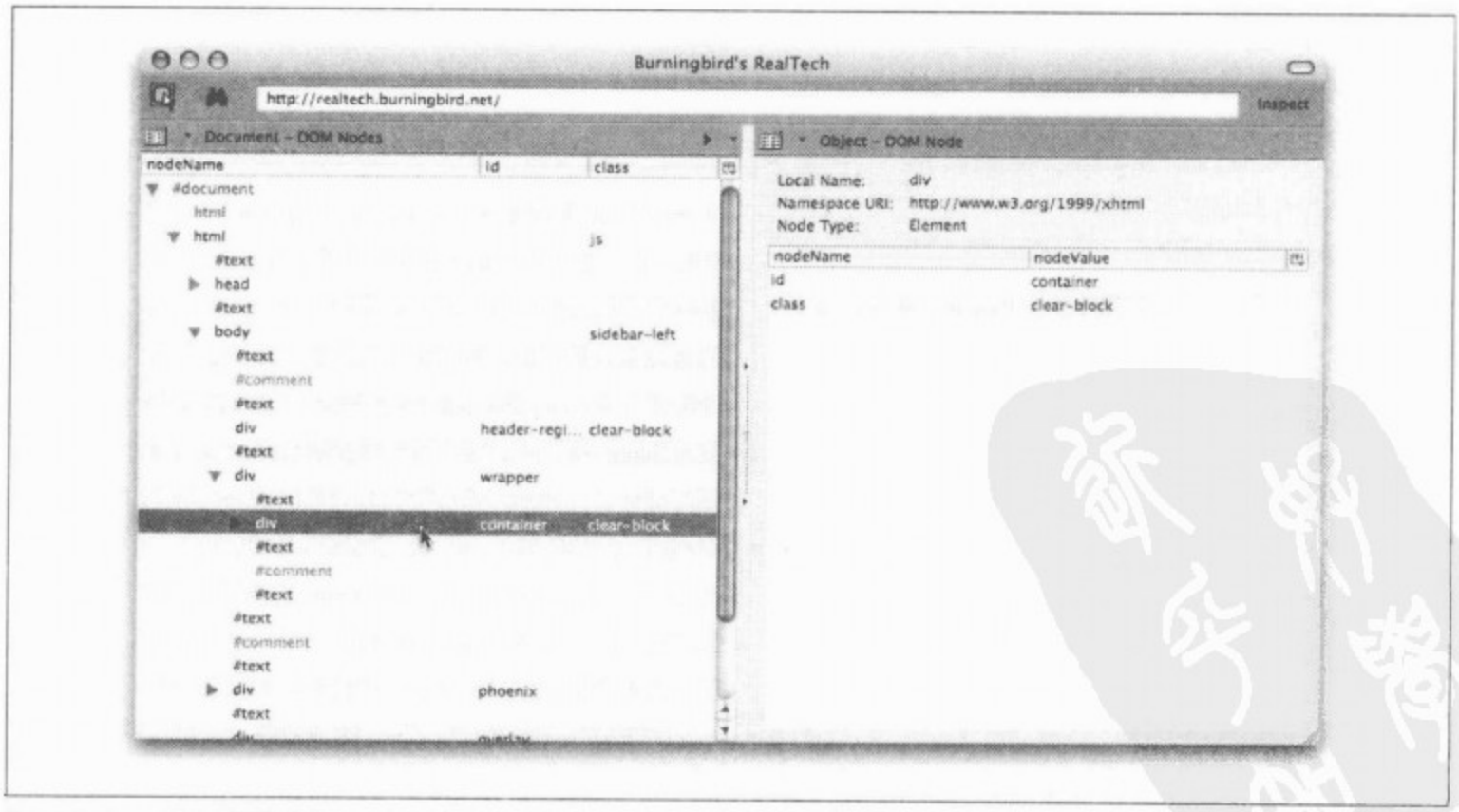


图 11.3 在我的网站 realtech.burningbird.net 上执行 DOM Inspector 时的屏幕截图

```
#document
html
   head
   body
      h1
      div
         p
            #text
         p
           #text
```

DOM（Core）提供了一个规范，使你可以通过两种不同的技术访问内容树上的节点：第一种是查询特定类型的所有标签；第二种是遍历不同的层级，也就是遍历整个树以探索每个层级上的每个节点。你不仅可以从树上读取节点，还可以删除或创建新的节点。

11.3.1 DOM 树

为了更好地理解文档树，我们来看一个如示例 11.4 所示的 Web 页面，它包含 head 和 body 两个小节、一个页面标题和一个 div 元素，而 div 元素中还有一个标题行 h1 和两个段落。其中一个段落是斜体字；另外一个则包含图像。换句话说，这是一个很典型的 Web 页面。

示例 11.4　一个典型的 Web 页面

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>Page as Tree</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
</head>
<body>
<div id="div1">
<h1>Header</h1>
<!-- paragraph one -->
<p>To better understand the document tree, consider a web page that has a head
and body section, a page title, and the body contains a div element that itself
contains a header and two paragraphs. One of the paragraphs contains <i>
italicized text</i>; the other has an image--not an uncommon web page.</p>
<!-- paragraph two -->
<p>Second paragraph with image. <img src="osprey.jpg" alt="bird" /></p>
</div>
</body>
</html>
```

在 DOM 树结构中，一个属于另一个元素的元素被称为子节点。在示例 11.4 中，文本节点就是段落元素或标题行元素的子节点；段落元素则是 div 元素的子节点，而 div 元素本身也是文档中 body 元素的子节点。作为子节点的容器元素通常称为父节点；在本

例中，div 元素是段落元素的父节点。它包含的同一级元素（也就是父节点相同）称为兄弟节点。这里的两个段落元素和标题行元素（h1）之间就称为兄弟节点。图 11.4 以图形展示了该页面的树形结构。

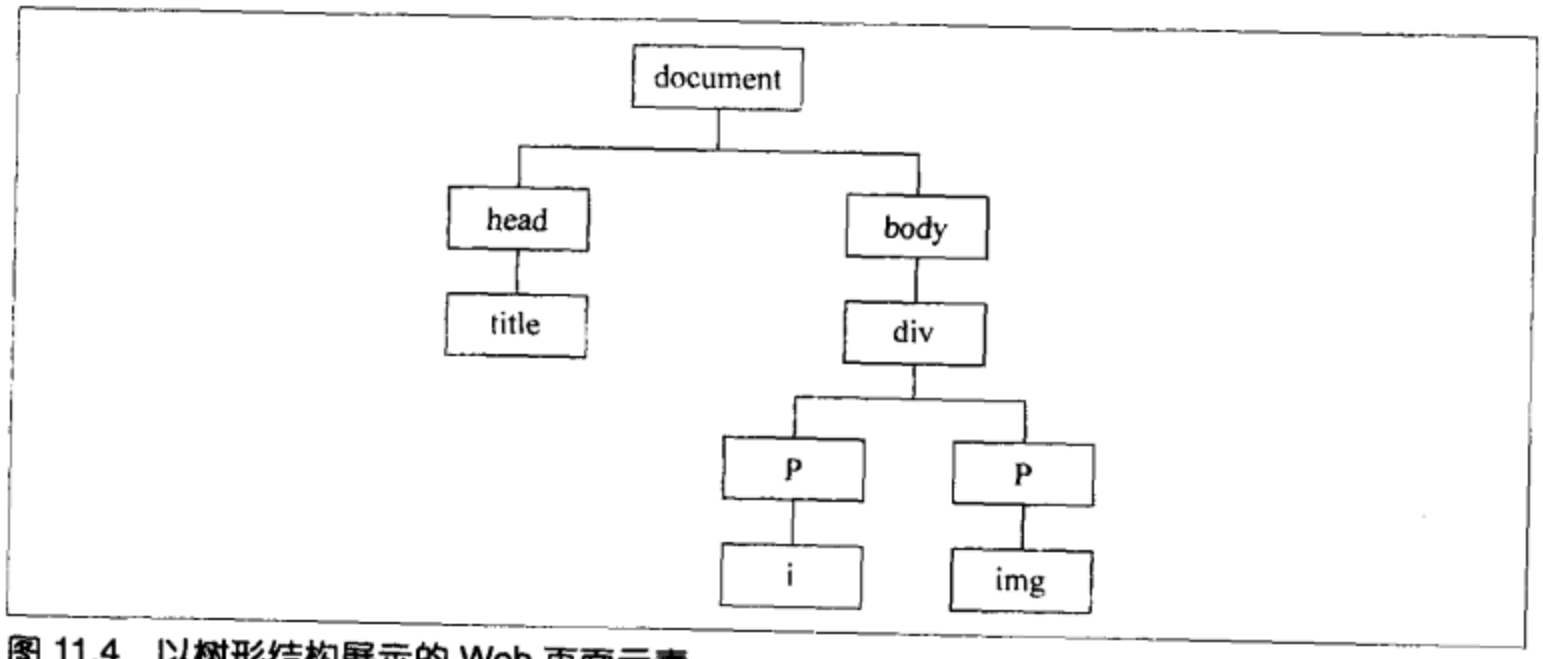


图 11.4　以树形结构展示的 Web 页面元素

如节点间关系之类的信息，可以通过每个节点共享的属性和方法访问，这些将在下一节中说明。

11.3.2　节点属性和方法

对于文档树上的每个节点，不管是什么类型，都有一个地方和其他节点相同：每个节点都拥有 DOM（Core）的 Node 对象中定义的基本属性和基本方法集。Node 对象的属性记录着与 DOM 内容树相关的关系，包括那些兄弟节点元素、子节点元素和父节点元素。它有一些属性用来记录节点的其他相关信息，包括类型、名字以及值（前提是可使用）。Node 对象的属性主要包括：

- nodeName　对象名称，如 head 元素的名称就是 HEAD；
- nodeValue　如果不是一个元素，则返回对象值；
- nodeType　用数字表示的节点类型；
- parentNode　当前节点的父节点；
- childNodes　由其子节点组成的 NodeList，前提是存在子节点；
- firstChild　由子节点组成的 NodeList 中的第一个节点；
- lastChild　由子节点组成的 NodeList 中的最后一个节点；
- previousSibling　如果当前节点是位于 NodeList 中的子节点，那么它表示的就是该列表中的前一个节点；

- nextSibling　如果当前节点是位于 NodeList 中的子节点，那么它表示的就是该列表中的下一个节点；
- attributes　一个 NamedNodeMap，它是以键/值对形式表示的，是该元素的属性列表；
- ownerDocument　拥有的 document 对象，当你拥有多个 document 对象时它比较有用；
- namespaceURI　命名空间的 URI，如果有的话，它是针对节点的；
- Prefix　命名空间的前缀，如果有的话，它是针对节点的；
- localName　如果指定了 namespaceURI 的话，它表示节点的本地名。

从 Node 的属性中，你可以看到 XML 对其的影响，特别是有些与命名空间相关的属性。不过，当你访问一个作为浏览器内节点的 HTML 元素时，命名空间相关属性的值通常都是 null。同样，这里有些属性是对于元素有效的，例如封装页面元素的 html 和 div 元素；有些属性则只对不是元素的 Node 对象有效，例如与段落元素相关的 text 对象。

为了对元素/非元素之间的区别有更好的理解，示例 11.5 中的应用程序将访问 Web 页面的 document.body 元素中的所有 Node 对象，然后弹出一个对话框，列出这个节点的属性。nodeType 属性值是用数字表示的节点类型，nodeName 是当前处理的实际对象名称。如果该节点不是一个元素，那么将输出 nodeValue 属性值；否则将输出 null。

另外，如果 Node 对象是一个元素，那么就将拥有样式属性。该应用程序将设置当前处理对象的背景颜色（使用随机色彩生成器），因此在页面处理过程中将会获得可视反馈（与此同时将在提示信息中输出背景颜色信息，这是第二种反馈方法）。

示例 11.5　访问 Node 的属性

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>Page as Tree</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript">
//<![CDATA[

// 随机色彩生成器
function randomColor() {
       var r=Math.floor(Math.random() * 255).toString(16);
       r = (r.length < 2) ? "0" + r : r;
       var g=Math.floor(Math.random() * 255).toString(16);
       g = (g.length < 2) ? "0" + g : g;
       var b=Math.floor(Math.random() * 255).toString(16);
       b = (b.length < 2) ? "0" + b : b;

      return "#"+r+g+b;
```

```
}

// 输出部分节点属性
function outputNodeProps(nd) {

   var strNode = "Node Type: " + nd.nodeType;
   strNode += "\nNode Name: " + nd.nodeName;
   strNode += "\nNode Value: " + nd.nodeValue;

   // 如果设置了样式(元素的属性)
   if (nd.style) {
      var clr = randomColor();
      nd.style.backgroundColor=clr;
      strNode += "\nbackgroundColor: " + clr;
   }

   // 输出该节点的属性
   alert(strNode);

   // 现在处理该节点的子节点
   var children = nd.childNodes;
   for(var i=0; i < children.length; i++) {
      outputNodeProps(children[i]);
   }
}

window.onload=function() {
   outputNodeProps(document.body);
}
//]]>
</script>
</head>
<body>
<div id="div1">
<h1>Header</h1>
<!-- paragraph one -->
<p>To better understand the document tree, consider a web page that has a head
and body section, a page title, and the body contains a div element that itself
contains a header and two paragraphs. One of the paragraphs contains
<i>italicized text</i>; the other has an image--not an uncommon web page.</p>
<!-- paragraph two -->
<p>Second paragraph with image. <img src="ospreythumb.jpg" alt="bird" /></p>
</div>
</body>
</html>
```

在该应用程序中，当 nodeValue 属性值不是 null 时，其 style 属性就将被设置。不过，当 nodeValue 属性有值时（即使是空白符），style 属性也是未设置的。图 11.5 中展示了该页面在 Safari 浏览器中的执行过程。

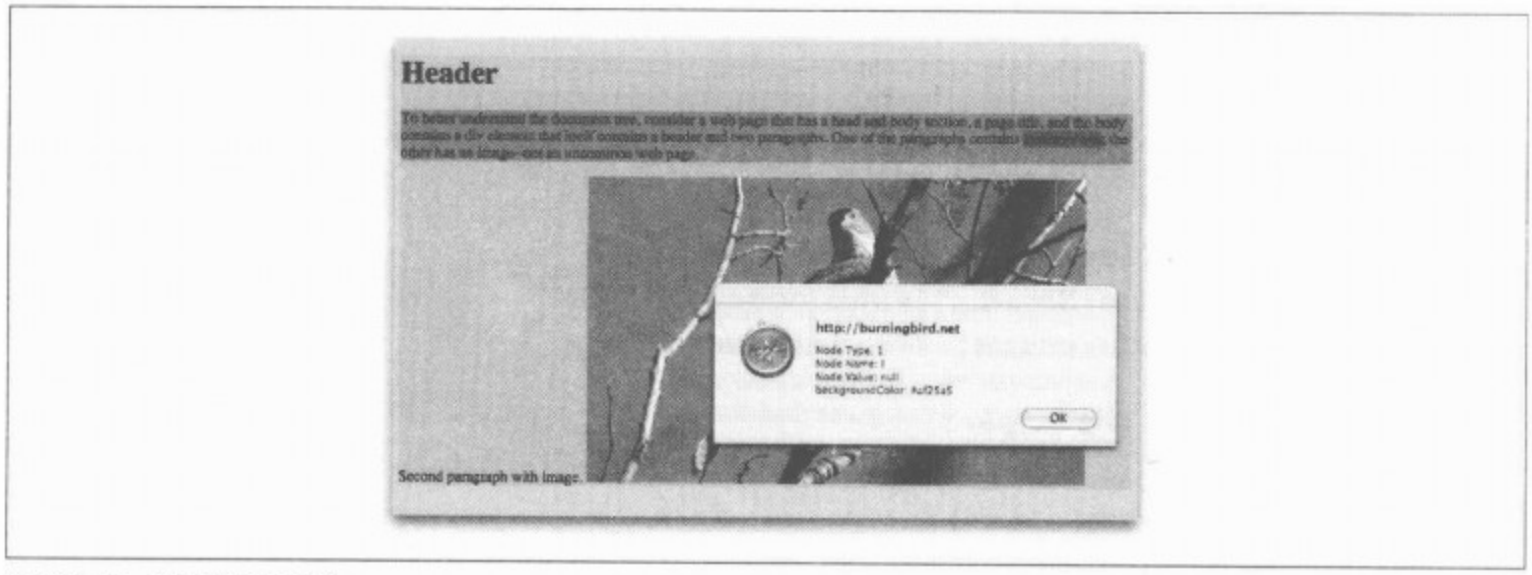


图 11.5　遍历页面树

另外，还要注意有些元素内会有文本，如 p（段落）元素，实际上包含的是一个指向 text 节点的引用，而文本信息是保存在 text 节点中的。一个简单的 Web 页面中拥有的节点数足以令你感到惊讶。

并不是所有浏览器都会给出相同的输出。例如，IE 对于标签外面的空白符是不会创建 text 对象的，而其他浏览器则会。如果你要创建一个能够修改 Web 页面结构的 JavaScript 程序，那么理解并记住它是十分重要的。

提示

遍历文档树的另一种方法是使用 W3C 中可选的 Level 2 遍历和排列规范（详细内容可以参阅 http://www.w3.org/TR/DOM-Level-2- Traversal-Range/）。该规范提供了一个 API，用来完成更复杂的遍历，还提供了排列对象的功能。

节点的 nodeType 属性是数字型的。与其使用 3 或 8 之类的值来搜索特定的节点，不如使用 DOM 规定的一组常量，它们分别表示一种具体的类型。这些常量包括：

- ELEMENT_NODE　Value of 1
- ATTRIBUTE_NODE　Value of 2
- TEXT_NODE　Value of 3
- CDATA_SECTION_NODE　Value of 4
- ENTITY_REFERENCE_NODE　Value of 5
- ENTITY_NODE　Value of 6
- PROCESSING_INSTRUCTION_NODE　Value of 7
- COMMENT_NODE　Value of 8
- DOCUMENT_NODE　Value of 9

- DOCUMENT_TYPE_NODE　Value of 10
- DOCUMENT_FRAGMENT_NODE　Value of 11
- NOTATION_NODE　Value of 12

使用这些常量，能够让你的代码更具可读性，无须记住每个数字表示什么意思。对于示例 11.5 而言，你可以添加以下代码，这样将显示出有意义的消息而不是数字型的 nodeType：

```
switch(nd.nodeType) {
  case Node.ELEMENT_NODE : strNode += "\nNode type is Element"; break;
  case Node.TEXT_NODE : strNode += "\nNode type is text"; break;
  case Node.COMMENT_NODE : strNode += "\nNode type is comment"; break;
  default : strNode += "\nNode type isn't specified"; break;
}
```

不幸的是，该常量的实现并不普遍。IE 就没有实现这些常量，当它们在 IE 中使用时将没有任何意义。不过有一种解决方案，那就是对 Node 对象进行扩展，使其包含这些常量，就像以下的代码所示，我们可以指定自己的常量：

```
Node.SHELLEY_TYPE = 23;
if (23 == Node.SHELLEY_TYPE) alert("I'm cool, I'm a nodeType constant");
```

不过，除了扩展对象之外，还可以创建一个全局的自定义对象，把这些值存到一个数组的属性中。在第 13 章中，我将更详细地介绍自定义对象。

除了自我标识和导航功能之外，Node 还提供了几个用来替换或插入新节点的方法。这些方法通常会和 document 对象协同使用。

11.3.3　DOM 核心文档对象

正如你所预料的那样，document 对象是与 Web 页面文档之间的核心接口。它提供的方法能够用来创建、删除页面元素，也能够控制它们在页面中出现的位置。它还提供了两个访问页面元素的方法，在之前的章节中都已经学习过了：getElementById 和 getElementsByTagName。

getElementsByTagName 方法将返回一个节点列表（NodeList），它表示的是页面中的所有指定标签的页面元素：

```
var list = document.getElementsByTagName("div");
```

接下来就可以遍历这个列表，并对每个节点做所需的处理。

提示

如果该文档的 DOCTYPE 设置为 HTML 4.01，那么所有元素引用都是大写的。如果该文档的 DOCTYPE 设置为 XHTML 1.0 或更高版本，那么元素标签将都是小写的。不过，我发现绝大多数浏览器都能够接受大写的标签名，即使 DOCTYPE 被设置成 XHTML。

为了展示 getElementsByTagName 方法，在示例 11.6 中使用了一个帧集，在一个面板中载入源文档，另一个载入脚本文档。

示例 11.6　一个打开示例页面和带有脚本的活动页面的帧集

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Frameset//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-frameset.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>Highlighting</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
</head>
<frameset cols="80%,*">
<frame name="docin" src="ch11-04.html" />
<frame name="docout" src="ch11-07.html" />
</frameset>
</html>
```

其中一个帧显示的页面是 ch11-07.html，其内容在示例 11.7 中列出了，它有 3 个用 div 元素实现的“按钮”，当点击它时，将显示 3 个值：强调显示的颜色、打开的源窗口以及搜索的元素标签。

示例 11.7　脚本页面将在帧中打开另一个文档，并且强调显示所有指定类型的元素

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>Page as Tree</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<style type="text/css">
div {
    border: 1px solid #000;
    padding: 5px;
}
</style>
<script type="text/javascript">
//<![CDATA[

var highlightColor = "#ffff00";
function changeColor() {
  highlightColor=prompt("Enter highlight color (hexidecimal format)",
highlightColor);
}

function loadPage() {
   var pageURL = prompt("Enter page in this domain","");
   top.docin.location.href=pageURL;
}
```

```
function highlightElements() {
   var elemTag = prompt("Enter tag element name to highlight:","p");
   var nodes = top.docin.document.getElementsByTagName(elemTag);

   // 强调显示每个元素
   for (var i = 0; i < nodes.length; i++) {
       var mynode = nodes[i];
       mynode.style.backgroundColor=highlightColor;
   }
}

//]]>
</script>
</head>
<body>
<div onclick="changeColor()">
<p>Click to change highlight color</p>
</div>
<div onclick="loadPage()">
<p>Click to load source page</p>
</div>
<div onclick="highlightElements()">
<p>Click to search for, and highlight, a specific tag</p>
</div>
</body>
</html>
```

该应用程序将在第一个面板中打开源文档，然后查找所有指定类型的元素，再使用指定的颜色强调显示它们。在本例中只有几个元素，包括列表项、段落以及 SVG（可伸缩的向量图）元素，如图 11.6 所示。

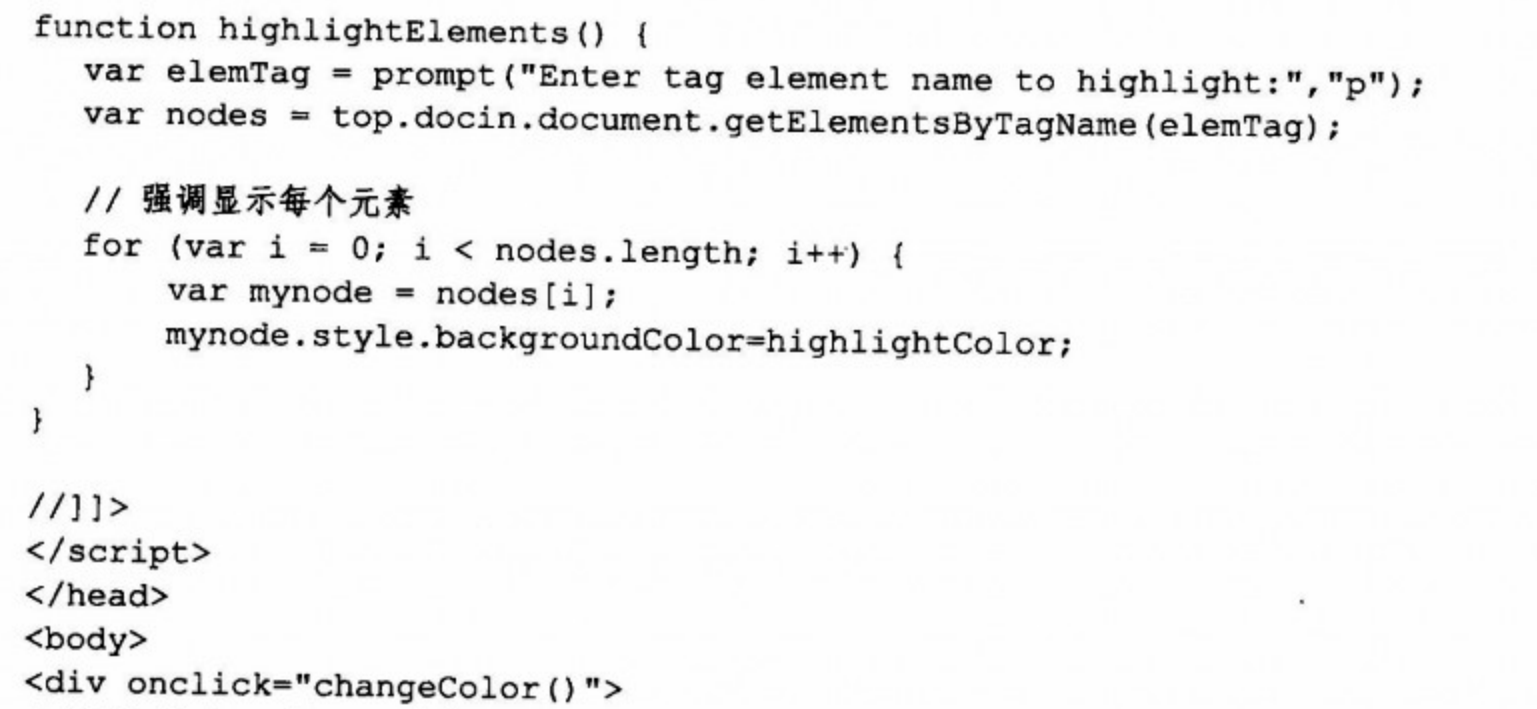
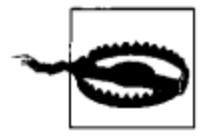


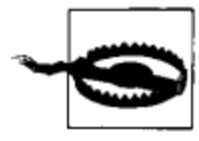
图 11.6　强调显示某些元素



警告

在示例 11.7 中，并不能载入任何文档。JavaScript 沙箱会阻止我在应用程序页面中调用属于另一个域的 document 对象的 getElementsByTagName 方法。换句话说，只要处理的页面和脚本页面属于同一个域，那么该应用程序就能够正常工作，否则就不行。

该脚本也能够对自己所在页面进行处理，在你想基于某个事件强调显示页面中的一类元素时特别有用，例如所有 input 元素或缩略图。



警告

由于在元素样式设置方面存在一个已知的缺陷，因此示例 11.7 是无法在 IE 8.0 beta2 中正常运行的，除非使用兼容性 meta 标签或改成 Compatibility View（兼容视图）。

另外一种访问元素的方法是新实现的 getElementsByClassName。该方法将返回一个 nodeList，它是由拥有指定类名的元素组成的，例如有一个如下所示的 div 元素：

```
<div class="test">...</div>
```

要访问这个 div 元素可以使用：

```
var nds = document.getElementsByClassName("test");
```

它有一种变体，那就是用空格列出多个类名，这样将得到类名与其中任何一项相等的所有元素的列表：

```
var nds = document.getElementsByClassName("test test2");
```

这是一个十分便于使用的方法。不幸的是，它只在 Safari 3.1、Opera 9.5、Firefox 3.x 及后续版本中实现；在 IE 中并未实现，包括 IE 8.0。

除了 getElementsByTagName 和 getElementByClassName 方法之外，document 对象中还有几个能够创建新对象的方法。我们将在本章稍后的“修改文档树”小节中介绍。不过我们首先还是看看 Element 对象以及其上下文的概念。

提示

此外还有一个新的名为 querySelector 的查询方法，它比 getElementsByClassName 更好，我们将在第 12 章中展示它的使用。

11.4 元素及其上下文内访问

在 DOM 核心中还有另一个重要的元素，那就是 Element。文档中所有页面元素都将从 Element 元素中继承 API 和属性。其主要功能是获取和设置属性值，以及检查属性是否存在：

- getAttribute(name)
- setAttribute(name,value)
- removeAttribute(name)
- getAttributeNode(name)
- setAttributeNode(attr)
- removeAttributeNode(attr)
- hasAttribute(name)

除此之外还有其他方法，绝大多数是与属性对应命名空间相关的，不过对于 Web 页面而言，这些方法是很少使用的。

Element 属性并不总是能够通过对象属性访问的。属性是对象类的组成部分，而不是类的实例。因此，属性可以与 document、Element、Node 甚至是如 HTMLDocumentElement 的 HTM 元素对象相关联。不过，如果你想使用元素的属性，而且它们不是对象类输出的属性，那么可以使用 Element 的方法。

以下是嵌入一个 Web 页面上的图像：

```
<img src="dotty.gif" width="100" alt="an image" align="left" />
```

以下代码将使用 Element.getAttribute 方法访问图像的属性，然后将它们连接成一个字符串，并输出在一个 alert 对话框中：

```
var img = document.images[0];
var imgStr = img.getAttribute("src") + " " +
             img.getAttribute("width") + " " +
             img.getAttribute("alt") + " " +
             img.getAttribute("align");
alert(imgStr);
```

下面的代码将使用 Element.setAttribute 方法修改 width 和 alt 属性值：

```
img.setAttribute("width","200");
img.setAttribute("alt","This was an image");
```

Element 也和 document 对象共享了一个方法——getElementsByTagName。只不过它处理的不是文档中的所有元素，而是根据文档上下文来处理元素。

实际上，在本书迄今为止的所有示例中，我们处理的或多或少都是 document 对象上下文中的东西。在很大程度上，这已经足够了。不过，有时你可能需要处理嵌套在其他元素中的元素。通过这个从 DOM Core API 中继承的功能，特别是 Node 和 Element 对象，你可以使用离散（discrete）的方法访问页面中的任何对象，如 getElementById 就能够在一个新的上下文中执行。

在下面这段 HTML 代码中，有两个包含段落元素的 div 元素：第一个 div 元素中有两个段落，第二个 div 元素中有一个段落。

```
<div id="div1">
<p>one</p>
<p>two</p>
</div>
<div id="div2">
<p>three</p>
</div>
```

这些段落元素都没有标识符，因此使用 getElementById 方法是无法访问每个段落元素的。所以你应该改用 getElementsByTagName 方法，并传入段落元素的标签（p）：

```
var ps = document.getElementsByTagName("p");
```

不过，如果这样做你将获得文档中所有的段落元素。这可能不是你所想要做的，但如何才能只获得第一个 div 元素中的段落元素呢?

要访问位于第一个 div 元素上下文中的段落元素，可以使用 getElementById 方法来访问这个 div 元素：

```
var div = document.getElementById("div1");
```

然后，由于 div 元素是从 Element 对象中继承的，因此可以使用 getElementsByTagName 来获取其包含的所有段落元素：

```
var ps = div.getElementsByTagName("p");
```

现在返回的节点列表中包含的段落元素就只有嵌套在标识符为 div1 的第一个 div 元素内的那两个。

随着越来越多的 Web 页面在设计时使用了 CSS，并且其元素是构建在层中的，而且还嵌套在其他层里，因此在某个元素的上下文中执行，是一种控制 JavaScript 应用程序影响页面中哪些组件的方法。当你使用这种方法修改文档时，它不是十分显而易见的。

11.5 修改文档树

之前，你已经看到如何使用 Element 对象的方法来修改特定的元素对象。要修改整个文档树还需要用 3 个不同的对象。

document 是所有页面元素的所有者/父节点。正是因为有这样的所有者关系，绝大多数用来创建新元素实例的工厂方法都是核心 document 对象的方法。不过，负责维护 Core API 中 navigation 对象的是 Node 对象，它提供了层次结构的文档树支持，每一个节点和其他节点间的关系和导航操作都遵循以下结构：父/子关系、兄弟/兄弟关系。最终，Element 元素提

供了一种访问其上下文内元素的方法，以对嵌套在该元素内的元素进行修改。当需要修改文档树时，这 3 个是最为本质的对象。

表 11.1 中列出了 document 对象的工厂方法，以及它们能够创建的核心对象类型。在该表中还提供了几个核心对象的简介。

表 11.1　document 对象的工厂方法

方　法	创建的对象	描　述
createElement(tagname)	Element	创建特定类型的元素
createDocumentFragment	DocumentFragment	DocumentFragment 是个轻量级的 document 对象，用于获取文档树中的一个小节
createTextNode(data)	Text	保存页面中的文本
createComment(data)	Comment	XML 注释
createCDATASection(data)	CDATASection	CDATA 小节
CreateProcessingInstructions (target,data)	ProcessingInstruction	XML 处理指令
createAttribute(name)	Attr	元素属性
createEntityReference(name)	EntityReference	创建后面放置的元素
CreateElementNS (namespaceURI,qualifiedName)	Element	创建带有命名空间的元素
CreateAttributeNS (namespaceURI,qualifiedName)	Attr	创建带有命名空间的属性

创建一个新节点（如 text 节点）很简单。你需要做的只是在 document 对象上调用相应的工厂方法即可，它将返回你需要的节点：

```
var txtNode = document.createTextNode("This is a new text node");
```

回忆一下，我们曾经说过工厂方法是创建不带构造函数的对象的一种方法。

当你拥有一个新节点之后，你可以像操作原有页面元素那样操作它。如果要在一个新的 div 元素中添加一个新的段落元素和一个 text 节点，可以使用以下代码：

```
var newDiv = document.createElement("div");
var newP = document.createElement('p');
var newText = document.createTextNode('New paragraph');
newP.appendChild(newText);
newDiv.appendChild(newP);
```

当你创建了一个新节点之后，就可以使用如下所示的 Node 方法将其直接添加到原有的文档树上：

- insertBefore(newChild,refChild)　在 refChild 指定的原有节点前插入一个新节点；

- replaceChild(newChild,oldChild)　替换原有的节点；
- removeChild(oldChild)　删除现有的子节点；
- appendChild(newChild)　为 document 对象添加一个子节点。

不过请记住，你必须在正确的上下文中调用这些方法。换句话说，你需要在包含要替换或删除的节点（或新节点要添加的位置）的元素上执行以上操作。

如果 Web 页面中有一个带有嵌套标题行的 div 元素，并且要替换的就是这个标题行，那么你需要访问这个 div 元素以修改其结构：

```
var div = document.getElementById("div1");
var hdr = document.getElementById("hdr1");
div.removeChild(hdr);
...
<div id="div1">
<h1 id="hdr1">Header</h1>
</div>
```

示例 11.8 更全面地演示了这些操作，它是本章上一个使用的静态页面的示例的一个变体。它包含段落、div 元素、图像和标题行。其脚本中有一个名为 changeDoc 的函数，当页面被点击（未载入）时将触发它。该脚本将操作页面文档树，删除原来的标题行，将图像替换成文本，然后在文档的最后添加一个新的标题行。

示例 11.8　修改一个文档

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
<head>
<title>Modifying Document</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript">
//<![CDATA[

document.onclick=changeDoc;

function changeDoc() {

   // 首先，删除标题行
   var hdr = document.getElementById("hdr1");
   var div = document.getElementById("div1");
   div.removeChild(hdr);

   // 将图像替换成文本
   var img = document.getElementById("img1");
   var p = document.getElementById("p2");
   var txt = document.createTextNode("New text node");
   p.replaceChild(txt,img);
```

```
    // 添加一个新元素
    var div2= document.createElement("div");
    div2.innerHTML="<h1>The End</h1>";
    document.body.appendChild(div2);
}
//]]>
</script>

</head>
<body>
<div id="div1">
<h1 id="hdr1">Header</h1>
<!-- paragraph one -->
<p id="p1">To better understand the document tree, consider a web page that
has a head and body section, has a page title, and contains a DIV element that
itself contains an H1 header and two paragraphs. One of the paragraphs contains
<i>italicized text</i>; the other has an image--not an uncommon web page.</p>
</div>
<!-- paragraph two -->
<p id="p2">Second paragraph with image. <img id="img1" src="dotty.gif"
alt="dotty" /></p>
</body>
</html>
```

图 11.7 展示了页面修改之前的效果，而图 11.8 则展示了修改之后的页面效果。

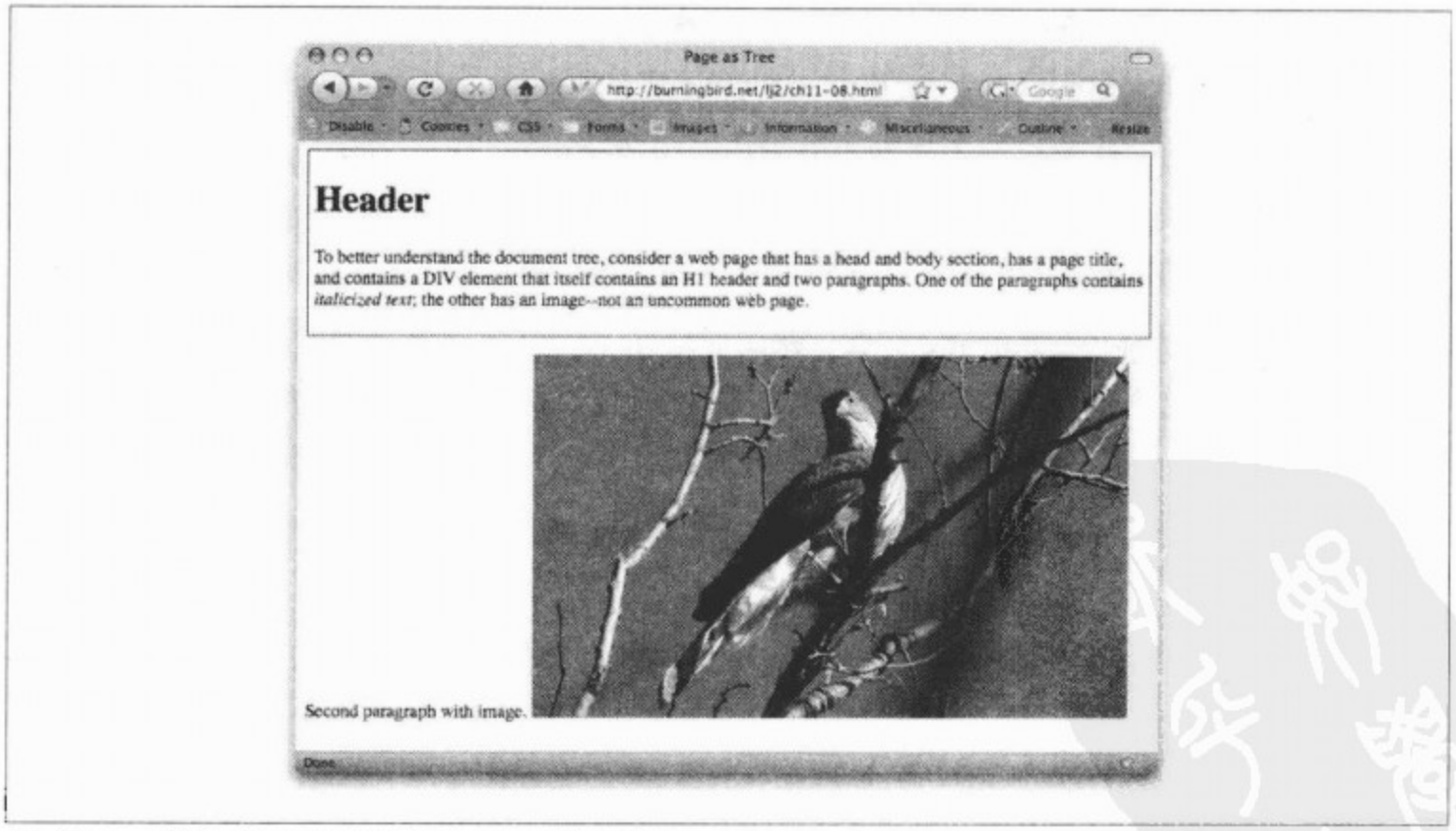


图 11.7　示例 11.8 中的 Web 页面（修改前）

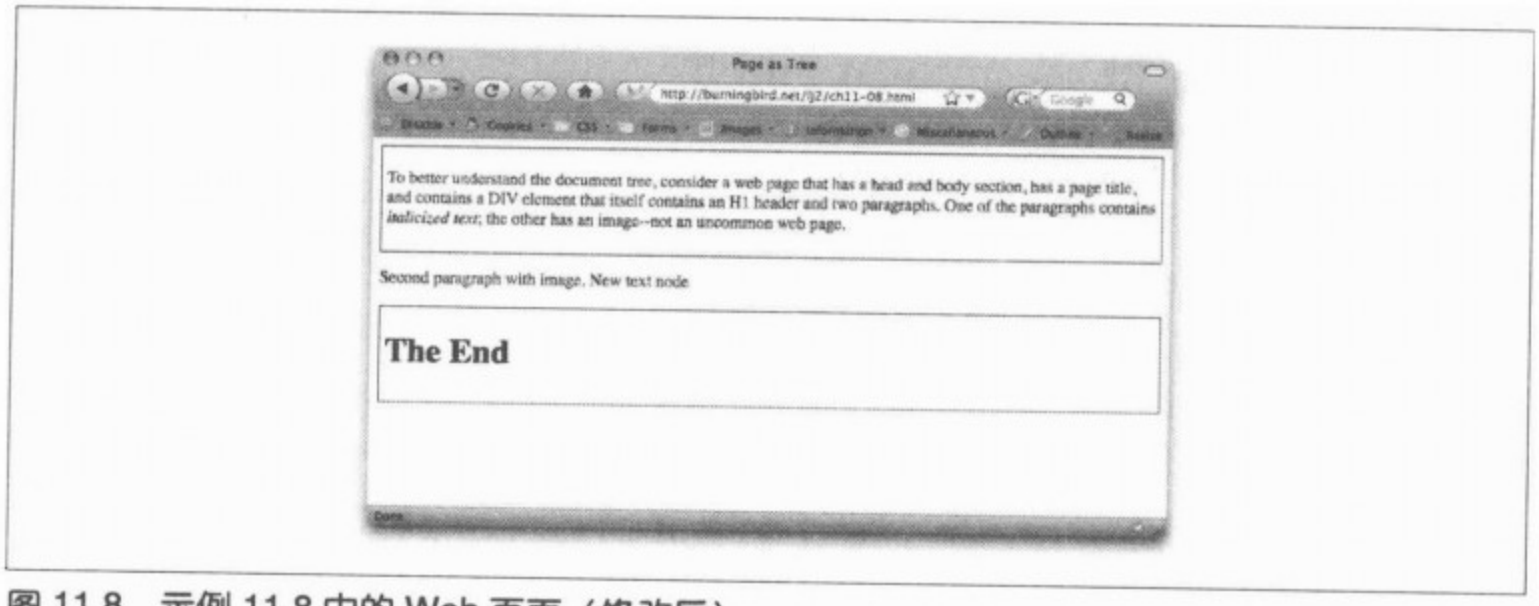
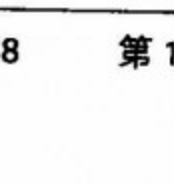


图 11.8　示例 11.8 中的 Web 页面（修改后）

对该文档的第一处修改是删除 div 元素中的标题行。访问这个 div 元素和其包含的标题行元素时使用的都是 getElementById 方法。当应用程序获得这个 div 元素和它包含的标题行元素之后，就将标题行元素作为参数传给在 div 元素上调用的 removeChild 方法，这样就将把该元素从页面中的这个 div 元素里删除。



提示

如果你想在 DOM 树中添加大量的节点，那么先使用 createDocumentFragment 方法创建一个 documentFragment，接着为其添加节点，然后将 documentFragment 添加到文档树中，这样做性能会有所增强。如果你对“通过 documentFragment 改进性能”感兴趣，可以阅读 John Resig 的博客（http://ejohn.org/blog/dom-documentFragments/）。

接下来的一处修改是将最后一个段落中的图像元素换成用 text 节点创建的文本信息。在将图像和段落载入到 JavaScript 变量时，使用的都是 getElementById 方法。然后创建了一个新的 text 节点，并将它和图像节点作为参数传给基于段落节点调用的 replaceChild 方法。

最后一处修改是创建一个新的 div 元素，然后使用 innerHTML 属性在新的 div 元素中添加一个新的标题行，这个示例再次混用了传统的 BOM 和新的 DOM 功能。然后将这个新的 div 元素添加到 body 元素上，这样就在文档最后面输出了一个写着“The End”的标题行。

本章到此就结束了，不过后面还有不少内容要继续讲解。

11.6　知识测验

1. 所有 HTML 元素都支持的属性有哪些？
2. 当指定一个已命名的元素时，如何使用 HTML DOM 找到其元素类型？
3. 指定一个位于核心 DOM 中的节点，如何输出其每个子节点的元素类型？

4. 如何找到指定页面中所有 div 元素的 ID（标识符）？

5. 对于下面这个元素，如果要访问它有哪三种不同类型的方法？

```
<div id="elem1" class="thediv">...</div>
```

6. 除了使用 innerHTML，如何将下面这个位于 div 元素中的标题行元素换成一个段落元素：

```
<div id="elem1">
<h1>This is a header</h1>
</div>
```

11.7 测验答案

1. 这样的属性包括 id、title、lang、dir 和 className。

2. 访问该元素的 tagName 属性。

3. 其解决方案是：

```
var children = targetNode.childNodes;
for (var i = 0; i < children.length; i++) {
   alert(children[i].nodeType);
}
```

4. 其解决方案是：

```
var divs = document.getElementsByTagName('div');
for (var i = 0; i < divs.length; i++) {
   alert(divs[i].id);
}
```

5. 这三种方法分别是：

```
var theDiv = document.getElementById('elem1');

var theDivs = document.getElementsByTagName('div');
var theDiv = theDivs[0];

var theDivs = document.getElementsByClassName('thediv');
var theDiv = theDivs[0];
```

6. 其解决方案是：

```
var targetElement = document.getElementById('elem1');
var specChild = targetElement.getElementsByTagName('h1')[0];
var newPara = document.createElement('p');
var paraTxt = document.createTextNode('hello');
newPara.appendChild(paraTxt);

targetElement.replaceChild(newPara,specChild);
```

第 12 章

动态页面

使用 JavaScript 创建的动态 Web 页面，先前通常称之为 DHTML（全称为 Dynamic HTML，即动态 HTML），而现在通常称之为 Ajax，它已经出现了近 12 年了。这一概念背景的关键元素除了 DOM 之外就是 CSS 了，虽然在早先几年没有通用的对象模型。

通过 CSS 就可以定义页面元素的外观，而无须依赖于外部应用程序、插件或者额外的图像。通过 CSS 和样式表，还能够将页面元素的展现与其组织分离开。

通过 DOM 甚至能够在 JavaScript 中访问样式表属性，在页面载入完成之后修改每个元素的属性。加上 CSS 的出现，这就构成了一种强大的手段，使得 Web 页面比以前更具交互性。

12.1 JavaScript、CSS 和 DOM

本章假定你对 CSS 比较熟悉，也知道如何为 Web 页面添加样式表。如果你对 CSS 不熟悉，那么最好在阅读本章之前先读一本关于 CSS 的好书。我推荐你看 Eric A. Meyer 写的 *CSS：The Definitive Guide*（中译版《CSS 权威指南》）。

12.1.1 样式属性

通常，你会通过 style 属性查询和设置 CSS 样式属性。样式作为属性的概念最初是微软提出的，而 W3C 是在 DOM Level 2 才采纳并放入其 CSS 模块中的。通过 W3C 的 DOM 规范，所有节点都有一个保存相关 style 对象的属性，这也就意味着你可以在 JavaScript 中修改任何页面元素的样式属性[1]。

要通过 JavaScript 修改任何样式设置，你首先必须使用一个上一章中介绍过的 DOM 方

[1] 译者注：原文中关于属性用了 Property 和 Attributes 两个词，Property 通常是对象、类的属性，Attributes 则表示是页面元素的属性；在译文中就不予以区分了。

法，以获得要处理的元素（或几个元素）：

```
var tstElem = document.getElementById("testelement");
```

要修改其样式属性，可以直接使用赋值语句：

```
tstElem.style.color="#fff";
```

这样的方法适用于任何有效的 CSS2 属性、有效的 X/HTML 对象以及相应元素的属性。例如，span 元素通常不会以块式显示（自动在该元素前后都添加分行符），因此设置其顶边距和底边距实际上没有什么意义。

示例 12.1 展示了修改 CSS 属性的方法，首先使用大家很熟悉的 getElementById 方法获取一个 div 元素，然后使用 style 对象设置不同的 CSS 属性。

示例 12.1　修改 div 元素的样式属性

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>DHTML</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript">
//<![CDATA[

window.onload=function() {
  setTimeout("changeCSS()",3000);
}

function changeCSS() {
  var div = document.getElementById("div1");
  div.style.backgroundColor="#f00";
  div.style.width="500px";
  div.style.color="#fff";
  div.style.height="200px";
  div.style.paddingLeft="50px";
  div.style.paddingTop="50px";
  div.style.fontFamily="Verdana";
  div.style.borderColor="#000";
}
//]]>
</script>
</head>
<body>
<div id="div1">
<p>Hello CSS</p>
</div>
</body>
```

```
</html>
```

请注意这个示例中 CSS 属性所采用的命名规范。如果 CSS 属性中有连字符，如 border-color，那么将去除连字符，并且第二个词的首字母用大写表示：也就是说 CSS 中的 border-color 在 JavaScript 中就变成了 borderColor。除此之外，在 JavaScript 中使用的 CSS 属性名和样式表中的属性名是一样的。

提示

CSS 中的 float 属性是个例外。在 IE 中使用的是 styleFloat，而其他浏览器则使用 cssFloat。

如果说修改样式属性很简单，那么读取样式属性就更简单了。如果你没有通过 JavaScript 设置样式属性，或者使用元素中的内联样式属性，即使你在样式表中设置了该值，那么属性值仍然要么是空的，要么是未定义的。这点很重要，因为当你动态修改元素的 CSS 属性时它可能会比别的东西更容易导致失败。最初呈现对象时将使用什么样式，取决于浏览器的内部设置，以及样式表中的设置和元素继承得到的样式。

警告

除非你通过 JavaScript 或直接使用元素的内联样式属性设置样式，否则当你通过脚本访问它时其值将是空的或者是未定义的，即使你已经通过样式表为其设置了值。

要访问这样的样式，需要使用其他属性，对于不同类型的浏览器要使用的属性是不同的。微软和 Opera 支持的是 currentStyle 属性，而 Mozilla 和 Safari/WebKit 则支持的是 window.getComputedStyle。不幸的是，它们在不同浏览器中无法保持一致。

对于 getComputedStyle 方法，你传入 CSS 属性时，必须采用和样式表中设置样式时相同的语法。不过，对于 currentStyle 方法而言，你将使用 JavaScript 符号。

示例 12.2 展示了该函数的一个变体，它将获取一个对象和一个特定 CSS 属性的样式设置。首先，它将检查浏览器是否支持 currentStyle 方法，如果不支持则检查是否支持 getComputedStyle 方法。接着通过一个 alert 对话框输出一个包含实际样式设置的消息。然后通过脚本设置其背景颜色，在设置之后将直接显示出样式属性，以说明当通过 JavaScript 设置了样式属性之后，其值就可以直接访问了。

示例 12.2　尝试获得 CSS 样式信息

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>DHTML</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<style type="text/css">
   #div1 { background-color: #ffff00 }
```

```
</style>
<script type="text/javascript">
//<![CDATA[

function getStyle(obj,jsprop,cssprop) {
   if (obj.currentStyle) {
      alert('currentStyle ' + obj.currentStyle[jsprop]);
   } else if (window.getComputedStyle) {
      alert('getComputed ' +
document.defaultView.getComputedStyle(obj,null).getPropertyValue(cssprop));
   } else {
     alert("neither currentStyle nor getComputed supported");
   }
}

function changeElement() {
    var obj = document.getElementById("div1");
    alert(obj.style.backgroundColor);
    getStyle(obj,"backgroundColor","background-color");
    obj.style.backgroundColor="#ff0000";
    alert(obj.style.backgroundColor);
}

document.onclick=changeElement;

//]]>
</script>
</head>
<body>
<div id="div1">
<p>Hello CSS</p>
</div>
</body>
</html>
```

在 Firefox 和 Safari 中打开并点击该页面，这时将返回一个 alert 对话框，首先是空值，然后是 getComputed rgb(255,255,0)，最后是 rgb(255,0,0)。如果在 Opera 和 IE 中载入该页面，那么将分别显示空值、currentStyle #ffff00 和#ff0000，只不过 IE 在第二个 alert 对话框中将显示 currentStyle #ffff00。

注意，在这段脚本中获取已计算值的语法是 document.defaultView.getComputedStyle，而不是 window.getComputedStyle。document.defaultView 属性是一个 DOM AbstractView 对象，它是所有视图的基类。你可以使用 window.getComputedStyle 方法，但是无法确保 window 元素中设置的就是 defaultView（当前视图），因此在不同浏览器上的执行结果是不可靠的。

即使当样式属性可访问时，其返回的内容也会因浏览器的不同而不同，我们来看一些

简单的实例，例如字体颜色。

Opera 将返回十六进制的颜色值：

```
#ff0000
```

IE 也将返回十六进制的颜色值，只不过它会采用简版描述，例如#ff0。但在 Firefox 和 Safari 中将返回 RGB 设置：

```
RGB(255,0,0)
```

如果你想得到一致的结果，就需要在 RGB 和十六进制两个表示法之间进行转换，而且还需要在简版和完整版的十六进制格式之间转换。

如果 div 元素的背景颜色未设置，那么它们将返回什么呢？假设 body 元素指定了背景颜色，但 div 元素没有，那么你会获得什么呢？

在 Firefox、IE 和 Opera 中，对于 div 元素而言默认的背景颜色是“transparent”（透明）。不过，在 Safari 中虽然默认背景颜色也是透明，但它不会返回“transparent”，而是返回一个等价的 RBGA（红-绿-蓝-Alpha）表达式 rgba(0,0,0,0)，最后一个值是 alpha 透明度设置，在本例中设置的是 0。

获取页面中的样式设置是充满挑战的，或许也比较有趣，或许也相当有用。当某个属性未直接设置时，那么其派生值是基于 CSS 规则计算的，因此不仅仅是在不同浏览器中可能不同，而且可能是永远不相同。Prototype、Dojo 和 jQuery 之类的 JavaScript 程序库之所以流行，正是因为它们能够解决这些差异，你就无须自己考虑了。

你在实现基于 CSS 的动态效果时，应尽量避免直接从页面样式设置中获取信息，这是一个不错的规则。另外，只要可能，应该使用程序变量来保存这些值，而且只使用设置属性时所需的样式。

CSS 样式属性通常可以分成以下几种类型：字体、边框、元素容器、定位、显示等。在本书接下来的部分中，我们将介绍几类属性，演示如何通过 JavaScript 处理它们。如果你打算通读本章，那么请不要犹豫，花些时间停下来尝试一下这些示例。

12.2 字体和文本

第一个呈现性的 HTML 元素就是 font（字体），它也是一个可以从老版本 HTML 中找到的元素，在所有的 Web 页面中都很常用。在构建 Web 页面时，字体和文本属性显然是常用的。修改文本或字体属性所得到的效果，很少有对元素样式属性的修改能够达到的。

注意，我说的是“文本或字体属性”，字体属性是针对字符本身的：字体集、大小、类型，以及其他与字符外观有关的元素。而文本属性则是对文本的修饰，包括文本修饰、

对齐方式等。

12.2.1 字体样式属性

字体有一些特定的样式属性。下面所列出的就是 CSS 名称和其相关的 JavaScript 可访问的样式属性：

- font-family　在 JavaScript 中可以通过 fontFamily 访问它。它用来修改字体的字体集（如 serif、Arial 或 Verdana）。如果你指定的字体集是由多个单词组成的，那么需要精确地给出字体集名称，包括其中的空格；
- font-size　在 JavaScript 中可以通过 fontSize 访问它。它用来设置字体大小。在设置字体大小时可以采用不同的单位。如果你使用 em 或 pt（如 12pt 或 2.5em）作为单位，那么字体大小会根据 Web 页面读者自己的设置而改变。如果你使用 px（像素），那么字体的大小和用户设置是无关的。你也可以使用一些预定义的字体大小，如 xx-small、x-small、small、medium、large、x-large 或 xx-large；还可以使用相对的大小设置，如 smaller 或 larger；另外还可以用百分比（相对于父元素的大小）来设置；
- font-size-adjust　在 JavaScript 中可以通过 fontSizeAdjust 访问它。它是字母 x 的高度与字体大小中指定高度之间的比值。该比值的设置是受限的，不过它也很少被用到；
- font-stretch　在 JavaScript 中可以通过 fontStretch 访问它。它用来扩展或压缩字体。其可选值包括：normal、wider、narrower、ultra-condensed、extra-condensed、condensed、semi-condensed、semi-expanded、expanded、ultra-expanded 和 extra-expanded；
- font-style　在 JavaScript 中可以通过 fontStyle 访问它。可选值包括 normal（默认）、italic 和 oblique；
- font-variant　在 JavaScript 中可以通过 fontVariant 访问它。如果你想使用特定字体的 samll-cap 变体，那么就将其值设置成 small-caps；
- font-weight　在 JavaScript 中可以通过 fontWeight 访问它。使用它可以设置字体的浓淡。其可选值包括 normal、bold、bolder、lighter；也可以用数字值，如 100、200、300、400、500、600、700、800 或 900。

使用 font 本身就可以一次性修改多个字体属性。下面就是一个实例：

```
div1.style.font="italic small-caps 400 14px verdana";
```

在此用来修改样式的一个字符串中包含了变体、浓淡、大小和字体集。CSS 属性中有很多像这样的便捷设置方法。在 JavaScript 中可以像在 CSS 中那样使用。在 CSS 中，冒号右边的都是样式设置；在 JavaScript 中，应该将其放在一个引号中，并放在赋值语句的右边。

12.2.2 文本属性

对于本章而言，我决定将影响文本显示的几个属性分成一组，它和字体属性不同，也不属于同一类。以下是我们经常设置的 CSS 文本属性。

- color　在 JavaScript 中可以用 color 访问它。它用来设置文本的颜色。
- line-height　在 JavaScript 中可以用 lineHeight 访问它。它用来设置两行之间的间距。为其指定的值和为字体大小设置的值一样，或者也可以设置为 normal。
- text-decoration　在 JavaScript 中可以用 textDecoration 访问它。可以设置为 none、underline、overline 或 line-through。请不要使用 blink。
- text-indent　在 JavaScript 中可以用 textIndent 访问它。它用来设置首行缩进量。
- text-transform　在 JavaScript 中可以用 textTransform 访问它。它可以设置为 none、capitalize（将每个单词的第一个字母变成大写）、uppercase 或 lowercase。
- white-space　在 JavaScript 中可以用 whiteSpace 访问它。可以设置为 normal、pre 或 nowrap。
- direction　在 JavaScript 中可以用 direction 访问它。可以设置为 ltr（从左到右）或 rtl（从右到左）。
- text-align　在 JavaScript 中可以用 textAlign 访问它。它用来设置文本的对齐方式，可以设置成 left、right、center 或 justify。
- word-spacing　在 JavaScript 中可以用 wordSpacing 访问它。它用来设置单词之间的空白位置，可以设置为 normal 或指定的长度值。

哪些字体或文本属性是经常修改的呢？最有用的修改之一就是增大一块文本的大小，使其更易于阅读，另外就是强调显示某些信息。在示例 12.3 中，页面上有两个链接，点击其中一个可以使文本变得很大，点击另一个可以使其回到原来的大小。注意在此最初是用样式表设置文本属性的，因此当文本“回到”较小的大小时，字体和文本设置是一样的，而不管每种浏览器上的默认设置是什么。

示例 12.3　修改一个文本块

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>readThis</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<style type="text/css">
div
{
```

```
    width: 600px;
}
#smaller
{
    background-color: #00f;
    cursor: pointer;
}
#larger
{
    background-color: #0f0;
    cursor: pointer;
}
#div1
{
  font-size: 15px;
  letter-spacing: normal;
  text-align: left;
  text-transform: none;
  line-height: 18px;
  font-weight: normal;
}
</style>
<script type="text/javascript">
//<![CDATA[
function makeMore() {
    var div = document.getElementById("div1");
    div.style.letterSpacing="5px";
    div.style.textAlign="justify";
    div.style.textTransform="uppercase";
    div.style.fontSize="25px";
    div.style.fontWeight="900";
    div.style.lineHeight="30px";
}

function makeLess() {
    var div = document.getElementById("div1");
    div.style.fontSize="15px";
    div.style.letterSpacing="normal";
    div.style.textAlign="left";
    div.style.textTransform="none";
    div.style.fontWeight="normal";
    div.style.lineHeight="18px";
}

window.onload=function() {
    document.getElementById('smaller').onclick=makeLess;
    document.getElementById('larger').onclick=makeMore;
}
//]]>
```

```
</script>
</head>
<body>
<div id="smaller">
<p>Make smaller</p>
</div>
<div id="larger">
<p>Make larger</p>
</div>
<div id="div1">
<p>
Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Praesent rutrum
erat a orci. Ut quis nisi. Curabitur eu nulla. Nullam vulputate tortor. Proin
scelerisque. Mauris eleifend odio non enim. Nunc tortor tortor, viverra at,
tempor eu, tincidunt in, risus. Duis libero. Aliquam a sapien et justo vehicula
volutpat. In at nibh in eros lacinia blandit. Donec nec ligula id nisl convallis
convallis. Suspendisse condimentum lacinia ante. Ut et ligula quis magna
pharetra rhoncus. Morbi ornare lobortis augue. Nunc convallis semper massa.
</p>
</div>
</body>
</html>
```

在你查看这个示例时（或者使用这种类型的调整时），可能会遇到文本大小没有增大的情况，不过它还是有效地展示了转换由 JavaScript 和 CSS 创建的样式的方法。另一种典型的应用场景是修改与某个表单元素或文本块相同的文字颜色，以说明它不匹配，从字面上看就是将其“淡出”。

12.3 定位和动画

在 CSS 出现之前，如果你想控制页面的布局，就必须使用 HTML 表格。要实现某种类型的动画，就只能够添加一些动画 GIF 或者如 Flash 之类的插件。

Netscape 和微软一起结束了这段历史，它们共同引入了一个名为 CSS-P 的规范，也就是 CSS 定位。如果你将页面看做一个图像，那么也有 x 和 y 坐标。使用 CSS-P，你可以通过这个坐标系统设置元素的位置。再加上 JavaScript，你还能够在页面上移动元素。

12.3.1 动态定位

它们提议的 CSS-P 属性最终被纳入了 CSS2 规范。在 CSS2 中，包含以下定位属性。

- position　该属性有 5 个可选值：relative、absolute、static、inherit 或 fixed。静态定位（static）是绝大多数元素的默认设置。这意味着它们是页面流的一部分，页面上的其他元素会影响该元素的定位，也会影响页面流中的所有元素。相对定位（relative）与静态定位相似，只不过元素可以通过偏移量来定位。如果将定位方式

设置成绝对定位（absolute），那么该元素就不再属于页面流了，你就可以在页面中为其设置绝对位置。这使你可以对元素进行分层，将一个元素叠在另一个元素上面，并放在页面的相同位置上。固定定位（fixed）与绝对定位相似，只不过该元素是基于某些视图进行定位的。

- top　元素的 top 属性用来设置它与容器顶部的相对位置。
- left　元素的 left 属性用来设置它与容器左边的相对位置。
- bottom　如果该属性的值为 0，则表示放在页面的底部。值越大，离页面底部越远。
- right　如果该属性的值为 0，则表示放在页面的最右边。值越大，元素就将位于页面的越左边。
- z-index　你有时可能需要添加 z-index 属性。如果你在页面上画一条垂直线，那么它就是 z-index。正如前面所说的那样，如果使用绝对定位，那么元素可以放在另一个元素的上面。它们将放在一个堆栈中，受到一两个因素的控制。第一个因素是它在页面上的位置。Web 页面中后定义的元素在堆栈中的位置更高；越早定义的元素在堆栈的越底部。第二个影响堆栈中元素位置的因素是 z-index。它的值可以是负数也可以是正数，如果设置为 0 则表示使用常规的呈现层（相对定位），负数表示该元素低于该层，正数表示该元素高于该层。

display 属性还会影响定位和布局，我们将在“显示、可视性和不透明性”小节中说明。在定位时，还会涉及 float 属性，但它和动态页面效果关系不大，因此在此我们就不做介绍了。

只有当定位方式设置为 static（默认定位值）之外的模式时，top、right、bottom、left 以及 z-index 属性才有效。如果将属性设置成负值，那么有可能使一些元素显示在可见页面之外。你还可以基于事件（如用户点击鼠标）移动元素。

fly-in 是动态 Web 页面效果的一种，它使元素看起来就像从文档外边“飞进来”一样。当你需要根据 Web 网页访问者单击鼠标或按下键盘引入一个新主题时（例如在指南中），这是一种不错的方法。

示例 12.4 展示了 3 个元素从左上角飞入的效果。在此使用了一个定时器来创建动画，当每张“幻灯片”到达指定的 x 位置时重置定时器，它们的 top 值比 value 值大（100+value*元素编号，以创建重叠的效果）。这些元素最初在页面上是“隐藏”的，放在左上角，由于设置的元素超过了页面的右下角，因此页面上将出现滚动条。

示例 12.4　元素的定位和“飞入式”动画

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
```

```
<title>Fly-Ins</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<style type="text/css">
div
{
	padding: 0 5px;
}
#nav
{
	background-color: #ccc;
	width: 100px;
	cursor: pointer;
}
#div1
{
	background-color: #00f;
	color: #fff;
	font-size: larger;
	position: absolute;
	width: 400px;
	height: 200px;
	left: -410px;
	top: -400px;
	}
#div2 {
	background-color: #ff0;
	color: #;
	font-size: larger;
	position: relative;
	width: 400px;
	height: 200px;
	left: -410px;
	top: -400px;
	z-index: 4;
	}
#div3 {
	background-color: #f00;
	color: #fff;
	font-size: larger;
	position: fixed;
	width: 400px;
	height: 200px;
	left: -410px;
	top: -400px;
	}
</style>
<script type="text/javascript">
//<![CDATA[
```

```
// 全局占位符
var slides = ["div1","div2","div3"];
var x = 0;
var y = 0;
var currentSlide = 0;

window.onload=function () {
   document.getElementById("nav").onclick=nextSlide;
}

function nextSlide() {
   setTimeout("moveBlock()",1000);
}

function moveBlock() {
   x+=20;
   y+=20;
   var obj = document.getElementById(slides[currentSlide]);
   obj.style.top = x + "px";
   obj.style.left = y + "px";
   if (x < (100 + currentSlide * 60)) {
      setTimeout("moveBlock()", 100);
   } else {
      currentSlide++;
      x = 0; y = 0;
   }
   if (currentSlide >= slides.length) {
     document.getElementById("nav").onclick=null;
   }
}

//]]>
</script>
</head>
<body>
<div id="nav">
<p>Next slide</p>
</div>
<div id="div1">
<p>Blue block that is absolutely positioned.</p>
</div>
<div id="div2">
<p>Yellow block that is relatively positioned, and given a z-index of 4.</p>
</div>
<div id="div3">
<p>Red block that has fixed positioning.</p>s
</div>
</body>
</html>
```

图 12.1 展示了该页面在 Safari 中打开并显示完 3 个盒子之后的效果。你会看到第二个黄色的盒子比前一个盒子移动得更远些，因为它是相对于第一个蓝色盒子进行移动的。另外，你会发现红色盒子被覆盖了，虽然在文档中它是比较晚创建的。这是因为黄色盒子采用了 z-index 定位。而红色和蓝色盒子的 z-index 值都是默认的 0。

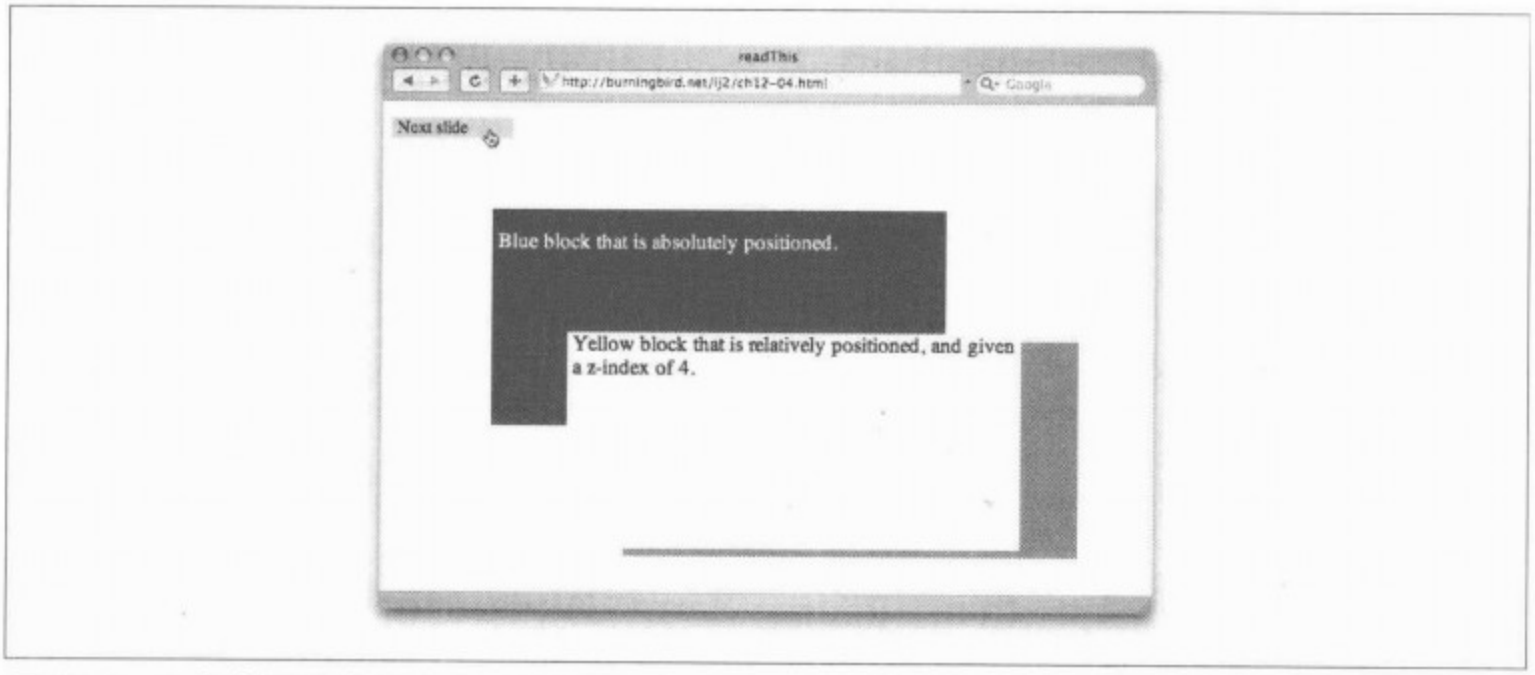


图 12.1　3 个盒子动态地飞入页面

为了使该页面更易于访问，你可以在 div 元素中添加一个超链接，以扮演导航按钮的角色。该链接能够改变打开带有飞入动画的页面的导航方式。另外，你也可以将这 3 个信息块都直接定位在页面上（也就是都可见的形式），然后使用脚本来隐藏它们，当然前提是 JavaScript 启用了。

另外一个常用的与动画有关的动态页面效果是跟踪 Web 页面访问者对元素的操作。该技术称为拖放，我们将在下一小节中介绍。

12.3.2　拖放操作

拖放操作是第一次引入就受到广泛欢迎的动态网页效果。购物车示例引爆了“拖放操作”的流行趋势。

再次引发人们关注“拖放操作”的是 Google Maps，它通过该技术使你能够在一个受限的空间内拖动地图。这是我第一次看到真正有效的拖放操作。我们在后续的章节中将会简单介绍 Google Maps 和与它相关的 API，不过现在，我们还是先看看自己实现的一个十分简单的拖放效果。

提示

Google Maps 如此令人激动的原因是当你在地图上平移时，应用程序实际上将从服务器上获取下一个地图片段，这是一种有时被称为 tiling（贴片）的过程，然后通过缓存机制将其整合到页面上。基于这样的方法，你会感觉从未遇到地图的尽头。它的确做得很棒。

示例 12.5 创建了一个 div 元素，然后将本书中的一个屏幕截图载入到该元素中。你可以通过拖放操作在容器元素内移动该图像。除了演示拖放行为之外，这个小型的应用程序还使用了 overflow 属性，将图像中超出容器的部分修剪掉。这一功能有效地避免了图像溢出已定义的空间。

示例 12.5　Google Maps 效果示范：在容器内拖放一个对象

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>Google Maps Effect</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<style type="text/css">
#div1 {
     overflow: hidden;
     position: relative;
     top: 100px;
     left: 100px;
     border: 5px solid #000;
     width: 400px;
     height: 200px;
}
img {
    border: 1px solid #000;
   }
</style>
<script type="text/javascript">
//<![CDATA[

// 全局变量
var dragObject  = null;
var mouseOffset = null;

// 捕获鼠标相关事件
document.onmousemove = mouseMove;
document.onmouseup   = mouseUp;

// 创建一个鼠标位置
function MousePoint(x,y) {
   this.x = x;
   this.y = y;
}

// 寻找鼠标位置
function mousePosition(evnt){
  var x = parseInt(evnt.clientX);
  var y = parseInt(evnt.clientY);
  return new MousePoint(x,y);
```

```
}

// 获取元素在页面中的位置偏移量
function getMouseOffset(target, evnt){
   var theEvent = evnt ? evnt : window.event;
   var mousePos  = mousePosition(theEvent);

   var x = mousePos.x - target.offsetLeft;
   var y = mousePos.y - target.offsetTop;
   return new MousePoint(x,y);
}

// 停止拖放
function mouseUp(evnt){
   dragObject = null;
}
// 当拖动时捕获鼠标的移动
function mouseMove(evnt){
   if (!dragObject) return;
   var theEvent = evnt ? evnt : window.event;
   var mousePos = mousePosition(theEvent);

   // 如果可拖放，设置新的绝对位置
   if(dragObject){
      dragObject.style.position = 'relative';

      dragObject.style.top      = mousePos.y - mouseOffset.y + "px";
      dragObject.style.left     = mousePos.x - mouseOffset.x + "px";
      return false;
    }
}

// 使对象可拖放
function makeDraggable(item){
   if (item) {
      item = document.getElementById(item);
      item.onmousedown = function(theEvent) {
                         dragObject  = this;
                         mouseOffset = getMouseOffset(this, theEvent);
                         return false; };
   }
}

window.onload=function() {
   makeDraggable("img1");
}

//]]>
</script>
```

```
</head>
<body>
<div id="div1" >
<img id="img1" src="fig0901.png" alt="Figure 9-1" />
</div>
</body>
</html>
```

这是本书迄今为止最复杂的示例，因此我们还是从头讲解一下这里的 JavaScript 代码。

- 在此创建了两个全局对象：dragObject 和 mouseOffset。前者是将要拖放的对象；后者是该对象的偏移量。偏移量是该对象相对于容器的位置，在本例中这个容器就是页面。我们还将捕获针对文档的 mousemove 和 mouseup 事件，然后分别将 mouseMove 和 mouseUp 作为其事件句柄赋给它。
- 接下来是一个名为 MousePoint 的对象。它只是用来封装两个坐标值：x 和 y。创建这样的对象可以使得传两个值更加简单。
- 接下来是名为 mousePosition 的函数。该函数将访问目标对象的 clientX 和 clientY 值，然后返回一个 MousePoint 对象，用来表示该对象相对于客户端窗口（窗口外面的 chrome 部分将被忽略）的相对 x 和 y 坐标。而 parseInt 函数用来确保返回值是数字型。
- 接下来是一个名为 getMouseOffset 函数，它有两个参数，一个是目标对象，另一个是事件。如果事件对象已经通过标准化解决了跨浏览器差异问题，那么该事件触发的鼠标位置将传给刚才介绍的 MousePosition 函数。然后根据对象的 offsetLeft 和 offsetTop 属性对其进行修改。如果我们没有做这样的计算，那么鼠标仍然能够移动对象，但其动作可能会很古怪，被拖动的对象可能会出现在鼠标指针的上面、下面或者边上。经过标准化处理之后，它将用来创建一个标准化后的 MousePoint 对象，该方法将返回这个对象。
- 接下来的函数是 mouseUp，它所做的事情是将 dragObject 设置为 null，以便停止拖放操作。再接下来是 mouseMove 函数，与拖放操作相关的绝大部分内容都在这里。在该函数中，如果没有设置可拖放对象，那么就将退出该函数。否则，将找到一个标准化处理之后的鼠标位置，然后将该对象放在一个相关的位置上，同时设置其 left 和 top 属性（在根据 offset 调整之后）。
- 最后一个函数是 makeDraggable，它用来将传入该函数的对象变成可拖放对象。这意味着需要为该对象的 mousedown 事件添加一个函数，它将为该对象设置一个可拖放对象，并且获得该对象的偏移量。

这看起来有大量的代码，但它实际上要比针对老版本浏览器所需的代码简单多了，因为绝大多数现代浏览器在定位方面共享了相同的属性。另外，Google Maps 还通过 Ajax 添加了一个复杂的元素来实现地图的不断刷新，因此你永远不会遇到地图不够用的现象。不过这个概念稍稍超出了本书的范畴，你可以自己做些研究。

12.4 大小和修剪

你可以通过 6 个 CSS 属性来控制元素的大小。前两个是 width 和 height，它们是最常用的，可以用来设置元素的绝对宽度和高度值。另外 4 个是 min-height、min-width、max-height 和 max-width，它们都是很易于使用的 CSS 属性（特别是在处理图像时），但在实现动态页面效果时并不常用。另外，绝大多数浏览器需要打补丁才能够支持后 4 个属性。

提示

实际上，元素的宽度和高度还受到其他几个属性的影响，包括元素的 border（边框）、margin（边距）、padding（填充）和 content（内容）。它们一起构成了 CSS 中针对块级元素的“盒子模型”，块级元素的意思是在元素之后一定会分行。关于盒子模型可以阅读 W3C 网站上的“Box model”页面（http://www.w3.org/TR/CSS2/box.htm）。

如果元素内容比元素本身大，就需要通过 CSS 的 overflow 属性来处理溢出的情况，你可以将其设置为 visible（显示所有内容，它将溢出元素的边框）、hidden（修剪溢出的内容）、scroll（修剪溢出的内容并提供滚动条）或者 auto（如果某些内容被隐藏，就修剪溢出的内容并提供滚动条）。在示例 12.5 中，overflow 属性设置为 hidden，因此超出 div 元素部分的图像将被修剪掉。

12.4.1 溢出和动态内容

当通过 Ajax 调用或其他事件动态替换元素内容时，填充在元素中的内容将会动态改变。要想确保其内容永远可访问，其中一种方法是将 overflow 属性设置成 auto。如果内容太大，就将提供滚动条。在示例 12.6 中有两个盒子：一个盒子中有大量文本，另一个只有很少的文本。其中，一个盒子允许内容溢出容器，另一个则将对其进行修剪。当你点击 Web 页面时，将切换两个盒子中的内容，因此你会看出它们分别是如何处理溢出的。

示例 12.6　内容修改与溢出设置的影响

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>Overflow</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<style type="text/css">
div { border: 1px solid #ccc; }
#div1 { width: 400px; height: 150px; overflow: visible; }
#div2 { position: absolute; left: 450px; top: 10px; width: 400px; height: 150px;
overflow: auto }
</style>
<script type="text/javascript">
//<![CDATA[
```

```
function switchContent() {
    var div1 = document.getElementById("div1").innerHTML;
    var div2 = document.getElementById("div2").innerHTML;
    document.getElementById("div1").innerHTML = div2;
    document.getElementById("div2").innerHTML = div1;
}

document.onclick=switchContent;

//]]>
</script>
</head>
<body>
<div id="div1">
<p>
One of the first presentation-specific HTML elements was font, and it's also
one of the older HTML elements you still find, all too frequently, in web
pages. It's not surprising that font and text properties were of such interest
in building web  pages. Few changes you can make to an element's style attributes
can have such an effect as changing the text or font properties. </p>
<p>
Notice I say text or font properties. The font has to do with the characters
themselves: their family, size, type, and other elements of the characters'
appearance. The text attributes, though, have more to do with decoration
attached to the text and include text decoration, alignment, and so on. </p>
</div><div id="div2"><p>Smaller item.</p>
</div>
</body>
</html>
```

当载入该页面时，较大的文本块将溢出容器，而较小的文本块中将有很多空白区域。当两个内容被切换之后，第一个盒子中将显示较少的文本，同时将出现大量的空白区域。解决这个问题的方法是改变盒子的尺寸。不幸的是，在实际的案例中，你可能很难确定适应于新内容所需的尺寸。

而第二个盒子将突然在右边出现一个滚动条，通过它可以在内容区域中滚动。与其根据猜测来调整盒子的大小，还不如将 overflow 属性设置为 auto，并触发滚动条。采用这种方法，盒子在页面上的显示是相对稳定的，其他元素也不会受到它的影响，带有较多空白区域的大盒子虽然不太好，但其内容仍然是可访问的。图 12.2 中展示了第一次载入时的页面效果；图 12.3 中展示了两个盒子中的内容互换之后的效果。

还有一种处理内容改变的方法，那就是使用只读属性 offsetWidth 和 offsetHeight 来确认内容的实际大小，以修改盒子的大小。但是在使用这些属性时，在不同浏览器中会产生一些差异。IE 会将边框和填充部分算在盒子大小内，而 Mozilla/Firefox 计算的大小则只包含实际的内容。

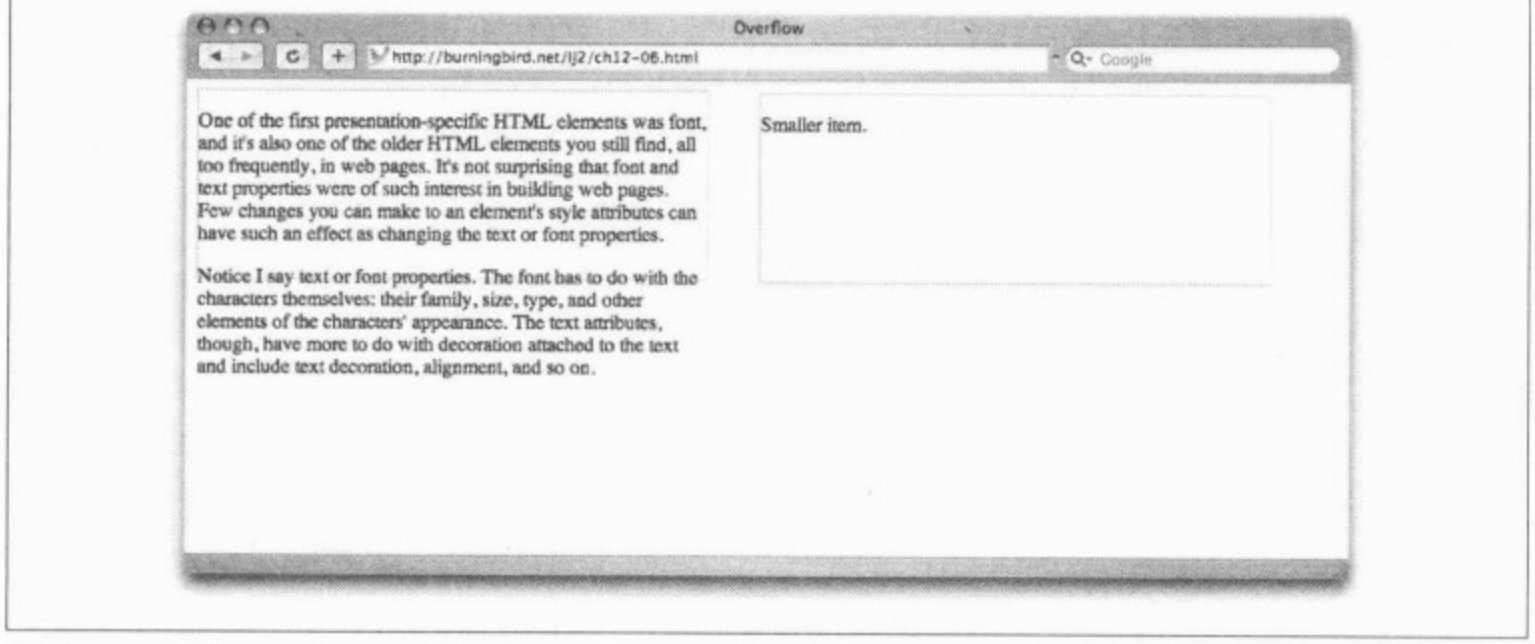


图 12.2　当在 overflow 属性设置为 visible 的容器中显示较多内容时

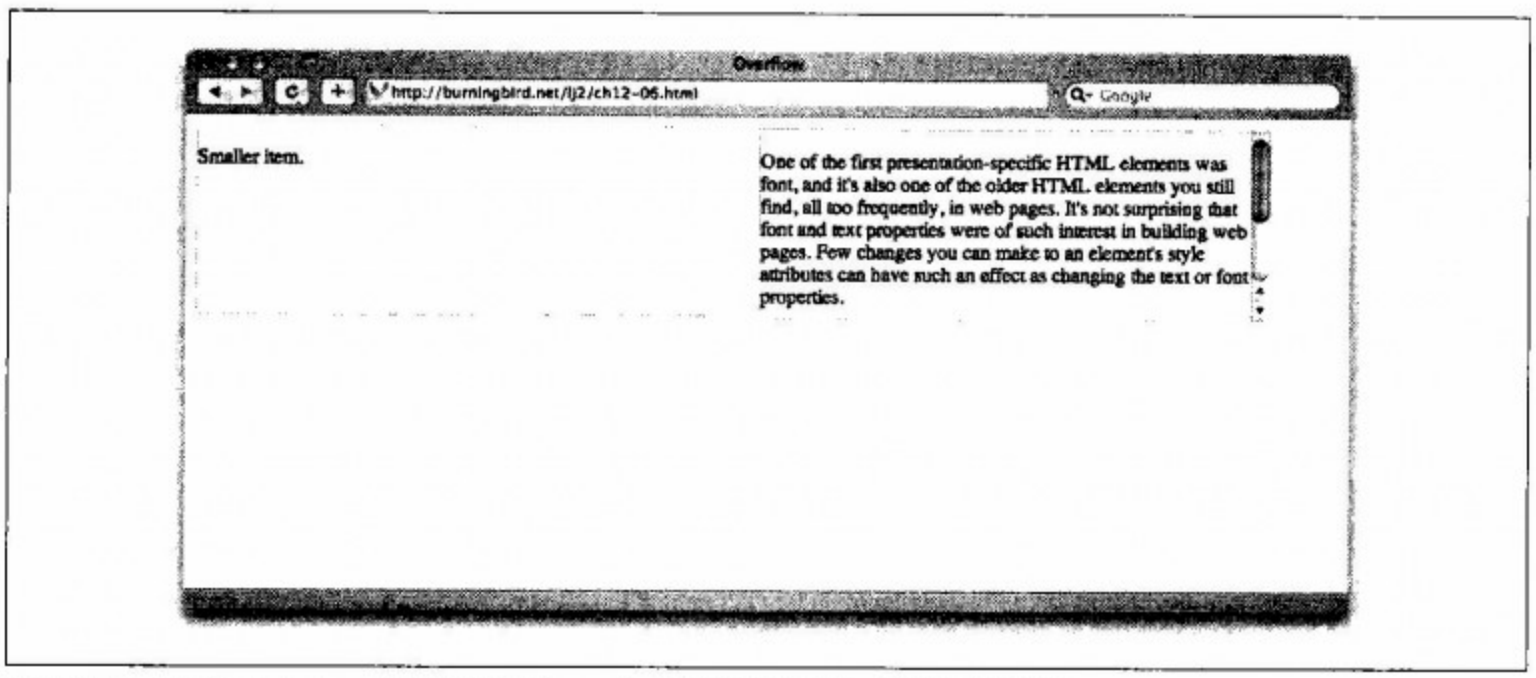


图 12.3　当在 overflow 属性设置为 auto 的容器中显示较多内容时

提示

你也可以使用前面定义的 getStyle 方法来访问元素的计算后宽度及高度值，只需将前面的 backgroundColor 改成 width 和 height 就行。

虽然 width 和 height 属性能够控制元素的大小，但它们无法控制元素中可见的内容。你可以通过元素相关的修剪矩形（clipping rectangle）来控制。

12.4.2　修剪矩形

CSS 中的 clip 属性可以指定一个形状以及该形状的大小。现在，该属性支持的形状只有矩形，可以用 rect 来指定，并定义其 4 个尺寸值：顶端、右边、底部和左边。

```
clip: rect(topval, rightval, bottomval, leftval);
```

这个修剪区域能够限制实际显示的元素内容。同时它要求将 position 属性设置成 absolute。

如果一个元素的宽度为 200 像素，长度为 300 像素，修剪区域为 rect(0px,200px,300px,0px)，那么将不会做任何修剪。当然，它取决于该元素是否拥有边框，以及其他可能影响高度和宽度值的其他设置。如果修剪区域为 rect(10px,190px,290px,10px)，那么将在每一边修剪掉 10 个像素。注意，要做修剪时，对于顶部和左边应该增大其值，对于底部和右边应该减小其值。

从动态 Web 页面的角度看，你可以通过修剪来创建各种滚动效果，无论该元素是否为动画。它可以用来创建新的“折叠效果”，它在 Ajax 应用程序中已经越来越常见，当你点击标题时，就会看到类似于“收起下面内容”的效果。

示例 12.7 中展示了修剪功能的简单应用，创建了一个可下拉的动画项。在文档任何地方点击一下鼠标，就将根据当前状态打开或收起该项目。该示例使用了一个定时器来完成动画效果；如果你对动画效果不感兴趣，也可以不使用定时器，直接完成打开或收起操作。

示例 12.7　使用定时器和修剪功能创建下拉动画

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>Clipping</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<style type="text/css">
#data1    {
          position: absolute;
          top: 100px; left: 100px;
          padding: 0;
          width: 200px;
          height: 200px;
          background-color: #ff0;
          clip: rect(0px,200px,200px,0px);
          }

</style>
<script type="text/javascript">
//<![CDATA[

// 全局变量

var bottom = 200;
var clipped = false;

document.onclick=testItem;

function testItem() {
```

```
  if (clipped) {
      unClipItem();
   } else {
      clipItem();
   }
}

function clipItem() {
   bottom-=20;
   var clip = "rect(0px,200px," + bottom + "px,0px)";
   var obj = document.getElementById("data1");
   obj.style.clip = clip;
   if (bottom == 40) {
      clipped=true;
   } else {
      setTimeout("clipItem()",100);
   }
}

function unClipItem() {
   bottom+=20;
   var clip = "rect(0px,200px," + bottom + "px,0px)";
   var obj = document.getElementById("data1");
   obj.style.clip=clip;
   if (bottom == 200) {
      clipped=false;
   } else {
      setTimeout("unClipItem()",100);
   }
}

//]]>
</script>
</head>
<body>
<div id="data1">
This is the text contained within the div block.
</div>
</body>
</html>
```

当第一次打开该页面时，上面将显示一个较长的、带有一些文字的黄色盒子。在 Web 页面上单击鼠标左键，将对盒子进行“修剪”，使其大小刚好容下文本信息。如果再次在 Web 页面上单击鼠标左键，则又将“打开”整个盒子。

注意，在检查状态时并不是直接从样式属性中获取 clipping 属性值，而是使用了全局变量。在第 13 章中，我们将展示如何创建一个对象并将所有函数和变量放到这个对象中，这样就可以避免到处都是全局变量。

12.5 显示、可视性和不透明性

对于 Web 页面元素而言，还有一件有趣的事，那就是它们可以变成完全透明或不可见的，但仍然影响页面的布局。这是因为在 CSS 中“不可见/透明”和“显示/不显示”不是同一个概念。

如果想隐藏一个元素，只需将其 visibility 属性设置为 hidden，将其设置为 visible 就能使其可见。你也可以将该属性设置为 inherit，那么该元素将从包含它的元素中继承该属性的设置值。你也可以将元素的 opacity 属性设置成完全透明，这样也能够隐藏该元素。不过，它和 visibility 属性一样，该元素仍然会存在于页面流中。

如果某个元素的 display 属性被设置成 none，那么它也将被隐藏；不过，该元素对页面布局的影响也将去除。要使其可见，有多种选择。如果你想让某元素像块级元素那样显示（在元素之后添加一个新行），可以将 display 属性设置为 block。如果你不想像块级元素那样显示，可以将 display 属性设置为 inline，那么它将显示在适当的位置，并且不会添加新行。

另外，你可以使用几个 HTML 元素的默认显示特性来显示元素，包括 inline-block、table、table-cell、list-item、compact、run-in 和 others。它们是相当强大的属性，你可以使用它们直到对修改结果感到满意。

12.5.1 实现正确效果的正确工具

有这么多隐藏和显示元素的方法，你可以使用哪种方法实现哪种效果呢？

如果你对元素应用的是绝对定位，并且想根据如鼠标点击或表单提交等事件决定是隐藏还是显示，那么应该使用 visibility 属性。它十分易于使用，而且绝对定位的元素和页面流无关。

如果隐藏的内容在显示时要将后续的页面元素推到后面去，例如点击一个填充在表单上的、可收缩的选项列表，我们可以使用 display 属性，使其值在 none 和 block 之间切换。

如果你想创建褪色效果，或者想不再强调某个页面元素，可以使用 opacity 属性。你可以令其最终完全透明，不过通常是完成褪色动画之后。使用 opacity 可以强调或提供可视的信息，也可以使用 opacity 实现信号过渡。

提示

用来实现信息提示的可视效果中也会添加一些文字性元素，这样那些使用非可视化的浏览器或可视化功能受限的浏览器的人，也能够收到相同的通知。永远都不要彻底依赖可视效果来向用户提供反馈。

现在我们来看看更加生动的操作。

12.5.2 即时信息

我以前看过的一些很棒的网站，能够在 Web 页面访问者需要时提供某种形式的帮助信息。即使是要求用户输入姓名时，你或许也想向用户提供一些关于隐私控制以及这些数据如何使用的相关解释。

你可以在输入域边上放一个链接，然后设置其 title 属性，这样就能提供一个“工具提示”式的帮助信息，不过这样做能够包含的信息量通常是受限的。你也可以弹出一个显示相关信息的对话框，如果信息比较长、比较详细，并且带有选项描述时，这样做会更好一些。不过对于介于这两者之间的情况（也就是信息不多也不少），可能最好的方法是直接在页面上显示该信息。

在很大程度上，表单将占据绝大部分的页面空间，大量的文字会使页面看起来很混乱。将这些信息直接放到页面上是一种方法，但它需要通过一些事件来触发。

这是你想创建的、最有用的动态 Web 页面效果，同时也是最简单的。示例 12.8 中列出的是该页面的代码，它包括两个表单元素，每个都带有一个隐藏的帮助信息块。在脚本中，当针对该元素的标签被点击时，如果已经显示了一些帮助项，那么可见的帮助信息就将隐藏，同时将显示新的帮助信息。

示例 12.8　使用隐藏的帮助域

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>Just-in-Time Help</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<style type="text/css">
.help { position: absolute;
        left: 300px;
        top: 20px;
        visibility: hidden;
        width: 100px;
        padding: 10px;
        border: 1px solid #ff0000;
      }
form {
   margin: 20px;
   background-color: #dfe1cb;
    padding: 20px;
    width: 200px;
}
label {cursor : help}

</style>
```

```
<script type="text/javascript">
//<![CDATA[

window.onload=function() {
   var labels = document.getElementsByTagName("label");
   for (var i = 0; i < labels.length; i++)
      labels[i].onclick=showHelp;
}

function showHelp(evnt) {
   var theEvent = evnt ? evnt : window.event;
   var theSrc = theEvent.target ? theEvent.target : theEvent.srcElement;

   // 首先隐藏所有帮助信息
   var item1 = document.getElementById("item1");
   var item2 = document.getElementById("item2");

   // 有选择性地显示正确的帮助信息
   item1.style.visibility="hidden";
   item2.style.visibility="hidden";

   if (theSrc.id == "label1")
      item1.style.visibility="visible";
   else
      item2.style.visibility="visible";
}

//]]>
</script>
</head>
<body>
<form action="">
<p>
<label id="label1">Item One</label>
<input type="text" /><br /><br />
<label id="label2">Item Two</label>
<input type="text" />
</p>
</form>
<div id="item1" class="help">
<p>This is the help for the first item. It only shows when you click on the
label for the item.</p>
</div>
<div id="item2" class="help">
<p>This is the help for the second item. It only shows when you click on the
label for the item.</p>
```

```
</div>
</body>
</html>
```

这里还添加了一些 CSS 作料，使该页面的“味道”更好些。表单元素设置了背景颜色，帮助信息块添加了红色边框，当鼠标指针移到每一项的输入标签上时，鼠标指针将改为帮助图标。它通常是由一个箭头加上小小的问号组成的，有的浏览器中只有单独的问号。这也是为 Web 页面读者提供暗示的一种最廉价的方法。图 12.4 展示了这个隐藏的帮助系统的执行情况。

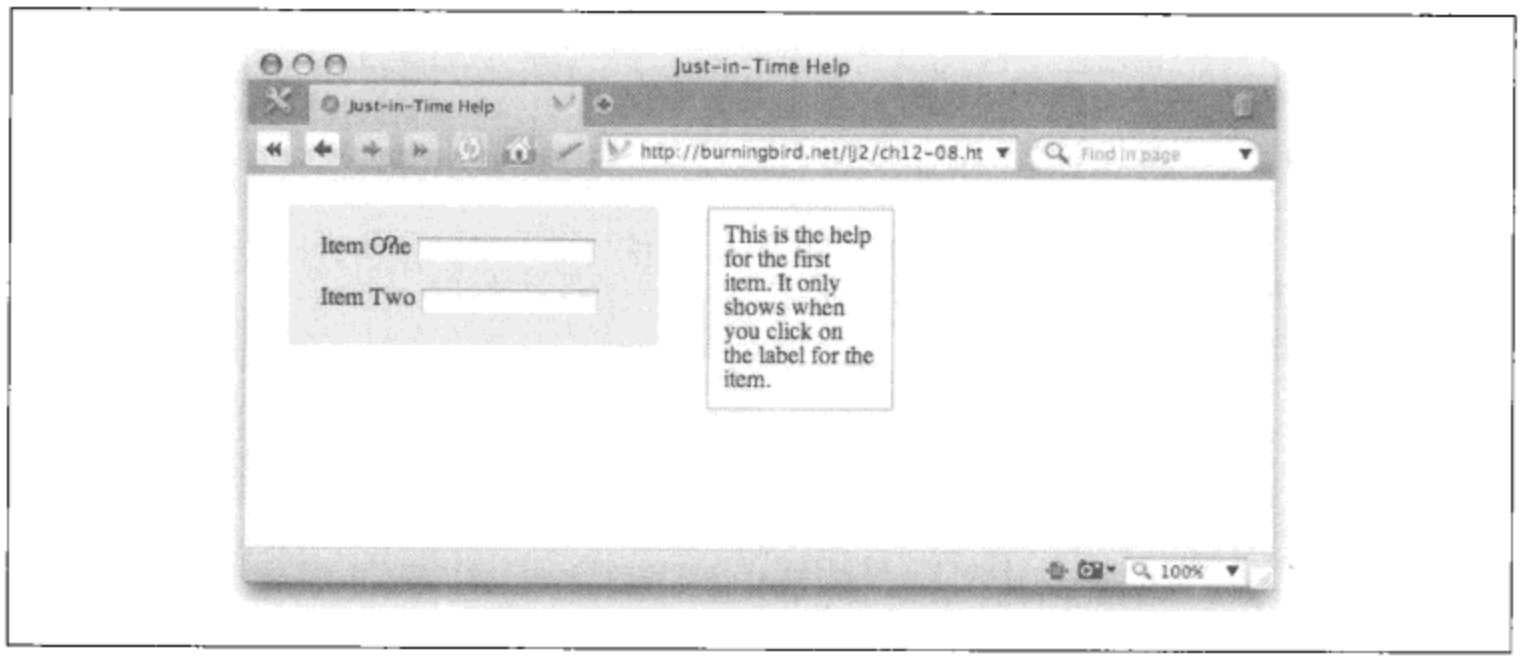


图 12.4　使用 visibility 属性实现联机帮助

12.6　再探 DOM：可折叠表单、查询选择器和类名

在本书前面的章节中，我们了解了在 Web 页面中查询元素的几种不同方法。我们最常用的方法是 getElementById，它通过一个标识符来获取特定的页面元素。我们还使用了 getElementsByTagName 方法，它用来获得拥有指定标签名的所有元素。我想最好还能够有一些通过 CSS 类名获取元素的方法，甚至可以使用 CSS 选择器，例如：

```
div > p:first-child
```

这种类型的功能的确存在，不过它仍然很新，在不同的浏览器中也有一些差异。Firefox、Opera 和 Safari 是通过 document 对象的 getElementsByClassName 方法来实现的，它最初是在为 HTML 5.0 提供支持时引入的，它将返回拥有指定类的所有元素：

```
var elems = document.getElementsByClassName('elements');
```

IE 8.0（除了 Safari/WebKit 之外）实现了另一种方法，就是众所周知的查询选择器（query selector）。它是基于现在还是草案的 W3C 规范（http://www.w3.org/TR/selectors-api/）实现的，它允许你查询与指定选择器模式匹配的所有元素，例如要实现之前用

getElementsByClassName 方法实现的功能，可以使用以下代码：

```
var elems = document.querySelectorAll('.elements');
```

为了确保类查询选择器功能能够运行在不同浏览器上，可以检查是否存在 querySelectorAll 方法，如果没有找到，则使用 getElementsByClassName 方法：

```
if (document.querySelectorAll) {
    elems = document.querySelectorAll('.elements');
 } else if (document.getElementsByClassName) {
     elems = document.getElementsByClassName('elements');
}
```

现在，如果浏览器支持 document.querySelectorAll 方法，则用它来构建 nodeList 对象，里面存储的是所有匹配的页面元素。否则就使用 document 对象的 getElementsByClassName 方法来获取相同的节点列表。

如果你想处理页面中特定类型的所有元素，而不是处理与特定 CSS 选择器或类匹配的元素，那么基于 CSS 选择器获取一个节点列表是很有帮助的。在实现一个可折叠的表单时，这种方法是十分有效的。

要将表单功能分拆到多个页面上是痛苦的，但在一个页面中显示过多的表单元素也是难以理解的。另外，in-place editing（即时编辑）功能越来越流行：对于拥有数据的用户，数据小节的标题是可用的，点击这些标题时将打开一个表单或输入域，用来修改该小节的数据。

对于这些场景，很适合采用 JavaScript 内建的 accordion（可折叠）功能。它是一种页面选择功能，只显示活动的内容，当显示这些隐藏的信息时将使页面中其他内容向后移动。Google、Flickr 等很多公司都使用了这种可折叠的内容。你可以将其看做提高可访问性的最简单的方法，而且也不会让人感到惊讶。如果没有启用 JavaScript，与标题相关的事件处理将正常显示不活动的内容，以及那些未打开的表单。你可以添加一些菜单项，打开一个独立的页面来显示表单，或者显示 noscript 标签中的内容。

示例 12.9 是本章最后一个示例，展示了带折叠功能的表单。在本例中，它是存放在堆栈中的表单元素集，如果 JavaScript 未启用，它们都将被显示。在此使用 div 元素标识了可点击的标签以及可折叠的表单小节。当该页面载入时，不再访问页面中的每个 div 元素并且添加 onclick 事件或将其 display 属性设置为 none，而是使用前面提到的类查询选择器功能访问此类元素或此类标签的 div 元素，不是收缩它就是添加事件句柄。

示例 12.9　实现可折叠的表单

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>Accordion</title>
```

```
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<style type="text/css">

.label
{
   background-color: #000033;
   width: 400px;
   padding: 10px;
   margin: 0 20px;
   color: #fff;
   text-align: center;
   border-bottom: 1px solid #ffffff;
   cursor: pointer;
}
.elements
{
   background-color: #ccd9ff;
   margin: 0 20px;
   padding: 10px;
   width: 400px;
   display: block;
}
</style>
<script type="text/javascript">
//<![CDATA[

window.onload=function() {
   var elems = null;
   var labels = null;
   if (document.querySelectorAll) {
      elems = document.querySelectorAll('div.elements');
      labels = document.querySelectorAll('div.label');
   } else if (document.getElementsByClassName) {
       elems = document.getElementsByClassName('elements');
       labels = document.getElementsByClassName('label');
   }
   if (elems) {
      for (var i = 0; i < elems.length; i++) {
        elems[i].style.display="none";
      }
      for (var i = 0; i < labels.length; i++) {
        labels[i].onclick=showBlock;
      }
   }

}

function showBlock(evnt) {
```

```
    var theEvent = evnt ? evnt : window.event;
    var theSrc = theEvent.target ? theEvent.target : theEvent.srcElement;
    var itemId = "elements" + theSrc.id.substr(5,1);
    var item = document.getElementById(itemId);
    if (item.style.display=='none') {
        item.style.display='block';
    } else {
        item.style.display='none';
    }
}

//]]>
</script>
</head>
<body>
<form action="">
<div class="label" id="label1">
Name
</div>
<div class="elements" id="elements1">
<label>First Name:</label><br /><input type="text" name="firstname" /> <br /> <br
/>
<label>Last Name:</label><br /><input type="text" name="lastname" /> <br /> <br
/>
</div>
<div class="label" id="label2">
Address
</div>
<div class="elements" id="elements2">
<label>Street Address:</label><br /><input type="text" name="street" /> <br /> <br
/>
<label>City:</label><br /><input type="text" name="city" /><br /><br />
<label>State:</label><br /><input type="text" name="state" /><br /> <br />
</div>
</form>
</body>
</html>
```

在示例 12.9 所示的应用程序中，如果浏览器既不支持 querySelectorAll 方法，也不支持 getElementsByClassName 方法，那么将适度地选择退而求其次的做法：表单小节不再折叠，整个页面的效果就像脚本被禁用一样。在 IE 6.0 中该应用程序就将显示这样的效果。

点击“标签”时将显示相关的小节，它是根据标签和可折叠小节所用的标识符完成的。你还可以添加更多的标签和相关的小节，JavaScript 代码并不需要修改，只要你使用相同的类名，然后维护好标签和元素之间的标识符映射关系即可。图 12.5 展示了该页面在 IE 8.0 中载入的效果，图 12.6 则显示了当其中一个小节展开时的效果。

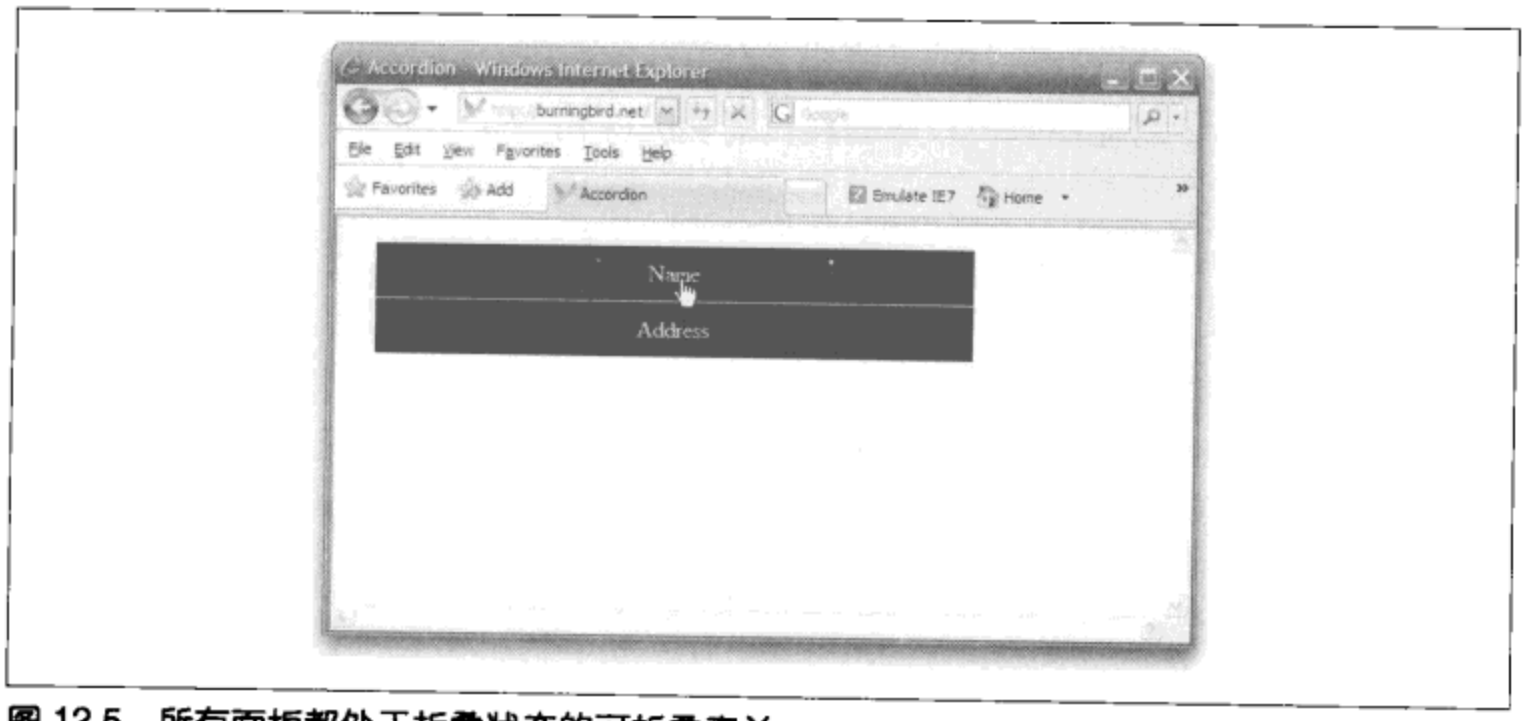


图 12.5　所有面板都处于折叠状态的可折叠表单

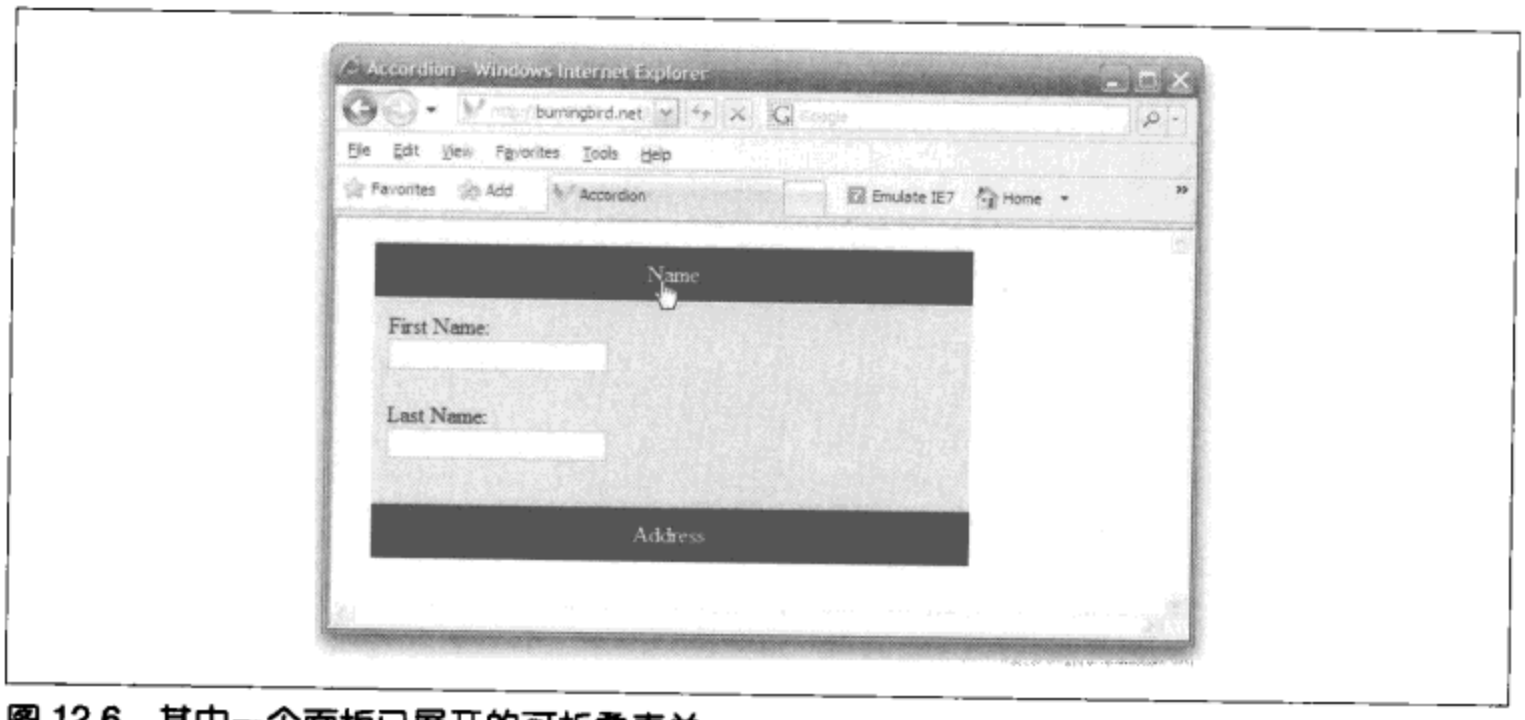


图 12.6　其中一个面板已展开的可折叠表单

因为能够在表单中添加新面板，所以你可能会想把这些功能打包成一组可复用的 JavaScript 函数。实际上，为了进一步简化应用程序，你可以将所有这些功能打包到一个 accordion 对象中。这也就将我们引到第 13 章中，在那里将介绍自定义的 JavaScript 对象。

12.7　知识测验

1. 当你在 JavaScript 程序中尝试使用 obj.style.color 方法访问一个元素的文本颜色时，却没有返回任何值。你知道这个字体颜色是通过样式表设置的。为什么没有返回任何值？你要如何修改程序才能够获得该值？写一段程序代码，访问名为“name”的元素的字体颜色，以演示你的方法。

2. 对于 div 中的文本，如何将其显示样式改成 14px 大、红色、行高 16px?

3. 如果之前的修改未生效，可能是什么原因导致的?

4. 让一个 HTML 元素块消失有哪三种方法？写一段程序代码，将名为 name 的元素彻底从页面布局中删除掉，以演示你的方法。

5. 如果拖放技术不是有效的购物车技术，那么对于这种类型的服务，你可以使用哪种动态 Web 页面效果?

12.8 测验答案

1. 你可以使用 getComputedStyle 方法或 currentStyle 方法，这取决于你使用的是哪种浏览器。另外一种方法是在页面载入时设置样式属性。

```
var nameDiv = document.getElementById("name");
var nameColor;
if (nameDiv.currentStyle) {
   nameColor = nameDiv.currentStyle['color'];
} else if (window.getComputedStyle) {
   nameColor =
document.defaultView.getComputedStyle(nameDiv,null).getPropertyVa
lue('color');
}
```

2. 其解决方案如下所示：

```
divElement.style.font="14px/16px";
divElement.style.color="#ff0000";
```

3. 如果这个在 div 元素中的文本实际上是属于另一个元素的，如一个段落元素，并且它有不同的样式设置，那么段落元素的样式设置将重载外面这个 div 元素的新样式设置。

4. 要让一个 div 元素消失，你可以将其 width 或 height 属性设置为 0，或者将整个元素完全修剪掉。你还可以将该元素的 visibility 属性设置为 hidden，或者将其 display 属性设置为 none。最后你还可以将其移到页面外，到最顶部或最左边。如果要从页面布局中删除该元素，需要修改 display 属性：

```
var nameDiv = document.getElementById("name");
name.style.display="none";
```

5. 除了使用拖放功能，还可以为显示“Buy me!”信息的链接或图像添加一个鼠标的 click 事件句柄，当这个 click 事件句柄被触发时，在应用程序中添加一些代码，以便自动将所选项添加到购物车中。

第 13 章

创建自定义 JavaScript 对象

在前面的章节中，我们介绍了 JavaScript 语言内建的对象，以及浏览器文档对象模型（DOM）。本章将关注如何创建自定义的 JavaScript 对象，并在其他的应用程序中复用。

13.1 JavaScript 对象和原型

JavaScript 对象通常有一个构造函数，以及相应的方法和属性。此外，JavaScript 中的所有对象都继承了 JavaScript 标准 Object 对象的功能。

Object 对象本身并没有什么特别之处，主要是提供了创建新对象的能力。与其他编程语言不同，JavaScript 对象并不遵循传统的面向对象的继承和类的原则，相反，JavaScript 对象功能是源于 JavaScript 中的一个特殊概念：原型（prototype）。

13.1.1 原型

在 Java 或 C++编程语言中，你可以创建一个从其他类中继承的类，这样它就将继承其父类的功能，并允许覆盖父类的某些功能。

而 JavaScript 中通过 Object 对象提供了一个构造函数，从而允许你创建新的对象。这个对象的构造函数能够为对象分配内存，并包含了所有属性。Object 对象还提供了一个名为 prototype 的属性，通过它可以扩展任意对象，包括内建对象，如 String 或 Number。prototype 属性也可以为对象创建新的方法和属性，而这些方法和属性并不是从类中继承的：

```
Number.prototype.add=function(val1,val2) { return val1 + val2; };
var num = new Number();
num.prototype.add = function(val1),
var sum = num.add(8,3);     // 结果为 11
```

在前面的章节中，有各种各样 JavaScript 动态特性的示例，例如如何在运行时创建新的方法和属性。通过 prototype，为对象实例和对象添加新的属性或方法存在着少量差别。

下列的代码与之前的代码不一样，因为在下面的实例中，add 方法只适用于对象实例，而不是对象的所有实例：

```
var num = new Number();
num.add=function(val1,val2) { return val1 + val2; };
var sum = num.add(8,3);
```

当通过 prototype 扩展对象时，扩展的方法或属性将适用于对象的所有实例。当扩展对象实例的时候，新的方法和属性只适用于该对象实例。

下面的示例可以很好地解释如何通过原型来扩展对象。示例 13.1 中介绍了如何使用 Object 的 prototype 属性为内建的 String 对象添加新的方法 trim。trim 方法的功能是去除字符串前后的空格。

示例 13.1　JavaScript 对象的创建和原型

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>String trim</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript">
//<![CDATA[

String.prototype.trim = function() {
  return (this.replace(/^[\s\xA0]+/, "").replace(/[\s\xA0]+$/, ""));
}

window.onload=function() {
   var sObj = new String("  This is the string  ");
   document.writeln("--" + sObj + "--" + "<br />");

   var sTxt = sObj.trim();
   document.writeln("--" + sTxt + "--");
}
//]]>
</script>
</head>
<body>
<p>some content</p>
</body>
</html>
```

该示例的输出为：

```
-- This is the string --
--This is the string--
```

尽管浏览器在显示页面时会去掉重复的空格，但是在第一个字符串的前后还是会各有

一个空格，而第二个字符串在调用 String 对象的新方法 trim 之后，前后的空格就将被去掉。

基于 prototype 属性，现在该页面中的任何 String 对象不仅可以调用 String 对象中的方法和属性，还可以调用新的 trim 函数。这时你不需要额外创建一个新的对象，而只是扩展了一下现有对象的功能。这也正是基于类和基于原型系统的一大差异。

如果不使用原型，那么也可以根据前面的章节中介绍的方法创建静态 trim 函数：

```
var str = " this is a string ";
str.trim = function() ...
```

不过，现在只有 str 实例才能访问该函数，如果希望扩展的是 String 对象，那么就需要使用 prototype 属性。JavaScript 的每个对象，包括自己创建的对象，都拥有 prototype 属性，可以用来扩展对象。

当访问该方法时，prototype 属性是如何工作的呢？当在特定对象上调用特定方法时，JavaScript 引擎首先将从最初的对象实现中寻找相应的属性/方法。如果没找到，就会在 prototype 集合中寻找相应的属性/方法。只有当引擎无法在全局对象实现或 prototype 集合中找到该属性或方法时，才会寻找在程序中添加到变量上的方法。

警告

当然，对现有对象进行扩展是危险的，因为新版本的 JavaScript 可能会为该对象添加同名但行为却不一样的方法，这个时候，就可能会破坏现有的应用程序。

因为 JavaScript 应用程序的功能是逐渐补充的，所以有些时候希望将代码封装成可复用的组件。下一小节将介绍如何创建自定义的 JavaScript 对象，以及可复用的 JavaScript 程序库。

提示

ECMAScript 即将推出的新版本，可能会命名为 3.1 或 4.0，它将为 JavaScript 对象提供一个新的方法——Object.freeze。根据我的理解，该新方法将“冻结”对象，也就是不允许通过 prototype 属性为该对象进行扩展，也就是提供了不可修改的类，从而使 JavaScript 更接近于基于类的编程语言。

13.2 创建自定义 JavaScript 对象

在本书的剩余部分中，我将介绍在 Ajax 应用程序中如何利用 JavaScript，在那里你将看到对 JavaScript 创建对象的改进是多么重要。在那时，你会发现第三方 Ajax 程序库更像是用其他编程语言开发的，而不是 JavaScript。事实上，许多第三方 Ajax 程序库的功能包含了一些 JavaScript 语言和其他编程语言的特征，同时也有其各自的优缺点。

现在 JavaScript 程序库的一大优点是提供了一些便捷的操作，如访问页面元素。在 Ajax 应用程序中，如果客户端和服务器端都采用相似的编程语言特性进行开发，就可以简化操作。

它也有一些缺点，主要是这些特征并不是 JavaScript 语言的特性，所以对于 JavaScript 开发人员来说，这样的代码就将显得难以阅读，且不易使用，特别是对那些不熟悉最新语言特性的人来说。

提示

关于是否应该创建强大但不易阅读的组件，Dan Webb 有一篇文章“Painless JavaScript Using Prototype”（http://www.sitepoint.com/ article/painless-javascript-prototype）值得一读。

关于创建对象的主题，我们还是回到老方法上，也就是 JavaScript 功能核心的函数。

13.2.1 深入函数

在近 10 年中，当创建 JavaScript 自定义对象时，人们通常会使用 JavaScript 函数。当然与普通函数相比，这里使用的函数略有些不同，例如函数的写法、私有/公有属性的定义甚至包括属性的封装方法都可能不一样。然而，如果要创建 JavaScript 的自定义对象，就必须从函数开始。

提示

在面向对象开发中，私有属性和方法是只能在对象内部访问的属性和方法。本对象以外的对象，只能访问公有的属性和方法。本章稍后将介绍更多相关细节。

在示例 13.2 中，我们创建了一个简单的对象 Tune，该对象只有一个参数，它是将赋给 Tune 对象的 title 属性的“歌曲标题”。在该对象中还有一个数组，通过两个方法可以访问该数组：addPerformer（一个字符串参数）方法和 listPerformers（不带参数）方法。

示例 13.2 创建自定义对象

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>Tune Object</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript">
//<![CDATA[

var Tune = function(title) {
   this.title = title;
   var performedBy = new Array();
   this.addPerformer = function (performer) {
```

```
      var i = performedBy.length;
      performedBy[i] = performer;
   }
   this.listPerformers = function() {
      var singers = "";
      for (var i = 0; i < performedBy.length; i++ ) {
         singers += performedBy[i] + " ";
      }
      return singers;
   }
}

window.onload=function () {
   var song = new Tune("Hello");
   song.addPerformer("Me");
   song.addPerformer("You");
   song.addPerformer("Us");
   document.writeln("Song is " + song.title);
   document.writeln("Performed by " + song.listPerformers());
}
//]]>
</script>
</head>
<body>
<p>some content</p>
</body>
</html>
```

在该页面中，我们通过 Object 构造函数创建了 Tune 对象实例，并传入一个字符串参数“Hello”。接着调用了 3 次 adPerformer 方法，分别传入 3 个参数：“Me”、“You”和“Us”。接着调用 listPerformers 方法，遍历 performers 数组，根据数组内容拼装并返回该字符串。该应用程序的功能是将歌曲标题和表演者输出到页面上。

如果我们对其做更深入的分析，会发现这段 JavaScript 代码创建了与对象 Tune 同名的函数。在前面的章节中曾经说过，JavaScript 的所有函数都是对象，所以创建函数，也就是创建了自定义对象。

该函数内部是两个属性和两个方法。在本示例中，实现这些方法的代码是对象声明的一部分。尽管 JavaScript 并不强制要求在对象声明中实现方法，然而这是编写代码的一种好习惯，能够提高代码的可维护性，并减少应用程序中全局对象的个数。当然，你也可以在对象外面创建函数，并将函数名称赋值于对象成员，例如：

```
function externalAddPerformer(performer) {
   ...
}
...
this.addPerformer = externalAddPerformer;
```

创建自定义对象的另一种方法是，创建对象实例，并使用对象的 prototype 为属性和方法进行赋值。示例 13.3 就将用这种 prototype 的方法创建 Tune 对象。

示例 13.3　使用 prototype 创建自定义对象

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>Tune Object</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript">
//<![CDATA[

function Tune (title) {
   this.title = title;
}
function printTitle() {
   document.writeln(this.title + "<br />");
}

window.onload=function() {
   Tune.prototype.print = printTitle;

   var oneTune = new Tune("One Title");
   oneTune.print();

   var anotherTune = new Tune("Another Title");
   anotherTune.print();
}

//]]>
</script>
</head>
<body>
<p>some content</p>
</body>
</html>
```

JavaScript 中的所有新对象都是从 Object 继承的，所以这些对象都拥有 prototype 属性，包括自定义对象 Tune。该示例不是为对象直接添加方法，而是创建一个用来输出歌曲标题的全局函数，然后通过 Tune 对象的 prototype 属性将其添加到 Tune 对象中。

prototype 的使用很简单，而且广为流行。然而，使用 prototype 属性也存在一定的风险，你的代码可能覆盖了现有的方法或属性，或者其他应用程序可能覆盖了你的实现。流行的 Ajax 框架 Prototype.js 就大量采用了 prototype 属性来扩展 JavaScript 功能，如 Array 对象以及 Object 对象。使用 Prototype.js 的一个问题是，该框架可能不能够与其他 JavaScript 程序库一起使用，因为该框架大量扩展了标准的 JavaScript 对象，可能会对其他 JavaScript

程序库造成一定的影响。

提示

Prototype.js 框架请参见 http://www.prototypejs.org。

这些示例中使用了 this 关键字为对象的 title 属性赋值。第一个示例还用于为方法赋值，但没有为 song 数组使用同样的赋值方式。这种用法突显出 JavaScript 对象的公有和私有成员的区别。

13.2.2 公有和私有属性

示例 13.2 和示例 13.3 中使用 this 关键字在对象创建的时候传入参数 title，为属性 title 赋值。这里 this 关键字是对父对象的引用，也就是新创建对象的实例。在示例中，this 起到的作用是创建一个公有属性，可允许对象以外的访问，例如在页面中输出歌曲标题：

```
document.writeln("Song is " + song.title);
```

在第一个示例中，this 关键字的使用也涉及两个方法：addPerformer 和 listPerformers。performers 数组并不是通过 this 关键字直接赋值的，而是由 var 关键字所创建的变量。在此使用的是 var 而不是 this，因此所创建的这个变量是对象的私有成员，只限于对象内部（包括对象内的方法）访问，而不允许对象外部访问。

为什么使用私有变量而不是公有变量？主要的原因是为了通过数据隐藏来保护数据，不允许应用程序的直接访问。

在某些时候，设计人员并不希望开发人员可以直接访问对象数据，因为这样做的结果往往会导致对象不可使用或破坏了对象等负面影响。通常情况下的做法是，提供一些获取或修改该数据的方法，而不直接暴露该变量。JavaScript 中数据隐藏的方法就是，使用 var 关键字创建私有变量，而不是 this 关键字的公有变量。

ECMAScript 的新版本中将提供 getter 和 setter 方法，从而使管理 JavaScript 对象的私有数据更加容易。

13.2.3 getter 和 setter

getter 和 setter 将加入到 ECMAScript 的下一个正式发布中，一部分浏览器也将支持该新发布。getter 和 setter 是获取或修改对象中数据的一种方式。一般在对象定义后，就在对象中添加 getter 和 setter 的功能。

示例 13.4 与示例 13.3 类似，它们之间唯一的区别是在示例 13.4 中不是将标题作为参数传入对象构造函数中，而是使用 getter 和 setter 管理数据，并在对象定义后添加这些 getter 和 setter 方法。

示例 13.4　介绍 getter 和 setter 的用法

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>Getters and Setters</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript">
//<![CDATA[

// Tune是一个基本的壳对象
function Tune () {

}

// 方法
function printTitle() {
   document.writeln(this.title + "<br />");
}

window.onload=function() {

   // 通过prototype属性扩展对象
   var t = Tune.prototype;

   // getter和setter
   t.__defineGetter__("title", function() { return "Title is " + this.myTitle; });
   t.__defineSetter__("title", function(tt) { this.myTitle = tt; });

   t.print = printTitle;

   var oneTune = new Tune;
   oneTune.title = "One Title";
   oneTune.print();

   var anotherTune = new Tune;
   anotherTune.title="Another Title";
   anotherTune.print();
}

//]]>
</script>
</head>
<body>
<p>some content</p>
</body>
</html>
```

在这段程序代码中，我特意用粗体字强调显示了 getter 和 setter 方法。请遵循本示例所

使用的语法，包括 defineGetter 和 defineSetter 关键字前后的两个下划线，以及使用 prototype 进行赋值。括号后紧接着是成员数据，在本示例中是 title，然后是设置和获取数据的函数。

使用 getter 和 setter，就是直接访问该成员数据以及直接为该成员数据赋值。请注意，从外部访问的成员数据（title）不与真实的成员数据（myTitle）同名。而且，在本示例的 getter 中多添加了一段字符串，这是用来测试是否真正调用到了 getter 和 setter 方法。

提示

另外一种方法是在对象初始化器中使用 getter 和 setter，或者使用一次性的对象类型（在本章稍后部分将做介绍）。关于 getter 和 setter 的更多内容可以参考 http://developer.mozilla.org/En/Core_JavaScript_1.5_Guide:Creating_New_Objects:Defining_Getters_and_ Setters。

到目前为止的所有示例，都是将基本的 JavaScript 对象（如 String）作为参数。当然也可以使用自定义对象来封装窗体的页面元素，这是一种消除浏览器差异的非常有效的方式。下一个小节将介绍 JavaScript 对象的封装。

13.3 对象封装

之前我们曾经说过，在创建新对象的时候应该如何将页面对象作为参数传入新对象中。自定义对象封装了页面对象，可以创建一系列功能从而隐藏实现细节。当要修改某个实现时，就不需要查找所有的 JavaScript 代码并逐一修改页面元素的属性，而只需要修改所调用的方法。

如果浏览器支持的底层实现发生了变化，对象封装可以隐藏所有实现的细节。这个时候不需要修改应用程序。隐藏浏览器特定细节，使得动态应用程序更容易维护。

此外，再也不必不断检查浏览器是否支持特定的功能。代码或者 JavaScript 程序库，可以在对象创建的时候（通常也就是载入页面的时候）进行相应的检查。

示例 13.5 中介绍了 JavaScript 里的对象封装，以及如何处理跨浏览器的差异。该应用程序中包含了一个小型的对象程序库，它隐藏了实现细节。该页面中有两个 div 元素，每个元素里各有一个图像元素。这两个图像元素在页面中采用绝对定位，并且一个是透明的，另一个是不透明的。当载入页面的时候，会调用函数，创建自定义对象的示例，并依次传入每个 div 元素。第一个元素的 opacity 属性将设置为 1.0（可以看见的），而第二个的 opacity 属性将设置为 0（完全透明）。点击该页面，会降低可见对象的 opacity 属性，并且同时提高之前透明对象的 opacity 属性，对调这两个元素的效果。

示例 13.5 对象封装

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
```

```
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>Encapsulating</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<meta http-equiv="X-UA-Compatible" content="IE=EmulateIE7" />
<style type="text/css">
div {
       position: absolute;
       top: 30px;
       left: 50px;
    }
#div2 {
      opacity: 0.0; filter: alpha(opacity=0);
}
#div1 img {
filter:
progid:DXImageTransform.Microsoft.AlphaImageLoader(src=fig0902.png,
sizingMethod='scale');
}
#div2 img {
filter:
progid:DXImageTransform.Microsoft.AlphaImageLoader(src=fig0903.png,
sizingMethod='scale');
}

</style>
<script type="text/javascript">
//<![CDATA[

// 全局
var theobjs = new Array();

document.onclick=function() {

   // 淡出 div1
   var currentOpacity = theobjs["div1"].objGetOpacity();

   // 结束转换
   if (currentOpacity <= 0) {
      document.onclick=null;
      return;
   }

   currentOpacity-=0.1;
   theobjs["div1"].objSetOpacity(currentOpacity);

   // 展现 div2
   currentOpacity = theobjs["div2"].objGetOpacity();
```

```
    currentOpacity+=0.1;
    theobjs["div2"].objSetOpacity(currentOpacity);
}

function DivObj(obj) {
    this.obj = obj;

    this.getAlphaOpacity = function () {
        var fltr = this.obj.style.filter;
        var indx1 = fltr.indexOf("opacity=");
        var indx2 = fltr.indexOf(")");
        fltr = fltr.substring(indx1+8,indx2) / 100;
        return fltr;
    };
    this.getCSSOpacity = function() {
        return parseFloat(this.obj.style.opacity);
    }
    this.objGetOpacity = (this.obj.style.filter === undefined) ?
this.getCSSOpacity:
this.getAlphaOpacity;
    this.alphaOpacity = function(value) {
        var opacity = value * 100;
        this.obj.style.filter="alpha(opacity="+opacity+")";
    };
    this.cssOpacity = function(value) {
        this.obj.style.opacity=value;
    };
    this.objSetOpacity = (this.obj.style.filter === undefined) ?
this.cssOpacity :
this.alphaOpacity;

}

window.onload=function() {

  theobjs["div1"] = new DivObj(document.getElementById("div1"));
  theobjs["div2"] = new DivObj(document.getElementById("div2"));

  // 设置最初的透明度
  theobjs["div1"].objSetOpacity(1.0);
  theobjs["div2"].objSetOpacity(0.0);
}

//]]>
</script>
</head>
<body>
<div id="div1">
<img src="fig0902.png" alt="" />
```

```
</div>
<div id="div2">
<img src="fig0903.png" alt="" />
</div>
</body>
</html>
```

该页面在 Safari 浏览器中转换到一半时的效果如图 13.1 所示。因为 IE8 对 opacity 属性的支持不够完整，所以使用 opacity 属性可能存在一些问题，但是该应用程序适用于所有目标浏览器（包括 IE7 和 IE8，遗憾的是不支持 IE6），如图 13.2 所示。

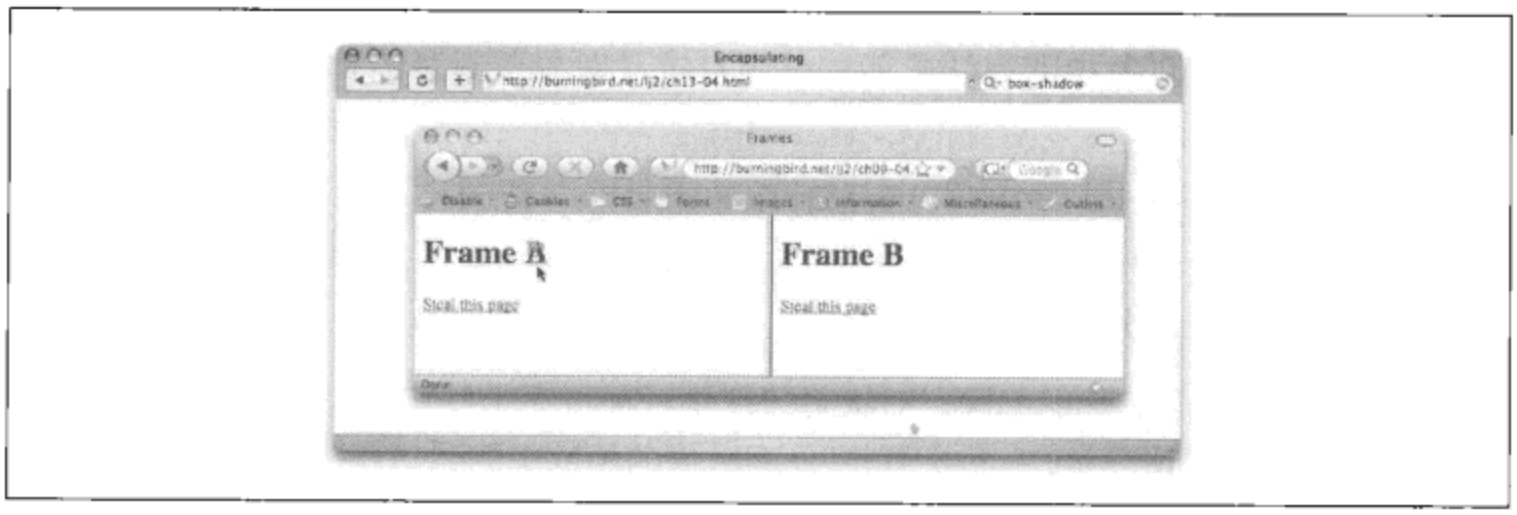


图 13.1　在 Safari 浏览器中测试 opacity 属性的跨浏览器实现

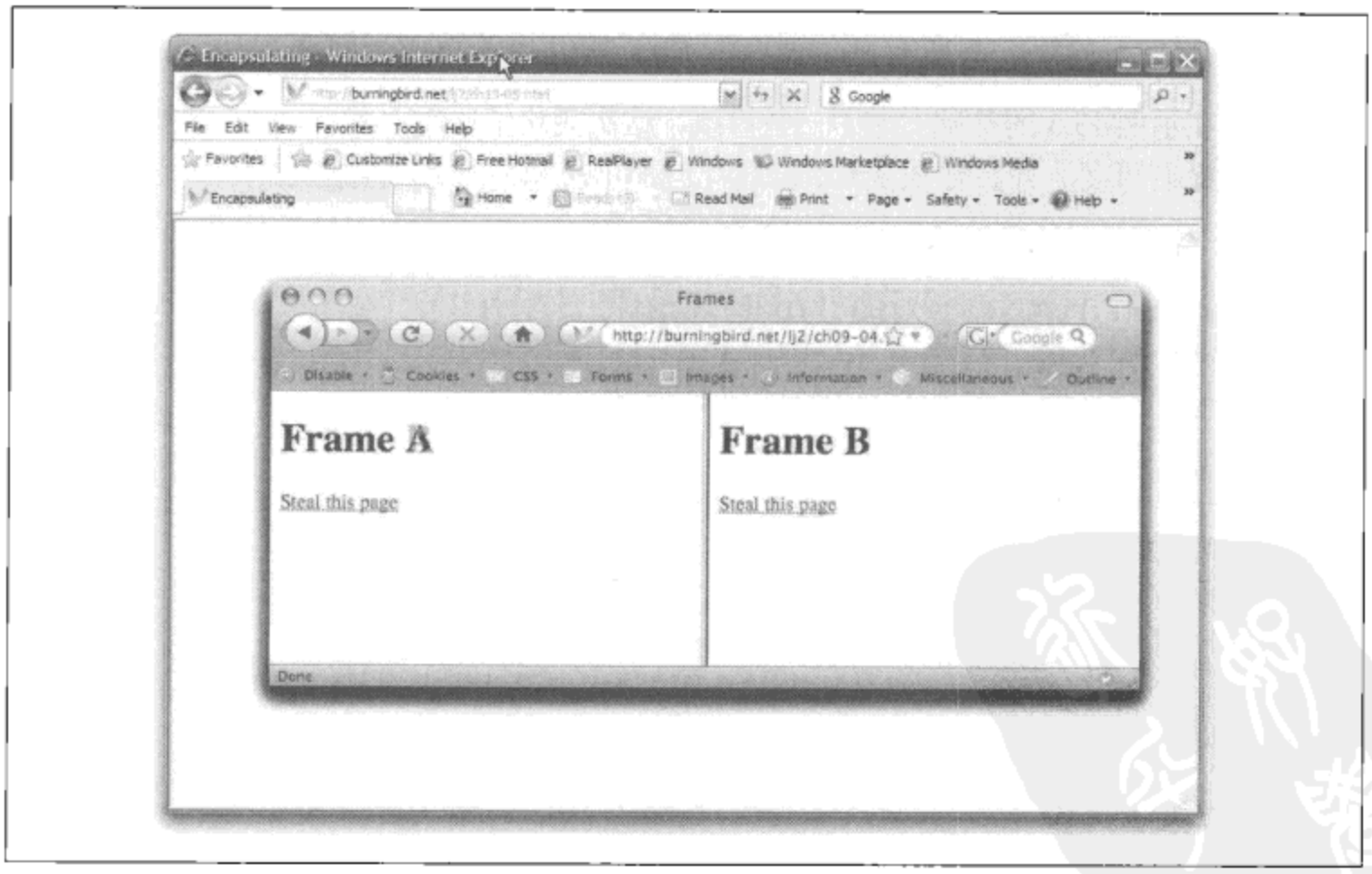


图 13.2　在 IE8 浏览器中测试 opacity 属性的跨浏览器实现

当创建新对象时，页面元素将通过对象引用传入到对象中，这就是页面元素和封装细节的对象之间的关联。该页面元素将赋给对象的新属性 obj。当调整 opacity 属性时，可以通过这个对象属性来访问页面元素。自定义对象和页面元素引用同时存在，除非故意删除或修改该对象元素，或者删除页面元素，例如：

```
theobjs["div1"].obj = null;
```

为了更好地防止对该属性的意外修改，可以将页面元素赋值给自定义对象的内部变量，例如：

```
var pageElement = obj;
```

现在，只有同一个对象中的方法才能够访问自定义对象所封装的页面元素，例如：

```
this.pageElement.style.opacity=value;
```

自定义对象、对象封装的使用已经不如过去来得重要，特别是动态页面在不同浏览器中有着不同的效果。然而，这还是一种隐藏浏览器差异的好方法，也符合应用程序开发中“一次编码，多次复用”的原则。

即使不存在跨浏览器差异的问题，对象封装也是一种将功能封装到可复用对象的有效方法。回到示例 13.5，请注意，在负责解析 opacity 值的函数中能闻出一些代码的“坏味道”，并且依赖于空格的做法可能不适用于所有情况，也比不上访问 CSS 属性那么简便。

然而，多亏了 JavaScript 中提供的 prototype 功能，它能够以 CSS 属性设置的方式来设置 opacity 属性。同样，为所有浏览器设置 filter 属性也不再成为问题。

示例 13.6 在示例 13.5 的基础上做了一些修改，这次简化了代码，通过 style 对象的 opacity 属性来设置透明度，并在数值变化的时候设置 filter 和 opacity 属性值。

示例 13.6　简化了 opacity 操作的对象封装

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>Encapsulating</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<meta http-equiv="X-UA-Compatible" content="IE=EmulateIE7" />
<style type="text/css">
div {
       position: absolute;
       top: 30px;
       left: 50px;
   }
#div2 {
      opacity: 0.0; filter: alpha(opacity=0);
}
#div1 img {
```

```
filter:
progid:DXImageTransform.Microsoft.AlphaImageLoader(src=fig0902.png,
sizingMethod='scale');
}
#div2 img {
filter:
progid:DXImageTransform.Microsoft.AlphaImageLoader(src=fig0903.png,
sizingMethod='scale');
}

</style>
<script type="text/javascript">
//<![CDATA[


// 全局
var theobjs = new Array();
document.onclick=function() {

   // 淡出 div1
   var currentOpacity = theobjs["div1"].objGetOpacity();

   // 结束转换
   if (currentOpacity <= 0) {
      document.onclick=null;
      return;
   }

   currentOpacity-=0.1;
   theobjs["div1"].objSetOpacity(currentOpacity);

   // 展现 div2
   currentOpacity = theobjs["div2"].objGetOpacity();
   currentOpacity+=0.1;
   theobjs["div2"].objSetOpacity(currentOpacity);
}

function DivObj(obj) {
   this.obj = obj;

   this.objGetOpacity = function() {
      return parseFloat(this.obj.style.opacity);
   }

   this.alphaOpacity = function(value) {
      var opacity = value * 100;
      this.obj.style.filter="alpha(opacity="+opacity+")";
   };
   this.cssOpacity = function(value) {
```

```
            this.obj.style.opacity=value;
    };
    this.objSetOpacity=function(value) {
        this.alphaOpacity(value);
        this.cssOpacity(value);
     }

}
window.onload=function() {

  theobjs["div1"] = new DivObj(document.getElementById("div1"));
  theobjs["div2"] = new DivObj(document.getElementById("div2"));

  // 设置最初的透明度
  theobjs["div1"].objSetOpacity(1.0);
  theobjs["div2"].objSetOpacity(0.0);
}

//]]>
</script>
</head>
<body>
<div id="div1">
<img src="fig0902.png" alt="" />
</div>
<div id="div2">
<img src="fig0903.png" alt="" />
</div>
</body>
</html>
```

同样，修改后的程序也适用于所有浏览器（IE 6.0 除外）。不过 IE 不支持 style.opacity 操作，同时 Opera、Safari 以及 Firefox 又不支持 filter opacity 操作，那又要怎么办？

这 4 种浏览器都支持 JavaScript 中的 prototype 对象，也就是说，可以在任意脚本对象中动态地添加新属性。这就包括了 IE 中 style 对象的 opacity 属性，以及其他浏览器的 filter 属性。因为其他 3 种浏览器都支持 CSS 样式设置，而 IE 支持 filter 设置，所以 opacity 属性的变化适用于这 4 种浏览器。此外，因为以 CSS 样式设置 opacity 属性非常方便，所以这也可以在 IE 中使用“状态保存机制”，尽管对属性的赋值并没有真正修改页面的任何元素。

将 prototype 和对象封装结合使用，能够真正地简化跨浏览器的实现。如同 JavaScript 中 prototype 天生的便利性一样，某些时候需要以更传统的面向对象思路进行开发，例如类的继承。

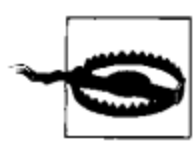
13.4 构造函数链和 JavaScript 继承

JavaScript 并不是典型的面向对象语言，与面向对象语言相比，JavaScript 也有自己的优

势。当然，传统面向对象设计的一些要素同样可以应用在 JavaScript 程序中。在之前的小节中介绍了面向对象设计的一大要素——封装。接下来的小节中将介绍另外一个要素——继承。

继承，就是允许一个对象继承另外一个对象中的方法和属性。这是面向类开发的一个基本要素，因为类可以从其他类中继承，并且在新的类中根据新的定义覆盖原有的函数。

在 JavaScript 中，可以利用一些类似的技术来实现该行为，例如 apply 和 call 方法（在 JavaScript 1.3 之后的版本都支持）。



警告

JavaScript 初学者可能会觉得 apply 和 call 方法不容易理解。所以思考这两个方法的用法可能会多花费你一些时间。

你可以在某个对象的上下文中调用 apply 函数，从而为另一个对象应用某个方法。该函数中定义了两个参数，一个表示调用对象的上下文，以 this 来表示，另一个是表示第二个对象方法参数的数组变量。call 函数与 apply 函数类似，唯一的区别是它不需要将第二个对象方法参数添加到数组中，而是可以直接列出每个参数。

回到之前的示例，当我们定义新对象的时候，首先要定义对象构造函数，并通过 new 关键字进行调用：

```
theobj = new DivObj(params);
```

apply 和 call 方法允许在另一个对象中调用某个方法。如果使用对象构造函数，那么将以方法和属性继承的方法链接构造函数。唯一的区别是传递参数的方式不同，但它们的行为是一致的。call 方法的第一个参数是指向当前对象的 this，而另外一个参数是该对象的构造函数：

```
obj.call(this,arg1,arg2,..., argn);
```

apply 方法的第一个参数同样是当前的对象，另一个参数是参数数组。如果当前对象有两个参数，而目标对象有 3 个参数，那么只需要传入前两个参数。

```
obj.apply(this,arguments);
```

apply 方法更适用于共享的参数集合，否则 call 方法则更为方便。

JavaScript 中的这两个方法可以用于链接构造函数。示例 13.7 通过 apply 函数和链接的构造函数来模拟面向对象中的继承。首先创建第一个对象 Tune，用来保存歌曲标题和类型信息。此外还有一个方法，它将返回包含这两个属性的字符串。第二个名为 Artist_tune 的对象，相比多出了一个保存艺术家信息的属性，以及一个返回所有属性的字符串的新方法。apply 方法将直接用于 tune 函数和对象。此外，当定义两个对象的时候，tune 的构造函数将赋给 Artist_tune 的 prototype 属性。

示例 13.7　通过 apply 方法实现构造函数链和继承

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>Constructor Chaining</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript">
//<![CDATA[

function Tune(title,type) {
   this.title = title;
   this.type = type;
   this.getTitle=function() {
     return "Song: " + this.title + " Type: " + this.type;
   }
}

function Artist_tune(title,type,artist) {
   this.artist = artist;
   this.toString("Artist is " + artist);
   Tune.apply(this,arguments);
   this.toString = function () {
     return "Artist: " + this.artist + " " + this.getTitle();
   }
}

window.onload=function() {
   Artist_tune.prototype = new Tune();

   var song = new Artist_tune("I want to hold your hand", "rock", "Beatles");
   document.writeln(song.toString());
}

//]]>
</script>
</head>
<body>
<p>some content</p>
</body>
</html>
```

Tune 对象实例将添加为 Artist_tune 对象的一个属性，通过 apply 方法链接这两个对象的构造函数。现在，当通过 Artist_tune 的 toString 方法输出歌曲信息的时候，对象上下文（也就是 this）将依次应用于这两个对象，获取艺术家信息（Artist_tune 的 artist 属性）和歌曲标题信息（Tune 的 title 属性）。

构造函数链在 Ajax 应用中非常普遍，特别是为某个对象绑定事件句柄时，这时事件句

柄不仅仅是事件对象的引用，而且还是对象上下文的引用，也就是 call 或 apply 方法的第一个参数。下面是一个我自己使用的 JavaScript 程序库，用于为对象绑定事件句柄。相关代码如下所示：

```
bindEventListener : function(obj, method) {
  return function(event) { method.call(obj, event || window.event)};
},

bindObjMethod : function (obj, method) {
   return function() { method.apply(obj, arguments); }
},...
```

这段代码与之前示例中所看到的对象实现并不一样。这也是下一小节中将要讨论的主题——一次性对象。

13.5 一次性对象

当你封装一个对象时，通常会使用之前小节中所介绍的方法来创建对象的多个实例。有些时候，我们仅仅需要一个对象实例，也就是一次性（one-off）对象。为什么称之为“一次性对象”？因为该对象只有一个实例。

通过 JavaScript 对象语法可以创建一次性对象，构建属性和方法的数组，并赋值给某个变量，例如：

```
var oneOff = {
    variablea : "valuea",
    variableb : "valueb",
    method : function () {
        return this.variablea + " " + this.variableb;
    }
}
```

在上述代码片段中，大括号引用了包含对象语法的键/值对，并且赋给变量 oneOff。因为 JavaScript 的对象皆为方法，方法也皆为对象，所以可以在对象中添加方法，以及其他静态且没有方法的属性。如果要访问一次性对象的成员属性或方法，可以使用键的数值，例如 variablea 或 variableb，然而对象以外则要使用标准的属性操作符，例如：

```
alert(oneOff.variablea);
alert(oneOff.method());
```

另一种方法是创建相同的一次性对象，例如：

```
var oneOff = new Object();
oneOff.variablea = "valuea";
oneOff.variableb = "valueb";
oneOff.method = function () {
```

```
                                    return this.variablea + " " + this.variableb;
                                    };
```

从 Object 对象中构造一个新的对象实例，接着将属性和方法添加到对象实例中。这个时候不需要使用 prototype，因为这些新属性和新方法并不需要添加到潜在对象中；所以可以直接添加到对象实例中。方法通过 this 关键字和命名的属性，就可以访问父对象的其他属性。

第二种方法中访问属性的代码如下所示：

```
alert(oneOff2.variableb);
alert(oneOff2.method());
```

这两种方法都以同样的方式访问对象属性。

创建一次性对象的最后一种方法是创建 Function 对象的实例：

```
var oneOff = new Function() {
             this.variablea = "variablea";
             this.variableb = "variableb";
             this.method = function () {
                           return this.variablea + " " + this.variableb;
                         }
             }
```

该方法访问对象属性的方式也和前面两种方法一样。

当需要在一个对象中封装一组方法和属性时，可以使用一次性对象，并在整个应用程序中复用该对象。当然这个时候不需要创建该对象的多个实例，而仅仅只需要一个实例。

在本书前面的章节中，我已经说明应该尽可能地避免全局变量。然而，并不是所有信息都应当存储在局部变量中。在使用大量单数值的全局变量和完全不使用全局变量之间的平衡点就是使用基于一次性对象的对象库。

示例 13.8 中创建了一次性对象 flipper，该对象中有 3 个方法：getStyle、setBackgroundColor 以及 flipColors。该对象的作用是保存指向两个 div 元素的引用，并且每次点击时对调这两个元素的颜色。该对象将获取指定的样式，设置 div 元素的背景颜色，相应地对调这两个元素的背景颜色。

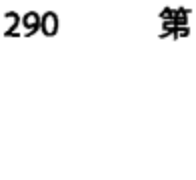
示例 13.8　对调两个元素背景颜色的一次性对象

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>One-Off/Object Literal</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<style type="text/css">
```

```
div
{
   margin: 20px;
   width: 200px; height: 200px
}
#div1
{
   background-color: #ffff00;
}
#div2
{
   background-color: #00ff00;
}
</style>
<script type="text/javascript">
//<![CDATA[

var flipper  = {
   obj1 : null,
   obj2 : null,
   getStyle : function (obj, jsStyleName, styleName) {
      if (obj.currentStyle) {
         return obj.currentStyle[jsStyleName};
       } else if (window.getComputedStyle) {
         return
document.defaultView.getComputedStyle(obj,null).getPropertyValue(styleName);
       } else {
         return undefined;
       }
   },
   setBackgroundColor : function(obj, color) {
      obj.style.backgroundColor=color;
   },
   flipColors : function()  {
      var color1 = this.getStyle(this.obj1, "backgroundColor",
"background-color");
      var color2 = this.getStyle(this.obj2, "backgroundColor",
"background-color");
      this.setBackgroundColor(this.obj1,color2);
      this.setBackgroundColor(this.obj2,color1);
   }
};

window.onload = function() {
   flipper.obj1 = document.getElementById("div1");
   flipper.obj2 = document.getElementById("div2");
}
```

```
document.onclick = function() {
  flipper.flipColors();
}

//]]>
</script>
</head>
<body>
<div id="div1">
<p>This is first square</p>
</div>
<div id="div2">
<p>This is the second square</p>
</div>
</body>
</html>
```

当页面载入之后，只要点击页面的任意位置就可以对调页面中两个 div 元素的背景颜色。当然，通过添加全局函数和变量也可以实现相同的功能，但是代码会显得不整洁、不安全（特别在使用多个 JavaScript 程序库的情况下），而且也不容易封装和维护。



提示

之前封装的对象都是使用大写的名字，而一次性对象则是用小写的名字。这是一种广泛应用的命名规范，用来区分可以初始化的对象（通过 new 关键字）和不可以初始化的对象。

当然，单次使用某对象需要相当多的工作量。对象通常会封装在不同的文件中，从而使得复用更加简单。

13.6 对象库：为复用而封装对象

本书的示例都是放在一个文件中的，包括 JavaScript 代码、CSS 样式等。之所以这样做，是因为这让你可以很容易地复制示例，也能够更容易地读懂代码。

然而在实际应用中，JavaScript 代码通常应该放在单独的文件（以.js 为后缀）中。将 JavaScript 代码放在单独的文件中，可以使代码更容易阅读，更容易修改代码。那么接踵而至的问题是到底需要多少个 JavaScript 文件呢？毕竟这些单独的文件会增加页面载入的速度。

将 JavaScript 代码分在不同文件中的良好规范是，将 JavaScript 对象分成不同的层次，例如访问层、业务方法层等。就像我自己，将一系列用于跨浏览器动态页面效果的对象，放在独立的 JavaScript 程序库文件中。通过独立的程序库文件，不需要访问动画对象的应用程序，可以只访问静态效果程序库。

独立的 JavaScript 程序库还包括图片扩展、创建伸缩效果以及使用第三方 Ajax 程序库

实现的各种效果。每个独立的 JavaScript 文件都在各自的 script 标签中，例如：

```
<script type="text/javascript" src="flipper.js">
</script>
```

或者

```
<script type="text/ecmascript" src="flipper.js">
</script>
```

当然，如果使用多个程序库文件，那么还要确保事件处理是正常的。本书的大多数示例都直接将事件句柄赋给事件监听器，例如：

```
window.onload=function { ...}
```

然而和上述方法相比，这种情况更倾向于使用自己封装的一次性对象，如 bb 变量，以及相应的事件处理方法，其代码如下所示：

```
var bb = {
// 添加事件监听器
manageEvent : function (eventObj, event, eventHandler) {
   if (eventObj.addEventListener) {
      eventObj.addEventListener(event, eventHandler,false);
   } else if (eventObj.attachEvent) {
      event = "on" + event;
      eventObj.attachEvent(event, eventHandler);
   }
},
...
}
```

接着，当需要将事件句柄添加到某个对象的事件中时，就可以使用如下所示的代码：

```
bb.manageEvent(window,"load", eventHandlerFunction);
```

现在不会覆盖事件句柄所赋的值，所以就不需要担心别的 JavaScript 程序库是否也对 window.load 事件处理函数进行了赋值。

独立的函数也可以放在独立的文件中，但问题是这可以导致要维护并管理太多小文件，并且在页面中可能包含过多的文件。所以为特定目的创建 JavaScript 程序库是比较合适的一种方法，这样不多也不少。

但是现在我们还无法追踪代码中的错误，所以接下来要介绍如何处理错误，这对创建自定义对象来说特别重要。

13.7 高级错误处理技术（try、throw 和 catch）

在应用程序中，经常会在调用函数时检查函数的返回值，然而这种方法并不是最理想的。更好的方法是调用函数时使用对象，而不对返回结果进行不断的测试，并且在代

码结尾处提供异常处理代码，以捕捉任何可能的错误。

自从 JavaScript 1.5 开始，try…catch…finally 语法就被添加到 JavaScript 语言中了。try 语句应放在代码最开始处，表示异常处理机制的开始。catch 语句则在代码最末尾，用来捕捉任何异常并对异常进行相应的处理。

finally 语句并不是必需的，通常用于处理无论错误与否都要执行的操作。finally 语句紧挨着 catch 语句，异常处理机制的示例代码如下所示：

```
try {
...
}
catch (e) {
...
}
finally {
...
}
```

在 JavaScript 1.5 引擎中，一共实现了 6 类错误：

- EvalError　eval 使用不正确所抛出的错误；
- RangeError　数字值超出范围所抛出的错误；
- ReferenceError　使用无效引用所抛出的错误；
- SyntaxError　无效语法所抛出的错误；
- TypeError　变量类型与预期不符合所抛出的错误；
- URIError　encodeURI()和 decodeURI()使用不正确所抛出的错误。

当捕捉到异常时，使用 instanceOf 可以判断所捕捉到的异常属于哪一类内建错误类型。下列代码故意抛出了 TypeError 并捕捉该错误。抛出的异常中包含 message 属性，通过它可以获知异常的具体信息。

```
try {
  var somearray = null;
  alert(somearray[18]);
} catch (e) {
   if (e instanceof TypeError) {
      alert("Type error: " + e.message);
   }
}
```

你还可以针对错误类型进行多次判断，记录下错误，甚至调用特殊的异常处理方法，而这一切的处理都应放在 catch 语句中。如果在无论成功与否的情况下都需要处理某些功能，那么可以在 finally 语句中包含相应的处理。

```
try {
  var somearray = null;
  alert(somearray[18]);
} catch (e) {
   if (e instanceof TypeError) {
      alert("Type error: " + e.message);
   }
}
finally {
   somearray = null;
}
```

这种复杂的异常处理模型更适用于对象构建系统，因为对象方法可以通过 throw 语句抛出异常，而不需要通过返回 null 或者其他错误值来表示。所以抛出任意异常类型，并且在代码中处理相应的错误，更适用于对象模型。

在示例 13.9 中，我们对之前的示例做了一定的修改，在修改透明度方法中判断传入函数的值是否是数字。如果数值不是数字，那么将抛出“NotANumber”的错误。接着在 objSetOpacity 方法中处理该错误。为了让错误发生，我们故意将字符串与数字相连接，从而将整个内容转换成字符串。这里就不重复整个应用程序，只是列出代码的一部分，并强调修改点。

示例 13.9　判断参数类型并相应地抛出错误

```
var theobjs = new Array();
document.onclick=function() {

   // 淡出 div1
   var currentOpacity = theobjs["div1"].objGetOpacity();

   // 结束转换
   if (currentOpacity <= 0) {
      document.onclick=null;
      return;
   }

   currentOpacity-=0.1;
   theobjs["div1"].objSetOpacity(currentOpacity);

   // 展示 div2
   currentOpacity = theobjs["div2"].objGetOpacity();
   currentOpacity+=0.1;
   theobjs["div2"].objSetOpacity(currentOpacity);
}

function DivObj(obj) {
   this.obj = obj;
```

```
this.objGetOpacity = function() {
   return this.obj.style.opacity;
}

this.alphaOpacity = function(value) {
   if (typeof value == "number") {
        var opacity = value * 100;
        this.obj.style.filter="alpha(opacity="+opacity+")";
   } else {
     throw "NotANumber";
   }
};
this.cssOpacity = function(value) {
   if (typeof value == "number") {
      this.obj.style.opacity=value;
   } else {
     throw "NotANumber";
   }
};
this.objSetOpacity=function(value) {
   value = "alpha is " + value;
   try {
     this.alphaOpacity(value);
     this.cssOpacity(value);
   } catch (e) {
     alert(e);
   }
}

window.onload=function() {

  theobjs["div1"] = new DivObj(document.getElementById("div1"));
  theobjs["div2"] = new DivObj(document.getElementById("div2"));

  // 设置最初的透明度
  theobjs["div1"].objSetOpacity(1.0);
  theobjs["div2"].objSetOpacity(0.0);
}
```

设置对象透明度的方法没有返回值，这样做只是为了证明返回值不是错误处理的唯一方式。相反，通过抛出异常，调用程序不必判断方法返回值的状态，并且该方法可以触发错误处理。

该示例使用了字符串文本，不过你也可以创建一个对象，例如一次性对象，并抛出该对象。如果需要传入更多的信息，而不仅仅是字符串，那么这是一种更好的方式。

13.8 知识测验

1. 如果要为 Number 对象创建一个名为 triple 的新方法，用来对当前 Number 对象的数值求 3 倍值。而且希望该方法能够适用于所有数字，那么要怎么实现呢?
2. 如何隐藏新对象的数据成员？为什么要这样做?
3. 创建一个参数为数字型的函数，并且当参数类型错误时返回错误。如果不使用返回值，应该怎样实现该功能?
4. 在之前的示例中可以看到这样的代码：

```
var theEvent = nsEvent ? nsEvent : window.event;
```

为什么在处理透明度差异的时候不能使用相同的功能呢?

5. 创建一个自定义对象，使其拥有 3 个公有方法：changeState、getColor、getState；以及两个私有数据成员：background 和 state。将 state 属性设置为 on，将 background 颜色属性设置为#fff。changeState 方法用来判断 state 是否为 on，如果 state 为 on，那么将 state 改成 off，并同时将颜色设置为#000。getColor 方法将返回背景颜色 background， getState 负责返回状态 state。

13.9 测验答案

1. 使用 Number 的 prototype 属性：

```
Number.prototype.triple = function() {
   var nmToTriple = this.valueOf() * 3;
   return nmToTriple;
}
var newNum = new Number (3.0);
alert(newNum.triple());
```

2. 通过 var 关键字声明数据成员，而不是 this 关键字。数据隐藏的目的是控制数据如何访问和修改。
3. 使用 throw 语句触发错误。然后通过 try…catch 语句捕捉异常。

```
if (typeof value != "number") {
   throw "NotANumber";
}
```

4. 与事件对象不一样，处理透明度的时候会牵扯更多业务模型上的差异，不仅仅是属性的不同，而且与赋值属性的可能值相关。
5. 下面是其中一种方法：

```
function Control() {
   var state = 'on';
   var background = '#fff';

   this.changeState = function() {
      if (state == 'on') {
         state = 'off';
         background = '#000';
      } else {
         state = 'on';
         background = '#fff';
      };
   }
   this.getState = function() {
      return state;
   };

   this.getColor = function() {
      return background;
   };
}
```

第 14 章

使用 Ajax

Ajax 最初所涉及的功能是在客户端页面和服务器间完成页内通信。而现在它不断延展到了动态网页效果领域，实现了许多被称为动态 HTML（DHTML）的特性。除了这些革新之外，Ajax 使大家对 JavaScript 特别是动态 JavaScript 功能产生了浓厚的兴趣。

本章中的 Ajax 示例和之前章节中的示例有所不同，Ajax 需要服务器端组件的配合。对于 Ajax 开发而言，选择 Ruby 开发服务器端程序是最流行的，不过所有服务器端程序开发语言都能够对 Ajax 请求进行处理。在本章的示例中，服务器端程序是使用 PHP 开发的，这是因为它是所有语言中与 JavaScript 最相似的，同时它也是应用最广泛的服务器端脚本语言。

提示

从技术角度而言，Ajax 应用程序并不一定非得靠服务器端语言来支持；你只需要让服务器端给予应答即可。因此也可以将服务器端换成一个静态的 HTML 或 XML 文件库，然后根据请求的 URI 做出相应的应答即可。不过，通过一个服务器端应用程序来对请求进行处理，并返回一个明确的应答还是 Ajax 应用程序最常用的模式，这也是本章和下一章中所演示的模式。

14.1 Ajax 的工作原理

Ajax 并没有我们想象中那么复杂。首先需要向服务器端发送一个请求，然后调用一个服务，接着返回数据。不过，和提交一个表单、载入一个新页面或使用 iframe 和远程脚本（在前面的章节中已经介绍过）不同，Ajax 是在同一个 Web 页面文档中完成这些活动的。

为了管理服务器端和客户端之间的异步通信，需要一个专门的对象，可能是微软早期提供的 ActiveXObject 或更通用和标准的 XMLHttpRequest。异步通信的意思是当请求发送时，客户端并不会暂停，并且一直等待服务器端的处理完成。与此同时，客户端将指定一个在请求状态发生改变时将调用的函数。

在这个函数中，将检查请求的状态，如果请求已经处理完成，并且没有发现服务器端出现错误，那么将对返回的数据进行处理并以某种形式更新到页面上。

对于 Web 页面的访问者而言，这些活动看起来就像在页面中发生的一样，要比纯粹的客户端/服务器端交互好得多。在访问服务器时，除非用户在监测服务器请求，或者应用程序提供了一些操作提示，否则用户是无法看到可视化的指示信息的。

现在你将站在 10 000 英尺的高空审视 Ajax，让我们首先看一下 Ajax 应用程序，然后在本章的余下部分仔细分析每个片段。

14.2 Hello Ajax World!

最常见的 Ajax 应用是基于用户触发的事件返回一些数据，例如当有人在下拉选择框中选择某个选项时。在本小节讨论的“Hello Ajax World”应用程序中，有一个用来选择“州”的下拉选择框，当用户选择某一个州时，就会在这个下拉选择框下面以列表的形式显示出该州中的所有城市。

在示例 14.1 所包含的 Web 页面中，有一个用来发出 Ajax 服务器调用的脚本。该页面中有一个表单，表单中有一个下拉选择框元素，并且预先载入了一些“州”的名字。

示例 14.1　第一个 Ajax 应用

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>City lookup</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<style type="text/css">
div.elem
{
   margin: 20px;
}
div#cities
{
  display: none;
}
</style>
<script type="text/javascript">
//<![CDATA[

// 全局变量
var xmlHttpObj;

function catchEvent(eventObj, event, eventHandler) {
   if (eventObj.addEventListener) {
      eventObj.addEventListener(event, eventHandler,false);
```

```
    } else if (eventObj.attachEvent) {
       event = "on" + event;
       eventObj.attachEvent(event, eventHandler);
    }
}

catchEvent(window,"load", function() {
   document.getElementById("cities").style.display="block";
   document.getElementById("submitButton").style.display="none";
   document.getElementById("stateList").onchange=populateList;
});
// 创建XHR对象
function getXmlHttp() {
   var xmlhttp = null;
   if (window.XMLHttpRequest) {
      xmlhttp = new XMLHttpRequest();
      if (xmlhttp.overrideMimeType) {
         xmlhttp.overrideMimeType('text/xml');
      }

   } else if (window.ActiveXObject) {
      xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");
   }
   return xmlhttp;
}

// 准备并发送XHR请求
function populateList() {
   var state = document.getElementById("stateList").value;
   var url = 'ch14-02.php?state=' + state;

   //如果不支持xmlHttpObj
   if (!xmlHttpObj)
      xmlHttpObj = getXmlHttp();
   if (!xmlHttpObj) return;

   xmlHttpObj.open('GET', url, true);
   xmlHttpObj.onreadystatechange = getCities;
   xmlHttpObj.send(null);

}

// 处理返回的数据
function getCities() {
   if(xmlHttpObj.readyState == 4 && xmlHttpObj.status == 200) {
      document.getElementById('cities').innerHTML =
xmlHttpObj.responseText;
   } else if (xmlHttpObj.readyState == 4 && xmlHttpObj.status != 200) {
      document.getElementById('cities').innerHTML = 'Error: preSearch Failed!';
   }
```

```
}
//]]>
</script>
</head>
<body>
<h3>Select State:</h3>
<form action="ch14-02.php" method="get">
<div class="elem">
<select id="stateList" name="state">
<option value="CA">California</option>
<option value="MO">Missouri</option>
<option value="WA">Washington</option>
<option value="ID">Idaho</option>
</select>
<p><input type="submit" value="Get Cities" id="submitButton" /></p>
</div>
<div class="elem" id="cities">
<p> </p>
</div>
</form>
</body>
</html>
```

在这个页面中的标签里，id 为 cities 的 div 元素是一个空元素，它将用来保存 Ajax 请求返回的应答信息。当结果返回时，该元素的 innerHTML 属性值将被新内容替代。图 14.1 展现了页面在 Ajax 调用前的样子。

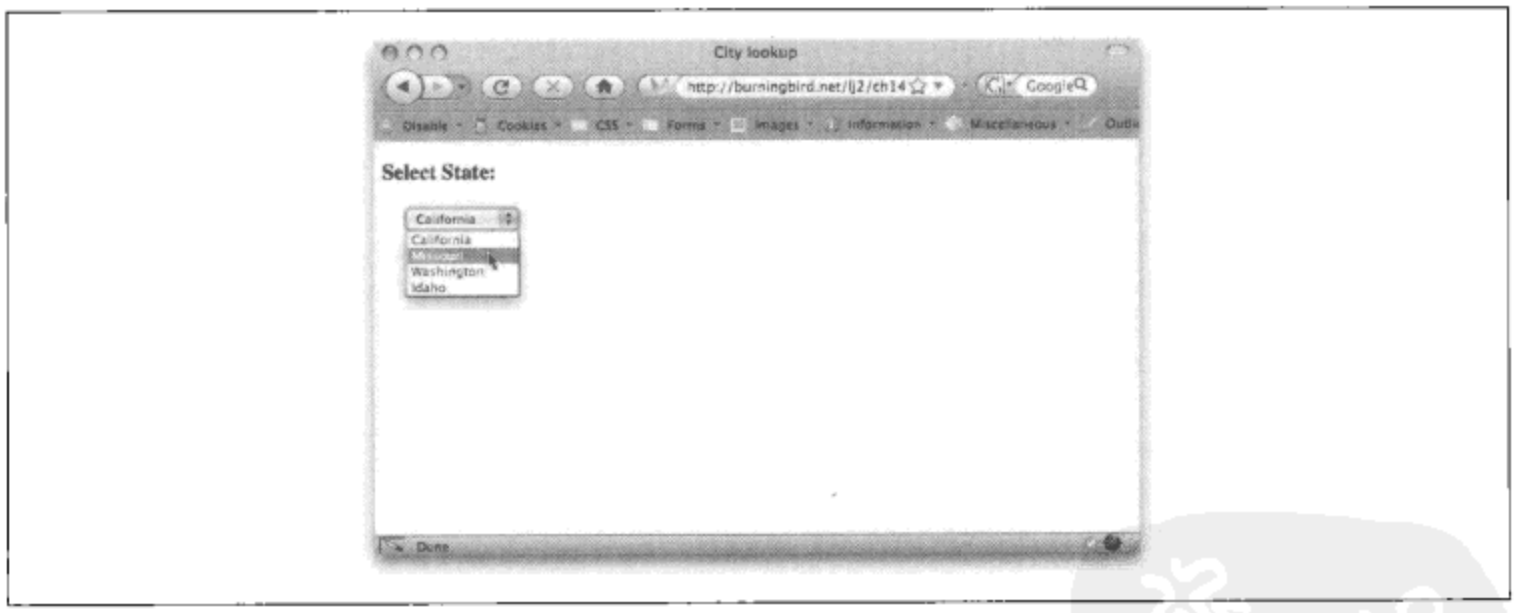


图 14.1　Ajax 调用前的 Web 页面

示例 14.2 列出了该应用程序的服务器端组件。这通常应该通过一个数据库请求来查询这里列举的多个州中的城市信息。不过为了尽量让该示例能够独立运作，在此将以静态字符串的形式给出用户所选州的城市信息。

示例 14.2　用 PHP 为 Ajax 应用程序编写的服务器端组件

```
<?php
```

```
//如果没有传入查询字符串，那么将无法搜索
if(empty($_REQUEST['state'])) {
    echo "No State Sent";
} else {
    //将查询字符串中开始和结尾处的空白符去除
    $search = trim($_REQUEST['state']);
    switch($search) {
      case "MO" :
         $result = "<ul><li>St. Louis</li>" .
                   "<li>Kansas City</li></ul>";
         break;
      case "WA" :
         $result = "<ul><li>Seattle</li>" .
                   "<li>Spokane</li>" .
                   "<li>Olympia</li></ul>";
         break;
      case "CA" :
         $result = "<ul><li>San Francisco</li>" .
                   "<li>Los Angeles</li>" .
                   "<li>Web 2.0 City</li>" .
                   "<li>BarCamp</li></ul>";
         break;
      case "ID" :
         $result = "<ul><li>Boise</li></ul>";
         break;
      default :
         $result = "No Cities Found";
         break;
      }
    echo "<h3>Cities:</h3><p>" . $result . "</p>";
}
?>
```

图 14.2 展示了当选中某个州之后页面的效果。

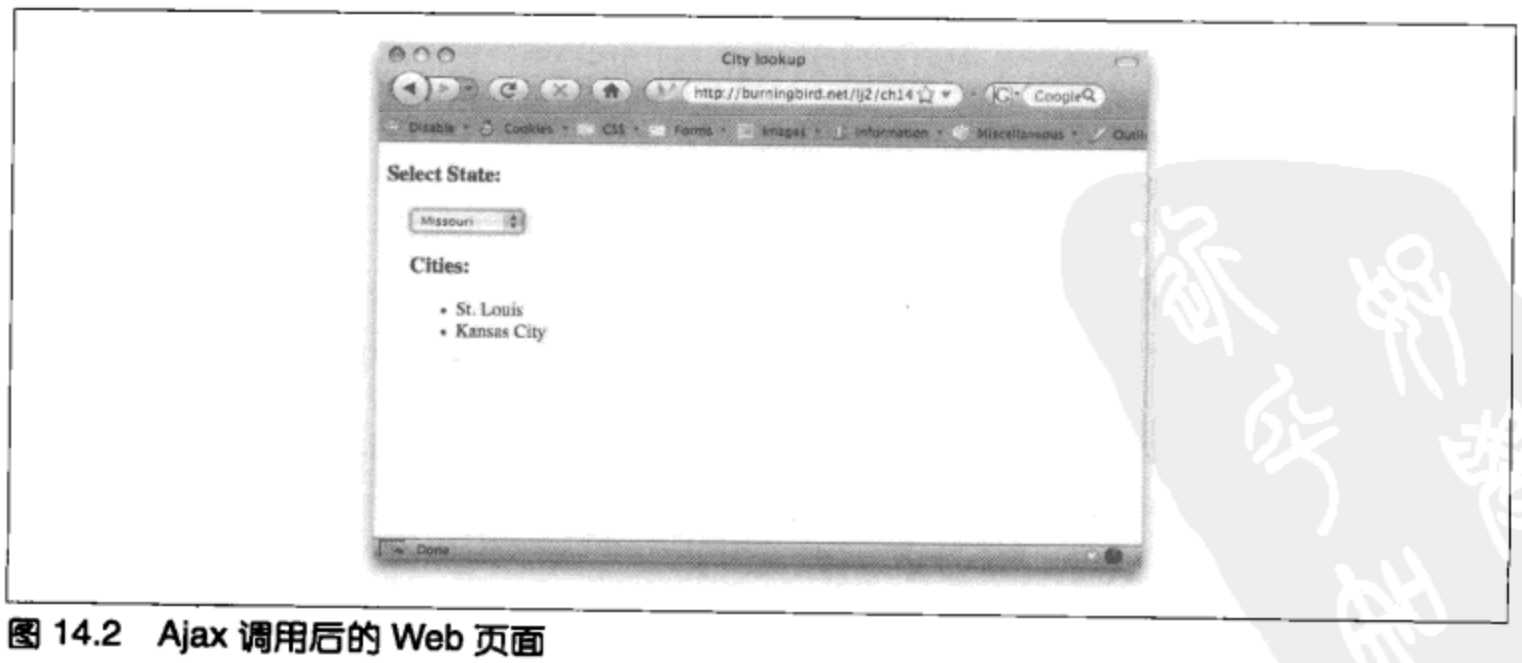


图 14.2　Ajax 调用后的 Web 页面

在接下来的几个小节中，我们将详细说明该页面的各个部分，并在适当的地方提供备选方案。

14.3 XMLHttpRequest 对象及请求的准备与发送

微软是第一个实现 XMLHttpRequest 的公司，在 IE 中它是以 ActiveX 对象的形式提供的。紧接着 Mozilla 就以更直接的方式实现了 XMLHttpRequest 对象，随后其他 Web 浏览器开发者也陆续提供了支持。虽然这两种格式的对象构造器不同，但其功能和方法却是共有的。

提示

微软现在也提供了 XMLHttpRequest 支持，但在老版本的 IE 中（包括 IE 6.x）仍然只支持 ActiveX 版本的对象，这也是我们为什么仍然要对该对象提供支持的原因。

在示例 14.1 中，XMLHttpRequest 对象是一个全局变量，因为很多函数都需要访问它，并且它不能作为参数传给某个函数。该页面中的第一个函数是 getXmlHttp，它负责创建该对象。

在 getXmlHttp 中，定义了一个初值为 null 的局部变量 xmlhttp。这段代码首先检查 window 对象中是否包含 XMLHttpRequest 对象。如果有则创建一个新的 XMLHttpRequest 对象并将其赋给刚才创建的局部变量。如果用户通过老版本的 IE 访问该页面，则创建一个 ActiveX 对象并将其赋给该局部变量，最后该函数的返回值就是这个局部变量。

```
// 创建 XHR 对象
function getXmlHttp() {
   var xmlhttp = null;
   if (window.XMLHttpRequest) {
      xmlhttp = new XMLHttpRequest();

   } else if (window.ActiveXObject) {
      xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");
   }
   return xmlhttp;
}
```

这将解决跨浏览器兼容问题。不是吗?

14.3.1 对象，对象，谁是对象

示例 14.1 展示了创建 XMLHttpRequest 对象的一种方法，那就是使用一个条件语句来检查对象是否存在。如果 XMLHttpRequest 对象不存在，那么就传入相应的 progID（程序 ID）以创建一个 ActiveX 对象，在此传入的是 Microsoft.XMLHTTP。但是还有一个问题，那就是在不同计算机上传给 ActiveXObject 方法的参数应该是不同的。该对象有不同的版

本，包括 MSXML2.XMLHttp、MSXML2.XMLHttp.3.0、MSXML2.XMLHttp.4.0 等。

你可以尝试创建各种版本的 XMLHttpReuqest 对象，但绝大多数 Ajax 程序库和应用程序只关注两种：老版本的 Microsoft.XMLHTTP 和最基本的新版本 MSXML2.XMLHttp。另外，在你尝试创建一个不存在的 ActiveX 对象时，微软的浏览器会抛出一个错误，开发人员可以利用这点来创建合适版本的对象：

```
try
{
    http_request = new ActiveXObject("Msxml2.XMLHTTP");
}
catch (e) {
    try
    {
        http_request = new ActiveXObject("Microsoft.XMLHTTP");
    }
    catch (e) {
        http_request = null;
    }
}
```

如果第一次对象创建没有成功，那么就接着尝试第二个。

现在这段代码变得更加健壮了，但代码量也大大增加了。为了便于复用，我们将把这段代码封装在一个函数中，它要么为全局变量 XMLHttpRequest 设置正确的值，要么将其设置成 null 以表示未能成功创建。最后，这段代码将改成：

```
function getXmlHttp() {
   var xmlhttp = null;
   if (window.XMLHttpRequest) {
      xmlhttp = new XMLHttpRequest();
   } else {
      try
        {
            xmlhttp = new ActiveXObject("Msxml2.XMLHTTP");
        }
        catch (e) {
            try
            {
                xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");
            }
            catch (e) {
               return xmlhttpl;
            }
        }
   }
   return xmlhttp;
}
```

当然，各种跨浏览器兼容问题都已经快解决了，因为 IE7 之后的版本都对 XMLHttpRequest 对象提供了直接的支持。不过，在 IE6 彻底消失之前，你还是需要面对这一复杂的问题。

现在我们已经有了一个 XMLHttpReuqest 对象，接下来可以更进一步了解它了。

14.3.2 XMLHttpRequest 对象的方法

XMLHttpReuqest 是一个很简单的对象，只提供了少量的方法和属性。不过它也无须费心提供大量复杂的方法。

以下是它提供的所有方法，这里是按照在应用程序中出现的频率高低排列的。

- open　open 方法的语法是 open(method,url[,async,username,password])。它用来创建到指定 URL 的连接（通过指定的连接方法）。可选的参数有 3 个，async 参数用来将请求设置成异步模式（true，默认值）或同步（false），username 和 password 用来指定服务器所需的用户名和密码。
- setRequestHeader　setRequestHeader 方法的语法是 setRequestHeader (label,value)。该方法为请求的头添加标记/值对。
- send　send 方法的语法是 send(content)。它是 XMLHttpRequest 对象的核心方法。它用来发送该请求，同时附上相应的数据。
- getAllResponseHeaders　getAllResponseHeaders 方法的语法是 getAllResponse Headers()。它将以字符串形式返回所有 HTTP 应答的头，其中包含激活超时值、内容类型、服务相关信息和日期信息。
- getResponseHeader　getResponseHeader 方法的语法是 getResponseHeader (label)。它将返回应答头中指定的信息。
- abort　其语法为 abort()，用来取消当前请求。

示例 14.1 中将对全局变量 xmlHttpObj 进行检测，判断其是否有值。如果没有则调用 getXmlHttp 函数，并将其返回值赋给 xmlHttpObj。它将多次检测该值，以确保请求未失效（也就是该应用环境支持 Ajax）。当其获得一个 XMLHttpRequest 对象的实例之后，就将着手处理 Ajax 请求的其他部分。

在前面讲到 XMLHttpRequest.oprn 方法时，我曾经说过第一个参数用来指定调用服务时用的请求方法。它与在传统的客户端/服务器端应用程序中使用的 HTTP 请求方法相同。Ajax 中最常用的方法是 POST 和 GET，而其他方法也是支持的，包括：

- GET　从服务器端请求数据；
- POST　向服务器端传送数据；
- DELETE　删除指定资源中的数据；

- PUT　将数据保存到指定资源上；
- HEAD　与 GET 相似，但服务器端应答只是头（用于获取信息）。

虽然 XMLHttpRequest 对象支持不同的请求方法，但 HTTP 服务器并不一定都能支持，因此 Ajax 应用程序主要只用 POST 和 GET 方法。

什么时候该用 POST，什么时候该用 GET 呢？正确的回答是使用 GET 来获取信息，使用 POST 来向服务器端传送数据。这就是著名的 RESTful 解决方案，因为它是 REST（Representational State Transfer，具象状态传输）的本质基础，绝大数 Web 服务都能支持。

不过，为了限制对服务的访问并为请求提供更高的安全保障，有些 Web 服务器要求你通过 POST 方法来查询数据。另外，POST 方法所支持的数据容量要比 GET 方法大，因此通过 GET 获取数据、POST 提交数据并不是铁的原则。

提示

对于 GET 和 POST 方法没有特定的规则，但你应该尽可能遵循最佳实践的做法，也就是使用 GET 方法访问数据，使用 POST 提交数据。

如果想揭开 XmlHttpRequest 神秘的面纱，不妨将基于 Ajax 和 XMLHttpRequest 的请求视为与传统表单提交类似的功能，它们之间的区别就是 Web 页面无须重新载入。

在示例 14.1 中，发出的请求是 GET 类型，因此 Web 页面的 URL 后面必须跟上所需的相关参数。如果该请求是 POST 类型，那么应该将 send 方法改成：

```
// 准备并发送 XHR 请求
function populateList() {
   var state = document.getElementById("stateList").value;
   var url = 'ch14-02.php';
   var qry = "state=" + state;

   //如果 xmlHttpObj 为空
   if (!xmlHttpObj)
      xmlHttpObj = getXmlHttp();
   if (!xmlHttpObj) return;

   xmlhttp.open('POST', url, true);
   xmlhttp.onreadystatechange = getCities;
   xmlhttp.setRequestHeader("Content-type",
"application/x-www-form-urlencoded");
   xmlhttp.send(qry);
}
```

content-type 属性要改成 urlencoded 形式，然后创建好查询字符串并通过 send 操作发送出去。除了这些修改之外，该方法和 GET 类型的 Ajax 调用是一样的。

XMLHttpRequest 对象除了提供 6 个方法之外，还提供了 6 个属性，其内容和含义如表 14.1 所示。

表 14.1　XMLHttpRequest 对象的属性

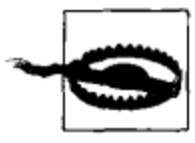
属　性	用　途
onreadystatechange	用来保存当请求的 ready 状态改变时调用的函数
readyState	有 5 个可选值：0 表示请求未初始化，1 表示请求处于 open 阶段，2 表示请求已发送，3 表示正在接收应答，4 表示应答已经收到。在大部分时候，我们感兴趣的是 readState=4 的情况
responseText	文本格式的应答信息
responseXML	XML 格式的应答信息，可以将其视为有效 XML 处理
status	返回请求的状态，如 404、500，或者是最希望的 200（表示一切正常）
statusText	以文字表示的请求状态

示例 14.1 将从一个下拉选择框中获取 Web 请求所需的数据，这意味着你可以控制要发送的数据。不过不应该期望 Web 页面的访问者能够确保数据是有效、安全的。

接下来的修改用来确保查询更加安全、“清洁”，包括请求的 URL 和查询字符串部分的内容。为了添加这一安全措施，可以对 populateList 方法进行修改，使用 encodeURIComponet 方法对表单中返回的值进行相应的处理：

```
// 准备并发送 XHR 请求
function populateList() {
   var state =
encodeURIComponent(document.getElementById("stateList").value);
   var url = 'ch14-02.php?state=' + state;

   // 如果 xmlHttpObj 为空
   if (!xmlHttpObj)
      xmlHttpObj = getXmlHttp();
   if (!xmlHttpObj) return;

   xmlHttpObj.open('GET', url, true);
   xmlHttpObj.onreadystatechange = getCities;
   xmlHttpObj.send(null);
}
```

encodeURIComponent 方法用来确保对相应的文本进行安全的“转义”，使其不会带入可能导致服务器出问题的 HTML 代码。



警告

在客户端使用 encodeURIComponent 方法仅完成了该任务的一部分，查询字符串也可能对服务器端应用程序产生破坏，例如在访问某 Web 服务的 Ajax 调用可能会调用其他 Web 服务。

现在你已经完成请求发送了，但当应答返回时该如何处理呢？

14.4 处理 Web 请求的应答

在发送了一个 Ajax 请求之后，应用程序中的下一个组件就将以某种形式处理请求的应答结果。

14.4.1 检查 Ajax 请求的 readyState 和 status 值

当 Ajax 模式的 Web 请求准备好之后，在应用程序中还需对一个名为 onreadystatechange 的属性进行设置。该属性的值应该设置成为回调函数的名称，这个回调函数将在该 XMLHttpRequest 对象的状态发生改变时被调用：

```
xmlHttpObj.onreadystatechange = getCities;
```

这个回调函数和之前看到过的一个事件句柄函数十分类似。onreadystatechange 与之前章节中看到的事件句柄函数的最大区别在于调用时机，它是否调用取决于服务器，而不属于你直接控制的范畴。对于 Ajax 请求而言，触发其的事件是 XMLHttpRequest 对象的 ready state 发生了改变。

对于示例 14.1 而言，在回调函数 getCities 中需要检查两个 XMLHttpRequest 属性，这必须放在所有功能实现之前完成：

```
if(xmlHttpObj.readyState == 4 && xmlHttpObj.status == 200) {...}
```

正如表 14.1 中所说的那样，readyState 有 5 个可选值，但我们感兴趣的只有 4 这个值，它表示请求已完成。所有的 Web 请求最终都将进入该状态，但并不是所有请求都是成功的，这就是为什么还需要对 status 属性进行检查。例如，如果 Web 服务页面突然被删除了，那么 state 的值将是 404 而不是预期的 200。你可以将请求的服务器端页面移到其他位置并改成其他名称，然后再运行这个 Ajax 应用程序就能看到这一结果。图 14.3 展示了当服务器端页面被移掉之后，在 Firefox 中（同时打开了 Firebug）载入该页面的效果。

图 14.3 当服务器端应用程序被移掉后在 Firebug 中显示的网络状态

由于在应用程序执行失败时，还需要向 Web 页面读者提供一些消息，因此将通过第二个条件语句来检查是否 readyState 值虽然为 4，但 status 值却不等于 200：

```
} else if (xmlHttpObj.readyState == 4 && xmlHttpObj.status != 200) {
```

如果该条件满足，那么你可以向应用程序的用户做出相应提示，虽然他们可能还认为一切正常。如果请求是成功的，那么接下来就是对 Web 服务请求的应答进行处理。

14.4.2 处理 Web 请求应答

很多 Ajax 调用都将收到应答数据，这些应答都将被整合到页面中去。在示例 14.1 和 14.2 中，应答的数据都将以 HTML 片段的形式组织，你只需将其复制到 Web 页面中即可。如果对你使用的 Web 服务信任，那么这是最简单的开发方法。

提示

并非所有的 Ajax 请求都将返回文本数据，但绝大多数情况都是如此。如果应答不是文本数据，那么 Web 服务将返回一个状态，通过对它的解析可以确保 Ajax 请求已成功处理。在第 15 章中，我们将演示如何对 XML 或 JSON（JavaScript 对象符号）格式的应答数据进行处理。

由于这里的应答是文本数据，因此需要通过 XMLHttpRequest 对象的 responseText 属性来获取，然后将其复制到页面中即可：

```
document.getElementById('cities').innerHTML = "<select>" +
xmlHttpObj.responseText + "</select>";
```

同样，这在 Ajax 中也不算复杂的。你甚至可以观察服务器端和客户端的通信过程，只要你的浏览器中的调试工具支持这样的操作。

示例 14.3 中包含一些曾经在示例 14.1 中出现过的代码，只不过按照本小节中讨论的内容（更进一步研究了 Ajax 处理过程）进行了一些修改。

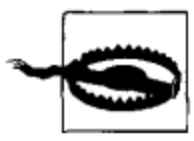
示例 14.3　“Hello Ajax World”应用程序修改版

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>City lookup</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<style type="text/css">
div.elem
{
   margin: 20px;
}
div#cities
{
  display: none;
```

```
}
</style>
<script type="text/javascript">
//<![CDATA[

// 全局变量
var xmlHttpObj;

function catchEvent(eventObj, event, eventHandler) {
   if (eventObj.addEventListener) {
      eventObj.addEventListener(event, eventHandler,false);
    } else if (eventObj.attachEvent) {
      event = "on" + event;
      eventObj.attachEvent(event, eventHandler);
    }
}

catchEvent(window,"load", function() {
   document.getElementById("cities").style.display="block";
   document.getElementById("submitButton").style.display="none";
   document.getElementById("stateList").onchange=populateList;
});
function getXmlHttp() {
   var xmlhttp = null;
   if (window.XMLHttpRequest) {
      xmlhttp = new XMLHttpRequest();
   } else {
      try
      {
         xmlhttp = new ActiveXObject("Msxml2.XMLHTTP");
      }
      catch (e) {
         try
         {
            xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");
         }
         catch (e) {
            return null;
         }
      }
   }
  return xmlhttp;
}

// 准备并发送XHR请求
function populateList() {
   var state =
encodeURIComponent(document.getElementById("stateList").value);
   var url = 'ch14-02.php?state=' + state;
```

```
   //如果 xmlHttpObj 为空
   if (!xmlHttpObj)
      xmlHttpObj = getXmlHttp();
   if (!xmlHttpObj) return;

   xmlHttpObj.open('GET', url, true);
   xmlHttpObj.onreadystatechange = getCities;
   xmlHttpObj.send(null);

}

// 处理应答
function getCities() {
   if(xmlHttpObj.readyState == 4 && xmlHttpObj.status == 200) {
      document.getElementById('cities').innerHTML =
xmlHttpObj.responseText;
   } else if (xmlHttpObj.readyState == 4 && xmlHttpObj.status != 200) {
      document.getElementById('cities').innerHTML = 'Error: preSearch Failed!';
   }
}
//]]>
</script>
</head>
<body>
<h3>Select State:</h3>
<form action="ch14-02.php" method="get">
<div class="elem">
<select id="stateList" name="state">
<option value="CA">California</option>
<option value="MO">Missouri</option>
<option value="WA">Washington</option>
<option value="ID">Idaho</option>
</select>
<p><input type="submit" value="Get Cities" id="submitButton" /></p>
</div>
<div class="elem" id="cities">
<p> </p>
</div>
</form>
</body>
</html>
```

当然，Ajax 并不仅仅只是代码，在接下来的小节中我们将说明这点。



警告

如果你想在本机尝试运行一个 Ajax 应用程序，那么很可能会遭遇安全限制。像 Firefox 这样的浏览器是不允许在本地文件系统上使用 XMLHttpRequest 的，这是为了防止恶意的 Ajax 应用程序访问本地文件系统。

14.5 Ajax：不仅是代码

Ajax 功能实际十分简单，你可能感到奇怪，这对老式的客户端/服务器端模式有什么影响。不过，Ajax 的内容远远超出前面这个例子中所看到的东西。

14.5.1 Ajax 的动态特性

只要你愿意，通过 Ajax 和其他基于 JavaScript 的功能，加上 Web 页面访问者的操作，甚至可以将整个网站放在一个页面上。但这样做的问题在于要为网站的内容重新创建一个视图将变得更加困难。

Ajax 和所有动态网页功能相似，无法创建出持久性的页面效果。在每次页面载入或用户完成一组操作之后，这些效果都将被重新创建。通过 URL 是无法访问到这些中间效果的，并且可能也无法打印出来。对于每个部分无法提供持久性的链接，同时 Web 页面访问者也无法将其添加到收藏夹中，除非特意写一些代码来解决这些问题。最重要的是，当 Web 页面访问者单击“后退”按钮时，它并不会回退到 Ajax/DHTML 的上一次操作上，而是彻底离开该页面。

有些完整的框架可以解决这些问题，它们采用了如为 Ajax 调用序列各添加一个<a>标签以示区别的方法，在本章最后一个小节中我将介绍一些用于解决该问题的 Ajax 程序库。不过在你研究这些之前，应该考虑一下这方面的功能是不是必需的。同时也必须考虑 Ajax 应用对 Web 页面的可访问性的影响。

14.5.2 Ajax 的可访问性和适度降格

Ajax 和各种动态 Web 页面效果类似，对于存在视力障碍的访问者而言，应用程序的可访问性会受到影响。例如，当你动态地修改了页面内容时，对那些通过语音朗读访问网站的用户而言这是不可见的。

关于 Ajax 和可访问性的话题，在 http://www.webaim.org/techniques/ajax/中很多不错的信息。除了对该问题的讨论之外，它还提供了一些其他网站的链接，上面也介绍了一些相关的信息。

此外，出于安全及其他理由，有些人会在浏览器中禁用 JavaScript。在之前的章节中，我们已经讨论了提供适度降格方案的重要性，但对于 Ajax 而言，最好的方法是当 JavaScript 被禁用时对其适度降格，以提供传统的客户端/服务器端模式的功能。

虽然在页面载入时，示例 14.1 中的表单不会显示出来，它还包含一个提交按钮，而该表单提交时将调用 Web 服务应用程序。但是当该页面载入时，会通过 JavaScript 隐藏该按钮。

但是如果在浏览器中禁用了 JavaScript 或者根本不支持 JavaScript，那么该按钮就会显

示出来，如图 14.4 所示，用户单击这个提交按钮将发出一个传统模式的 Web 服务请求，从而跳转到另一个页面中，在该页面中将显示出应答的结果，如图 14.5 所示。

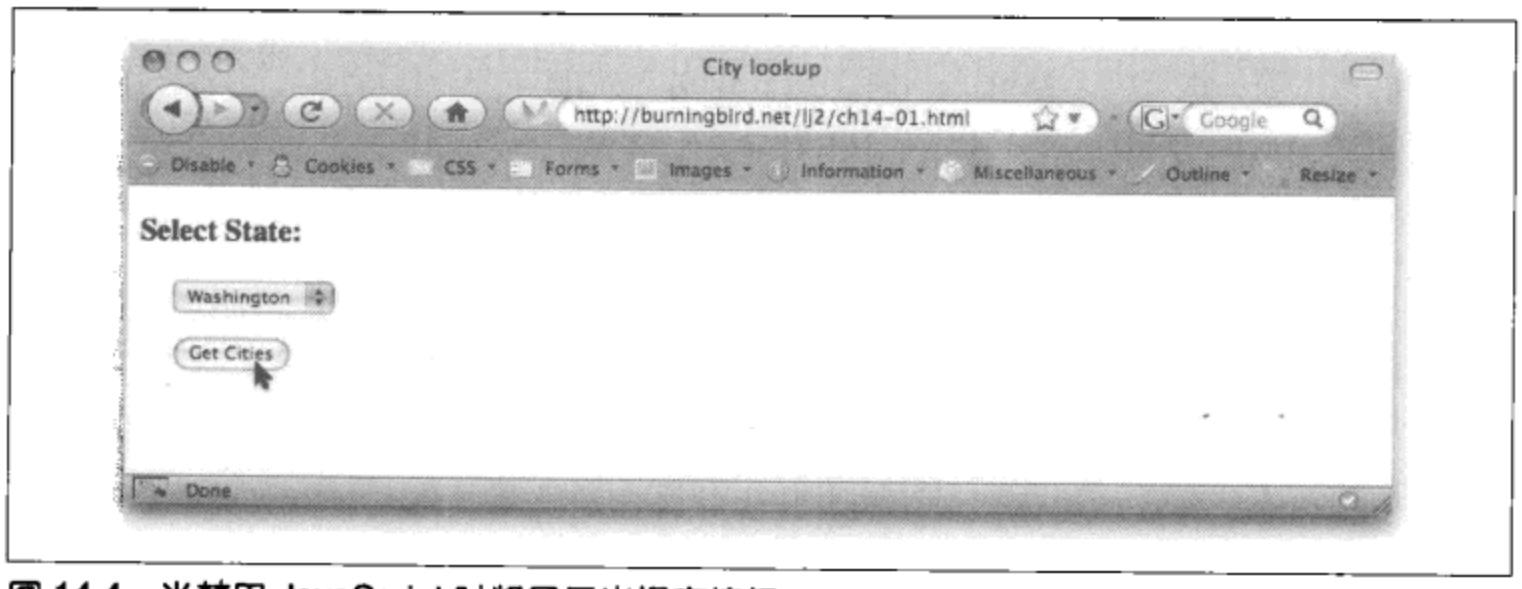


图 14.4　当禁用 JavaScript 时将显示出提交按钮

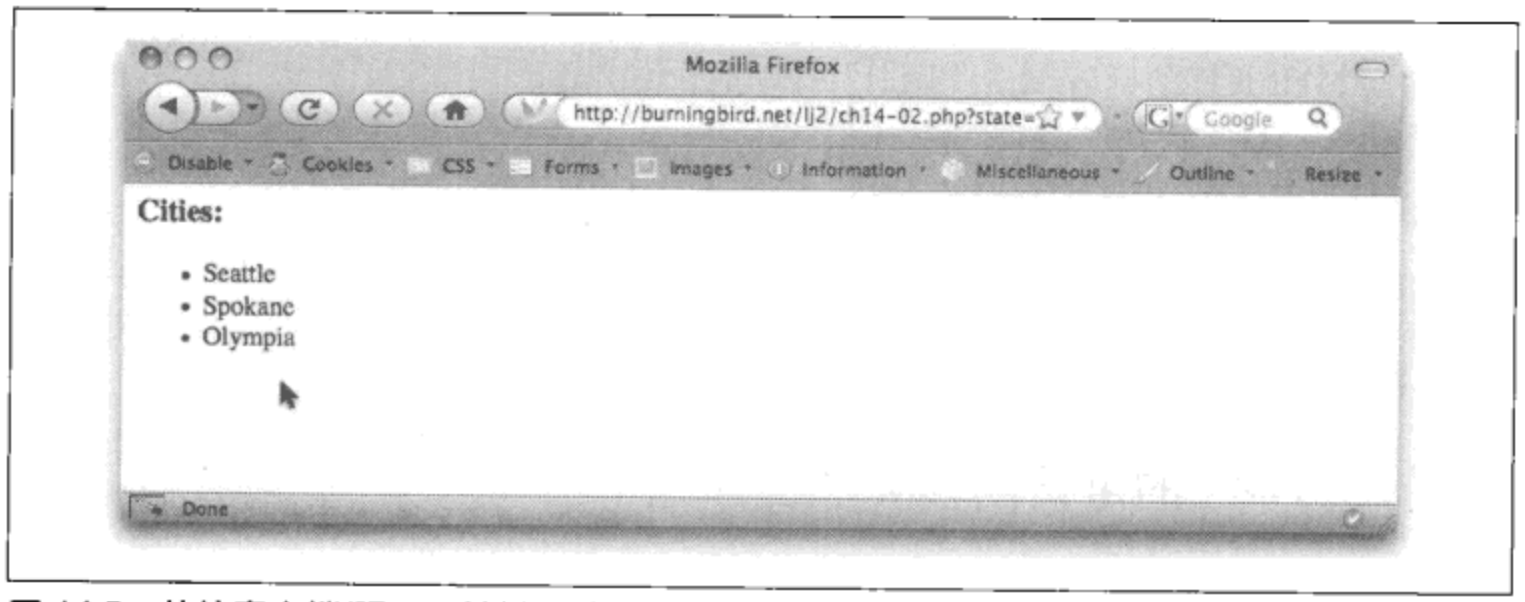


图 14.5　传统客户端/服务器端模式的请求所产生的应答

不过，适度降格方案的关键不仅仅在客户端。只要可能发出更传统的 Web 服务请求，服务器端都将以格式化数据予以应答，即使该页面是直接访问的也将“看起来一切正常”。同样，在第 15 章中我将对其做更彻底的演示。

14.5.3　安全和工作区

Ajax 如此流行的原因之一在于使用它很安全，它和绝大数 Web 应用程序（需要许多相同的安全措施）一样安全。它的安全保障来源于 JavaScript 沙箱以及它对 XMLHttpRequest 的影响。

在本章的示例中，服务器端页面与向服务器发送请求的页面处于相同的服务器、相同的域中。如果你尝试将服务器放在另一个域中，就会出错。为什么呢？因为 Ajax 操作也遵循 JavaScript 中相同源、相同域的规则：在 Web 页面中只能调用相同服务器（相

同域）的 Web 服务。

在 IE 中，可以通过设置使请求能访问其他域的服务，但其他浏览器都不允许。Firefox 只支持使用数字签名的脚本实现跨域操作，但其他浏览器连这都不支持。这意味着你要么将页面访问的 Web 服务限制在特定的域中，要么查找一个工作区。

其中一个方法是通过代理。如果在 Web 服务器上安装了代理软件，那么对服务器的调用将通过代理发送，由代理完成相应的请求分发工作。

通过 Web 代理能够实现什么呢？在示例 14.1 和 14.2 中，如果负责查询城市信息的 Web 服务来自于不同的域（例如 getcitiesperstate.com），那么根据同域安全策略，调用的服务只能位于自己域中的页面，而这时我们却需要发出一个跨域 Web 服务请求。但跨越限制对服务器端应用程序而言是没有影响的。因此可以由服务器端程序发出该请求，当请求的数据返回时，服务器端程序再将其返回给客户端页面即可。

对于 Ajax 程序而言，该 Web 服务请求是它发送的，并且是由本地的 Web 服务执行的。而我们并不需关心本地 Web 服务实际上是通过访问远程 Web 服务来实现的。

在如 Google 和 Yahoo!所提供的 Web 服务中，它将 Web 服务请求封装在 script 标签中，而不是使用 XMLHttpRequest。下面就是一个使用该方法的示例：

```
<script src="http://maps.google.com/maps?file=api&amp;v=2&amp;key=yourkey"
type="text/javascript">
</script>
```

另外，你可以让 Web 服务器重写一个 Web 请求，并将请求重定向到另一个位置上。这需要使用 Apache 服务器的 mod_rewrite 服务，或者是其他 Web 服务器的其他服务，不过绝大多数 Web 服务器都提供了该功能。

提示

如果你想在自己的网站上添加 Ajax 功能，可以阅读我写的一本名为 *Adding Ajax* 的书，它也是 O'Reilly 出版社出版的。

14.6 JavaScript 和 Ajax 程序库

本章的示例都是直接将 JavaScript 脚本放在 Web 页面上的，这是为了更易于阅读和修改。不过正如之前的章节中所说的那样，通常应该将 JavaScript 代码放在一个单独的文件中，这样就可以在需要使用这些 JavaScript 功能的 Web 页面中引用它们。

同样，我们前面只考虑如何直接使用 JavaScript 完成任务，并没有借助于任何优秀的 JavaScript 程序库，它们对于开发人员而言是免费的。毕竟本书是教你 JavaScript 的使用方法，而不是 Prototype、Dojo 或 jQuery。

但在实际的应用程序开发时，特别是要添加这些 Ajax 功能时，应该使用这些程序库。

这些程序库中的代码是经过严格测试的，也做过了性能调优，可以使你的工作变得更加简单。即使你很喜欢 JavaScript 开发，也应该尽可能使用经过良好测试的、已开发完成的 JavaScript 程序库。

例如，在优秀的 Lightbox 程序库（现在的版本是 Lightbox2，你可以在 http://www.huddletogether.com/projects/lightbox2 下载到它）发布之前，我一直使用着自己编写的 JavaScript 程序库 photo-enlarging。

如果我想创建一个复杂的 Ajax 应用，肯定会使用一些外部的程序库以便简化自己的工作。以下就是一些应用广泛、成熟、健壮的 Ajax/JavaScript 程序库：

- Prototype　它被视为 Ajax 程序库的老祖宗。在 http://prototypejs.org/中可以下载到 Prototype 程序库，还能够找到相关的文档。
- Dojo　最完整、博学的 Ajax/JavaScript 程序库，提供了许多能够生成精美图形效果的功能。在 http://dojotoolkit.org/中可以下载到 Dojo 程序库和相关的文档。
- jQuery　这或许算得上是应用最广泛的 JavaScript 程序库，它在大多数主流的内容管理系统中广为应用，包括 WordPress 和 Drupal。在 http://jquery.com/中可以下载到 jQuery 程序库和相关的文档。

此外还有许多可用的程序库，但这 3 个算得上是 Ajax 领域的领军者。

如何使用这些程序库呢？在此我们主要介绍 jQuery，它是我最为熟悉的程序库，你可以像引用自己的 JavaScript 文件那样将其脚本引用到自己的 Web 页面上：

```
<script type="text/javascript" src="path/to/jquery.js"></script>
<script type="text/javascript">
     // 你自己的应用程序
 </script>
```

使用第三方 JavaScript 程序库和复用自己的代码类似，只是在两个方面需要特别注意：使用传统的事件句柄和全局变量时。

对全局变量的使用带来的影响是很显然的，如果你使用的全局变量的名称和第三方程序库中的全局变量相同，那么在你程序代码中定义的全局变量就将覆盖程序库中定义的全局变量，这时应用程序就可能出错。而与传统事件句柄相关的问题并不太明显。不明显的原因是当遇到怪异的行为时，它将令你陷入难以忍受的调试中。

如果你使用了如 window.onload 之类的传统事件句柄，那么就可能覆盖第三方程序库中基于 window.onload 事件句柄的功能。例如，如果程序库为 window.onload 事件添加了相应的句柄，而你又直接对其做了赋值：

```
window.onload=eventHandlerFunction;
```

那么你的代码就将“擦除”在第三方程序库中所指定的事件句柄。

你最好避免使用之前章节中所介绍的 DOM 2 事件句柄。不过，与其自己编写程序来实现所需的功能，还不如直接使用这些程序库中提供的功能。例如，如果你想获取所有 div 元素的 click 事件，那么使用 jQuery 只需采用以下形式：

```
$("div").click(clickEventHandlerFunction);
```

代码中的$是 Ajax 技术中很常用，它用来返回一个或一组符合指定标准的页面元素。在 jQuery 中，它所使用的是 CSS 选择语法。例如，根据该语法规则，如果要获得页面中所有的<p>元素，可以使用如下代码：

```
$("p")...
```

如果你想获得某个元素，可以使用一个元素标识符，并在前面添加一些细节信息，就像在 CSS 中那样：

```
$("#p1")...
```

如果想获得指定 CSS 类的所有元素，则可以使用：

```
$(".p1")...
```

这样更好，不是吗？

想了解 jQuery 最基本的工作原理，可以通过 jQuery 网站的指南中提供的名为“Hello jQuery”的应用程序。将其代码从指南中复制出来，然后为其创建一个单独的 Web 应用程序即可，其代码如示例 14.4 所示。

示例 14.4　使用 jQuery 创建的“Hello Ajax World”应用程序

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>Hello, World, jQuery style!</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />

<script type="text/javascript" src="jquery.js"></script>
<script type="text/javascript">
//<![CDATA[

$(document).ready(function() {
   $("#p1").click(function() {
     alert("Hello world!");
   });
 });

//]]>
</script>
</head>
<body>
```

```
<p id="p1" style="background-color: #ff0; padding: 10px; width: 150px">Click
to say
"Hi!"</p>
</body>
</html>
```

如果你自己手动创建该示例，则要留意一些问题。在 jQuery 中，作为 onload 事件参数传入的是 document 对象，而非 window 对象。

即使在这个非常简单的例子中，你也会看到第三方程序库是如何简化了代码、节省了开发时间的，如果你将它和第 1 章中的例子比较，就会更明显地体会到这一点。

在你开始使用其他人的代码之前，你还需要完成 JavaScript 学习之旅。不过本书的内容很快就要结束，只剩下最后一章了。

14.7 知识测验

1. XMLHttpRequest 请求也可以设置成同步模式（也就是停下来等待应答），虽然这看起来和 Ajax 的概念有些背道而驰。但你记得应该如何发出这样的请求吗？
2. 当请求接收到应答时，就需要对其进行处理。那么如何指定一个在收到服务应答时调用的函数呢？
3. 对于 Ajax 请求而言，表示成功需要对哪两个状态进行判断？表示请求完成呢？
4. 在你能下载到的本书源文件包中，有一个单独的名为 drink 的子目录。在这个子目录中有一个简单示例（名为 drink recipe，最早出现于我写的 *Adding Ajax* 这本书中）的 3 个版本，但都是完整的。

在该应用程序的第一个版本中，饮料类型是通过下拉选择框选择的，同时它将触发一个发往 recipe.php 的 Ajax 请求。所选的饮料名称将通过 Ajax 请求发出。这个 Web 服务请求的代码应该如何写？假定保存饮料名称的变量名为 drink，XMLHttpRequest 对象名为 xmlhttp，处理 Ajax 请求应答的函数名为 printRecipe。

5. 对于上面这个例子，将 HTML 格式的应答添加到名为 recipe 的 div 元素中可以采用什么方法？

14.8 测验答案

1. XMLHttpRequest.open 函数的第三个、可选的参数是一个布尔值。将该参数设置为 true（默认）将创建一个异步请求；如果将其设置成 false 则将创建一个同步请求。
2. 当获得 XMLHttpRequest 对象的引用并调用了其 open 方法之后，将 onReadyStateChange 属性值赋为回调函数的名称。

3. XMLHttpRequest 对象的 readyState 属性值为 4 就表示请求完成，要表明服务请求是成功的，还必须确保其 status 属性值为 200。

4. 准备该 Web 请求的其中一种方法是，从第一个示例中获取 JavaScript 代码，包括 drink1.html 和其相关的 JavaScript 文件 getdrink2.js。注意，对于输入项需要通过 encodeURIComponet 方法进行格式化，然后将饮料类型附在这个 Web 服务请求的 URL 中：

```
var drink =
encodeURIComponent(document.getElementById('drink').value);
var qry = "drink=" + drink;
var url = 'recipe.php?' + qry;
xmlhttp.open('GET', url, true);
xmlhttp.onreadystatechange = printRecipe;
xmlhttp.send(null);
```

5. 虽然 drink1.html 中的示例使用了一个很复杂的过程，但还有一种简单的方法，可以直接使用 innerHTML 属性将饮料配方添加到元素中：

```
if(xmlhttp.readyState == 4 && xmlhttp.status == 200) {
   var recipe = document.getElementById("recipe");
   recipe.innerHTML = xmlhttp.responseText;
}
```

第 15 章

Ajax 数据：XML 或 JSON

在第 14 章，我们分析了一个简单的 Ajax 应用程序，它向一个 Web 服务发出申请，应答是以 HTML 片段的形式返回的。然后我们通过元素的 innerHTML 属性将该片段复制到 Web 页面中。尽管这是一种有效的方法，但也有一些限制，特别是希望对返回的应答数据进行一些处理时。对于 Ajax 调用所返回的应答数据而言，有两种处理的方法比较流行：一种是 XML，另一种就是 JSON（JavaScript 对象符号）。

本章将说明如何修改第 14 章中的应用程序，使其以 XML 或 JSON 的格式返回应答数据，并且讨论何时应该采用哪一个。如果基于应答的数据创建第二个下拉选择框，并将所有城市信息作为选项放入，这肯定要比直接将其作为列表显示在页面上强。

15.1 XML 格式的 Ajax 应答

将应答以 XML 格式表示，其好处之一就是要比简单的字符串或 HTML 片段更加多元。另外，你还可以像对待 Web 页面元素那样，直接通过 DOM 方法来操作 Ajax 调用返回的 XML 格式应答。

15.1.1 数据的 MIME 类型

使用 XML 也会带来一定的负担。例如，在第 14 章中的示例 14.1 中，服务器应答是以 html/text 格式返回的，应用程序可以通过 XMLHttpRequest 的 responseText 属性获取应答信息。而当使用 XML 格式时，服务器端应用程序返回的应答内容的 MIME 类型是 text/xml，因此我们最终需要通过 XMLHttpRequest 对象的 responseXML 容器来获取应答信息。

如果无法确保服务器返回正确的 MIME 类型，可以考虑使用 XMLHttpRequest 对象中提供的 overrideMimeType 方法，它可以对服务器应答的内容类型进行修改。如果返回

的应答是 XML 格式的，你想以 XML 格式对其进行处理，但服务器端应用程序没有返回正确的内容类型，则可以使用 XMLHttpRequest.overrideMimeType 方法对其进行修改：

```
if (window.XMLHttpRequest) {
   xmlhttp = new XMLHttpRequest();
   if (xmlhttp.overrideMimeType) {
      xmlhttp.overrideMimeType('text/xml');
   }
}
```

但 overrideMimeType 也有一个缺点，它并不是所有浏览器都支持的，IE 和 Opera 都不支持它。对 Opera 而言这不是问题，因为它一定会返回 responseXML。不过考虑到 IE 的限制，最好的方法还是确保应用程序获得格式正确的数据，如果有可能，就应确保服务器端应用程序在返回数据时设置了正确的内容类型。

15.1.2 在服务器端生成 XML 数据

除了使用正确的内容类型之外，Web 服务应用程序还应确保生成的是有效的 XML，它应该有一个封装所有其他数据的根元素。示例 15.1 中的程序将根据 Web 请求中传入的“州”，以 XML 格式返回相应的“城市”信息。注意，对于每个“城市”都有两个元素：value 和 title。value 值是作为下拉选择框的选项使用的，而 title 则是用来在页面上显示的。

示例 15.1 基于 PHP 实现的 Ajax 程序，现在将返回 XML 格式的应答

```
<?php

// 如果未传入查询字符串，那将无法进行查询
if(empty($_REQUEST['state'])) {
    echo "<city>No State Sent</city>";
} else {
    // 去除传入的查询字符串最前面和最后面的空白符
    $search = trim($_REQUEST['state']);
    switch($search) {
      case "MO" :
         $result = "<city><value>stlou</value><title>St.
Louis</title></city>" .
                   "<city><value>kc</value><title>Kansas
City</title></city>";
         break;
      case "WA" :
         $result = "<city><value>seattle</value><title>
Seattle</title></city>" .
                   "<city><value>spokane</value><title>
Spokane</title></city>" .
                   "<city><value>olympia</value><title>
Olympia</title></city>";
         break;
      case "CA" :
```

```
        $result = "<city><value>sanfran</value><title>San
Francisco</title></city>" .
                  "<city><value>la</value><title>Los
Angeles</title></city>" .
                   "<city><value>web2</value><title>Web 2.0
City</title></city>" .

                   "<city><value>barcamp></value><title>BarCamp</title></city>";
        break;
      case "ID" :
        $result = "<city><value>boise</value><title>Boise</title></city>";
        break;
      default :
        $result = "<city><value></value><title>No Cities
Found</title></city>";
        break;
      }
     $result ='<?xml version="1.0" encoding="UTF-8" ?>' .
             "<cities>" . $result . "</cities>";

   header("Content-Type: text/xml; charset=utf-8");

    echo $result;
}
?>
```

在示例 15.1 中是手动生成 XML 的，不过用于创建 Web 服务的编程语言（如 PHP）大多都提供了一些程序库，通过它们生成 XML 数据是很容易的。例如，在 PHP 5.x 及后续版本中都提供了 DOM XML 功能，其使用的方法和本书之前章节中所介绍的方法类似。

示例 15.2 展示了一个能实现与示例 15.1 相同功能的 PHP 页面，不过在此使用了 DOM。

示例 15.2　在示例 15.1 的基础上改成使用 PHP DOM

```
<?php
// 如果未传入查询字符串，那将无法进行查询
if(empty($_REQUEST['state'])) {
    echo "<city>No State Sent</city>";
} else {
    // 去除传入的查询字符串最前面和最后面的空白符
    $search = trim($_REQUEST['state']);
    $cities = array();
    switch($search) {
      case "MO" :
        $cities [] = array(
                'value' => 'kc',
                 'title' => "Kansas City");
        $cities [] = array(
                 'value' => 'stlou',
                 'title' => "St. Louis");
```

```
    break;
case "WA" :
    $cities [] = array(
            'value' => 'seattle',
             'title' => "Seattle");
    $cities [] = array(
             'value' => 'spokane',
             'title' => "Spokane");
    $cities [] = array(
             'value' => 'olympia',
             'title' => "Olympia");
    break;
case "CA" :
    $cities [] = array(
            'value' => 'sanfran',
             'title' => "San Francisco");
    $cities [] = array(
             'value' => 'la',
             'title' => "Los Angeles");
    $cities [] = array(
             'value' => 'web2',
             'title' => "Web 2.0 City");
    break;
case "ID" :
    $cities [] = array(
            'value' => 'boise',
             'title' => "Boise");
        break;
    default :
        $cities [] = array(
                'value' => '',
                 'title' => "No Cities Found");
        break;
    }

  header("Content-Type: text/xml; charset=utf-8");

}

  $doc = new DOMDocument();
  $doc->formatOutput = true;

  $r = $doc->createElement("cities");
  $doc->appendChild( $r );

  foreach ($cities as $city) {
    $c = $doc->createElement( "city" );
```

```
    $value  = $doc->createElement( "value" );
    $value->appendChild(
      $doc->createTextNode($city['value']));
    $c->appendChild( $value );

    $title = $doc->createElement( "title" );
    $title->appendChild(
      $doc->createTextNode($city['title']));
    $c->appendChild( $title );
    $r->appendChild( $c );
}
echo $doc->saveXML();
?>
```

虽然示例 15.2 看起来比示例 15.1 长得多，但如果采用示例 15.1 中使用的方法，那么要处理从数据库查询语句中返回的一打（甚至是上百条）记录时，复杂性就会大大增加。

提示

如 MySQL 之类的数据库也能以 XML 格式返回 SQL 查询结果，这样就能避开这一复杂的过程。

15.1.3 在客户端处理 XML 数据

服务器端应用程序写完后，接下来还需要修改用 JavaScript 开发的客户端应用程序，示例 15.3 中的代码就是在第 14 章中示例 14.1 的基础上演变而来的，它将对 HTML 片段的处理部分改成处理 XML 数据的代码。特别需要注意的是从 responseXML 属性获取数据的部分。

示例 15.3 修改客户端应用程序，以完成对 XML 格式应答的处理

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>City lookup</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<style type="text/css">
div.elem
{
   margin: 20px;
}
div#cities
{
  display: none;
}
</style>
<script type="text/javascript">
//<![CDATA[
```

```
// 全局变量
var xmlHttpObj;

function catchEvent(eventObj, event, eventHandler) {
   if (eventObj.addEventListener) {
      eventObj.addEventListener(event, eventHandler,false);
    } else if (eventObj.attachEvent) {
      event = "on" + event;
      eventObj.attachEvent(event, eventHandler);
    }
}

catchEvent(window,"load", function() {
   document.getElementById("cities").style.display="block";
   document.getElementById("submitButton").style.display="none";
   document.getElementById("stateList").onchange=populateList;
});

// 获取 XMLHttpRequest 对象
function getXmlHttp() {
   var xmlhttp = null;
     if (window.XMLHttpRequest) {
      xmlhttp = new XMLHttpRequest();

      // XML MIME 类型
      if (xmlhttp.overrideMimeType) {
         xmlhttp.overrideMimeType('text/xml');
      }
   } else {
      try
        {
           xmlhttp = new ActiveXObject("Msxml2.XMLHTTP");
        }
        catch (e) {
           try
           {
              xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");
           }
           catch (e) {
              return null;
           }
        }
   }
   return xmlhttp;
}

// 生成服务请求
function populateList() {
```

```
   var state =
encodeURIComponent(document.getElementById("stateList").value);
   var url = 'ch15-01.php?state=' + state;

   // 如果 xmlHttpObj 的值为空
   if (!xmlHttpObj)
      xmlHttpObj = getXmlHttp();
   if (!xmlHttpObj) return;

   xmlHttpObj.open('GET', url, true);
   xmlHttpObj.onreadystatechange = getCities;
   xmlHttpObj.send(null);

}

// 生成城市信息
function getCities() {
   if(xmlHttpObj.readyState == 4 && xmlHttpObj.status == 200) {
     try {

        // 获取显示城市信息的下拉选择框
        var citySelection = document.getElementById("citySelection");

        // 在外部 XML 文档中也可以使用 DOM 方法
       var citynodes = xmlHttpObj.responseXML.getElementsByTagName('city');

        // 遍历每个 city 节点
        for (var i = 0; i < citynodes.length; i++) {
          var name = value = null;

          // 获取城市的显示名称和选项值
          for (var j = 0; j < citynodes[i].childNodes.length; j++) {
            var elem = citynodes[i].childNodes[j].nodeName;
            var nodevalue =
citynodes[i].childNodes[j].firstChild.nodeValue;
            if (elem == 'value') {
                value = nodevalue;
            } else {
                name = nodevalue;
            }
          }

          // 为下拉列表添加新选项
          citySelection.options[i] = new Option(name,value);
        }
     } catch (e) {
         alert(e.message);
     }
   } else if (xmlHttpObj.readyState == 4 && xmlHttpObj.status != 200) {
```

```
      document.getElementById('cities').innerHTML = 'Error: No Cities!';
   }
}

//]]>
</script>
</head>
<body>
   <h3>Select State:</h3>
   <form action="ch14-02.php" method="get">
      <div class="elem">
         <select id="stateList" name="state">
            <option value="CA">California</option>
            <option value="MO">Missouri</option>
            <option value="WA">Washington</option>
            <option value="ID">Idaho</option>
         </select>
      </div>
   <h3>Cities:</h3>
   <div class="elem" id="cities">
      <select id="citySelection">
         <option> </option>
      </select>
   </div>
   <p>
      <input type="submit" value="Lookup" id="submitButton" />
   </p>
   </form>
</body>
</html>
```

接下来，我们对处理服务器应答的程序做进一步分析。

首先，对通过请求的 responseXML 属性获得的应答 XML 信息调用 DOM 函数 getElementsByTagName。该函数将返回一组子节点，每个子节点对应着 XML 中的一个城市信息。而在每个子节点中又各有两个子节点，一个是选项值，另一个是该选项所显示的标题信息。

应用程序并不会假定返回给 Web 页面的 XML 信息是排好顺序的(value 始终放在前面，title 一定放在后面)，而是遍历 childNodes 的 nodeList，以获得其 nodeName 值。然后检查是不是 value，如果是则将这个 nodeValue 赋给 value，如果不是则将其赋给 tilte。当分析了保存城市信息的 childNodes 之后，将使用 value 和 title 值来为下拉选择框创建一个新选项，然后添加到下拉选择框 city 的 options 数组中，接下来再处理下一个城市信息。

这些代码都放在一个异常处理中，这是因为 DOM 函数可能会抛出一个异常，从而导致浏览器无法完成相应的处理。当使用 Ajax 时，在程序代码中应用异常处理机制是一个

好习惯。

另外还要注意，该表单的 action 属性值设置的是第 14 章中示例 14.2 所展示的 PHP 应用程序。如果浏览器的 JavaScript 被禁用，那么该应用程序会提供一个访问者能够正常阅读的应答，它比直接获取本小节输出的 XML 格式的应答（或者是下一小节中输出的 JSON 格式）要更好些。

采用刚才演示的方法，不管 XML 有几层嵌套，都可以访问到所需的节点。不过这是基于一个假定的，那就是你知道 XML 的微妙之处，它毕竟和 JavaScript 是两种不同的语言。这个问题就引发了大家对一种新格式的兴趣，那就是 JSON。

提示

如果你使用的是属性而不是节点，那么可以通过 DOM 方法的 getAttribute 方法来从 XML 文档中获取相应的值。

15.2 JSON

正如其网站中所说的那样，JSON 是一种“轻量级的数据交换格式”。和一组由逗号分开的字符串，以及处理复杂（成本高昂）的 XML 相比，JSON 是一种很容易将服务器端数据结构转成 JavaScript 对象的数据格式。

提示

JSON 是 Douglas Crockford 发布的，他也是 *JavaScript: The Good Parts*（O' Reilly 出版社出版）一书的作者。JSON 的网站是 http://www.json.org/。

JSON 实际上就是使用 JavaScript 语法定义的对象。一个对象的语法包括一对大括号以及其中的成员：

```
object{ } or object { string : value...}
```

对于数组而言，它是由一组元素加上一对方括号组成的：

```
array[] or array[value,value,...,value]
```

在这个子集中指定的值，和变量及相关值（字符串或数字）遵从相同的规则，这些规则是在 JavaScript ECMA-262 规范第 3 版中定义的。对于这个子集还有一个需要强调的地方，它和 JSON 之间的最主要区别在于 JavaScript 可以用单引号或双引号来引用字符串，而 JSON 只支持双引号。

15.2.1 一个简单的 JSON 应用程序

JSON 和 XML、HTML 一样，也能够手动创建，因为它实际上就是另一种字符串。不过，现在越来越多的用于 Web 服务开发的编程语言都提供了对 JSON API 的支持，而且绝大多数都提供了针对 JSON 编码数据的编码器，可以完成编码和解码

操作。

不过鉴于该示例的设计意图，我们将手动创建该数据结果。示例 15.4 中包含了一个新的服务器端应用程序，它将以 JOSN 格式返回应答信息。这里使用了一个对象数组，每个元素都有 value 和 title 属性。

示例 15.4　在 PHP 脚本中使用简单的 JSON

```
<?php

// 如果未传入查询字符串，那将无法进行查询
if(empty($_REQUEST["state"])) {
    echo "<city>No State Sent</city>";
} else {
    //Remove whitespace from beginning & end of passed search.
    $search = trim($_REQUEST["state"]);
    switch($search) {
      case "MO" :
        $result = '[ { "value" : "stlou", "title" : "St. Louis" }, ' .
                  '{ "value" : "kc", "title" :" Kansas City" } ]';
        break;
      case "WA" :
        $result = '[ { "value" : "seattle", "title" : "Seattle" }, ' .
                  '  { "value" : "spokane", "title" : "Spokane" }, ' .
                  '  { "value" : "olympia", "title" : "Olympia" } ]';
        break;
      case "CA" :
        $result = '[ { "value" : "sanfran", "title" : "San Francisco" },' .
                  '  { "value" : "la",     "title" : "Los Angeles"   },' .
                  '  { "value" : "web2",   "title" : "Web 2.0 City"  }, ' .
                  '  { "value" : "barcamp", "title" : "BarCamp"      } ]';
        break;
      case "ID" :
        $result = '[ { "value" : "boise", "title" : "Boise" } ]';
        break;
      default :
        $result = '[ { "value" : "", "title" : "No Cities Found" } ]';
        break;
    }

    echo $result;
}
?>
```

要在 Web 页面中使用这个数据结构，需要通过 responseText 属性获得这个 JSON 格式的数据，然后将其传给 eval 函数，这个函数在本书中还没有介绍过。eval 函数是 JavaScript 内置的一个函数，你可以将任何字符串传给它，它将对传入的字符串进行“计算”，其结果和将这个字符串作为 JavaScript 代码嵌入到页面中是一样的。

在处理 Web 服务返回的 JSON 格式字符串时，eval 函数将对该数据进行“计算”，并将其结果存储到本地的程序变量中。示例 15.5 就是针对 JSON 数据结构做过调整之后的 Web 页面。

示例 15.5　在服务器端和客户端间使用 JSON 格式数据

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>City lookup</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<style type="text/css">
div.elem
{
   margin: 20px;
}
div#cities
{
  display: none;
}
</style>
<script type="text/javascript">
//<![CDATA[

// 全局变量
var xmlHttpObj;

function catchEvent(eventObj, event, eventHandler) {
   if (eventObj.addEventListener) {
      eventObj.addEventListener(event, eventHandler,false);
    } else if (eventObj.attachEvent) {
      event = "on" + event;
      eventObj.attachEvent(event, eventHandler);
    }
}

catchEvent(window,"load", function() {
   document.getElementById("cities").style.display="block";
   document.getElementById("submitButton").style.display="none";
   document.getElementById("stateList").onchange=populateList;
});
function getXmlHttp() {
   var xmlhttp = null;
   if (window.XMLHttpRequest) {
      xmlhttp = new XMLHttpRequest();
   } else {
      try
        {
```

```
            xmlhttp = new ActiveXObject("Msxml2.XMLHTTP");
        }
        catch (e) {
            try
            {
                xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");
            }
            catch (e) {
              return null;
            }
        }
    }
    return xmlhttp;
}

function populateList() {
    var state =
encodeURIComponent(document.getElementById("stateList").value);
    var url = 'ch15-04.php?state=' + state;

    // 如果 xmlHttpObj 的值为空
    if (!xmlHttpObj)
       xmlHttpObj = getXmlHttp();
    if (!xmlHttpObj) return;

    xmlHttpObj.open('GET', url, true);
    xmlHttpObj.onreadystatechange = getCities;
    xmlHttpObj.send(null);

}

// 处理 JSON 格式的城市列表
function getCities() {

    if(xmlHttpObj.readyState == 4 && xmlHttpObj.status == 200) {
      try {

          // 对 JSON 格式数据进行 "计算"
          eval("var response = ("+ xmlHttpObj.responseText+")");
          var citySelection = document.getElementById("citySelection");
          var name = value = null;

          // 处理由 JSON 对象返回的数据
          for (var i = 0; i < response.length; i++) {
             name = response[i].title;
             value = response[i].value;
             citySelection.options[i] = new Option(name,value);
          }
       } catch (e) {
```

```
            alert(e.message);
        }
    } else if (xmlHttpObj.readyState == 4 && xmlHttpObj.status != 200) {
        document.getElementById('cities').innerHTML = 'Error: No Cities!';
    }
}

//]]>
</script>
</head>
<body>
<h3>Select State:</h3>
<form action="ch14-02.php" method="get">
<div class="elem">
<select id="stateList" name="state">
<option value="CA">California</option>
<option value="MO">Missouri</option>
<option value="WA">Washington</option>
<option value="ID">Idaho</option>
</select>
</div>
<h3>Cities:</h3>
<div class="elem" id="cities">
<select id="citySelection">
<option> </option>
</select>
</div>
<p>
<input type="submit" value="Lookup" id="submitButton" />
</p>
</form>
</body>
</html>
```

使用 JSON 还需要注意一个问题，eval 函数会引发一些安全隐患。由于返回的可能是任何字符串，这就对 Web 页面的安全带来了一些潜在的问题，如果对 Web 服务不够信任，最好别通过 eval 函数对返回的值进行处理。如果哪一天不需要依赖于 eval 函数，我想一定是个好消息。

15.2.2 JSON 对象

ECMAScript 工作组宣布将在 2009 年发布新版本的 ECMAScript，其中就提供了一个用于支持 JSON 的新对象，专门用于创建、解析 JSON 格式的数据。

微软公司在 IE8 beta 2 版本中第一个实现了 JSON 对象。所幸的是，在 http://www.json.org/js.html 上可以找到一个 JSON JavaScript 程序库，它提供了一些基于该程序库的 JSON 对象。

提示

在 http://msdn.microsoft.com/en-us/library/cc836458(VS.85).aspx 上可以找到与微软公司实现的 JSON 对象相关的文档。

JSON 最终会朝着什么方向发展现在仍然无法确定，但有理由确信至少有两个方法是能够让你感兴趣的，它们都是用来直接访问 JSON 对象的静态方法：

- JSON.parse　基于指定的 JSON 格式字符串创建一个 JavaScript 对象；
- JSON.stringify　将一个 JavaScript 对象序列化成一个 JSON 格式的字符串。

在 Ajax 应用程序中使用 JSON.parse 方法是很简单的。你只需要调用该方法，并将应答的文本信息传入即可：

```
var response = JSON.parse(xmlHttpObj.responseText);
```

JSON.stringify 方法就没这么简单了。对于 JSON2 程序库而言，它有两个参数，而对于微软的 JSON 对象而言则需要 3 个参数。需要关注的只有前两个参数，第一个参数是要序列化的对象，第二个参数是可选的，用来指定替代函数或数组，用来过滤和转换其结果。在下面的示例中 stringify 方法的应用，就是 JSON 支持网站中提供的示例的一个变体：

```
var new objFormattedAsJson = JSON.stringify(myObject, function (key, value) {
    if (typeof value === 'number' && !isFinite(value)) {
        return String(value);
    }
    return value;
});
```

示例 15.6 是对示例 15.5 的修改，它添加了对 JSON2 程序库的引用，同时使用了 JSON 对象。通过整合这个程序库，就可以在任何浏览器上正确运行。

示例 15.6　JSON 版本的城市信息查询应用程序，已修改成基于 JSON 对象实现

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
    "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>City lookup</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<style type="text/css">
div.elem
{
   margin: 20px;
}
div#cities
{
  display: none;
}
```

```
</style>

<script type="text/javascript" src="json2.js">
</script>
<script type="text/javascript">
//<![CDATA[

// 全局变量
var xmlHttpObj;

function catchEvent(eventObj, event, eventHandler) {
   if (eventObj.addEventListener) {
      eventObj.addEventListener(event, eventHandler,false);
    } else if (eventObj.attachEvent) {
      event = "on" + event;
      eventObj.attachEvent(event, eventHandler);
    }
}

catchEvent(window,"load", function() {
   document.getElementById("cities").style.display="block";
   document.getElementById("submitButton").style.display="none";
   document.getElementById("stateList").onchange=populateList;
});
function getXmlHttp() {
   var xmlhttp = null;
   if (window.XMLHttpRequest) {
      xmlhttp = new XMLHttpRequest();
   } else {
      try
        {
           xmlhttp = new ActiveXObject("Msxml2.XMLHTTP");
        }
        catch (e) {
           try
           {
              xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");
           }
           catch (e) {
              return null;
           }
        }
   }
   return xmlhttp;
}

function populateList() {
   var state =
encodeURIComponent(document.getElementById("stateList").value);
   var url = 'ch15-04.php?state=' + state;
```

```
    // 如果 xmlHttpObj 值为空
    if (!xmlHttpObj)
       xmlHttpObj = getXmlHttp();
    if (!xmlHttpObj) return;

    xmlHttpObj.open('GET', url, true);
    xmlHttpObj.onreadystatechange = getCities;
    xmlHttpObj.send(null);
}
// 处理 JSON 格式的城市列表
function getCities() {

    if(xmlHttpObj.readyState == 4 && xmlHttpObj.status == 200) {
      try {

          // 对 JSON 字符串进行 "计算"
          var response = JSON.parse(xmlHttpObj.responseText);

          var citySelection = document.getElementById("citySelection");
          var name = value = null;

          // 处理从 JSON 对象返回的数据
          for (var i = 0; i < response.length; i++) {
             name = response[i].title;
             value = response[i].value;
             citySelection.options[i] = new Option(name,value);
          }
      } catch (e) {
            alert(e.message);
      }
    } else if (xmlHttpObj.readyState == 4 && xmlHttpObj.status != 200) {
      document.getElementById('cities').innerHTML = 'Error: No Cities!';
    }
}

//]]>
</script>
</head>
<body>
<h3>Select State:</h3>
<form action="ch14-02.php" method="get">
<div class="elem">
<select id="stateList" name="state">
<option value="CA">California</option>
<option value="MO">Missouri</option>
<option value="WA">Washington</option>
<option value="ID">Idaho</option>
</select>
```

```
</div>
<h3>Cities:</h3>
<div class="elem" id="cities">
<select id="citySelection">
<option> </option>
</select>
</div>
<p>
<input type="submit" value="Lookup" id="submitButton" />
</p>
</form>
</body>
</html>
```

现在你已经可以访问 JSON 字符串了，而且结果也更安全。

如你所示，JSON 方法要比 XML 方法“更简单”，虽然比 HTML 片段复杂一些。但是，不要将是否简单作为决定采用哪种格式的主要因素。你可能无法决定数据是以什么格式发送的，或许也不得不处理不同格式的结果数据。另外，如果要处理越来越复杂的对象，使用 XSLT 将 XML 转成可读的格式，或许是最终降低工作量的好方法。

如果你需要直接面对如关系型数据库、资源描述框架（Resource Description Framework，RDF）或可伸缩的向量图形（Scalable Vector Graphics，SVG），那么始终使用 XML 格式是一种好方法，这样就能够使要处理的数据保持一致的格式。

提示

使用 XML 时还要考虑的问题是命名空间。命名空间可以为元素名添加一些修饰（例如 content:name），避免和其他元素出现冲突。在名为 getElementsByTagNameNS 的 DOM 方法中，就有一个用来指定命名空间的参数，但并不是所有浏览器都能支持它，其中就包括 IE。

到现在为止，Ajax 都不是个很复杂的过程，使用也相当简单，这也是我在这些示例中想传达的意思。更重要的是，你可以完全不考虑服务器端应用程序和客户端页面之间的数据是如何传输的。

15.3 知识测验

1. 在第 14 章中，我介绍了一个放在本书源代码包 drink 子目录下的 Ajax 示例。该应用程序将从下拉列表中获取饮料名称，然后通过 Ajax 获取其配方信息并通过 JavaScript 显示出来，如图 15.1 所示。第一个示例是以 HTML 片段的形式组织饮料配方信息的。而第二个示例（drink2.html 以及其相关的 JavaScript 文件 getdrink3.js），则是以 XHTML 格式返回数据的。

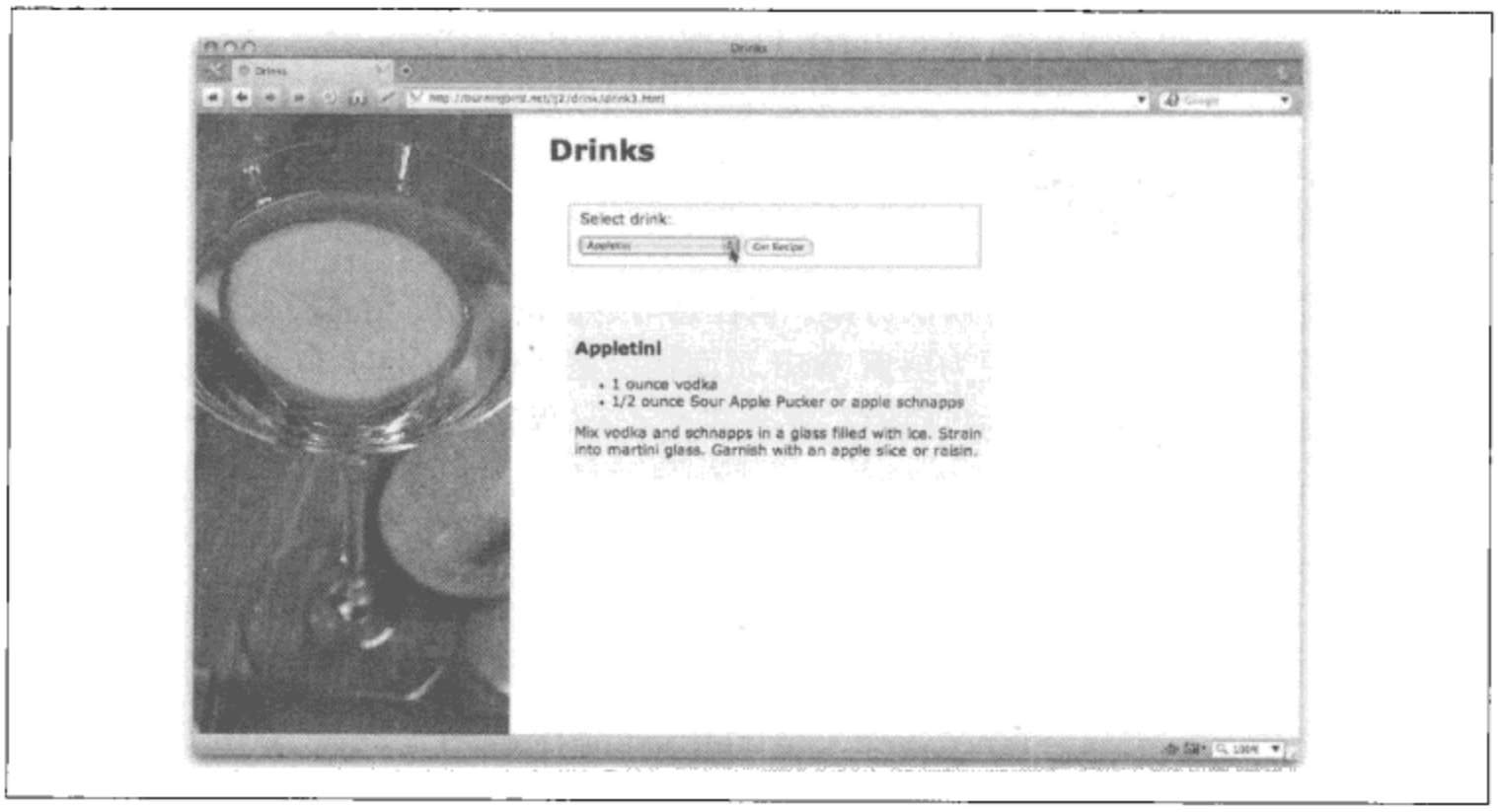


图 15.1　Drink 应用程序

如果要通过 JavaScript 创建一个名为 recipe 的 div 元素可以采用什么方法？接下来可以使用一组 DOM 函数对以 XML 格式返回的配方信息进行访问，并将其转成 HTML 格式，那么应该使用什么函数才能将其添加到新的 recipe 元素上？配方中的各种成分将以列表项的形式列出，配方说明将以段落形式显示，而配方名称将替换为 h3 元素，并将其放在配方成分的前面。

应答中的 XML 数据格式如下所示：

```
<recipe>
<title>Appletini</title>
<ingredient>1 ounce vodka</ingredient>
<ingredient>1/2 ounce Sour Apple Pucker or apple schnapps</ingredient>
-
<instruction>
Mix vodka and schnapps in a glass filled with ice. Strain into martini glass.
Garnish with an apple slice or raisin.
</instruction>
</recipe>
```

注意，该问题的正确答案并不是唯一的。

2. drink 应用程序的第三处变化是服务器应答将采用 JSON 格式。请记住在问题 1 中列出的 Web 页面参数，当使用了 JSON 对象之后，应该如何处理以 JSON 格式返回的数据？

以下就是可能从服务器获得的 JSON 对象实例：

```
{ "title" : "Appletini", "ingredients" : [ { "ingredient" : "1 ounce vodka" },{
"ingredient" : "1/2 ounce Sour Apple Pucker or apple schnapps"}], "instruction" :
```

```
"Mix vodka and schnapps in a glass filled with ice. Strain into martini glass.
Garnish with an apple slice or raisin."}
```

15.4 测验答案

1. 以下是该问题的其中一种解决方案。请注意对返回的 XML 信息所调用的 DOM 方法，它和创建元素（以便为更新 Web 页面构建格式化的内容）所用的方法是相同的。当你看到如下所示的代码时，就会知道 Ajax 程序库为什么会如此流行：

```
if(xmlhttp.readyState == 4 && xmlhttp.status == 200) {
   var body = document.getElementsByTagName('body');
  // 如果已经存在则删除它
   if (document.getElementById('recipe')) {
      body[0].removeChild(document.getElementById('recipe'));
   }
   var recipe = document.createElement('div');
   recipe.id = 'recipe';
   recipe.className='recipe';
   // 添加标题
   var title =
xmlhttp.responseXML.getElementsByTagName('title')[0].firstChild.
nodeValue;
   var titleNode = document.createElement('h3');
   titleNode.appendChild(document.createTextNode(title));
   recipe.appendChild(titleNode);
   // 添加配方成分
   var ul = document.createElement("ul");
   var ingredients =
xmlhttp.responseXML.getElementsByTagName('ingredient');
   for (var i = 0; i < ingredients.length; i++) {
      var x = document.createElement('li');
x.appendChild(document.createTextNode(ingredients[i].firstChild.
      nodeValue));
ul.appendChild(x);
   }
   recipe.appendChild(ul);
   // 添加使用说明
   var instr =
xmlhttp.responseXML.getElementsByTagName('instruction')[0].firstC
hild.nodeValue;
   var instrNode = document.createElement('p');
  instrNode.appendChild(document.createTextNode(instr));
   recipe.appendChild(instrNode);

   // 添加到 body 元素中
   body[0].appendChild(recipe);
}
```

2. 下面是针对该问题的其中一种解决方案。应用程序首先获取 body 元素的引用，然

后删除现有的 id 为 recipe 的 div 元素，然后在添加新的配方之前“重新设置”画布区域。你自己想的解决方案可能不太一样，以下只是其中一种解决方案：

```
if(xmlhttp.readyState == 4 && xmlhttp.status == 200) {

  var body = document.getElementsByTagName('body');

  // 如果已存在则删除它
  if (document.getElementById('recipe')) {
    body[0].removeChild(document.getElementById('recipe'));
  }
  var recipe = document.createElement('div');
  recipe.id = 'recipe';
  recipe.className='recipe';

  // 创建对象
  var recipeObj = JSON.parse(xmlhttp.responseText);

  // 添加标题
  var title = recipeObj['title'];
  var titleNode = document.createElement('h3');
  titleNode.appendChild(document.createTextNode(title));
  recipe.appendChild(titleNode);

  // 添加配方成分
  var ul = document.createElement('ul');
  var ingredients = recipeObj.ingredients;
  for (var i = 0; i < ingredients.length; i++) {
    var x = document.createElement('li');
    x.appendChild(document.createTextNode
(ingredients[i].ingredient));
    ul.appendChild(x);
  }
  recipe.appendChild(ul);

  // 添加使用说明
  var instr = recipeObj.instruction;
  var instrNode = document.createElement('p');
  instrNode.appendChild(document.createTextNode(instr));
  recipe.appendChild(instrNode);
  body[0].appendChild(recipe);
}
```

计算机精品学习资料大放送

软考官方指定教材及同步辅导书下载 | 软考历年真是解析与答案

软考视频 | 考试机构 | 考试时间安排

Java 一览无余：**Java** 视频教程 | **Java SE** | **Java EE**

.Net 技术精品资料下载汇总：**ASP.NET** 篇

.Net 技术精品资料下载汇总：**C#**语言篇

.Net 技术精品资料下载汇总：**VB.NET** 篇

撼世出击：**C/C++**编程语言学习资料尽收眼底 电子书+视频教程

Visual C++(VC/MFC)学习电子书及开发工具下载

Perl/CGI 脚本语言编程学习资源下载地址大全

Python 语言编程学习资料(电子书+视频教程)下载汇总

最新最全 **Ruby**、**Ruby on Rails** 精品电子书等学习资料下载

数据库精品学习资源汇总：**MySQL** 篇 | **SQL Server** 篇 | **Oracle** 篇

最强 **HTML/xHTML**、**CSS** 精品学习资料下载汇总

最新 **JavaScript**、**Ajax** 典藏级学习资料下载分类汇总

网络最强 **PHP** 开发工具+电子书+视频教程等资料下载汇总

UML 学习电子资下载汇总 软件设计与开发人员必备

经典 **LinuxCBT** 视频教程系列 **Linux** 快速学习视频教程一帖通

天罗地网：精品 **Linux** 学习资料大收集(电子书+视频教程) **Linux** 参考资源大系

Linux 系统管理员必备参考资料下载汇总

Linux shell、内核及系统编程精品资料下载汇总

UNIX 操作系统精品学习资料<电子书+视频>分类总汇

FreeBSD/OpenBSD/NetBSD 精品学习资源索引 含书籍+视频

Solaris/OpenSolaris 电子书、视频等精华资料下载索引

O'REILLY®

JavaScript学习指南（第2版）

通过列举JavaScript应用的最佳实践和示例，本书展示了如何将该语言集成到浏览器环境中，及如何在符合标准的网站中应用这些已通过实践验证的编码技术。

本书内容：

- JavaScript应用程序的结构，包括基本的语句和程序控制结构；
- 标识JavaScript中的不同对象String、Number、Boolean、函数等；
- 使用浏览器调试工具和排错技术；
- 事件处理机制、表单事件以及带表单的JavaScript应用程序；
- 基于浏览器对象模型（BOM）、文档对象模型（DOM）以及所创建的自定义对象完成开发；
- 浏览器端的cookie及更新的客户端存储技术；
- 在Ajax应用程序中使用XML或JSON表示法的细节。

本书遵循已被验证的学习法则，帮助读者逐步理解各种概念，使读者掌握在各种浏览器中创建强大的、快速响应的应用程序的方法。

“无论你是初学者还是有经验的程序员，当你学习一门新语言时，我都会强烈建议你阅读本书。Shelley所采用的直观易懂的教学方法，能够帮助你掌握该语言的基础和细节，以便你能够在自己网站上更好地使用它。”

——Anthony T. Holdener III,
Ajax: The Definitive Guide
一书的作者

Shelley Powers在实践中应用Web技术并发表各种与Web技术有关的文章已经长达13年之久。她最近在O'Reilly出版的书涉及语义Web、Ajax、JavaScript和Web图形等多个领域。她还是狂热的业余摄影师，同时也是Web开发的狂热爱好者。

www.oreilly.com

O'Reilly Media, Inc.授权人民邮电出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)

分类建议：计算机/网络技术/JavaScript

人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-21404-1

9 787115 214041 >

ISBN 978-7-115-21404-1

定价：45.00 元