



图灵程序设计丛书 Web 开发系列

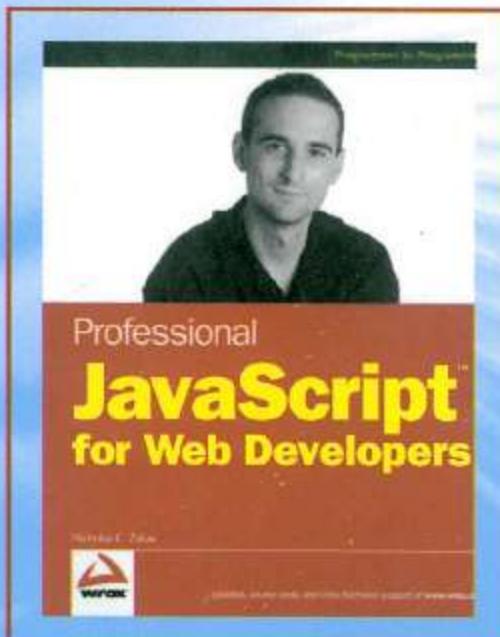


Professional JavaScript for Web Developers

JavaScript 高级程序设计

Nicholas C. Zakas 著
曹力 张欣 等译

- JavaScript 最新经典教程
- Amazon 超级畅销书
- Ajax 程序员必备



人民邮电出版社
POSTS & TELECOM PRESS

图书在版编目 (CIP) 数据

JavaScript 高级程序设计 / 扎卡斯著；曹力等译。—北京：人民邮电出版社，2006.11（2007.7 重印）
(图灵程序设计丛书)

ISBN 978-7-115-15209-1

I. J... II. ①扎... ②曹... III. JAVA 语言—程序设计 IV. TP312

中国版本图书馆 CIP 数据核字 (2006) 第 102589 号

内 容 提 要

JavaScript 是目前 Web 客户端开发的主要编程语言，也是 Ajax 的核心技术之一。本书从最早期 Netscape 浏览器中的 JavaScript 开始讲起，直到当前它对 XML 和 Web 服务的具体支持，内容主要涉及 JavaScript 的语言特点、JavaScript 与浏览器的交互、更高级的 JavaScript 技巧，以及与在 Web 应用程序中部署 JavaScript 解决方案有关的问题，如错误处理、调试、安全性、优化/混淆化、XML 和 Web 服务，最后介绍应用所有这些知识来创建动态用户界面。

本书适合有一定编程经验的开发人员阅读，也可作为高校相关专业课程的教材。

图灵程序设计丛书

JavaScript 高级程序设计

-
- ◆ 著 Nicholas C. Zakas
译 曹 力 张 欣 等
责任编辑 傅志红
- ◆ 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>
北京顺义振华印刷厂印刷
新华书店总店北京发行所经销
- ◆ 开本：800×1000 1/16
印张：34.5
字数：815 千字 2006 年 11 月第 1 版
印数：16 001 – 20 000 册 2007 年 7 月北京第 5 次印刷
- 著作权合同登记号 图字：01-2006-3165 号
ISBN 978-7-115-15209-1/TP
-

定价：59.00 元

读者服务热线：(010) 88593802 印装质量热线：(010) 67129223

版 权 声 明

Original edition, entitled *Professional JavaScript for Web Developers* by Nicholas C.Zakas, by Wiley Publishing, Inc. Copyright © 2005 by Wiley Publishing, Inc.

All rights reserved. This translation published under license.

The Wrox Brand trade dress is a trademark of Wiley Publishing, Inc. in the United States and/or other countries. Used by permission.

Translation edition published by POSTS & TELECOM PRESS Copyright © 2006 .

本书简体中文版由 Wiley Publishing, Inc. 授权人民邮电出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

Wrox 商标是 Wiley 出版公司在美国及其他国家使用的商标，经许可后才能使用。

版权所有，侵权必究。

译 者 序

JavaScript 是赋予网页活力与交互性的主要手段之一，全世界每天都有无数网页在依靠 JavaScript 完成各种关键任务。随着 Web 2.0 和 Ajax 进入主流，JavaScript 已经被推到了舞台中心，使用它来开发更大更复杂的程序势在必行，更多开发人员和 Web 设计师需要熟练掌握 JavaScript。人们发现，由于 JavaScript 同时具有面向对象、过程和函数型语言三类语言的特性，将灵活性与强大功能融于一身，要想真正掌握到能够开发复杂程序的程度，其实并不容易。而 JavaScript 方面完备的开发工具和好书的缺乏，更使这种情况雪上加霜。

让人高兴的是，人民邮电出版社及时地引进了本书，弥补了这一空白。书中世界知名的 JavaScript 专家，用通俗易懂的语言，将 JavaScript 各种相关的技术娓娓道来。在简要讲述了 JavaScript 的语言核心之后，很快转向高级主题，紧贴 Web 开发者目前面对的各种问题。更难得的是，作者还涵盖了当今各个流行浏览器的区别，并帮助读者解决这些问题。

本书主要针对有一定开发经验的读者。刚学 JavaScript 的朋友，可以按部就班成为高手，而已经是高手的朋友，则可以将本书作为参考手册。初学 JavaScript 的朋友，可以选择人民邮电出版社即将出版的最佳入门书：Tom Negrino 的《JavaScript 基础教程》（英文名 *JavaScript for the World Wide Web*）。此外，Jeremy Keith 的《JavaScript DOM 编程艺术》（英文名 *DOM Scripting*，中文版人民邮电出版社即将出版）是目前 Amazon 上排名最高的 JavaScript 图书，书中将如何结合 JavaScript 和 DOM 创造各种绚丽的动态效果发挥得淋漓尽致。需要参考书的朋友，可以考虑 David Flanagan 的《JavaScript 权威指南》，但国内中文版不是最新版本，许多内容已经过时。此外，Danny Goodman 的《JavaScript Bible》是一部类似的大部头著作，但是口碑不如前者，国内尚无译本，需要指出的是，市场上有一本名为《JavaScript 编程宝典》的书，封面装帧完全模仿此书而且价格不菲，其实与 Goodman 完全没有关系，实际上是国人欺世之作，请大家不要上当。

本书第 1~5 章由张欣翻译，第 6~15 章由曹力翻译，第 16~20 章由王霄翻译，全书由张欣统稿、润色及审校。我们深深地感谢我们的家人和朋友。在翻译过程中，他们给予了我们莫大的关心、支持和帮助。限于我们的水平，译文中的疏漏和错误再所难免，请广大读者批评指正。

译 者

2006 年 7 月

前　　言

虽然，服务器端的 Netscape Enterprise Server 和 Active Server Pages (ASP) 都曾经支持 JavaScript，但它主要还是 Web 浏览器使用的客户端脚本语言。目前，JavaScript 主要用于帮助开发者与网页和 Web 浏览器窗口本身进行交互。

不太严格地说，JavaScript 是基于 Java 的¹，而 Java 是一种面向对象程序设计语言，因能够以嵌入式 applet 的形式用于 Web 而流行起来。虽然 JavaScript 的语法和程序设计方法都与 Java 相似，但它并非 Java 的简化版本。相反，JavaScript 是一种独立的语言，它存在于全世界所有的 Web 浏览器中，能够增强用户与 Web 站点和 Web 应用程序之间的交互。

本书从最早期 Netscape 浏览器中的 JavaScript 开始讲起，直到当前它对 XML 和 Web 服务的具体支持。你将学到如何扩展这种语言，以使它适应特殊的需求；学到如何在没有 Java 或隐藏框架这些媒介的情况下，创建无缝的客户机—服务器通信。简而言之，你将学到如何应用 JavaScript 解决 Web 开发者面对的各种问题。

本书内容

本书针对开发人员介绍 JavaScript，包括很多高级的、重要的特性。

本书开始部分探讨了 JavaScript 的起源及其迄今为止的发展。之后详细介绍了构成 JavaScript 实现的各个组件，着重介绍了 ECMAScript 和 DOM（文档对象模型）这样的标准。此外还讨论了在不同 Web 浏览器中使用的 JavaScript 实现的不同。

基于上述讨论，本书开始介绍 JavaScript 的基本概念，包括面向对象程序设计、继承以及它在各种标记语言（如 HTML）中的使用。在探讨了浏览器检测技术，并介绍了在 JavaScript 中如何使用正则表达式后，本书对事件和事件处理进行了深度考察。之后，应用所有这些知识来创建动态的用户界面。

本书最后一部分讨论在 Web 应用程序中部署 JavaScript 解决方案的相关问题，包括错误处理、调试、安全性、优化/模糊化、XML 和 Web 服务。

读者对象

本书针对以下三类读者群：

- 熟悉面向对象程序设计方法的有经验的开发人员。他们由于 JavaScript 与传统的 OO 语言（如 Java 和 C++）相关，所以想学习它。

1. 从语言特性来说，JavaScript 与 Java 并不相同，但 JavaScript 开发过程中确是以 Java 为模仿对象的，请参考 JavaScript 之父 Brendan Eich 的回忆文章。——编者注

□ 尝试提高 Web 站点和 Web 应用程序可用性的 Web 应用程序开发人员。

□ 想更好地理解 JavaScript 语言的初学者。

此外，本书也适用熟悉下列相关技术的读者：

□ XML

□ XSLT

□ Java

□ Web Services

□ HTML

□ CSS

本书不适合没有计算机科学基础背景的初学者，也不适合只想在 Web 站点添加一些简单用户交互特性的读者。这些读者应该阅读人民邮电出版社即将出版的《JavaScript 基础教程》。

环境配置

要运行本书中的示例，需要下列软件：

□ Windows 2000、Windows Server 2003、Windows XP 或 Mac OS X；

□ IE 5.5 或更高版本（Windows）、Mozilla 1.0 或更高版本（所有平台）、Opera 7.5 或更高版本（所有平台）、Safari 1.2 或更高版本（Mac OS X）。

可以从图灵网站 www.turingbook.com 或 <http://www.wrox.com> 的本书配套网页下载书中示例的完整源代码。

本书结构

第 1 章 JavaScript 是什么

这一章解释了 JavaScript 的起源，包括它是怎样产生的，如何发展的以及现状如何。介绍到的概念包括 JavaScript 与 ECMAScript、DOM（文档对象模型）和 BOM（浏览器对象模型）之间的关系。此外还讨论了与 ECMA 和 W3C 的有关标准。

第 2 章 ECMAScript 基础

这一章分析了 JavaScript 所基于的核心技术——ECMAScript。从变量和函数的声明到原始值与引用值的使用和理解，本章描述了编写 JavaScript 代码必需的基础语法和概念。

第 3 章 对象基础

这一章的重点是用 JavaScript 进行面向对象的程序设计（OOP）的基础，涵盖的主题包括用各种方法定义定制的对象、创建对象实例以及了解 JavaScript 和 Java 中 OOP 的异同。

第 4 章 继承

这一章继续探讨 JavaScript 中的 OOP，描述继承机制的工作机理，其中讨论了各种实现继承的方法，并且还比较了它们与 Java 中继承性的异同。

第 5 章 浏览器中的 JavaScript

这一章解释了如何把 JavaScript 嵌入到用各种语言（如 HTML、SVG 和 XUL）编写的网页中。此外还介绍了 BOM（浏览器对象模型）以及它的各种对象和接口。

第 6 章 DOM 基础

这一章介绍了在 JavaScript 中实现的 DOM，包括专门适用于 Web 开发者的 DOM 概念。之后，将这些概念应用在使用 HTML、SVG 和 XUL 编写的示例中。

第 7 章 正则表达式

这一章的重点是 JavaScript 实现的正则表达式，这是进行数据验证和字符串操作的强有力工具。本章探讨了正则表达式的起源、语法以及它在各种程序设计语言中的用法。本章的结尾探讨了正则表达式在 JavaScript 实现中的异同。

第 8 章 检测浏览器和操作系统

这一章解释了编写能在各种 Web 浏览器上运行的 JavaScript 脚本的重要性。它讨论了两种检测浏览器的方法，即对象/特性检测法和 user-agent 字符串检测法，并逐一列出了每种方法的优点和缺点。

第 9 章 事件

这一章讨论了 JavaScript 中最重要的概念之一——事件。事件是把 JavaScript 和任何用标记语言编写的 Web 用户界面连接在一起的主要方法。本章介绍了事件处理的各种方法和事件流的概念（包括冒泡和捕获）。

第 10 章 高级 DOM 技术

这一章介绍了一些较高级的 DOM 特性，包括范围和样式表操作。本章给出了一个例子，用于说明何时以及如何使用这些技术，此外还讨论了如何实现跨浏览器的支持。

第 11 章 表单和数据完整性

这一章讨论了使用表单时数据验证的重要性。在介绍处理验证的方法时，还应用了前面介绍

的概念，如正则表达式、事件和 DOM 操作。

第 12 章 表格排序

这一章应用前面介绍的多种特性来实现客户端的动态表格排序。其中包括用 JavaScript 进行排序的深度讨论，以及如何用事件、DOM 操作和比较运算符开发各种 Web 浏览器都能使用的通用表格排序协议。

第 13 章 拖放

这一章解释了拖放的概念以及它们在 JavaScript 和 Web 浏览器中的应用。其中讨论了系统拖放和模拟拖放的概念，章末创建了一个能跨浏览器使用的标准拖放界面。

第 14 章 错误处理

这一章通过讨论 `try...catch` 语句和 `onerror` 事件处理函数的用法来介绍 JavaScript 中事件处理的概念。另一个主题是用 `throw` 语句创建定制的错误消息以及 JavaScript 调试器的用法。

第 15 章 JavaScript 中的 XML

这一章介绍了 JavaScript 用于读取和操作 XML（可扩展标记语言）数据的特性，解释了各种 Web 浏览器中的支持和对象的不同，还为跨浏览器编码提供了建议。此外，本章还介绍了如何用 XSLT 语言转换客户端的 XML 数据。

第 16 章 客户端与服务器端的通信

这一章探讨了 JavaScript 与服务器通信的各种方法。这些方法包括使用 `cookie` 和基于 JavaScript 的 HTTP 请求。此外，这一章还解释了如何在不使用隐藏框架的情况下实现 GET 和 POST HTTP 请求。

第 17 章 Web 服务

这一章介绍了如何用 JavaScript 提供 Web 服务，其中讨论了在 IE 和 Mozilla 中使用的方法，还为没有内置 Web 服务支持的浏览器提供了一种添加 Web 服务的基本解决方案。

第 18 章 与插件进行交互

这一章解释了 JavaScript 与各种浏览器插件（如 Java applet、SVG 文档和 ActiveX 控件）之间的通信方法。其他主题包括如何编写能与 JavaScript 一起使用的插件。

第 19 章 部署问题

这一章的重点是完成 JavaScript 编码后的操作。本章讨论了在把 JavaScript 解决方案部署到 Web 站点或 Web 应用程序之前要做的一些操作。其中的主题包括安全性、国际化、优化和知识产权保护。

第 20 章 JavaScript 的未来

这一章考查了 JavaScript 的未来，介绍了这种语言的发展方向。其中讨论了 ECMAScript 的 ECMAScript 4 和 XML。

本书约定

为了帮助读者更充分地利用本书，方便阅读，我们在本书中采用如下约定：

在这样的矩形框中给出的内容都是重要的、不应忘记的信息，它与周围的内容直接相关。

对当前讨论的主题可能有一些提示、技巧和旁注，这些都将用楷体显示。

正文中还包括以下样式：

- 在初次介绍重要术语时，使用楷体突出强调；
- 用 Ctrl+A 这样的形式说明键盘按键；
- 正文中的文件名、URL 和代码使用 Courier 字体显示；
- 代码有两种形式：代码示例中，新出现的代码或重要代码用灰色背景突出显示；对当前讨论不太重要的代码或者是前面已经出现过的代码不用灰色背景强调。

In code examples we highlight new and important code with a gray background.

The gray highlighting is not used for code that's less important in the present context or has been shown before.

源代码

在使用本书中的例子时，你可以手工键入所有代码，也可以直接使用本书在网上随附的源代码文件。本书中用到的所有源代码文件都可以从 www.turingbook.com 或 www.wrox.com 下载。访问 Wrox 网站时，只要找到本书的英文版书名 (*Professional JavaScript for Web Developers*) [可以使用搜索 (Search) 框，也可以使用某个书目列表]，并点击该书详细信息网页上的下载代码 (Download Code) 链接，就可以得到本书的所有源代码。

由于会有许多书名字雷同，最佳的方法是利用 ISBN 搜索，本书的英文版 ISBN 是 0-7645-7908-8。

下载代码之后，你只需用最习惯的压缩工具解压就可以了。此外，还可以在 Wrox 的主下载页面 <http://www.wrox.com/dynamic/books/download.aspx> 处找到本书及其他 Wrox 出版的书的配套代码。

勘误表

我们一直努力确保代码或正文中没有错误。不过，是人都会犯错误。如果你发现了我们出版的书中的错误，不论是拼写错误还是代码错误，都请告知我们，我们将非常感谢。这样能节省其他读者的时间，同时还能帮助我们提高内容的准确性。

在 <http://www.wrox.com> 处，用 Search 框或名字列表找到本书的名字，然后在本书的主页上点击 Book Errata 链接，可以找到本书的勘误表。在这个页面上可以找到由 Wrox 的编辑发布的、已经发现的所有错误。在 www.wrox.com/misc-pages/booklist.shtml 处可以找到 Wrox 出版的所有书的列表，其中有每本书的勘误表的链接。

如果在 Book Errata 页面上没有找到你发现的错误，请访问 www.wrox.com/contact/techsupport.shtml 页面，填写其中的表单，把你发现的错误发送给我们。我们将检查你提交的信息，如果正确，就会把它发布在本书的勘误表页面上，并在本书以后的版本中纠正这一错误。

p2p.wrox.com

要与作者或其他人讨论有关问题，请加入 P2P 论坛 (p2p.wrox.com)。这个论坛是一个基于 Web 的系统，你可以在此发表有关 Wrox 图书和相关技术的消息，并与其他读者和技术用户交流。论坛针对你感兴趣的主题提供订购功能，论坛新发布相关消息时，会通过电子邮件通知你。Wrox 作者、编辑、其他行业专家以及其他读者也会造访这些论坛。

在 <http://p2p.wrox.com> 上，你会看到许多论坛，这些论坛不仅可以帮助你阅读本书，还有助于你开发自己的应用程序。要想加入论坛，只需遵循以下几个步骤：

- (1) 访问 p2p.wrox.com，并点击 Register (注册) 链接；
- (2) 阅读使用条文，并点击 Agree (同意)；
- (3) 填写加入论坛的必要信息，如果想提供其他可选信息，也可以相应填写，点击 Submit (提交)；
- (4) 你将收到一个电子邮件，其中说明如何验证你的账户，并完成加入过程。

如果只是阅读论坛中的消息，无需加入 p2p。不过，如果你想发布自己的消息，就必须加入论坛。

一旦加入，就可以发布新的消息了，还可以对其他用户发布的消息做出响应。你在任何时刻都可以在 Web 上阅读消息。如果希望某个论坛能通过电子邮件向你发送新发布的消息，请点击论坛列表中该论坛名旁边的 Subscribe to this Forum (订购此论坛) 图标。

要了解如何使用 Wrox p2p 的更多信息，请阅读 p2p FAQ，在此解释了这个论坛软件如何工

作的相关问题，另外还回答了与 p2p 和 Wrox 图书有关的许多常见问题。要阅读 FAQ，可以点击任何 p2p 页面上的 FAQ 链接。

本书索引可以从图灵网站的本书配套网页下载。索引中的页码为英文原书页码，与书中边栏的页码一致。

致谢

尽管本书的封面上只署了一个名字，但是完成本书的却不只一人。没有众人的帮助，本书是不可能完成的。

首先要感谢的是 Wiley 出版公司的每一个人，尤其是 Jim Minatel 和 Sharon Nash，他们为每位新作者提供了所有必需的指导和支持。

有很多人就一本好的 JavaScript 书应该具有什么内容为我提供了许多意见，包括 Keith Ciociola、Ken Fearnley、John Rajan 和 Douglas Swatski，向他们致谢。

特别感谢审阅本书初稿的每个人，他们是 Erik Arvidsson、Bradley Baumann、Guilherme Blanco、Douglas Crockford、Jean-Luc David、Emil A. Eklund、Brett Fielder、Jeremy McPeak 和 Micha Schopman。他们的每一条意见都使得本书更趋于完美。

感谢 Ed Bernard 和 Frances Bernard 医生，他们在本书编写过程中以及过去几年间，使我保持最好的健康状况。

最后，感谢我的家人——妈妈、爸爸和 Greg 还有我善解人意的女友 Emily。他们的爱和支持帮助我成就本书。

目 录

第 1 章 JavaScript 是什么	1
1.1 历史简述	1
1.2 JavaScript 实现	2
1.2.1 ECMAScript	3
1.2.2 DOM	5
1.2.3 BOM	8
1.3 小结	8
第 2 章 ECMAScript 基础	9
2.1 语法	9
2.2 变量	10
2.3 关键字	12
2.4 保留字	12
2.5 原始值和引用值	13
2.6 原始类型	13
2.6.1 typeof 运算符	14
2.6.2 Undefined 类型	14
2.6.3 Null 类型	15
2.6.4 Boolean 类型	15
2.6.5 Number 类型	15
2.6.6 String 类型	17
2.7 转换	18
2.7.1 转换成字符串	18
2.7.2 转换成数字	19
2.7.3 强制类型转换	20
2.8 引用类型	22
2.8.1 Object 类	22
2.8.2 Boolean 类	23
2.8.3 Number 类	23
2.8.4 String 类	24
2.8.5 instanceof 运算符	28
2.9 运算符	28
2.9.1 一元运算符	28
2.9.2 位运算符	32
2.9.3 Boolean 运算符	37
2.9.4 乘性运算符	40
2.9.5 加性运算符	41
2.9.6 关系运算符	42
2.9.7 等性运算符	43
2.9.8 条件运算符	45
2.9.9 赋值运算符	45
2.9.10 逗号运算符	46
2.10 语句	46
2.10.1 if 语句	46
2.10.2 迭代语句	47
2.10.3 有标签的语句	48
2.10.4 break 语句和 continue 语句	48
2.10.5 with 语句	50
2.10.6 switch 语句	50
2.11 函数	51
2.11.1 无重载	53
2.11.2 arguments 对象	53
2.11.3 Function 类	54
2.11.4 闭包	56
2.12 小结	57
第 3 章 对象基础	58
3.1 面向对象术语	58
3.1.1 面向对象语言的要求	58
3.1.2 对象的构成	59
3.2 对象应用	59
3.2.1 声明和实例化	59
3.2.2 对象引用	59
3.2.3 对象废除	59

3.2.4 早绑定和晚绑定	60	5.1.5 隐藏还是不隐藏	113
3.3 对象的类型	60	5.1.6 <noscript/>标签	113
3.3.1 本地对象	60	5.1.7 XHTML 中的改变	114
3.3.2 内置对象	70	5.2 SVG 中的 JavaScript	116
3.3.3 宿主对象	75	5.2.1 SVG 基础	116
3.4 作用域	75	5.2.2 SVG 中的<script/>标签	117
3.4.1 公用、受保护和私有作用域	75	5.2.3 SVG 中的标签放置	118
3.4.2 静态作用域并非静态的	76	5.3 BOM	119
3.4.3 关键字 this	76	5.3.1 window 对象	119
3.5 定义类或对象	78	5.3.2 document 对象	130
3.5.1 工厂方式	78	5.3.3 location 对象	133
3.5.2 构造函数方式	80	5.3.4 navigator 对象	135
3.5.3 原型方式	80	5.3.5 screen 对象	136
3.5.4 混合的构造函数/原型方式	81	5.4 小结	137
3.5.5 动态原型方法	82	第 6 章 DOM 基础	138
3.5.6 混合工厂方式	83	6.1 什么是 DOM?	138
3.5.7 采用哪种方式	84	6.1.1 XML 简介	138
3.5.8 实例	84	6.1.2 针对 XML 的 API	141
3.6 修改对象	86	6.1.3 节点的层次	141
3.6.1 创建新方法	86	6.1.4 特定语言的 DOM	144
3.6.2 重定义已有方法	87	6.2 对 DOM 的支持	145
3.6.3 极晚绑定	88	6.3 使用 DOM	145
3.7 小结	88	6.3.1 访问相关的节点	145
第 4 章 继承	89	6.3.2 检测节点类型	146
4.1 继承机制实例	89	6.3.3 处理特性	147
4.2 继承机制的实现	90	6.3.4 访问指定节点	148
4.2.1 继承的方式	90	6.3.5 创建和操作节点	150
4.2.2 一个更实际的例子	96	6.4 HTML DOM 特征功能	155
4.3 其他继承方式	100	6.4.1 让特性像属性一样	155
4.3.1 zInherit	100	6.4.2 table 方法	156
4.3.2 xbObjects	104	6.5 遍历 DOM	158
4.4 小结	108	6.5.1 NodeIterator	158
第 5 章 浏览器中的 JavaScript	109	6.5.2 TreeWalker	163
5.1 HTML 中的 JavaScript	109	6.6 测试与 DOM 标准的一致性	165
5.1.1 <script/>标签	109	6.7 DOM Level 3	166
5.1.2 外部文件格式	110	6.8 小结	166
5.1.3 内嵌代码和外部文件	111	第 7 章 正则表达式	167
5.1.4 标签放置	111	7.1 正则表达式支持	167

7.1.1 使用 RegExp 对象	168	8.4.1 方法学	202
7.1.2 扩展的字符串方法	169	8.4.2 第一步	202
7.2 简单模式	170	8.4.3 检测 Opera	204
7.2.1 元字符	170	8.4.4 检测 Konqueror/Safari	206
7.2.2 使用特殊字符	170	8.4.5 检测 IE	208
7.2.3 字符类	172	8.4.6 检测 Mozilla	209
7.2.4 量词	174	8.4.7 检测 Netscape Communicator 4.x	210
7.3 复杂模式	177	8.5 平台/操作系统检测脚本	211
7.3.1 分组	177	8.5.1 方法学	211
7.3.2 反向引用	178	8.5.2 第一步	212
7.3.3 候选	179	8.5.3 检测 Windows 操作系统	212
7.3.4 非捕获性分组	180	8.5.4 检测 Macintosh 操作系统	214
7.3.5 前瞻	181	8.5.5 检测 Unix 操作系统	214
7.3.6 边界	182	8.6 全部脚本	215
7.3.7 多行模式	183	8.7 例子：登录页面	219
7.4 理解 RegExp 对象	184	8.8 小结	224
7.4.1 实例属性	184	第 9 章 事件	225
7.4.2 静态属性	185	9.1 今天的事件	225
7.5 常用模式	186	9.2 事件流	226
7.5.1 验证日期	187	9.2.1 冒泡型事件	226
7.5.2 验证信用卡号	188	9.2.2 捕获型事件	227
7.5.3 验证电子邮件地址	192	9.2.3 DOM 事件流	228
7.6 小结	193	9.3 事件处理函数/监听函数	229
第 8 章 检测浏览器和操作系统	194	9.3.1 IE	230
8.1 navigator 对象	194	9.3.2 DOM	231
8.2 检测浏览器的方式	194	9.4 事件对象	232
8.2.1 对象/特征检测法	194	9.4.1 定位	233
8.2.2 user-agent 字符串检测法	195	9.4.2 属性/方法	233
8.3 user-agent 字符串简史	196	9.4.3 相似性	235
8.3.1 Netscape Navigator 3.0 与 IE3.0	196	9.4.4 区别	238
8.3.2 Netscape Communicator 4.0 与 IE 4.0	197	9.5 事件的类型	240
8.3.3 IE 5.0 及更高版本	198	9.5.1 鼠标事件	240
8.3.4 Mozilla	198	9.5.2 键盘事件	244
8.3.5 Opera	200	9.5.3 HTML 事件	246
8.3.6 Safari	201	9.5.4 变化事件	251
8.3.7 结语	201	9.6 跨平台的事件	252
8.4 浏览器检测脚本	201	9.6.1 EventUtil 对象	252
		9.6.2 添加/删除事件处理函数	252

9.6.3 格式化 event 对象	254	11.4.1 访问选项	309
9.6.4 获取事件对象	258	11.4.2 获取/更改选中项	309
9.6.5 示例	259	11.4.3 添加选项	310
9.7 小结	260	11.4.4 删除选项	311
第 10 章 高级 DOM 技术	261	11.4.5 移动选项	312
10.1 样式编程	261	11.4.6 重新排序选项	313
10.1.1 DOM 样式的方法	263	11.5 创建自动提示的文本框	313
10.1.2 自定义鼠标提示	264	11.5.1 匹配	314
10.1.3 可折叠区域	265	11.5.2 内部机制	314
10.1.4 访问样式表	266	11.6 小结	316
10.1.5 最终样式	270	第 12 章 表格排序	317
10.2 innerText 和 innerHTML	271	12.1 起点——数组	317
10.3 outerText 和 outerHTML	273	12.2 对单列的表格排序	319
10.4 范围	274	12.2.1 比较函数	320
10.4.1 DOM 中的范围	274	12.2.2 sortTable() 函数	320
10.4.2 IE 中的范围	284	12.3 对多列表格进行排序	323
10.4.3 范围在实际中的应用	288	12.3.1 比较函数生成器	323
10.5 小结	288	12.3.2 修改 sortTable() 方法	324
第 11 章 表单和数据完整性	289	12.3.3 逆序排列	325
11.1 表单基础	289	12.3.4 对不同的数据类型进行排序	327
11.2 对<form/>元素进行脚本编写	291	12.3.5 高级排序	330
11.2.1 获取表单的引用	291	12.4 小结	334
11.2.2 访问表单字段	291	第 13 章 拖放	335
11.2.3 表单字段的共性	292	13.1 系统拖放	335
11.2.4 聚焦于第一个字段	292	13.1.1 拖放事件	336
11.2.5 提交表单	293	13.1.2 数据传输对象 dataTransfer	341
11.2.6 仅提交一次	294	13.1.3 dragDrop() 方法	345
11.2.7 重置表单	295	13.1.4 优点及缺点	346
11.3 文本框	295	13.2 模拟拖放	346
11.3.1 获取/更改文本框的值	296	13.2.1 代码	347
11.3.2 选择文本	297	13.2.2 创建放置目标	349
11.3.3 文本框事件	298	13.2.3 优点及缺点	352
11.3.4 自动选择文本	298	13.3 zDragDrop	352
11.3.5 自动切换到下一个	299	13.3.1 创建可拖动元素	352
11.3.6 限制 textarea 的字符数	300	13.3.2 创建放置目标	353
11.3.7 允许/阻止文本框中的字符	301	13.3.3 事件	353
11.3.8 使用上下按键操作数字文本	306	13.3.4 例子	354
11.4 列表框和组合框	308	13.4 小结	355

第 14 章 错误处理	356	16.1.1 cookie 的成分	416
14.1 错误处理的重要性	356	16.1.2 其他安全限制	417
14.2 错误和异常	357	16.1.3 JavaScript 中的 cookie	417
14.3 错误报告	358	16.1.4 服务器端的 cookie	419
14.3.1 IE (Windows)	358	16.1.5 在客户端与服务器端之间 传递 cookie	422
14.3.2 IE (MacOS)	359	16.2 隐藏框架	423
14.3.3 Mozilla (所有平台)	359	16.3 HTTP 请求	426
14.3.4 Safari (MacOS)	360	16.3.1 使用 HTTP 首部	428
14.3.5 Opera 7 (所有平台)	361	16.3.2 实现的复制品	429
14.4 处理错误	362	16.3.3 进行 GET 请求	430
14.4.1 onerror 事件处理函数	362	16.3.4 进行 POST 请求	430
14.4.2 try...catch 语句	365	16.4 LiveConnect 请求	431
14.5 调试技巧	370	16.4.1 进行 GET 请求	431
14.5.1 使用警告框	370	16.4.2 进行 POST 请求	433
14.5.2 使用 Java 控制台	371	16.5 智能 HTTP 请求	435
14.5.3 将消息写入 JavaScript 控制台 (仅限 Opera 7+)	372	16.5.1 get() 方法	435
14.5.4 抛出自定义错误	372	16.5.2 post() 方法	438
14.5.5 JavaScript 校验器	373	16.6 实际使用	439
14.6 调试器	374	16.7 小结	439
14.6.1 Microsoft Script Debugger	374	第 17 章 Web 服务	440
14.6.2 Venkman	376	17.1 Web 服务快速入门	440
14.7 小结	383	17.1.1 Web 服务是什么?	440
第 15 章 JavaScript 中的 XML	384	17.1.2 WSDL	441
15.1 浏览器中的 XML DOM 支持	384	17.2 IE 中的 Web 服务	443
15.1.1 IE 中的 XML DOM 支持	384	17.2.1 使用 WebService 组件	444
15.1.2 Mozilla 中 XML DOM 支持	388	17.2.2 WebService 组件例子	445
15.1.3 通用接口	393	17.3 Mozilla 中的 Web 服务	447
15.2 浏览器中的 XPath 支持	403	17.3.1 加强的特权	447
15.2.1 XPath 简介	403	17.3.2 使用 SOAP 方法	448
15.2.2 IE 中的 XPath 支持	404	17.3.3 使用 WSDL 代理	451
15.2.3 Mozilla 中的 XPath 支持	404	17.4 跨浏览器的方案	454
15.3 浏览器中的 XSLT 支持	408	17.4.1 WebService 对象	454
15.3.1 IE 中的 XSLT 支持	410	17.4.2 Temperature 服务	456
15.3.2 Mozilla 中 XSLT 支持	413	17.4.3 使用 TemperatureService 对象	458
15.4 小结	415	17.5 小结	458
第 16 章 客户端与服务器端的通信	416	第 18 章 与插件进行交互	459
16.1 cookie	416	18.1 为何使用插件	459
		18.2 流行的插件	460

18.3 MIME 类型	460	19.1.3 Mozilla 特有的问题	488
18.4 嵌入插件	461	19.1.4 资源限制	490
18.4.1 加入参数	461	19.2 国际化	491
18.4.2 Netscape 4.x	462	19.2.1 使用 JavaScript 检测语言	491
18.5 检测插件	462	19.2.2 策略	492
18.5.1 检测 Netscape 式插件	463	19.2.3 字符串的思考	492
18.5.2 检测 ActiveX 插件	467	19.3 优化 JavaScript	495
18.5.3 跨浏览器检测	469	19.3.1 下载时间	495
18.6 Java applet	470	19.3.2 执行时间	499
18.6.1 嵌入 applet	470	19.4 知识产权的问题	512
18.6.2 在 JavaScript 中引用 applet	471	19.4.1 混淆	512
18.6.3 创建 applet	471	19.4.2 Microsoft Script Encoder (仅 IE)	513
18.6.4 JavaScript 到 Java 的通信	472	19.5 小结	514
18.6.5 Java 到 JavaScript 的通信	475	第 20 章 JavaScript 的未来	515
18.7 Flash 动画	477	20.1 ECMAScript 4	515
18.7.1 嵌入 Flash 动画	477	20.1.1 Netscape 的提案	515
18.7.2 引用 Flash 动画	478	20.1.2 实现	521
18.7.3 JavaScript 到 Flash 的通信	478	20.2 ECMAScript for XML	522
18.7.4 Flash 到 JavaScript 通信	481	20.2.1 途径	522
18.8 ActiveX 控件	483	20.2.2 for each..in 循环	524
18.9 小结	485	20.2.3 新的类	524
第 19 章 部署问题	486	20.2.4 实现	532
19.1 安全性	486	20.3 小结	532
19.1.1 同源策略	486	索引 (图灵网站下载)	
19.1.2 窗口对象问题	487		

JavaScript 是什么



当 JavaScript 在 1995 年首次出现时，它的主要目的还只是处理一些输入的有效性验证，而在此之前这个工作是留给诸如 Perl 之类的服务器端语言来完成的。之前，要确定一个特定的字段是否空缺或者输入的值是否无效，必须与服务器进行往返的交互。Netscape Navigator 通过引入 JavaScript 来试图改变这种情况。这种直接在客户端处理一些基本的有效性验证的能力，在刚普及使用电话线调制解调器（28.8kbit/s 的速率）的时代，可是一个令人振奋的新特性。但以如此慢的速度与服务器往返交互，对耐性是一种考验。

从那以后，JavaScript 便成长为市面上每一个主要 Web 浏览器都具备的重要特性。同时 JavaScript 不仅仅局限于简单的数据有效性验证，现在几乎可以与浏览器窗口及其内容的每一个方面进行交互。微软公司在早期版本的浏览器中仅支持自己的客户端脚本语言——VBScript，但最后也不得不加入了自己的 JavaScript 实现。

从本章中你可以了解到 JavaScript 是如何以及为何出现的，从它简陋的开始到如今涵盖各种特性的实现。为了能充分发挥 JavaScript 全部的潜力，了解它的本质、历史及局限性是十分重要的。确切地说，本章着重讲解：

- JavaScript 和客户端脚本编程的起源；
- JavaScript 语言的不同部分；
- 与 JavaScript 相关的标准；
- 主流 Web 浏览器中的 JavaScript 支持。

1.1 历史简述

大概在 1992 年，一家称作 Nombas 的公司开始开发一种叫做 C 减减（C-minus-minus，简称 Cmm）的嵌入式脚本语言。Cmm 背后的理念很简单：一个足够强大可以替代宏操作（macro）的脚本语言，同时保持与 C（和 C++）足够的相似性，以便开发人员能很快学会。这个脚本语言捆绑在一个叫做 CEnvi 的共享软件产品中，它首次向开发人员展示了这种语言的威力。Nombas 最终把 Cmm 的名字改成了 ScriptEase，原因是后面的部分（mm）听起来过于“消极”，同时字母 C “令人害怕” (<http://www.nombas.com/us/scripting/history.htm>)。现在 ScriptEase

已经成为了 Nombas 产品背后的主要驱动力。当 Netscape Navigator 崭露头角时，Nombas 开发了一个可以嵌入网页中的 CEnvi 的版本。这些早期的试验称为 Espresso Page（浓咖啡般的页面），它们代表了第一个在万维网上使用的客户端脚本语言。而 Nombas 丝毫没有料到它的理念将会成为因特网的一块重要基石。

当网上冲浪越来越流行时，对于开发客户端脚本的需求也逐渐增大。此时，大部分因特网用户还仅仅通过 28.8kbit/s 的调制解调器来连接到网络，即便这时网页已经不断地变得更大和更复杂。而更加加剧用户痛苦的是，仅仅为了简单的表单有效性验证，就要与服务器端进行多次的往返交互。设想一下，用户填完一个表单，点击提交按钮，等待了 30 秒钟的处理后，看到的却是一条告诉你忘记填写一个必要的字段。那时正处于技术革新最前沿的 Netscape，开始认真考虑一种开发客户端脚本语言来解决简单的处理问题。

当时工作于 Netscape 的 Brendan Eich，开始着手为即将在 1995 年发行的 Netscape Navigator 2.0 开发一个称之为 LiveScript 的脚本语言，当时的目的是同时在浏览器和服务器（本来要叫它 LiveWire 的）端使用它。Netscape 与 Sun 公司联手及时完成 LiveScript 实现。就在 Netscape Navigator 2.0 即将正式发布前，Netscape 将其更名为 JavaScript，目的是为了利用 Java 这个因特网时髦词汇。Netscape 的赌注最终得到回报，JavaScript 从此变成了因特网的必备组件。

因为 JavaScript 1.0 如此成功，Netscape 在 Netscape Navigator 3.0 中发布了 1.1 版。恰巧那个时候，微软决定进军浏览器，发布了 IE 3.0 并搭载了一个 JavaScript 的克隆版，叫做 JScript（这样命名是为了避免与 Netscape 潜在的许可纠纷）。微软步入 Web 浏览器领域的这重要一步虽然令其声名狼藉，但也成为 JavaScript 语言发展过程中的重要一步。

在微软进入后，有 3 种不同的 JavaScript 版本同时存在：Netscape Navigator 3.0 中的 JavaScript、IE 中的 JScript 以及 CEnvi 中的 ScriptEase。与 C 和其他编程语言不同的是，JavaScript 并没有一个标准来统一其语法或特性，而这 3 种不同的版本恰恰突出了这个问题。随着业界担心的增加，这个语言标准化显然已经势在必行。

1997 年，JavaScript 1.1 作为一个草案提交给欧洲计算机制造商协会（ECMA）。第 39 技术委员会（TC39）被委派来“标准化一个通用、跨平台、中立于厂商的脚本语言的语法和语义”(<http://www.ecma-international.org/memento/TC39.htm>)。由来自 Netscape、Sun、微软、Borland 和其他一些对脚本编程感兴趣的公司的程序员组成的 TC39 锤炼出了 ECMA-262，该标准定义了叫做 ECMAScript 的全新脚本语言。

在接下来的几年里，国际标准化组织及国际电工委员会（ISO/IEC）也采纳 ECMAScript 作为标准（ISO/IEC-16262）。从此，Web 浏览器就开始努力（虽然有着不同程度的成功和失败）将 ECMAScript 作为 JavaScript 实现的基础。

1.2 JavaScript 实现

尽管 ECMAScript 是一个重要的标准，但它并不是 JavaScript 唯一的部分，当然，也不是唯一被标准化的部分。实际上，一个完整的 JavaScript 实现是由以下 3 个不同部分组成的（见图 1-1）：

- 核心 (ECMAScript);
- 文档对象模型 (DOM);
- 浏览器对象模型 (BOM)。

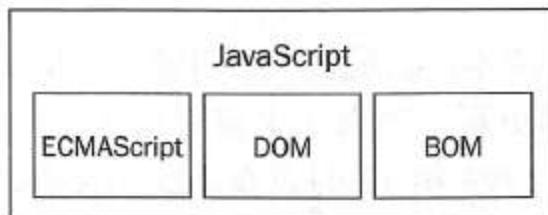


图 1-1

1.2.1 ECMAScript

ECMAScript 并不与任何具体浏览器相绑定，实际上，它也没有提到用于任何用户输入输出的方法（这点与 C 这类语言不同，它需要依赖外部的库来完成这类任务）。那么什么才是 ECMAScript 呢？ECMA-262 标准（第 2 段）的描述如下：

“ECMAScript 可以为不同种类的宿主环境提供核心的脚本编程能力，因此核心的脚本语言是与任何特定的宿主环境分开进行规定的……”

Web 浏览器对于 ECMAScript 来说是一个宿主环境，但它并不是唯一的宿主环境。事实上，还有不计其数的其他各种环境（例如 Nombas 的 ScriptEase 和 Macromedia 同时用在 Flash 与 Director MX 中的 ActionScript）可以容纳 ECMAScript 实现。那么 ECMAScript 在浏览器之外规定了些什么呢？简单地说，ECMAScript 描述了以下内容：

- 语法；
- 类型；
- 语句；
- 关键字；
- 保留字；
- 运算符；
- 对象。

ECMAScript 仅仅是一个描述，定义了脚本语言的所有属性、方法和对象。其他的语言可以实现 ECMAScript 来作为功能的基准，JavaScript 就是这样（见图 1-2）。

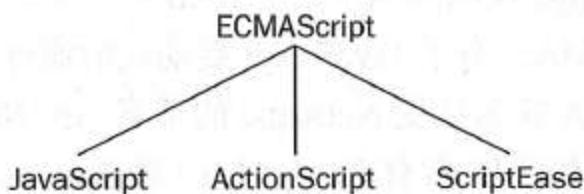


图 1-2

每个浏览器都有它自己的 ECMAScript 接口的实现，然后这个实现又被扩展，包含了 DOM

和 BOM(在以下几节中再讨论)。当然还有其他实现并扩展了 ECMAScript 的语言,例如 Windows 脚本宿主(Windows Scripting Host, WSH)、Macromedia 的 Flash 与 Director MX 中的 ActionScript, 以及 Nombas ScriptEase。

1. ECMAScript 的版本

ECMAScript 分成几个不同的版本, 它是在一个叫做 ECMA-262 的标准中定义的。和其他标准一样, ECMA-262 会被编辑和更新。当有了主要更新时, 就会发布一个标准的新版。最新 ECMA-262 的版本是第三版, 于 1999 年 12 月发布。ECMA-262 的第一版在本质上和 Netscape 的 JavaScript 1.1 是一样的, 只是把所有与浏览器相关的代码删除了, 此外还有一些小的调整。首先, ECMA-262 要求对 Unicode 标准的支持(以便支持多语言)。第二, 它要求对象是平台无关的(Netscape 的 JavaScript 1.1 事实上有不同的对象实现, 例如 Date 对象, 是依赖于平台的)。这也是 JavaScript 1.1 和 1.2 为什么不符合 ECMA-262 规范第一版的主要原因。

ECMA-262 的第二版大部分更新本质上是编辑性的。这次标准的更新是为了与 ISO/IEC-16262 的严格一致, 也并没有特别添加、更改和删除内容。ECMAScript 实现一般不会遵守第二版。

EMCA-262 第三版是该标准第一次真正的更新。它提供了对字符串处理、错误定义和数值输出的更新。同时, 它还增加了正则表达式、新的控制语句、try...catch 异常处理的支持, 以及一些为使标准国际化而做的小改动。一般来说, 它标志着 ECMAScript 成为一种真正的编程语言。

2. 何谓 ECMAScript 符合性

在 ECMA-262 中, ECMAScript 符合性 (conformance) 有明确的定义。一个脚本语言必须满足以下四项基本原则:

- 符合的实现必须按照 ECMA-262 中所描述的支持所有的“类型、值、对象、属性、函数和程序语法及语义”(ECMA-262, 第 1 页);
- 符合的实现必须支持 Unicode 字符标准 (UCS);
- 符合的实现可以增加没有在 ECMA-262 中指定的“额外的类型、值、对象、属性和函数”。ECMA-262 将这些增加描述为规范中未给定的新对象或对象的新属性;
- 符合的实现可以支持没有在 ECMA-262 中定义的“程序和正则表达式语法”(意思是可以替换或者扩展内建的正则表达式支持)。

所有的 ECMAScript 实现必须符合以上标准。

3. Web 浏览器中的 ECMAScript 支持

含有 JavaScript 1.1 的 Netscape Navigator 3.0 在 1996 年发布。然后, JavaScript 1.1 规范被作为一个新标准的草案提交给 ECMA。有了 JavaScript 轰动性的流行, Netscape 十分高兴地开始开发 1.2 版。但有一个问题: ECMA 并未接受 Netscape 的草案。在 Netscape Navigator 3.0 发布后不久, 微软就发布了 IE 3.0。该版本的 IE 含有 JScript 1.0(微软自己的 JavaScript 实现的名称), 原本计划可以与 JavaScript 1.1 相提并论。然而, 由于文档不全以及一些不当的重复特性, JScript 1.0 远远没有达到 JavaScript 1.1 的水平。

在 ECMA-262 第一版定稿之前, 发布含有 JavaScript 1.2 的 Netscape Navigator 4.0 是在 1997

年，在那年晚些时候，ECMA-262 标准被接受并标准化。因此，JavaScript 1.2 并不和 ECMAScript 的第一版兼容，虽然 ECMAScript 应该基于 JavaScript 1.1。

JScript 的下一步升级是 IE 4.0 中加入的 JScript 3.0（2.0 版是随微软的 IIS 3.0 一起发布的，但并未包含在浏览器中）。微软大力宣传 JScript 3.0 是世界上第一个真正符合 ECMA 标准的脚本语言。而那时，ECMA-262 还并没有最终定稿，所以 JScript 3.0 也遭受了和 JavaScript 1.2 同样的命运——它还是没能符合最终的 ECMAScript 标准。

Netscape 选择在 Netscape Navigator 4.06 中升级它的 JavaScript 实现。JavaScript 1.3 使 Netscape 终于完全符合了 ECMAScript 第一版。Netscape 加入了对 Unicode 标准的支持，并让所有的对象保留了在 JavaScript 1.2 中引入的新特性的同时实现了平台独立。

当 Netscape 将它的源代码作为 Mozilla 项目公布于众时，本来计划 JavaScript 1.4 将会嵌入到 Netscape Navigator 5.0 中。然而，一个冒进的决定——要完全从头重新设计 Netscape 的代码，破坏了这个工作。JavaScript 1.4 仅仅作为一个 Netscape Enterprise Server 的服务器端脚本语言发布，以后也没有被放入浏览器中。

如今，所有的主流 Web 浏览器都遵守 ECMA-262 第三版。下面的表格列出了大部分流行 Web 浏览器中的 ECMAScript 支持：

浏览器	ECMAScript 符合性
Netscape Navigator 2.0	-
Netscape Navigator 3.0	-
Netscape Navigator 4.0~4.05	-
Netscape Navigator 4.06~4.79	Edition 1
Netscape 6.0+ (Mozilla 0.6.0+)	Edition 3
Internet Explorer 3.0	-
Internet Explorer 4.0	-
Internet Explorer 5.0	Edition 1
Internet Explorer 5.5+	Edition 3
Opera 6.0~7.1	Edition 2
Opera 7.2+	Edition 3
Safari 1.0+/Konqueror ~2.0+	Edition 3

1.2.2 DOM

DOM（文档对象模型）是 HTML 和 XML 的应用程序接口（API）。DOM 将把整个页面规划成由节点层级构成的文档。HTML 或 XML 页面的每个部分都是一个节点的衍生物。请考虑下面的 HTML 页面：

```
<html>
  <head>
    <title>Sample Page</title>
  </head>
```

```
<body>
  <p>Hello World!</p>
</body>
</html>
```

这段代码可以用 DOM 绘制一个节点层次图（如图 1-3 所示）。

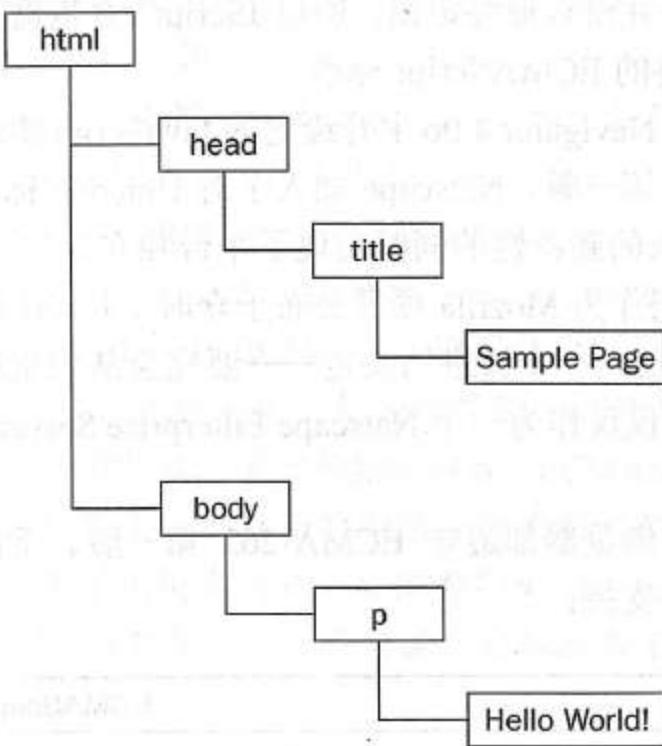


图 1-3

6

DOM 通过创建树来表示文档，从而使开发者对文档的内容和结构具有空前的控制力。用 DOM API 可以轻松地删除、添加和替换节点。

1. 为什么DOM必不可少

自从 IE 4.0 和 Netscape Navigator 4.0 开始支持不同形式的动态 HTML (DHTML)，开发者首次能够在不重载网页的情况下修改它的外观和内容。这是 Web 技术的一大飞跃，不过也带来了巨大的问题。Netscape 和微软各自开发自己的 DHTML，从而结束了 Web 开发者只编写一个 HTML 页面就可以在所有浏览器中访问的时期。

业界决定必须要做点什么以保持 Web 的跨平台特性，他们担心如果放任 Netscape 和微软公司这样做，Web 必将分化为两个独立的部分，每一部分只适用于特定的浏览器。因此，负责制定 Web 通信标准的团体 W3C (World Wide Web Consortium) 就开始制定 DOM。

2. DOM的各个Level

DOM Level 1 是 W3C 于 1998 年 10 月提出的。它由两个模块构成，即 DOM Core 和 DOM HTML。前者提供了基于 XML 的文档的结构图，以方便访问和操作文档的任意部分；后者添加了一些 HTML 专用的对象和方法，从而扩展了 DOM Core。

注意，DOM 不是 JavaScript 专有的，事实上许多其他语言都实现了它。不过，Web 浏览器中的 DOM 已经用 ECMAScript 实现了，现在是 JavaScript 语言的一个很大组成部分。

DOM Level 1 只有一个目标，即规划文档的结构，DOM Level 2 的目标就广泛多了。对原始 DOM 的扩展添加了对鼠标和用户界面事件(DHTML 对此有丰富的支持)、范围、遍历(重复执行 DOM 文档的方法)的支持，并通过对象接口添加了对 CSS(层叠样式表)的支持。由 Level 1 引入的原始 DOM Core 也加入了对 XML 命名空间的支持。

DOM Level 2 引入几种 DOM 新模块，用于处理新的接口类型：

- DOM 视图——描述跟踪文档的各种视图(即 CSS 样式化之前和 CSS 样式化之后的文档)的接口；
- DOM 事件——描述事件的接口；
- DOM 样式——描述处理基于 CSS 样式的接口；
- DOM 遍历和范围——描述遍历和操作文档树的接口。

DOM Level 3 引入了以统一的方式载入和保存文档的方法(包含在新模块 DOM Load and Save 中)以及验证文档(DOM Validation)的方法，从而进一步扩展了 DOM。在 Level 3 中，DOM Core 被扩展为支持所有的 XML 1.0 特性，包括 XML Infoset、XPath 和 XML Base。

在学习 DOM 时，可能会遇到有人引用 DOM Level 0。注意，根本没有 DOM Level 0 这个标准，它只是 DOM 的一个历史参考点(DOM Level 0 指的是 IE 4.0 和 Netscape Navigator 4.0 中支持的原始 DHTML)。

3. 其他DOM

除了 DOM Core 和 DOM HTML 外，还有其他几种语言发布了自己的 DOM 标准。这些语言都是基于 XML 的，每种 DOM 都给对应语言添加了特有的方法和接口：

- 可缩放矢量图形(SVG) 1.0；
- 数学标记语言(MathML) 1.0；
- 同步多媒体集成语言(SMIL)。

此外，其他语言也开发了自己的 DOM 实现，如 Mozilla 的 XML 用户界面语言(XUL)。不过，只有上面列出的几种语言是 W3C 的推荐标准。

4. Web浏览器中的DOM支持

DOM 在被 Web 浏览器开始实现之前就已经是一种标准了。IE 首次尝试支持 DOM 是在 5.0 版本中，不过其实直到 5.5 版本才具有真正的 DOM 支持，IE 5.5 实现了 DOM Level 1。从那时起，IE 就没有再引入新的 DOM 功能。

Netscape 直到 Netscape 6(Mozilla 0.6.0) 才引入 DOM 支持。目前，Mozilla 具有最好的 DOM 支持，实现了完整的 Level 1、几乎所有的 Level 2 以及一部分 Level 3。(Mozilla 开发小组的目标是构造一个与标准 100% 兼容的浏览器，他们的工作得到了回报。)

Opera 直到 7.0 版本才加入 DOM 支持，还有 Safari 也实现了大部分 DOM Level 1。它们几乎都与 IE 5.5 处于同一水平，有些情况下，甚至超过了 IE 5.5。不过，就对 DOM 的支持而论，所有浏览器都远远落后于 Mozilla。下表列出了常用浏览器对 DOM 的支持。

浏 览 器	DOM 兼容性
Netscape Navigator 1.0-4.x	-
Netscape 6.0+ (Mozilla 0.6.0+)	Level 1、Level 2、Level 3 (部分)
IE 2.0-4.x	-
IE 5.0	Level 1 (最小)
IE 5.5+	Level 1 (几乎全部)
Opera 1.0-6.0	-
Opera 7.0+	Level 1 (几乎全部)、Level 2 (部分)
Safari 1.0+/Konqueror ~2.0+	Level 1

8

1.2.3 BOM

IE 3.0 和 Netscape Navigator 3.0 提供了一种特性——BOM (浏览器对象模型)，可以对浏览器窗口进行访问和操作。使用 BOM，开发者可以移动窗口、改变状态栏中的文本以及执行其他与页面内容不直接相关的动作。使 BOM 独树一帜且又常常令人怀疑的地方在于，它只是 JavaScript 实现的一部分，没有任何相关的标准。

BOM 主要处理浏览器窗口和框架，不过通常浏览器特定的 JavaScript 扩展都被看作 BOM 的一部分。这些扩展包括：

- 弹出新的浏览器窗口；
- 移动、关闭浏览器窗口以及调整窗口大小；
- 提供 Web 浏览器详细信息的导航对象；
- 提供装载到浏览器中页面的详细信息的定位对象；
- 提供用户屏幕分辨率详细信息的屏幕对象；
- 对 cookie 的支持；
- IE 扩展了 BOM，加入了 ActiveXObject 类，可以通过 JavaScript 实例化 ActiveX 对象。

由于没有相关的 BOM 标准，每种浏览器都有自己的 BOM 实现。有一些事实上的标准，如具有一个窗口对象和一个导航对象，不过每种浏览器可以为这些对象或其他对象定义自己的属性和方法。本书第 5 章详细介绍了这些实现的不同之处。

1.3 小结

本章介绍了 JavaScript 这种客户端 Web 浏览器脚本语言。你已经了解了构成 JavaScript 完整实现的各个部分：

- JavaScript 的核心 ECMAScript 描述了该语言的语法和基本对象；
- DOM 描述了处理网页内容的方法和接口；
- BOM 描述了与浏览器进行交互的方法和接口。

此外，本章还探讨了 JavaScript 的历史，使你了解到该语言的各个部分是如何发展而来的，以及历史上浏览器是如何处理各种标准的实现的。

9

10



有些简单的 JavaScript 功能在浏览器中很容易实现。因特网上有无数的文章说明如何用 JavaScript 实现“傻瓜式的 Web 小把戏”，它们包括如何弹出用户提示信息、交换图片以及创建简单的游戏等。虽然这些功能给 Web 站点增加了趣味性，不过只是复制粘贴其代码，并不能让你理解它们为什么能起作用以及如何起作用。这一章分析 JavaScript 的核心 ECMAScript，目的在于让你深入了解 JavaScript 是如何运作的。

如前一章所述，ECMAScript 提供了实现通用程序设计任务必需的 JavaScript 的语法、运算符和基本对象。

2.1 语法

熟悉 Java、C 和 Perl 这些语言的开发者会发现 ECMAScript 的语法很容易掌握，因为它借用了这些语言的语法。Java 和 ECMAScript 有一些关键语法特性相同，也有一些完全不同。

ECMAScript 的基础概念如下：

- **区分大小写。**与 Java 一样，变量、函数名、运算符以及其他一切东西都是区分大小写的，也就是说，变量 test 不同于变量 Test。
- **变量是弱类型的。**与 Java 和 C 不同，ECMAScript 中的变量无特定的类型，定义变量时只用 var 运算符，可以将它初始化为任意的值。这样可以随时改变变量所存数据的类型（尽管应该避免这样做）。一些示例如下：

```
var color = "red";
var num = 25;
var visible = true;
```

- **每行结尾的分号可有可无。**Java、C 和 Perl 都要求每行代码以分号（;）结束才符合语法。ECMAScript 则允许开发者自行决定是否以分号结束一行代码。如果没有分号，ECMAScript 就把这行代码的结尾看作该语句的结尾（与 Visual Basic 和 VBScript 相似），前提是这样没有破坏代码的语义。最好的代码编写习惯是总加入分号，因为没有分号，有些浏览器就不能正确运行，不过根据 ECMAScript 标准，下面两行代码的语法都是正确的：

```
var test1 = "red";
var test2 = "blue";
```

□ **注释与 Java、C 和 PHP 语言的注释相同。** ECMAScript 借用了这些语言的注释语法。有两种类型的注释——单行注释和多行注释。单行注释以双斜线（//）开头。多行注释以单斜线和星号（/*）开头，以星号加单斜线结尾（*/）。

```
//this is a single-line comment

/* this is a multi-
line comment */
```

□ **括号表明代码块。** 从 Java 中借鉴的另一个概念是代码块。代码块表示一系列应该按顺序执行的语句，这些语句被封装在左括号（{）和右括号（}）之间。例如：

```
if (test1 == "red") {
    test1 = "blue";
    alert(test1);
}
```

如果你对 ECMAScript 的语法细节感兴趣，可以在 ECMA 的 Web 站点 www.ecma-international.org 下载 *ECMAScript Language Specification* (ECMA-262)。

2.2 变量

如前所述，ECMAScript 中的变量是用 var 运算符（variable 的缩写）加变量名定义的，例如：

```
var test = "hi";
```

在这个例子中，声明了变量 test，并把它的值初始化为 "hi"（字符串）。由于 ECMAScript 是弱类型的，所以解释程序会为 test 自动创建一个字符串值，无需明确的类型声明。还可以用一个 var 语句定义两个或多个变量：

12

```
var test = "hi", test2 = "hola";
```

前面的代码定义了变量 test，初始值为 "hi"，还定义了变量 test2，初始值为 "hola"。不过用同一个 var 语句定义的变量不必具有相同的类型，如下所示：

```
var test = "hi", age = 25;
```

这个例子除了（再次）定义 test 外，还定义了 age，并把它初始化为 25。即使 test 和 age 属于两种不同的数据类型，在 ECMAScript 中这样定义也是完全合法的。

与 Java 不同，ECMAScript 中的变量并不一定要初始化（它们是在幕后初始化的，将在后面讨论这一点）。因此，下面一行代码也是有效的：

```
var test;
```

此外，与 Java 不同的还有变量可以存放不同类型的值。这是弱类型变量的优势。例如，可以把变量初始化为字符串类型的值，之后把它设置为数字值，如下所示：

```
var test = "hi";
alert(test); //outputs "hi"
//do something else here
test = 55;
alert(test); //outputs "55"
```

这段代码将毫无问题地输出字符串值和数字值。但是，如前所述，使用变量时，好的编码习惯是始终存放相同类型的值。

变量名需要遵守两条简单的规则：

- 第一个字符必须是字母、下划线（_）或美元符号（\$）。
- 余下的字符可以是下划线、美元符号或任何字母或数字字符。

下面的变量名都是合法的：

```
var test;
var $test;
var $1;
var _$te$t2;
```

当然，只是因为变量名的语法正确并不意味着就该使用它们。变量还应遵守以下某条著名的命名规则：

- Camel 标记法——首字母是小写的，接下来的单词都以大写字母开头。例如：

```
var myTestValue = 0, mySecondTestValue = "hi";
```

13

- Pascal 标记法——首字母是大写的，接下来的单词都以大写字母开头。例如：

```
var MyTestValue = 0, MySecondTestValue = "hi";
```

- 匈牙利类型标记法——在以 Pascal 标记法命名的变量前附加一个小写字母（或小写字母序列），说明该变量的类型。例如，i 表示整数，s 表示字符串，如下所示：

```
var iMyTestValue = 0, sMySecondTestValue = "hi";
```

下面的表列出了用匈牙利类型标记法定义 ECMAScript 变量使用的前缀。本书都采用这些前缀，以使示例代码更易于阅读：

类 型	前 缀	示 例
数组	a	aValues
布尔型	b	bFound
浮点型（数字）	f	fValue
函数	fn	fnMethod
整型（数字）	i	iValue
对象	o	oType
正则表达式	re	rePattern
字符串	s	sValue
变型（可以是任何类型）	v	vValue

ECMAScript 另一个有趣方面（也是与大多数程序设计语言的主要区别）是在使用变量之前不必声明。例如：

```
var sTest = "hello ";
sTest2 = sTest + "world";
alert(sTest2); //outputs "hello world"
```

在上面的代码中，首先，`sTest` 被声明为字符串类型的值 "hello"。接下来的一行，用变量 `sTest2` 把 `sTest` 与字符串 "world" 连在一起。变量 `sTest2` 并没有用 `var` 运算符定义，这里只是插入了它，就像已经声明过它。

ECMAScript 的解释程序遇到未声明过的标识符时，用该变量名创建一个全局变量，并将其初始化为指定的值。这是该语言的便利之处，不过如果不能紧密跟踪变量，这样做也很危险。最好的习惯是像使用其他程序设计语言一样，总是声明所有变量（要了解为什么总是应该声明变量，请参阅第 19 章）。

2.3 关键字

ECMA-262 定义了 ECMAScript 支持的一套关键字（keyword）。这些关键字标识了 ECMAScript 语句的开头和/或结尾。根据规定，关键字是保留的，不能用作变量名或函数名。下面是 ECMAScript 关键字的完整列表：

<code>break</code>	<code>else</code>	<code>new</code>	<code>var</code>
<code>case</code>	<code>finally</code>	<code>return</code>	<code>void</code>
<code>catch</code>	<code>for</code>	<code>switch</code>	<code>while</code>
<code>continue</code>	<code>function</code>	<code>this</code>	<code>with</code>
<code>default</code>	<code>if</code>	<code>throw</code>	
<code>delete</code>	<code>in</code>	<code>try</code>	
<code>do</code>	<code>instanceof</code>	<code>typeof</code>	

如果把关键字用作变量名或函数名，可能得到诸如“Identifier expected”（应该有标识符）这样的错误消息。

2.4 保留字

ECMAScript 还定义了一套保留字（reserved word）。保留字在某种意义上是为将来的关键字而保留的单词。因此，保留字不能被用作变量名或函数名。ECMA-262 第 3 版中保留字的完整列表如下：

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

如果将保留字用作变量名或函数名，那么除非将来的浏览器实现了该保留字，否则很可能收不到任何错误消息。当浏览器将其实现后，该单词将被看作关键字，如此将出现关键字错误。

2.5 原始值和引用值

在 ECMAScript 中，变量可以存放两种类型的值，即原始值和引用值。

- 原始值 (primitive value) 是存储在栈 (stack) 中的简单数据段，也就是说，它们的值直接存储在变量访问的位置。
- 引用值 (reference value) 是存储在堆 (heap) 中的对象，也就是说，存储在变量处的值是一个指针 (point)，指向存储对象的内存处。

15

为变量赋值时，ECMAScript 的解释程序必须判断该值是原始类型的，还是引用类型的。要实现这一点，解释程序则需尝试判断该值是否为 ECMAScript 的原始类型之一，即 Undefined、Null、Boolean 和 String 型。由于这些原始类型占据的空间是固定的，所以可将它们存储在较小的内存区域——栈中。这样存储便于迅速查寻变量的值。

在许多语言中，字符串都被看作引用类型，而非原始类型，因为字符串的长度是可变的。ECMAScript 打破了这一传统。

如果一个值是引用类型的，那么它的存储空间将从堆中分配。由于引用值的大小会改变，所以不能把它放在栈中，否则会降低变量查寻的速度。相反，放在变量的栈空间中的值是该对象存储在堆中的地址。地址的大小是固定的，所以把它存储在栈中对变量性能无任何负面影响（如图 2-1 所示）。

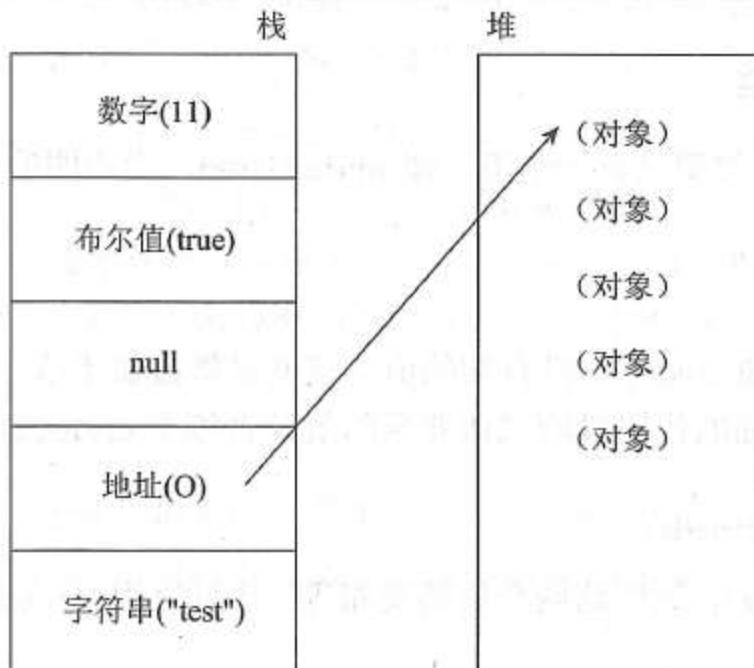


图 2-1

2.6 原始类型

如前所述，ECMAScript 有 5 种原始类型 (primitive type)，即 Undefined、Null、Boolean、

Number 和 String。ECMA-262 把术语类型 (type) 定义为值的一个集合，每种原始类型定义了它包含的值的范围及其字面量表示形式。ECMAScript 提供了 `typeof` 运算符来判断一个值是否在某种类型的范围内。可以用这种运算符判断一个值是否表示一种原始类型；如果它是原始类型，还可以判断它表示哪种原始类型。

2.6.1 `typeof` 运算符

16

`typeof` 运算符有一个参数，即要检查的变量或值。例如：

```
var sTemp = "test string";
alert(typeof sTemp); //outputs "string"
alert(typeof 95); //outputs "number"
```

对变量或值调用 `typeof` 运算符将返回下列值之一：

- "undefined"，如果变量是 `Undefined` 型的。
- "boolean"，如果变量是 `Boolean` 型的。
- "number"，如果变量是 `Number` 型的。
- "string"，如果变量是 `String` 型的。
- "object"，如果变量是一种引用类型或 `Null` 型的。

你也许会问，为什么 `typeof` 运算符对于 `null` 值会返回 "object"。这实际上是 JavaScript 最初实现中的一个错误，然后被 ECMAScript 沿用了。现在，`null` 被认为是对象的占位符，从而解释了这一矛盾，但从技术上来说，它仍然是原始值。

2.6.2 `Undefined` 类型

如前所述，`Undefined` 类型只有一个值，即 `undefined`。当声明的变量未初始化时，该变量的默认值是 `undefined`。

```
var oTemp;
```

前面一行代码声明变量 `oTemp`，没有初始值。该变量将被赋予值 `undefined`，即 `Undefined` 类型的字面量。可以用下面的代码段测试该变量的值是否等于 `undefined`：

```
var oTemp;
alert(oTemp == undefined);
```

这段代码将显示 "true"，说明这两个值确实相等。还可以用 `typeof` 运算符显示该变量的值是 `undefined`。

```
var oTemp;
alert(typeof oTemp); //outputs "undefined"
```

注意，值 `undefined` 并不同于未定义的值。但是，`typeof` 运算符并不真正区分这两种值。考虑下面的代码：

```

var oTemp;

//make sure this variable isn't defined
//var oTemp2;

//try outputting
alert(typeof oTemp);    //outputs "undefined"
alert(typeof oTemp2);   //outputs "undefined"

```

17

前面的代码对两个变量输出的都是“undefined”，即使只有变量 oTemp2 是未被声明过的。如果对 oTemp2 使用除 typeof 之外的其他运算符的话，会引起错误，因为其他运算符只能用于已声明的变量上。例如，下面的代码将引发错误：

```

//make sure this variable isn't defined
//var oTemp2;

//try outputting
alert(oTemp2 == undefined); //causes error

```

当函数无明确返回值时，返回的也是值 undefined，如下所示：

```

function testFunc() {
    //leave the function blank
}
alert(testFunc() == undefined); //outputs "true"

```

2.6.3 Null 类型

另一种只有一个值的类型是 Null，它只有一个专用值 null，即它的字面量。值 undefined 实际上是从值 null 派生来的，因此 ECMAScript 把它们定义为相等的。

```
alert(null == undefined); //outputs "true"
```

尽管这两个值相等，但它们的含义不同。undefined 是声明了变量但未对其初始化时赋予该变量的值，null 则用于表示尚未存在的对象（在讨论 typeof 运算符时，简单地介绍过这一点）。如果函数或方法要返回的是对象，那么找不到该对象时，返回的通常是 null。

2.6.4 Boolean 类型

Boolean 类型是 ECMAScript 中最常用的类型之一。它有两个值 true 和 false（即两个 Boolean 字面量）。即使 false 不等于 0，0 也可以在必要时被转换成 false，这样在 Boolean 语句中使用两者都是安全的。

```

var bFound = true;
var bLost = false;

```

2.6.5 Number 类型

ECMA-262 中定义的最特殊的类型是 Number 型。这种类型既可以表示 32 位的整数，还可

18

以表示 64 位的浮点数。直接输入的（而不是从另一个变量访问的）任何数字都被看作 Number 型的字面量。例如，下面的代码声明了存放整数值的变量，它的值由字面量 55 定义：

```
var iNum = 55;
```

整数也可以被表示为八进制（以 8 为底）或十六进制（以 16 为底）的字面量。八进制字面量的首数字必须是 0，其后的数字可以是任何八进制数字（0 到 7），如下面代码所示：

```
var iNum = 070; //070 is equal to 56 in decimal
```

要创建十六进制的字面量，首位数字必须为 0，其后接字母 x，然后是任意的十六进制数字（0 到 9 和 A 到 F）。这些字母可以是大写的，也可以是小写的。例如：

```
var iNum = 0x1f; //0x1f is equal to 31 in decimal
var iNum2 = 0xAB; //0xAB is equal to 171 in decimal
```

尽管所有整数都可表示为八进制或十六进制的字面量，但所有数学运算返回的都是十进制结果。

要定义浮点值，必须包括小数点和小数点后的一位数字（例如，用 1.0 而不是 1）。这被看作浮点数字面量。例如：

```
var fNum = 5.0;
```

浮点字面量的有趣之处在于，用它进行计算前，真正存储的是字符串。

对于非常大或非常小的数，可以用科学记数法表示浮点值。采用科学记数法，可以把一个数表示为数字（包括十进制数字）加 e（或 E），后面加乘以 10 的倍数。不明白？下面是一个示例：

```
var fNum = 3.125e7;
```

该符号表示的是数 31250000。把科学记数法转化成计算式就可以得到该值： 3.125×10^7 ，即等于 $3.125 \times 10 \times 10 \times 10 \times 10 \times 10 \times 10$ 。

也可用科学记数法表示非常小的数，例如 0.000000000000003 可以表示为 3-e17（这里，10 被升到-17 次幂，意味着需要被 10 除 17 次）。ECMAScript 默认把具有 6 个或 6 个以上前导 0 的浮点数转换成科学记数法。

也可用 64 位 IEEE 754 形式存储浮点值，这意味着十进制值最多可以有 17 个十进制位。17 位之后的值将被截去，从而造成一些小的数学误差。

几个特殊值也被定义为 Number 类型的。前两个是 Number.MAX_VALUE 和 Number.MIN_VALUE，它们定义了 Number 值集合的外边界。所有 ECMAScript 数都必须在这两个值之间。不过计算生成的数值结果可以不落在这两个数之间。

当计算生成的数大于 Number.MAX_VALUE 时，它将被赋予值 Number.POSITIVE_INFINITY，意味着不再有数字值。同样，生成的数值小于 Number.MIN_VALUE 的计算也会被赋予值 Number.NEGATIVE_INFINITY，也意味着不再有数字值。如果计算返回的是无穷大值，那么生成

的结果不能再用于其他计算。

事实上,有专门的值表示无穷大,(如你所猜测的)即 `Infinity`。`Number.POSITIVE_INFINITY` 的值为 `Infinity`, `Number.NEGATIVE_INFINITY` 的值为 `-Infinity`。

由于无穷大数可以是正数也可以是负数,所以可用一个方法判断一个数是否是有穷的(而不是单独测试每个无穷数)。可以对任何数调用 `isFinite()` 方法,以确保该数不是无穷大。例如:

```
var iResult = iNum * some_really_large_number;
if (isFinite(iResult)) {
    alert("Number is finite.");
} else {
    alert("Number is infinite.");
}
```

最后一个特殊值是 `NaN`,表示非数(Not a Number)。`NaN`是个奇怪的特殊值。一般说来,这种情况发生在类型(String、Boolean等)转换失败时。例如,要把单词 `blue`转换成数值就会失败,因为没有与之等价的数值。与无穷大值一样,`NaN`也不能用于算术计算。`NaN`的另一个奇特之处在于,它与自身不相等,这意味着下面的代码将返回 `false`:

```
alert(NaN == NaN); //outputs "false"
```

出于这种原因,不推荐使用 `NaN`值本身。函数 `isNaN()`会做得相当好:

```
alert(isNaN("blue")); //outputs "true"
alert(isNaN("123")); //outputs "false"
```

2.6.6 String 类型

`String`类型的独特之处在于,它是唯一没有固定大小的原始类型。可以用字符串存储0或更多的Unicode字符,由16位整数表示(Unicode是一种国际字符集,本书后面将讨论它)。

字符串中每个字符都有特定的位置,首字符从位置0开始,第二个字符在位置1,依此类推。这意味着字符串中的最后一个字符的位置一定是字符串的长度减1(如图2-2所示)。

这个字符串的长度是6

位置	h	e	l	l	o	!
	0	1	2	3	4	5

图 2-2

字符串字面量是由双引号(“)或单引号(‘)声明的。而Java则是用双引号声明字符串,用单引号声明字符。但是,由于ECMAScript没有字符类型,所以可使用这两种表示法中的任何一种。例如,下面的两行代码都有效:

```
var sColor1 = "blue";
var sColor2 = 'blue';
```

String 类型还包括几种字符字面量, Java、C 和 Perl 的开发者应该对此非常熟悉。下表列出了 ECMAScript 的字符字面量:

字面量	含义
\n	换行
\t	制表符
\b	空格
\r	回车
\f	换页符
\\\	反斜杠
'	单引号
"	双引号
\0nnn	八进制代码 nnn (n 是 0 到 7 中的一个八进制数字) 表示的字符
\xnn	十六进制代码 nn (n 是 0 到 F 中的一个十六进制数字) 表示的字符
\unnnnn	十六进制代码 nnnn (n 是 0 到 F 中的一个十六进制数字) 表示的 Unicode 字符

2.7 转换

所有程序设计语言最重要的特征之一是具有进行类型转换的能力, ECMAScript 给开发者提供了大量简单的转换方法。大多数类型具有进行简单转换的方法, 还有几个全局方法可以用于更复杂的转换。无论哪种情况, 在 ECMAScript 中, 类型转换都是简短的一步操作。

2.7.1 转换成字符串

ECMAScript 的 Boolean 值、数字和字符串的原始值的有趣之处在于它们是伪对象, 这意味着它们实际上具有属性和方法。例如, 要获得字符串的长度, 可以采用下面的代码:

```
var sColor = "blue";
alert(sColor.length); //outputs "4"
```

尽管 "blue" 是原始类型的字符串, 它仍然具有属性 length, 用于存放该字符串的大小。总而言之, 3 种主要的原始值 Boolean 值、数字和字符串都有 `toString()` 方法, 可以把它们的值转换成字符串。

也许你会问, “字符串还有 `toString()` 方法, 这不是多余的吗?” 是的, 的确如此, 不过 ECMAScript 定义所有对象都有 `toString()` 方法, 无论它是伪对象, 还是真的对象。因为 String 类型属于伪对象, 所以它一定有 `toString()` 方法。

Boolean 型的 `toString()` 方法只是输出 "true" 或 "false", 结果由变量的值决定:

```
var bFound = false;
alert(bFound.toString()); //outputs "false"
```

Number 类型的 `toString()` 方法比较特殊, 它有两种模式, 即默认模式和基模式。采用默

认模式, `toString()`方法只是用相应的字符串输出数字值(无论是整数、浮点数还是科学记数法), 如下所示:

```
var iNum1 = 10;
var fNum2 = 10.0;
alert(iNum1.toString()); //outputs "10"
alert(fNum2.toString()); //outputs "10"
```

在默认模式中,无论最初采用什么表示法声明数字,Number类型的`toString()`方法返回的都是数字的十进制表示。因此,以八进制或十六进制字面量形式声明的数字输出时都是十进制形式的。

采用Number类型的`toString()`方法的基本模式,可以用不同的基输出数字,例如二进制的基是2,八进制的基是8,十六进制的基是16。基只是要转换成的基数的另一种叫法而已,它是`toString()`方法的参数:

```
var iNum = 10;
alert(iNum1.toString(2)); //outputs "1010"
alert(iNum1.toString(8)); //outputs "12"
alert(iNum1.toString(16)); //outputs "A"
```

在前面的示例中,以3种不同的形式输出了数字10,即二进制形式、八进制形式和十六进制形式。HTML采用十六进制数表示每种颜色,在HTML中处理数字时这种功能非常有用。

22

对数字调用`toString(10)`与调用`toString()`相同,它们返回的都是该数字的十进制形式。

2.7.2 转换成数字

ECMAScript提供了两种把非数字的原始值转换成数字的方法,即`parseInt()`和`parseFloat()`。正如你可能想到的,前者把值转换成整数,后者把值转换成浮点数。只有对String类型调用这些方法,它们才能正确运行;对其他类型返回的都是`Nan`。

在判断字符串是否是数字值前,`parseInt()`和`parseFloat()`都会仔细分析该字符串。`parseInt()`方法首先查看位置0处的字符,判断它是否是个有效数字;如果不是,该方法将返回`Nan`,不再继续执行其他操作。但如果该字符是有效数字,该方法将查看位置1处的字符,进行同样的测试。这一过程将持续到发现非有效数字的字符为止,此时`parseInt()`将把该字符之前的字符串转换成数字。例如,如果要把字符串"1234blue"转换成整数,那么`parseInt()`将返回1234,因为当它检测到字符b时,就会停止检测过程。字符串中包含的数字字面量会被正确转换为数字,因此字符串"0xA"会被正确转换为数字10。不过,字符串"22.5"将被转换成22,因为对于整数来说,小数点是无效字符。一些示例如下:

```
var iNum1 = parseInt("1234blue"); //returns 1234
var iNum2 = parseInt("0xA"); //returns 10
var iNum3 = parseInt("22.5"); //returns 22
var iNum4 = parseInt("blue"); //returns Nan
```

`parseInt()`方法还有基模式,可以把二进制、八进制、十六进制或其他任何进制的字符串

转换成整数。基是由 `parseInt()` 方法的第二个参数指定的，所以要解析十六进制的值，需如下调用 `parseInt()` 方法：

```
var iNum1 = parseInt("AF", 16); //returns 175
```

当然，对二进制、八进制，甚至十进制（默认模式），都可以这样调用 `parseInt()` 方法：

```
var iNum1 = parseInt("10", 2); //returns 2
var iNum2 = parseInt("10", 8); //returns 8
var iNum2 = parseInt("10", 10); //returns 10
```

如果十进制数包含前导 0，那么最好采用基数 10，这样才不会意外地得到八进制的值。例如：

```
var iNum1 = parseInt("010"); //returns 8
var iNum2 = parseInt("010", 8); //returns 8
var iNum3 = parseInt("010", 10); //returns 10
```

在这段代码中，两行代码都把字符串 "010" 解析成了一个数字。第一行代码把这个字符串看作八进制的值，解析它的方式与第二行代码（声明基数为 8）相同。最后一行代码声明基数为 10，所以 `iNum3` 最后等于 10。

23 `parseFloat()` 方法与 `parseInt()` 方法的处理方式相似，从位置 0 开始查看每个字符，直到找到第一个非有效的字符为止，然后把该字符之前的字符串转换成数字。不过，对于这个方法来说，第一个出现的小数点是有效字符。如果有两个小数点，第二个小数点将被看作无效的，`parseFloat()` 方法会把这个小数点之前的字符串转换成数字。这意味着字符串 "22.34.5" 将被解析成 22.34。

使用 `parseFloat()` 方法的另一不同之处在于，字符串必须以十进制形式表示浮点数，而不能用八进制形式或十六进制形式。该方法会忽略前导 0，所以八进制数 0908 将被解析为 908。对于十六进制数 0xA，该方法将返回 `NaN`，因为在浮点数中，x 不是有效字符。此外，`parseFloat()` 也没有基模式。

下面是使用 `parseFloat()` 方法的示例：

```
var fNum1 = parseFloat("1234blue"); //returns 1234.0
var fNum2 = parseFloat("0xA"); //returns NaN
var fNum3 = parseFloat("22.5"); //returns 22.5
var fNum4 = parseFloat("22.34.5"); //returns 22.34
var fNum5 = parseFloat("0908"); //returns 908
var fNum6 = parseFloat("blue"); //returns NaN
```

2.7.3 强制类型转换

还可使用强制类型转换（type casting）处理转换值的类型。使用强制类型转换可以访问特定的值，即使它是另一种类型的。ECMAScript 中可用的 3 种强制类型转换如下：

- `Boolean(value)`——把给定的值转换成 `Boolean` 型；
- `Number(value)`——把给定的值转换成数字（可以是整数或浮点数）；
- `String(value)`——把给定的值转换成字符串。

用这三个函数之一转换值，将创建一个新值，存放由原始值直接转换成的值。这会造成意想不到的后果。

当要转换的值是至少有一个字符的字符串、非 0 数字或对象（下一节将讨论这一点）时，`Boolean()` 函数将返回 `true`。如果该值是空字符串、数字 0、`undefined` 或 `null`，它将返回 `false`。可以用下面的代码段测试 `Boolean` 型的强制类型转换。

```
var b1 = Boolean("");           //false - empty string
var b2 = Boolean("hi");        //true - non-empty string
var b3 = Boolean(100);          //true - non-zero number
var b4 = Boolean(null);         //false - null
var b5 = Boolean(0);            //false - zero
var b6 = Boolean(new Object()); //true - object
```

`Number()` 的强制类型转换与 `parseInt()` 和 `parseFloat()` 方法的处理方式相似，只是它转换的是整个值，而不是部分值。还记得吗，`parseInt()` 和 `parseFloat()` 方法只转换第一个无效字符之前的字符串，因此 "4.5.6" 将被转换为 "4.5"。用 `Number()` 进行强制类型转换，"4.5.6" 将返回 `Nan`，因为整个字符串值不能转换成数字。如果字符串值能被完整地转换，`Number()` 将判断是调用 `parseInt()` 方法还是调用 `parseFloat()` 方法。下表说明了对不同的值调用 `Number()` 方法会发生的情况：

24

用法	结果
<code>Number(false)</code>	0
<code>Number(true)</code>	1
<code>Number(undefined)</code>	<code>Nan</code>
<code>Number(null)</code>	0
<code>Number("5.5")</code>	5.5
<code>Number("56")</code>	56
<code>Number("5.6.7")</code>	<code>Nan</code>
<code>Number(new Object())</code>	<code>Nan</code>
<code>Number(100)</code>	100

最后一种强制类型转换方法 `String()` 是最简单的，因为它可把任何值转换成字符串。要执行这种强制类型转换，只需要调用作为参数传递进来的值的 `toString()` 方法，即把 1 转换成 "1"，把 `true` 转换成 "true"，把 `false` 转换成 "false"，依此类推。强制转换成字符串和调用 `toString()` 方法的唯一不同之处在于，对 `null` 或 `undefined` 值强制类型转换可以生成字符串而不引发错误：

```
var s1 = String(null);    //"null"
var oNull = null;
var s2 = oNull.toString(); //won't work, causes an error
```

在处理 ECMAScript 这样的弱类型语言时，强制类型转换非常有用，不过应该确保使用值的正确。

2.8 引用类型

引用类型通常叫作类（class），也就是说，遇到引用值时，所处理的就是对象。本书将讨论大量的 ECMAScript 预定义引用类型。从现在起，将重点讨论与已经讨论过的原始类型紧密相关的引用类型。

从传统意义上来说，ECMAScript 并不真正具有类。事实上，除了说明不存在类，在 ECMA-262 中根本没有出现“类”这个词，ECMAScript 定义了“对象定义”，逻辑上等价于其他程序设计语言中的类。本书选择使用术语“类”，因为大多数开发者对此更熟悉一些。

对象是由 new 运算符加上要实例化的类的名字创建的，例如，下面代码创建了 Object 类的实例：

```
var o = new Object();
```

这种语法与 Java 语言的相似，不过当有不止一个参数时，ECMAScript 要求使用括号。如果没有参数，如前面代码所示，括号可以省略：

25

```
var o = new Object;
```

第 3 章将更深入地探讨对象及其行为。这一节的重点是具有等价的原始类型的引用类型。

尽管括号不是必需的，但为避免混乱，最好使用括号。

2.8.1 Object 类

Object 类自身用处不大，不过在了解其他类之前，还是应该先了解它。因为 ECMAScript 中的 Object 类与 Java 中的 `java.lang.Object` 相似，ECMAScript 中的所有类都由这个类继承而来，Object 类中的所有属性和方法都会出现在其他类中，所以理解了 Object 类，就可以更好地理解其他类。

Object 类具有下列属性：

- `Constructor`——对创建对象的函数的引用（指针）。对于 `Object` 类，该指针指向原始的 `Object()` 函数。
- `Prototype`——对该对象的对象原型的引用。第 3 章将进一步讨论原型。对于所有的类，它默认返回 `Object` 对象的一个实例。

Object 类还有几个方法：

- `HasOwnProperty(property)`——判断对象是否有某个特定的属性。必须用字符串指定该属性（例如，`o.hasOwnProperty("name")`）。
- `IsPrototypeOf(object)`——判断该对象是否为另一个对象的原型。
- `PropertyIsEnumerable(property)`——判断给定的属性是否可以用 `for...in` 语句（本章后面将讨论该语句）进行枚举。

- `ToString()`——返回对象的原始字符串表示。对于 `Object` 类，ECMA-262 没有定义这个值，所以不同的 ECMAScript 实现具有不同的值。
- `ValueOf()`——返回最适合该对象的原始值。对于许多类，该方法返回的值都与 `toString()` 的返回值相同。
- 上面列出的每种属性和方法都会被其他类覆盖。

26

2.8.2 Boolean类

`Boolean` 类是 `Boolean` 原始类型的引用类型。要创建 `Boolean` 对象，只需要传递 `Boolean` 值作为参数：

```
var oBooleanObject = new Boolean(true);
```

`Boolean` 对象将覆盖 `object` 类的 `valueOf()` 方法，返回原始值，即 `true` 或 `false`。`ToString()` 方法也会被覆盖，返回字符串“`true`”或“`false`”。遗憾的是，在 ECMAScript 中很少使用 `Boolean` 对象，即使使用，也不易理解。

问题通常出现在 `Boolean` 表达式中使用 `Boolean` 对象时。例如：

```
var oFalseObject = new Boolean(false);
var bResult = oFalseObject && true; //outputs true
```

在这段代码中，用 `false` 值创建 `Boolean` 对象。然后用这个值与原始值 `true` 进行 AND 操作。在 `Boolean` 运算中，`false` 和 `true` 进行 AND 操作的结果是 `false`。不过，在这行代码中，计算的是 `oFalseObject`，而不是它的值 `false`。正如前面讨论过的，在 `Boolean` 表达式中，所有对象都会被自动转换为 `true`，所以 `oFalseObject` 的值是 `true`。然后 `true` 再与 `true` 进行 AND 操作，结果为 `true`。

虽然你应该了解 `Boolean` 对象的可用性，不过最好还是使用 `Boolean` 原始值，避免发生这一节提到的问题。

2.8.3 Number类

正如你可能想到的，`Number` 类是 `Number` 原始类型的引用类型。要创建 `Number` 对象，采用下列代码：

```
var oNumberObject = new Number(55);
```

你应该已认出本章前面小节中讨论特殊值（如 `Number.MAX_VALUE`）时提到的 `Number` 类。所有特殊值都是 `Number` 类的静态属性。

要得到数字对象的 `Number` 原始值，只需要使用 `valueOf()` 方法：

```
var iNumber = oNumberObject.valueOf();
```

当然，`Number` 类也有 `toString()` 方法，在讨论类型转换的小节中已经详细讨论过该方法。

27 除从 Object 类继承的标准方法外，Number 类还有几个处理数值的专用方法。

toFixed()方法返回的是具有指定位数小数的数字的字符串表示。例如：

```
var oNumberObject = new Number(99);
alert(oNumberObject.toFixed(2)); //outputs "99.00"
```

这里，toFixed()方法的参数是 2，说明了应该显示几位小数。该方法将返回"99.00"，空的小数位由 0 补充。对于处理货币的应用程序，该方法非常有用。toFixed()方法能表示具有 0 到 20 位小数的数字，超出这个范围的值会引发错误。

与格式化数字相关的另一方法是 toExponential()，它返回的是用科学记数法表示的数字的字符串形式。与toFixed()方法相似，toExponential()方法也有一个参数，指定要输出的小数的位数。例如：

```
var oNumberObject = new Number(99);
alert(oNumberObject.toExponential(1)); //outputs "9.9e+1"
```

这段代码的结果是输出"9.9e+1"，前面解释过，它表示 9.9×10^1 。问题是，如果不知道要用哪种形式（预定形式或指数形式）表示数字怎么办？可以使用toPrecision()方法。

toPrecision()方法根据最有意义的形式来返回数字的预定形式或指数形式。它有一个参数，即用于表示数的数字总数（不包括指数）。例如：

```
var oNumberObject = new Number(99);
alert(oNumberObject.toPrecision(1)); //outputs "1e+2"
```

这段代码的任务是用一位数字表示数 99，结果为"1e+2"，以另外的形式表示即 100。的确，toPrecision()方法会对数进行舍入，从而得到尽可能接近真实值的数。由于用 2 位以下数字不可能表示 99，所以必须进行这样的舍入。不过，如果想用 2 位数字表示 99，就容易多了：

```
var oNumberObject = new Number(99);
alert(oNumberObject.toPrecision(2)); //outputs "99"
```

当然，输出的是"99"，因为这正是该数的准确表示。不过，如果指定的位数多于需要的位数又如何呢？

```
var oNumberObject = new Number(99);
alert(oNumberObject.toPrecision(3)); //outputs "99.0"
```

在这种情况下，toPrecision(3)等价于toFixed(1)，输出的是"99.0"。

toFixed()、toExponential()和toPrecision()方法都会进行舍入操作，以便用正确的小数位数正确地表示一个数。

与 Boolean 对象相似，Number 对象也很重要，不过应该少用这种对象，以避免发生潜在的问题。只要可能，都使用数字的原始表示法。

2.8.4 String 类

String 类是 String 原始类型的对象表示法，它是以下列方式创建的：

```
var oStringObject = new String("hello world");
```

String对象的 valueOf()方法和 toString()方法都会返回 String型的原始值：

```
alert(oStringObject.valueOf() == oStringObject.toString()); //outputs "true"
```

如果运行这段代码，输出是“true”，说明这些值真的相等。

String类是ECMAScript中的比较复杂的引用类型之一。同样，本节的重点只是String类的基本功能。更多的高级功能将分别在本书适合的主题中进行介绍。

String类具有属性length，它是字符串中的字符个数：

```
var oStringObject = new String("hello world");
alert(oStringObject.length); //outputs "11"
```

这个例子输出的是“11”，即“hello world”中的字符个数。注意，即使字符串包含双字节的字符（与ASCII字符相对，ASCII字符只占用一个字节），每个字符也只算一个字符。

String类还有大量的方法。首先，两个方法charAt()和charCodeAt()访问的是字符串中的单个字符。在2.6.6节中介绍过，第一个字符的位置是0，第二个字符的位置是1，依此类推。这两个方法都有一个参数，即要操作的字符的位置。charAt()方法返回的是包含指定位置处的字符的字符串：

```
var oStringObject = new String("hello world");
alert(oStringObject.charAt(1)); //outputs "e"
```

在字符串“hello world”中，位置1处的字符是“e”，因此调用charAt(1)返回的是“e”。如果想得到的不是字符，而是字符代码，那么可以调用charCodeAt()：

```
var oStringObject = new String("hello world");
alert(oStringObject.charCodeAt(1)); //outputs "101"
```

29

这个例子输出“101”，即小写字母“e”的字符代码。

接下来是concat()方法，用于把一个或多个字符串连接到String对象的原始值上。该方法返回的是String原始值，保持原始的String对象不变：

```
var oStringObject = new String("hello ");
var sResult = oStringObject.concat("world");
alert(sResult); //outputs "hello world"
alert(oStringObject); //outputs "hello "
```

在上面这段代码中，调用concat()方法返回的是“hello world”，而String对象存放的仍然是“hello”。出于这种原因，较常见的是用加号（+）连接字符串，因为这种形式从逻辑上表明了真正行为：

```
var oStringObject = new String("hello ");
var sResult = oStringObject + "world";
alert(sResult); //outputs "hello world"
alert(oStringObject); //outputs "hello "
```

迄今为止，已讨论过连接字符串的方法，访问字符串中的单个字符的方法。不过如果无法确

定在某个字符串中是否确实存在一个字符，应该调用什么方法呢？这时，可调用 `indexOf()` 和 `lastIndexOf()` 方法。

`indexOf()` 和 `lastIndexOf()` 方法返回的都是指定的子串在另一个字符串中的位置（或-1，如果没找到这个子串）。这两个方法的不同之处在于，`indexOf()` 方法是从字符串的开头（位置 0）开始检索子串，而 `lastIndexOf()` 则是从字符串的结尾开始检索子串的。例如：

```
var oStringObject = new String("hello world");
alert(oStringObject.indexOf("o")); //outputs "4"
alert(oStringObject.lastIndexOf("o")); //outputs "7"
```

这里，第一个“o”字符串出现在位置 4，即“hello”中的“o”；最后一个“o”字符串出现在位置 7，即“world”中的“o”。如果该字符串中只有一个“o”字符串，那么 `indexOf()` 和 `lastIndexOf()` 方法返回的位置相同。

下一个方法是 `localeCompare()`，对字符串值进行排序。该方法有一个参数——要进行比较的字符串，返回的是下列 3 个值之一：

- 如果 `String` 对象按照字母顺序排在参数中的字符串之前，返回负数（最常见的是-1，不过真正返回的值是由实现决定的）。
- 如果 `String` 对象等于参数中的字符串，返回 0。
- 如果 `String` 对象按照字母顺序排在参数中的字符串之后，返回正数（最常见的是 1，不过同样，真正返回的值是由实现决定的）。

示例如下：

```
var oStringObject = new String("yellow");
alert(oStringObject.localeCompare("brick")); //outputs "1"
alert(oStringObject.localeCompare("yellow")); //outputs "0"
alert(oStringObject.localeCompare("zoo")); //outputs "-1"
```

在这段代码中，字符串“yellow”与 3 个值进行了对比，即“brick”、“yellow”和“zoo”。由于按照字母顺序排列，“yellow”位于“brick”之后，所以 `localCompare()` 返回 1；“yellow”等于“yellow”，所以 `localCompare()` 返回 0；“zoo”位于“yellow”之后，`localCompare()` 返回-1。再强调一次，由于返回的值是由实现决定的，所以最好以下面的方式调用 `localCompare()`：

```
var oStringObject1 = new String("yellow");
var oStringObject2 = new String("brick");
var iResult = sTestString.localeCompare("brick");
if(iResult < 0) {
    alert(oStringObject1 + " comes before " + oStringObject2);
} else if (iResult > 0) {
    alert(oStringObject1 + " comes after " + oStringObject2);
} else {
    alert("The two strings are equal");
}
```

采用这种结构，可以确保这段代码在所有实现中都能正确运行。

`localCompare()` 的独特之处在于，实现所处的区域（`locale`，兼指国家/地区和语言）确切说明了这种方法运行的方式。在美国，英语是 ECMAScript 实现的标准语言，`localCompare()` 是

区分大小写的，大写字母在字母顺序上排在小写字母之后。不过，在其他区域情况可能并非如此。

ECMAScript 提供了两种方法从子串创建字符串值，即 `slice()` 和 `substring()`。这两种方法返回的都是要处理的字符串的子串，都接受一个或两个参数。第一个参数是要获取的子串的起始位置，第二个参数（如果使用的话）是要获取子串终止前的位置（也就是说，获取终止位置处的字符不包括在返回的值内）。如果省略第二个参数，终止位就默认为字符串的长度。与 `concat()` 方法一样，`slice()` 和 `substring()` 方法都不改变 `String` 对象自身的值。它们只返回原始的 `String` 值，保持 `String` 对象不变。

```
var oStringObject = new String("hello world");
alert(oStringObject.slice(3));           //outputs "lo world"
alert(oStringObject.substring(3));       //outputs "lo world"
alert(oStringObject.slice(3, 7));        //outputs "lo w"
alert(oStringObject.substring(3,7));     //outputs "lo w"
```

在这个例子中，`slice()` 和 `substring()` 方法的用法相同，返回的值也一样。当只有参数 3 时，两个方法返回的都是 "lo world"，因为 "hello" 中的第二个 "l" 位于位置 3 上。当有两个参数 3 和 7 时，两个方法返回的都是 "lo w"（"world" 中的字母 "o" 位于位置 7 上，所以它不包括在结果中）。为什么有两个功能完全相同的方法呢？事实上，这两个方法并不完全相同，不过只在参数为负数时，它们处理参数的方式才稍有不同。

对于负数参数，`slice()` 方法会用字符串的长度加上参数，`substring()` 方法则将其作为 0 处理（也就是说将忽略它）。例如：

```
var oStringObject= new String("hello world");
alert(oStringObject.slice(-3));          //outputs "rld"
alert(oStringObject.substring(-3));      //outputs "hello world"
alert(oStringObject.slice(3, -4));       //outputs "lo w"
alert(oStringObject.substring(3,-4));    //outputs "hel"
```

31

这样即可看出 `slice()` 和 `substring()` 方法的主要不同。当只有参数 -3 时，`slice()` 返回 "rld"，`substring()` 则返回 "hello world"。这是因为对于字符串 "helloworld"，`slice(-3)` 将被转换成 `slice(8)`，而 `substring(-3)` 则被转换成 `substring(0)`。同样，使用参数 3 和 -4 时，差别也很明显。`slice()` 方法将被转换成 `slice(3,7)`，与前面的例子相同，返回 "lo w"。而 `substring()` 方法则将这两个参数解释为 `substring(3,0)`，实际上即 `substring(0,3)`，因为 `substring()` 总是把较小的数字作为起始位，较大的数字作为终止位。因此，`substring(3,-4)` 返回的是 "hel"。这里的最后一行代码用来说明如何使用这些方法。

最后一套要讨论的方法涉及大小写转换。有 4 种方法用于执行大小写转换，即 `toLowerCase()`、`toLocaleLowerCase()`、`toUpperCase()` 和 `toLocaleUpperCase()`。从名字上可以看出它们的用途，前两种方法用于把字符串转换成全小写的，后两种方法用于把字符串转换成全大写的。`toLowerCase()` 和 `toUpperCase()` 方法是原始的，是以 `java.lang.String` 中的相同方法为原型实现的。`toLocaleLowerCase()` 和 `toLocaleUpperCase()` 方法是基于特定的区域实现的（与 `localeCompare()` 的用法相同）。在许多区域中，区域特定的方法都与通用的方法完全相同。不过，有几种语言对 Unicode 大小写转换应用了特定的规则（例如土耳其语），因此必须使

用区域特定的方法才能进行正确的转换。

```
var oStringObject= new String("Hello World");
alert(oStringObject.toLocaleUpperCase()); //outputs "HELLO WORLD"
alert(oStringObject.toUpperCase()); //outputs "HELLO WORLD"
alert(oStringObject.toLocaleLowerCase()); //outputs "hello world"
alert(oStringObject.toLowerCase()); //outputs "hello world"
```

这段代码中，`toUpperCase()`和`toLocaleUpperCase()`方法输出的都是"HELLO WORLD"，`toLowerCase()`和`toLocaleLowerCase()`方法输出的都是"hello world"。一般说来，如果不知道在以哪种编码运行一种语言，则使用区域特定的方法比较安全。

记住，`String`类的所有属性和方法都可应用于`String`原始值上，因为它们是伪对象。

2.8.5 instanceof 运算符

在使用`typeof`运算符时采用引用类型存储值会出现一个问题，无论引用的是什么类型的对象，它都返回"object"。ECMAScript引入了另一个Java运算符`instanceof`来解决这个问题。

`instanceof`运算符与`typeof`运算符相似，用于识别正在处理的对象的类型。与`typeof`方法不同的是，`instanceof`方法要求开发者明确地确认对象为某特定类型。例如：

```
32
var oStringObject = new String("hello world");
alert(oStringObject instanceof String); //outputs "true"
```

这段代码问的是“变量`oStringObject`是否为`String`类的实例？”`oStringObject`的确是`String`类的实例，因此结果是"true"。尽管不像`typeof`方法那样灵活，但是在`typeof`方法返回"object"的情况下，`instanceof`方法还是很有用的。

2.9 运算符

ECMA-262 定义了一套用于操作变量的运算符。运算符的范围从算术运算符（如加号和减号）和位运算符到关系运算符和等性运算符。对值执行的原始动作在任何时候都被看作运算符。

2.9.1 一元运算符

一元运算符只有一个参数，即要操作的对象或值。它们是ECMAScript中最简单的运算符。

1. `delete`

`delete`运算符删除对以前定义的对象属性或方法的引用。例如：

```
var o = new Object;
o.name = "Nicholas";
alert(o.name); //outputs "Nicholas"
delete o.name;
alert(o.name); //outputs "undefined"
```

这个例子中，删除了`name`属性，意味着强制解除对它的引用，将其设置为`undefined`（即

创建的未初始化的变量的值)。

`delete` 运算符不能删除开发者未定义的属性和方法。例如，下面的代码将引发错误：

```
delete o.toString;
```

即使 `toString` 是有效的方法名，这行代码也会引发错误，因为 `toString()` 方法是原始的 ECMAScript 方法，不是开发者定义的。

2. void

`void` 运算符对任何值都返回 `undefined`。该运算符通常用于避免输出不应该输出的值，例如，从 HTML 的`<a>`元素调用 JavaScript 函数时。要正确做到这一点，函数不能返回有效值，否则浏览器将清空页面，只显示函数的结果。例如：

```
<a href="javascript:window.open('about:blank')">Click Me</a>
```

如果把这行代码放入到 HTML 页面，点击其中的链接，即可看到屏幕上显示“[object]”(如图 2-3 所示)。这是因为 `window.open()` 方法(第 5 章将对该方法和其他与窗口有关的方法做进一步讨论)返回了对新打开的窗口的引用。然后该对象将被转换成要显示的字符串。



图 2-3

要避免这种结果，可以用 `void` 运算符调用 `window.open()` 函数：

```
<a href="javascript:void(window.open('about:blank'))">Click Me</a>
```

这使 `window.open()` 调用返回 `undefined`，它不是有效的值，不会显示在浏览器窗口中。记住，没有返回值的函数真正返回的都是 `undefined`。

3. 前增量/前减量运算符

直接从 C(和 Java) 借用的两个运算符是前增量运算符和前减量运算符。所谓前增量运算符，就是在数值上加 1，形式是在变量前放两个加号 (`++`)：

```
var iNum = 10;  
++iNum
```

第二行代码把 `iNum` 增加到了 11，它实质上等价于：

```
var iNum = 10;  
iNum = iNum + 1;
```

同样，前减量运算符是从数值上减 1，形式是在变量前放两个减号 (--)：

```
var iNum = 10;
--iNum;
```

在这个例子中，第二行代码把 iNum 的值减到 9。

34 在使用前缀式运算符时，注意增量和减量运算都发生在计算表达式之前。考虑下面的例子：

```
var iNum = 10;
--iNum;
alert(iNum);           //outputs "9"
alert(--iNum);         //outputs "8"
alert(iNum);           //outputs "8"
```

第二行代码对 iNum 进行减量运算，第三行代码显示的结果是 ("9")。第四行代码又对 iNum 进行减量运算，不过这次前减量运算和输出操作出现在同一个语句中，显示的结果是 "8"。为了证明已实现了所有的减量操作，第五行代码又输出一次 "8"。

在算术表达式中，前增量和前减量运算符的优先级是相同的，因此要按照从左到右的顺序计算之。例如：

```
var iNum1 = 2;
var iNum2 = 20;
var iNum3 = --iNum1 + ++iNum2;    //equals 22
var iNum4 = iNum1 + iNum2;        //equals 22
```

在前面的代码中，iNum3 等于 22，因为表达式要计算的是 $1+21$ 。变量 iNum4 也等于 22，也是 $1+21$ 。

4. 后增量/后减量运算符

还有两个直接从 C (和 Java) 借用的运算符，即后增量运算符和后减量运算符。后增量运算符也是给数值加 1，形式为在变量后加两个加号 (++)：

```
var iNum = 10;
iNum++
```

不出所料，后减量运算符也是从数值上减 1，形式为在变量后加两个减号 (--)：

```
var iNum = 10;
iNum--;
```

第二行代码把 iNum 的值减到 9。

与前缀式运算符不同的是，后缀式运算符是在计算过包含它们的表达式后才进行增量或减量运算的。考虑下面的例子：

```
var iNum = 10;
iNum--;
alert(iNum);           //outputs "9"
alert(iNum--);          //outputs "9"
alert(iNum);           //outputs "8"
```

与前缀式运算符的例子相似，第二行代码对 iNum 进行减量运算，第三行代码显示结果 ("9")。

第四行代码再次显示 `iNum` 的值，不过这次是在同一个语句中应用后减量运算符。由于减量运算发生在计算过表达式之后，所以这条语句显示的数是 "9"。执行了第五行代码后，`alert` 函数显示的是 "8"，因为在执行第四行代码之后和执行第五行代码之前，执行了后减量运算。

在算术表达式中，后增量和后减量运算符的优先级是相同的，因此要按照从左到右的顺序计算之。例如：

```
var iNum1 = 2;
var iNum2 = 20;
var iNum3 = iNum1-- + iNum2++; //equals 22
var iNum4 = iNum1 + iNum2; //equals 22
```

在上面的代码中，`iNum3` 等于 22，因为表达式要计算的是 $2+20$ 。变量 `iNum4` 也等于 22，不过它计算的是 $1+21$ ，因为增量和减量运算都在给 `iNum3` 赋值后才发生。

5. 一元加法和一元减法

大多数人都熟悉一元加法和一元减法，它们在 ECMAScript 中的用法与高中数学中学到的用法相同。一元加法本质上对数字无任何影响：

```
var iNum= 25;
iNum = +iNum;
alert(iNum); //outputs "25"
```

这段代码对数字 25 应用了一元加法，返回的还是 25。尽管一元加法对数字无作用，但对字符串却有有趣的效果，会把字符串转换成数字。

```
var sNum = "25";
alert(typeof sNum); //outputs "string"
var iNum = +sNum;
alert(typeof iNum); //outputs "number"
```

这段代码把字符串 "25" 转换成真正的数字。当一元加法运算符对字符串进行操作时，它计算字符串的方式与 `parseInt()` 相似，主要的不同是只有对以 "0x" 开头的字符串（表示十六进制数字），一元运算符才把它转换成十进制的值。因此，用一元加法转换 "010"，得到的总是 10，而 "0xB" 将被转换成 11。

另一方面，一元减法就是对数值求负（例如把 25 转换成 -25）：

```
var iNum= 25;
iNum = -iNum;
alert(iNum); //outputs "-25"
```

与一元加法运算符相似，一元减法运算符也会把字符串转换成近似的数字，此外还会对该值求负。例如：

```
var sNum = "25";
alert(typeof sNum); //outputs "string"
var iNum = -sNum;
alert(iNum); //outputs "-25"
alert(typeof iNum); //outputs "number"
```

在上面的代码中，一元减法运算符将把字符串 "25" 转换成 -25（一元减法运算符对十六进制

值和十进制的处理方式与一元加法运算符相似，只是它还会对该值求负)。

2.9.2 位运算符

下面这套运算符是在数字底层（即表示数字的 32 个数位）进行操作的。探讨这些运算符之前，首先详细介绍一下 ECMAScript 中的整数。

1. 重温整数

ECMAScript 整数有两种类型，即有符号整数（允许用正数和负数）和无符号整数（只允许用正数）。在 ECMAScript 中，所有整数字面量默认都是有符号整数。这到底意味着什么呢？

有符号整数使用前 31 位表示整数的数值，用第 32 位表示整数的符号，0 表示正数，1 表示负数。数值的范围从 -2147483648 到 2147483647。

可以以两种不同的方式存储二进制形式的有符号整数，一种用于存储正数，一种用于存储负数。正数是以真二进制形式存储的，前 31 位中的每一位都表示 2 的幂，从第 1 位（位 0）开始，表示 2^0 ，第 2 位（位 1）表示 2^1 ，依此类推。没有用到的位由 0 填充，即忽略不计。例如，图 2-4 展示的是数 18 的表示法。

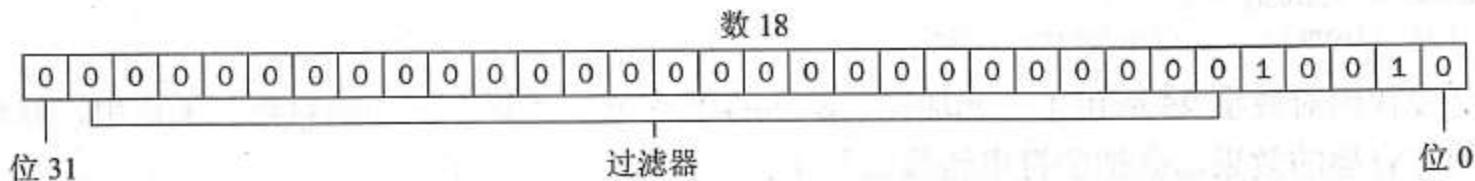


图 2-4

18 的二进制版本只用了前 5 位，它们是这个数字的有效位。把数字转换成二进制字符串（如前所述），就只能看到有效位：

```
var iNum = 18;
alert(iNum.toString(2)); //outputs "10010"
```

37

这段代码只输出“10010”，而不是 18 的 32 位表示。其他的数位并不重要，因为仅使用前 5 位即可确定这个十进制数值（如图 2-5 所示）。

1	0	0	1	0
---	---	---	---	---

$$(2^4 \times 1) + (2^3 \times 0) + (2^2 \times 0) + (2^1 \times 1) + (2^0 \times 0)$$

$$16 + 0 + 0 + 2 + 0$$

18

图 2-5

负数也存储为二进制代码，不过采用的形式是二进制补码。计算数字二进制补码的步骤有三步：

(1) 确定该数字的非负版本的二进制表示（例如，要计算 -18 的二进制补码，首先要确定 18 的二进制表示）；

(2) 求得二进制反码，即要把 0 替换为 1，把 1 替换为 0；

(3) 在二进制反码上加 1。

要确定 -18 的二进制表示，首先必须得到 18 的二进制表示，如下所示：

```
0000 0000 0000 0000 0000 0001 0010
```

接下来，计算二进制反码，如下所示：

```
1111 1111 1111 1111 1111 1110 1101
```

最后，在二进制反码上加 1，如下所示：

```
1111 1111 1111 1111 1111 1110 1101
```

1

```
1111 1111 1111 1111 1111 1110 1110
```

因此，-18 的二进制表示即 1111 1111 1111 1111 1111 1111 1110 1110。记住，在处理有符号整数时，开发者不能访问位 31。

有趣的是，把负整数转换成二进制字符串后，ECMAScript 并不以二进制补码的形式显示，而是用数字绝对值的标准二进制代码前加负号的形式输出。例如：

```
var iNum = -18;
alert(iNum.toString(2)); //outputs "-10010"
```

这段代码输出的是 "-10010"，而非二进制补码，这是为避免访问位 31。为了简便，ECMAScript 用一种简单的方式处理整数，使得开发者不必关心它们的用法。38

另一方面，无符号整数把最后一位作为另一个数位处理。在这种模式中，第 32 位不表示数字的符号，而是值 2^{31} 。由于这个额外的位，无符号整数的数值范围为 0 到 4294967295。对于小于等于 2147483647 的整数来说，无符号整数看来与有符号整数一样，而大于 2147483647 的整数则要使用位 31（在有符号整数中，这一位总是 0）。把无符号整数转换成字符串后，只返回它们的有效位。

记住，所有整数字面量都默认存储为有符号整数。只有用 ECMAScript 的位运算符才能创建无符号整数。

2. 位运算 NOT

位运算 NOT 由否定号 (~) 表示，它是 ECMAScript 中为数不多的与二进制算术有关的运算符之一。位运算 NOT 是三步的处理过程：

- (1) 把运算数转换成 32 位数字；
- (2) 把二进制形式转换成它的二进制反码；
- (3) 把二进制反码转换成浮点数。

例如：

```
var iNum1 = 25;           //25 is equal to 000000000000000000000000000011001
var iNum2 = ~iNum1;       //convert to 1111111111111111111111111100110
alert(iNum2);             //outputs "-26"
```

位运算 NOT 实质上是对数字求负，然后减 1，因此 25 变为 -26。用下面的方法也可以得到同样的效果：

```
var iNum1 = 25;
var iNum2 = -iNum1 - 1;
alert(iNum2); //outputs "-26"
```

3. 位运算AND

位运算 AND 由和号 (&) 表示，直接对数字的二进制形式进行运算。它把每个数字中的数位对齐，然后用下面的规则对同一位置上的两个数位进行 AND 运算：

第一个数字中的数位	第二个数字中的数位	结 果
1	1	1
1	0	0
0	1	0
0	0	0

39

例如，要对数字 25 和 3 进行 AND 运算，代码如下所示：

```
var iResult = 25 & 3;
alert(iResult); //outputs "1"
```

25 和 3 进行 AND 运算的结果是 1。为什么？分析如下：

```
25 = 0000 0000 0000 0000 0000 0001 1001
3 = 0000 0000 0000 0000 0000 0000 0011
-----
AND = 0000 0000 0000 0000 0000 0000 0001
```

可以看到，在 25 和 3 中，只有一个数位（位 0）存放的都是 1，因此，其他数位生成的都是 0，所以结果为 1。

4. 位运算OR

位运算 OR 由符号 (|) 表示，也是直接对数字的二进制形式进行运算。在计算每个位时，OR 采用下列规则：

第一个数字中的数位	第二个数字中的数位	结 果
1	1	1
1	0	1
0	1	1
0	0	0

仍然使用 AND 运算所用的例子，对 25 和 3 进行 OR 运算，代码如下：

```
var iResult = 25 | 3;
alert(iResult); //outputs "27"
```

25 和 3 进行 OR 运算的结果是 27：

```

25 = 0000 0000 0000 0000 0000 0001 1001
3 = 0000 0000 0000 0000 0000 0000 0011
-----
OR = 0000 0000 0000 0000 0000 0001 1011

```

可以看到，在两个数字中，共有 4 个数位存放的是 1，这些数位被传递给结果。二进制代码 11011 等于 27。

5. 位运算XOR

位运算 XOR 由符号 (^) 表示，当然，也是直接对二进制形式进行运算。XOR 不同于 OR，当只有一个数位存放的是 1 时，它才返回 1。真值表如下：

40

第一个数字中的数位	第二个数字中的数位	结 果
1	1	0
1	0	1
0	1	1
0	0	0

对 25 和 3 进行 XOR 运算，代码如下：

```

var iResult = 25 ^ 3;
alert(iResult); //outputs "26"

```

25 和 3 进行 XOR 运算的结果是 26：

```

25 = 0000 0000 0000 0000 0000 0001 1001
2 = 0000 0000 0000 0000 0000 0000 0011
-----
XOR = 0000 0000 0000 0000 0000 0001 1010

```

可以看到，两个数字中共有 4 个数位存放的是 1，它们被传递给结果。二进制代码 11010 等于 26。

6. 左移运算

左移运算由两个小于号表示 (<<)。它把数字中的所有数位向左移动指定的数量。例如，把数字 2（等于二进制中的 10）左移 5 位，结果为 64（等于二进制中的 1000000）：

```

var iOld = 2;           //equal to binary 10
var iNew = iOld << 5;   //equal to binary 1000000 which is decimal 64

```

注意，在左移数位时，数字右边多出 5 个空位。左移运算用 0 填充这些空位，使结果为完整的 32 位数字（如图 2-6 所示）。

41

注意，左移操作保留数字的符号位。例如，如果把 -2 左移 5 位，得到的是 -64，而不是 64。“符号仍然存储在第 32 位中吗？”是的，不过这在 ECMAScript 后台进行，开发者不能直接访问第 32 个数位。即使输出二进制字符串形式的负数，显示的也是负号形式（例如，-2 将显示为 -10，而不是 111111111111111111111111111110）。

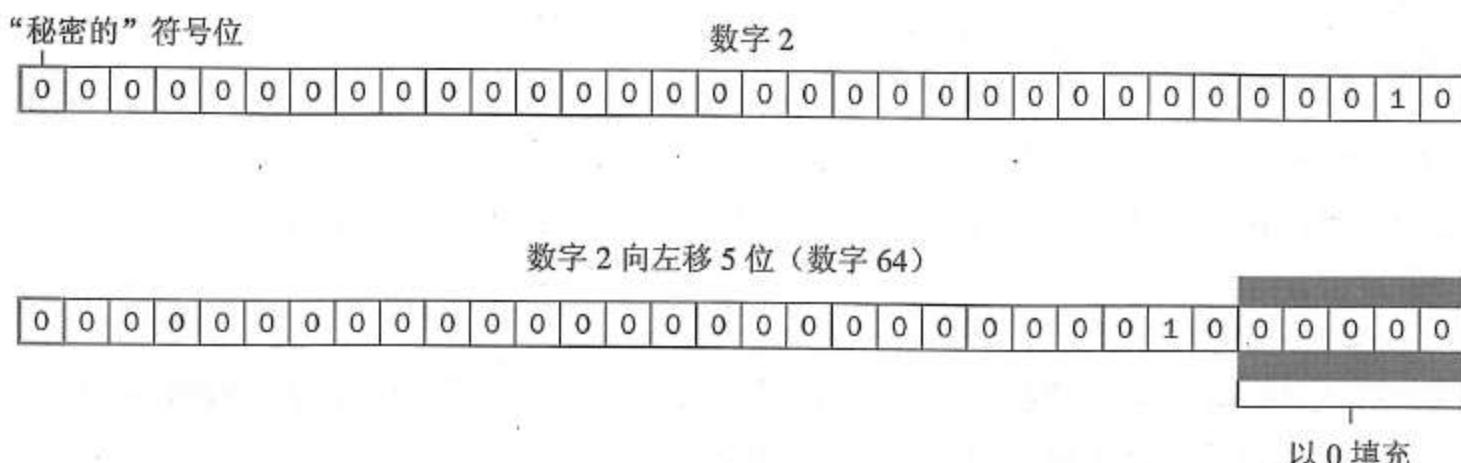


图 2-6

7. 有符号右移运算

有符号右移运算符由两个大于号 (`>>`) 表示，它将把 32 位数字中的所有数位整体右移，同时保留该数的符号（正号或负号）。有符号右移运算恰好与左移运算相反。例如，把 64 右移 5 位，将变为 2：

```
var iOld = 64;           //equal to binary 1000000
var iNew = iOld >> 5;    //equal to binary 10 with is decimal 2
```

同样，移动数位后会造成空位。这次，空位位于数字的左侧，但位于符号位之后（如图 2-7 所示）。ECMAScript 用符号位的值填充这些空位，创建完整的数字。



图 2-7

8. 无符号右移运算

无符号右移由三个大于号 (`>>>`) 表示，它将把无符号 32 位数中的所有数位整体右移。对于正数，无符号右移运算的结果与有符号右移运算一样。用有符号右移运算中的例子，把 64 右移 5 位，将变为 2：

```
var iOld = 64;           //equal to binary 1000000
var iNew = iOld >>> 5;   //equal to binary 10 with is decimal 2
```

对于负数，情况就不同了。无符号右移运算用 0 填充所有空位。对于正数，这与有符号右移运算的操作完全一样，而负数则被作为正数来处理。由于无符号右移运算的结果是一个 32 位的正数，所以负数的无符号右移运算得到的总是一个非常大的数字。例如，如果把 -64 右移 5 位，将得到 134217726。如何得到这种结果的呢？

要实现这一点，需要把这个数字转换成无符号的等价形式（尽管该数字本身还是有符号的），可以通过以下代码获得这种形式：

```
var iUnsigned64 = -64 >>> 0;
```

然后，用 Number 类型的 `toString()` 方法获取它的真正的位表示，采用的基为 2：

```
alert(iUnsigned64.toString(2));
```

这将生成 11111111111111111111111111000000，即有符号整数-64 的二进制补码表示，不过它等于无符号整数 4294967232。出于这种原因，使用无符号右移运算符要小心。

2.9.3 Boolean 运算符

Boolean 运算符与等式运算符同等重要，是它们使得程序设计语言得以正常运行。如果不能测试两个值之间的关系，那么像 if...else 和循环这样的语句就没用了。Boolean 运算符有三种，即 NOT、AND 和 OR。

1. 逻辑NOT

在 ECMAScript 中，逻辑 NOT 运算符与 C 和 Java 中的逻辑 NOT 运算符相同，都由感叹号 (!) 表示。与逻辑 OR 和逻辑 AND 运算符不同的是，逻辑 NOT 运算符返回的一定是 Boolean 值。逻辑 NOT 运算符的行为如下：

- 如果运算数是对象，返回 false。
 - 如果运算数是数字 0，返回 true。
 - 如果运算数是 0 以外的任何数字，返回 false。
 - 如果运算数是 null，返回 true。
 - 如果运算数是 NaN，返回 true。
 - 如果运算数是 undefined，发生错误。

通常，该运算符用于控制循环（后面有相关的讨论）：

```
var bFound = false;  
var i = 0;  
  
while (!bFound) {  
    if (aValues[i] == vSearchValue) {  
        bFound = true;  
    } else {  
        i++;  
    }  
}
```

在这个例子中， Boolean 变量（bFound）用于记录检索是否成功。找到问题中的数据项时， bFound 将被设置为 true， !bFound 将等于 false，意味着运行将跳出 while 循环。

判断 ECMAScript 变量的 Boolean 值时，也可以使用逻辑 NOT 运算符。这样做需要在一行代码中使用两个逻辑 NOT 运算符。无论运算数是什么类型的，第一个 NOT 运算符返回 Boolean 值。

第二个 NOT 将对该 Boolean 值求负，从而给出变量真正的 Boolean 值。

```
var bFalse = false;
var sBlue = "blue";
var iZero = 0;
var iThreeFourFive = 345;
var oObject = new Object();
document.write("The Boolean value of bFalse is " + (!!bFalse));
document.write("<br />The Boolean value of sBlue is " + (!!sBlue));
document.write("<br />The Boolean value of iZero is " + (!!iZero));
document.write("<br />The Boolean value of iThreeFourFive is " +
  (!!iThreeFourFive));
document.write("<br />The Boolean value of oObject is " + (!!oObject));
```

运行这个例子，生成的输出如下所示：

```
The Boolean value of bFalse is false
The Boolean value of sBlue is true
The Boolean value of iZero is false
The Boolean value of iThreeFourFive is true
The Boolean value of oObject is true
```

2. 逻辑AND运算符

在 ECMAScript 中，逻辑 AND 运算符用双和号 (`&&`) 表示：

```
var bTrue = true;
var bFalse = false;
var bResult = bTrue && bFalse;
```

下面的真值表描述了逻辑 AND 运算符的行为：

运算数 1	运算数 2	结 果
true	true	true
true	false	false
false	true	false
false	false	false

逻辑 AND 运算的运算数可以是任何类型的，不止是 Boolean 值。如果某个运算数不是原始的 Boolean 型值，逻辑 AND 运算并不一定返回 Boolean 值：

- 如果一个运算数是对象，另一个是 Boolean 值，返回该对象。
- 如果两个运算数都是对象，返回第二个对象。
- 如果某个运算数是 `null`，返回 `null`。
- 如果某个运算数是 `Nan`，返回 `Nan`。
- 如果某个运算数是 `undefined`，发生错误。

与 Java 中的逻辑 AND 运算相似，ECMAScript 中的逻辑 AND 运算也是简便运算，即如果第一个运算数决定了结果，就不再计算第二个运算数。对于逻辑 AND 运算来说，如果第一个运算数是 `false`，那么无论第二个运算数的值是什么，结果都不可能等于 `true`。考虑下面的例子：

```
var bTrue = true;
var bResult = (bTrue && bUnknown); //error occurs here
alert(bResult); //this line never executes
```

这段代码在进行逻辑 AND 运算时将引发错误，因为变量 `bUnknown` 是未定义的。变量 `bTrue` 的值为 `true`，因此逻辑 AND 运算将继续计算变量 `bUnknown`。这样做就会引发错误，因为 `bUnknown` 的值是 `undefined`，不能用于逻辑 AND 运算。如果修改这个例子，把第一个运算数设为 `false`，那么就不会发生错误：

```
var bFalse = false;
var bResult = (bFalse && bUnknown);
alert(bResult); //outputs "false"
```

在这段代码中，脚本将输出逻辑 AND 运算返回的值，即字符串 "`false`"。即使变量 `bUnknown` 的值为 `undefined`，它也不会被计算，因为第一个运算数的值是 `false`。在使用逻辑 AND 运算符时，必须记住它的这种简便计算特性。

3. 逻辑OR运算符

ECMAScript 中的逻辑 OR 运算符与 Java 中的相同，都由双竖线 (`||`) 表示：

```
var bTrue = true;
var bFalse = false;
var bResult = bTrue || bFalse;
```

下面的真值表描述了逻辑 OR 运算符的行为：

运算数 1	运算数 2	结果
<code>true</code>	<code>true</code>	<code>true</code>
<code>true</code>	<code>false</code>	<code>true</code>
<code>false</code>	<code>true</code>	<code>true</code>
<code>false</code>	<code>false</code>	<code>false</code>

45

与逻辑 AND 运算符相似，如果某个运算数不是 Boolean 值，逻辑 OR 运算并不一定返回 Boolean 值：

- 如果一个运算数是对象，另一个是 Boolean 值，返回该对象。
- 如果两个运算数都是对象，返回第一个对象。
- 如果某个运算数是 `null`，返回 `null`。
- 如果某个运算数是 `Nan`，返回 `Nan`。
- 如果某个运算数是 `undefined`，发生错误。

与逻辑 AND 运算符一样，逻辑 OR 运算也是简便运算。对于逻辑 OR 运算符来说，如果第一个运算数值为 `true`，就不再计算第二个运算数。例如：

```
var bTrue = true;
var bResult = (bTrue || bUnknown);
alert(bResult); //outputs "true"
```

与前面的例子相同，变量 `bUnknown` 是未定义的。不过，由于变量 `bTrue` 的值为 `true`，

bUnknown 不会被计算，因此输出的是 "true"。如果把 bTrue 的值改为 false，将发生错误：

```
var bFalse = false;
var bResult = (bTrue || bUnknown);      //error occurs here
alert(bResult);                      //this line never executes
```

2.9.4 乘性运算符

本节讨论的是三种乘性运算符，即乘法运算符、除法运算符和取模运算符。这些运算符与 Java、C、Perl 等语言中的同类运算符的运算方式相似，不过它们还具有一些自动的类型转换功能，这一点需要注意。

1. 乘法运算符

乘法运算符由星号 (*) 表示，如你所料，用于两个数相乘。ECMAScript 中的乘法语法与 C 语言中的相同：

```
var iResult = 34 * 56;
```

不过，在处理特殊值时，ECMAScript 中的乘法还有一些特殊行为：

- 如果运算数都是数字，执行常规的乘法运算，即两个正数或两个负数相乘结果为正数，两个运算数符号不同，结果为负数。如果结果太大或太小，那么生成的结果就是 Infinity 或 -Infinity。
- 如果某个运算数是 NaN，结果为 NaN。
- Infinity 乘以 0，结果为 NaN。
- Infinity 乘以 0 以外的任何数字，结果为 Infinity 或 -Infinity，由第二个运算数的符号决定。
- Infinity 乘以 Infinity，结果为 Infinity。

2. 除法运算符

除法运算符由斜线 (/) 表示，用第二个运算数除第一个运算数：

```
var iResult = 66 / 11;
```

与乘法运算符相似，对于特殊的值，除法运算符也有特殊行为：

- 如果运算数都是数字，执行常规的除法运算，即两个正数或两个负数结果为正数，两个运算数符号不同，结果为负数。如果结果太大或太小，那么生成的结果就是 Infinity 或 -Infinity。
- 如果某个运算数是 NaN，结果为 NaN。
- Infinity 被 Infinity 除，结果为 NaN。
- Infinity 被任何数字除，结果为 Infinity。
- 0 除一个非无穷大的数字，结果为 NaN。
- Infinity 被 0 以外的任何数字除，结果为 Infinity 或 -Infinity，由第二个运算数的符号决定。

3. 取模运算符

取模（余数）运算符由百分号（%）表示，使用方式如下：

```
var iResult = 26 % 5; //equal to 1
```

与其他乘性运算符相似，对于特殊的值，取模运算符也有特殊行为：

- 如果运算数都是数字，执行常规的算术除法运算，返回除法运算得到的余数。
- 如果被除数是 `Infinity`，或者除数是 0，结果为 `Nan`。
- `Infinity` 被 `Infinity` 除，结果为 `Nan`。
- 如果除数是无穷大的数，结果为被除数。
- 如果被除数为 0，结果为 0。

2.9.5 加性运算符

在程序设计语言中，加性运算符（即加号和减号）通常是最简单的数学运算符。不过在 ECMAScript 中，每个加性运算符都有大量的特殊行为。

1. 加法运算符

加法运算符（+）的用法如你所料：

```
var iResult = 1 + 2;
```

47

与乘性运算符一样，在处理特殊值时，加性运算符也有一些特殊方式。如果两个运算数都是数字，将执行算术加法，根据加法规则返回结果。

- 某个运算数是 `Nan`，结果为 `Nan`。
- `Infinity` 加 `Infinity`，结果为 `Infinity`。
- `-Infinity` 加 `-Infinity`，结果为 `-Infinity`。
- `Infinity` 加 `-Infinity`，结果为 `Nan`。
- `+0` 加 `+0`，结果为 `+0`。
- `-0` 加 `+0`，结果为 `+0`。
- `-0` 加 `-0`，结果为 `-0`。

不过，如果某个运算数是字符串，那么采用下列规则：

- 如果两个运算数都是字符串，把第二个字符串连接到第一个字符串上。
- 如果只有一个运算数是字符串，把另一个运算数转换成字符串，结果是两个字符串连接成的字符串。

例如：

```
var result1 = 5 + 5; //two numbers
alert(result); //outputs "10"
var result2 = 5 + "5"; //a number and a string
alert(result); //outputs "55"
```

这段代码说明了加法运算符的两种模式之间的差别。正常情况下，`5+5` 等于 10（原始数值），

如上面代码段中的前两行代码所示。不过，如果把一个运算数改为字符串 "5"，那么结果将变为 "55"（原始的字符串值），因为另一个运算数也会被转换成字符串。

为避免 JavaScript 中的一种常见错误，在使用加法运算符时，一定要仔细检查运算数的数据类型。

2. 减法运算符

减法运算符 (-) 是另一个十分常用的运算符：

```
var iResult = 2 - 1;
```

与加法运算符一样，减法运算符也有特殊的规则以处理 ECMAScript 中的各种类型转换：

48

- 如果两个运算数都是数字，将执行算术减法，返回结果。
- 某个运算数是 NaN，结果为 NaN。
- Infinity 减 Infinity，结果为 NaN。
- -Infinity 减 -Infinity，结果为 NaN。
- Infinity 减 -Infinity，结果为 Infinity。
- -Infinity 减 Infinity，结果为 -Infinity。
- +0 减 +0，结果为 +0。
- -0 减 -0，结果为 -0。
- -0 减 +0，结果为 +0。
- 某个运算数不是数字，结果为 NaN。

2.9.6 关系运算符

关系运算符小于 (<)、大于 (>)、小于等于 (<=) 和大于等于 (>=) 执行的是两个数的比较运算，比较方式与算术比较运算相同。每个关系运算符都返回一个 Boolean 值：

```
var bResult1 = 5 > 3;      //true
var bResult2 = 5 < 3;      //false
```

不过，对两个字符串应用关系运算符，它们的行为则不同。许多人认为小于表示“在字母顺序上靠前”，大于表示“在字母顺序上靠后”，但事实并非如此。对于字符串，第一个字符串中每个字符的代码都会与第二个字符串中对应位置上的字符的代码进行数值比较。完成这种比较操作后，返回一个 Boolean 值。问题在于大写字母的代码都小于小写字母的代码，这就意味着可能会遇到下列情况：

```
var bResult = "Brick" < "alphabet";
alert(bResult);    //outputs "true"
```

在这个例子中，字符串 "Brick" 小于字符串 "alphabet"，因为字母 B 的字符代码是 66，字母 a 的字符代码是 97。要强制性得到按照真正的字母顺序比较的结果，必须把两个运算数转换成相同的大小写形式（全大写或全小写的），然后再进行比较：

```
var bResult = "Brick".toLowerCase() < "alphabet".toLowerCase();
alert(bResult); //outputs "false"
```

把两个运算数都转换成小写的，确保了能正确识别出“alphabet”在字母顺序上位于“Brick”之前。

另一种棘手的状况发生在比较两个字符串形式的数字时，例如：

```
var bResult = "23" < "3";
alert(bResult); //outputs "true"
```

这段代码比较字符串“23”和“3”，将输出“true”。两个运算数都是字符串，所以比较的是它们的字符代码（“2”的字符代码是 50，“3”的字符代码是 51）。不过，如果把某个运算数改为数字，那么结果就有趣了：

```
var bResult = "23" < 3;
alert(bResult); //outputs "false"
```

这里，字符串“23”将被转换成数字 23，然后与数字 3 进行比较，结果不出所料。无论何时比较一个数字和一个字符串，ECMAScript 都会把字符串转换成数字，然后按照数字顺序比较它们。对于与前面的示例类似的情况处理方法如此，不过，如果字符串不能转换成数字又该如何呢？考虑下面的例子：

```
var bResult = "a" < 3;
alert(bResult);
```

你能预料到这段代码输出什么吗？字母“a”不能转换成有意义的数字。不过，如果对它调用 `parseInt()` 方法，返回的是 `Nan`。根据规则，任何包含 `Nan` 的关系运算都要返回 `false`，因此这段代码也输出 `false`：

```
var bResult = "a" >= 3;
alert(bResult);
```

通常，如果小于运算的两个值返回 `false`，那么大于等于运算必须返回 `true`，不过如果某个数字是 `Nan`，情况则非如此。

2.9.7 等性运算符

判断两个变量是否相等是程序设计中非常重要的运算。在处理原始值时，这种运算相当简单，但涉及对象，任务就稍有点复杂。ECMAScript 提供了两套运算符处理这个问题，等号和非等号用于处理原始值，全等号和非全等号用于处理对象。

1. 等号和非等号

在 ECMAScript 中，等号由双等号（`==`）表示，当且仅当两个运算数相等时，它返回 `true`。非等号是感叹号加等号（`!=`），当且仅当两个运算数不相等时，它返回 `true`。为确定两个运算数是否相等，这两个运算符都会进行类型转换。

执行类型转换的基本规则如下：

- 如果一个运算数是 Boolean 值，在检查相等性之前，把它转换成数字值。`false` 转换成 0，

`true` 转换成 1。

- 如果一个运算数是字符串，另一个是数字，在检查相等性之前，要尝试把字符串转换成数字。
- 如果一个运算数是对象，另一个是字符串，在检查相等性之前，要尝试把对象转换成字符串（调用 `toString()` 方法）。
- 50 如果一个运算数是对象，另一个是数字，在检查相等性之前，要尝试把对象转换成数字。

在进行比较时，该运算符还遵守下列规则：

- 值 `null` 和 `undefined` 相等。
- 在检查相等性时，不能把 `null` 和 `undefined` 转换成其他值。
- 如果某个运算数是 `Nan`，等号将返回 `false`，非等号将返回 `true`。重要提示：即使两个运算数都是 `Nan`，等号仍然返回 `false`，因为根据规则，`Nan` 不等于 `Nan`。
- 如果两个运算数都是对象，那么比较的是它们的引用值。如果两个运算数指向同一个对象，那么等号返回 `true`，否则两个运算数不等。

下表列出了一些特殊情况及它们的结果：

表达式	值
<code>null == undefined</code>	<code>true</code>
<code>"NaN" == NaN</code>	<code>false</code>
<code>5 == NaN</code>	<code>false</code>
<code>NaN == NaN</code>	<code>false</code>
<code>NaN != NaN</code>	<code>true</code>
<code>false == 0</code>	<code>true</code>
<code>true == 1</code>	<code>true</code>
<code>true == 2</code>	<code>false</code>
<code>undefined == 0</code>	<code>false</code>
<code>null == 0</code>	<code>false</code>
<code>"5" == 5</code>	<code>true</code>

2. 全等号和非全等号

等号和非等号的同类运算符是全等号和非全等号。这两个运算符所做的与等号和非等号相同，只是它们在检查相等性前，不执行类型转换。全等号由三个等号（`==`）表示，只有在无需类型转换运算数就相等的情况下，才返回 `true`。例如：

```
var sNum = "55";
var iNum = 55;
alert(sNum == iNum); //outputs "true"
alert(sNum === iNum); //outputs "false"
```

在这段代码中，第一个警告使用等号比较字符串 "55" 和数字 55，输出 "true"。如前所述，这是因为字符串 "55" 将被转换成数字 55，然后才与另一个数字 55 进行比较。第二个警告使用全等号在没有类型转换的情况下比较字符串和数字，当然，字符串不等于数字，所有输出 "false"。

非全等号由感叹号加两个等号 (`!= =`) 表示，只有在无需类型转换运算数不相等的情况下，才返回 `true`。例如：

```
var sNum = "55";
var iNum = 55;
alert(sNum != iNum); //outputs "false"
alert(sNum !== iNum); //outputs "true"
```

这里，第一个警告使用非等号，把字符串 "55" 转换成数字 55，使得它与第二个运算数 55 相等。因此，计算结果为 `false`，因为两个运算数被看作是相等的。第二个警告使用的是非全等号。该运算是问：“`sNum` 与 `iNum` 不同吗？”这个问题的答案是：“是的 (`true`)”，因为 `sNum` 是字符串，而 `iNum` 是数字，它们当然不同。

2.9.8 条件运算符

条件运算符是 ECMAScript 中功能最多的运算符，它的形式与 Java 中的相同：

```
variable = boolean_expression ? true_value : false_value;
```

该表达式主要是根据 `boolean_expression` 的计算结果有条件的为变量赋值。如果 `boolean_expression` 为 `true`，就把 `true_value` 赋给变量，如果它是 `false`，就把 `false_value` 赋给变量。例如：

```
var iMax = (iNum1 > iNum2) ? iNum1 : iNum2;
```

在这个例子中，`iMax` 将被赋予数字中的最大值。表达式声明如果 `iNum1` 大于 `iNum2`，则把 `iNum1` 赋予 `iMax`。但如果表达式为 `false`(即 `iNum2` 大于或等于 `iNum1`)，则把 `iNum2` 赋予 `iMax`。

2.9.9 赋值运算符

简单的赋值运算由等号 (=) 实现，只是把等号右边的值赋予等号左边的变量。例如：

```
var iNum = 10;
```

复合赋值运算是由乘性运算符、加性运算符或位移运算符加等号 (=) 实现的。这些赋值运算符是下列这些常见情况的缩写形式：

```
var iNum = 10;
iNum = iNum + 10;
```

可以用一个复合赋值运算符改写第二行代码：

```
var iNum = 10;
iNum += 10;
```

每种主要的算术运算以及其他几个运算都有复合赋值运算符：

- 乘法/赋值 (`* =`);
- 除法/赋值 (`/ =`);

- 取模/赋值 (%=);
- 加法/赋值 (+=);
- 减法/赋值 (-=);
- 左移/赋值 (<<=);
- 有符号右移/赋值 (>>=);
- 无符号右移/赋值 (>>>=)。

2.9.10 逗号运算符

用逗号运算符可以在一条语句中执行多个运算。例如：

```
var iNum1=1, iNum2=2, iNum3=3;
```

逗号运算符最常用于变量声明中。

2.10 语句

ECMA-262 描述了 ECMAScript 的几种语句 (statement)。语句主要定义了 ECMAScript 的大部分语法，通常是采用一个或多个关键字，完成给定的任务。语句可以非常简单，例如通知函数退出，也可以非常复杂，如声明一组要反复执行的命令。这一节将介绍所有标准的 ECMAScript 语句。

2.10.1 if 语句

if 语句是 ECMAScript 中最常用的语句之一（事实上在许多语言中都是如此）。if 语句的语法如下：

```
if (condition) statement1 else statement2
```

其中 condition 可以是任何表达式，计算的结果甚至不必是真正的 Boolean 值，ECMAScript 会把它转换成 Boolean 值。如果条件计算结果为 true，执行 statement1，如果条件计算结果为 false，执行 statement2。每个语句都可以是单行代码，也可以是代码块（一组置于括号中的代码行）。例如：

```
if (i > 25)
    alert("Greater than 25.");      //one-line statement
else {
    alert("Less than or equal to 25."); //block statement
}
```

使用代码块被认为是一种编程最佳实践，即使要执行的代码只有一行。这样做可以使每个条件要执行什么一目了然。

还可以串联使用多个 if 语句，如下所示：

```
if (condition1) statement1 else if (condition2) statement2 else statement3
```

例如：

```
if (i > 25) {
    alert("Greater than 25.")
} else if (i < 0) {
    alert("Less than 0.");
} else {
    alert("Between 0 and 25, inclusive.");
}
```

2.10.2 迭代语句

迭代语句又叫循环语句，声明一组要反复执行的命令，直到满足了某些条件为止。循环通常用于迭代数组的值（因此而得名），或者执行重复的算术任务。ECMAScript 为了这种处理提供了四种迭代语句。

1. do-while语句

do-while 语句是后测试循环，即退出条件在执行过循环内部的代码之后计算。这意味着在计算表达式之前，至少会执行循环主体一次。语法如下：

```
do {
    statement
} while (expression);
```

例如：

```
var i = 0;
do {
    i += 2;
} while (i < 10);
```

2. while语句

while 语句是前测试循环。这意味着退出条件是在执行循环内部的代码之前计算的。因此，循环主体可能根本不被执行。语法如下：

```
while(expression) statement
```

例如：

```
var i = 0;
while (i < 10) {
    i += 2;
}
```

3. for语句

for 语句是前测试循环，而且在进入循环之前，能够初始化变量，并定义循环后要执行的代码。语法如下：

```
for (initialization; expression; post-loop-expression) statement
```

例如：

```
for (var i=0; i < iCount; i++) {
    alert(i);
}
```

这段代码定义了初始值为 0 的变量 `i`。只有当条件表达式 (`i < iCount`) 的值为 `true` 时，才进入 `for` 循环，这样循环主体可能不被执行。如果执行了循环主体，那么将执行循环后表达式，并迭代变量 `i`。

4. for-in语句

`for-in` 语句是严格的迭代语句，用于枚举对象的属性。语法如下：

```
for (property in expression) statement
```

例如：

```
for (sProp in window) {
    alert(sProp);
}
```

这里，`for-in` 语句用于显示 BOM `window` 对象的所有属性。前面讨论过的方法 `property-`

55 `IsEnumerable()` 是 ECMAScript 中专门用于说明属性是否可以用 `for-in` 语句访问的方法。

2.10.3 有标签的语句

可以用下列语法给语句加标签，以便以后调用：

```
label: statement
```

例如：

```
start: var iCount = 10;
```

在这个例子中，标签 `start` 可被后来的 `break` 语句或 `continue` 语句引用。

2.10.4 break 语句和 continue 语句

`break` 和 `continue` 语句对循环中的代码执行提供了更严格的控制。`break` 语句可以立即退出循环，阻止再次反复执行任何代码，而 `continue` 语句只是退出当前循环，根据控制表达式还允许继续进行下一次循环。例如：

```
var iNum = 0;

for (var i=1; i < 10; i++) {
    if (i % 5 == 0) {
        break;
    }
    iNum++;
}

alert(iNum); //outputs "4"
```

在上面的代码中，`for`循环将从 1 到 10 迭代变量 `i`。在循环主体中，`if`语句将（使用取模运算符）检查 `i` 的值是否能被 5 整除。如果能被 5 整除，将执行 `break`语句，警告显示“4”，即在退出循环前执行循环的次数。如果用 `continue`语句代替这个例子中的 `break`语句，结果将不同：

```
var iNum = 0;

for (var i=1; i < 10; i++) {
    if (i % 5 == 0) {
        continue;
    }
    iNum++;
}

alert(iNum); //outputs "8"
```

这里，警告将显示“8”，即执行循环的次数。可能执行的循环总数为 9，不过当 `i` 的值为 5 时，将执行 `continue`语句，会使循环跳过表达式 `iNum++`，返回循环开头。

`break`语句和 `continue`语句都可以与有标签的语句联合使用，返回代码中的特定位置。通常，当循环内部还有循环时，会这样做，如下面的例子所示：

```
var iNum = 0;

outermost:
for (var i=0; i < 10; i++) {
    for (var j=0; j < 10; j++) {
        if (i == 5 && j == 5) {
            break outermost;
        }
        iNum++;
    }
}

alert(iNum); //outputs "55"
```

在这个例子中，标签 `outermost` 表示的是第一个 `for`语句。正常情况下，每个 `for`语句执行 10 次代码块，意味着 `iNum++`正常情况下将被执行 100 次，在执行完成时，`iNum` 应该等于 100。这里的 `break`语句有一个参数，即停止循环后要跳转到的语句的标签。这样 `break`语句不止能跳出内部 `for`语句（即使用变量 `j`的语句），还能跳出外部 `for`语句（即使用变量 `i`的语句）。因此，`iNum`最后的值是 55，因为当 `i` 和 `j` 的值都等于 5 时，循环将终止。可以以同样的方式使用 `continue`语句：

```
var iNum = 0;

outermost:
for (var i=0; i < 10; i++) {
    for (var j=0; j < 10; j++) {
        if (i == 5 && j == 5) {
            continue outermost;
        }
        iNum++;
    }
}
```

```

        }
        iNum++;
    }

}

alert(iNum); //outputs "95"

```

在这个例子中，`continue`语句会迫使循环继续，不止是内部循环，外部循环也如此。当 `j` 等于 5 时出现这种情况，意味着内部循环将减少 5 次迭代，致使 `iNum` 的值为 95。

57 可以看到，与 `break` 和 `continue` 联合使用的有标签语句的功能非常强大，不过过度使用它们会给调试代码带来麻烦。要确保使用的标签具有说明性，不要嵌套太多层循环。

2.10.5 with 语句

`with` 语句用于设置代码在特定对象中的作用域。它的语法如下：

```
with (expression) statement;
```

例如：

```

var sMessage = "hello world";
with(sMessage) {
    alert(toUpperCase()); //outputs "HELLO WORLD"
}

```

这段代码中，`with` 语句用于字符串，所以在调用 `toUpperCase()` 方法时，解释程序将检查该方法是否是本地函数。如果不是，它将检查伪对象 `sMessage`，看它是否为该对象的方法。然后警告将输出 "HELLO WORLD"，因为解释程序找到了字符串 "hello world" 的 `toUpperCase()` 方法。

`with` 语句是运行缓慢的代码段，尤其是在已设置了属性值时。大多数情况下，如果可能，最好避免使用它。

2.10.6 switch 语句

`if` 语句的姊妹语句是 `switch` 语句，开发者可以用它为表达式提供一系列情况（`case`）。`switch` 语句的语法如下：

```

switch (expression) {
    case value: statement
    break;
    case value: statement
    break;
    case value: statement
    break;
    ...
    case value: statement
    break;
    default: statement
}

```

每个情况都是表示“如果 *expression* 等于 *value*, 就执行 *statement*”。关键字 *break* 会使代码执行跳出 *switch* 语句。没有关键字 *break*, 代码执行就会继续进入下一个情况。

关键字 *default* 说明了表达式的结果不等于任何一种情况时的操作(事实上, 它是 *else* 从句)。

switch 语句主要是为避免让开发者编写下面这种代码:

```
if (i == 25)
    alert("25");
else if (i == 35)
    alert("35");
else if (i == 45)
    alert("45");
else
    alert("Other");
```

等价的 *switch* 语句如下:

```
switch (i) {
    case 25: alert("25");
                break;
    case 35: alert("35");
                break;
    case 45: alert("45");
                break;
    default: alert("Other");
}
```

ECMAScript 和 Java 中的 *switch* 语句有两点不同。在 ECMAScript 中, *switch* 语句可以用于字符串, 而且能用不是常量的值说明情况:

```
var BLUE = "blue", RED = "red", GREEN = "green";

switch (sColor) {
    case BLUE: alert("Blue");
                break;
    case RED: alert("Red");
                break;
    case GREEN: alert("Green");
                break;
    default: alert("Other");
}
```

这里, *switch* 语句用于字符串 *sColor*, 声明 *case* 使用的是变量 *BLUE*、*RED* 和 *GREEN*, 这在 ECMAScript 中是完全有效的。

2.11 函数

函数是一组可以随时随地运行的语句, 它们是 ECMAScript 的核心。函数是由关键字 *function*、函数名加一组参数以及置于括号中的要执行的代码声明的。函数的基本语法如下:

```
function functionName(arg0, arg1, ..., argN) {
    statements
}
```

例如：

```
function sayHi(sName, sMessage) {
    alert("Hello " + name + "," + sMessage);
}
```

函数可以通过其名字加置于括号中的参数（如果有多个参数，中间用逗号分隔）调用。调用 sayHi() 函数的代码如下：

```
sayHi("Nicholas", "how are you today?");
```

这段代码生成的警告窗口如图 2-8 所示。

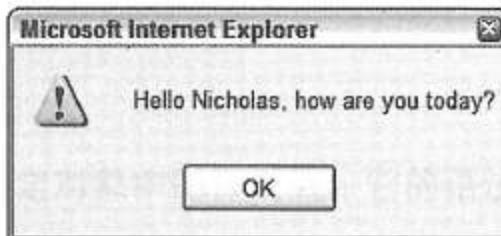


图 2-8

函数 sayHi() 未声明返回值，不过不必专门声明它（如在 Java 中使用 void）。同样的，即使函数确实有返回值，也不必明确地声明它。该函数只需要使用 return 运算符后跟要返回的值即可。

```
function sum(iNum1, iNum2) {
    return iNum1 + iNum2;
}
```

下面的代码把 sum 函数返回的值赋予一个变量：

```
var iResult = sum(1,1);
alert(iResult); //outputs "2"
```

另一个重要的概念是，与在 Java 中一样，函数在执行过 return 语句后停止执行代码。因此，return 语句后的代码都不会被执行。例如，下面函数中的警告信息就不会显示：

```
function sum(iNum1, iNum2) {
    return iNum1 + iNum2;
    alert(iNum1 + iNum2); //never reached
}
```

一个函数中可以有多个 return 语句，如下所示：

```
function diff(iNum1, iNum2) {
    if (iNum1 > iNum2) {
        return iNum1 - iNum2;
    } else {
        return iNum2 - iNum1;
    }
}
```

上面的函数用于返回两个数字的差。要实现这一点，必须用较大的数减去较小的数，因此用 if 语句决定执行哪个 return 语句。

如果函数无返回值，那么可以调用没有参数的 `return` 运算符，随时退出函数。例如：

```
function sayHi(sMessage) {
    if (sMessage == "bye") {
        return;
    }

    alert(sMessage);
}
```

这段代码中，如果 `sMessage` 总等于字符串 "bye"，那么就永远不会出现警告消息"。

如果函数无明确的返回值，或调用了没有参数的 `return` 语句，那么它真正返回的值是 `undefined`。

2.11.1 无重载

ECMAScript 中的函数不能重载。考虑到 ECMAScript 与其他支持重载的高级程序设计语言相似，所以它不支持重载的特点不免让人感到意外。可用相同的名字在同一个作用域中定义两个函数，而不会引发错误，但真正使用的是后一个函数。考虑下面的例子：

```
function doAdd(iNum) {
    alert(iNum + 100);
}

function doAdd(iNum) {
    alert(iNum + 10);
}

doAdd(10);
```

你认为这段代码会显示什么？警告将显示 "20"，因为第二个 `doAdd()` 函数的定义覆盖了第一个定义。虽然这让开发者有些头痛，不过可以使用 `arguments` 对象避开这种限制。

61

2.11.2 `arguments` 对象

在函数代码中，使用特殊对象 `arguments`，开发者无需明确指出参数名，就能访问它们。例如，在函数 `sayHi()` 中，第一个参数是 `message`。用 `arguments[0]` 也可以访问这个值，即第一个参数的值（第一个参数位于位置 0，第二个参数位于位置 1，依此类推）。因此，无需明确命名参数，就可以重写函数：

```
function sayHi() {
    if (arguments[0] == "bye") {
        return;
    }

    alert(arguments[0]);
}
```

还可用 `arguments` 对象检测传递给函数的参数个数，引用属性 `arguments.length` 即可。下面的代码将输出每次调用函数使用的参数个数：

```
function howManyArgs() {
    alert(arguments.length);
}

howManyArgs("string", 45);      //outputs "2"
howManyArgs();                  //outputs "0"
howManyArgs(12);                //outputs "1"
```

这个代码段将依次显示“2”、“0”和“1”。有了 `arguments` 对象，开发者就要检查传递给函数的参数个数。

与其他程序设计语言不同，ECMAScript 不会验证传递给函数的参数个数是否等于函数定义的参数个数。开发者定义的函数都可以接受任意个数的参数（根据 Netscape 的文档，最多能接受 25 个），而不会引发任何错误。任何遗漏的参数都会以 `undefined` 传递给函数，多余的参数将忽略。

用 `arguments` 对象判断传递给函数的参数个数，即可模拟函数重载：

```
function doAdd() {
    if(arguments.length == 1) {
        alert(arguments[0] + 10);
    } else if (arguments.length == 2) {
        alert(arguments[0] + arguments[1]);
    }
}

doAdd(10);          //outputs "20"
doAdd(30, 20);     //outputs "50"
```

62

只有一个参数时，`doAdd()` 函数才会给数字加 10，如果有两个参数，只是把这两个数相加，返回它们的和。所以 `doAdd(10)` 输出的是“20”，而 `doAdd(30, 20)` 输出的是“50”。虽然不如重载那么好，不过已足可避开 ECMAScript 的这种限制。

2.11.3 Function 类

ECMAScript 最令人感兴趣的可能莫过于函数实际上是功能完整的对象。`Function` 类可以表示开发者定义的任何函数。用 `Function` 类直接创建函数的语法如下：

```
var function_name = new Function(argument1, argument2,...,argumentN, function_body);
```

在这种形式中，每个 `argument` 都是一个参数，最后一个参数是函数主体（要执行的代码）。这些参数必须是字符串。记得下面这个函数吗？

```
function sayHi(sName, sMessage) {
    alert("Hello " + sName + "," + sMessage);
}
```

还可以如下定义它：

```
var sayHi = new Function("sName", "sMessage", "alert(\"Hello \" + sName + \", \" + sMessage + \")");
```

诚然，因为字符串的关系，这种形式写起来有些困难，但有助于理解函数只不过是一种引用类型，它们的行为与用 `Function` 类明确创建的函数行为相同。还记得下面的例子吗？

```
function doAdd(iNum) {
    alert(iNum + 100);
}

function doAdd(iNum) {
    alert(iNum + 10);
}

doAdd(10); //outputs "20"
```

如你所知，第二个函数重载了第一个函数，使 `doAdd(10)` 输出了“20”，而不是“110”。如果以下面的形式重写该代码块，这种概念就清楚了：

```
doAdd = new Function("iNum", "alert(iNum + 100)");
doAdd = new Function("iNum", "alert(iNum + 10)");
doAdd(10);
```

观察这段代码，很显然，`doAdd` 的值被改成了指向不同对象的指针。函数名只是指向函数对象的引用值，行为就像其他指针一样。甚至可以使两个变量指向同一个函数：

```
var doAdd = new Function("iNum", "alert(iNum + 10) ");
var alsoDoAdd = doAdd;
doAdd(10); //outputs "20"
alsoDoAdd(10); //outputs "20"
```

这里，变量 `doAdd` 被定义为函数，然后 `alsoDoAdd` 被声明为指向同一个函数的指针。用这两个变量都可以执行该函数的代码，输出相同的结果——“20”。因此，如果函数名只是指向函数的变量，那么可以把函数作为参数传递给另一个函数吗？是的！

```
function callAnotherFunc(fnFunction, vArgument) {
    fnFunction(vArgument);
}

var doAdd = new Function("iNum", "alert(iNum + 10)");

callAnotherFunc(doAdd, 10); //outputs "20"
```

在这个例子中，`callAnotherFunction()` 有两个参数——要调用的函数和传递给该函数的参数。该代码把 `doAdd()` 函数传递给 `callAnotherFnction()` 函数，参数为 10，输出“20”。

尽管可用 `Function` 构造函数创建函数，但最好不要使用它，因为用它定义函数比用传统方式要慢得多。不过，所有函数都应看作是 `Function` 类的实例。

因为函数是引用类型，所以它们也有属性和方法。ECMAScript 定义的属性 `length` 声明了

函数期望的参数个数。例如：

```
function doAdd(iNum) {
    alert(iNum + 10);
}

function sayHi() {
    alert("Hi");
}

alert(doAdd.length); //outputs "1"
alert(sayHi.length); //outputs "0"
```

函数 `doAdd()` 定义了一个参数，因此它的 `length` 是 1；`sayHi()` 没有定义参数，所以 `length` 是 0。记住，无论定义了几个参数，ECMAScript 函数可以接受任意多个参数（最多 25 个）。属性 `length` 只是为查看默认情况下预期的参数个数提供了一种简便的方式。

64 Function 对象也有与所有对象共享的标准 `valueOf()` 方法和 `toString()` 方法。这两个方法返回的都是函数的源代码，在调试时尤其有用。例如：

```
function doAdd(iNum) {
    alert(iNum + 10);
}

alert(doAdd.toString());
```

这段代码输出了 `doAdd()` 函数的文本（如图 2-9 所示）。

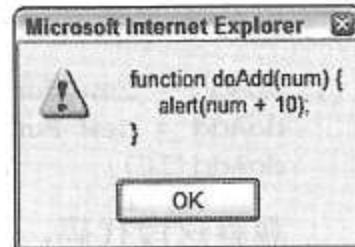


图 2-9

还有两个 `Function` 类的方法与对象的讨论相关，下一章将讨论它们。

2.11.4 闭包

ECMAScript 最容易让人误解的一点是它支持闭包（closure）。所谓闭包，是指词法表示包括不必计算的变量的函数，也就是说，该函数能使用函数外定义的变量。在 ECMAScript 中使用全局变量是一个简单的闭包实例。考虑下面的代码：

```
var sMessage = "Hello World!";

function sayHelloWorld() {
    alert(sMessage);
}

sayHelloWorld();
```

在这段代码中，脚本被载入内存后，并未为函数 `sayHelloWorld()` 计算变量 `sMessage` 的值。该函数捕获 `sMessage` 的值只是为以后使用，也就是说，解释程序知道在调用该函数时要检查 `sMessage` 的值。`sMessage` 将在函数调用 `sayHelloWorld()` 时（最后一行）被赋值，显示消息 "Hello World!"。

在一个函数中定义另一个函数会使闭包变得更复杂，如下所示：

```
var iBaseNum = 10;

function addNumbers(iNum1, iNum2) {
    function doAddition() {
        return iNum1 + iNum2 + iBaseNum;
    }
    return doAddition();
}
```

65

这里，函数 `addNumbers()` 包括函数 `doAddition()`（闭包）。内部函数是个闭包，因为它将获取外部函数的参数 `iNum1` 和 `iNum2` 以及全局变量 `iBaseNum` 的值。`addNumbers()` 的最后一步调用了内部函数，把两个参数和全局变量相加，并返回它们的和。这里要掌握的重要概念是 `doAddition()` 函数根本不接受参数，它使用的值是从执行环境中获取的。

可以看到，闭包是 ECMAScript 中非常强大有用的一部分，可以用于执行复杂的计算。就像使用任何高级函数一样，在使用闭包时要当心，因为它们可能会变得非常复杂。

2.12 小结

这一章介绍了 ECMAScript 的基础：

- 一般语法；
- 用关键字 `var` 定义变量；
- 原始值和引用值；
- 基础的原始类型（`Undefined`、`Null`、`Boolean`、`Number` 和 `String`）；
- 基础的引用类型（`Object`、`Boolean`、`Number` 和 `String`）；
- 运算符和语句；
- 函数。

理解 ECMAScript 是 JavaScript 程序设计的重要部分，所以本章可能是全书最重要的一章。很好的掌握这些核心概念，对理解本书余下的主题至关重要。

下一章的重点是 ECMAScript 面向对象的方面，包括如何创建自己的类以及如何建立继承性。

66

ECMAScript 对象是 JavaScript 比较特殊的特性之一。第 2 章介绍过一切都是对象（包括函数）的概念。本章的重点是如何操作及使用这些对象，以及如何创建自己的对象，以根据需要增加专用的功能。

3.1 面向对象术语

ECMA-262 把对象 (object) 定义为“属性的无序集合，每个属性存放一个原始值、对象或函数”。严格说来，这意味着对象是无特定顺序的值的数组。尽管 ECMAScript 如此定义对象，但它更通用的定义是基于代码的名词（人、地点或事物）表示。

每个对象都由类定义，可以把类看作对象的配方。类不仅要定义对象的接口 (interface)（开发者访问的属性和方法），还要定义对象的内部工作（使属性和方法发挥作用的代码）。编译器和解释程序都根据类的说明构建对象。

程序使用类创建对象时，生成的对象叫做类的实例 (instance)。对类生成的对象的个数的唯一限制来自于运行代码的机器的物理内存。每个实例的行为相同，但实例处理一组独立的数据。由类创建对象实例的过程叫做实例化 (instantiation)。

如第 2 章所述，ECMAScript 并没有正式的类。相反，ECMA-262 把对象定义描述为对象的配方。这是 ECMAScript 的一种逻辑上的折中方案，因为对象定义实际上是对象自身（稍后将对此进行解释）。即使类并不真正存在，本书也把对象定义叫做类，因为大多数开发者对此术语更熟悉，而且从功能上说，两者等价。

对象定义存放在一个函数——构造函数中。构造函数不是一种特殊函数，它只不过是用于创建对象的常规函数。在本章后面的小节将介绍如何创建自己的构造函数。

3.1.1 面向对象语言的要求

一种面向对象语言需要向开发者提供四种基本能力：

- (1) 封装——把相关的信息（无论数据或方法）存储在对象中的能力。
- (2) 聚集——把一个对象存储在另一个对象内的能力。

(3) 继承——由另一个类（或多个类）得来类的属性和方法的能力。

(4) 多态——编写能以多种方法运行的函数或方法的能力。

ECMAScript 支持这些要求，因此可被看作面向对象的。

3.1.2 对象的构成

在 ECMAScript 中，对象由特性（attribute）构成，特性可以是原始值，也可以是引用值。如果特性存放的是函数，它将被看作对象的方法（method），否则该特性被看作属性（property）。

3.2 对象应用

前一章简要谈及对象的使用，现在要详细介绍它们了。对象的创建或销毁都在 JavaScript 执行过程中发生，理解这种范式的含义对理解整个语言至关重要。

3.2.1 声明和实例化

对象是用关键字 new 后跟要实例化的类的名字创建的：

```
var oObject = new Object();
var oStringObject = new String();
```

第一行代码创建了 Object 类的一个实例，并把它存储在变量 oObject 中。第二行代码创建了 String 类的一个实例，把它存储在变量 oStringObject 中。如果构造函数无参数，括号则不是必需的，因此可以采用下面的形式重写上面的两行代码：

```
var oObject = new Object;
var oStringObject = new String;
```

68

3.2.2 对象引用

在第 2 章中，介绍了引用类型的概念。在 ECMAScript 中，不能访问对象的物理表示，只能访问对象的引用。每次创建对象，存储在变量中的都是该对象的引用，而不是对象本身。

3.2.3 对象废除

ECMAScript 有无用存储单元收集程序，意味着不必专门销毁对象来释放内存。当再没有对对象的引用时，称该对象被废除（dereference）了。运行无用存储单元收集程序时，所有废除的对象都被销毁。每当函数执行完它的代码，无用存储单元收集程序都会运行，释放所有的局部变量，还有在一些其他不可预知的情况下，无用存储单元收集程序也会运行。

把对象的所有引用都设置为 null，可以强制性的废除对象。例如：

```
var oObject = new Object;
//do something with the object here
oObject = null;
```

当变量 `oObject` 设置为 `null` 后，对第一个创建的对象的引用就不存在了。这意味着下次运行无用存储单元收集程序时，该对象将被销毁。

每用完一个对象后，就将其废除，来释放内存，这是个好习惯。这样还确保不再使用已经不能访问的对象，从而防止程序设计错误的出现。此外，旧的浏览器（如 IE/Mac）没有全面的无用存储单元回收程序，所以在卸载页面时，对象可能不能被正确销毁。废除对象和它的所有特性是确保内存使用正确的最好方法。

废除对象的所有引用时要当心。如果一个对象有两个或更多引用，则要正确废除该对象，必须将其所有引用都设置为 `null`。

3.2.4 早绑定和晚绑定

所谓绑定（binding），即把对象的接口与对象实例结合在一起的方法。

· 早绑定（early binding）是指在实例化对象之前定义它的特性和方法，这样编译器或解释程序就能提前转换机器代码。在 Java 和 Visual Basic 这样的语言中，有了早绑定，就可以在开发环境中使用 IntelliSense（即给开发者提供其对象中特性和方法列表的功能）。ECMAScript 不是强类型语言，所以不支持早绑定。

另一方面，晚绑定（late binding）指的是编译器或解释程序在运行前，不知道对象的类型。使用晚绑定，无需检查对象的类型，只需要检查对象是否支持特性和方法即可。ECMAScript 中的所有变量都采用晚绑定方法，这样就允许执行大量的对象操作，而无任何惩罚。

69

3.3 对象的类型

在 ECMAScript 中，所有对象并非同等创建的。一般说来，可以创建并使用的对象有三种。

3.3.1 本地对象

ECMA-262 把本地对象（native object）定义为“独立于宿主环境的 ECMAScript 实现提供的对象”。简单说来，本地对象就是 ECMA-262 定义的类（引用类型）。它们包括：

<code>Object</code>	<code>Function</code>	<code>Array</code>	<code>String</code>
<code>Boolean</code>	<code>Number</code>	<code>Date</code>	<code>RegExp</code>
<code>Error</code>	<code>EvalError</code>	<code>RangeError</code>	<code>ReferenceError</code>
<code>SyntaxError</code>	<code>TypeError</code>	<code>URIError</code>	

你已经从上一章了解了一些本地对象（`Object`、`Function`、`String`、`Boolean` 和 `Number`），本书后面的章节中还会讨论一些本地对象。现在要讨论的两种重要的本地对象是 `Array` 和 `Date`。

1. `Array` 类

与 Java 不同的是，在 ECMAScript 中有真正的 `Array` 类。可以如下创建 `Array` 对象：

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

如果预先知道数组中项的个数，可以用参数传递数组的大小：

```
var aValues = new Array(20);
```

使用这两个方法，一点要使用括号，与它们在 Java 中的用法相似：

```
var aColors = new Array();
aColors[0] = "red";
aColors[1] = "green";
aColors[2] = "blue";
```

这里创建了一个数组，并定义了三个数组项，即"red"、"green"和"blue"。每增加一个数组项，数组的大小就动态地增长。

此外，如果知道数组应该存放的值，还可用参数声明这些值，创建大小与参数个数相等的Array对象。例如，下面的代码将创建一个有三个字符串的数组：

```
var aColors = new Array("red", "green", "blue");
```

70

与字符串类似，数组中的第一个项位于位置 0，第二个项位于位置 1，依此类推。可通过使用方括号中放置要读取的项的位置来访问特定的项。例如，要用刚才定义的数组输出字符串"green"，可以采用下面的代码：

```
alert(aColors[1]); //outputs "green"
```

可用属性 length 得到数组的大小。与字符串的 length 属性一样，数组的 length 属性也是最后一个项的位置加 1，意味着具有三个项的数组中的项的位置是从 0 到 2。

```
var aColors = new Array("red", "green", "blue");
alert(aColors.length); //outputs "3"
```

前面提过，数组可以根据需要增大或减小。因此，如果要为前面定义的数组增加一项，只需把要存放的值放入下一个未使用的位置即可：

```
var aColors = new Array("red", "green", "blue");
alert(aColors.length); //outputs "3"
aColors[3] = "purple";
alert(aColors.length); //outputs "4"
```

在这段代码中，下一个未使用的位置是 3，所以值"purple"将被赋予它。增加一项使数组的大小从 3 变成了 4。但如果把值放在这个数组的位置 25 处会怎样呢？ECMAScript 将把从 3 开始到 24 的所有位置都填上值 null，然后在位置 25 处放上正确的值，并把数组大小增大为 26：

```
var aColors = new Array("red", "green", "blue");
alert(aColors.length); //outputs "3"
aColors[25] = "purple";
aColors.length; //outputs "26"
```

数组最多可以存放 4294967295 项，这应该可满足大多数程序设计的需要。如果要添加更多的项，则会发生异常。

还可以用字面量表示定义 Array 对象，即使用方括号 ([和])，用逗号分隔值。例如，可以

用下面的形式重写前面的例子：

```
var aColors = ["red", "green", "blue"];
alert(aColors.length); //outputs "3"
aColors[25] = "purple";
alert(aColors.length); //outputs "26"
```

注意，在这个例子中，未明确使用 `Array` 类。方括号暗示把其中的值存放在 `Array` 对象中。

71 用这种方式声明的数组与用传统方式声明的数组相同。

`Array` 对象覆盖了 `toString()` 方法和 `valueOf()` 方法，返回特殊的字符串。该字符串是通过对每项调用 `toString()` 方法，然后用逗号把它们连接在一起构成的。例如，对具有项 "red"、"green" 和 "blue" 的数组调用 `toString()` 方法或 `valueOf()` 方法，返回的是字符串 "red,green,blue"。

```
var aColors = ["red", "green", "blue"];
alert(aColors.toString()); //outputs "red,green,blue"
alert(aColors.valueOf()); //outputs "red,green,blue"
```

类似的，`toLocaleString()` 方法返回的也是由数组项构成的字符串。唯一的区别是得到的值是通过调用每个数组项的 `toLocaleString()` 方法得到的。许多情况下，该方法返回的值都与 `toString()` 方法返回的值相同，也是用逗号连接字符串。

```
var aColors = ["red", "green", "blue"];
alert(aColors.toLocaleString()); //outputs "red,green,blue"
```

由于开发者也可能希望在数组之外创建这样的值，所以 ECMAScript 提供了方法 `join()`，它唯一的用途就是连接字符串值。`join()` 方法只有一个参数，即数组项之间使用的字符串。考虑下面的例子：

```
var aColors = ["red", "green", "blue"];
alert(aColors.join(","));
//outputs "red,green,blue"
alert(aColors.join("-spring-"));
//outputs "red-spring-green-spring-blue"
alert(aColors.join("]["));
//outputs "red][green][blue"
```

这里用方法 `join()` 创建了三种不同的数组表示。第一个 `join()` 方法使用逗号，本质上与调用 `toString()` 方法或 `valueOf()` 方法等价。第二个和第三个 `join()` 方法使用不同的字符串，在数组项之间创建了奇怪的分隔符（可能不怎么有用）。理解的重点在于任何字符串都可以用作分隔符。

此刻也许你想知道，既然 `Array` 具有把自己转换成字符串的方法，那么 `String` 是否有把自己转换成数组的方法呢？答案是肯定的。`String` 类的方法 `split()` 正于此。`split()` 方法只有一个参数。可能有读者已经猜到，该参数就是被看作数组项之间的分隔符的字符串。因此，如果有一个由逗号分隔的字符串，就可以用下面的代码把它转换成 `Array` 对象：

```
var sColors = "red,green,blue";
var aColors = sColors.split(",");
```

如果把空字符串声明为分隔符，那么 `split()` 方法返回的数组中的每个项是字符串的字符，

例如：

```
var sColors = "green";
var aColors = sColors.split("");
alert(aColors.toString()); //outputs "g,r,e,e,n"
```

这里，字符串"green"将被转换成字符串数组"g"、"r"、"e"、"e"和"n"。如果需要逐个字符的解析字符串，这种功能非常有用。

Array 对象具有两个 String 类具有的方法，即 concat() 和 slice() 方法。concat() 方法处理数组的方式几乎与它处理字符串的方式完全一样。参数将被附加在数组末尾，返回的函数值是新的 Array 对象（包括原始数组中的项和新的项）。例如：

```
var aColors = ["red", "green", "blue"];
var aColors2 = arr.concat("yellow", "purple");
alert(aColors2.toString()); //outputs "red,green,blue,yellow,purple"
alert(aColors.toString()); //outputs "red,green,blue"
```

在这个例子中，用 concat() 方法把字符串"yellow"和"purple"加到数组中。数组 aColors2 包括 5 个值，而原始数组 aColors 仍只有 3 个值。可通过对两个数组分别调用 toString() 方法证明这一点。

slice() 方法也与 String 类中的 slice() 方法非常相似，返回的是具有特定项的新数组。类似于 String 类的方法，Array 类的 slice() 方法也接受一个或两个参数，即要提取的项的起始位置和结束位置。如果只有一个参数，该方法将返回从该位置开始到数组结尾的所有项；如果有两个参数，该方法将返回第一个位置和第二个位置间的所有项，不包括第二个位置处的项。例如：

```
var aColors = ["red", "green", "blue", "yellow", "purple"];
var aColors2 = arr.slice(1);
var aColors3 = arr.slice(1, 4);
alert(aColors2.toString()); //outputs "green,blue,yellow,purple"
alert(aColors3.toString()); //outputs "green,blue,yellow"
```

这里，aColors2 具有 arr 中从位置 1 开始的所有项。因为字符串"green"位于位置 1，所以它是新数组中的第一个元素。对于 aColors3，调用 slice() 方法时有两个参数，即 1 和 4。字符串"green"位于位置 1，字符串"purple"位于位置 4，所以 aColors3 存放的是"green"、"blue"和"yellow"，因为 slice() 方法只返回后一个位置之前的项。

ECMAScript 的 Array 类的一个有趣之处是它提供的方法使数组的行为与其他数据类型的行为相似。例如，Array 对象的动作就像一个栈。所谓栈，是一种限制了插入和删除数据项操作的数据结构。栈又叫做后进先出（LIFO）结构，意思是最近添加的项是最先删除的项。栈中的插入和删除操作都只发生在一个位置，即栈顶部。

通俗点说，把栈想象成一堆碟子更容易理解。如果想在一堆碟子上再加一个碟子，需要把它放在这堆碟子的顶部。把一个项加到栈中，叫做把这个项推入该栈，它将被加在栈的顶部（如图 3-1 所示）。

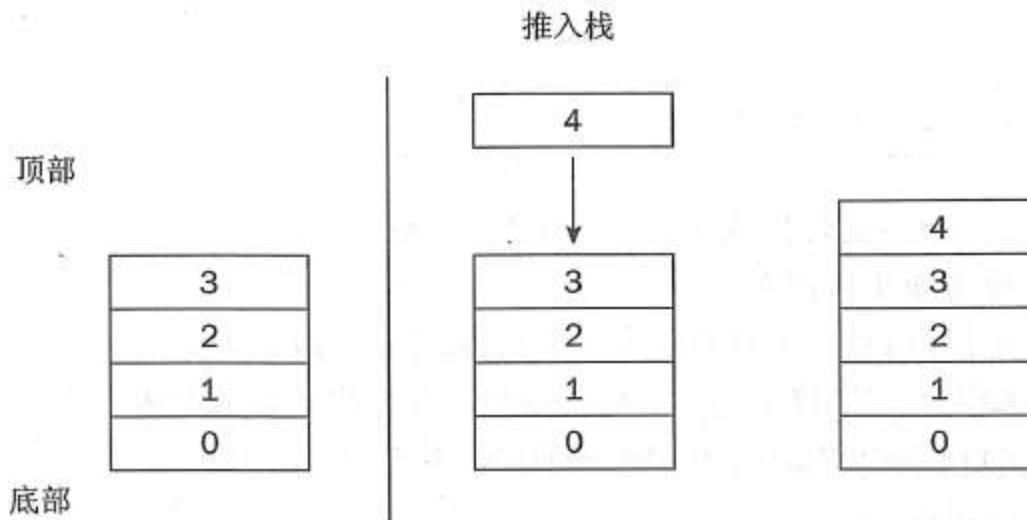


图 3-1

当要撤掉晚餐使用的碟子时，你会做什么？当然是撤掉一堆碟子顶部的碟子，把它放在桌子上。同样的，栈这种数据结构的处理方式也是如此，只删除最上面的项。从栈中删除一项，叫做从栈中弹出该项（如图 3-2 所示）。

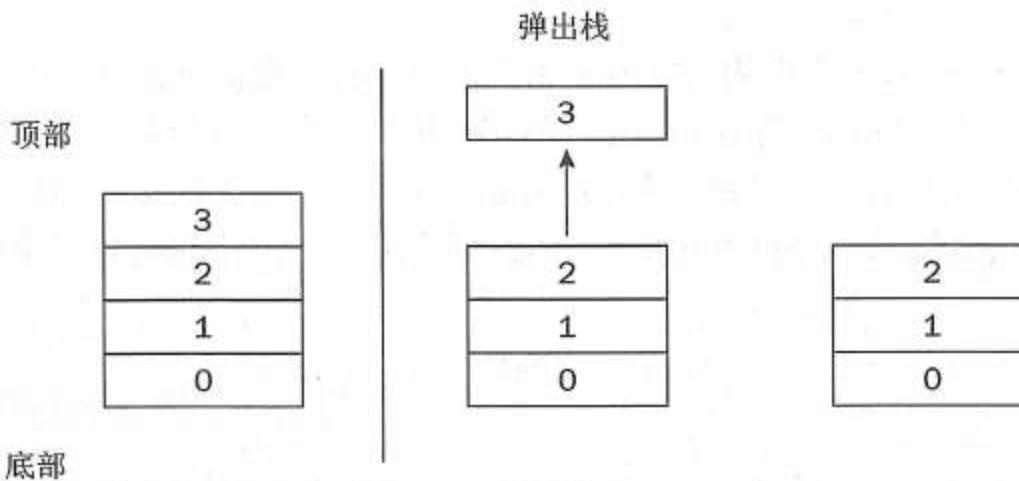


图 3-2

为更便利地使用这种功能，`Array` 对象提供了两个方法 `push()` 和 `pop()`。如你所料，`push()` 方法用于在 `Array` 结尾添加一个或多个项，`pop()` 方法用于删除最后一个数组项 (`length-1`)，返回它作为函数值。考虑下面的示例：

```
var stack = new Array;
stack.push("red");
stack.push("green");
stack.push("yellow");
alert(stack.toString());      //outputs "red,green,yellow"
var vItem = stack.pop();
alert(vItem);                //outputs "yellow"
alert(stack.toString());      //outputs "red,green"
```

在上面的代码中，创建了空的 `Array` 对象，然后几次调用 `push()` 方法（注意，虽然示例中调用 `push()` 方法时只用了一个参数，事实上可以根据需要给它传递多个参数）。在填充数组后，输出字符串值 ("red,green,yellow")，以证明所有项都被加入。然后，调用 `pop()` 方法，它只

返回最后一项 "yellow"，这个项被存入变量 vItem。然后数组只剩下两个字符串，即 "red" 和 "green"。

`push()` 方法实际上与前面例子中的手动添加数组项一样。可以如下重写该示例：

```
var stack = new Array;
stack[0] = "red";
stack[1] = "green";
stack[2] = "yellow";
alert(stack.toString());           //outputs "red,green,yellow"
var vItem = stack.pop();
alert(vItem);                   //outputs "yellow"
alert(stack.toString());           //outputs "red,green"
```

`Array` 还提供了操作第一项的方法。方法 `shift()` 将删除数组中的第一个项，将其作为函数值返回。另一个方法是 `unshift()` 方法，它把一个项放在数组的第一个位置，然后把余下的项向下移动一个位置。例如：

```
var aColors = ["red", "green", "yellow"];
var vItem = aColors.shift();
alert(aColors.toString());         //outputs "green,yellow"
alert(vItem);                     //outputs "red"
aColors.unshift("black");
alert(aColors.toString());         //outputs "black,green,yellow"
```

在这个例子中，从数组中删除字符串 "red" (`shift()`)，只留下 "green" 和 "yellow"。用 `unshift()` 方法，把字符串 "black" 放在数组的开头，从而代替 "red" 成为第一个位置的新值。

通过调用 `shift()` 和 `push()` 方法，可以使 `Array` 对象具有队列一样的行为。所谓队列 (queue) 是对元素的插入和删除操作具有限制的数据结构的一员。队列又叫做后进后出 (LIFO) 结构，意思是最近加入的元素最后删除。元素的插入操作只发生在队列的尾部而删除操作则发生在队列头部。

把队列想象成电影院内所排的队伍。当新人到达，要买票时，他们需要走到队伍的末尾（如图 3-3 所示）。这种操作传统上叫做 `put` 或入队 (enqueue)。

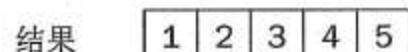
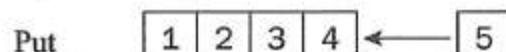
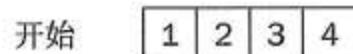


图 3-3

他们将等待，直到移动到队伍的开头（在此买票）为止。完成购买后，人们将离开队伍的开头，进入电影院（如图 3-4 所示）。这种操作传统上叫做 `get` 或出队 (dequeue)。

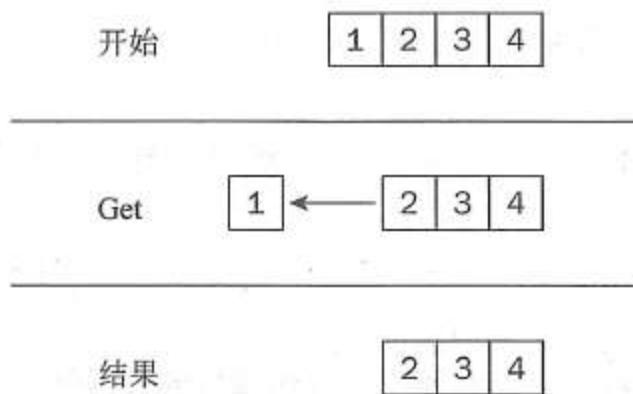


图 3-4

虽然方法名不同，但功能相同。用 `push()` 方法把数据项加入队列（即在数组结尾添加数据项），然后用 `shift()` 方法把它从队列中删除：

```
var queue = ["red", "green", "yellow"];
queue.push("black");
alert(queue.toString());           //outputs "red,green,yellow,black"
var sNextColor = queue.shift();
alert(sNextColor);               //outputs "red"
alert(queue.toString());         //outputs "green,yellow,black"
```

在这个例子中，用 `push()` 方法把字符串 "black" 加到队列的尾部。为得到下一种颜色，调用 `shift()` 方法，获取 "red"，队列中只剩下 "green"、"yellow" 和 "black"。

有两种与数组项的顺序有关的方法，即 `reverse()` 和 `sort()` 方法。如你所料，`reverse()` 方法颠倒数组项的顺序。因此，如果想颠倒 "red"、"green" 和 "blue" 的顺序，可以采用下列代码：

```
var aColors = ["red", "green", "blue"];
aColors.reverse();
alert(aColors.toString());      //outputs "blue,green,red"
```

另一方面，`sort()` 方法将根据数组项的值按升序为它们排序。要进行这种排序，首先调用 `toString()` 方法，将所有值转换成字符串，然后根据字符代码比较数组项（在用小于号对字符串进行运算一节介绍过）。例如：

```
var aColors = ["red", "green", "blue", "yellow"];
aColors.sort();
alert(aColors.toString());      //outputs "blue,green,red,yellow"
```

这段代码使用字符代码对字符串 "red"、"green"、"blue" 和 "yellow" 按照字母顺序进行排序。因为所有值都是字符串，所以这种排序是符合逻辑的。不过，如果值是数字，结果就显得奇怪了：

```
var aColors = [3, 32, 2, 5]
aColors.sort();
alert(aColors.toString());      //outputs "2,3,32,5"
```

在对数字 3、32、2 和 5 进行排序时，方法 `sort()` 把数组项的顺序调整为 2、3、32 和 5。如前所述，出现这种情况的是因为，数字是被转换成字符串，然后再按照字符代码进行比较的。

这个问题可以克服。我将在第 12 章中对其进一步讨论。

迄今为止，最复杂的方法是 `splice()`。这种方法的作用真的非常简单：只是把数据项插入数组的中部。不过，该方法用于插入这些项的方式的变体却大有用处：

- **删除**——只需要声明两个参数，就可以从数组中删除任意多个项，这两个参数是要删除的第一个项的位置和要删除的项的个数。例如 `arr.splice(0, 2)` 将删除数组 `arr` 中的前两项。
- **替换而不删除**——声明三个参数就可以把数据项插入指定的位置，这三个参数是起始位置、0（要删除的数组项的个数）和要插入的项。此外，还可以用第四个、第五个或更多个参数指定其他要删除的项。例如，`arr.splice(2, 0, "red", "green")` 将在位置 2 处插入 "red" 和 "green"。
- **替换并删除**——声明三个参数就可以把数据项插入指定的位置，这三个参数是起始位置、要删除的数组项的个数以及要插入的项。此外，还可以指定要插入的更多的项。要插入的项的个数不必等于删除的项的个数。例如，`arr.splice(2, 1, "red", "green")` 将删除数组 `arr` 中位置 2 处的项，然后在位置 2 处插入 "red" 和 "green"。

可以看到，`Array` 类是用途非常多、十分有用的对象。第 12 章将探讨如何以更实用的方式使用数组，不过到目前为止，你只需要了解这么多。

2. Date类

ECMAScript 中的 `Date` 类基于 Java 中的 `java.util.Date` 类的早期版本。与 Java 一样，ECMAScript 把日期存储为距离 UTC 时间 1970 年 1 月 1 日凌晨 12 点的毫秒数。UTC 是 Universal Time Code，即通用时间代码（也叫做 Greenwich Mean Time，格林尼治标准时间）的缩写，是所有时区的基准标准时间。以毫秒数存储时间可以确保 Java 和 ECMAScript 免受恐怖的“千年虫”问题的侵害，该问题在 20 世纪 90 年代后期影响了许多较老的大型计算机。计算机可以精确地表示出 1970 年 1 月 1 日之前及之后 285 616 年的日期，这意味着除非你可以活过 200000 年，否则日期存储不成问题。

用下面代码可以创建新的 `Date` 对象：

```
var d = new Date();
```

这行代码用当前的日期和时间创建新的 `Date` 对象。创建新 `Date` 对象时，可以以两种方式设置日期和时间的值。第一种方法是，只声明距离 1970 年 1 月 1 日凌晨 12 点的毫秒数：

```
var d = new Date(0);
```

还有 `parse()` 和 `UTC()` 两种方法（在 Java 中是静态方法）可以与创建 `Date` 对象的方法一起使用。`parse()` 方法接受字符串为参数，把该字符串转换成日期值（即毫秒表示）。ECMA-262 未定义 `parse()` 方法接受的日期格式，所以这由 ECMAScript 的实现特定，通常是地点特定的。例如，在美国，大多数实现支持下面的日期格式：

- mm/dd/yyyy（例如 6/13/2004）
- mmmm dd,yyyy（例如 January 12,2004）

例如，如果为 2004 年 5 月 25 日创建 Date 对象，可以使用 `parse()` 方法获得它的毫秒表示，然后将该值传递给 `Date` 构造函数：

```
var d = new Date(Date.parse("May 25, 2004"));
```

如果传递给 `parse()` 方法的字符串不能转换成日期，该函数返回 `Nan`。

`UTC()` 方法返回的也是日期的毫秒表示，但参数不同，是日期中的年、月、日、小时、分、秒和毫秒。使用该方法时，必须声明年和月，其他参数可选。设置月份时要格外注意，因为它的值是从 0 到 11，0 代表一月，11 代表十二月，因此，要设置 2004 年 2 月 5 号，可以使用下列代码：

```
var d = new Date(Date.UTC(2004, 1, 5));
```

这里，1 表示二月，即第二个月。这与用户设置日期输入的值不同。可以想到，其他可能存在这种例外情况的参数是小时，采用 24 时制，而不是 12 时制。因此，要设置 2004 年 2 月 5 号下午 1:05 分，可以使用下面的代码：

```
var d = new Date(Date.UTC(2004, 1, 5, 13, 5));
```

创建日期的第二种方法是直接声明 `UTC()` 方法接受的参数：

```
var d = new Date(2004, 1, 5);
```

声明参数的顺序相同，（除了年和月外）它们不必都出现。

`Date` 类是少有的几个覆盖了 `toString()` 方法和 `valueOf()` 方法的类之一。`valueOf()` 方法返回日期的毫秒表示，`toString()` 方法返回由实现特定的字符串，采用人们可读懂的格式。因此，不能依赖 `toString()` 方法执行任何一致的操作。例如，在美国，IE 把 2003 年 2 月 2 号显示为“Sun Feb 2 00:00:00 EST 2003”，而 Mozilla 则把它显示为“Sun Feb 2 2003 00:00:00 GMT-0400 (Eastern Daylight Time)”。
78

还有其他几个用于创建特定日期的字符串表示的方法：

- `toDateString()`——以实现的特定的格式显示 `Date` 的日期部分（即只有月、日和年）；
- `toTimeString()`——以实现的特定的格式显示 `Date` 的时间部分（即小时、分、秒和时区）；
- `toLocaleString()`——以地点特定的格式显示 `Date` 的日期和时间；
- `toLocaleDateString()`——以地点特定的格式显示 `Date` 的日期部分；
- `toLocaleTimeString()`——以地点特定的格式显示 `Date` 的时间部分；
- `toUTCString()`——以实现特定的格式显示 `Date` 的 UTC 时间。

以上每种方法根据不同的实现和地点，输出不同的值，因此，使用它们时，必须多加练习。

可能你还没有领会到，`Date` 类对 UTC 日期和时间有很强的依赖性。`Date` 类用方法 `getTimezoneOffset()` 来说明某个时区与 UTC 时间的关系，该方法返回当前时区比 UTC 提前或落后的分钟数。例如，对于 U.S. Eastern Daylight Saving Time（美国东部夏令时），`getTimezoneOffset()` 返回 300，即比 UTC 时间落后 5 个小时（300 分钟）。

还可用 `getTimezoneOffset()` 方法判断时区使用的是否是夏令时。实现这一点需要创建任

意年份的1月1日的日期，然后创建该年份的7月1日的日期，比较时区偏移量。如果分钟数不等，说明该时区使用的是夏令时，如果相等，则该时区使用的不是夏令时。

```
var d1 = new Date(2004, 0, 1);
var d2 = new Date(2004, 6, 1);
var bSupportsDaylightSavingTime = d1.getTimezoneOffset() != d2.getTimezoneOffset();
```

Date类其余的方法（列在下表中）均用于设置或获取日期值的某部分。

方 法	说 明
getTime()	返回日期的毫秒表示
setTime(milliseconds)	设置日期的毫秒表示
getFullYear()	返回用四位数字表示的日期的年份（如2004而不只是04）
getUTCFullYear()	返回用四位数字表示的UTC日期的年份
setFullYear(year)	设置日期的年份，参数必须是四位数字的年份值
setUTCFullYear(year)	设置UTC日期的年份，参数必须是四位数字的年份值
getMonth()	返回日期的月份值，由数字0（1月）到11（12月）表示
getUTCMonth()	返回UTC日期的月份值，由数字0（1月）到11（12月）表示
setMonth(month)	设置日期的月份为大于等于0的数字。对于大于11的数字，开始累计年数
setUTCMonth(month)	设置UTC日期的月份为大于等于0的数字。对于大于11的数字，开始累计年数
getDate()	返回该日期该月中的某天
getUTCDate()	返回该UTC日期该月中的某天
setDate(date)	设置该日期该月中的某天
setUTCDate(date)	设置该UTC日期该月中的某天
getDay()	返回该日期为星期几
getUTCDay()	返回该UTC日期为星期几
setDay(day)	设置该日期为星期几
setUTCDay(day)	设置该UTC日期为星期几
getHours()	返回日期中的小时值
getUTCHours()	返回UTC日期中的小时值
setHours(hours)	设置日期中的小时值
setUTCHours(hours)	设置UTC日期中的小时值
getMinutes()	返回日期中的分钟值
getUTCMinutes()	返回UTC日期中的分钟值
setMinutes(minutes)	设置日期中的分钟值
setUTCMinutes(minutes)	设置UTC日期中的分钟值
getSeconds()	返回日期中的秒值
getUTCSeconds()	返回UTC日期中的秒值
setSeconds(seconds)	设置日期中的秒值
setUTCSeconds(seconds)	设置UTC日期中的秒值

(续)

方 法	说 明
getMilliseconds()	返回日期中的毫秒值。注意，这不是自 1970 年 1 月 1 日以后的毫秒值，而是当前时间中的毫秒值，例如 4:55:34.20，其中 20 即为时间的毫秒值
getUTCMilliseconds()	返回 UTC 日期中的毫秒值
setMilliseconds(milliseconds)	设置日期中的毫秒值
setUTCMilliseconds(milliseconds)	设置 UTC 日期中的毫秒值

3.3.2 内置对象

ECMA-262 把内置对象（built-in object）定义为“由 ECMAScript 实现提供的、独立于宿主环境的所有对象，在 ECMAScript 程序开始执行时出现”。这意味着开发者不必明确实例化内置对象，它已被实例化了。ECMA-262 只定义了两个内置对象，即 `Global` 和 `Math`（它们也是本地对象，根据定义，每个内置对象都是本地对象）。

1. Global 对象

`Global` 对象是 ECMAScript 中最特别的对象，因为实际上它根本不存在。如果尝试编写下面的代码，将得到错误：

```
var pointer = Global;
```

错误消息显示 `Global` 不是对象，但刚才不是说 `Global` 是对象吗？没错。这里需要理解的主要概念是，在 ECMAScript 中，不存在独立的函数，所有函数都必须是某个对象的方法。本书前面介绍的函数，如 `isNaN()`、`isFinite()`、`parseInt()` 和 `parseFloat()` 等，看起来都像独立的函数。实际上，它们都是 `Global` 对象的方法。而且 `Global` 对象的方法不止这些。

`encodeURI()` 和 `encodeURIComponent()` 方法用于编码传递给浏览器的 URI（统一资源标识符）。有效的 URI 不能包含某些字符，如空格。这两个方法用于编码 URI，这样用专门的 UTF-8 编码替换所有的非有效字符，就可以使浏览器仍能够接受并理解它们。

`encodeURI()` 方法用于处理完整的 URI（例如，`http://www.wrox.com/illegal value.htm`），而 `encodeURIComponent()` 用于处理 URI 的一个片断（如前面的 URI 中的 `illegal value.htm`）。这两个方法的主要区别是 `encodeURI()` 方法不对 URI 中的特殊字符进行编码，如冒号、前斜杠、问号和英镑符号，而 `encodeURIComponent()` 则对它发现的所有非标准字符进行编码。例如：

```
var sUri = "http://www.wrox.com/illegal value.htm#start";
alert(encodeURI(sUri));
alert(encodeURIComponent(sUri));
```

这段代码输出两个值：

```
http://www.wrox.com/illegal%20value.htm#start
http%3A%2Fwww.wrox.com%2Fillegal%20value.htm%23start
```

可以看到，除空格外，第一个 URI 无任何改变，空格被替换为`%20`。第二个 URI 中的所有非

字母数字字符都被替换成它们对应的编码，基本上使这个 URI 变得无用。这就是 encodeURI() 可以处理完整 URI，而 encodeURIComponent() 只能处理附加在已有 URI 末尾的字符串的原因。

自然，还有两个方法用于解码编码过的 URI，即 decodeURI() 和 decodeURIComponent()。如你所料，这两个方法所做的恰与其对应的方法相反。decodeURI() 方法只对用 encodeURI() 方法替换的字符解码。例如，%20 将被替换为空格，而%23 不会被替换，因为它表示的是英镑符号 (#)，encodeURI() 并不替换这个符号。同样的，decodeURIComponent() 会解码所有 encodeURIComponent() 编码过的字符，意味着它将对所有的特殊值解码。例如：

```
var sUri = "http%3A%2F%2Fwww.wrox.com%2Fillegal%20value.htm%23start";
alert(decodeURI(sUri));
alert(decodeURIComponent(sUri));
```

这段代码输出两个值：

```
http%3A%2F%2Fwww.wrox.com%2Fillegal value.htm%23start
http://www.wrox.com/illegal value.htm#start
```

在这个例子中，变量 uri 存放的是用 encodeURIComponent() 编码的字符串。生成的值说明了应用两个解码方法时会发生的事情。第一个值由 decodeURI() 输出，把%20 替换成空格。第二个值由 decodeURIComponent() 输出，替换所有的特殊。

这些 URI 方法 encodeURI()、encodeURIComponent()、decodeURI() 和 decodeURIComponent() 代替了 BOM 的 escape() 和 unescape() 方法。URI 方法更可取，因为它们会对所有 Unicode 符号编码，而 BOM 方法只能对 ASCII 符号正确编码。尽量避免使用 escape() 和 unescape() 方法。

最后一个方法可能是整个 ECMAScript 语言中最强大的方法，即 eval() 方法。该方法就像整个 ECMAScript 的解释程序，接受一个参数，即要执行的 ECMAScript (或 JavaScript) 字符串。例如：

```
eval("alert('hi')");
```

这行代码的功能等价于下面的代码：

```
alert("hi");
```

82

当解释程序发现 eval() 调用时，它将把参数解释为真正的 ECMAScript 语句，然后把它插入该函数所在的位置。这意味着 eval() 调用内部引用的变量可在参数以外定义：

```
var msg = "hello world";
eval("alert(msg);")
```

这里，变量 msg 是在 eval() 调用的环境外定义的，而警告仍然显示的是文本"hello worla"，因为第二行代码将被替换为一行真正的代码。同样，可以在 eval() 调用内部定义函数或变量，然后在函数外的代码中引用：

```
eval("function sayHi() { alert('hi'); }");
sayHi();
```

这里，函数 `sayHi()` 是在 `eval()` 调用内部定义的。因为该调用将被替换为真正的函数，所以仍可在接下来的一行中调用 `sayHi()`。

这种功能非常强大，不过也非常危险。使用 `eval()` 时要极度小心，尤其在给它传递用户输入的数据时。恶意的用户可能会插入对站点或应用程序的安全性有危害的值（叫做代码注入）。

`Global` 对象不只有方法，它还有属性。还记得那些特殊值 `undefined`、`NaN` 和 `Infinity` 吗？它们都是 `Global` 对象的属性。此外，所有本地对象的构造函数也都是 `Global` 对象的属性。下表较详细地说明了 `Global` 对象的所有属性：

属 性	说 明
<code>undefined</code>	<code>Undefined</code> 类型的字面量
<code>NaN</code>	非数的专用数值
<code>Infinity</code>	无穷大值的专用数值
<code>Object</code>	<code>Object</code> 的构造函数
<code>Array</code>	<code>Array</code> 的构造函数
<code>Function</code>	<code>Function</code> 的构造函数
<code>Boolean</code>	<code>Boolean</code> 的构造函数
<code>String</code>	<code>String</code> 的构造函数
<code>Number</code>	<code>Number</code> 的构造函数
<code>Date</code>	<code>Date</code> 的构造函数
<code>RegExp</code>	<code>RegExp</code> 的构造函数
<code>Error</code>	<code>Error</code> 的构造函数
<code>EvalError</code>	<code>EvalError</code> 的构造函数
<code>RangeError</code>	<code>RangeError</code> 的构造函数
<code>ReferenceError</code>	<code>ReferenceError</code> 的构造函数
<code>SyntaxError</code>	<code>SyntaxError</code> 的构造函数
<code>TypeError</code>	<code>TypeError</code> 的构造函数
<code>URIError</code>	<code>URIError</code> 的构造函数

2. Math对象

`Math` 对象是在高中数学课就学过的内置对象。它知道解决最复杂的数学问题的所有公式，如果给它要处理的数字，即能计算出结果。

`Math` 对象有几个属性，主要是数学界的专用值。下表列出了这些属性：

属性	说明
E	值 e, 自然对数的底
LN10	10 的自然对数
LN2	2 的自然对数
LOG2E	以 2 为底 e 的对数
LOG10E	以 10 为底 e 的对数
PI	值 π
SQRT1_2	1/2 的平方根
SQRT2	2 的平方根

虽然这些值的意义与用法不在本书讨论范围内，但如果清楚它们是什么，在需要时，即可使用它们。

Math 对象还包括许多专门用于执行简单的及复杂的数学计算的方法。

方法 `min()` 和 `max()` 用于判断一组数中的最大值和最小值。这两个方法都可接受任意多个参数：

```
var iMax = Math.max(3, 54, 32, 16);
alert(iMax); //outputs "54"
var iMin = Math.min(3, 54, 32, 16);
alert(iMin); //outputs "3"
```

84

对于数字 3、54、32 和 16，`max()` 返回 54，`min()` 返回 3。用这些方法，可免去用循环或 `if` 语句来判断一组数中的最大值。

另一个方法 `abs()` 返回数字的绝对值。绝对值是负数的正值版本（正数的绝对值就是它自身）。

```
var iNegOne = Math.abs(-1);
alert(iNegOne); //outputs "1"
var iPosOne = Math.abs(1);
alert(iPosOne); //outputs "1"
```

这个例子中，`abs(-1)` 返回 1，`abs(1)` 也返回 1。

下一组方法用于把小数舍入成整数。处理舍入操作的方法有三个，即 `ceil()`、`floor()` 和 `round()`，它们的处理方法不同：

- 方法 `ceil()` 表示向上舍入函数，总是把数字向上舍入到最接近的值。
- 方法 `floor()` 表示向下舍入函数，总是把数字向下舍入到最接近的值。
- 方法 `round()` 表示标准的舍入函数，如果数字与下一个整数的差不超过 0.5，则向上舍入，否则向下舍入。这是在初中学过的舍入规则。

为说明每种方法的处理方式，考虑使用值 25.5：

```
alert(Math.ceil(25.5)); //outputs "26"
alert(Math.round(25.5)); //outputs "26"
alert(Math.floor(25.5)); //outputs "25"
```

对于 `ceil()` 和 `round()`，传递 25.5，返回的是 26，而 `floor()` 返回的是 25。注意不要交替使用这些方法，因为最后可能得到与预期不符的结果。

另一组方法与指数的用法有关。这些方法包括 `exp()`，用于把 `Math.E` 升到指定的幂；`log()` 用于返回特定数字的自然对数；`pow()` 用于把指定的数字升到指定的幂；`sqrt()` 用于返回指定数字的平方根。

方法 `exp()` 和 `log()` 本质上功能相反，`exp()` 把 `Math.E` 升到特定的幂，`log()` 则判断 `Math.E` 的多少次指数才等于指定的值。例如：

```
85 var iNum = Math.log(Math.exp(10));
      alert(iNum);
```

这里，首先用 `exp()` 把 `Math.E` 升到 10 次幂，然后 `log()` 返回 10，即等于数字 `iNum` 必需的指数。很多人都对此感到迷茫。全世界的高中生和数学系的大学生都被此类问题难倒过。如果你对自然对数一无所知，那么有可能永远都不需要为它编写代码。

方法 `pow()` 用于把数字升到指定的幂，如把 2 升到 10 次幂（在数学中表示为 2^{10} ）：

```
var iNum = Math.pow(2, 10);
```

`pow()` 的第一个参数是基数，此例子中是 2。第二个参数是要升到的幂，此例子中是 10。

不建议把 `Math.E` 作为 `pow()` 方法的基数。最好使用 `exp()` 对 `Math.E` 进行升幂运算，因为它是专用运算，计算出的值更精确。

这组方法中的最后一个方法是 `sqrt()`，用于返回指定数字的平方根。它只有一个参数，即要求平方根的数字。要求 4 的平方根，只需要用一行代码：

```
var iNum = Math.sqrt(4);
      alert(iNum); //outputs "2"
```

当然，4 的平方根是 2，就是这行代码的输出。

你也许会问“为什么平方根必须利用指数”？实际上，数字平方根就是它的 $1/2$ 次幂。例如， $2^{1/2}$ 就是 2 的平方根。

`Math` 对象还有一整套三角函数方法。下表列出了这些方法：

方 法	说 明
<code>acos(x)</code>	返回 x 的反余弦值
<code>asin(x)</code>	返回 x 的反正弦值
<code>atan(x)</code>	返回 x 的反正切值
<code>atan2(y, x)</code>	返回 y/x 的反余弦值
<code>cos(x)</code>	返回 x 的余弦值
<code>sin(x)</code>	返回 x 的正弦值
<code>tan(x)</code>	返回 x 的正切值

即使这些方法是 ECMA-262 定义的，结果也是由实现决定的，因为每个值的计算方法都有很多，从而使得不同的实现生成的结果的精度也不同。

Math对象的最后一个方法是random(), 该方法返回一个0到1之间的随机数, 不包括0和1。这是在主页上显示随机引述或新闻的站点常用的工具。可用下面的形式调用random()方法, 在某个范围内选择随机数:

```
number = Math.floor(Math.random() * total_number_of_choices + first_possible_value)
```

这里使用方法floor(), 因为random()返回的都是小数值, 也就是说, 用它乘以一个数, 然后再加上一个数, 得到的仍然是小数值。通常你想选择一个随机整数值。因此, 必须使用floor()方法。如果想选择一个1到10之间的数, 代码如下:

```
var iNum = Math.floor(Math.random() * 10 + 1);
```

可能出现的值有10个(1到10), 这些值中的第一个是1。如果想选择2到10之间的值, 代码如下:

```
var iNum = Math.floor(Math.random() * 9 + 2);
```

从2到10, 只有9个数字, 所以选项总数为9, 其中第一个值是2。许多时候, 使用计算选项总数的函数和第一个可用的值更容易些:

```
function selectFrom(iFirstValue, iLastValue) {
    var iChoices = iLastValue - iFirstValue + 1;
    return Math.floor(Math.random() * iChoices + iFirstValue);
}

//select from between 2 and 10
var iNum = selectFrom(2, 10);
```

使用函数, 可很容易地选择Array中的随机项:

```
var aColors = ["red", "green", "blue", "yellow", "black", "purple", "brown"];
var sColor = aColors[selectFrom(0, aColors.length-1)];
```

这里, selectFrom()函数的第二个参数是数组的长度减1, 即数组中最后一个元素的位置。

3.3.3 宿主对象

所有非本地对象都是宿主对象(host object), 即由ECMAScript实现的宿主环境提供的对象。所有BOM和DOM对象都是宿主对象, 本书将在后面的章节讨论它们。

87

3.4 作用域

任何程序设计语言的程序员都懂得作用域的概念, 即某些变量的适用范围。

3.4.1 公用、受保护和私有作用域

在传统的面向对象程序设计中, 主要关注于公用和私有作用域。公用作用域中的对象属性可以从对象外部访问, 即开发者创建对象的实例后, 就可使用它的公用属性。而私有作用域中的属

性只能在对象内部访问，即对于外部世界来说，这些属性并不存在。这也意味着如果类定义了私有属性和方法，则它的子类也不能访问这些属性和方法。

最近，另一种作用域流行起来，即受保护作用域。虽然在不同语言中，受保护作用域的应用的规则不同，但一般说来，它都用于定义私有的属性和方法，只是这些属性和方法还能被其子类访问。

对 ECMAScript 讨论这些作用域几乎毫无意义，因为 ECMAScript 中只存在一种作用域——公用作用域。ECMAScript 中的所有对象的所有属性和方法都是公用的。因此，定义自己的类和对象时，必须格外小心。记住，所有属性和方法默认都是公用的。

许多开发者都在网上提出了有效的属性作用域模式，解决了 ECMAScript 的这种问题。由于缺少私有作用域，开发者们制定了一个规约，说明哪些属性和方法应该被看作私有的。这种规约规定在属性名前后加下划线。例如：

```
obj._color_ = "red";
```

这段代码中，属性 `color` 是私有的。记住，这些下划线并不改变这些属性是公用属性的事实，它只是告诉其他开发者，应该把该属性看作私有的。

有些开发者还喜欢用单下划线说明私有成员，例如 `obj._color`。

3.4.2 静态作用域并非静态的

静态作用域定义的属性和方法任何时候都能从同一个位置访问。在 Java 中，类可具有静态属性和方法，无需实例化该类的对象，即可访问这些属性和方法，例如 `java.net.URLEncoder` 类，它的函数 `encode()` 即是静态方法。

严格说来，ECMAScript 并没有静态作用域。不过，它可以给构造函数提供属性和方法。还记得吗，构造函数只是函数。函数是对象，对象可以有属性和方法。例如：

```
function sayHi() {
    alert("hi");
}

sayHi.alternate = function() {
    alert("hola");
};

sayHi();           //outputs "hi"
sayHi.alternate(); //outputs "hola"
```

这里，方法 `alternate()` 实际上是函数 `sayHi` 的方法。可以像调用常规函数一样调用 `sayHi()` 输出 "hi"，也可以调用 `sayHi.alternate()` 输出 "hola"。即使如此，`alternate()` 也是 `sayHi()` 公用作用域中的方法，而不是静态方法。

3.4.3 关键字 `this`

在 ECMAScript 中，要掌握的最重要的概念之一是关键字 `this` 的用法，它用在对象的方法

中。关键字 this 总是指向调用该方法的对象，例如：

```
var oCar = new Object;
oCar.color = "red";
oCar.showColor = function () {
    alert(this.color); //outputs "red"
};
```

这里，关键字 this 用在对象的 showColor() 方法中。在此环境中，this 等于 car，下面的代码与上面代码的功能相同：

```
var oCar = new Object;
oCar.color = "red";
oCar.showColor = function () {
    alert(oCar.color); //outputs "red"
};
```

那么为什么使用 this 呢？因为在实例化对象时，总是不能确定开发者会使用什么样的变量名。使用 this，即可在任意多个地方重用同一个函数。考虑下面的例子：

```
function showColor() {
    alert(this.color);
}

var oCar1 = new Object;
oCar1.color = "red";
oCar1.showColor = showColor;

var oCar2 = new Object;
oCar2.color = "blue";
oCar2.showColor = showColor;

oCar1.showColor(); //outputs "red"
oCar2.showColor(); //outputs "blue"
```

89

在这段代码中，首先用 this 定义函数 showColor()，然后创建两个对象（oCar1 和 oCar2），一个对象的 color 属性被设置为 "red"，另一个对象的 color 属性被设置为 "blue"。两个对象都被赋予了属性 showColor，指向原始的 showColor() 函数（注意这里不存在命名问题，因为一个是全局函数，而另一个是对象的属性）。调用每个对象的 showColor() 方法，oCar1 的输出 "red"，而 oCar2 的输出 "blue"。这是因为调用 oCar1.showColor() 时，函数中的 this 关键字等于 oCar1，调用 oCar2.showColor() 时，函数中的 this 关键字等于 oCar2。

注意，引用对象的属性时，必须使用 this 关键字。例如，如果采用下面的代码，showColor() 方法不能运行：

```
function showColor() {
    alert(color);
}
```

如果不使用对象或 this 关键字引用变量，ECMAScript 就会把它看作局部变量或全局变量。然后该函数将查找名为 color 的局部或全局变量，但是不会找到的。结果如何？该函数将在警告中

显示“null”。

3.5 定义类或对象

使用预定义对象的能力只是面向对象语言的能力的一部分，它真正的强大之处在于能够创建自己专用的类和对象。与 ECMAScript 中的许多特性一样，可以用各种方法实现这一点。

3.5.1 工厂方式

因为对象的属性可在对象创建后动态定义，所以许多开发者都在初次引入 JavaScript 时编写类似下面的代码：

```
var oCar = new Object;
oCar.color = "red";
oCar.doors = 4;
oCar.mpg = 23;
oCar.showColor = function () {
    alert(this.color);
};
```

在这段代码中，创建对象 car。然后给它设置几个属性：它的颜色是红色，有四个门，每加仑油可跑 23 英里。最后一个属性实际上是指向函数的指针，意味着该属性是个方法。执行这段代码后，就可以使用对象 car。问题是可能需要创建多个 car 实例。

要解决此问题，开发者创造了能创建并返回特定类型的对象的工厂函数（factory function）。

例如，函数 createCar() 可用于封装前面列出的创建 car 对象的操作：

```
function createCar() {
    var oTempCar = new Object;
    oTempCar.color = "red";
    oTempCar.doors = 4;
    oTempCar.mpg = 23;
    oTempCar.showColor = function () {
        alert(this.color)
    };
    return oTempCar;
}

var oCar1 = createCar();
var oCar2 = createCar();
```

这里，前面的所有代码都包含在 createCar() 函数中，此外还有一行额外的代码，返回 car 对象作为 (oTempCar) 函数值。调用此函数时，将创建新对象，并赋予它所有必要的属性，复制出一个前面说明的 car 对象。使用该方法，可以容易地创建 car 对象的两个版本 (oCar1 和 oCar2)，它们的属性完全一样。当然，还可以修改 createCar() 函数，给它传递各个属性的默认值，而不是赋予属性默认值：

```

function createCar(sColor, iDoors, iMpg) {
    var oTempCar = new Object();
    oTempCar.color = sColor;
    oTempCar.doors = iDoors;
    oTempCar.mpg = iMpg;
    oTempCar.showColor = function () {
        alert(this.color)
    };

    return oTempCar;
}

var oCar1 = createCar("red", 4, 23);
var oCar2 = createCar("blue", 3, 25);
oCar1.showColor();      //outputs "red"
oCar2.showColor();      //outputs "blue"

```

给 `createCar()` 函数加上参数，即可为要创建的 `car` 对象的 `color`、`doors` 和 `mpg` 属性赋值。这使两个对象具有相同的属性，却有不同的属性值。

虽然 ECMAScript 越来越正式化，但创建对象的方法却被置之不理，且其规范化至今还遭人反对。一部分是语义上的原因（它看起来不像使用带有构造函数的 `new` 运算符那么正规），一部分是功能上的原因。功能问题在于用这种方式必须创建对象的方法。前面的例子中，每次调用函数 `createCar()`，都要创建新函数 `showColor()`，意味着每个对象都有自己的 `showColor()` 版本，事实上，每个对象都共享了同一个函数。91

有些开发者在工厂函数外定义对象的方法，然后通过属性指向该方法，从而避开这个问题：

```

function showColor() {
    alert(this.color);
}

function createCar(sColor, iDoors, iMpg) {
    var oTempCar = new Object();
    oTempCar.color = sColor;
    oTempCar.doors = iDoors;
    oTempCar.mpg = iMpg;
    oTempCar.showColor = showColor;
    return oTempCar;
}

var oCar1 = createCar("red", 4, 23);
var oCar2 = createCar("blue", 3, 25);
oCar1.showColor();      //outputs "red"
oCar2.showColor();      //outputs "blue"

```

在这段重写的代码中，在函数 `createCar()` 前定义了函数 `showColor()`。在 `createCar()` 内部，赋予对象一个指向已经存在的 `showColor()` 函数的指针。从功能上讲，这样解决了重复创建函数对象的问题，但该函数看起来不像对象的方法。

所有这些问题引发了开发者定义的构造函数的出现。

3.5.2 构造函数方式

创建构造函数就像定义工厂函数一样容易。第一步选择类名，即构造函数的名字。根据惯例，这个名字的首字母大写，以使它与首字母通常是小写的变量名区分开。除了这点不同，构造函数看起来很像工厂函数。考虑下面的例子：

```
function Car(sColor, iDoors, iMpg) {
    this.color = sColor;
    this.doors = iDoors;
    this.mpg = iMpg;
    this.showColor = function () {
        alert(this.color)
    };
}

var oCar1 = new Car("red", 4, 23);
var oCar2 = new Car("blue", 3, 25);
```

你可能已经注意到第一个差别了，在构造函数内部无创建对象，而是使用 `this` 关键字。使用 `new` 运算符调用构造函数时，在执行第一行代码前先创建一个对象，只有用 `this` 才能访问该对象。然后可以直接赋予 `this` 属性，默认情况下是构造函数的返回值（不必明确使用 `return` 运算符）。

92

现在，用 `new` 运算符和类名 `car` 创建对象，就更像创建 ECMAScript 中一般对象了。你也许会问，这种方式在管理函数方面是否存在与前一种方式相同的问题呢？是的。

就像工厂函数，构造函数会重复生成函数，为每个对象都创建独立的函数版本。不过，与工厂函数相似，也可以用外部函数重写构造函数，同样的，这么做语义上无任何意义。这正是下面要讲的原型方式的优势所在。

3.5.3 原型方式

该方式利用了对象的 `prototype` 属性，可把它看成创建新对象所依赖的原型。这里，用空构造函数来设置类名。然后所有的属性和方法都被直接赋予 `prototype` 属性。重写前面的例子，代码如下所示：

```
function Car() {}

Car.prototype.color = "red";
Car.prototype.doors = 4;
Car.prototype.mpg = 23;
Car.prototype.showColor = function () {
    alert(this.color);
};

var oCar1 = new Car();
var oCar2 = new Car();
```

在这段代码中，首先定义构造函数（Car），其中无任何代码。接下来的几行代码，通过给 Car 的 prototype 属性添加属性去定义 Car 对象的属性。调用 new Car() 时，原型的所有属性都被立即赋予要创建的对象，意味着所有 Car 实例存放的都是指向 showColor() 函数的指针。从语义上讲，所有属性看起来都属于一个对象，因此解决了前面两种方式存在的两个问题。此外，使用该方法，还能用 instanceof 运算符检查给定变量指向的对象的类型。因此，下面的代码将输出 true：

```
alert(oCar1 instanceof Car); //outputs "true"
```

看起来是个非常好的解决方案。遗憾的是，它并不尽如人意。

首先，这个构造函数没有参数。使用原型方式时，不能通过给构造函数传递参数初始化属性的值，因为 car1 和 car2 的 color 属性都等于 "red"，doors 属性都等于 4，mpg 属性都等于 23。这意味着必须在对象创建后才能改变属性的默认值，这点很令人讨厌，但还没完。真正的问题出现在属性指向的是对象，而不是函数时。函数共享不会造成任何问题，但对象却很少是被多个实例共享的。考虑下面的例子：

```
function Car() {
}
Car.prototype.color = "red";
Car.prototype.doors = 4;
Car.prototype.mpg = 23;
Car.prototype.drivers = new Array("Mike", "Sue");
Car.prototype.showColor = function () {
    alert(this.color);
};

var oCar1 = new Car();
var oCar2 = new Car();

oCar1.drivers.push("Matt");

alert(oCar1.drivers); //outputs "Mike,Sue,Matt"
alert(oCar2.drivers); //outputs "Mike,Sue,Matt"
```

93

这里，属性 drivers 是指向 Array 对象的指针，该数组中包含两个名字 "Mike" 和 "Sue"。由于 drivers 是引用值，Car 的两个实例都指向同一个数组。这意味着给 car1.drivers 添加值 "Matt"，在 car2.drivers 中也能看到。输出这两个指针中的任何一个，结果都是显示字符串 "Mike,Sue,Matt"。

由于创建对象时有这么多问题，你一定会想，是否有种合理的创建对象的方法呢？答案是有，需要联合使用构造函数和原型方式。

3.5.4 混合的构造函数/原型方式

联合使用构造函数和原型方式，就可像用其他程序设计语言一样创建对象。这种概念非常简单，即用构造函数定义对象的所有非函数属性，用原型方式定义对象的函数属性（方法）。结果

所有函数都只创建一次，而每个对象都具有自己的对象属性实例。再重写前面的例子，代码如下：

```
function Car(sColor, iDoors, iMpg) {
    this.color = sColor;
    this.doors = iDoors;
    this.mpg = iMpg;
    this.drivers = new Array("Mike", "Sue");
}

Car.prototype.showColor = function () {
    alert(this.color);
};

var oCar1 = new Car("red", 4, 23);
var oCar2 = new Car("blue", 3, 25);

oCar1.drivers.push("Matt");

alert(oCar1.drivers); //outputs "Mike,Sue,Matt"
alert(oCar2.drivers); //outputs "Mike,Sue"
```

94

现在就更像创建一般对象了。所有的非函数属性都在构造函数中创建，意味着又可用构造函数的参数赋予属性默认值了。因为只创建 `showColor()` 函数的一个实例，所以没有内存浪费。此外，给 `oCar1` 的 `drivers` 数组添加“Matt”值，不会影响 `oCar2` 的数组，所以输出这些数组的值时，`oCar1.drivers` 显示的是“Mike, Sue, Matt”，而 `oCar2.drivers` 显示的是“Mike, Sue”。因为使用了原型方式，所以仍然能利用 `instanceof` 运算符判断对象的类型。

这种方式是 ECMAScript 采用的主要方式，它具有其他方式的特性，却没有它们的副作用。不过，有些开发者仍觉得这种方法不够完美。

3.5.5 动态原型方法

对于习惯使用其他语言的开发者来说，使用混合的构造函数/原型方式感觉不那么和谐。毕竟，定义类时，大多数面向对象语言都对属性和方法进行了视觉上的封装。考虑下面的 Java 类：

```
class Car {
    public String color = "red";
    public int doors = 4;
    public int mpg = 23;

    public Car(String color, int doors, int mpg) {
        this.color = color;
        this.doors = doors;
        this.mpg = mpg;
    }

    public void showColor() {
        System.out.println(color);
    }
}
```

Java很好地打包了Car类的所有属性和方法，因此看见这段代码就知道它要实现什么功能，它定义了一个对象的信息。批评混合的构造函数/原型方式的人认为，在构造函数内存找属性，在其外部找方法的做法不合逻辑。因此，他们设计了动态原型方法，以提供更友好的编码风格。

动态原型方法的基本想法与混合的构造函数/原型方式相同，即在构造函数内定义非函数属性，而函数属性则利用原型属性定义。唯一的区别是赋予对象方法的位置。下面是用动态原型方法重写的Car类：

```
function Car(sColor, iDoors, iMpg) {
    this.color = sColor;
    this.doors = iDoors;
    this.mpg = iMpg;
    this.drivers = new Array("Mike", "Sue");

    if (typeof Car._initialized == "undefined") {
        Car.prototype.showColor = function () {
            alert(this.color);
        };

        Car._initialized = true;
    }
}
```

直到检查`typeof Car._initialized`是否等于"undefined"之前，这个构造函数都未发生变化。这行代码是动态原型方法中最重要的部分。如果这个值未定义，构造函数将用原型方式继续定义对象的方法，然后把`Car._initialized`设置为`true`。如果这个值定义了（它的值为`true`时，`typeof`的值为`Boolean`），那么就不再创建该方法。简而言之，该方法使用标志`(_initialized)`来判断是否已给原型赋予了任何方法。该方法只创建并赋值一次，传统的OOP开发者会高兴地发现，这段代码看起来更像其他语言中的类定义了。

3.5.6 混合工厂方式

这种方式通常是在不能应用前一种方式时的变通方法。它的目的是创建假构造函数，只返回另一种对象的新实例。这段代码看来与工厂函数非常相似：

```
function Car() {
    var oTempCar = new Object();
    oTempCar.color = "red";
    oTempCar.doors = 4;
    oTempCar.mpg = 23;
    oTempCar.showColor = function () {
        alert(this.color)
    };

    return oTempCar;
}
```

与经典方式不同，这种方式使用`new`运算符，使它看起来像真正的构造函数：

```
var car = new Car();
```

由于在 `Car()` 构造函数内部调用了 `new` 运算符，所以将忽略第二个 `new` 运算符（位于构造函数之外）。在构造函数内部创建的对象被传递回变量 `var`。

96

这种方式在对象方法的内部管理方面与经典方式有着相同的问题。强烈建议：除非万不得已（请参阅第 15 章），还是避免使用这种方式。

3.5.7 采用哪种方式

如前所述，目前使用最广泛的是混合的构造函数/原型方式。此外，动态原型方法也很流行，在功能上与构造函数/原型方式等价。可以采用这两种方式中的任何一种。不过不要单独使用经典的构造函数或原型方式，因为这样会给代码引入问题。

3.5.8 实例

对象令人感兴趣的一点是用它们解决问题的方式。ECMAScript 中最常见一个问题就是字符串连接的性能。与其他语言类似，ECMAScript 的字符串是不可变的，即它们的值不能改变。考虑下面的代码：

```
var str = "hello ";
str += "world";
```

实际上，这段代码在幕后执行的步骤如下：

- (1) 创建存储"hello"的字符串。
- (2) 创建存储"world"的字符串。
- (3) 创建存储连接结果的字符串。
- (4) 把 `str` 的当前内容复制到结果中。
- (5) 把"world"复制到结果中。
- (6) 更新 `str`，使它指向结果。

每次完成字符串连接都会执行步骤 2 到 6，使得这种操作非常消耗资源。如果重复这一过程几百次，甚至几千次，就会造成性能问题。解决方法是用 `Array` 对象存储字符串，然后用 `join()` 方法（参数是空字符串）创建最后的字符串。想像用下面的代码代替前面的代码：

```
var arr = new Array;
arr[0] = "hello ";
arr[1] = "world";
var str = arr.join("");
```

这样，无论在数组中引入多少字符串都不成问题，因为只在调用 `join()` 方法时才会发生连接操作。此时，执行的步骤如下：

- (1) 创建存储结果的字符串。
- (2) 把每个字符串复制到结果中的合适位置。

虽然这种解决方法很好，但还有更好的方法。问题是这段代码不能确切反映出它的意图。要

使它更容易理解，可以用 `StringBuffer` 类打包该功能：

```
function StringBuffer() {
    this._strings_ = new Array();
}
StringBuffer.prototype.append = function (str) {
    this._strings_.push(str);
};

StringBuffer.prototype.toString = function () {
    return this._strings_.join("");
};
```

这段代码首先要注意的是 `strings` 属性，本意是私有属性。它只有两个方法，即 `append()` 和 `toString()` 方法。`append()` 方法有一个参数，它把该参数附加到字符串数组中，`toString()` 方法调用数组的 `join()` 方法，返回真正连接成的字符串。要用 `StringBuffer` 对象连接一组字符串，可以用下面的代码：

```
var buffer = new StringBuffer();
buffer.append("hello ");
buffer.append("world");
var result = buffer.toString();
```

可用下面代码测试 `StringBuffer` 对象和传统的字符串连接方法的性能：

```
var d1 = new Date();
var str = "";
for (var i=0; i < 10000; i++) {
    str += "text";
}
var d2 = new Date();

document.write("Concatenation with plus: " + (d2.getTime() - d1.getTime()) + " milliseconds");

var oBuffer = new StringBuffer();
d1 = new Date();
for (var i=0; i < 10000; i++) {
    oBuffer.append("text");
}
var sResult = buffer.toString();
d2 = new Date();

document.write("<br />Concatenation with StringBuffer: " + (d2.getTime() - d1.getTime()) + " milliseconds");
```

这段代码对字符串连接进行两个测试，第一个使用加号，第二个使用 `StringBuffer` 类。每个操作都连接 10000 个字符串。日期值 `d1` 和 `d2` 用于判断完成操作需要的时间。记住，创建新 `Date` 对象时，如果没有参数，赋予对象的是当前的日期与时间。要计算连接操作历经多少时间，把日期的毫秒表示（`getTime()` 方法的返回值）相减即可。这是衡量 JavaScript 性能的常用方法。该测试的结果应该说明使用 `StringBuffer` 类比使用加号节省了 50%~66% 的时间。

3.6 修改对象

创建对象只是使用 ECMAScript 的乐趣的一部分。你喜欢修改已有对象的行为吗？这在 ECMAScript 中是完全可能的，所以可为 `String`、`Array`、`Number` 或其他任意一种对象设计出你想要的任何方法，因为有无限的可能性。

还记得本章前面的小节中介绍的 `prototype` 属性吗？你已经知道，每个构造函数都有个 `prototype` 属性，可用于定义方法。你还不知道的是，在 ECMAScript 中，每个本地对象也有个用法完全相同的 `prototype` 属性。

3.6.1 创建新方法

可以用 `prototype` 属性为任何已有的类定义新方法，就像处理自己的类一样。例如，还记得 `Number` 类的 `toString()` 方法吗，如果给它传递 16，它将输出十六进制的字符串。难道用 `toHexString()` 方法处理这个操作不是更好吗？创建它很简单：

```
Number.prototype.toHexString = function () {
    return this.toString(16);
};
```

在此环境中，关键字 `this` 指向 `Number` 的实例，因此可完全访问 `Number` 的所有方法。有了这段代码，可实现下面操作：

```
var iNum = 15;
alert(iNum.toHexString()); //outputs "F"
```

由于数字 15 等于十六进制中的 F，因此警告将显示“F”。还记得将数组用作队列的讨论吗？唯一漏掉的是命名正确的方法。可以给 `Array` 类添加两个方法 `enqueue()` 和 `dequeue()`，只让它们反复调用已有的 `push()` 和 `shift()` 方法即可：

```
Array.prototype.enqueue = function(vItem) {
    this.push(vItem);
};

Array.prototype.dequeue = function() {
    return this.shift();
};
```

当然，还可添加与已有方法无关的方法。例如，假设要判断某个项在数组中的位置，没有本地方法可以做这种事情。则可以轻松地创建下面的方法：

```
Array.prototype.indexOf = function (vItem) {
    for (var i=0; i < this.length; i++) {
        if (vItem == this[i]) {
            return i;
        }
    }
    return -1;
}
```

该方法 `indexOf()` 与 `String` 类的同名方法保持一致，在数组中检索每个项，直到发现与传进来的项相等的项为止。如果找到相等的项，则返回该项的位置，否则，返回-1。使用这种定义，可以采用下面的代码：

```
var aColors = new Array("red", "green", "yellow");
alert(aColors.indexOf("green")); //outputs "1"
```

最后，如果想给 ECMAScript 中的每个本地对象添加新方法，必须在 `Object` 对象的 `prototype` 属性上定义它。如上一章所述，所有本地对象都继承了 `Object` 对象，所以对 `Object` 对象做任何改变，都会反应在所有本地对象中。例如，如果想添加一个用警告输出对象的当前值的方法，可以采用下面的代码：

```
Object.prototype.showValue = function () {
    alert(this.valueOf());
}

var str = "hello";
var iNum = 25;
str.showValue(); //outputs "hello"
iNum.showValue(); //outputs "25"
```

这里，`String` 和 `Number` 对象都从 `Object` 对象继承了 `showValue()` 方法，分别在它们的对象上调用该方法，将显示 "hello" 和 "25"。

3.6.2 重定义已有方法

就像能给已有的类定义新方法一样，也可重定义已有的方法。如前一章所述，函数名只是指向函数的指针，因此可以轻易地使它指向其他函数。如果修改了本地方法，如 `toString()`，会出现什么情况呢？

```
Function.prototype.toString = function () {
    return "Function code hidden";
}
```

前面的代码完全合法，运行结果完全符合预期：

```
function sayHi() {
    alert("hi");
}

alert(sayHi.toString()); //outputs "Function code hidden"
```

也许你还记得，第 2 章中介绍过 `Function` 的 `toString()` 方法通常输出的是函数的源代码。覆盖该方法，可以返回另一个字符串（在这个例子中，返回 "Function code hidden"）。不过，`toString()` 指向的原始函数怎样了呢？它将被无用存储单元回收程序回收，因为它被完全废弃了。没能够恢复原始函数的办法，所以在覆盖原始方法前，比较安全的做法是存储它的指针，以便以后的使用。有时你甚至可能在新方法中调用原始方法：

```
Function.prototype.originalToString = Function.prototype.toString;

Function.prototype.toString = function () {
    if (this.originalToString().length > 100) {
        return "Function too long to display.";
    } else {
        return this.originalToString();
    }
};
```

在这段代码中，第一行代码把对当前 `toString()` 方法的引用保存在属性 `originalToString` 中。然后用定制的方法覆盖了 `toString()` 方法。新方法将检查该函数源代码的长度是否大于 100。如果是，就返回错误消息，说明该函数代码太长，否则调用 `originalToString()` 方法，返回函数的源代码。

3.6.3 极晚绑定

从技术上来说，根本不存在极晚绑定。本书采用该术语描述 ECMAScript 中的一种现象，即能够在对象实例化后再定义它的方法。例如：

```
var o = new Object;

Object.prototype.sayHi = function () {
    alert("hi");
};

o.sayHi();
```

在大多数程序设计语言中，必须在实例化对象之前定义对象的方法。这里，方法 `sayHi()` 是在创建 `Object` 类的一个实例后才添加进来的。在传统语言中不仅没听说过这种操作，也没听说过该方法还会自动赋予 `Object` 的实例并能立即使用（接下来的一行）。

不建议使用极晚绑定方法，因为很难对其跟踪和记录。不过，还是应该了解这种可能。

101

3.7 小结

ECMAScript 语言为 JavaScript 实现提供了完整的面向对象语言能力。在这一章中，学到了 ECMA-262 中定义的三种不同类型的对象：本地对象、内置对象和宿主对象。

探讨了 `Array` 对象和 `Date` 对象，学习了它们的方法、属性及各种古怪之处。此外还学习了两个内置对象，即 `Global` 和 `Math` 对象，并了解了 `Global` 对象与其他对象的区别。

这一章还介绍了完全定义自己的对象的能力，其中探讨了几种实现该能力的方法，讨论了它们的优点及缺点。

最后，学习了如何修改已有对象，加入新的方法或覆盖已有的方法。

下一章将结束对 JavaScript Core——ECMAScript 的介绍，讨论继承性的问题。

102

真正的面向对象语言必须支持继承机制，即一个类能够重用（继承）另一个类的方法和属性。在上一章中，学会了如何定义类的属性和方法，如果想让两个类使用同样的方法该如何做呢？这就需要引入继承机制。

4.1 继承机制实例

说明继承机制最简单的方法是，利用一个经典的例子——几何形状。实际上，几何形状只有两种，即椭圆形（是圆形的）和多边形（具有一定数量的边）。圆是椭圆形的一种，它只有一个焦点。三角形、矩形和五边形都是多边形的一种，具有不同数量的边。正方形是矩形的一种，所有的边等长。这就构成了一种完美的继承关系。

在这个例子中，形状（Shape）是椭圆形（Ellipse）和多边形（Polygon）的基类（base class）（所有类都由它继承而来）。椭圆具有一个属性 *foci*，说明椭圆具有的焦点的个数。圆形（Circle）继承了椭圆形，因此圆形是椭圆形的子类（subclass），椭圆形是圆形的超类（superclass）。同样，三角形（Triangle）、矩形（Rectangle）和五边形（Pentagon）都是多边形的子类，多边形是它们的超类。最后，正方形（Square）继承了矩形。

最好用图来解释这种继承关系，这是 UML（统一建模语言）的用武之地。UML 的主要用途之一是，可视化地表示像继承这样的复杂对象关系。图 4-1 是解释 Shape 和它的子类之间关系的 UML 图示。

在 UML 中，每个方框表示一个类，由类名说明。Triangle、Rectangle 和 Pentagon 顶部的线段汇集在一起，指向 Shape，说明这些类都由 Shape 继承而来。同样，从 Square 指向 Rectangle 的箭头说明了它们之间的继承关系。

如果有兴趣学习 UML，可以参考三位 UML 创始人所著的《UML 用户指南（第二版）》¹。

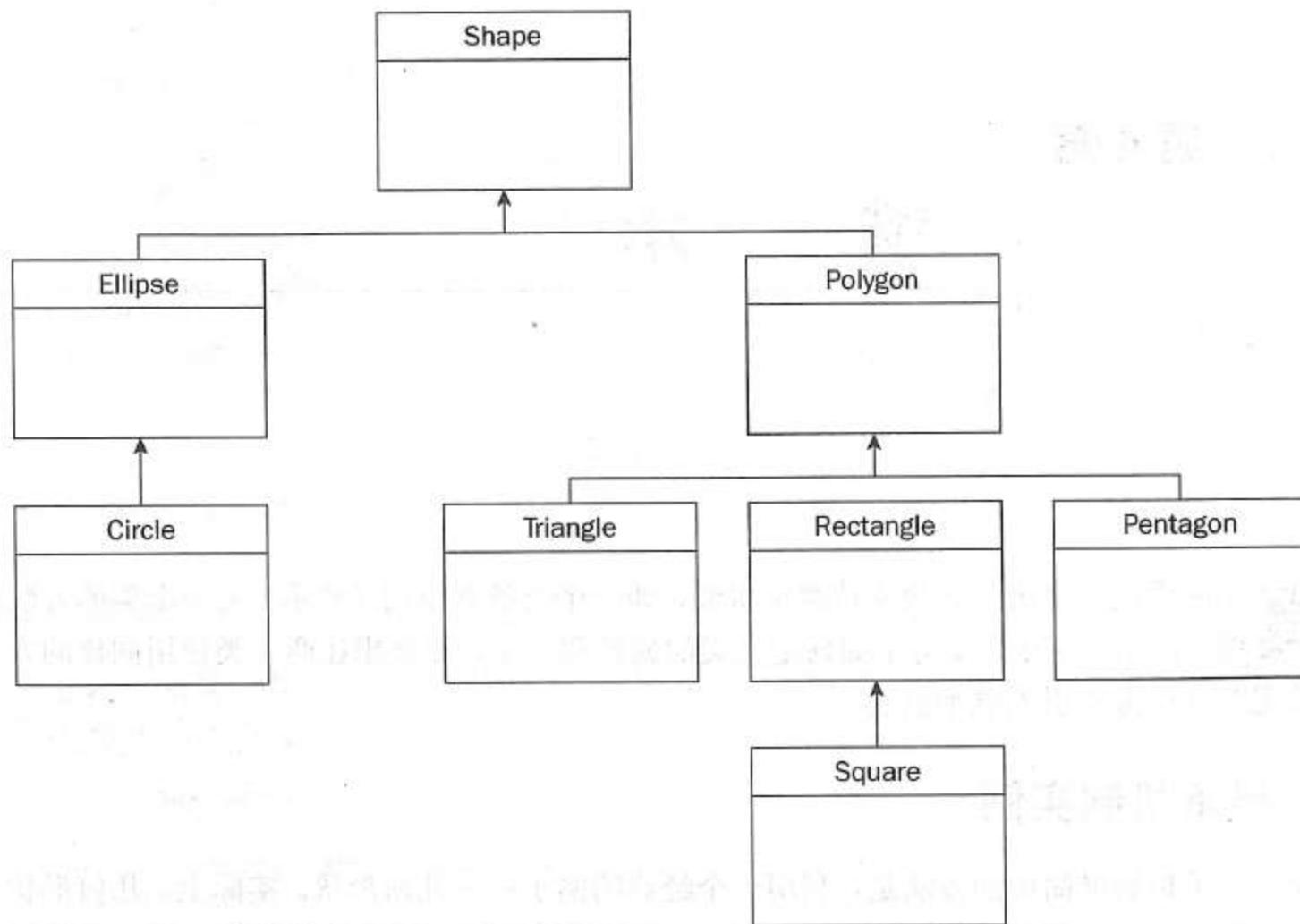


图 4-1

4.2 继承机制的实现

要用 ECMAScript 实现继承机制，首先从基类入手。所有开发者定义的类都可作为基类。出于安全原因，本地类和宿主类不能作为基类，这样可以防止公用访问编译过的浏览器级的代码，因为这些代码可以被用于恶意攻击。

选定基类后，就可以创建它的子类了。是否使用基类完全由你决定。有时，你可能想创建一个不能直接使用的基类，它只是用于给子类提供通用的函数。在这种情况下，基类被看作抽象类。

尽管 ECMAScript 并没有像其他语言那样严格地定义抽象类，但有时它的确会创建一些不允许使用的类。通常，我们称这种类为抽象类。

创建的子类将继承超类的所有属性和方法，包括构造函数及方法的实现。记住，所有属性和方法都是公用的，因此子类可直接访问这些方法。子类还可添加超类中没有的新属性和方法，也可以覆盖超类中的属性和方法。

4.2.1 继承的方式

和其他功能一样，ECMAScript 中实现继承的方式不止一种。这是因为 JavaScript 中的继承

机制并不是明确规定，而是通过模仿实现的。这意味着所有的继承细节并非完全由解释程序处理。作为开发者，你有权决定最适用的继承方式。

1. 对象冒充

构想原始的 ECMAScript 时，根本没打算设计对象冒充（object masquerading）。它是在开发者开始理解函数的工作方式，尤其是如何在函数环境中使用 `this` 关键字后才发展出来的。

其原理如下：构造函数使用 `this` 关键字给所有属性和方法赋值（即采用类声明的构造函数方式）。因为构造函数只是一个函数，所以可使 `ClassA` 的构造函数成为 `ClassB` 的方法，然后调用它。`ClassB` 就会收到 `ClassA` 的构造函数中定义的属性和方法。例如，用下面的方式定义 `ClassA` 和 `ClassB`：

```
function ClassA(sColor) {
    this.color = sColor;
    this.sayColor = function () {
        alert(this.color);
    };
}

function ClassB(sColor) {
```

还记得吗？关键字 `this` 引用的是构造函数当前创建的对象。不过在这个方法中，`this` 指向的是所属的对象。这个原理是把 `ClassA` 作为常规函数来建立继承机制，而不是作为构造函数。如下使用构造函数 `ClassB` 可以实现继承机制：

```
function ClassB(sColor) {
    this.newMethod = ClassA;
    this.newMethod(sColor);
    delete this.newMethod;
}
```

在这段代码中，为 `ClassA` 赋予了方法 `newMethod`（记住，函数名只是指向它的指针）。然后调用该方法，传递给它的是 `ClassB` 构造函数的参数 `sColor`。最后一行代码删除了对 `ClassA` 的引用，这样以后就不能再调用它。

所有的新属性和新方法都必须在删除了新方法的代码行后定义。否则，可能会覆盖超类的相关属性和方法：

```
function ClassB(sColor, sName) {
    this.newMethod = ClassA;
    this.newMethod(sColor);
    delete this.newMethod;

    this.name = sName;
    this.sayName = function () {
        alert(this.name);
    };
}
```

为证明前面的代码有效，可以运行下面的例子：

```

var objA = new ClassA("red");
var objB = new ClassB("blue", "Nicholas");
objA.sayColor();      //outputs "red"
objB.sayColor();      //outputs "blue"
objB.sayName();       //outputs "Nicholas"

```

有趣的是，对象冒充可以支持多重继承。也就是说，一个类可以继承多个超类。用 UML 表示的多继承机制如图 4-2 所示。

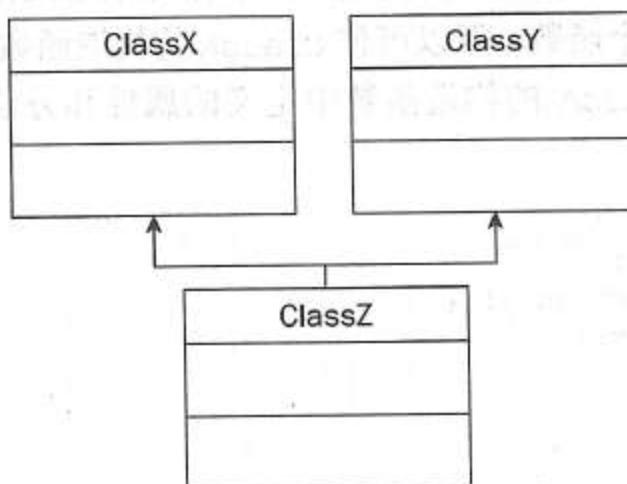


图 4-2

例如，如果存在两个类 ClassX 和 ClassY，ClassZ 想继承这两个类，可以使用下面的代码：

```

function ClassZ() {
    this.newMethod = ClassX;
    this.newMethod();
    delete this.newMethod;

    this.newMethod = ClassY;
    this.newMethod();
    delete this.newMethod;
}

```

106

这里存在一个弊端，如果 ClassX 和 ClassY 具有同名的属性或方法，ClassY 具有高优先级，因为它从后面的类继承。除这点小问题之外，用对象冒充实现多继承机制轻而易举。

由于这种继承方法的流行，ECMAScript 的第三版为 Function 对象加入了两个新方法，即 call() 和 apply()。

2. call() 方法

call() 方法是与经典的对象冒充方法最相似的方法。它的第一个参数用作 this 的对象。其他参数都直接传递给函数自身。例如：

```

function sayColor(sPrefix, sSuffix) {
    alert(sPrefix + this.color + sSuffix);
}

var obj = new Object();
obj.color = "red";

```

```
//outputs "The color is red, a very nice color indeed."
sayColor.call(obj, "The color is ", ", a very nice color indeed.");
```

在这个例子中，函数 sayColor() 在对象外定义，即使它不属于任何对象，也可以引用关键字 this。对象 obj 的 color 属性等于"red"。调用 call() 方法时，第一个参数是 obj，说明应该赋予 sayColor() 函数中的 this 关键字值是 obj。第二个和第三个参数是字符串。它们与 sayColor() 函数中的参数 prefix 和 suffix 匹配，最后生成的消息"The color is red, a very nice color indeed" 将被显示出来。

要与继承机制的对象冒充方法一起使用该方法，只需将前三行的赋值、调用和删除代码替换即可：

```
function ClassB(sColor, sName) {
    //this.newMethod = ClassA;
    //this.newMethod(sColor);
    //delete this.newMethod;
    ClassA.call(this, sColor);

    this.name = sName;
    this.sayName = function () {
        alert(this.name);
    };
}
```

这里，想让 ClassA 中的关键字 this 等于新创建的 ClassB 对象，因此 this 是第一个参数。第二个参数 sColor 对两个类来说都是唯一的参数。

107

3. apply()方法

apply() 方法有两个参数，用作 this 的对象和要传递给函数的参数的数组。例如：

```
function sayColor(sPrefix, sSuffix) {
    alert(sPrefix + this.color + sSuffix);
}

var obj = new Object();
obj.color = "red";

//outputs "The color is red, a very nice color indeed."
sayColor.apply(obj, new Array("The color is ", ", a very nice color indeed."));
```

这个例子与前面的例子相同，只是现在调用的是 apply() 方法。调用 apply() 方法时，第一个参数仍是 obj，说明应该赋予 sayColor() 中的 this 关键字值是 obj。第二个参数是由两个字符串构成的数组，与 sayColor() 的参数 prefix 和 suffix 匹配。生成的消息仍是"The color is red, a very nice color indeed"，将被显示出来。

该方法也用于替换前三行的赋值、调用和删除新方法的代码：

```
function ClassB(sColor, sName) {
    //this.newMethod = ClassA;
    //this.newMethod(sColor);
    //delete this.newMethod;
```

```

ClassA.apply(this, new Array(sColor));

this.name = sName;
this.sayName = function () {
    alert(this.name);
};

}

```

同样的，第一个参数仍是 this。第二个参数是只有一个值 color 的数组。可以把 ClassB 的整个 arguments 对象作为第二个参数传递给 apply() 方法：

```

function ClassB(sColor, sName) {
    //this.newMethod = ClassA;
    //this.newMethod(sColor);
    //delete this.newMethod;
    ClassA.apply(this, arguments);

    this.name = sName;
    this.sayName = function () {
        alert(this.name);
    };
}

```

108

当然，只有超类中的参数顺序与子类中的参数顺序完全一致时才可以传递参数对象。如果不是，就必须创建一个单独的数组，按照正确的顺序放置参数。此外，还可使用 call() 方法。

4. 原型链

继承这种形式在 ECMAScript 中原本是用于原型链的。上一章介绍了定义类的原型方式。原型链扩展了这种方式，以一种有趣的方式实现继承机制。

在上一章中学过，prototype 对象是个模板，要实例化的对象都以这个模板为基础。总而言之，prototype 对象的任何属性和方法都被传递给那个类的所有实例。原型链利用这种功能来实现继承机制。

如果用原型方式重定义前面例子中的类，它们将变为下列形式：

```

function ClassA() {}

ClassA.prototype.color = "red";
ClassA.prototype.sayColor = function () {
    alert(this.color);
};

function ClassB() {}

ClassB.prototype = new ClassA();

```

原型链的神奇之处在于突出显示的代码行。这里，把 ClassB 的 prototype 属性设置成 ClassA 的实例。这很有意义，因为想要 ClassA 的所有属性和方法，但又不想逐个将它们赋予 ClassB 的 prototype 属性。还有比把 ClassA 的实例赋予 prototype 属性更好的方法吗？

注意，调用 ClassA 的构造函数时，没有给它传递参数。这在原型链中是标准做法。要确保构造函数没有任何参数。

与对象冒充相似，子类的所有属性和方法都必须出现在 prototype 属性被赋值后，因为在它之前赋值的所有方法都会被删除。为什么？因为 prototype 属性被替换成了新对象，添加了新方法的原始对象将被销毁。所以，为 ClassB 类添加 name 属性和 sayName() 方法的代码如下：

```
function ClassB() {
}

ClassB.prototype = new ClassA();
ClassB.prototype.name = "";
ClassB.prototype.sayName = function () {
    alert(this.name);
};
```

109

可通过运行下面的例子测试这段代码：

```
var objA = new ClassA();
var objB = new ClassB();
objA.color = "red";
objB.color = "blue";
objB.name = "Nicholas";
objA.sayColor(); //outputs "red"
objB.sayColor(); //outputs "blue"
objB.sayName(); //outputs "Nicholas"
```

此外，在原型链中， instanceof 运算符的运行方式也很独特。对 ClassB 的所有实例， instanceof 为 ClassA 和 ClassB 都返回 true。例如：

```
var objB = new ClassB();
alert(objB instanceof ClassA); //outputs "true";
alert(objB instanceof ClassB); //outputs "true"
```

在 ECMAScript 的弱类型世界中，这是极其有用的工具，不过使用对象冒充时不能使用它。

原型链的弊端是不支持多重继承。记住，原型链会用另一类型的对象重写类的 prototype 属性。

5. 混合方式

这种继承方式使用构造函数定义类，并未使用任何原型。对象冒充的主要问题是必须使用构造函数方式（如上一章中学到的），这不是最好的选择。不过如果使用原型链，就无法使用带参构造函数了。开发者该如何选择呢？答案很简单，两者都用。

在前一章，你学过创建类的最好方式是用构造函数方式定义属性，用原型方式定义方法。这种方式同样适用于继承机制，用对象冒充继承构造函数的属性，用原型链继承 prototype 对象的方法。用这两种方式重写前面的例子，代码如下：

```

function ClassA(sColor) {
    this.color = sColor;
}

ClassA.prototype.sayColor = function () {
    alert(this.color);
};

110 function ClassB(sColor, sName) {
    ClassA.call(this, sColor);
    this.name = sName;
}

ClassB.prototype = new ClassA();

ClassB.prototype.sayName = function () {
    alert(this.name);
};

```

在此例子中，继承机制由两行突出显示的代码实现。在第一行突出显示的代码中，在 ClassB 构造函数中，用对象冒充继承 ClassA 类的 sColor 属性。在第二行突出显示的代码中，用原型链继承 ClassA 类的方法。由于这种混合方式使用了原型链，所以 instanceof 运算符仍能正确运行。

下面的例子测试了这段代码：

```

var objA = new ClassA("red");
var objB = new ClassB("blue", "Nicholas");
objA.sayColor();      //outputs "red"
objB.sayColor();      //outputs "blue"
objB.sayName();       //outputs "Nicholas"

```

4.2.2 一个更实际的例子

在真正的 Web 站点和应用程序中，几乎不可能创建名为 ClassA 和 ClassB 的类，更可能的是创建表示特定事物（如形状）的类。考虑本章开头所述的形状的例子，Polygon、Triangle 和 Rectangle 类就构成了一组很好的探讨数据。

1. 创建基类

首先考虑 Polygon 类。哪些属性和方法是必需的？首先，一定要知道多边形的边数，所以应该加入整数属性 sides。还有什么是多边形必需的？也许你想知道多边形的面积，那么加入计算面积的方法 getArea()。图 4-3 展示了该类的 UML 表示。

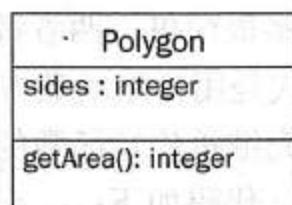


图 4-3

在 UML 中，属性由属性名和类型表示，位于紧接类名之下的单元中。方法位于属性之下，说明方法名和返回值的类型。

在 ECMAScript 中，可以如下编写类：

```
function Polygon(iSides) {
    this.sides = iSides;
}

Polygon.prototype.getArea = function () {
    return 0;
};
```

注意，`Polygon` 类不够详细精确，还不能使用，方法 `getArea()` 返回 0，因为它只是一个占位符，以便子类覆盖。

2. 创建子类

现在考虑创建 `Triangle` 类。三角形具有三条边，因此这个类必须覆盖 `Polygon` 类的 `sides` 属性，把它设置为 3。还要覆盖 `getArea()` 方法，使用三角形的面积公式，即 $1/2 \times \text{底} \times \text{高}$ 。但如何得到底和高的值呢？需要专门输入这两个值，所以必须创建 `base` 属性和 `height` 属性。`Triangle` 类的 UML 表示如图 4-4 所示。

该图只展示了 `Triangle` 类的新属性及覆盖过的方法。如果 `Triangle` 类没有覆盖 `getArea()` 方法，图中将不会列出它。它将被看作从 `Polygon` 类保留下来的方法。完整的 UML 图还展示了 `Polygon` 和 `Triangle` 类之间的关系（图 4-5），使它显得更清楚。

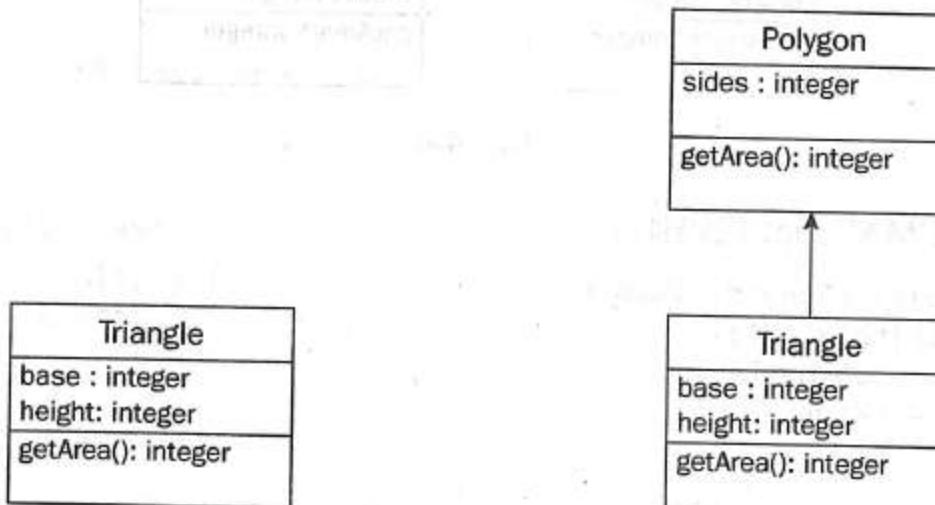


图 4-4

图 4-5

在 UML 中，决不会重复显示继承的属性和方法，除非该方法被覆盖（或被重载，这在 ECMAScript 中是不可能的）。

`Triangle` 类的代码如下：

```
function Triangle(iBase, iHeight) {
    Polygon.call(this, 3);
    this.base = iBase;
    this.height = iHeight;
}
```

```

Triangle.prototype = new Polygon();
Triangle.prototype.getArea = function () {
    return 0.5 * this.base * this.height;
};

```

注意，虽然 `Polygon` 的构造函数只接受一个参数 `sides`，`Triangle` 类的构造函数却接受两个参数，即 `base` 和 `height`。这是因为三角形的边数是已知的，且不想让开发者改变它。因此，使用对象冒充时，3 作为对象的边数被传给 `Polygon` 的构造函数。然后，把 `base` 和 `height` 的值赋予适当的属性。

在用原型链继承方法后，`Triangle` 将覆盖 `getArea()` 方法，提供为三角形面积定制的计算。

最后一个类是 `Rectangle`，它也继承 `Polygon`。矩形有四条边，面积是用长度×宽度计算的，长度和宽度即成为该类必需的属性。在前面的 UML 图中，要把 `Rectangle` 类填充在 `triangle` 类的旁边，因为它们的超类都是 `Polygon`（如图 4-6 所示）。

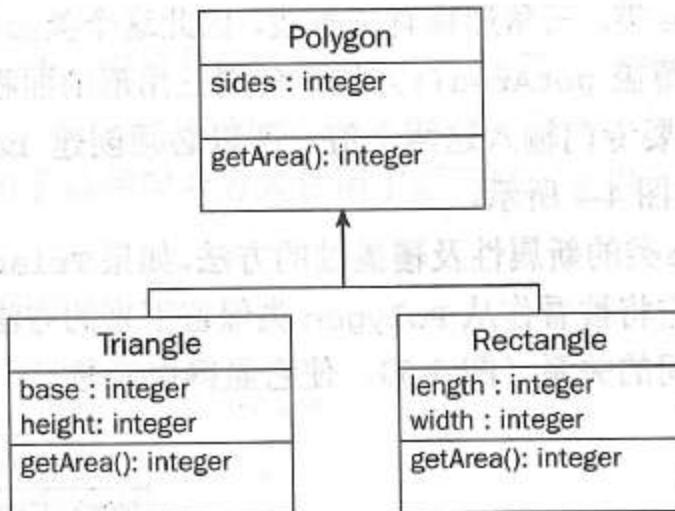


图 4-6

`Rectangle` 的 ECMAScript 代码如下：

```

function Rectangle(iLength, iWidth) {
    Polygon.call(this, 4);
    this.length = iLength;
    this.width = iWidth;
}

Rectangle.prototype = new Polygon();
Rectangle.prototype.getArea = function () {
    return this.length * this.width;
};

```

注意，`Rectangle` 构造函数不把 `sides` 作为参数，同样的，常量 4 被直接传给 `Polygon` 构造函数。与 `Triangle` 相似，`Rectangle` 引入了两个新的作为构造函数的参数的属性，然后覆盖 `getArea()` 方法。

3. 测试代码

可以运行下面代码来测试为该示例创建的代码：

```

var triangle = new Triangle(12, 4);
var rectangle = new Rectangle(22, 10);

alert(triangle.sides);           //outputs "3"
alert(triangle.getArea());       //outputs "24"

alert(rectangle.sides);          //outputs "4"
alert(rectangle.getArea());       //outputs "220"

```

这段代码创建一个三角形，底为 12，高为 4，还创建一个矩形，长为 22，宽为 10。然后输出每种形状的边数及面积，证明 `sides` 属性的赋值正确，`getArea()` 方法返回正确的值。三角形的面积应为 24，矩形的面积应该是 220。

4. 采用动态原型方法如何？

前面的例子用对象定义的混合构造函数/原型方式展示继承机制，那么可以使用动态原型来实现继承机制吗？不能。

继承机制不能采用动态化的原因是，`prototype` 对象的唯一性。看下面代码（这段代码不正确，却值得研究）：

```

function Polygon(iSides) {
    this.sides = iSides;

    if (typeof Polygon._initialized == "undefined") {

        Polygon.prototype.getArea = function () {
            return 0;
        };

        Polygon._initialized = true;
    }
}

function Triangle(iBase, iHeight) {
    Polygon.call(this, 3);
    this.base = iBase;
    this.height = iHeight;

    if (typeof Triangle._initialized == "undefined") {

        Triangle.prototype = new Polygon();
        Triangle.prototype.getArea = function () {
            return 0.5 * this.base * this.height;
        };

        Triangle._initialized = true;
    }
}

```

114

上面的代码展示了用动态原型定义的 `Polygon` 和 `Triangle` 类。错误在于突出显示的设置 `Triangle.prototype` 属性的代码。从逻辑上讲，这个位置是正确的，但从功能上讲，却是无效的。从技术上说来，在代码运行前，对象已被实例化，并与原始的 `prototype` 对象联系在一起。

了。虽然用极晚绑定可使对原型对象的修改正确地反映出来，但替换 `prototype` 对象却不会对该对象产生任何影响。只有未来的对象实例才会反映出这种改变，这就使第一个实例变得不正确。

要正确使用动态原型实现继承机制，必须在构造函数外赋予新的 `prototype` 对象，如下所示：

```
function Triangle(iBase, iHeight) {
    Polygon.call(this, 3);
    this.base = iBase;
    this.height = iHeight;

    if (typeof Triangle._initialized == "undefined") {

        Triangle.prototype.getArea = function () {
            return 0.5 * this.base * this.height;
        };

        Triangle._initialized = true;
    }
}

Triangle.prototype = new Polygon();
```

这段代码有效，因为是在任何对象实例化前给 `prototype` 对象赋值的。遗憾的是，这意味着不能把这段代码完整的封装在构造函数中了，而这正是动态原型的主旨。

4.3 其他继承方式

115 由于 ECMAScript 继承机制的限制（如缺少专有作用域，不能简单地访问超类的方法），世界各地的开发者都在为创建实现继承机制的其他方式而不懈地努力着。这一节讲解可以代替标准 ECMAScript 继承机制的继承方式。

4.3.1 zInherit

原型链实际上是把对象的所有方法复制给类的 `prototype` 对象。还有其他方法可以实现它吗？有，利用 `zInherit` 库（可以从 <http://www.nczonline.net/downloads> 处下载），不必使用原型链，也可实现方法继承。这个小库支持所有的现代浏览器（Mozilla、IE、Opera、Safari）及一些旧的浏览器（Netscape 4.x、IE、Mac）。

要使用 `zInherit` 库，必须用`<script>`标签加入 `zinherit.js`。第 5 章将详细讨论引入外部的 JavaScript 文件。

`zInherit` 库给 `Object` 类添加了两个方法，`inheritFrom()` 和 `instanceOf()`。如你所料，`inheritFrom()` 方法负担重任，负责复制指定类的所有方法。下面一行代码用原型链使 `ClassB` 继承 `ClassA` 的方法：

```
ClassB.prototype = new ClassA();
```

可用下面代码替换上面的代码：

```
ClassB.prototype.inheritFrom(ClassA);
```

`inheritFrom()`方法接受一个参数，即要复制的方法所属的类。注意，与原型链相对的是，这种方式并未真正创建要继承的类的实例，这样更安全，开发者也无需担心构造函数的参数。

为确保正确地实现继承，必须在原型赋值之处调用 `inheritFrom()` 方法。

`instanceOf()`方法是 `instanceof` 运算符的替代品。因为这种方式根本不使用原型链，所以这行代码无效：

```
ClassB instanceof ClassA
```

`instanceOf()`方法弥补了这项损失，与 `inheritFrom()`一起使用，可以跟踪所有的超类：

```
ClassB.instanceOf(ClassA);
```

1. 再探多边形

整个多边形的例子都可用 zInherit 库重写，只需要替换两行（突出显示）代码：

```
function Polygon(iSides) {
    this.sides = iSides;
}

Polygon.prototype.getArea = function () {
    return 0;
};

function Triangle(iBase, iHeight) {
    Polygon.call(this, 3);
    this.base = iBase;
    this.height = iHeight;
}

Triangle.prototype.inheritFrom(Polygon);

Triangle.prototype.getArea = function () {
    return 0.5 * this.base * this.height;
};

function Rectangle(iLength, iWidth) {
    Polygon.call(this, 4);
    this.length = iLength;
    this.width = iWidth;
}

Rectangle.prototype.inheritFrom(Polygon);

Rectangle.prototype.getArea = function () {
    return this.length * this.width;
};
```

可以用与前面相同的例子测试这段代码，只需添加两行代码，测试 `instanceOf()` 方法的输

出即可：

```
var triangle = new Triangle(12, 4);
var rectangle = new Rectangle(22, 10);

alert(triangle.sides);
alert(triangle.getArea());

alert(rectangle.sides);
alert(rectangle.getArea());

alert(triangle instanceof Triangle);      //outputs "true"
alert(triangle instanceof Polygon);        //outputs "true"

alert(rectangle instanceof Rectangle);     //outputs "true"
alert(rectangle instanceof Polygon);        //outputs "true"
```

117

最后四行代码用于测试 `instanceOf()` 方法，都应该返回 `true`。

2. 动态原型支持

如前所述，原型链不能真正满足动态原型主旨，即把类的所有代码放置在它的构造函数中。`zInherit` 库修正了这个问题，它允许在构造函数内部调用 `inheritFrom()` 方法。

让我们再看看前面的多边形动态原型的例子，现在加入 `zInherit` 库：

```
function Polygon(iSides) {
    this.sides = iSides;

    if (typeof Polygon._initialized == "undefined") {

        Polygon.prototype.getArea = function () {
            return 0;
        };

        Polygon._initialized = true;
    }
}

function Triangle(iBase, iHeight) {
    Polygon.call(this, 3);
    this.base = iBase;
    this.height = iHeight;

    if (typeof Triangle._initialized == "undefined") {

        Triangle.prototype.inheritFrom(Polygon);
        Triangle.prototype.getArea = function () {
            return 0.5 * this.base * this.height;
        };

        Triangle._initialized = true;
    }
}
```

```

function Rectangle(iLength, iWidth) {
    Polygon.call(this, 4);
    this.length = iLength;
    this.width = iWidth;

    if (typeof Rectangle._initialized == "undefined") {

        Rectangle.prototype.inheritFrom(Polygon);
        Rectangle.prototype.getArea = function () {
            return this.length * this.width;
        };

        Rectangle._initialized = true;
    }
}

```

118

前面的代码中突出显示的两行代码实现了 Triangle 类和 Rectangle 类对 Polygon 类的继承。这种方法成功的原因是，使用 inheritFrom()方法时，未重写 prototype 对象，只是为其加入方法而已。使用这种方法，即可避开原型链的限制，实现动态原型本意。

3. 多重继承支持

zInherit 库最有用的特性之一是支持多重继承，原型链不支持这种能力。同样，使这种支持成为可能的关键是 inheritFrom()方法不替换 prototype 对象。

要继承属性和方法，inheritFrom()方法必须与对象冒充一起使用。考虑下面的例子：

```

function ClassX() {
    this.messageX = "This is the X message. ";

    if (typeof ClassX._initialized == "undefined") {

        ClassX.prototype.sayMessageX = function () {
            alert(this.messageX);
        };

        ClassX._initialized = true;
    }
}

function ClassY() {
    this.messageY = "This is the Y message. ";

    if (typeof ClassY._initialized == "undefined") {

        ClassY.prototype.sayMessageY = function () {
            alert(this.messageY);
        };

        ClassY._initialized = true;
    }
}

```

ClassX 和 ClassY 类都是小类，都只有一个属性及一个方法。假设现有 ClassZ 类，需要继

承这两个类。可以如下定义该类：

```
function ClassZ() {
    ClassX.apply(this);
    ClassY.apply(this);
    this.messageZ = "This is the Z message. ";

    if (typeof ClassZ._initialized == "undefined") {

        ClassZ.prototype.inheritFrom(ClassX);
        ClassZ.prototype.inheritFrom(ClassY);

        ClassZ.prototype.sayMessageZ = function () {
            alert(this.messageZ);
        };

        ClassZ._initialized = true;
    }
}
```

119

注意继承属性的两行代码(使用 `apply()`方法)和继承方法的两行代码(使用 `inheritFrom()`方法)。如前所述，发生继承的顺序非常重要。通常按照继承属性的顺序继承方法比较好(意味着如果先继承 `ClassX` 的属性，然后继承 `ClassY` 的属性，那么也应该按照这种顺序继承它们的方法)。

下面的代码是测试多重继承的示例：

```
var objZ = new ClassZ();
objZ.sayMessageX(); //outputs "This is X message."
objZ.sayMessageY(); //outputs "This is Y message."
objZ.sayMessageZ(); //outputs "This is Z message."
```

前面的代码调用了三个方法：

(1) `sayMessageX()`方法是从 `ClassX` 继承而来，它访问 `messageX` 属性，该属性也是从 `ClassX` 继承而来。

(2) `sayMessageY()`方法是从 `ClassY` 继承而来，它访问 `messageY` 属性，该属性也是从 `ClassY` 继承而来。

(3) `sayMessageZ()`方法是在 `ClassZ` 中定义的，它访问 `messageZ` 属性，该属性也是在 `ClassZ` 定义的。

这三种方法应该输出各属性对应的消息，说明多重继承成功。

4.3.2 xbObjects

Netscape 的 DevEdge 站点 (<http://devedge.netscape.com>) 有许多对 Web 开发者有用的信息和脚本工具。工具之一是 `xbObjects` (可以从 <http://archive.bclary.com/xbProjects-docs/xbObject/> 处下载)，由 Netscape 公司的 Bob Clary 于 2001 年 Netscape 6 (Mozilla 0.6) 发布时编写而成。它支持从那时起的所有 Mozilla 版本及其他现代浏览器 (IE、Opera 和 Safari)。

1. 目的

xbObjects 的目的是为 JavaScript 提供更强的面向对象范型，不仅支持继承，还支持方法的重载和调用超类方法的能力。要实现这一点，xbObjects 需执行许多步。

第一步，必须注册类，此时，需定义它是由哪个类继承而来。用下面的调用可以实现这一点：

```
_classes.registerClass("Subclass_Name", "Superclass_Name");
```

120

这里，子类和超类名都以字符串形式传进来，而不是指向它们的构造函数的指针。这个调用必须放在指定子类的构造函数前。

如果新的类未继承任何类，调用 registerClass() 时也可以只用第一个参数。

第二步，在构造函数内调用 defineClass() 方法，传给它类名及被 Clary 称为原型函数 (prototype function) 的指针，该函数用于初始化对象的所有属性和方法（之后会介绍更多），例如：

```
_classes.registerClass("ClassA");

function ClassA(color) {
    _classes.defineClass("ClassA", prototypeFunction);

    function prototypeFunction() {
        //...
    }
}
```

可以看到，原型函数 (prototypeFunction()) 位于构造函数内部。它的主要用途是在适当的时候把所有方法赋予该类（在这一点上与动态原型相似）。

下一步（迄今为止是第三步）是为该类创建 init() 方法。该方法负责设置该类的所有属性，它必须接受与构造函数相同的参数。作为一种规约，init() 方法总是在 defineClass() 方法后调用。例如：

```
_classes.registerClass("ClassA");

function ClassA(sColor) {
    _classes.defineClass("ClassA", prototypeFunction);

    this.init(sColor);

    function prototypeFunction() {

        ClassA.prototype.init = function (sColor) {
            this.parentMethod("init");
            this.color = sColor;
        };
    }
}
```

你可能已注意到 init() 方法中调用的 parentMethod() 方法。xbObjects 以这种方式允许类调用它的超类的方法。parentMethod() 方法接受任意多个参数，但第一个参数总是要调用的父

[121] 类方法的名字（该参数必须是字符串，而不是函数指针），所有其他参数都被传给父类的方法。

在这个例子中，首先调用 `init()` 方法，这是 `xbObjects` 运行所必需的。即使 `ClassA` 未注册超类，`xbObjects` 都会为它创建一个所有类的默认超类，即超类方法 `init()` 所属的类。

第四步也是最后一步，在原型函数内添加其他类的方法：

```
_classes.registerClass("ClassA");

function ClassA(sColor) {
    _classes.defineClass("ClassA", prototypeFunction);

    this.init(sColor);

    function prototypeFunction() {

        ClassA.prototype.init = function (sColor) {
            this.parentMethod("init");
            this.color = sColor;
        };

        ClassA.prototype.sayColor = function () {
            alert(this.color);
        };
    }
}
```

然后，即可以以常规方式创建 `ClassA` 的实例：

```
var objA = new ClassA("red");
objA.sayColor(); //outputs "red"
```

2. 重载多边形

此时，你一定想知道是否可以用 `xbObjects` 重写多边形的例子，下面就是重写后的代码。

首先，重写 `Polygon` 类，非常简单：

```
_classes.registerClass("Polygon");

function Polygon(sides) {

    _classes.defineClass("Polygon", prototypeFunction);

    this.init(sides);

    function prototypeFunction() {

        Polygon.prototype.init = function(iSides) {
            this.parentMethod("init");
            this.sides = iSides;
        };

        Polygon.prototype.getArea = function () {
            return 0;
        };
    }
}
```

[122]

```

    }
}

```

接下来，重写 Triangle 类，是这个例子中第一个真正用到继承的类：

```

_classes.registerClass("Triangle", "Polygon");

function Triangle(iBase, iHeight) {
    _classes.defineClass("Triangle", prototypeFunction);

    this.init(iBase, iHeight);

    function prototypeFunction() {
        Triangle.prototype.init = function(iBase, iHeight) {
            this.parentMethod("init", 3);
            this.base = iBase;
            this.height = iHeight;
        };

        Triangle.prototype.getArea = function () {
            return 0.5 * this.base * this.height;
        };
    }
}

```

注意在构造函数之前调用 registerClass()，并建立继承关系。此外，init()方法的第一行调用超类的 init()方法，参数是 3，把 sides 属性设置成 3。余下的就是为 base 和 height 属性赋值。

Rectangle 类与 Triangle 类类似：

```

_classes.registerClass("Rectangle", "Polygon");

function Rectangle(iLength, iWidth) {
    _classes.defineClass("Rectangle", prototypeFunction);

    this.init(iLength, iWidth);

    function prototypeFunction() {
        Rectangle.prototype.init = function(iLength, iWidth) {
            this.parentMethod("init", 4);
            this.length = iLength;
            this.width = iWidth;
        };

        Rectangle.prototype.getArea = function () {
            return this.length * this.width;
        };
    }
}

```

该类与 Triangle 类之间的主要区别（除 registerClass() 和 defineClass() 类的调用外）是调用超类方法 init() 的参数是 4。然后，添加 length 属性和 width 属性，覆盖 getArea() 方法。

4.4 小结

本章介绍了 ECMAScript（从而也是 JavaScript）中用对象冒充和原型链实现的继承概念。学会结合使用这些方式才是建立类之间的继承机制的最好方式。

最后，还介绍了其他两种建立继承机制的方式，即 zInherit 和 xbObjects。这些 JavaScript 库都可以从因特网上免费得到，它们为对象继承引入了新的不同的能力。

对 JavaScript 的核心 ECMAScript 的讨论到此为止。接下来的章节将在这个基础上介绍更多

在前面几章中，学习了 JavaScript 的核心 ECMAScript 以及该语言工作方式的基础知识。从本章开始，重点将转移到如何在 Web 浏览器中使用 JavaScript。

自 Netscape Navigator 2.0 初次引入 JavaScript 以来，Web 浏览器已经有了长足的发展。今天的浏览器不再只能处理传统的 HTML 文件，它们能处理各种格式的文件。具有讽刺意味的是，这些文件中的大多数都采用 JavaScript 作为动态改变客户端内容的方式。这一章探讨如何把 JavaScript 嵌入 HTML 及其他语言，并介绍了 BOM（浏览器对象模型）的一些基本概念。

5.1 HTML 中的 JavaScript

当然，第一个利用嵌入式 JavaScript 的语言还是 HTML，因此首先讨论的自然是如何在 HTML 中使用 JavaScript。HTML 中嵌入 JavaScript 是从引入用于 JavaScript 的标签和为 HTML 的一些通用部分增加了新特性开始的。

5.1.1 <script/> 标签

HTML 页面中包含 JavaScript 使用<script/>标签实现的。该标签通常放置在页面的<head/>标签中，最初定义的<script/>标签具有一个或两个特性，language 特性声明要使用的脚本语言，src 特性是可选的，声明要加入页面的外部 JavaScript 文件。language 特性一般被设置为 JavaScript，不过也可用它声明 JavaScript 的确切版本，如 JavaScript 1.3（如果省略 language 特性，浏览器默认使用最新的 JavaScript 版本）。

125

尽管<script/>最初是为 JavaScript 设计的，但可以用于声明任意多种不同的客户端脚本语言，language 特性用于声明使用的代码的类型。例如，可把 language 特性设置为 VBScript，使用 IE 的 VBScript 语言（只适用于 Windows）。

如果未声明 src 特性，在<script/>中即可以任意形式编写 JavaScript 代码。如声明了 src 特性，那么<script/>中的代码可能就是无效的（由浏览器决定）。例如：

```

<html>
  <head>
    <title>Title of Page</title>
    <script language="JavaScript">
      var i = 0;
    </script>
    <script language="JavaScript" src="../scripts/external.js"></script>
  </head>
  <body>
    <!-- body goes here -->
  </body>
</html>

```

这个例子中既有内嵌的 JavaScript 代码，又有对外部 JavaScript 文件的链接。使用 `src` 特性，即可像引用图像和样式表一样引用 JavaScript 文件。

虽然大多数浏览器并未要求，但根据规约，外部 JavaScript 文件的扩展名应为 `.js`（这样可以使用 JSP、PHP 或其他服务器端的脚本语言动态生成 JavaScript 代码）。

5.1.2 外部文件格式

外部 JavaScript 语言的格式非常简单。事实上，它们只包含 JavaScript 代码的纯文本文件。在外部文件中不需要 `<script/>` 标签，引用文件的 `<script/>` 标签出现在 HTML 页中。这使得外部 JavaScript 文件看起来很像其他程序设计语言的源代码文件。

例如，考虑下面的内嵌代码：

```

<html>
  <head>
    <title>Title of Page</title>
    <script language="JavaScript">
      function sayHi() {
        alert("Hi");
      }
    </script>
  </head>
  <body>
    <!-- body goes here -->
  </body>
</html>

```

126

要把函数 `sayHi()` 放在外部文件 `external.js` 中，需要复制函数文本自身（如图 5-1 所示）。

external.js

```

function sayHi() {
  alert("Hi");
}

```

图 5-1

然后可更新 HTML 代码，加入这个外部文件：

```
<html>
  <head>
    <title>Title of Page</title>
    <script language="JavaScript" src="external.js"></script>
  </head>
  <body>
    <!-- body goes here -->
  </body>
</html>
```

对于 JavaScript 源文件中可加入哪些代码并无规定，这意味着可以给 JavaScript 文件加入任意多个类定义、函数，等等。

5.1.3 内嵌代码和外部文件

何时应该采用内嵌代码，何时采用外部文件呢？虽然关于这一点并无确定而且简洁的规则可循，不过一般认为，大量的 JavaScript 代码不应内嵌在 HTML 文件中，原因如下：

- **安全性**——只要查看页面的源代码，任何人都可确切地知道其中的代码做了什么。如果怀有恶意的开发者查看了源代码，就可能发现安全漏洞，危及整个站点或应用程序的安全。此外，在外部文件中还可加入版权和其他知识产权通告，而不打断页面流。
- **代码维护**——如果 JavaScript 代码散布于多个页面，那么代码维护将变成一场恶梦。把所有 JavaScript 文件放在一个目录中要容易得多，这样在发生 JavaScript 错误时，就不会对放置代码的位置有任何疑问。
- **缓存**——浏览器会根据特定的设置缓存所有外部链接的 JavaScript 文件，这意味着如果两个页面使用同一个文件，只需要下载该文件一次。这将加快下载速度。把同一段代码放在多个页面中，不只浪费，还增加了页面大小，从而增加下载时间。

127

5.1.4 标签放置

一般说来，所有代码和函数的定义都在 HTML 页的<head/>标签中，这样在显示页面主体后，代码就被完全装载进浏览器，可供使用了。唯一该出现在<body/>标签中的是调用前面定义的函数的代码。

<script/>放在<body/>内时，只要脚本所属的那部分页面被载入浏览器，脚本就会被执行。这样在载入整个页面之前，也可执行 JavaScript 代码。例如：

```
<html>
  <head>
    <title>Title of Page</title>
    <script language="JavaScript">
      function sayHi() {
        alert("Hi");
      }
    </script>
  </head>
  <body>
    <!-- body goes here -->
  </body>
</html>
```

```

        }
    </script>
</head>
<body>
    <script language="JavaScript">
        sayHi();
    </script>
    <p>This is the first text the user will see.</p>
</body>
</html>

```

在这段代码中，方法 `sayHi()` 在页面显示所有文本前调用，这意味着警告消息将在文本 "This is the first text the user will see." 显示前弹出。建议不采用这种在页面的 `<body>` 标签内调用 JavaScript 函数的方法，应该尽量避免它。相反的，建议在页面主体中只使用事件处理函数（event handler），例如：

```

<html>
    <head>
        <title>Title of Page</title>
        <script language="JavaScript">
            function sayHi() {
                alert("Hi");
            }
        </script>
    </head>
    <body>
        <input type="button" value="Call Function" onclick="sayHi()" />
    </body>
</html>

```

128

这里，使用 `<input/>` 标签创建一个按钮，点击它时调用 `sayHi()` 方法。`onclick` 特性声明一个事件处理函数，即响应特定事件的代码。第 9 章将详细讨论事件和事件处理函数。

注意，开始载入页面时，JavaScript 就开始运行了，因此有可能调用尚未存在的函数。在前面的例子中，把原来的 `<script/>` 标签放在函数调用后就会引发错误：

```

<html>
    <head>
        <title>Title of Page</title>
    </head>
    <body>
        <script language="JavaScript">
            sayHi();
        </script>
        <p>This is the first text the user will see.</p>
        <script language="JavaScript">
            function sayHi() {
                alert("Hi");
            }
        </script>
    </body>
</html>

```

这个例子将引发错误，因为在定义 sayHi() 之前就调用了它。由于 JavaScript 是从上到下载入的，所以在遇到第二个<script/>标签前，函数 sayHi() 还不存在。注意这种问题，此外，如前所述，使用事件和事件处理函数调用 JavaScript 函数。

5.1.5 隐藏还是不隐藏

初次引入 JavaScript 时，只有一种浏览器支持它，因此大家开始关心，不支持 JavaScript 的浏览器如何处理<script/>标签及其中包含的代码。最后，设计了一种用于对旧的浏览器隐藏 JavaScript 代码（这是一个短语，在当今因特网上的许多 Web 站点的源代码中都能找到它）的格式。下面的代码在内嵌代码周围加入 HTML 注释，这样其他浏览器就不会在屏幕上显示这段代码。

```
<script language="JavaScript"><!-- hide from older browsers
    function sayHi() {
        alert("Hi");
    }
//-->
</script>
```

第一行紧接起始标签<script>开始一条 HTML 注释。这样做是有效的，因为浏览器仍然把该行余下的部分看成 HTML 的一部分，JavaScript 代码从下一行开始。接下来的是常规的函数定义。第 2 行到倒数第 2 行是最有趣的部分，因为它以单行 JavaScript 注释标记（两个前斜线）开始，后面是 HTML 注释的结尾标记（-->）。这一行仍被看作 JavaScript 代码，所以单行注释标记是避免语法错所必需的。不过，旧的浏览器只承认 HMLT 注释的结束标记，因此，将忽略所有 JavaScript 代码。但是，支持 JavaScript 的浏览器只忽略该行，继续执行</script>标签。

129

尽管这种隐藏代码的方法在 Web 早期非常流行，今天却不再是必需的。目前，大多数 Web 浏览器都支持 JavaScript，而不支持 JavaScript 的浏览器通常足够聪明，自己就能够忽略 JavaScript 代码。

5.1.6 <noscript/>标签

不支持 JavaScript 的浏览器另外令人关注的是如何提供替代的内容。隐藏代码只是解决方法的一部分，开发者还需要一种方法，声明在 JavaScript 不能用时应该显示的内容。解决方法是采用<noscript/>标签，它可包含任何 HTML 代码（除<script/>）。支持或启用 JavaScript 的浏览器会忽略这些 HTML 代码，不支持或者禁用 JavaScript 的浏览器则显示<noscript/>的内容。例如：

```
<html>
    <head>
        <title>Title of Page</title>
        <script language="JavaScript">
            function sayHi() {
                alert("Hi");
            }
        </script>
```

```

</head>
<body>
    <script language="JavaScript">
        sayHi();
    </script>
    <noscript>
        <p>Your browser doesn't support JavaScript. If it did support
JavaScript, you would see this message: Hi!</p>
    </noscript>
        <p>This is the first text the user will see if JavaScript is enabled. If
JavaScript is disabled this is the second text the user will see.</p>
    </body>
</html>

```

在这个例子中，`<noscript>`标签中有一条消息，告诉用户浏览器不支持 JavaScript。第 8

130

章解释`<noscript>`的实际用法。

5.1.7 XHTML 中的改变

近来，随着 XHTML（可扩展 HTML）标准的出现，`<script>`标签也经历了一些改变。该标签不再用 `language` 特性，而用 `type` 特性声明内嵌代码或要加入的外部文件的 mime 类型，JavaScript 的 mime 类型是“text/javascript”。例如：

```

<html>
    <head>
        <title>Title of Page</title>
        <script type="text/javascript">
            var i = 0;
        </script>
        <script type="text/javascript" src="../scripts/external.js"></script>
    </head>
    <body>
        <!-- body goes here -->
    </body>
</html>

```

即使许多浏览器不完全支持 XHTML，但大多数开发者现在都用 `type` 特性，而不用 `language` 特性，以提供更好的 XHTML 支持。省略 `language` 特性不会带来任何问题，因为如前所述，所有浏览器都默认`<script>`的该属性值为 JavaScript。

XHTML 的第二个改变是使用 CDATA 段。XML 中的 CDATA 段用于声明不应被解析为标签的文本（XHTML 也是如此），这样就可以使用特殊字符，如小于（<）、大于（>）、和号（&）和双引号（"），而不必使用它们的字符实体。考虑下面的代码：

```

<script type="text/javascript">
    function compare(a, b) {
        if (a < b) {
            alert("A is less than B");
        } else if (a > b) {
            alert("A is greater than B");
        } else {

```

```

        alert("A is equal to B");
    }
}
</script>

```

这个函数相当简单，它比较数字 a 和 b，然后显示消息说明它们的关系。但是，在 XHTML 中，这段代码是无效的，因为它使用了三个特殊符号，即小于、大于和双引号。要修正这个问题，必须分别用这三个字符的 XML 实体 <、> 和 " 替换它们：

```

<script type="text/javascript">
    function compare(a, b) {
        if (a &lt; b) {
            alert(&quot;A is less than B&quot;);
        } else if (a &gt; b) {
            alert(&quot;A is greater than B&quot;);
        } else {
            alert(&quot;A is equal to B&quot;);
        }
    }
</script>

```

131

这段代码存在两个问题。首先，开发者不习惯用 XML 实体编写代码。这使代码很难读懂。其次，在 JavaScript 中，这种代码实际上将视为有语法错，因为解释程序不知道 XML 实体的意思。用 CDATA 段即可以以常规形式（即易读的语法）编写 JavaScript 代码。正式加入 CDATA 段的方法如下：

```

<script type="text/javascript"><![CDATA[
    function compare(a, b) {
        if (a < b) {
            alert("A is less than B");
        } else if (a > b) {
            alert("A is greater than B");
        } else {
            alert("A is equal to B");
        }
    }
]]></script>

```

虽然这是正式方式，但还要记住，大多数浏览器都不完全支持 XHTML，这就带来主要问题，即这在 JavaScript 中是个语法错误，因为大多数浏览器还不认识 CDATA 段。

当前使用的解决方案模仿了“对旧浏览器隐藏”代码的方法。使用单行的 JavaScript 注释，可在不影响代码语法的情况下嵌入 CDATA 段：

```

<script type="text/javascript">
//<![CDATA[
    function compare(a, b) {
        if (a < b) {
            alert("A is less than B");
        } else if (a > b) {
            alert("A is greater than B");
        } else {
            alert("A is equal to B");
        }
    }
]]></script>

```

```

        }
    //}
//]]>
</script>

```

现在，这段代码在不支持 XHTML 的浏览器中也可运行。

132 与 `type` 特性一样，随着开发者为浏览器中的 XHTML 准备更好的支持，CDATA 的这种用法也越来越流行。但是，为避免 CDATA 的问题，最好还是用外部文件引入 JavaScript 代码。

5.2 SVG 中的 JavaScript

SVG 是一种崭露头角的基于 XML 的语言，用于在 Web 上绘制矢量图形。矢量图形不同于光栅图形（位图），它们定义的是三角形、线段及它们之间的关系，而不只是定义图像的每个像素的颜色。这样生成的图像无论大小，看起来都是相同的。随着这种语言的日益流行，矢量图形程序（如 Adobe Illustrator）已经开始加入 SVG 导出功能。

尽管目前没有浏览器内置支持 SVG（虽然 Mozilla 2.0 将支持它），但许多公司（包括著名的 Adobe 和 Corel）都在编写 SVG 插件，以便使大多数浏览器都能显示 SVG 图形。

5.2.1 SVG 基础

介绍 SVG 这种语言不在本书的讨论范围内。不过，对该语言稍有理解有助于 JavaScript 的讨论。

下面是一个简单的 SVG 示例：

```

<?xml version="1.0"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.0//EN"
"http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd">
<svg xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http://www.w3.org/1999/xlink"
width="100%" height="100%">
    <desc>
        An image of a square and a circle.
    </desc>
    <defs>
        <rect id="rect1" width="200" height="200" fill="red" x="10" y="10"
stroke="black"/>
        <circle id="circle1" r="100" fill="white" stroke="black" cx="200"
cy="200"/>
    </defs>
    <g>
        <use xlink:href="#rect1" />
        <use xlink:href="#circle1" />
    </g>
</svg>

```

这个例子将在正方形的右下角画一个圆（如图 5-2 所示）。

注意，SVG 文件的开头是 XML 序言（prolog）`<?xml version="1.0"?>`，声明该语言是基于 XML 的。接下来的 SVG DTD 是可有可无的，不过通常都会被加入。

最外层的标签是`<svg>`，把该文件定义为SVG图像。特性`width`和`height`可被设置为任何值，包括百分数和像素，不过这里为简单起见，把它们设置为100%。注意声明了两个XML命名空间，一个用于SVG，一个用于XLink。XLink定义`href`这样的链接行为，将来的 XHTML版本很可能支持它。目前，SVG带动了对XLink的基本支持。

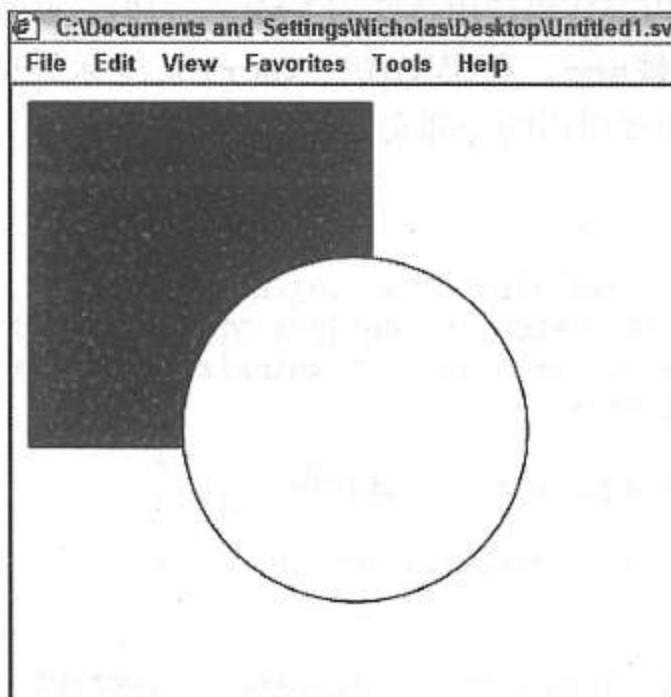


图 5-2

接下来的标签是`<desc>`，包含图像的说明。可把`<desc>`标签看作HTML中的`<title>`标签，它说明了图像中包含的内容，而不是如何在页面上显示图像。紧接着是`<defs>`标签，定义图像中要使用的资源及形状。在这个例子中，定义了一个矩形和一个圆。除非专门用在真正的图像中，否则这些形状不会被显示。

`<defs>`之后是`<g>`标签，即group的缩写。标签`<g>`很特殊，因为它是最外层的，从而封装了该可视图像。在最外层的`<g>`标签内可多次使用`<g>`标签（就像HTML中的`<div>`），以构成形状组。

在这个例子中，有两个`<use>`标签指向`<defs>`段中的形状。`<use>`标签的`xlink:href`特性指向形状的ID（前面有英镑符号#），从而把这个形状带到该可视图像中。`<defs>`中定义的形状可用多个`<use>`标签在图像中多次使用。SVG的这种能力使它成为基于XML的语言中代码重用的典范。

当然，SVG最令人兴奋的一点是对JavaScript的优秀支持，用JavaScript可以对SVG图像的各个部分进行操作。

5.2.2 SVG 中的`<script>`标签

SVG采用的`<script>`标签与将JavaScript加入页面的`<script>`标签相似。但这个`<script>`标签不同于HTML中的`<script>`标签：

□ **type** 特性是必需的。**type** 特性可被设置为 `text/javascript` 或 `text/ecmascript`，不过从技术上讲，前者才是正确的。

134

□ **language** 特性是不合法的。加入这个特性，SVG 代码就会无效。

□ 内嵌代码必须使用 CDATA 段。由于 SVG 是真正基于 XML 的语言，所以它能正确地支持 CDATA 段，因此内嵌代码使用特殊的 XML 字符时，必须使用 CDATA 段。

□ 使用 **xlink:href** 代替 **src**。在 SVG 中，`<script/>` 标签中没有 **src** 特性，而是使用 **xlink:href** 特性说明要引用的外部文件。

例如：

```
<?xml version="1.0"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.0//EN"
"http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd">
<svg xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http://www.w3.org/1999/xlink"
width="100%" height="100%">
  <desc>
    An image of a square and a circle.
  </desc>
  <script type="text/javascript"><![CDATA[
    var i = 0;
  ]]></script>
  <script type="text/javascript" xlink:href="../scripts/external.js"></script>
  <defs>
    <rect id="rect1" width="200" height="200" fill="red" x="10" y="10"
stroke="black"/>
    <circle id="circle1" r="100" fill="white" stroke="black" cx="200"
cy="200"/>
  </defs>
  <g>
    <use xlink:href="#rect1" />
    <use xlink:href="#circle1" />
  </g>
</svg>
```

这段代码中，SVG 有两个正确的`<script/>`标签。第一个包含内嵌代码，被 CDATA 段包围着，因此使用特殊字符不会产生任何问题。第二个使用 **xlink:href** 特性引用外部文件。

5.2.3 SVG 中的标签放置

由于 SVG 中不存在`<head/>`区，所以`<script/>`标签几乎可以放在任何位置。但是，通常它们放在以下地方：

- 紧接在`<desc/>`标签后；
- 在`<defs/>`标签中；
- 恰好位于最外层的`<g/>`标签前。

`<script/>`标签不能放在形状内部，如`<rect/>`或`<circle/>`，也不能放在滤光器、梯度或

135

其他定义外观的标签内。

5.3 BOM

讨论浏览器中的 JavaScript，不能不讨论 BOM（浏览器对象模型），它提供了独立于内容而与浏览器窗口进行交互的对象。

BOM 由一系列相关的对象构成。图 5-3 展示了基本的 BOM 体系结构。

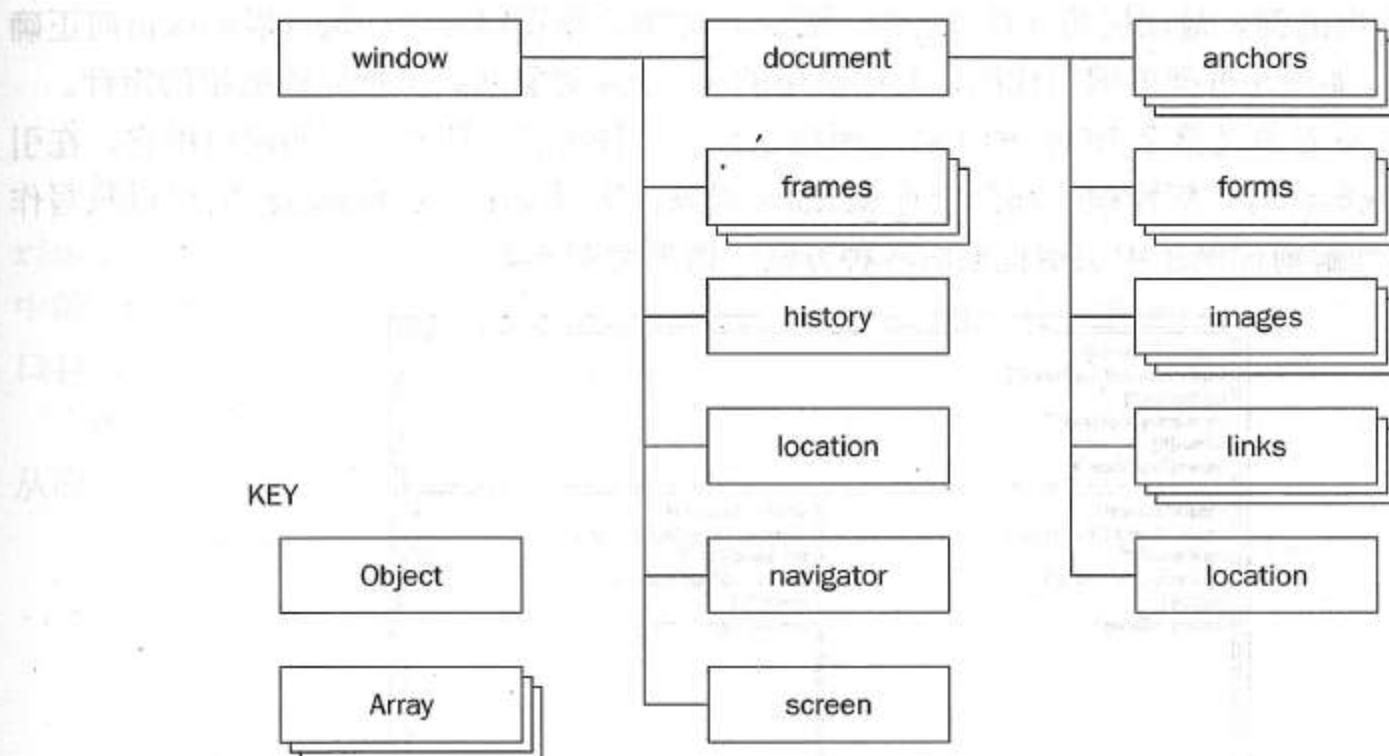


图 5-3

可以看到，window 对象是整个 BOM 的核心，所有对象和集合都以某种方式回接到 window 对象。我从这个对象着手讨论 BOM。

5.3.1 window 对象

window 对象表示整个浏览器窗口，但不必表示其中包含的内容。此外，window 还可用于移动或调整它表示的浏览器的大小，或者对它产生其他影响。

如果页面使用框架集合，每个框架都由它自己的 window 对象表示，存放在 frames 集合中。在 frames 集合中，可用数字（由 0 开始，从左到右，逐行的）或名字对框架进行索引。考虑下面的例子：

```

<html>
  <head>
    <title>Frameset Example</title>
  </head>
  <frameset rows="100,*">
    <frame src="frame.htm" name="topFrame" />
    <frameset cols="50%,50%">
      <frame src="anotherframe.htm" name="leftFrame" />
      <frame src="yetanotherframe.htm" name="rightFrame" />
    </frameset>
  </frameset>

```

```
</frameset>
</html>
```

这段代码创建一个框架集，其中包括一个顶层框架和两个底层框架。这里，可以用 `window.frames[0]` 或 `window.frames["topFrame"]` 引用框架，也可以用 `top` 对象代替 `window` 对象引用这些框架（例如 `top.frames[0]`）。

`top` 对象指向的都是最顶层的（最外层的）框架，即浏览器窗口自身。这可以确保指向正确的框架。此后，如果在框架内编写代码，其中引用的 `window` 对象就只是指向该框架的指针。

由于 `window` 对象是整个 BOM 的中心，所以它享有一种特权，即不需要明确引用它。在引用函数、对象或集合时，解释程序都会查看 `window` 对象，所以 `window.frames[0]` 可以只写作 `frame[0]`。要理解前面例子中引用框架的各种方法，请参考图 5-4。

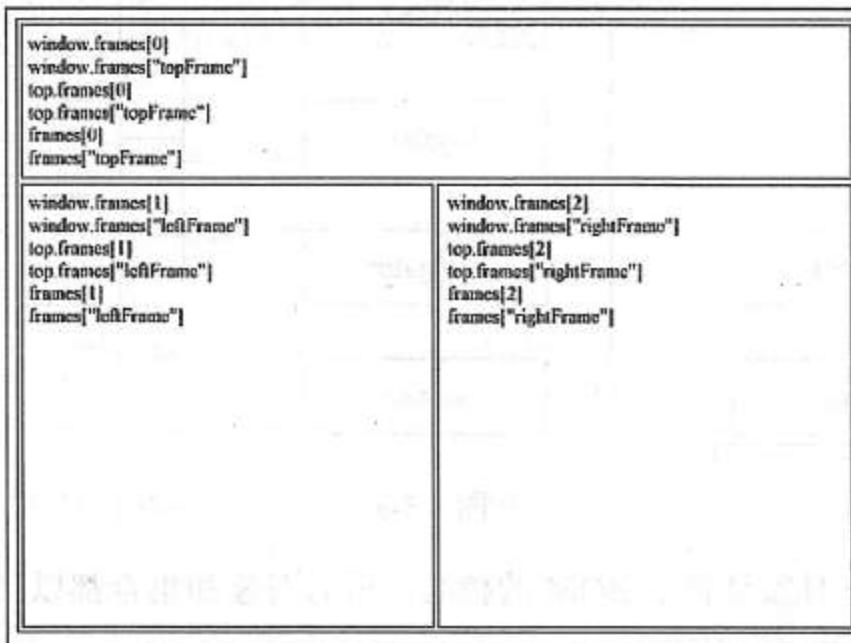


图 5-4

也可以直接用框架的名字访问它，如 `window.leftFrame`。不过 `frames` 集合更常用，因为它能确切地表示出代码的意图。

137 window 另一个实例是 `parent`。`parent` 对象与装载文件的框架集一起使用，要装载的文件也是框架集。假设名为 `frameset1.htm` 的文件包含下面的代码：

```
<html>
  <head>
    <title>Frameset Example</title>
  </head>
  <frameset rows="100,*">
    <frame src="frame.htm" name="topFrame" />
    <frameset cols="50%,50%">
      <frame src="anotherframe.htm" name="leftFrame" />
      <frame src="anotherframeset.htm" name="rightFrame" />
    </frameset>
  </frameset>
</html>
```

如果是名为 anotherframeset.htm 的文件包含这段代码呢？

```
<html>
  <head>
    <title>Frameset Example</title>
  </head>
  <frameset cols="100,*">
    <frame src="red.htm" name="redFrame" />
    <frame src="blue.htm" name="blueFrame" />
  </frameset>
</html>
```

第一个文件 frameset1.htm 被载入浏览器时，它将把 anotherframeset.htm 载入到 rightFrame。如果代码写在 redFrame（或 blueFrame）中，parent 对象就指向 frameset1.htm 中的 rightFrame。但如果代码写在 topFrame 中，parent 对象就指向 top 对象，因为浏览器窗口自身被看作所有顶层框架的父框架。

图 5-5 访问 window 对象的 name 属性，它存储的是框架的名字（不过一定是空白或 top），从而证明了这一点。

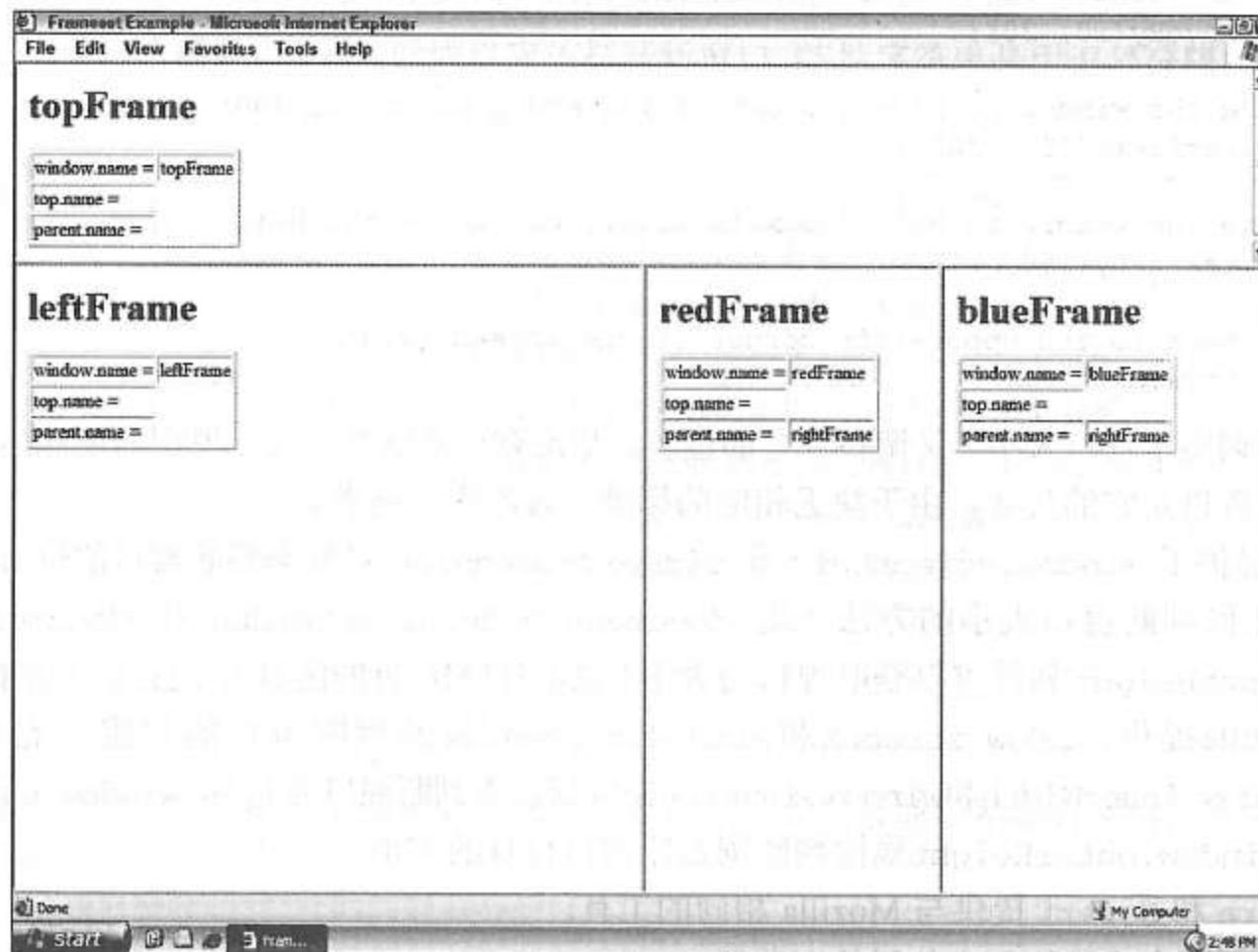


图 5-5

一个更加全局化的窗口指针是 self，它总是等于 window（是的，有点多余，加入它是因为它比 parent 更合适。它澄清了正在使用的不是框架的父框架，而是它自身）。

如果页面上没有框架，window 和 self 就等于 top，frames 集合的长度为 0。

138 还可以连锁引用 window，例如 `parent.parent.frames["topFrame"]`，不过通常不赞成使用这种形式，因为框架结构的任何改变都会造成代码错误。

1. 窗口操作

如前所述，`window` 对象对操作浏览器窗口（和框架）非常有用。这意味着，开发者可以移动或调整浏览器窗口的大小。可用四种方法实现这些操作：

- `moveBy(dx, dy)`——把浏览器窗口相对当前位置水平移动 `dx` 个像素，垂直移动 `dy` 个像素。`dx` 值为负数，向左移动窗口，`dy` 值为负数，向上移动窗口。
- `moveTo(x, y)`——移动浏览器窗口，使它的左上角位于用户屏幕的 `(x, y)` 处。可以使用负数，不过这样会把部分窗口移出屏幕的可视区域。
- `resizeBy(dw, dh)`——相对于浏览器窗口的当前大小，把它口的宽度调整 `dw` 个像素，高度调整 `dh` 个像素。`dw` 为负数，把缩小窗口的宽度，`dy` 为负数，缩小窗口的高度。
- `resizeTo(w, h)`——把窗口的宽度调整为 `w`，高度调整为 `h`。不能使用负数。

139 例如：

```
//move the window right by 10 pixels and down by 20 pixels
window.moveBy(10, 20);

//resize the window to have a width of 150 and a height of 300
window.resizeTo(150, 300);

//resize the window to be 150 pixels wider, but leave the height alone
window.resizeBy(150, 0);

//move back to the upper-left corner of the screen (0,0)
window.moveTo(0, 0);
```

假设既调整了窗口大小，又调整了它的位置，却没有记录这些改变。现在需要知道该窗口在屏幕上的位置以及它的尺寸。由于缺乏相应的标准，就产生了问题。

- IE 提供了 `window.screenLeft` 和 `window.screenTop` 对象来判断窗口的位置，但未提供任何判断窗口大小的方法。用 `document.body.offsetWidth` 和 `document.body.offsetHeight` 属性可以获取视口的大小（显示 HTML 页的区域），但它们不是标准属性。
- Mozilla 提供 `window.screenX` 和 `window.screenY` 属性判断窗口的位置。它还提供了 `window.innerWidth` 和 `window.innerHeight` 属性来判断视口的大小，`window.outerWidth` 和 `window.outerHeight` 属性判断浏览器窗口自身的大小。
- Opera 和 Safari 提供与 Mozilla 相同的工具。

所以，问题变成了了解用户使用的浏览器。

尽管移动浏览器窗口和调整它的大小是种很酷的操作，但应该尽量少用它们。移动浏览器窗口和调整它的大小会对用户产生影响，因此专业的 Web 站点和 Web 应用程序都避免使用它们。

2. 导航和打开新窗口

用 JavaScript 可以导航到指定的 URL，并用 `window.open()` 方法打开新窗口。该方法接受四个参数，即要载入新窗口的页面的 URL、新窗口的名字（为目标所用）、特性字符串和说明是否用新载入的页面替换当前载入的页面的 Boolean 值。一般只用前三个参数，因为最后一个参数只有在调用 `window.open()` 方法却不打开新窗口时才有效。

如果用已有框架的名字作为 `window.open()` 方法的第二个参数调用它，那么 URL 所指的页面就会被载入该框架。例如，要把页面载入名为“`topFrame`”的框架，可以使用下面的代码：

```
window.open("http://www.wrox.com/", "topFrame");
```

这行代码的行为就像用户点击一个链接，该链接的 `href` 为 `http://www.wrox.com/`，`target` 为“`topFrame`”。专用的框架名 `_self`、`_parent`、`_top` 和 `_blank` 也是有效的。

如果声明的框架名无效，`window.open()` 将打开新窗口，该窗口的特性由第三个参数（特性字符串）决定。如果省略第三个参数，将打开新的浏览器窗口，就像点击了 `target` 被设置为 `_blank` 的链接。这意味着新浏览器窗口的设置与默认浏览器窗口的设置（工具栏、地址栏和状态栏都是可见的）完全一样。

如果使用第三个参数，该方法就假设应该打开新窗口。特性字符串是用逗号分隔的设置列表，它定义新创建的窗口的某些方面。下表显示了各种设置：

设置	值	说 明
<code>left</code>	Number	说明新创建的窗口的左坐标。不能为负数*
<code>top</code>	Number	说明新创建的窗口的上坐标。不能为负数*
<code>height</code>	Number	设置新创建的窗口的高度。该数字不能小于 100*
<code>width</code>	Number	设置新创建的窗口的宽度。该数字不能小于 100*
<code>resizable</code>	yes,no	判断新窗口是否能通过拖动边线调整大小。默认值是 no
<code>scrollable</code>	yes,no	判断新窗口的视口容不下要显示的内容时是否允许滚动。默认值是 no
<code>toolbar</code>	yes,no	判断新窗口是否显示工具栏。默认值是 no
<code>status</code>	yes,no	判断新窗口是否显示状态栏。默认值是 no
<code>location</code>	yes,no	判断新窗口是否显示（Web）地址栏。默认值是 no

140

* 将在第 19 章详细讨论这些浏览器的安全性特征。

141

如前所述，特性字符串是用逗号分隔的，因此在逗号或等号前后不能有空格。例如，下面的字符串是无效的：

```
window.open("http://www.wrox.com/", "wroxwindow",
            "height=150, width= 300, top=10, left= 10, resizable =yes");
```

由于逗号后和几个等号前后的空格，所以该字符串无效。删除空格，它就能正常运行：

```
window.open("http://www.wrox.com/", "wroxwindow",
            "height=150, width=300, top=10, left=10, resizable=yes");
```

`window.open()` 方法将返回 `window` 对象作为它的函数值，该 `window` 对象就是新创建的窗口（如果给定的名字是已有的框架名，则为框架）。用这个对象，可以操作新创建的窗口：

```
var oNewWin = window.open("http://www.wrox.com/", "wroxwindow",
    "height=150,width=300,top=10,left=10,resizable=yes");

oNewWin.moveTo(100, 100);
oNewWin.resizeTo(200, 200);
```

还可以使用该对象调用 `close()` 方法关闭新创建的窗口：

```
oNewWin.close();
```

如果新创建的窗口中有代码，还可以用下面的代码关闭其自身：

```
window.close();
```

这段代码只对新创建的窗口有效。如果在主浏览器窗口中调用 `window.close()` 方法，将得到一条消息：提示该脚本试图关闭窗口，询问是否真的要关闭该窗口。通用规则是，脚本可以关闭它们打开的任何窗口，但不能关闭其他窗口。

新窗口还有对打开它的窗口的引用，存放在 `opener` 属性中。只在新窗口的最高层 `window` 对象才有 `opener` 属性，这样用 `top.opener` 访问它会更安全。例如：

```
var oNewWin = window.open("http://www.wrox.com/", "wroxwindow",
    "height=150,width=300,top=10,left=10,resizable=yes");

alert(oNewWin.opener == window); //outputs "true"
```

在这个例子中，打开一个新窗口，然后测试它的 `opener` 属性是否等于 `window` 对象，从而证明 `opener` 属性确实指向 `window` 对象（该警告显示 "true"）。

某些情况下，打开新窗口对用户有帮助，但一般说来，最好尽量少弹出窗口。许多企业都突然开始引入 Web 站点上的弹出式广告，大多数用户对此都觉得很讨厌。于是，许多用户都安装了弹出式窗口的拦截程序，除非用户允许打开某些弹出式窗口，否则它将拦截所有弹出式窗口。记住，弹出式窗口拦截程序并不知道合法弹出式窗口与广告之间的区别，因此最好在弹出窗口时警告用户。

3. 系统对话框

除弹出新的浏览器窗口，还可使用其他方法向用户弹出信息，即利用 `window` 对象的 `alert()`、`confirm()` 和 `prompt()` 方法。

你已对调用 `alert()` 方法的语法了如指掌，因为迄今为止许多示例中都使用了该方法。它只接受一个参数，即要显示给用户的文本。调用 `alert()` 方法后，浏览器将创建一个具有 `OK` 按钮的系统消息框，显示指定的文本。例如，下面的代码将显示图 5-6 所示的消息框：

```
alert("Hello world! ");
```

通常在提示用户注意某些不能控制的东西（如错误）时，使用警告对话框。通常用户在表单中输入无效数据时，显示警告对话框。

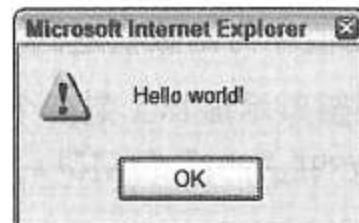


图 5-6

第二种类型的对话框通过调用 `confirm()` 方法显示的。确认对话框看起来与警告对话框相似，因为都是向用户显示信息。这两种对话框的主要区别是确认对话框中除 `OK` 按钮外还有 `Cancel` 按钮，这样允许用户说明是否执行指定的动作。例如，下面的代码显示的确认对话框如图 5-7 所示：

```
confirm("Are you sure? ");
```

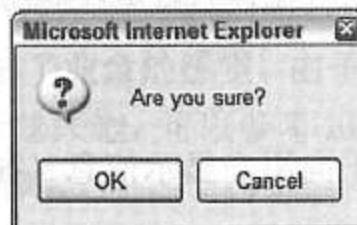


图 5-7

143

为判断用户点击的是 `OK` 按钮还是 `Cancel` 按钮，`confirm()` 方法返回一个 Boolean 值，如果点击的是 `OK` 按钮，返回 `true`，点击的是 `Cancel` 按钮，返回 `false`。确认对话框的典型用法如下：

```
if (confirm("Are you sure? ")) {
    alert("I'm so glad you're sure! ");
} else {
    alert("I'm sorry to hear you're not sure. ");
}
```

在这个例子中，第一行代码向用户显示确认对话框，这个 `confirm()` 方法是 `if` 语句的条件。如果用户点击 `OK` 按钮，显示的警告消息是 "I'm so glad you're sure!"，如果用户点击的是 `Cancel` 按钮，则显示的警告消息是 "I'm sorry to hear you're not sure!"。通常在用户尝试删除某些东西（例如删除他或她信箱中的电子邮件）时显示这种类型的提示。

最后的对话框通过调用 `prompt()` 方法显示，如你所料，该对话框提示用户输入某些信息。除 `OK` 按钮和 `Cancel` 按钮外，该对话框还有文本框，要求用户在此输入某些数据。`prompt()` 方法接受两个参数，即要显示给用户的文本和文本框中的默认文本（如果不需，可以是空串）。下面的代码将显示图 5-8 所示的对话框：

```
prompt("What's your name? ", "Michael");
```

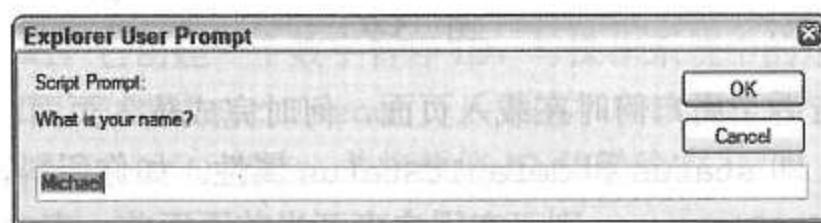


图 5-8

如果点击 OK 按钮，将文本框中的值作为函数值返回。如果点击 Cancel 按钮，返回 null。通常以下面的方式使用 `prompt()` 方法：

```
var sResult = prompt("What is your name? ", "");
if (sResult != null) {
    alert("Welcome, " + sResult);
}
```

关于这三种对话框，还有最后几点要说。首先，所有对话框窗口都是系统窗口，意味着不同的操作系统（有时是不同的浏览器）显示的窗口可能不同。这也意味着你对以什么字体、颜色等外观显示窗口没有任何控制权。

其次，这些对话框是模态（modal）的，意思是如果用户未点击 OK 按钮或 Cancel 按钮关闭该对话框，就不能在浏览器窗口中做任何操作。这是控制用户行为，以确保安全交付重要信息的通用方法。

4. 状态栏

144

状态栏是底部边界内的区域，用于向用户显示信息（如图 5-9 所示）。

状态栏

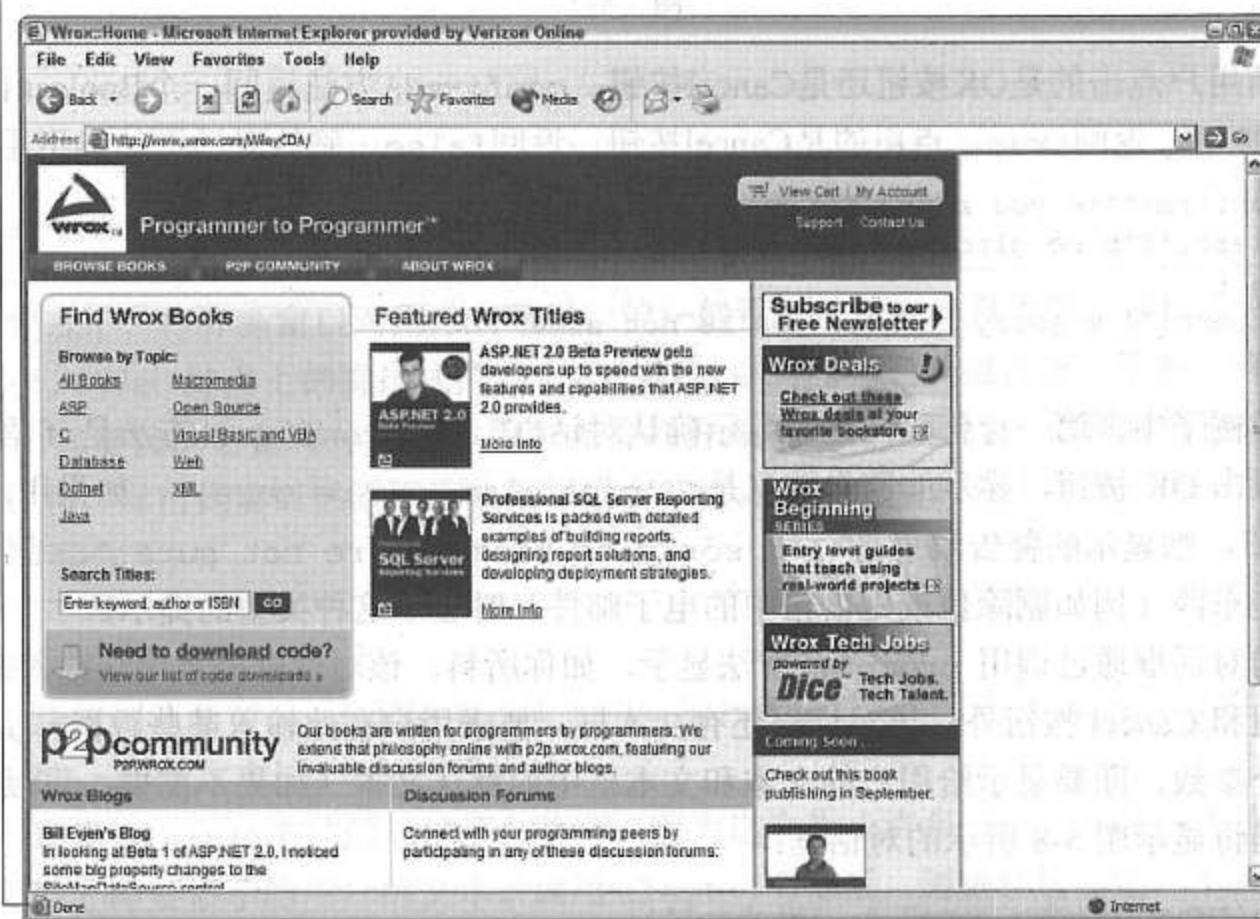


图 5-9

一般说来，状态栏告诉了用户何时在载入页面，何时完成载入页面。但可以用 `window` 对象的两个属性设置它的值，即 `status` 和 `defaultStatus` 属性。如你所料，`status` 可以使状态栏的文本暂时改变，而 `defaultStatus` 则可在用户离开当前页面前一直改变该文本。例如，在第一次载入页面时，可使用默认的状态栏消息：

```
window.defaultStatus = "You are surfing www.wrox.com. ";
```

你也许还想在用户把鼠标移到某个链接上时显示该链接的信息：

```
<a href="books.htm" onmouseover="window.status='Information on Wrox books.'>Books</a>
```

在使用 JavaScript URL 时这点非常有用，因为默认情况下，当鼠标移到链接上时，浏览器默认在状态栏中显示 href 特性的值。设置 window.status 属性，可向用户隐瞒链接实现的细节：

```
<a href="javascript:goSomewhere(1,2,3,4)" onmouseover="window.status='Information on Wrox books.'">Books</a>
```

145

注意不要过度使用状态栏，以使它分散用户的注意力。例如，许多站点仍使用滚动消息代码在状态栏中滚动文本。这种技巧不仅没用，且很讨厌，非常的不专业，给那些完全可以不用它的 Web 站点或 Web 应用程序添加了业余的感觉。由于本书介绍的是专业 JavaScript，所以不介绍滚动文本代码。但如果你对此感兴趣，可以参考 <http://javascript.internet.com-scrolls/>，在此可以找到大量此类脚本。

5. 时间间隔和暂停

Java 开发者熟悉对象的 wait() 方法，可使程序暂停，在继续执行下一行代码前，等待指定的时间量。这种功能非常有用，遗憾地是，JavaScript 未提供相应的支持。但这种功能并非完全不能实现，有几种方法可以采用。

JavaScript 支持暂停和时间间隔，这可有效地告诉浏览器应该何时执行某行代码。所谓暂停，是在指定的毫秒数后执行指定的代码。时间间隔是反复执行指定的代码，每次执行之间等待指定的毫秒数。

可以用 window 对象的 setTimeout() 方法设置暂停。该方法接受两个参数，要执行的代码和在执行它之前要等待的毫秒数（1/1000 秒）。第一个参数可以是代码串（与 eval() 函数的参数相同），也可是函数指针。例如，下面的代码都在 1 秒钟后显示一条警告：

```
setTimeout("alert('Hello world!')", 1000);
setTimeout(function() { alert("Hello world!"); }, 1000);
```

当然，还可以引用以前定义的函数：

```
function sayHelloWorld() {
    alert("Hello world!");
}

setTimeout(sayHelloWorld, 1000);
```

调用 setTimeout() 时，它创建一个数字暂停 ID，与操作系统中的进程 ID 相似。暂停 ID 本质上是要延迟的进程的 ID，在调用 setTimeout() 后，就不应该再执行它的代码。要取消还未执行的暂停，可调用 clearTimeout() 方法，并将暂停 ID 传递给它：

```
var iTimeoutId = setTimeout("alert('Hello world!')", 1000);
```

```
//nevermind
clearTimeout(iTimeoutId);
```

你也许会问：“为什么要定义暂停，又在执行它之前将其取消呢？”请考虑现在大多数应用程序中可见的工具提示。当把鼠标移动到一个按钮上时，停留一会，等待出现黄色的文本框，提示该按钮的功能。如果只是短暂地把鼠标移到该按钮上，然后很快将其移到另一个按钮上，那么第一个按钮的工具提示就不会显示。这就是要在执行暂停代码前取消它的原因。因为你在执行代码前只想等待指定的时间量。如果用户的操作产生了不同的结果，则需要取消该暂停。

时间间隔与暂停的运行方式相似，只是它无限次地每隔指定的时间段就重复一次指定的代码。可调用 `setInterval()` 方法设置时间间隔，它的参数与 `setTimeout()` 相同，是要执行的代码和每次执行之间等待的毫秒数。例如：

```
setInterval("alert('Hello world!') ", 1000);
setInterval(function() { alert("Hello world!"); }, 1000);

function sayHelloWorld() {
    alert("Hello world!");
}

setInterval(sayHelloWorld, 1000);
```

此外，与 `setTimeout()` 类似，`setInterval()` 方法也创建时间间隔 ID，以标识要执行的代码。`ClearInterval()` 方法可用这个 ID 阻止再次执行该代码。显然，这一点在使用时间间隔时更重要，因为如果不取消时间间隔，就会一直执行它，直到页面被卸载为止。下面是时间间隔用法的一个常见示例：

```
var iNum = 0;
var iMax = 100;
var iIntervalId = null;

function incNum() {
    iNum++;

    if (iNum == iMax) {
        clearInterval(iIntervalId);
    }
}

iIntervalId = setInterval(incNum, 500);
```

在这段代码中，每隔 500 毫秒，就对数字 `iNum` 进行一次增量运算，直到它达到最大值(`iMax`)，此时该时间间隔将被清除。也可用暂停实现该操作，这样即不必跟踪时间间隔的 ID，代码如下：

```
var iNum = 0;
var iMax = 100;

function incNum() {
    iNum++;

    if (iNum != iMax) {
```

```

        setTimeout(incNum, 500);
    }
}

setTimeout(incNum, 500);

```

这段代码使用链接暂停，即 `setTimeout()` 执行的代码也调用了 `setTimeout()`。如果在执行过增量运算后，`iNum` 不等于 `iMax`，就调用 `setTimeout()` 方法。不必跟踪暂停 ID，也不必清除它，因为代码执行后，将销毁暂停 ID。

147

那么应该使用哪种方法呢？这由使用情况决定。在执行一组指定的代码前等待一段时间，则使用暂停。如果要反复执行某些代码，就使用时间间隔。

6. 历史

可以访问浏览器窗口的历史。所谓历史，是用户访问过的站点的列表。出于安全原因，所有导航只能通过历史完成，不能得到浏览器历史中包含的页面的 URL。

不必通过时间机器实现历史导航，只需使用 `window` 对象的 `history` 属性及它的相关方法即可。

`go()` 方法只有一个参数，即前进或后退的页面数。如果是负数，就在浏览器历史中后退。如果是正数，就前进（这种差别就像 Back 和 Forward 按钮之间的差别）。

因此，后退一页，可用下面的代码：

```
window.history.go(-1);
```

当然，`window` 对象的引用不是必需的，也可使用下面的代码：

```
history.go(-1);
```

通常用该方法创建网页中嵌入的 Back 按钮，例如：

```
<a href="javascript:history.go(-1)">Back to the previous page</a>
```

要前进一页，只需要使用正数 1：

```
history.go(1);
```

另外，用 `back()` 和 `forward()` 方法可以实现同样的操作：

```
//go back one
history.back();
```

```
//go forward one
history.forward();
```

这些代码更有意义一些，因为它们精确地反应出浏览器的 Back 和 Forward 按钮的行为。

148

虽然不能使用浏览器历史中的 URL，但可以用 `length` 属性查看历史中的页面数：

```
alert("There are currently " + history.length + " pages in history.");
```

如果想前进或后退多个页面，想知道是否可以这样做，那么上面的代码就非常有用。

5.3.2 document 对象

`document` 对象实际上是 `window` 对象的属性，如你所知，`window` 对象的任何属性和方法都可直接访问，所以下面这行代码将返回“true”：

```
alert(window.document == document);
```

这个对象的独特之处是它是唯一一个既属于 BOM 又属于 DOM（下一章将讨论 DOM 中的 `document` 对象）的对象。从 BOM 角度看，`document` 对象由一系列集合构成，这些集合可以访问文档的各个部分，并提供页面自身的信息。再有，由于 BOM 没有可以指导实现的标准，所以每个浏览器实现的 `document` 对象都稍有不同，这一节的重点是最常用的功能。

下表列出了 BOM 的 `document` 对象的一些通用属性：

属性	说 明
<code>alinkColor</code>	激活的链接的颜色，如 <code><body alink="color"></code> 定义的*
<code>bgColor</code>	页面的背景颜色，如 <code><body bgcolor="color"></code> 定义的*
<code>fgColor</code>	页面的文本颜色，如 <code><body text="color"></code> 定义的*
<code>lastModified</code>	最后修改页面的日期，是字符串
<code>linkColor</code>	链接的颜色，如 <code><body link="color"></code> 定义的*
<code>referrer</code>	浏览器历史中后退一个位置的 URL
<code>title</code>	<code><title></code> 标签中显示的文本
<code>URL</code>	当前载入的页面的 URL
<code>vlinkColor</code>	访问过的链接的颜色，如 <code><body vlink="color"></code> 定义的*

* 反对使用这些属性，因为它们引用了`<body>`标签中的旧 HTML 特性。应该用样式表脚本代替它们。

149 `lastModified` 属性获取的是最后一次修改页面的日期的字符串表示，用作旁注，除非你想在主页上显示最后的修改日期（用服务器端的技术也可实现）。同样，`referrer` 属性用处也不大，除非你想跟踪用户是从哪里链接过来的（也许可以查看该用户是通过 Google 或是其他搜索引擎访问你的站点的）。同样也可以用服务器端的技术实现它。

`title` 属性是可读写的，所以可随时改变页面的标题，无论 HTML 页面的内容是什么。当站点使用了框架，且只有一个框架改变，其他框架保持不变时，该属性非常有用。可以用该属性改变框架的标题，从而反映出载入了新页面：

```
top.document.title = "New page title";
```

`URL` 属性也是可读可写的，所以可用它获取当前页面的 URL，或者把它设置为新的 URL，把窗口导航到新页面。例如：

```
document.URL = "http://www.wrox.com/";
```

如前所述，`document` 对象也有许多集合，提供对载入的页面的各个部分的访问。下表列出了这些集合：

集 合	说 明
anchors	页面中所有锚的集合（由 >表示）
applets	页面中所有 applet 的集合
embeds	页面中所有嵌入式对象的集合（由<embed/>标签表示）
forms	页面中所有表单的集合
images	页面中所有图像的集合
links	页面中所有链接的集合（由 ><a/>表示）

与 window.frame 集合相似，可用数字或名字引用 document 对象的每个集合，也就是说可用 document.images[0] 或 document.images["image_name"] 访问图像。考虑下面的 HTML 页面：

```
<html>
  <head>
    <title>Document Example</title>
  </head>
  <body>
    <p>Welcome to my <a href="home.htm">home</a> away from home.</p>
    
    <form method="post" action="submit.cgi" name="frmSubscribe">
      <input type="text" name="txtEmail" />
      <input type="submit" value="Subscribe" />
    </form>
  </body>
</html>
```

150

访问该文档各个部分的方法如下：

- 用 document.links[0] 访问链接；
- 用 document.images[0] 或 document.images["imgHome"] 访问图像；
- 用 document.forms[0] 或 document.forms["frmSubscribe"] 访问表单。

此外，链接和图像等的所有特性都变成了该对象的属性。例如，document.images[0].src 是获取第一个图像的 src 特性的代码。

最后，BOM 的 document 对象还有几个方法。最常用的方法之一是 write() 或它的兄弟方法 writeln()。这两个方法都接受一个参数，即要写入文档的字符串。如你所料，它们之间唯一的区别是 writeln() 方法将在字符串末尾加一个换行符 (\n)。

这两个方法都会把字符串的内容插入到调用它们的位置。这样浏览器就会像处理页面中的常规 HTML 代码一样处理这个文档字符串。考虑下面的页面：

```
<html>
  <head>
    <title>Document Write Example</title>
  </head>
  <body>
    <h1><script type="text/javascript">document.write("this is a
test")</script></h1>
  </body>
</html>
```

该页面在浏览器中看来与下面的页面一样：

```
<html>
  <head>
    <title>Document Write Example</title>
  </head>
  <body>
    <h1>this is a test</h1>
  </body>
</html>
```

可以使用这种功能动态地引入外部 JavaScript 文件。例如：

```
<html>
  <head>
    <title>Document Example</title>
    <script type="text/javascript">
      document.write("<script type=\"text/javascript\" src=\"external.js\">" +
                    "</scr" + "ipt>");
    </script>
  </head>
  <body>
  </body>
</html>
```

151

这段代码在页面上写了一个`<script>`标签，将使浏览器像常规一样载入外部 JavaScript 文件。注意字符串`"</script>"`被分成两部分（`"</src"`和`"ipt>"`）。这是必要的，因为每当浏览器遇到`<script>`，它都假定其中的代码块是完整的（即使它出现在 JavaScript 字符串中）。假设前面的例子未把`"</script>"`分成两部分：

```
<html>
  <head>
    <title>Document Example</title>
    <script type="text/javascript">
      document.write("<script type=\"text/javascript\" src=\"external.js\">" +
                    "</script>"); //this will cause a problem
    </script>
  </head>
  <body>
  </body>
</html>
```

浏览器显示如下网页：

```
<html>
  <head>
    <title>Document Example</title>
    <script type="text/javascript">
      document.write("<script type=\"text/javascript\" src=\"external.js\">" +
                    "</script>" +
                    "</script>")
    </script>
  </head>
```

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

可以看到，忘记把字符串 "`</script>`" 分成两部分引起了严重的混乱。首先，在 `<script>` 标签内有个语法错，因为 `document.write()` 的调用漏掉了闭括号。其次，有两个 `</script>` 标签。这就是在使用 `document.write()` 方法把 `<script>` 标签写入页面时一定要把 "`</script>`" 字符串分开的原因。

记住，要插入内容属性，必须在完全载入页面前调用 `write()` 和 `writeln()` 方法。如果任何一个方法是在页面载入后调用的，它将抹去页面的内容，显示指定的内容。

与 `write()` 和 `writeln()` 方法密切相关的是 `open()` 和 `close()` 方法。`open()` 方法用于打开已经载入的文档以便进行编写，`close()` 方法用于关闭 `open()` 方法打开的文档，本质上是告诉它显示写入其中的所有内容。通常把这些方法结合在一起，用于向框架或新打开的窗口中写入内容，如下所示：

```
var oNewWin = window.open("about:blank", "newwindow",
    "height=150,width=300,top=10,left=10,resizable=yes");

oNewWin.document.open();
oNewWin.document.write("<html><head><title>New Window</title></head>");
oNewWin.document.write("<body>This is a new window!</body></html>");
oNewWin.document.close();
```

这个例子打开空白页（使用本地的 "about:blank" URL），然后写入新页面。要正确实现这一操作，在调用 `write()` 前，先调用 `open()` 方法。写完后，调用 `close()` 方法完成显示。当你想显示无需返回服务器的页面时，这种方法非常有用。

5.3.3 location 对象

BOM 中最有用的对象之一是 `location` 对象，它是 `window` 对象和 `document` 对象的属性（对此没有什么标准，导致了一些混乱）。`location` 对象表示载入窗口的 URL，此外，它还可以解析 URL：

- `hash`——如果 URL 包含 #，该方法将返回该符号之后的内容（例如，`http://www.somewhere.com/index#selection1` 的 `hash` 等于 "#selection1"）。
- `host`——服务器的名字（如 `www.wrox.com`）。
- `hostname`——通常等于 `host`，有时会省略前面的 `www`。
- `href`——当前载入的页面的完整 URL。
- `pathname`——URL 中主机名后的部分。例如，`http://www.somewhere.com/pictures/index.htm` 的 `pathname` 是 "/pictures/index.htm"。
- `port`——URL 中声明的请求的端口。默认情况下，大多数 URL 没有端口信息，所以该属

性通常是空白的。像 `http://www.somewhere.com:8080/index.htm` 这样的 URL 的 port 属性等于 8080。

- protocol——URL 中使用的协议，即双斜杠（//）之前的部分。例如，`http://www.wrox.com` 中的 protocol 属性等于 `http:`，`ftp://www.wrox.com` 的 protocol 属性等于 `ftp:`。
- search——执行 GET 请求的 URL 中的问号（?）后的部分，又称为查询字符串。例如，`http://www.somewhere.com/search.htm?term=javascript` 中的 search 属性等于 `?term=javascript`

`location.href` 是最常用的属性，用于获取或设置窗口的 URL（在这一点上，它类似于 `document.URL` 属性）。改变该属性的值，就可导航到新页面：

```
location.href = "http://www.wrox.com/";
```

153

采用这种方式导航，新地址将被加到浏览器的历史栈中，放在前一个页面后，意味着 Back 按钮会导航到调用了该属性的页面。

`assign()` 方法实现的是同样的操作：

```
location.assign("http://www.wrox.com");
```

这两种方法都可以采用，不过大多数开发者选用 `location.href` 属性，因为它更精确地表达了代码的意图。

如果不想让包含脚本的页面能从浏览器历史中访问，可使用 `replace()` 方法。该方法所作的操作与 `assign()` 方法一样，但它多了一步操作，即从浏览器历史中删除包含脚本的页面，这样就不能通过浏览器的 Back 和 Forward 按钮访问它了。你可以自己试试看：

```
<html>
  <head>
    <title>You won't be able to get back here</title>
  </head>
  <body>
    <p>Enjoy this page for a second, because you won't be coming back here.</p>
    <script type="text/javascript">
      setTimeout(function () {
        location.replace("http://www.wrox.com/");
      }, 1000);
    </script>
  </body>
</html>
```

把这个页面载入浏览器中，等它导航到新页面后，再点击 Back 按钮。

`location` 对象还有个 `reload()` 方法，可重新载入当前页面。`reload()` 方法有两种模式，即从浏览器缓存中重载，或从服务器端重载。究竟采用哪种模式由该方法的参数值决定，如果是 `false`，则从缓存中载入，如果是 `true`，则从服务器端载入（如果省略参数，默认值为 `false`）。因此，要从服务器端重载当前页面，可以使用下面的代码：

```
location.reload(true);
```

要从缓存重载当前页面，可以采用下面两行代码中的任意一行：

```
location.reload(false);
location.reload();
```

reload()方法调用后的代码可能被执行，也可能不被执行，这由网络延迟和系统资源等因素决定。因此，最好把 **reload()** 调用放在最后一行。

`location` 对象的最后一个方法是 `toString()`，它仅返回 `location.href` 的值。因此，下面两行代码是等价的：

```
alert(location);
alert(location.href);
```

154

本节采用示例介绍了 `location` 对象。记住，`location` 对象是 `window` 对象和 `document` 对象的属性，所以 `window.location` 和 `document.location` 互相等价，可以交换使用。

5.3.4 navigator 对象

`navigator` 对象是最早实现的 BOM 对象之一，Netscape Navigator 2.0 和 IE 3.0 引入了它。它包含大量有关 Web 浏览器的信息。它也是 `window` 对象的属性，可以用 `window.navigator` 引用它，也可以用 `navigator` 引用。

虽然微软公司最初把 Netscape 的浏览器称为 `navigator`，但 `navigator` 对象成了一种事实标准，用于提供 Web 浏览器的信息。（微软除 `navigator` 外，还有自己的对象 `clientInformation`，但它们两个提供的数据完全相同。）

同样，缺乏标准阻碍了 `navigator` 对象的发展，因为各个浏览器决定支持该对象的属性和方法。下表列出了最常用的属性和方法以及最常用的四种浏览器（IE、Mozilla、Opera 和 Safari）中哪个支持它们。

属性/方法	说 明	IE	Moz	Op	Saf
<code>appCodeName</code>	浏览器代码名的字符串表示（如“Mozilla”）	×	×	×	×
<code>appName</code>	官方浏览器名的字符串表示	×	×	×	×
<code>appMinorVersion</code>	额外版本信息的字符串表示	×	—	—	—
<code>appVersion</code>	浏览器版本信息的字符串表示	×	×	×	×
<code>browserLanguage</code> *	浏览器或操作系统的语言的字符串表示	×	—	×	—
<code>cookieEnabled</code>	说明是否启用了 cookie 的 Boolean 值	×	×	×	—
<code>cpuClass</code>	CPU 类别的字符串表示（“x86”、“68K”、“Alpha”、“PPC”或“other”）	×	—	—	—
<code>javaEnabled()</code>	说明是否启用了 Java 的 Boolean 值	×	×	×	×
<code>language</code>	浏览器语言的字符串表示	—	×	×	×

(续)

	属性/方法	说 明	IE	Moz	Op	Saf
155	mimeTypes	注册到浏览器的 mime 类型的数组	—	×	×	×
	onLine	说明浏览器是否连接到因特网上的 Boolean 值	×	—	—	—
	oscpu	操作系统或 CPU 的字符串表示	—	×	—	—
	platform	运行浏览器的计算机平台的字符串表示	×	×	×	×
	plugins	安装在浏览器中的插件的数组	×	×	×	×
	preference()	用于设置浏览器首选项的函数	—	×	×	—
	product	产品名的字符串表示（如“Gecko”）	—	×	—	×
	productSub	有关产品的额外信息的字符串表示（如 Gecko 版本）	—	×	—	×
	opsProfile		—	—	—	—
	securityPolicy		—	×	—	—
	systemLanguage*	操作系统语言的字符串表示	×	—	—	—
	taintEnabled()	说明是否启用了数据感染的 Boolean 值	×	×	×	×
	userAgent	用户代理头字符串的字符串表示	×	×	×	×
	userLanguage*	操作系统语言的字符串表示	×	—	—	—
	userProfile	允许访问浏览器用户档案的对象	×	—	—	—
	vendor	品牌浏览器名的字符串表示（如“Netscape6”或“Netscape”）	—	×	—	×
	vendorSub	品牌浏览器的额外信息的字符串表示（如 Netscape 的版本）	—	×	—	×

* 大多数情况下，`browserLanguage`、`systemLanguage` 和 `userLanguage` 相同。

在判断浏览器页面采用的是哪种浏览器方面时，`navigator` 对象非常有用。在因特网上可迅速检索到许多检测浏览器的方法，它们都大量地利用了 `navigator` 对象。第 9 章将介绍如何用 `navigator` 对象检测浏览器及操作系统。

5.3.5 screen 对象

虽然出于安全原因，有关用户系统的大多数信息都被隐藏了，但还可以用 `screen` 对象获取某些关于用户屏幕的信息（不出所料，它也是 `window` 对象的属性）。

`screen` 对象通常包含下列属性（不过，许多浏览器都加入了自己的属性）：

- `availHeight`——窗口可以使用的屏幕的高度（以像素计），其中包括操作系统元素（如 Windows 工具栏）需要的空间。
- `availWidth`——窗口可以使用的屏幕的宽度（以像素计）。
- `colorDepth`——用户表示颜色的位数。大多数系统采用 32 位。
- `height`——屏幕的高度，以像素计。
- `width`——屏幕的宽度，以像素计。

确定新窗口的大小时，`availHeight` 和 `availWidth` 属性非常有用。例如，可以使用下面的

代码填充用户的屏幕：

```
window.moveTo(0, 0);
window.resizeTo(screen.availWidth, screen.availHeight);
```

此外，这些数据与站点的流量工具一起使用，可以判断用户的图形接受能力。

5.4 小结

这一章介绍了Web浏览器中的JavaScript。它涵盖了把JavaScript加入HTML和SVG页的方法，并解释了两者之间的差别。此外，还讨论了XHTML对将JavaScript加入HTML页的影响以及如何为此做好准备。

在本章后面的小节中，学到了BOM及它提供的各种对象。了解了window对象是JavaScript世界的核心，其他所有BOM对象都不过是window对象的属性。

这一章解释了如何操作浏览器窗口和框架，用JavaScript移动它们，调整它们的大小。用location对象，可以访问和改变窗口的地址。用history对象，可以在用户访问过的页面中前进或后退。

最后，学到了如何用navigator对象和screen对象获取用户浏览器和屏幕的信息。

157

158



自 第一次使用 HTML 将因特网上相关的文档连接起来后，DOM 也许是 Web 上最伟大的创新了。DOM 给予开发者空前的对 HTML 的访问能力，并使开发者能将 HTML 作为 XML 文档来处理和查看。DOM 代表着由微软公司和 Netscape 公司所引领的动态 HTML 向真正跨平台的、语言独立的解决方案的重要演变。

6.1 什么是 DOM？

在开始详细介绍什么是 DOM 之前，你首先要了解是什么促使了它的诞生。尽管 DOM 很大程度上受到浏览器中动态 HTML 影响，但 W3C 还是将它最先应用于 XML。

6.1.1 XML 简介

XML（可扩展标记语言）是从称为 SGML（标准通用标记语言）的更加古老的语言派生出来的。SGML 的主要目的是定义使用标签来表示数据的标记语言的语法。

标签由包围在一个小于号（<）和一个大于号（>）之间的文本组成，例如<tag>。起始标签（start tag）表示一个特定区域的开始，例如<start>；结束标签（end tag）定义了一个区域的结束，除了在小于号之后紧跟着一个斜线（/）外，和起始标签基本一样，例如</end>。SGML 还定义了标签的特性（attribute），它们是定义在小于号和大于号之间的值，例如中的 src 特性。如果你觉得它看起来很熟悉的话，应该知道，基于 SGML 的语言的最著名实现就是原始的 HTML。

SGML 常用来定义针对 HTML 的文档类型定义(DTD)，同时它也常用于编写 XML 的 DTD。SGML 的问题就在于，它允许出现一些奇怪的语法，这让创建 HTML 的解析器成为一个大难题：

- 某些起始标签不允许出现结束标签，例如 HTML 中标签。包含了结束标签就会出现错误。
- 某些起始标签可以选择性出现结束标签或者隐含了结束标签，例如 HTML 中<p>标签，当出现另一个<p>标签或者某些其他标签时，便假设在这之前有一个结束标签。
- 某些起始标签要求必须出现结束标签，例如 HTML 中<script>标签。

- 标签可以以任何顺序嵌套。即使结束标签不按照起始标签的逆序出现也是允许的，例如，
`This is a <i> sample string</i>' 是正确的。
- 某些特性要求必须包含值，例如 `` 中的 `src` 特性。
- 某些特性不要求一定有值，例如 `<td nowrap>` 中的 `nowrap` 特性。
- 定义特性的两边有没有加上双引号都是可以的，所以 `` 和 `` 都是允许的。

这些问题使建立一个 SGML 语言的解析器变成了一项艰巨的任务。判断何时应用以上规则的困难导致了 SGML 语言的定义一直停滞不前。以这些问题作为出发点，XML 逐渐步入我们的视野。

XML 去掉了之前令许多开发人员头疼的 SGML 的随意语法。在 XML 中，采用了如下的语法：

- 任何的起始标签都必须有一个结束标签。
- 可以采用另一种简化语法，可以在一个标签中同时表示起始和结束标签。这种语法是在大于符号之前紧跟一个斜线 (/)，例如 `<tag />`。XML 解析器会将其翻译成 `<tag></tag>`。
- 标签必须按合适的顺序进行嵌套，所以结束标签必须按镜像顺序匹配起始标签，例如
`this is a <i>sample</i> string'。这好比是将起始和结束标签看作是数学中的左右括号：在没有关闭所有的内部括号之前，是不能关闭外面的括号的。
- 所有的特性都必须有值。
- 所有的特性都必须在值的周围加上双引号。

这些规则使得开发一个 XML 解析器要简便得多，而且也除去了了解析 SGML 中花在判断何时地应用那些奇怪语法规则上的工作。仅仅在 XML 出现后的前六年就衍生出多种不同的语言，包括 MathML、SVG、RDF、RSS、SOAP、XSLT、XSL-FO，而同时也将 HTML 改进为 XHTML。

如果需要关于 SGML 和 XML 具体技术上的对比，请查看 W3C 的注解，位于：<http://www.w3.org/TR/NOTE-sgml-xml.html>

160

如今，XML 已经是世界上发展最快的技术之一。它的主要目的是使用文本以结构化的方式来表示数据。在某些方面，XML 文件也类似于数据库，提供数据的结构化视图。这里是一个 XML 文件的例子：

```

<?xml version="1.0"?>
<books>
    <!-- begin the list of books -->
    <book isbn="0764543555">
        <title>Professional JavaScript for Web Developers</title>
        <author>Nicholas C. Zakas</author>
        <desc><![CDATA[
Professional JavaScript for Web Developers brings you up to speed on the latest
innovations in the world of JavaScript. This book provides you with the details of
JavaScript implementations in Web browsers and introduces the new capabilities
relating to recently-developed technologies such as XML and Web Services.
]]></desc>
    </book>

```

```

<?page render multiple authors ?>
<book isbn="0764570773">
    <title>Beginning XML, 3rd Edition</title>
    <author>David Hunter</author>
    <author>Andrew Watt</author>
    <author>Jeff Rafter</author>
    <author>Jon Duckett</author>
    <author>Danny Ayers</author>
    <author>Nicholas Chase</author>
    <author>Joe Fawcett</author>
    <author>Tom Gaven</author>
    <author>Bill Patterson</author>
    <desc><![CDATA[
Beginning XML, 3rd Edition, like the first two editions, begins with a broad
overview of the technology and then focuses on specific facets of the various
specifications for the reader. This book teaches you all you need to know about XML:
what it is, how it works, what technologies surround it, and how it can best be used
in a variety of situations, from simple data transfer to using XML in your Web
pages. It builds on the strengths of the first and second editions, and provides new
material to reflect the changes in the XML landscape - notably RSS and SVG.
    ]]></desc>
</book>
<book isbn="0764543555">
    <title>Professional XML Development with Apache Tools</title>
    <author>Theodore W. Leung</author>
    <desc><![CDATA[
If you're a Java programmer working with XML, you probably already use some of the
tools developed by the Apache Software Foundation. This book is a code-intensive
guide to the Apache XML tools that are most relevant for Java developers, including
Xerces, Xalan, FOP, Cocoon, Axis, and Xindice.
    ]]></desc>
</book>"*****"
</books>

```

161

每个 XML 文档都由 XML 序言开始，在前面的代码中的第一行便是 XML 序言，`<?xml version="1.0"?>`。这一行代码会告诉解析器和浏览器，这个文件应该按照前面讨论过的 XML 规则进行解析。第二行代码，`<books>`，则是文档元素（document element），它是文件中最外面的标签（我们认为元素（element）是起始标签和结束标签之间的内容）。所有其他的标签必须包含在这个标签之内来组成一个有效的 XML 文件。XML 文件的第二行并不一定要包含文档元素；如果有注释或者其他内容，文档元素可以迟些出现。

范例文件中的第三行代码是注释，你会发现它与 HTML 中使用的注释风格是一样的。这是 XML 从 SGML 中继承的语法元素之一。

页面再往下的一些地方，可以发现`<desc>`标签里有一些特殊的语法。`<![CDATA[]]>`代码用于表示无需进行解析的文本，允许诸如大于号和小于号之类的特殊字符包含在文本中，而无需担心破坏 XML 的语法。文本必须出现在`<![CDATA[和]]>`之间才能合适地避免被解析。这样的文本称为 Character Data Section，简称 CData Section。

下面的一行就是在第二本书的定义之前的：

```
<?page render multiple authors ?>
```

虽然它看上去很像 XML 序言，但实际上是一种称为处理指令（processing instruction）的不同类型的语法。处理指令（以下简称 PI）的目的是为了给处理页面的程序（例如 XML 解析器）提供额外的信息。PI 通常情况下是没有固定格式的，唯一的要求是紧随第一个问号必须至少有一个字母。在此之后，PI 可以包含除了小于号和大于号之外的任何字符串序列。

最常见的 PI 是用来指定 XML 文件的样式表：

```
<?xml-stylesheet type="text/css" href="MyStyles.css" ?>
```

这个 PI 一般会直接放在 XML 序言之后，通常由 Web 浏览器使用，来将 XML 数据以特殊的样式显示出来。

如果你对 XML 感兴趣，想学习更多关于它及其应用的内容，请参见人民邮电出版社即将出版的《XML 与 DOM 基础教程》。

6.1.2 针对 XML 的 API

将 XML 定义为一种语言之后，就出现了使用常见的编程语言（如 Java）来同时表现和处理 XML 代码的需求。

首先出现的是 Java 上的 SAX(Simple API for XML)项目。SAX 提供了一个基于事件的 XML 解析的 API。从其本质上来说，SAX 解析器从文件的开头出发，从前向后解析，每当遇到起始标签或者结束标签、特性、文本或者其他 XML 语法时，就会触发一个事件。然后，当事件发生时，具体要怎么做就由开发人员决定。

因为 SAX 解析器仅仅按照文本的方式来解析它们，所以 SAX 更轻量、更快速。而它们的主要缺点是在解析中无法停止、后退或者不从文件开始，直接访问 XML 结构中的指定部分。

DOM 是针对 XML 的基于树的 API。它关注的不仅仅是解析 XML 代码，而是使用一系列互相关联的对象来表示这些代码，而这些对象可以被修改且无需重新解析代码就能直接访问它们。

使用 DOM，只需解析代码一次来创建一个树的模型；某些时候会使用 SAX 解析器来完成它。在这个初始解析过程之后，XML 已经完全通过 DOM 模型来表现出来，同时也不再需要原始的代码。尽管 DOM 比 SAX 慢很多，而且，因为创建了相当多的对象而需要更多的开销，但由于它使用上的简便，因而成为 Web 浏览器和 JavaScript 最喜欢的方法。

162

注意 DOM 是语言无关的 API，这意味着它的实现并不与 Java、JavaScript 或者其他语言绑定。然而，鉴于本书的目的，我将大部分的注意力放在 JavaScript 的实现上。

6.1.3 节点的层次

那么基于树的 API 到底指什么呢？当谈论 DOM 树（也称之为文档）的时候，实际上谈论的是节点（node）的层次。DOM 定义了 Node 的接口以及许多种节点类型来表示 XML 节点的各个方面：

- Document——最顶层的节点，所有的其他节点都是附属于它的。
- DocumentType——DTD 引用（使用<!DOCTYPE>语法）的对象表现形式，例如<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">。它不能包含子节点。
- DocumentFragment——可以像 Document 一样来保存其他节点。
- Element——表示起始标签和结束标签之间的内容，例如<tag></tag>或者<tag/>。这是唯一可以同时包含特性和子节点的节点类型。
- Attr——代表一对特性名和特性值。这个节点类型不能包含子节点。
- Text——代表 XML 文档中的在起始标签和结束标签之间，或者 CDATA Section 内包含的普通文本。这个节点类型不能包含子节点。
- CDATASection——<![CDATA[]]>的对象表现形式。这个节点类型仅能包含文本节点 Text 作为子节点。
- Entity——表示在 DTD 中的一个实体定义，例如<!ENTITY foo "foo">。这个节点类型不能包含子节点。
- EntityReference——代表一个实体引用，例如"。这个节点类型不能包含子节点。
- ProcessingInstruction——代表一个 PI。这个节点类型不能包含子节点。
- Comment——代表 XML 注释。这个节点类型不能包含子节点。
- Notation——代表在 DTD 中定义的记号。这个很少用到，所以在本书中不会讨论。

163

一个文档是由任意数量的节点的层次组成。考虑下面的 XML 文档：

```
<?xml version="1.0"?>
<employees>
  <!-- only employee -->
  <employee>
    <name>Michael Smith</name>
    <position>Software Engineer</position>
    <comments><![CDATA[
      His birthday is on 8/14/68.
    ]]></comments>
  </employee> ""
</employees>
```

这段代码可以用一个 DOM 文档。

在图 6-1 中，每个矩形代表在 DOM 文档树中的一个节点，粗体文本表示节点的类型，非粗体的文本代表该节点的内容。

注释和<employee/>节点都被认为是<employees/>的子节点，因为它们在这棵树中直接在<employees/>节点的下面。同样的，我们也认为<employees/>是注释和<employee/>节点的父节点。

类似的，<name/>、<position/>以及<comments/>都被认为是<employee/>的子节点，同时，因为它们在 DOM 树中处于同一层上，有着相同的父节点，所以认为它们是兄弟（sibling）关系。

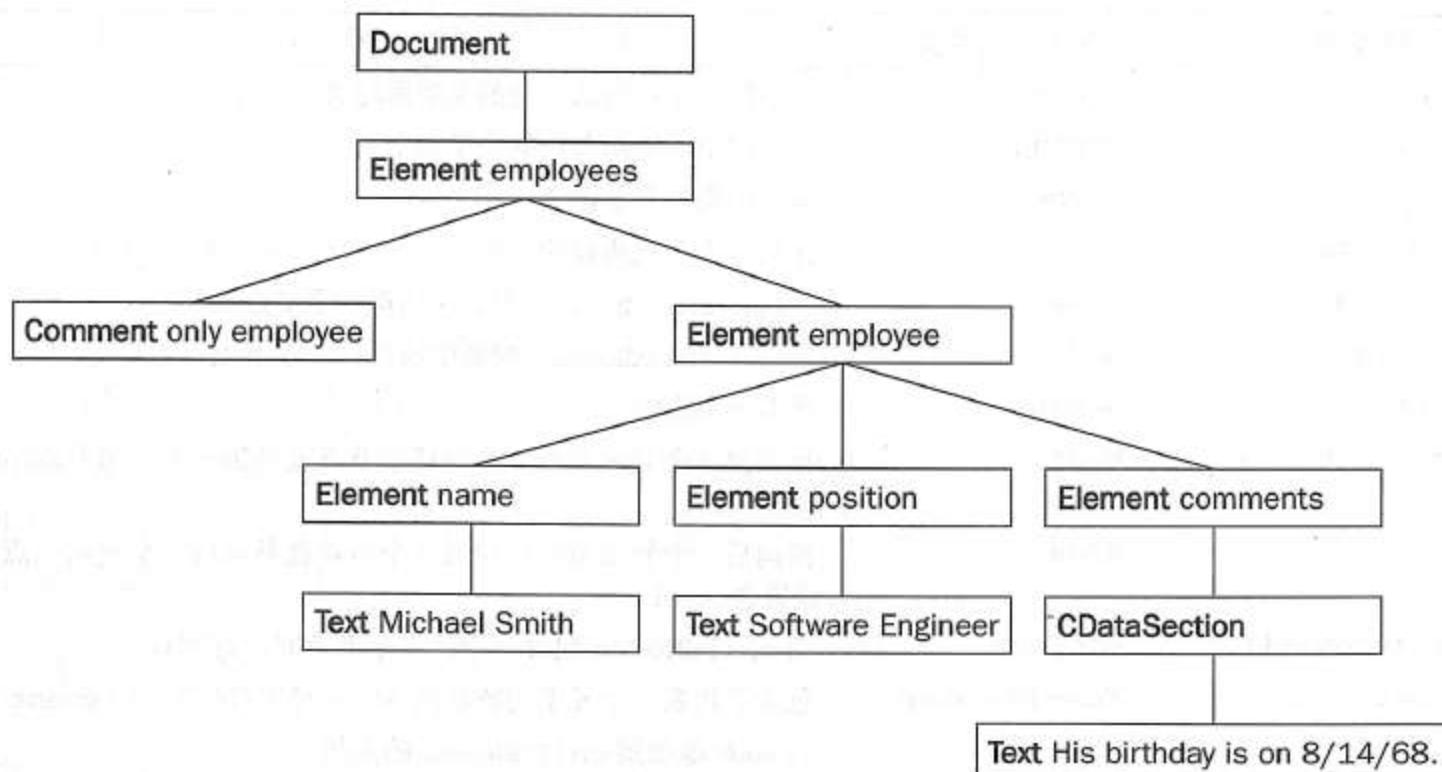


图 6-1

我们还认为`<employees>`节点是这一节中所有节点的祖先，其中包括它自己的子节点（注释和`<employee>`）以及子节点的子节点（`<name>`、`<position>`，等等，直到文本节点“His birthday is on 8/14/68”）。并认为文档节点是文档中所有节点的祖先。

`Node` 接口定义了对应不同节点类型的 12 个常量（它们会在即将讨论的 `nodeType` 特性中使用到）：

- `Node.ELEMENT_NODE` (1)
- `Node.ATTRIBUTE_NODE` (2)
- `Node.TEXT_NODE` (3)
- `Node.CDATA_SECTION_NODE` (4)
- `Node.ENTITY_REFERENCE_NODE` (5)
- `Node.ENTITY_NODE` (6)
- `Node.PROCESSING_INSTRUCTION_NODE` (7)
- `Node.COMMENT_NODE` (8)
- `Node.DOCUMENT_NODE` (9)
- `Node.DOCUMENT_TYPE_NODE` (10)
- `Node.DOCUMENT_FRAGMENT_NODE` (11)
- `Node.NOTATION_NODE` (12)

`Node` 接口也定义了一些所有节点类型都包含的特性和方法。我们在下面的表格中列出了这些特性和方法：

特性/方法	类型/返回类型	说 明
nodeName	String	节点的名字；根据节点的类型而定义
nodeValue	String	节点的值；根据节点的类型而定义
nodeType	Number	节点的类型常量值之一
ownerDocument	Document	指向这个节点所属的文档
firstChild	Node	指向在 childNodes 列表中的第一个节点
lastChild	Node	指向在 childNodes 列表中的最后一个节点
childNodes	NodeList	所有子节点的列表
previousSibling	Node	指向前一个兄弟节点；如果这个节点就是第一个兄弟节点，那么该值为 null
nextSibling	Node	指向后一个兄弟节点；如果这个节点就是最后一个兄弟节点，那么该值为 null
hasChildNodes()	Boolean	当 childNodes 包含一个或多个节点时，返回真
attributes	NamedNodeMap	包含了代表一个元素的特性的 Attr 对象；仅用于 Element 节点
appendChild(node)	Node	将 node 添加到 childNodes 的末尾
removeChild(node)	Node	从 childNodes 中删除 node
replaceChild (newnode, oldnode)	Node	将 childNodes 中的 oldnode 替换成 newnode
insertBefore (newnode, refnode)	Node	在 childNodes 中的 refnode 之前插入 newnode

除节点外，DOM 还定义了一些助手对象，它们可以和节点一起使用，但不是 DOM 文档必有的部分。

- NodeList——节点数组，按照数值进行索引；用来表示一个元素的子节点。
 - NamedNodeMap——同时用数值和名字进行索引的节点表；用于表示元素特性。
- 这些助手对象为处理 DOM 文档提供附加的访问和遍历方法。具体用法将在后面讨论。

6.1.4 特定语言的 DOM

任何基于 XML 的语言，如 XHTML 和 SVG，因为它们从技术上来说还是 XML，仍然可以利用刚刚介绍的核心 DOM。然而，很多语言会继续定义它们自己的 DOM 来扩展 XML 核心以提供语言的特色功能。

开发 XML DOM 的同时，W3C 还一起开发了一种特别针对 XHTML（以及 HTML）的 DOM。这个 DOM 定义了一个 HTMLDocument 和一个 HTMLElement 作为这种实现的基础。每个 HTML 元素通过它自己的 HTMLElement 类型来表示，例如 HTMLDivElement 代表了<div>，但有少数元素除外，它们只包含 HTMLElement 提供的属性和方法。在本书后面的部分中，会不断介绍不同的 HTML DOM 的特点以及核心 XML DOM 的特点。

普通 HTML 并不是合法的 XML。然而，大部分当前的 Web 浏览器都很宽容，依然可以将一个 HTML 文档解析为合适的 DOM 文档（即使没有 XML 序言）。但是，在编写 Web 页面的时候最好使用 XHTML 代码以消除那些坏的编码习惯。

W3C也发布了一些特定语言的DOM，包括SVG (<http://www.w3.org/TR/SVG>)，SMIL Animation (<http://www.w3.org/TR/smil-animation>)和MathML (<http://www.w3.org/TR/MathML2>)。

6.2 对DOM的支持

正如前面所说的，并不是所有的浏览器对DOM的支持都一样。一般来说，Mozilla对DOM标准支持最好，支持几乎所有的DOM Level 2，以及部分DOM Level 3。在Mozilla之后，Opera和Safari也在完全支持上做了突出工作，极大地缩小了和标准之间的差距，支持几乎所有的DOM Level 1和大部分DOM Level 2。这个领域中落在最后的是IE，它对DOM Level 1的实现都还不完整，尚有很多方面有待完善。

6.3 使用DOM

`document`对象是BOM的一部分，同时也是HTML DOM的`HTMLDocument`对象的一种表现形式，反过来说，它也是XML DOM Document对象。JavaScript中的大部分处理DOM的过程都利用`document`对象，所以我们从这里开始讨论还是比较合理的。

6.3.1 访问相关的节点

在下面的几节中考虑下面的HTML页面：

```
<html>
  <head>
    <title>DOM Example</title>
  </head>
  <body>
    <p>Hello World!</p>
    <p>Isn't this exciting?</p>
    <p>You're learning to use the DOM!</p>
  </body>
</html>
```

要访问`<html/>`元素（你应该明白这是该文件的`document`元素），你可以使用`document`的`documentElement`特性：

```
var oHtml = document.documentElement;
```

由于IE 5.5 中的DOM实现的错误，`document.documentElement`会返回`<body/>`元素。IE 6.0 已经修复了这个错误。

167

现在变量`oHtml`包含一个表示`<html/>`的`HTMLElement`对象。如果你想取得`<head/>`和`<body/>`元素，下面的可以实现：

```
var oHead = oHtml.firstChild;
var oBody = oHtml.lastChild;
```

也可以使用 `childNodes` 特性来完成同样的工作。只需把它当成普通的 JavaScript `Array`，使用方括号标记：

```
var oHead = oHtml.childNodes[0];
var oBody = oHtml.childNodes[1];
```

你还可以通过使用 `childNodes.length` 特性来获取子节点的数量：

```
alert(oHtml.childNodes.length); //outputs "2"
```

注意方括号标记其实是 `NodeList` 在 JavaScript 中的简便实现。实际上正式的从 `childNodes` 列表中获取子节点的方法是使用 `item()` 方法：

```
var oHead = oHtml.childNodes.item(0);
var oBody = oHtml.childNodes.item(1);
```

HTML DOM 页定义了 `document.body` 作为指向 `<body>` 元素的指针：

```
var oBody = document.body;
```

有了 `oHtml`、`oHead` 和 `oBody` 这三个变量，就可以先尝试确定它们之间的关系：

```
alert(oHead.parentNode == oHtml); //outputs "true"
alert(oBody.parentNode == oHtml); //outputs "true"
alert(oBody.previousSibling == oHead); //outputs "true"
alert(oHead.nextSibling == oBody); //outputs "true"
alert(oHead.ownerDocument == document); //outputs "true"
```

这一小段代码测试并验证了 `oBody` 和 `oHead` 的 `parentNode` 特性都是指向 `oHtml` 变量，同时使用 `previousSibling` 和 `nextSibling` 特性来建立它们之间的关系。最后一行确认了 `oHead` 的 `ownerDocument` 特性事实上是指向该文档。

不同浏览器在判断何为 `Text` 节点上存在一些差异。某些浏览器，如 Mozilla，认为元素之间的空白都是 `Text` 节点；而另一些浏览器，如 IE，会全部忽略这些空白。因为使用 Mozilla 方式很难确定哪些空白是 `Text` 节点，本书将采用 IE 的方式。

6.3.2 检测节点类型

我们可以通过使用 `nodeType` 特性检验节点类型：

```
alert(document.nodeType); //outputs "9"
alert(document.documentElement.nodeType); //outputs "1"
```

这个例子中，`document.nodeType` 返回 9，等于 `Node.DOCUMENT_NODE`；同时 `document.documentElement.nodeType` 返回 1，等于 `Node.ELEMENT_NODE`。

也可以用 `Node` 常量来匹配这些值：

```
alert(document.nodeType == Node.DOCUMENT_NODE); //outputs "true"
alert(document.documentElement.nodeType == Node.ELEMENT_NODE); //outputs "true"
```

这段代码可以在 Mozilla 1.0+、Opera 7.0+ 和 Safari 1.0+ 上正常运行。不幸的是，IE 不支持这

些常量，所以这些代码在IE上会产生错误。所幸，可以通过定义匹配节点类型的常量来纠正这种情况，正如下面这样：

```
if (typeof Node == "undefined") {
    var Node = {
        ELEMENT_NODE: 1,
        ATTRIBUTE_NODE: 2,
        TEXT_NODE: 3,
        CDATA_SECTION_NODE: 4,
        ENTITY_REFERENCE_NODE: 5,
        ENTITY_NODE: 6,
        PROCESSING_INSTRUCTION_NODE: 7,
        COMMENT_NODE: 8,
        DOCUMENT_NODE: 9,
        DOCUMENT_TYPE_NODE: 10,
        DOCUMENT_FRAGMENT_NODE: 11,
        NOTATION_NODE: 12
    }
}
```

当然，另外一种选择是直接使用数值（不过这容易让人混淆，因为很多人无法记住这些节点类型的值）。

6.3.3 处理特性

正如前面提到的，即便Node接口已具有attributes方法，且已被所有类型的节点继承，然而，只有Element节点才能有特性。Element节点的attributes属性其实是NamedNodeMap，它提供一些用于访问和处理其内容的方法：

- `getNamedItem(name)`——返回nodeName属性值等于name的节点；
- `removeNamedItem(name)`——删除nodeName属性值等于name的节点；
- `setNamedItem(node)`——将node添加到列表中，按其nodeName属性进行索引；
- `item(pos)`——像NodeList一样，返回在位置pos的节点；

169

请记住这些方法都是返回一个Attr节点，而非特性值。

NamedNodeMap对象也有一个length属性来指示它所包含的节点的数量。

当NamedNodeMap用于表示特性时，其中每个节点都是Attr节点，它的nodeName属性被设置为特性名称，而nodeValue属性被设置为特性的值。例如，假设有这样一个元素：

```
<p style="color: red" id="p1">Hello world!</p>
```

同时，假设变量oP包含指向这个元素的一个引用。于是可以这样访问id特性的值：

```
var sId = oP.attributes.getNamedItem("id").nodeValue;
```

当然，还可以用数值方式访问id特性，但这样稍微有些不直观：

```
var sId = oP.attributes.item(1).nodeValue;
```

还可以通过给 `nodeValue` 属性赋新值来改变 `id` 特性：

```
oP.attributes.getNamedItem("id").nodeValue = "newId";
```

`Attr` 节点也有一个完全等同于（同时也完全同步于）`nodeValue` 属性的 `value` 属性，并且有 `name` 属性和 `nodeName` 属性保持同步。我们可以随意使用这些属性来修改或变更特性。

因为这个方法有些累赘，DOM 又定义了三个元素方法来帮助访问特性：

- `getAttribute(name)`——等于 `attributes.getNamedItem(name).value`；
- `setAttribute(name, newvalue)`——等于 `attribute.getNamedItem(name).value = newvalue`；
- `removeAttribute(name)`——等于 `attributes.removeNamedItem(name)`。

这些方法相当有用，可以直接处理特性值，完全地隐藏 `Attr` 节点。所以，要获取前面用的 `<p/>` 的 `id` 特性，只需这样做：

```
var sId = oP.getAttribute("id");
```

同时要更改 ID，可以这样做

```
oP.setAttribute("id", "newId");
```

正如你所看到的，这些方法要比使用 `NamedNodeMap` 的方法简洁得多。

6.3.4 访问指定节点

现在已经知道如何访问父节点和子节点，但是如果想访问文档中位置很深的某个节点（或者一组节点），要怎么做呢？当然，你不想逐个检查子节点直到遇到要访问的那个节点。为在这种情况下助你一臂之力，DOM 提供一些方法来方便地访问指定的节点。

1. `getElementsByName()`

核心（XML）DOM 定义了 `getElementsByName()` 方法，用来返回一个包含所有的 `tagName`（标签名）特性等于某个指定值的元素的 `NodeList`。在 `Element` 对象中，`tagName` 特性总是等于小于号之后紧随的名称——例如，`` 的 `tagName` 是 "img"。下一行代码返回文档中所有 `` 元素的列表：

```
var oImgs = document.getElementsByTagName("img");
```

在把所有图形都存于 `oImgs` 后，只需使用方括号标记或者 `item()` 方法 (`getElementsByName()` 返回一个和 `childNodes` 一样的 `NodeList`)，就可以像访问子节点那样逐个访问这些节点了：

```
alert(oImgs[0].tagName); //outputs "IMG"
```

这行代码输出第一个图像的 `tagName`（标签名），输出的是 "IMG"。由于某些原因，大部分浏览器按照大写来记录标签名，即使 XHTML 约定指出标签名应当全部小写：

但是假如你只想获取在某个页面第一个段落的所有图像呢？可以通过对第一个段落元素调用 `getElementsByName()` 来完成，像这样：

```
var oPs = document.getElementsByTagName("p");
var oImgsInP = oPs[0].getElementsByTagName("img");
```

可以使用一个星号的方法来获取 document 中的所有元素：

```
var oAllElements = document.getElementsByTagName("*");
```

这行代码可以返回 document 中包含的所有元素而不管它们的标签名。

当参数是一个星号的时候，IE 6.0 并不返回所有的元素。必须使用 `document.all` 来替代它。

2. `getElementsByName()`

HTML DOM 定义了 `getElementsByName()`，它用来获取所有 `name` 特性等于指定值的元素的。
考虑下面的 HTML：

```
<html>
  <head>
    <title>DOM Example</title>
  </head>
  <body>
    <form method="post" action="dosomething.cgi">
      <fieldset>
        <legend>What color do you like?</legend>
        <input type="radio" name="radColor" value="red" /> Red<br />
        <input type="radio" name="radColor" value="green" /> Green<br />
        <input type="radio" name="radColor" value="blue" /> Blue<br />
      </fieldset>
      <input type="submit" value="Submit" />
    </form>
  </body>
</html>
```

171

这个页面会询问用户喜欢哪种颜色。所有单选按钮都用同样的名称（`name` 特性），因为只要这个字段返回一个值（即选定的选项的 `value` 特性）即可。若要获得所有单选按钮元素的引用，可以使用下面的代码：

```
var oRadios = document.getElementsByName("radColor");
```

然后，就可以像处理其他元素那样处理这些单选按钮了：

```
alert(oRadios[0].getAttribute("value")); //outputs "red"
```

IE 6.0 和 Opera 7.5 在这个方法的使用上还存在一些错误。首先，它们还会返回 `id` 等于给定名称的元素。第二，它们仅仅检查 `<input/>` 和 `` 元素。

3. `getElementById()`

这是 HTML DOM 定义的第二种方法，它将返回 `id` 特性等于指定值的元素。在 HTML 中，`id` 特性是唯一的——这意味着没有两个元素可以共享同一个 `id`。毫无疑问这是从文档树中获取

单个指定元素最快的方法。

假设有下列 HTML 页面：

```
<html>
  <head>
    <title>DOM Example</title>
  </head>
  <body>
    <p>Hello World!</p>
    <div id="div1">This is my first layer</div>
  </body>
</html>
```

172

要访问 ID 为 "div1" 的 `<div />` 元素，可以使用 `getElementsByName()`：

```
var oDivs = document.getElementsByTagName("div");
var oDiv1 = null;
for (var i=0; i < oDivs.length; i++) {
  if (oDivs[i].getAttribute("id") == "div1") {
    oDiv1 = oDivs[i];
    break;
}
}
```

或者，可以使用 `getElementById()`：

```
var oDiv1 = document.getElementById("div1");
```

可以看到，这种获取指定元素的引用的方法效率更高。

如果给定的 ID 匹配某个元素的 `name` 特性，IE 6.0 还会返回这个元素。这是一个 bug，也是你必须非常小心的一个问题。

6.3.5 创建和操作节点

迄今为止，已经学过了如何访问文档中的不同节点，不过这仅仅是使用 DOM 所能实现的功能中的很小一部分。还能添加、删除、替换（或者其他操作）DOM 文档中的节点。正是这些功能使得 DOM 具有真正意义上的动态性。

1. 创建新节点

DOM Document（文档）中有一些方法用于创建不同类型的节点，即便在所有的浏览器中的浏览器 `document` 对象并不需要全部支持所有的方法。下面的表格列出了包含在 DOM Level 1 中的方法，并列出不同的浏览器是否支持项。

方 法	描 述	IE	MOZ	OP	SAF
<code>createAttribute(name)</code>	用给定名称 <code>name</code> 创建特性节点	×	×	×	-
<code>createCDATASection(text)</code>	用包含文本 <code>text</code> 的文本子节点创建一个 CDATA Section	-	×	-	-
<code>createComment(text)</code>	创建包含文本 <code>text</code> 的注释节点	×	×	×	×

(续)

方 法	描 述	IE	MOZ	OP	SAF	
createDocumentFragment()	创建文档碎片节点	×	×	×	×	
createElement(tagname)	创建标签名为 <i>tagname</i> 的元素	×	×	×	×	173
createEntityReference(name)	创建给定名称的实体引用节点	-	×	-	-	
createProcessingInstruction(target, data)	创建包含给定 <i>target</i> 和 <i>data</i> 的 PI 节点	-	×	-	-	
createTextNode(text)	创建包含文本 <i>text</i> 的文本节点	×	×	×	×	

注: IE = Windows 的 IE 6; MOZ = 任意平台的 Mozilla 1.5; OP=任意平台的 Opera 7.5; SAF=MacOS 的 Safari 1.2

最常用到的几个方法是: `createDocumentFragment()`、`createElement()`和`createTextNode()`; 其他的一些方法要么就是没什么用 (`createComment()`)，要么就是浏览器的支持不够，目前还不能用。

2. `createElement()`、`createTextNode()`、`appendChild()`

假设有如下 HTML 页面:

```
<html>
  <head>
    <title>createElement() Example</title>
  </head>
  <body>

  </body>
</html>
```

现在想使用 DOM 来添加下列代码到上面这个页面中:

```
<p>Hello World!</p>
```

这里可以使用 `createElement()`和`createTextNode()`来达到目的。下面是实现步骤:

首先，创建`<p>`元素:

```
var oP = document.createElement("p");
```

第二，创建文本节点:

```
var oText = document.createTextNode("Hello World!");
```

下一步，把文本节点加入到元素中。可以用在本章前面简要提到的 `appendChild()`方法来完成这个任务。每种节点类型都有 `appendChild()`方法，它的用途是将给定的节点添加到某个节点的 `childNodes` 列表的尾部。在这个例子中，应将文本节点追加到`<p>`元素中:

```
oP.appendChild(oText);
```

174

不过还没完成全部操作。已经创建了一个`<p>`元素和一个文本节点，并且将它们关联在一起了，但这个元素在文档中仍然没有一席之地。要实际可见，必须将这个元素附加到 `document.body`

元素或者其中任意子节点上。然后，可以再次使用 `appendChild()` 方法：

```
document.body.appendChild(oP);
```

要把这些代码放到可运行范例中，只要创建一个包含每一步的函数，并且使用 `onload` 事件句柄在页面载入后调用这个函数（关于事件将在第 9 章详细介绍）：

```
<html>
  <head>
    <title>createElement() Example</title>
    <script type="text/javascript">
      function createMessage() {
        var oP = document.createElement("p");
        var oText = document.createTextNode("Hello World! ");
        oP.appendChild(oText);
        document.body.appendChild(oP);
      }
    </script>
  </head>
  <body onload="createMessage()">
    </body>
</html>
```

当运行这段代码时，“Hello World!”的消息会自动显示，就好像它本来就是这个 HTML 文档的一部分。

在这里，我必须谨慎地告诉你所有的 DOM 操作必须在页面完全载入之后才能进行。当页面正在载入时，要向 DOM 插入相关代码是不可能的，因为在页面完全下载到客户端机器之前，是无法完全构建 DOM 树的。因为这个原因，必须使用 `onload` 事件句柄来执行所有的代码。

3. `removeChild()`、`replaceChild()` 和 `insertBefore()`

自然的，可以添加一个节点，当然也可以删除一个节点，这就是 `removeChild()` 所要做的事。这个方法接受一个参数，要删除的节点，然后将这个节点作为函数的返回值返回。所以，例如，如果有已经包含“Hello World!”消息的页面，要把这个消息删除，可以使用类似下面方法：

```
<html>
  <head>
    <title>removeChild() Example</title>
    <script type="text/javascript">
      function removeMessage() {
        var oP = document.body.getElementsByTagName("p")[0];
        document.body.removeChild(oP);
      }
    </script>
  </head>
  <body onload="removeMessage()">
    <p>Hello World!</p>
  </body>
</html>
```

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

这个页面载后，显示空白页面，因为在你能警到消息之前，它已经被删除了。尽管它能运行，最好还是使用节点的 `parentNode` 特性来确保每次你都能访问到它真正的父节点：

```
<html>
  <head>
    <title>removeChild() Example</title>
    <script type="text/javascript">
      function removeMessage() {
        var oP = document.body.getElementsByName("p")[0];
        oP.parentNode.removeChild(oP);
      }
    </script>
  </head>
  <body onload="removeMessage()">
    <p>Hello World!</p>
  </body>
</html>
```

但假如想将这个消息替换成新的内容，要怎么做呢？如果是这种情况，则可以使用 `replaceChild()` 方法。

`replaceChild()` 方法有两个参数：被添加的节点和被替换的节点。这样，可以创建一个包含新消息的元素，并用它替换原来包含 "Hello World!" 消息的 `<p>` 元素。

```
<html>
  <head>
    <title>replaceChild() Example</title>
    <script type="text/javascript">
      function replaceMessage() {
        var oNewP = document.createElement("p");
        var oText = document.createTextNode("Hello Universe! ");
        oNewP.appendChild(oText);
        var oOldP = document.body.getElementsByName("p")[0];
        oOldP.parentNode.replaceChild(oNewP, oOldP);
      }
    </script>
  </head>
  <body onload="replaceMessage()">
    <p>Hello World!</p>
  </body>
</html>
```

176

这个范例页面将消息 "Hello World!" 替换成 "Hello Universe!" 注意这段代码仍然使用 `parentNode` 特性来确保使用了正确父节点来进行操作。

当然，可能想让两个消息同时出现。如果想让新消息出现在老消息之后，只要使用 `appendChild()` 方法：

```
<html>
  <head>
    <title>appendChild() Example</title>
```

```

<script type="text/javascript">
    function appendMessage() {
        var oNewP = document.createElement("p");
        var oText = document.createTextNode("Hello Universe! ");
        oNewP.appendChild(oText);
        document.body.appendChild(oNewP);
    }
</script>
</head>
<body onload="appendMessage()">
    <p>Hello World!</p>
</body>
</html>

```

然而，如果想让新消息出现在旧消息之前，就使用 `insertBefore()` 方法。这个方法接受两个参数：要添加的节点和插在哪个节点之前。在这个例子中，第二个参数是包含 "Hello World!" 的 `<p>` 元素：

```

<html>
    <head>
        <title>insertBefore() Example</title>
        <script type="text/javascript">
            function insertMessage() {
                var oNewP = document.createElement("p");
                var oText = document.createTextNode("Hello Universe! ");
                oNewP.appendChild(oText);
                var oOldP = document.getElementsByTagName("p")[0];
                document.body.insertBefore(oNewP, oOldP);
            }
        </script>
    </head>
    <body onload="insertMessage()">
        <p>Hello World!</p>
    </body>
</html>

```

4. `createDocumentFragment()`

一旦把节点添加到 `document.body`（或者它的后代节点）中，页面就会更新并反映出这个变化。对于少量的更新，这是很好的，就像在前面的例子中那样。然而，当要向 `document` 添加大量数据时，如果逐个添加这些变动，这个过程有可能会十分缓慢。为解决这个问题，可以创建一个文档碎片，把所有的新节点附加其上，然后把文档碎片的内容一次性添加到 `document` 中。

假设你想创建十个新段落。若使用前面学到的方法，可能会写出这种代码：

```

var arrText = ["first", "second", "third", "fourth", "fifth", "sixth", "seventh",
    "eighth", "ninth", "tenth"];

for (var i=0; i < arrText.length; i++) {
    var oP = document.createElement("p");
    var oText = document.createTextNode(arrText[i]);
    oP.appendChild(oText);
    document.body.appendChild(oP);
}

```

这段代码运行良好，但问题是它调用了十次 `document.body.appendChild()`，每次都要产生一次页面刷新。这时，文档碎片就十分有用：

```
var arrText = ["first", "second", "third", "fourth", "fifth", "sixth", "seventh",
    "eighth", "ninth", "tenth"];

var oFragment = document.createDocumentFragment();

for (var i=0; i < arrText.length; i++) {
    var oP = document.createElement("p");
    var oText = document.createTextNode(arrText[i]);
    oP.appendChild(oText);
    oFragment.appendChild(oP);
}

document.body.appendChild(oFragment);
```

在这段代码中，每个新的`<p>`元素都被添加到文档碎片中。然后，这个碎片被作为参数传递给 `appendChild()`。这里对 `appendChild()` 的调用实际上并不是把文档碎片节点本身追加到 `<body>` 元素中；而是仅仅追加碎片中的子节点。然后，可以看到很明显的性能提升：调用 `document.body.appendChild()` 一次来替代十次，这意味着只需要进行一次屏幕刷新。

6.4 HTML DOM 特征功能

核心 DOM 的特性和方法是通用的，是为了在各种情况下操作所有 XML 文档而设计的。HTML DOM 的特性和方法是在专门针对 HTML 的同时也让一些 DOM 操作更加简便。这包括将特性作为属性进行访问的能力，以及特定于元素的属性和方法，这些扩展可以完成一些常见的任务，例如搭建表格，更加简便快速。

6.4.1 让特性像属性一样

大部分情况下，HTML DOM 元素中包含的所有特性都是可作为属性。例如，假设有如下图像元素：

```

```

178

如果要使用核心的 DOM 来获取和设置 `src` 和 `border` 特性，那么要用 `getAttribute()` 和 `setAttribute()` 方法：

```
alert(oImg.getAttribute("src"));
alert(oImg.getAttribute("border"));
oImg.setAttribute("src", "mypicture2.jpg");
oImg.setAttribute("border", "1");
```

然而，使用 HTML DOM，可以使用同样名称的属性来获取和设置这些值：

```
alert(oImg.src);
alert(oImg.border);
oImg.src = "mypicture2.jpg";
oImg.border = "1";
```

唯一的特性名和属性名不一样的特例是 `class` 特性，它是用来指定应用于某个元素的一个 CSS 类，例如：

```
<div class="header"></div>
```

因为 `class` 在 ECMAScript 中是一个保留字，在 JavaScript 中，它不能被作为变量名、属性名或者函数名。于是，相应的属性名就变成 `className`：

```
alert(oDiv.className);
oDiv.className = "footer";
```

通过使用属性来访问特性的方式来替代 `getAttribute()` 和 `setAttribute()` 并无实质性的益处，除了可能缩减代码的长度以及令代码变得易读。

IE 在 `setAttribute()` 上有个很大的问题：当你使用它时，变更并不会总是正确地反应出来。如果你打算支持 IE，最好尽可能使用属性。

6.4.2 table 方法

假设想使用 DOM 来创建如下的 HTML 表格：

```
<table border="1" width="100%">      <tbody>
    <tr>
        <td>Cell 1,1</td>
        <td>Cell 2,1</td>
    </tr>
    <tr>
        <td>Cell 1,2</td>
        <td>Cell 2,2</td>
    </tr>
</tbody>
</table>
```

179

如果想通过核心 DOM 方法来完成这个任务，你的代码可能会像这样：

```
//create the table
var oTable = document.createElement("table");
oTable.setAttribute("border", "1");
oTable.setAttribute("width", "100%");

//create the tbody
var oTBody = document.createElement("tbody");
oTable.appendChild(oTBody);

//create the first row
var oTR1 = document.createElement("tr");
oTBody.appendChild(oTR1);
var oTD11 = document.createElement("td");
oTD11.appendChild(document.createTextNode("Cell 1,1"));
oTR1.appendChild(oTD11);
var oTD21 = document.createElement("td");
```

```

oTD21.appendChild(document.createTextNode("Cell 2,1"));
oTR1.appendChild(oTD21);

//create the second row
var oTR2 = document.createElement("tr");
oTBody.appendChild(oTR2);
var oTD12 = document.createElement("td");
oTD12.appendChild(document.createTextNode("Cell 1,2"));
oTR2.appendChild(oTD12);
var oTD22 = document.createElement("td");
oTD22.appendChild(document.createTextNode("Cell 2,2"));
oTR2.appendChild(oTD22);
//add the table to the document body
document.body.appendChild(oTable);

```

这段代码十分的冗长而且有些难于理解。为了协助建立表格，HTML DOM 给<table>、<tbody>和<tr>等元素添加了一些特性和方法。

给<table>元素添加了以下内容：

- caption——指向<caption>元素（如果存在）；
- tBodies——<tbody>元素的集合；
- tFoot——指向<tfoot>元素（如果存在）；
- tHead——指向<thead>元素（如果存在）；
- rows——表格中所有行的集合；
- createTHead()——创建<thead>元素并将其放入表格；
- createTFoot()——创建<tfoot>元素并将其放入表格；
- createCaption()——创建<caption>元素并将其放入表格；
- deleteTHead()——删除<thead>元素；
- deleteTFoot()——删除<tfoot>元素；
- deleteCaption()——删除<caption>元素；
- deleteRow(position)——删除指定位置上的行；
- insertRow(position)——在 rows 集合中的指定位置上插入一个新行。

180

<tbody>元素添加了以下内容：

- rows——<tbody>中所有行的集合；
- deleteRow(position)——删除指定位置上的行；
- insertRow(position)——在 rows 集合中的指定位置上插入一个新行。

<tr>元素中添加了以下内容：

- cells——<tr>元素中所有的单元格的集合；
- deleteCell(position)——删除给定位置上的单元格；
- insertCell(position)——在 cells 集合的给定位置上插入一个新的单元格。

上面的一切意味着什么？简单地说，它意味着如果使用这些简便的属性和方法就可以大大降低创建表格的复杂度：

```

//create the table
var oTable = document.createElement("table");
oTable.setAttribute("border", "1");
oTable.setAttribute("width", "100%");

//create the tbody
var oTBody = document.createElement("tbody");
oTable.appendChild(oTBody);
//create the first row
oTBody.insertRow(0);
oTBody.rows[0].insertCell(0);
oTBody.rows[0].cells[0].appendChild(document.createTextNode("Cell 1,1"));
oTBody.rows[0].insertCell(1);
oTBody.rows[0].cells[1].appendChild(document.createTextNode("Cell 2,1"));

//create the second row
oTBody.insertRow(1);
oTBody.rows[1].insertCell(0);
oTBody.rows[1].cells[0].appendChild(document.createTextNode("Cell 1,2"));
oTBody.rows[1].insertCell(1);
oTBody.rows[1].cells[1].appendChild(document.createTextNode("Cell 2,2"));
//add the table to the document body
document.body.appendChild(oTable);

```

在这段代码中，创建`<table>`和`<tbody>`元素的方式没有改变。在这一段中改变的是如何创建两个表格行，用到了HTML DOM Table（表格）的属性和方法。要创建第一行，对`<tbody>`元素调用`insertRow()`方法，并传给它一个参数0，表示新增的一行应该放在什么位置上。然后，可以通过`oTBody.rows[0]`来引用新增的这一行，因为这一行已经被自动创建并添加到`<tbody>`元素中的第0个位置。

以类似的方式可以创建单元格——对`<tr>`元素调用`insertCell()`并传入要创建单元格的位置。可以通过`oTBody.rows[0].cells[0]`来引用新创建的这个单元格，因为单元格已经被创建并插入到这一行的第0个位置。

虽然从技术角度上来说，两种代码都是正确的，但是使用这些特性和方法来创建表格使得代码变得更加有逻辑且更加易读。

6.5 遍历 DOM

到目前为止，我们讨论的功能都仅仅是DOM Level 1的部分。本节将介绍一些DOM Level 2的功能，尤其是和遍历DOM文档相关的DOM Level 2遍历（traversal）和范围（range）规范中的对象。这些功能只有在Mozilla和Konqueror/Safari中才有。

6.5.1 NodeIterator

第一个有关的对象是NodeIterator，用它可以对DOM树进行深度优先的搜索，如果要查找页面中某个特定类型的信息（或者元素），这是相当有用的。要理解NodeIterator到底做了什么，考虑下面的HTML页面：

```

<html>
  <head>
    <title>Example</title>
  </head>
  <body>
    <p>Hello <b>World!</b></p>
  </body>
</html>

```

这个页面可以转换成由图 6-2 表示的 DOM 树。

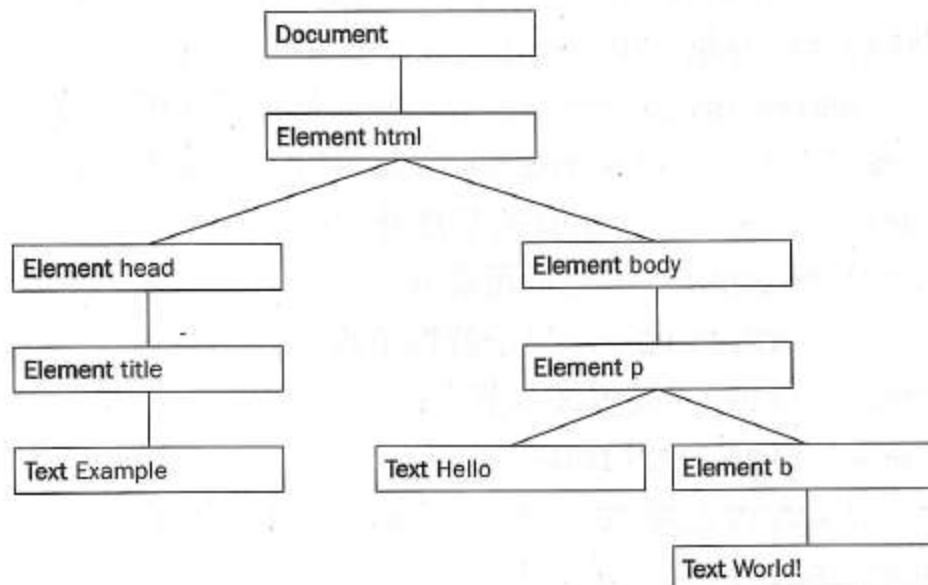


图 6-2

当使用 `NodeIterator` 时，可以从 `document` 元素（`<html/>`）开始，按照一种系统的路径——也就是大家熟知的深度优先搜索——遍历整个 DOM 树。在这种搜索方式中，遍历从父节点开始，到子节点，再到子节点的子节点，如此继续，尽可能往深入，直到不能再往下走为止。然后，遍历过程向上回退一层，并进入下一个子节点。例如，在前面展示的 DOM 树中，遍历过程在回退到`<body/>`前，先访问了`<html/>`，然后`<head/>`，然后`<title/>`，然后是文本节点 "example"。图 6-3 显示了这个遍历过程的完整路径。

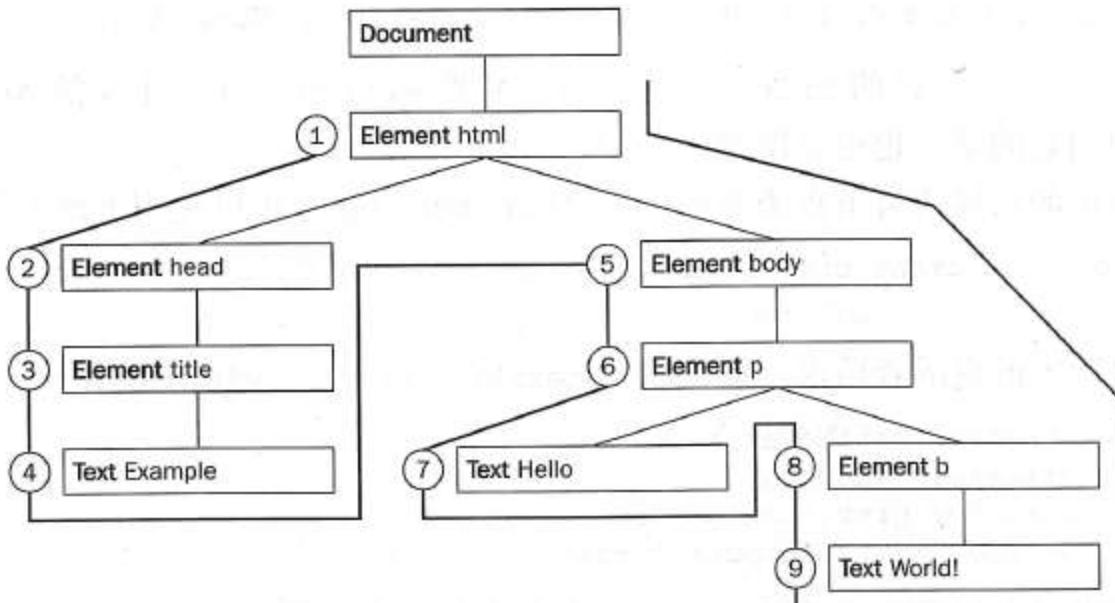


图 6-3

182
183

最佳的思考深度优先搜索的方法就是从第一个节点左边的第一个节点起沿着树的最外沿画线。只要这条线从左侧经过某个节点，那么这个节点就是搜索中下一个出现的节点（图 6-3 中的粗线代表了这条线）。

要创建 `NodeIterator` 对象，请使用 `document` 对象的 `createNodeIterator()` 方法。这个方法接受四个参数：

- (1) `root`——从树中开始搜索的那个节点。
- (2) `whatToShow`——一个数值代码，代表哪些节点需要访问。
- (3) `filter`——`NodeFilter` 对象，用来决定需要忽略哪些节点。
- (4) `entityReferenceExpansion`——布尔值，表示是否需要扩展实体引用。

通过应用下列一个或多个常量，`whatToShow` 参数可以决定哪些节点可以访问：

- `NodeFilter.SHOW_ALL`——显示所有的节点类型；
- `NodeFilter.SHOW_ELEMENT`——显示元素节点；
- `NodeFilter.SHOW_ATTRIBUTE`——显示特性节点；
- `NodeFilter.SHOW_TEXT`——显示文本节点；
- `NodeFilter.SHOW_CDATA_SECTION`——显示 CData section 节点；
- `NodeFilter.SHOW_ENTITY_REFERENCE`——显示实体引用节点；
- `NodeFilter.SHOW_ENTITY`——显示实体节点；
- `NodeFilter.SHOW_PROCESSING_INSTRUCTION`——显示 PI 节点；
- `NodeFilter.SHOW_COMMENT`——显示注释节点；
- `NodeFilter.SHOW_DOCUMENT`——显示文档节点；
- `NodeFilter.SHOW_DOCUMENT_TYPE`——显示文档类型节点；
- `NodeFilter.SHOW_DOCUMENT_FRAGMENT`——显示文档碎片节点；
- `NodeFilter.SHOW_NOTATION`——显示记号节点。

可以通过使用二进制或操作符来组合多个值：

```
var iWhatToShow = NodeFilter.SHOW_ELEMENT | NodeFilter.SHOW_TEXT;
```

`createNodeIterator()` 的 `filter`（过滤器）参数可以指定一个自定义的 `NodeFilter` 对象，但是如果不想使用它的话，也可以留空 (`null`)。

要创建最简单的访问所有节点类型的 `NodeIterator` 对象，可以使用下面的代码：

```
var iterator = document.createNodeIterator(document, NodeFilter.SHOW_ALL, null, false);
```

要在搜索过程中前进或者后退，可以使用 `nextNode()` 和 `previousNode()` 方法：

```
var node1 = iterator.nextNode();
var node2 = iterator.nextNode();
var node3 = iterator.previousNode();
alert(node1 == node3); //outputs "true"
```

例如，假设想列出某个区域内指定`<div>`中包含的所有元素。下列代码可以完成这个任务：

184

```

<html>
  <head>
    <title>NodeIterator Example</title>
    <script type="text/javascript">

      var iterator = null;

      function makeList() {
        var oDiv = document.getElementById("div1");
        iterator = document.createNodeIterator(oDiv,
        NodeFilter.SHOW_ELEMENT, null, false);

        var oOutput = document.getElementById("text1");
        var oNode = iterator.nextNode();
        while (oNode) {
          oOutput.value += oNode.tagName + "\n";
          oNode = iterator.nextNode();
        }

      }

    </script>
  </head>
  <body>
    <div id="div1">
      <p>Hello <b>World!</b></p>
      <ul>
        <li>List item 1</li>
        <li>List item 2</li>
        <li>List item 3</li>
      </ul>
    </div>
    <textarea rows="10" cols="40" id="text1"></textarea><br />
    <input type="button" value="Make List" onclick="makeList()" /> </body>
</html>

```

当点击了按钮后，将使用包含在 div1 中的元素的标签名来填充<textarea/>：

P
B
UL
LI
LI
LI

但假设不想在结果中包含<p/>元素。这就不能仅使用 whatToShow 参数来完成。这种情况下，你需要自定义一个 NodeFilter 对象。

NodeFilter 对象只有一个方法：acceptNode()。如果应该访问给定的节点，那么该方法返回 NodeFilter.FILTER_ACCEPT；如果不应该访问给定节点，则返回 NodeFilter.FILTER_REJECT。然而，不能使用 NodeFilter 类来创建这个对象，因为这个类是一个抽象类。在 Java 或一些其他的语言中，必须重新定义一个 NodeFilter 的子类，不过，因为这是在 JavaScript 中，就不必这么做了。

现在，只要创建任意一个有acceptNode()方法的对象，就可以将它传给createNodeIterator()方法，如下所示：

```
var oFilter = new Object;
oFilter.acceptNode = function (oNode) {
    //filter logic goes here
};
```

若要禁止<p/>元素节点，只需检查 tagName 属性，如果它等于“P”，就返回 NodeFilter.FILTER_REJECT：

```
var oFilter = new Object;
oFilter.acceptNode = function (oNode) {
    return (oNode.tagName == "P") ? NodeFilter.FILTER_REJECT :
        NodeFilter.FILTER_ACCEPT;
};
```

如果将这段代码包含在前面的例子中，代码如下：

```
<html>
    <head>
        <title>NodeIterator Example</title>
        <script type="text/javascript">

            var iterator = null;

            function makeList() {
                var oDiv = document.getElementById("div1");
                var oFilter = new Object;
                oFilter.acceptNode = function (oNode) {
                    return (oNode.tagName == "P") ?
                        NodeFilter.FILTER_REJECT : NodeFilter.FILTER_ACCEPT;
                };

                iterator = document.createNodeIterator(oDiv,
                    NodeFilter.SHOW_ELEMENT, oFilter, false);

                var oOutput = document.getElementById("text1");
                var oNode = iterator.nextNode();
                while (oNode) {
                    oOutput.value += oNode.tagName + "\n";
                    oNode = iterator.nextNode();
                }
            }

        </script>
    </head>
    <body>
        <div id="div1">
            <p>Hello <b>World!</b></p>
            <ul>
                <li>List item 1</li>
                <li>List item 2</li>
                <li>List item 3</li>
```

```

</ul>
</div>
<textarea rows="10" cols="40" id="text1"></textarea><br />
<input type="button" value="Make List" onclick="makeList()" />      </body>
</html>

```

当这次再点击按钮时，`<textarea/>`中就会出现下列内容：

```

UL
LI
LI
LI

```

注意“P”和“B”都没有出现在列表中。这是因为排除`<p/>`元素后，不仅从迭代搜索中去掉了它，也去掉了它的所有后代节点。因为``是`<p/>`的一个子节点，所以它也被跳过。

`NodeIterator`对象展示了一种有序的自顶向下遍历整个DOM树（或者仅仅其中一部分）的方式。然而可能想遍历到树的特定区域时，再看看某个节点的兄弟节点或者子节点。如果是这种情况，可以使用`TreeWalker`。

6.5.2 TreeWalker

`TreeWalker`有点像`NodeIterator`的大哥：它有`NodeIterator`所有的功能(`nextNode()`和`previousNode()`)，并且添加了一些遍历方法：

- `parentNode()`——进入当前节点的父节点；
- `firstChild()`——进入当前节点的第一个子节点；
- `lastChild()`——进入当前节点的最后一个子节点；
- `nextSibling()`——进入当前节点的下一个兄弟节点；
- `previousSibling()`——进入当前节点的前一个兄弟节点。

要开始使用`TreeWalker`，其实完全可以像使用`NodeIterator`那样，只要把`createNodeIterator()`的调用改为调用`createTreeWalker()`，这个函数接受同样的参数：

```

<html>
  <head>
    <title>TreeWalker Example</title>
    <script type="text/javascript">

      var walker = null;

      function makeList() {
        var oDiv = document.getElementById("div1");
        var oFilter = new Object;
        oFilter.acceptNode = function (oNode) {
          return (oNode.tagName == "P") ?
            NodeFilter.FILTER_REJECT : NodeFilter.FILTER_ACCEPT;
        };

        walker = document.createTreeWalker(oDiv, NodeFilter.SHOW_ELEMENT,
        oFilter, false);
      }
    </script>
  </head>
  <body>
    <ul>
      <li>A</li>
      <li>B</li>
      <li>C</li>
    </ul>
    <div id="div1">
      <p>D</p>
      <ul>
        <li>E</li>
        <li>F</li>
      </ul>
    </div>
    <input type="button" value="Make List" onclick="makeList()" />
  </body>
</html>

```

```

        var oOutput = document.getElementById("text1");
        var oNode = walker.nextSibling();
        while (oNode) {
            oOutput.value += oNode.tagName + "\n";
            oNode = walker.nextSibling();
        }

    }

</script>
</head>
<body>
    <div id="div1">
        <p>Hello <b>World!</b></p>
        <ul>
            <li>List item 1</li>
            <li>List item 2</li>
            <li>List item 3</li>
        </ul>
    </div>
    <textarea rows="10" cols="40" id="text1"></textarea><br />
    <input type="button" value="Make List" onclick="makeList()" />      </body>
</html>

```

自然，这样一个简单的例子不能展示 TreeWalker 的真正能力。当不想直接翻遍整个 DOM 树时，TreeWalker 十分有用。例如，假设只想访问前面所显示的 HTML 页面中的``元素。可以写一个只接受标签名为“LI”的元素的过滤器，而这里，可以使用 TreeWalker 来进行更有目的性的遍历：

```

<html>
    <head>
        <title>TreeWalker Example</title>
        <script type="text/javascript">

            var walker = null;

            function makeList() {
                var oDiv = document.getElementById("div1");

                walker = document.createTreeWalker(oDiv, NodeFilter.SHOW_ELEMENT,
                    null, false);
                var oOutput = document.getElementById("text1");
                walker.firstChild(); //go to <p>
                walker.nextSibling(); //go to <ul>
                var oNode = walker.firstChild(); //go to first <li>
                while (oNode) {
                    oOutput.value += oNode.tagName + "\n";
                    oNode = walker.nextSibling();
                }

            }

        </script>

```

```

</head>
<body>
  <div id="div1">
    <p>Hello <b>World!</b></p>
    <ul>
      <li>List item 1</li>
      <li>List item 2</li>
      , <li>List item 3</li>
    </ul>
  </div>
  <textarea rows="10" cols="40" id="text1"></textarea><br />
  <input type="button" value="Make List" onclick="makeList()" />
</body>
</html>

```

在这个例子中，创建了 TreeWalker 并立刻调用了 `firstChild()` 方法，将 walker 指到 `<p>` 元素（因为 `<p>` 是 `div1` 的第一个子节点）。在接下来的一行里调用 `nextSibling()` 后，walker 指到了 `<p>` 的下一个兄弟节点 `` 上。然后，调用 `firstChild()` 返回到 `` 下的第一个 `` 元素。接下来，在循环内部，通过使用 `nextSibling()` 方法对剩下的 `` 元素进行迭代。

点击按钮后，应该会看到如下输出：

```

LI
LI
LI

```

最后要说的是，如果对将要遍历的 DOM 树的结构有所了解的话，TreeWalker 更加有用；而同时，如果并不知道这个结构的话，使用 `NodeIterator` 更加实际有效。

6.6 测试与 DOM 标准的一致性

现在你肯定可以说出 DOM 的许多方面。正因如此，你需要一种方法来确定给定的 DOM 实现到底支持 DOM 的哪些部分。有趣的是，这个对象就叫做 `implementation`。

`implementation` 对象是 DOM 文档的一个特性，因此，也是浏览器 `document` 对象的一部分。`implementation` 唯一的方法是 `hasFeature()`，它接受两个参数：要检查的特征和特征的版本。例如，如果想检查对 XML DOM Level 1 的支持，可以这样调用：

```
var bXmlLevel1 = document.implementation.hasFeature("XML", "1.0");
```

下面的表格列出了所有的 DOM 特征以及相应需要检查的版本：

189

特 征	支持的版本	描 述
Core	1.0, 2.0, 3.0	基本的 DOM，给予了用层次树来表示文档的能力
XML	1.0, 2.0, 3.0	核心的 XML 扩展，增加了对 CDATA Section、处理指令和实体的支持
HTML	1.0, 2.0	XML 的 HTML 扩展，增加了对 HTML 特定元素和实体的支持
Views	2.0	基于特定样式完成对文档的格式化
StyleSheets	2.0	为文档关联样式表
CSS	2.0	支持级联样式表 1 (CSS Level 1)

(续)

特征	支持的版本	描述
CSS2	2.0	支持级联样式表 2 (CSS Level 2)
Events	2.0	通用 DOM 事件
UIEvents	2.0	用户界面事件
MouseEvents	2.0	由鼠标引起的事件 (点击、鼠标经过, 等等)
MutationEvents	2.0	当 DOM 树发生改变时引发的事件
HTMLEvents	2.0	HTML 4.01 的事件
Range	2.0	操作 DOM 树中某个特定范围的对象和方法
Traversal	2.0	遍历 DOM 树的方法
LS	3.0	在文件和 DOM 树之间同步地载入和存储
LS-Async	3.0	在文件和 DOM 树之间异步地载入和存储
Validation	3.0	用于修改 DOM 树之后仍然保持其有效性的方法

尽管这个相当方便, 但是, 使用 `implementation.hasFeature()` 有其明显的缺陷——决定 DOM 实现是否对 DOM 标准的不同的部分相一致的, 正是去进行实现的人 (或公司)。
190 要让这个方法对于任何值都返回 `true`, 那是很简单的, 但这并不一定表示这个 DOM 实现真的和所有的标准都一致了。目前为止, 最精确的浏览器要数 Mozilla, 但它多少也有一些并不完全和 DOM 标准一致的地方, 这个方法却返回为 `true`。

6.7 DOM Level 3

2004 年 4 月, DOM Level 3 作为 W3C 的一个推荐标准被提出。目前为止, 还没有哪个浏览器已经完全实现该标准, 只有 Mozilla 已实现了一部分。还不知道 Web 浏览器要用多久才能把剩下的 DOM 特性加上去, 因为 IE 近四年都没有重大的更新了 (这里的意思是它对 DOM 的支持程度没有什么变化)。Mozilla 许诺将尽可能与标准保持兼容, 它仍然继续在对 DOM 支持方面遥遥领先。然而, Opera 重写了它的核心浏览器组件来更好地支持 DOM 标准, 并有一种和最新科技齐头并进的势头。甚至 Apple 的 Safari 浏览器 (基于 Konqueror 的) 也在不断跟进, 有计划地尽可能多地实现 DOM 功能。

6.8 小结

本章介绍了文档对象模型 (DOM) 的基本接口。了解到 DOM 如何将基于 XML 的文档组织成由任意节点组成的层次树。还学习了在文档中出现的不同的节点类型以及如何处理、添加、删除 DOM 树中的节点。

另外, 这一章还讲述了针对 HTML DOM 的特征功能, 如将一些特性转为对象属性和一些针对表格的方法, 这些功能让编写 HTML 与传统的 DOM 方式相比更加简单。

最后, 还学习了 DOM 遍历规范中的 `NodeIterator` 和 `Treewalker` 对象, 通过它们按照合理的方式遍历 DOM 树。

唯一没有在本章中介绍的 DOM 的主要部分是事件和事件处理, 这些将在第 9 章中详细介绍。

正则表达式

以前，测试字符串中是否存在某些特定形式是项很艰巨的任务。这常用到 `charAt()` 和 `indexOf()` 之类的字符串函数。Perl 等语言实现了称为正则表达式的解决方案，它是基于 `grep` 和 `ed` 这些 Unix 管理工具发展而来的。

正则表达式是具有特殊语法的字符串，用来表示指定字符或字符串在另一个字符串中出现的情况。这些模式字符串，有的十分简单，有的十分复杂，它们可以实现很多功能，从删除字符串中的空格到验证信用卡号的有效性等等。

JavaScript 内置支持正则表达式已经很久了，甚至要久于一些高级语言，如 Java（在 JDK1.4 中才引入了直接的正则表达式支持）。因为正则表达式用法十分复杂，可以用一本书来讲述它；所以，我们专辟一章了解 JavaScript 是如何实现正则表达式的。

7.1 正则表达式支持

JavaScript 对正则表达式的支持是通过 ECMAScript 中的 `RegExp` 类实现的。`RegExp` 对象的构造函数可以带一个或两个参数。第一个参数（或只有一个参数）是描述需要进行匹配的模式字符串；如果还有第二个参数，这个参数则指定了额外的处理指令。

最基本的正则表达式就是普通的字符串。例如，要匹配单词 "cat"，可以这样定义正则表达式：

```
var reCat = new RegExp("cat");
```

193

这个正则表达式只会匹配字符串中出现的第一个单词 "cat"，而且，它是区分大小写的。如果要让这个正则表达式匹配所有出现的 "cat"，需要给构造函数添加第二个参数：

```
var reCat = new RegExp("cat", "g");
```

在这一行代码里，第二个参数 "g" 是 `global` 的缩写，表示要搜索字符串中出现的全部 "cat"，而不是在找到第一个匹配后就停止。如果还要让模式不区分大小写，可以给第二个参数添加字符 "i" ("i" 是 `case-insensitive` 中 `insensitive` 的缩写)：

```
var reCat = new RegExp("cat", "gi");
```

有些正则表达式字面量使用 Perl 风格的语法：

```
var reCat = /cat/gi;
```

正则表达式字面量由一条斜线开始，跟着是字符串模式，然后是另一条斜线。如果还要指定额外的处理指令，如“g”和“i”，直接跟在第二个斜线的后面（就像前面的例子中那样）。

7.1.1 使用 RegExp 对象

创建一个 RegExp 对象后，把它应用到字符串上。RegExp 和 String 的一些方法都可使用。

首先用正则表达式要做的可能是判断某个字符串是否匹配指定的模式。最简单的情况，RegExp 有个 test() 方法，如果给定字符串（只有一个参数）匹配这个模式，它就返回 true，否则返回 false：

```
var sToMatch = "cat";
var reCat = /cat/;
alert(reCat.test(sToMatch)); //outputs "true"
```

在这个例子中，警告对话框输出“true”，因为模式匹配字符串。即时模式只在字符串中出现一次，也认为是一个匹配，test() 将返回真。但如果想访问模式的每一次出现呢？可以使用 exec() 方法。

RegExp 的 exec() 方法，有一个字符串参数，返回一个数组。数组中的第一个条目是第一个匹配；其他的是反向引用（这个稍后再讨论）：

```
var sToMatch = "a bat, a Cat, a fAt baT, a faT cat";
var reAt = /at/;
var arrMatches = reAt.exec(sToMatch);
```

这里，arrMatches 只包含一个条目：第一个“at”的实例（在单词“bat”中的那个）。但是为什么这样只能返回包含一个条目的数组呢？假设想获得某个模式所有出现呢？这里需要用到另外一种方法。

String 对象有个 match() 方法，它会返回一个包含在字符串中的所有匹配的数组。这个方法调用 string 对象，同时传给它一个 RegExp 对象：

```
var sToMatch = "a bat, a Cat, a fAt baT, a faT cat";
var reAt = /at/gi;
var arrMatches = sToMatch.match(reAt);
```

这段代码将按“at”在字符串 sToMatch 中出现的顺序在 arrMatches 中填入所有“at”的实例。这里是代码执行完毕后 arrMatches 数组中的内容：

下 标	值	来 自
0	"at"	"bat"
1	"at"	"Cat"
2	"At"	"fAt"
3	"aT"	"baT"
4	"aT"	"faT"
5	"at"	"cat"

另一个叫做 `search()` 的字符串方法的行为与 `indexOf()` 有些类似，但是它使用一个 `RegExp` 对象而非仅仅一个子字符串。`Search()` 方法返回在字符串中出现的一个匹配的位置。

```
var sToMatch = "a bat, a Cat, a fAt baT, a faT cat";
var reAt = /at/gi;
alert(sToMatch.search(reAt)); //outputs "3"
```

在这个例子中，警告对话框输出“3”，因为“at”第一次在字符串中出现的位置是3。值得注意的是，全局匹配正规表达式（带参数 `g`）在使用 `search()` 时不起作用。

7.1.2 扩展的字符串方法

在本书前面讨论过的两个字符串方法，其实也可以接受正则表达式作为参数。第一个方法是 `replace()`，它可以用另一个字符串（第二个参数）来替换某个子串（第一个参数）的所有匹配。例如：

```
var sToChange = "The sky is red. ";
alert(sToChange.replace("red", "blue")); //outputs "The sky is blue."
```

这里，子字符串“red”被字符串“blue”替代，最终输出“The sky is blue.”。事实上，也可以给第一个参数传递一个正则表达式：

```
var sToChange = "The sky is red. ";
var reRed = /red/;
alert(sToChange.replace(reRed, "blue")); //outputs "The sky is blue."
```

这段代码和前面的例子的结果一样，输出“The sky is blue.”。

也可以指定一个函数作为 `replace()` 的第二个参数。这个函数可以接受一个参数，即匹配了的文本，并返回应当进行替换的文本。例如：

```
var sToChange = "The sky is red. ";
var reRed = /red/;
var sResultText = sToChange.replace(reRed, function(sMatch) {
    return "blue";
});

alert(sResultText); //outputs "The sky is blue."
```

在这个例子中，在函数中的 `sMatch` 的值总为“red”（因为这是唯一匹配的模式）。“red”的首次出现被替换为函数的返回值“blue”。函数加上正则表达式处理文本替换的能力是十分强大的，可让你使用所有 JavaScript 的功能来决定什么才是替换文本。

195

注意 在前面的三个例子中，都只是替换给定字符串中第一个出现的“red”。如果要替换“red”的所有出现，必须指明表达式为`/red/g`。

第二种方法是 `split()`，它可以将字符串分割成一系列子串并通过一个数组将它们返回，比如：

```
var sColor = "red,blue,yellow,green";
var arrColors = sColor.split(","); //split at each comma
```

前面的代码创建一个数组 arrColors，数组包含了四个条目，“red”、“blue”、“yellow”和“green”。使用正则表达式来替代逗号字符串也可以实现同样的功能：

```
var sColor = "red,blue,yellow,green";
var reComma = /\,/;
var arrColors = sColor.split(reComma); //split at each comma
```

注意正则表达式 reComma 中必须在逗号字符前有一个反斜杠。逗号在正则表达式语法中有特殊的含义，这当然并非在这里所期望的。

在这些方法中，你可能还没有看出来使用正则表达式代替普通字符串的好处。记住本章这一节中的例子是非常简单的，仅仅用来介绍概念；稍后我们便介绍较为复杂的模式，会展示出使用正则表达式代替普通字符串的强大威力。

196

7.2 简单模式

目前为止，本章中使用的模式还是相当简单，仅直接由字符串组成。然而，一个正则表达式具有很多的组成部分而非仅仅匹配指定字符。元字符、字符类和量词等都是正则表达式语法中非常重要的组成部分，适当应用可以取得很好的效果。

7.2.1 元字符

在前一节中发现，逗号必须进行转义（在前面加上反斜杠）才能正确匹配。这是因为逗号是一个元字符，元字符是正则表达式语法的一部分。这里是正则表达式用到的所有元字符：

```
( [ { \ ^ $ | ) ? * + .
```

任何时候要在正则表达式中使用这些元字符，都必须对它们进行转义。因此，要匹配一个问号，正则表达式应该这样表示：

```
var reQMark = /\?/;
```

或者这样表示：

```
var reQMark = new RegExp("\\?");
```

注意到第二行中的两个反斜杠了吗？这是需要大家去理解的重要概念。当正则表达式以这种形式（第二行的，非字面量）表示时，所有的反斜杠都必须用两个反斜杠来替换，因为 JavaScript 字符串解析器会按照翻译\n 的方式尝试翻译\?。为了保证不会出现这个问题，在元字符的前面加上两个反斜杠（我们称之为双重转义）。这个小小的 gotcha 就是多数开发者更偏好使用字面量语法的原因。

7.2.2 使用特殊字符

你可以通过直接使用字符来表示它们本身，也可以使用它们的 ASCII 代码或者 Unicode 代码

指定字符。要使用 ASCII 来表示一个字符，则必须指定一个两位的十六进制代码，并在前面加上 \x。例如，字符 b 的 ASCII 码为 98，等于十六进制的 62，因此，表示字符 b 可以使用 \x62：

```
var sColor = "blue";
var reB = /\x62/;
alert(reB.test(sColor)); //outputs "true"
```

这段代码匹配 "blue" 中的字母 b。

另外，也可以使用八进制代替十六进制来指定字符代码，直接在反斜杠之后跟上八进制数值。例如，b 等于八进制的 142，所以下面应该这样：

```
var sColor = "blue";
var reB = /\142/;
alert(reB.test(sColor)); //outputs "true"
```

如果要使用 Unicode 来表示字符，必须指定字符串的四位的十六进制表示形式。因此 b 应该是 \u0062：

```
var sColor = "blue";
var reB = /\u0062/;
alert(reB.test(sColor)); //outputs "true"
```

注意这种方式如果使用 RegExp 构造函数来表示字符，则仍然需要使用两个反斜杠：

```
var sColor = "blue";
var reB = new RegExp("\\u0062");
alert(reB.test(sColor)); //outputs "true"
```

另外，还有其他一些预定义的特殊字符，如下表所列：

字 符	描 述
\t	制表符
\n	换行符
\r	回车符
\f	换页符
\a	alert 字符
\e	escape 字符
\cx	与 X 相对应的控制字符
\b	回退字符
\v	垂直制表符
\0	空字符

如果通过 RegExp 构造函数来使用它们，则都必须进行双重转义。

假设想删除字符串中所有的换行符（常用于处理用户输入的文本）。可以这样做：

```
var sNewString = sStringWithNewLines.replace(/\n/g, "");
```

7.2.3 字符类

字符类是用于测试的字符的组合。通过将一些字符放入方括号中，可以很有效地告诉正则表达式去匹配第一个字符、第二个字符、第三个字符等等。例如，要匹配字符 a、b 和 c，字符类应该是 [abc]。这个称之为简单类（simple class），因为它确切地指定了要查找的那些字符。

1. 简单类

假设想匹配 "bat"、"cat" 和 "fat"。使用简单类可以很方便的解决这个问题：

```
var sToMatch = "a bat, a Cat, a fAt baT, a faT cat";
var reBatCatRat = /[bcf]at/gi;
var arrMatches = sToMatch.match(reBatCatRat);
```

其中 arrMatches 数组现在存放了这些值："bat"、"Cat"、"fAt"、"baT"、"faT" 和 "cat"。也可以在简单类（或者其他字符类）中包含特殊字符。这里假设把字符 b 替换成它的 Unicode 形式：

```
var sToMatch = "a bat, a Cat, a fAt baT, a faT cat";
var reBatCatRat = /[\u0062cf]at/gi;
var arrMatches = sToMatch.match(reBatCatRat);
```

这段代码和前面的例子具有相同的执行结果。

2. 负向类

有时候，除了特定的一些，你可能会想要匹配所有的字符。这种情况下，可以使用负向类（negation class），它可以指定要排除的字符。例如，要匹配除了 a 和 b 的所有字符，那么这个字符类就是 [^ab]。脱字符号（^）告诉正则表达式字符不能匹配后面跟着的字符。

回到前面的例子，如果只想获取包含 at 但不能以 b 或 c 开头单词呢？

```
var sToMatch = "a bat, a Cat, a fAt baT, a faT cat";
var reBatCatRat = /^[^bc]at/gi;
var arrMatches = sToMatch.match(reBatCatRat);
```

在这个例子中，arrMatches 包含 "fAt" 和 "faT"，因为这些字符串匹配了一个以 at 结尾但不以 b 或 c 开头的字符序列。

3. 范围类

到现在为止，字符类仍然要求输入所有的包含或者排除的字符。假设要匹配所有的字母表中的字符，但是又实在不想逐个输入。这时，可以使用范围类（range class）指定从 a 到 z 之间的范围：[a-z]。这里最关键的地方是那条横线（-），它在这里应该看作是“从什么到什么”而不是减号（所以这个类应该读作从 a 到 z 而不是 a 减号 z）。

199 注意 [a-z] 仅仅匹配了小写字母，除非正则表达式使用 i 选项来指明是不区分大小写的。
如果仅仅要匹配大写字母，请使用 [A-Z]。

当要测试的字符恰好是按照字符编码的顺序排列的时候，范围类就很管用。考虑下列例子：

```
var sToMatch = "num1, num2, num3, num4, num5, num6, num7, num8, num9";
var reOneToFour = /num[1-4]/gi;
var arrMatches = sToMatch.match(reOneToFour);
```

在执行完毕后，`arrMatches` 中将包含“num1”、“num2”、“num3”和“num4”，因为他们都匹配 `num` 后面跟着字符 1~4。

也可以使用负向范围类，这样可以排除给定范围内的所有字符。例如，要排除字符 1~4，使用的类应该是`[^1-4]`。

4. 组合类

组合类（combination class）是由几种其他的类组合而成的字符类。例如，假设要匹配所有从 a~m 的字母以及从 1~4 的数字，以及一个换行符，那么所用到的类应该这样：

`[a-m1-4\n]`

注意在内部的类之间不要有空格。

JavaScript/ECMAScript 不支持某些其他正则表达式实现中的联合类（union class）和交叉类（intersection class）。这意味着你不能有类似`[a-m[p-z]]`或者`[a-m[^b-e]]`之类的模式出现。

5. 预定义类

由于某些模式会反复用到，所以可以使用一组预定义字符类以让我们更方便地指定复杂类。下表列出了所有的预定义类。

代 码	等 同 于	匹 配
.	<code>[^\n\r]</code>	除了换行和回车之外的任意字符
\d	<code>[0-9]</code>	数字
\D	<code>[^0-9]</code>	非数字字符
\s	<code>[\t\n\x0B\f\r]</code>	空白字符
\S	<code>[^ \t\n\x0B\f\r]</code>	非空白字符
\w	<code>[a-zA-Z_0-9]</code>	单词字符（所有的字母、所有的数字和下划线）
\W	<code>[^a-zA-Z_0-9]</code>	非单词字符

200

使用预定义类可以明显地使模式匹配变得简单。例如，假设想匹配 3 个数字，如果不用`\d`的话，代码会类似这样：

```
var sToMatch = "567 9838 abc";
var reThreeNums = /[0-9][0-9][0-9]/;
alert(reThreeNums.test(sToMatch)); //outputs "true"
```

使用`\d`，正则表达式变得更加明了：

```
var sToMatch = "567 9838 abc";
var reThreeNums = /\d\d\d/;
alert(reThreeNums.test(sToMatch)); //outputs "true"
```

7.2.4 量词

量词 (quantifier) 可以指定某个特定模式出现的次数。当指定某个模式应当出现的次数时，可以指定硬性数量（例如，某个字符应该出现三次），也可以指定软性数量（例如，这个字符至少应该出现一次，不过可以重复任意次）。

1. 简单量词

下表列出了指定特定模式数量的几种不同方法：

代 码	描 述
?	出现零次或一次
*	出现零次或多次（任意次）
+	出现一次或多次（至少出现一次）
{n}	一定出现 n 次
{n,m}	至少出现 n 次但不超过 m 次
{n,}	至少出现 n 次

例如，假设想匹配单词 bread, read 或 red。使用问号量词，则可以只要使用一个表达式就可以匹配这三个：

201 var reBreadReadOrRed = /b?rea?d/;

可以这样理解这个正则表达式：b 出现零次或一次，跟着 r，跟着 e，跟着出现零次或一次的 a，跟着 d。前面的正则表达式和下面的这个正则表达式是一样的：

```
var reBreadReadOrRed = /b{0,1}rea{0,1}d/;
```

在这个表达式中，问号被花括号代替。在花括号中是数字 0 和 1，前者表示出现次数的最小值，后者表示最大值。这个表达式理解上和前面的是一样的；仅是表现形式不同而已。两个表达式都是正确的。

假设已创建了正则表达式来匹配字符串 bd、bad、baad 和 baaad，我们就用这个来具体描述其他几个量词。下表描述了一些可能的解法，以及匹配哪个单词。

正则表达式	匹 配
ba?d	"bd", "bad"
ba*d	"bd", "bad", "baad", "baaad"
ba+d	"bad", "baad", "baad"
ba{0,1}d	"bd", "bad"
ba{0,}d	"bd", "bad", "baad", "baaad"
ba{1,}d	"bad", "baad", "baad"

可以看到，这六个表达式中只有两个恰当地解决了问题： ba^*d 和 $ba\{0,\}d$ 。注意，其实这两个表达式是相等的，因为星号就表示 0 次或多次，与 $\{0,\}$ 一样。类似的，第一个和第四和表达式是相等的，第三个和第六个是相等的。

量词也可以和字符类一起使用，所以如果想匹配字符串 bead、baed、beed、baad、bad 和 bed，可以使用下面这个正则表达式：

```
var reBeadBaedBeedBaadBedBad = /b[ae]\{1,2\}d/;
```

这个表达式表示：字符类 [ae] 可以出现最少一次，最多两次。

2. 贪婪的、惰性的和支配性的量词

这三种正则表达式量词分别是：贪婪的、惰性的和支配性的。

贪婪量词先看整个的字符串是否匹配。如果没有发现匹配，它去掉该字符串中的最后一个字符，并再次尝试。如果还是没有发现匹配，那么再次去掉最后一个字符，这个过程会一直重复直到发现一个匹配或者字符串不剩任何字符。到目前为止讨论的所有量词都是贪婪的。202

惰性量词先看字符串中的第一个字母是否匹配。如果单独这一个字符还不够，就读入下一个字符，组成两个字符的字符串。如果还是没有发现匹配，惰性量词继续从字符串中添加字符直到发现匹配或者整个字符串都检查过也没有匹配。惰性量词和贪婪量词的工作方式恰好相反。

支配量词只尝试匹配整个字符串。如果整个字符串不能产生匹配，不做进一步尝试。支配量词其实简单的说，就是一刀切。

怎样表示贪婪、惰性和支配量词呢？正是完全使用星号、加号和问号。例如，单独一个问号 (?) 是贪婪的，但一个问号后面再跟一个问号 (??) 就是惰性的。要使问号成为支配量词，在后面加上一个加号 (?+)。下面的表格列出了所有学过的贪婪、惰性和支配性的量词。

贪 婪	惰 性	支 配	描 述
?	??	?+	零次或一次出现
*	*?	*+	零次或多次出现
+	+?	++	一次或多次出现
{n}	{n}?	{n}+	恰好 n 次出现
{n,m}	{n,m}?	{n,m}+	至少 n 次至多 m 次出现
{n,}	{n,}?	{n,}+	至少 n 次出现

要说明这三种不同的量词之间的区别，请看下面的例子：

```
var sToMatch = "abbbaabbbbaaabbb1234";
var re1 = /.+bbb/g;      //greedy
var re2 = /.+?bbb/g;    //reluctant
var re3 = /.++bbb/g;   //possessive
```

想匹配跟着 bbb 的任意字母。最后，想要获得的是匹配 "abbb"、"aabbb" 和 "aaabbb"。然而，这三个表达式只有一个能正确地返回这个结果，你能猜出是哪个吗？

如果你猜的是 re2，恭喜你答对了！你已经理解了贪婪、惰性和支配量词之间的差别。第一

个表达式 re1，是贪婪的，所以它首先看整个字符串。这样，下面正是匹配的过程：

```
re1.test("abbbaabbbbaaabbb1234"); //false - no match
re1.test("abbbaabbbbaaabbb123"); //false - no match
re1.test("abbbaabbbbaaabbb12"); //false - no match
re1.test("abbbaabbbbaaabbb1"); //false - no match
re1.test("abbbaabbbbaaabbb"); //true - match!
```

所以 re1 返回的唯一结果是"abbbaabbbbaaabbb"。记住，点代表的是任意字符，b 也包括在

203 内，因此"abbbaabbbbaaa"匹配表达式.*的部分，"bbb"匹配表达式中 bbb 的部分。

对于第二个表达式 re2，匹配过程如下：

```
re2.test("a"); //false - no match
re2.test("ab"); //false - no match
re2.test("abb"); //false - no match
re2.test("abbb"); //true - match!
//store this result and start with next letter

re2.test("a"); //false - no match
re2.test("aa"); //false - no match
re2.test("aab"); //false - no match
re2.test("aabb"); //false - no match
re2.test("aabbb"); //true - match!
//store this result and start with next letter

re2.test("a"); //false - no match
re2.test("aa"); //false - no match
re2.test("aaa"); //false - no match
re2.test("aaab"); //false - no match
re2.test("aaabb"); //false - no match
re2.test("aaabbb"); //true - match!
//store this result and start with next letter

re2.test("1"); //false - no match
re2.test("12"); //false - no match
re2.test("123"); //false - no match
re2.test("1234"); //false - no match
//done
```

由于 re2 包含一个惰性量词，所以它能按照预期的返回"abbb"、"aabbb"和"aaabbb"。

最后一个正则表达式 re3，其实没有返回结果，因为它是支配性的。这里是匹配过程：

```
re3.test("abbbaabbbbaaabbb1234"); //false - no match
```

因为支配量词仅做一次测试，如果这个测试失败，就得不到结果。在这里，字符串结尾是"1234"，这导致表达式不能匹配。如果字符串是"abbbaabbbbaaabbb"，那么 re3 可以和 re1 返回同样的结果。

204 浏览器对支配量词的支持还很不完善。IE 和 Opera 不支持支配量词，如果要用它，就会抛出一个错误。Mozilla 不会产生错误，但是它会将支配量词看作是贪婪的。

7.3 复杂模式

如前面几节讨论的那样，正则表达式可以表示简单的模式，当然也可以表达复杂的模式。复杂的模式不仅仅由字符类和量词组成，也可以由分组、反向引用、前瞻和其他一些强大的正则表达式功能组成。这一节将介绍这些概念，这样就可以更方便地使用正则表达式进行复杂的字符串操作。

7.3.1 分组

到本章为止，已学过的正则表达式都是处理一个字符接着一个字符的基本内容。正如所预期的，特定的字符序列（不仅包含单独的字符）可以重复它们自身。要处理字符序列，正则表达式支持分组功能。

分组是通过用一系列括号包围一系列字符、字符类以及量词来使用的。例如，假设想匹配字符串 "dogdog"。使用目前获得的知识，可能估计表达式应该类似：

```
var reDogDog = /dogdog/g;
```

尽管这是可以的，但有点浪费。如果不知道 dog 在字符串中到底出现几次时该怎么办？如果 dog 重复次数过多呢？你可以使用分组来重写这个表达式，如下：

```
var reDogDog = /(dog){2}/g;
```

表达式中的括号的意思是字符序列"dog"将在一行上连续出现两次。但是并不限制在分组后使用花括号，可以使用任意量词：

```
var re1 = /(dog)?/; //match zero or one occurrences of "dog"
var re2 = /(dog)*/; //match zero or more occurrences of "dog"
var re3 = /(dog)+/; //match one or more occurrences of "dog"
```

通过混合使用字符、字符类和量词，甚至可以实现一些相当复杂的分组：

```
var re = /([bd]ad?)*/; //match zero or more occurrences of "ba", "da", "bad", or
"dad"
```

同时也不介意将分组放在分组中间：

```
var re = /(mom( and dad)?)/; //match "mom" or "mom and dad"
```

这个表达式要求"mom"是必需的，但是字符串" and dad"可以出现零次或一次。分组还可以用来弥补一些 JavaScript 所缺乏的一些语言功能。

大部分带有字符串的编程语言中，用于去掉字符串头部和尾部的空格的方法，往往都会作为一个标准提供。然而，JavaScript，从它诞生起就没有一个标准的方法。幸好，正则表达式（在分组的帮助下）使得建立一个字符串的 trim() 方法，十分方便。

幸好有了匹配所有空白字符的\s 字符类，匹配头部和尾部空白的正则表达式才十分简单：

```
var reExtraSpace = /^\\s*(.*?)\\s+$/;
```

这个正则表达式将查找字符串开头的零个或多个空白，跟着是任意数目的字符（在分组中捕

获的字符), 最后在字符串结尾处又是零个或多个空白。通过配合使用 String 对象的 replace() 方法以及反向引用, 就可以定义自己的 trim() 方法:

```
String.prototype.trim = function () {
    var reExtraSpace = /^[\s+.*?\s+]/;
    return this.replace(reExtraSpace, "$1");
};
```

有这个方法后, 就可以快速的为字符串去掉首尾的空格:

```
var sTest = "    this is a test    ";
alert("[" + sTest + "]");      //outputs "[this is a test]"
alert("[" + sTest.trim() + "]"); //outputs "[this is a test]"
```

7.3.2 反向引用

那么, 在表达式计算完后还可以怎样利用分组? 每个分组都被存放在一个特殊的地方以备将来使用。这些存储在分组中的特殊值, 我们称之为反向引用 (backreference)。

反向引用是按照从左到右遇到的左括号字符的顺序进行创建和编号的。例如, 表达式 (A? (B? (c?))) 将产生编号从 1~3 的三个反向引用:

- (1) (A? (B? (c?)))
- (2) (B? (c?))
- (3) (c?)

反向引用可以有几种不同的使用方法。

首先, 使用正则表达式对象的 test()、match() 或 search() 方法后, 反向引用的值可以从 RegExp 构造函数中获得。例如:

```
var sToMatch = "#123456789";
var reNumbers = /\#(\d+)/;
reNumbers.test(sToMatch);
alert(RegExp.$1);           //outputs "123456789"
```

这个例子尝试匹配后面跟着几个或多个数字的镑符号 (the pound sign)。并对数字进行分组以存储他们。在调用 test() 方法后, 所有的反向引用都被保存在 RegExp 构造函数中, 从 RegExp.\$1 (它保存了第一个反向引用) 开始, 如果还有第二个反向引用, 就是 RegExp.\$2, 如果第三个反向引用存在, 就是 RegExp.\$3, 依此类推。因为该组匹配了 "123456789", 所以 RegExp.\$1 中就存储了这个字符串。

然后, 还可以直接在定义分组的表达式中包含反向引用。这可以通过使用特殊转义序列如 \1、\2 等等实现。例如:

```
var sToMatch = "dogdog";
var reDogDog = /(dog)\1/;
alert(reDogDog.test(sToMatch)); //outputs "true"
```

正则表达式 reDogDog 首先创建单词 dog 的组, 然后又被特殊转义序列 \1 引用, 使得这个正则表达式等同于 /dogdog/。

第三，反向引用可以用在 String 对象的 replace() 方法中，这通过使用特殊字符序列 \$1、\$2 等等来实现。描述这种功能的最佳例子是调换字符串中的两个单词的顺序。假设想将字符串 "1234 5678" 变成 "5678 1234"。可以通过下面的代码来实现：

```
var sToChange = "1234 5678";
var reMatch = /(\d{4}) (\d{4})/;
var sNew = sToChange.replace(reMatch, "$2 $1");
alert(sNew); //outputs "5678 1234"
```

在这个例子中，正则表达式有两个分组，每一个分组有四个数字。在 replace() 方法的第二个参数中，\$2 等同于 "5678"，而 \$1 等同于 "1234"，对应于它们在表达式中出现的顺序。

7.3.3 候选

有时候要构建一个能够正确匹配想得所有可能性的模式是十分困难的。如果要对同一个表达式同时匹配 "red" 和 "black" 时要怎么做呢？这些单词完全没有相同的字符，这样就要写两个不同的正则表达式，并分别对两字符串进行匹配，像这样：

```
var sToMatch1 = "red";
var sToMatch2 = "black";
var reRed = /red/;
var reBlack = /black/;
alert(reRed.test(sToMatch1) || reBlack.test(sToMatch1)); //outputs "true"
alert(reRed.test(sToMatch2) || reBlack.test(sToMatch2)); //outputs "true"
```

这虽然完成了任务，但是十分冗长。还有另一种方式就是使用正则表达式的候选操作符。

候选操作符和 ECMAScript 的二进制异或一样，是一个管道符 (|)，它放在两个单独的模式之间，正如下面的例子：

```
var sToMatch1 = "red";
var sToMatch2 = "black";
var reRedOrBlack = /(red|black)/;
alert(reRedOrBlack.test(sToMatch1)); //outputs "true"
alert(reRedOrBlack.test(sToMatch2)); //outputs "true"
```

207

在这里，reRedOrBlack 匹配 "red" 或者 "black"，同时测试每个字符串都返回 "true"。因为两个备选项存放在一个分组中的，不管哪个被匹配了，都会存在 RegExp.\$1 中以备将来使用（同时也可在表达式中使用 \1）。在第一个测试中，RegExp.\$1 等于 "red"，在第二个中，它等于 "blue"。

也可以指定任何数目候选项，只要你能想到的，只要在之间加入候选操作符：

```
var sToMatch1 = "red";
var sToMatch2 = "black";
var sToMatch3 = "green";
var reRedOrBlack = /(red|black|green)/;
alert(reRedOrBlack.test(sToMatch1)); //outputs "true"
alert(reRedOrBlack.test(sToMatch2)); //outputs "true"
alert(reRedOrBlack.test(sToMatch3)); //outputs "true"
```

OR 模式在实践中一种通常的用法是从用户输入中删除不合适的单词，这对于在线论坛来说是非常重要的。通过针对这些敏感单词使用 OR 模式和 `replace()` 方法，则可以很方便地在帖子发布之前去掉敏感内容：

```
var reBadWords = /badword|anotherbadword/gi;
var sUserInput = "This is a string using badword1 and badword2.";
var sFinalText = sUserInput.replace(reBadWords, "*****");
alert(sFinalText); //output "This is a string using **** and *****"
```

这个例子中指定 "badword1" 和 "badword2" 是不合适的。表达式 `reBadWords` 使用“或”操作符来指定这两个单词（注意全局和大小写不分标志都已设置）。当使用 `replace()` 方法后，所有非法单词都被替换成四个星号（众所周知的四字母的单词表示法）。

也可以用星号替换敏感词中的每一个字母，也就是说最后出现的文本中星号的数量和敏感词中的字符数量是一样的。使用函数来作为 `replace()` 方法的第二个参数，就可以达到这个目的：

```
var reBadWords = /badword|anotherbadword/gi;
var sUserInput = "This is a string using badword1 and badword2.";
var sFinalText = sUserInput.replace(reBadWords, function(sMatch) {
    return sMatch.replace(/./g, "*");
});
alert(sFinalText); //output "This is a string using ***** and *****"
```

在这段代码中，作为第二个参数传给 `replace()` 方法的函数其实还使用了另外一个正则表达式。当执行函数之后，`sMatch` 包含了敏感词汇中的一个。最方便快捷的将每个字符替换成星号的方法是，对 `sMatch` 使用 `replace()` 方法，指定一个匹配任意字符的模式（也就是句号）并将其替换成一个星号（注意要设置全局标志）。这种技巧可以保证，不合适的评论不会出现在你的在线论坛或者 BBS 上。

208

7.3.4 非捕获性分组

创建反向引用的分组，我们称之为捕获性分组。同时还有一种非捕获性分组，它不会创建反向引用。在较长的正则表达式中，存储反向引用会降低匹配速度。通过使用非捕获性分组，仍然可以拥有与匹配字符串序列同样的能力，而无需存储结果的开销。

如果要创建一个非捕获性分组，只要在左括号的后面加上一个问号和一个紧跟的冒号：

```
var sToMatch = "#123456789";
var reNumbers = /#(?:\d+)/;
reNumbers.test(sToMatch);
alert(RegExp.$1);           //outputs ""
```

这个例子的最后一行代码输出一个空字串，因为该分组是非捕获性的。正因如此，`replace()` 方法就不能通过 `RegExp.$x` 变量来使用任何反向引用，或在正则表达式中使用它。看看运行下面的代码会怎样：

```
var sToMatch = "#123456789";
var reNumbers = /#(?:\d+)/;
alert(sToMatch.replace(reNumbers, "abcd$1")); //outputs "abcd$1"
```

这段代码输出 "abcd\$1" 而不是 "abcd123456789"，因为 "\$1" 在这里并不被看成是一个反向引用，而被直接翻译成字符。

正则表达式又一个十分常用的方式是去掉文本中所有的 HTML 标签，尤其是在论坛和 BBS 上，这可以防止游客在他们的发贴中插入恶意或是无意错误的 HTML。删除 HTML 标记的正则表达式很简单，只要用一个简单的表达式：

```
var reTag = /<(?:.|\\s)*?>/g;
```

这个表达式匹配一个小于号 (<) 后面跟着任何文本（这是在一个非捕获性分组内指定的），然后跟着一个大于号 (>)，这有效地匹配了所有的 HTML 标签。这里使用非捕获性分组是因为在小于号和大于号之间出现的内容并不重要（这些都是要被删除的）。你可以使用这个模式给 String 对象创建自己的 stripeHTML() 方法：

```
String.prototype.stripHTML = function () {
    var reTag = /<(?:.|\\s)*?>/g;
    return this.replace(reTag, "");
};
```

使用这个方法也很简单：

```
var sTest = "<b>This would be bold</b>";
alert(sTest.stripHTML()); //outputs "This would be bold"
```

209

7.3.5 前瞻

有时候，可能希望，当某个特定的字符分组出现在另一个字符串之前时，才去捕获它。如果使用“前瞻”就可以让这个过程变得十分简单。

前瞻 (lookahead) 就和它的名字一样，它告诉正则表达式运算器向前看一些字符而不移动其位置。同样存在正向前瞻和负向前瞻。正向前瞻检查的是接下来出现的是不是某个特定字符集。而负向前瞻则是检查接下来的不应该出现的特定字符集。

创建正向前瞻要将模式放在 (?= 和) 之间。注意这不是分组，虽然它也用到括号。事实上，分组不会考虑前瞻的存在（无论正向的还是负向的）。考虑下列代码：

```
var sToMatch1 = "bedroom";
var sToMatch2 = "bedding";
var reBed = /(bed(?=room))/;
alert(reBed.test(sToMatch1)); //outputs "true"
alert(RegExp.$1); //outputs "bed"
alert(reBed.test(sToMatch2)); //outputs "false"
```

在这个例子中，reBed 只匹配后面跟着 "room" 的 "bed"。因此，它能匹配 sToMatch1 而不能匹配 sToMatch2。在用表达式测试 sToMatch1 后，这段代码输出 RegExp.\$1 的内容，是 "bed"，而不是 "bedroom"。模式的 "room" 的部分是包含在前瞻中的，所以没有作为分组的一部分返回。

天平的另一端是负向前瞻，要创建它，则要将模式放在 (?!= 和) 之间。前面的例子也可以改成用负向前瞻匹配 "bedding" 而不是 "bedroom"。

```

var sToMatch1 = "bedroom";
var sToMatch2 = "bedding";
var reBed = /(bed(?!room))/;
alert(reBed.test(sToMatch1));      //outputs "false"
alert(reBed.test(sToMatch2));      //outputs "true"
alert(RegExp.$1);                //outputs "bed"

```

这里，表达式变成只匹配后面不跟着“room”的“bed”，所以模式匹配“bedding”而不是“bedroom”。在测试 sToMatch2 后， RegExp.\$1 还是包含“bed”，而不是“bedding”。

尽管 JavaScript 支持正则表达式前瞻，但它不支持后瞻。后瞻可以匹配这种模式：“匹配 b 当且仅当它前面没有 a”。

7.3.6 边界

210 边界 (boundary) 用于正则表达式中表示模式的位置。下表列出了几种可能的边界：

边 界	描 述
^	行开头
\$	行结尾
\b	单词的边界
\B	非单词的边界

假设想查找一个单词，但要它只出现在行尾，则可以使用美元符号 (\$) 来表示它：

```

var sToMatch = "Important word is the last one.";
var reLastWord = /(\w+)\.$/;
reLastWord.test(sToMatch);
alert(RegExp.$1);           //outputs "one"

```

例子中的正则表达式查找的是，在一行结束之前出现的跟着句号的一个或多个单词字符。当用这个表达式测试 sToMatch 时，它返回“one”。还可以很容易地获取一行的第一个单词，只要使用脱字符号 (^)：

```

var sToMatch = "Important word is the last one.";
var reFirstWord = /^(\w+)/;
reFirstWord.test(sToMatch);
alert(RegExp.$1);           //outputs "Important"

```

在这个例子中，正则表达式查找行起始位置后的一个或多个单词字符。如果遇到非单词字符，匹配停止，返回“Important”。这个例子也可以用单词边界实现：

```

var sToMatch = "Important word is the last one.";
var reFirstWord = /^(.+?)\b/;
reFirstWord.test(sToMatch);
alert(RegExp.$1);           //outputs "Important"

```

这里，正则表达式用惰性量词来制定在单词边界之前可以出现任何字符，且可出现一次或多次（如果使用贪婪性量词，表达式就匹配整个字符串）。

使用单词边界可以方便地从字符串中抽取单词：

```
var sToMatch = "First second third fourth fifth sixth"
var reWords = /\b(\S+?)\b/g;
var arrWords = sToMatch.match(reWords);
```

正则表达式 `reWords` 使用单词边界 (`\b`) 和非空白字符类 (`\S`) 从句子中抽取单词。执行完后，`arrWords` 数组将会包含 "First"、"second"、"third"、"fourth"、"fifth" 和 "sixth"。注意行的开始和行的结束，通常由 `^` 和 `$` 表示的位置，对应地也认为是单词的边界，所以 "First" 和 "sixth" 也在结果中。当然，这并不是唯一的一种获取句子中所有单词的方法。

211

事实上，更加简单的方法是使用单词字符类 (`\w`)：

```
var sToMatch = "First second third fourth fifth sixth"
var reWords = /(\w+)/g;
var arrWords = sToMatch.match(reWords);
```

这又是一个不同的表达式含义获得相同功能的例子。

7.3.7 多行模式

上节中，学过了行边界的开始和结束的匹配。如果字符串只有单独一行，那么显而易见。但如果字符串中包含很多行呢？当然可以使用 `split()` 方法将字符串分割成行与行的数组，但就得对每一行单独进行正则表达式测试。

要描述这个问题，先看下面的例子：

```
var sToMatch = "First second\nthird fourth\nfifth sixth"
var reLastWordOnLine = /(\w+)\$/g;
var arrWords = sToMatch.match(reLastWordOnLine);
```

这段代码中的正则表达式想要匹配行末的一个单词。唯一包含在 `arrWords` 中的匹配是 "sixth"，因为只有它在字符串的结尾处。然而，在 `sToMatch` 中有两个换行符，所以其实 "second" 和 "fourth" 也应该返回。这就是引入多行模式的原因。

要指定多行模式，只要在正则表达式后面添加一个 `m` 选项。这会让 \$ 边界匹配换行符 (`\n`) 以及字符串真正的结尾。如果添加了这个选项，前面的例子将返回 "second"、"fourth" 和 "sixth"：

```
var sToMatch = "First second\nthird fourth\nfifth sixth"
var reLastWordOnLine = /(\w+)\$/gm;
var arrWords = sToMatch.match(reLastWordOnLine);
```

多行模式同样也会改变 `^` 边界的行为，这时它会匹配换行符之后的位置。例如，前面的例子中，要从字符串中获取字符 "First"、"third" 和 "fifth"，则可以这么做：

```
var sToMatch = "First second\nthird fourth\nfifth sixth"
var reFirstWordOnLine = /^(\w+)/gm;
var arrWords = sToMatch.match(reFirstWordOnLine);
```

若不指定多行模式，表达式将只返回 "First"。

7.4 理解 RegExp 对象

JavaScript 中的每个正则表达式都是一个对象，同其他的对象一样。你已经知道正则表达式

212 是用 RegExp 对象来表示的，还知道它拥有方法（在前面的几节中已经讨论过）。但 RegExp 对象还有一些属性，RegExp 实例和构造函数都有属性。两者的属性在创建模式和进行测试的时候都会发生改变。

7.4.1 实例属性

RegExp 的实例有一些开发人员可以使用的属性：

- global——Boolean 值，表示 g（全局选项）是否已设置。
- ignoreCase——Boolean 值，表示 i（忽略大小写选项）是否已设置。
- lastIndex——整数，代表下次匹配将会从哪个字符位置开始（只有当使用 exec() 或 test() 函数才会填入，否则为 0）。
- multiline——Boolean 值，表示 m（多行模式选项）是否已设置。
- source——正则表达式的源字符串形式。例如，表达式 / [ba] */ 的 source 将返回 "[ba] *"。

一般不会使用 global、ignoreCase、multiline 和 source 属性，因为一般之前就已知道了这些数据：

```
var reTest = /[ba]*/i;
alert(reTest.global);           //outputs "false"
alert(reTest.ignoreCase);       //outputs "true"
alert(reTest.multiline);        //outputs "false"
alert(reTest.source);          //outputs "[ba] *"
```

真正有用的属性是 lastIndex，它可以告诉你正则表达式在某个字符串中停止之前，查找了多远：

```
var sToMatch = "bbq is short for barbecue";
var reB = /b/g;
reB.exec(sToMatch);
alert(reB.lastIndex);          //outputs "1"
reB.exec(sToMatch);
alert(reB.lastIndex);          //outputs "2"
reB.exec(sToMatch);
alert(reB.lastIndex);          //outputs "18"
reB.exec(sToMatch);
alert(reB.lastIndex);          //outputs "21"
```

在这个例子中，正则表达式 reB 查找的是 b。当它首次检测 sToMatch 时，它发现在第一个位置——也就是位置 0——是 b；因此，lastIndex 属性就被设置成 1，而再次调用 exec() 时就从这个地方开始执行。再次调用 exec()，表达式在位置 1 又发现了 b，于是将 lastIndex 设置为 2。第三次调用时，发现 b 在位置 17，于是又将 lastIndex 设置为 18，如此继续。

如果想从头开始匹配，则可以将 lastIndex 设为 0。

```

var sToMatch = "bbq is short for barbecue";
var reB = /b/g;
reB.exec(sToMatch);
alert(reB.lastIndex); //outputs "1"
reB.lastIndex = 0;
reB.exec(sToMatch);
alert(reB.lastIndex); //outputs "1"

```

213

修改这段代码后，两次 exec() 的调用都是在位置 0 找到 b，所以两次 lastIndex 的值都是 "1"。

7.4.2 静态属性

静态的 RegExp 属性对所有的正则表达式都有效。这些属性也都与众不同的，因为它们都有两个名字：一个复杂名字和一个以美元符号开头的简短名字。下表中列出了这些属性：

长 名	短 名	描 述
input	\$_	最后用于匹配的字符串（传递给 exec() 或 test() 的字符串）
lastMatch	\$&	最后匹配的字符
lastParen	\$+	最后匹配的分组
leftContext	\$`	在上次匹配的前面的子串
multiline	\$*	用于指定是否所有的表达式都使用多行模式的布尔值
rightContext	\$'	在上次匹配之后的子串

这些属性可以告诉你，关于刚使用 exec() 或 test() 完成的匹配的一些特定信息。例如：

```

var sToMatch = "this has been a short, short summer";
var reShort = /(s)hort/g;
reS.test(sToMatch);
alert(RegExp.input); //outputs "this has been a short, short summer"
alert(RegExp.leftContext); //outputs "this has been a "
alert(RegExp.rightContext); //outputs ", short summer"
alert(RegExp.lastMatch); //outputs "short"
alert(RegExp.lastParen); //outputs "s"

```

这一个例子描述了几个不同的属性应该如何使用：

- input 属性总是等于测试用的字符串。
- RegExp.leftContext 包含第一个实例 "short" 之前的所有字符，同时 RegExp.rightContext 包含第一个实例 "short" 之后的所有字符。
- lastMatch 的属性包含最后匹配整个正则表达式的字符串，也就是 "short"。
- lastParen 属性包含最后匹配的分组，这里就是 "s"。

214

也可以使用这些属性的短名字，不过对其中的大部分名字必须使用方括号标记，因为有些名字在 ECMAScript 的语法中是不合法的。

```

var sToMatch = "this has been a short, short summer";
var reShort = /(s)hort/g;
reShort.test(sToMatch);

```

```

alert(RegExp.$_);           //outputs "this has been a short, short summer"
alert(RegExp[$`]);          //outputs "this has been a "
alert(RegExp[$']);          //outputs ", short summer"
alert(RegExp[$&]);          //outputs "short"
alert(RegExp[$+]);          //outputs "s"

```

记住每次执行 `exec()` 或 `test()` 时，所有的属性（除 `multiline` 外）都会被重新设置。例如：

```

var sToMatch1 = "this has been a short, short summer";
var sToMatch2 = "this has been a long, long summer";
var reShort = /(s)hort/g;
var reLong = /(l)ong/g;

reShort.test(sToMatch1);
alert(RegExp.$_);           //outputs "this has been a short, short summer"
alert(RegExp[$`]);          //outputs "this has been a "
alert(RegExp[$']);          //outputs ", short summer"
alert(RegExp[$&]);          //outputs "short"
alert(RegExp[$+]);          //outputs "s"

reLong.test(sToMatch1);
alert(RegExp.$_);           //outputs "this has been a long, long summer"
alert(RegExp[$`]);          //outputs "this has been a "
alert(RegExp[$']);          //outputs ", long summer"
alert(RegExp[$&]);          //outputs "long"
alert(RegExp[$+]);          //outputs "l"

```

这里，第二个正则表达式 `reLong`，是在 `reShort` 之后使用的。所有的 `RegExp` 属性都设成新的值。

而 `multiline` 属性在这里不同于其他属性，因为它不依赖最后一次执行的匹配。相反，它可以设置所有的正则表达式的 `m` 选项。

```

var sToMatch = "First second\nthird fourth\nfifth sixth"
var reLastWordOnLine = /(\w+)\$/g;
RegExp.multiline = true;
var arrWords = sToMatch.match(reLastWordOnLine);

```

这段代码执行结束后，`arrWords` 包含 "second"、"fourth" 和 "sixth"，就像 `m` 选项已在表达式中设置过一样。

IE 和 Opera 并不支持 `RegExp.multiline`，所以最好单独的对每个表达式设置 `m` 选项而不要直接设置这个标记。

7.5 常用模式

在 web 上，正则表达式常用来在发送数据到服务器之前对用户的输入进行验证。毕竟，这个是 JavaScript 诞生的原因。

Web 上最常见的几种用于测试的模式是如下：

- 日期；
- 信用卡号；
- URL；
- E-mail 地址。

每个数据类型都代表了一个不同的待解决的问题。有些只涉及数字，有些涉及非字母字符，还有些包含可以忽略的字符。通过学习这四个模式，就可以提高自己的正则表达式技巧。

7.5.1 验证日期

对大多数 Web 开发人员来说，日期确实是一个很头疼的问题。先不管时髦的基于层的日历系统，用户其实就想能够直接输入一个日期。大部分开发人员很怕让用户手工输入日期。现在大家使用多种不同的方法，更不要说国际化的月份和日期的名称！很多网站用三个表单字段来输入日期条目，通常有两个组合框（一个是月份，另一个是月份中的日），然后还有一个用于输入年份的文本框（不过有时候，这个也是一个组合框）。尽管这个方法是可行的，但是用户依然希望能直接输入日期，这可比在三个字段中切换并把鼠标移上移下来查找选择条目要方便得多。

回头考虑一下第 3 章关于 `Date.parse()` 的讨论，它可以将几种字符串模式转换成日期的毫秒表示。我们快速回顾一下，有这些支持模式：

- m/d/yyyy (如 6/13/2004)；
- mmmm d, yyyy (如 January 12, 2004)。

但如果想让用户以 dd/mm/yyyy 的格式（如 25/06/2004，在欧洲很流行）输入日期的话，要怎么办？当然，可以借助正则表达式。

要了解其中一个模式，先想想不同的日期表现格式。例如，月份一般是两个数字，从 01~12；而天数一般也是两个数字，从 01~31。所以，月份和天数必须是两个数字，而年份必须是四位数。现在我们写一个简单的模式：

```
var reDate = /\d{1,2}\/\d{1,2}\/\d{4}/;
```

这个模式基本匹配格式 dd/mm/yyyy，但它并没有考虑到月份和天数的有效范围。所以，错误的结果，如 55/44/2004，也可以匹配成功。要解决这个问题，要先考虑仅仅匹配天数的模式：第一个数字只能是 0~3，第二个数字（这个是必需的）可以是 0~9 的任意数字。于是，天数的比较符合逻辑的模式应该如下：

```
var reDay = /[0-3]?[0-9]/;
```

然而这个表达式还匹配了 32~39，这也是不可能出现的日期。通过使用候选项和字符类，就可以得到几乎没有漏洞的日期模式：

```
var reDay = /0[1-9]|1[0-2][0-9]|3[01]/;
```

现在这个正则表达式可以正确匹配所有的天数值，0 也可以出现在 1~9 之前，但不能出现两个 0，或者数字由 1 和 2 开头的后面可以带任何 0~9 的数字。最后，如果天数由 3 开头，那么后

面只能有 0 或 1。接下来，让我们来处理月份。

月份的情况和天数的有点像，模式也类似，除了少了很多选项：

```
var reMonth = /0[1-9]|1[0-2]/;
```

这个模式可以匹配 01~12 的所有数字，没有特例。最后让我们创建年份的模式。

考虑一下现有的两种用户输入年份的方式。第一个是，让用户自己输入想要输入的（只要是字符），并让他们自己解决将来可能出现的无效日期的问题。另一种方法是将日期限制在 1900 ~ 2099 年之间，考虑到 2099 年来的那一天，用这些代码的系统肯定已经被放进垃圾箱了。

为了能构建一个完整的例子，并考虑第二个方式。可以用以下正则表达式来完成：

```
var reYear = /19|20\d{2}/;
```

所有在 1900~2099 年之间的年份都可以匹配这个模式。这个模式声明，年份必须以 19 或 20 开头，后面跟两个数字。

将以上三个天数、月份、年份的模式组合成起来，就成了：

```
var reDate = /(?:0[1-9]|12)[0-9]|3[01])\/(?:0[1-9]|1[0-2])\/(?:19|20\d{2})/;
```

注意整个模式是将日期的每个部分放入非捕获性分组中，这是为了确保可选项不会相互冲突。当然，如果需要，也可以使用捕获性分组。

最后，使用一个函数来检测日期是否有效是很容易的，可以把正则表达式包装成函数 `isValidDate()` 并进行测试：

```
function isValidDate(sText) {
    var reDate = /(?:0[1-9]|12)[0-9]|3[01])\/(?:0[1-9]|1[0-2])\/(?:19|20\d{2})/;
    return reDate.test(sText);
}
```

然后调用 `isValidDate()` 函数：

```
alert(isValidDate("5/5/2004")); //outputs "true"
alert(isValidDate("10/12/2009")); //outputs "true"
alert(isValidDate("6/13/2000")); //outputs "false"
```

7.5.2 验证信用卡号

如果有一个电子商务网站，常需要处理信用卡号的验证。并不是每一个错误的信用卡号都是骗子；有时候仅仅是人们输入错误或者过早地按了回车。为了避免再返回服务器，可以创建一些简单的模式来判断一个信用卡号是否有效。

我们从考虑 MasterCard 的信用卡号开始，MasterCard 必须包含 16 位数字。在这 16 个数字中，前两个数字必须是 51~55 之间的数字。下面是一个简单的模式：

```
var reMasterCard = /^5[1-5]\d{14}\$/;
```

注意脱字符号和美元符号的使用，它们表示输入字符串的开头和结尾以保证匹配整个字符串，而不仅仅是它的一部份。这个模式可行，但是 MasterCard 号也可以在每四个数码之间有空

格或者短横，例如 5555-5555-5555-5555，这个也必须考虑到。

```
var reMasterCard = /^5[1-5]\d{2}[\s-]?\d{4}[\s-]?\d{4}[\s-]?\d{4}$/;
```

其实，真正检验信用卡号还要用到 Luhn 算法。Luhn 算法是用验证唯一标示符的方法，常常用来验证信用卡号。然而，必须从用户输入中抽取出数字，才能将数字应用于这个算法，也就是说，必须加入捕获性分组：

```
var reMasterCard = /^(5[1-5]\d{2})[\s-]?( \d{4})[\s-]?( \d{4})[\s-]?( \d{4})$/;
```

现在可以开始构建用来验证 MasterCard 号的函数了。第一步是用这个模式测试给定的字符串。如果字符串匹配成功，则把四个数字组放入一个字符串中（例如，“5432-1234-5678-9012”应该被转换成“5432123456789012”）：

```
function isValidMasterCard(sText) {
    var reMasterCard = /^(5[1-5]\d{2})[\s-]?( \d{4})[\s-]?( \d{4})[\s-]?( \d{4})$/;

    if (reMasterCard.test(sText)) {
        var sCardNum = RegExp.$1 + RegExp.$2 + RegExp.$3 + RegExp.$4;
        //Luhn algorithm here
    } else {
        return false;
    }
}
```

218

Luhn 算法有四步。第一步是从卡号的最后一个数字开始，并逆向地逐个将奇数位置（1、3，等等）的数字相加。要判断当前的这个数字是不是在偶数位置上，可以使用 Boolean 标志（叫做 bIsOdd）。起始时，标记为 true，因为最后一个位置是 15。

在一个单独的函数中定义 luhn 算法是很有效的，因为这样其他的函数就可以很方便地调用它：

```
function luhnCheckSum(sCardNum) {
    var iOddSum = 0;
    var bIsOdd = true;

    for (var i=sCardNum.length-1; i >= 0; i--) {
        var iNum = parseInt(sCardNum.charAt(i));

        if (bIsOdd) {
            iOddSum += iNum;
        }

        bIsOdd = !bIsOdd;
    }
}
```

下一步是将偶数位置的数字相加，但这里有些麻烦事：必须先将数字乘以 2，然后，如果结

果是两位数，必须将两个位上的数相加，然后将结果加入到总和中。这话有点唠叨，那么考虑一下信用卡号“5432-1234-5678-9012”。现在已经将奇数位置的数字相加，等于 $4+2+2+4+6+8+0+2 = 28$ 。下面这一步，要将偶数位置的数字乘以 2，也就是说 5、3、1、3、5、7、9 和 1 都要乘以 2，得 10、6、2、6、10、14、16 和 2。因为 10、10、14 和 16 都是有两位的，必须将这几个数字两位相加，最后得到 1、6、2、6、1、5、7 和 2。将这些数字相加，并将结果存储，得 28。

将这个算法加入到代码中，得到：

```
function luhnCheckSum(sCardNum) {
    var iOddSum = 0;
    var iEvenSum = 0;
    var bIsOdd = true;

    for (var i=sCardNum.length-1; i >= 0; i--) {
        var iNum = parseInt(sCardNum.charAt(i));

        if (bIsOdd) {
            iOddSum += iNum;
        } else {
            iNum = iNum * 2;
            if (iNum > 9) {
                iNum = eval(iNum.toString().split("").join("+"));
            }
            iEvenSum += iNum;
        }
        bIsOdd = !bIsOdd;
    }
}
```

219

将 `else` 语句加入到 `if (bIsOdd)` 中就完成了偶数位置相加的任务。如果数字大于 9（表示这个数有两位），就对数字进行转化（用本书前面讲到的几种不同的方法）：

- (1) 用 `toString()` 方法将数字转化成字符串。
- (2) 用 `split()` 将字符串分割成有两个数字的数组。例如，12 将会被分割为包含“1”和“2”的数组。
- (3) 用 `join()` 并用加号将数组组合起来，这样“1”和“2”将得到“1+2”。
- (4) 将最后的结果字符串传给 `eval()`，它可以将字符串翻译为代码并执行（这样“1+2”就返回 3）。

现在，最后一步是将两个总和（从奇数部分和偶数部分得来的）相加并对结果进行取模（取余数）操作。如果数字是合法的，那么结果应该可以被 10 整除（也就是应该等于 20、30、40 等）。

```
function luhnCheckSum(sCardNum) {
    var iOddSum = 0;
    var iEvenSum = 0;
    var bIsOdd = true;
```

```

for (var i=sCardNum.length-1; i >= 0; i--) {
    var iNum = parseInt(sCardNum.charAt(i));
    if (bIsOdd) {
        iOddSum += iNum;
    } else {
        iNum = iNum * 2;
        if (iNum > 9) {
            iNum = eval(iNum.toString().split("").join("+"));
        }
        iEvenSum += iNum;
    }
    bIsOdd = !bIsOdd;
}
return ((iEvenSum + iOddSum) % 10 == 0);
}

```

把这个方法加入到 isValidMasterCard() 中，就完成了验证：

```

function isValidMasterCard(sText) {
    var reMasterCard = /^(5[1-5]\d{2})[\s\-\-]?(\d{4})[\s\-\-]?(\d{4})[\s\-\-]?(\d{4})$/;
    if (reMasterCard.test(sText)) {
        var sCardNum = RegExp.$1 + RegExp.$2 + RegExp.$3 + RegExp.$4;
        return luhnCheckSum(sCardNum);
    } else {
        return false;
    }
}

```

现在可以用这样的方法传入 MasterCard 号：

```

alert(isValidMasterCard("5432 1234 5678 9012")); //outputs "false"
alert(isValidMasterCard("5432-1234-5678-9012")); //outputs "false"
alert(isValidMasterCard("5432123456789012")); //outputs "false"

```

至于其他类的信用卡号，则必须了解管理这些卡号的规则。

Visa 卡号可能有 13 位或 16 位，且首位数字总是为 4，因此，匹配 Visa 卡号（不带空格）的模式应该是：

```
var reVisa = /(4\d{12}(?:\d{3}))?$/;
```

关于这个模式还有一些注意事项：

- 一个非捕获性分组包含卡号最后三个数字，是因为这三个数字单独使用的不多。
- 在最后的非捕获性分组后的问号，表示要么出现三个数字，要么就没有数字。

有了完整的正则表达式，只要抽取数字并将其应用于 Luhn 算法：

```

function isValidVisa(sText) {
    var reVisa = /^(4\d{12}(\?:\d{3}))?\$/;

    if (reVisa.test(sText)) {
        return luhnCheckSum(RegExp.$1);
    } else {
        return false;
    }
}

```

关于更多信用卡号模式和使用Luhn算法的信息，可以查看<http://www.beachnet.com/~hstiles/cardtype.html>。

221

7.5.3 验证电子邮件地址

创建模式来匹配所有有效的电子邮件地址确实是一个大事业。定义有效的电子邮件地址的标准的是 RFC 2822，它定义以下的一些模式是合法的：

- john@somewhere.com
- john.doe@somewhere.com
- John Doe <john.doe@somewhere.com>
- "john.doe"@somewhere.com
- john@[10.1.3.1]

然而，实际上，一般只会看到前三种变体，且只有前两种才是用户会输入到网站（或 Web 应用）上的文本框中的。正因如此，这一节将仅关注这两种模式的验证。

现已知道了有效的电子邮件地址的基本格式，是一串字符（可以是数字、字母、横线、点等等，除了空格），跟着 at (@) 符号，最后还有一些字符。你也知道（也许仅仅是下意识的），在 @之前至少有一个字符，其后必须至少有三个字符，这三个字符中第二个还必须是一个句号（a@a.b 是有效地址，而 a@a 和 a@a.都是无效的）。

@前后的文本必须符合同样的规则：不能以句号开始或结束，且不能有两个句号连续出现。因此，正则表达式如下：

```
var reEmail = /^(?:\w+\.\?)*\w+@\(?:\w+\.\?)*\w+\$/;
```

这个表达式以非捕获性分组 (?:\w+\.\?) 开始，它告诉你任何数量的单词字符可以跟零个或一个句号。这一部分可以出现零次或多次（例如 a.b.c.d），所以对这个分组使用星号。

表达式下面一个部分是 \w+\@，这保证在 @之前至少有一个单词字符。然后紧跟一个非捕获性分组 (?:\w+\.\.)，它可以出现一次或多次，所以使用加号。表达式最后一部分是 \w+\\$，表示一行最后的字符必须是一个单词字符，不允许出现类似 “john@doe.” 之类的地址。

最后，把这个模式装到一个函数中就可以了：

```
function isValidEmail(sText) {
    var reEmail = /^(?:\w+\.\w+@\w+\.\w+)$/;
    return reEmail.test(sText);
}
```

可以这样调用这个函数：

```
alert("john.doe@somewhere.com");      //outputs "true"
alert("john.doe@somewhere.");          //outputs "false"
```

222

7.6 小结

这一章介绍了 JavaScript/ECMAScript 的正则表达式实现。它包括两种不同的正则表达式声明方式：Perl 风格的和使用 `RegExp` 构造函数，以及它们可以使用的各种不同的属性和方法。

你学会了如何创建不同类型的正则表达式，从简单地使用字符字面量到使用字符类、量词和分组。另外，还学习了高级的正则表达式技巧，如候选项、前瞻、边界和多行模式。

最后，本章告诉你如何使用正则表达式来解决不同的问题，包括验证日期、信用卡号和电子邮件地址，以及如何从文本中删除多余的空格和不必要的 HTML 标签。

223

224



Web 编程的一个重要工作是确定目标浏览器和操作系统。无论是构建一个简单的网站还是一个复杂的 Web 应用，在工作开始之前都必须确定这些重要的信息。因为，不同的浏览器支持不同级别的 HTML 和 JavaScript，而且往往操作系统不同支持级别也不同，如果了解了目标，就可以节省很多时间和金钱。这能够保证程序中不会包含那些用户不可用的特性。

今天，因为在多种不同的平台上存在着大量的 Web 浏览器，所以这项工作变得更加富有挑战性。Windows 用户可以使用 IE、Mozilla 和 Opera；Macintosh 用户可以使用 IE、Mozilla，现在还有 Safari；Unix 用户可以使用 Mozilla 以及 Konqueror。为所有这些浏览器进行开发确实需要深谋远虑认真计划以应对它们之间的异同。

这一章将深入了解 JavaScript 对浏览器和操作系统的检测，来使你为开发跨平台解决方案做好准备。

8.1 navigator 对象

在客户端浏览器检测中最重要的对象是 navigator 对象。navigator 对象是最早实现的 BOM 对象之一（始于 Netscape Navigator 2.0 和 IE 3.0）。正如在第 5 章中提到的，它包含一些可以向你提供浏览器信息的属性，诸如名称、版本号和平台。

尽管微软反对使用术语 navigator，因为这指代了 Netscape 的浏览器，但 navigator 对象还是成为提供 Web 浏览器信息事实上的标准。（除了 navigator 对象外，微软还有一个叫做 clientInformation 的对象，不过其实它们提供的信息是类似的。）

8.2 检测浏览器的方式

其实与 JavaScript 中很多其他工作一样，进行浏览器检测也有多种不同的形式。目前，一般使用两种浏览器检测方式：对象/特征检测和 user-agent 字符串检测。每种方式都有它的优点与缺点，当准备部署你的 Web 解决方案时，一定理解合理的使用这两种方式。

8.2.1 对象/特征检测法

对象检测法（也叫做特征检测法）是一种判断浏览器能力（而非目标浏览器的确切型号）的

通用方法。大部分 JavaScript 专家认为这个方法最为合适，因为他们认为针对那些使得准确判断所使用浏览器变得困难的变化而进行编写的脚本是经得起未来考验的。

对象检测法涉及到，在使用一个给定对象之前要先检查它的存在。例如，假设要使用 DOM 方法 `document.getElementById()`，但是不确定浏览器是不是支持它。可以使用以下代码：

```
if (document.getElementById) {
    //the method exists, so use it here
} else {
    //do something else
}
```

前面的代码检查该方法是否存在。我们已经学过，如果一个属性（或者方法）不存在，就会返回 `undefined`。应该还记得 `undefined` 值翻译成 Boolean 值时就是 `false`。所以，如果 `document.getElementById()` 不存在，代码就跳到 `else` 子句去执行；否则，就执行前一部分的语句。

注意如果要检查函数是否存在，不能出现括号。如果加入了括号，解释器就会尝试调用这个函数，如果函数不存在就会产生错误。

如果更关注浏览器的能力而不在乎它实际的身份，就可以使用这种检测方法。贯穿本书，你可以看到有些例子中使用对象检测法，而另一些例子中使用 user-agent 字符串检测法则更为恰当。

8.2.2 user-agent 字符串检测法

user-agent 字符串检测法是最古老的浏览器检测方式。每个访问网站的程序都必须提供一个 user-agent 字符串来向服务器确定它的身份。以前，这个信息只能在服务器上通过 CGI 环境变量 `HTTP_USER_AGENT` 访问（通过 `$ENV{'HTTP_USER_AGENT'}` 访问）。不过，JavaScript 在 `navigator` 对象中引入了 `userAgent` 属性来提供客户端对 user-agent 字符串的访问。

```
var sUserAgent = navigator.userAgent;
```

226

user-agent 字符串提供了关于 Web 浏览器的大量信息，包括浏览器的名称和版本。这就是，网站流量统计软件也使用 user-agent 字符串来判断，网站的访客中有多少用户使用特定浏览器或者操作系统的原因。下面的表格显示了一些常见的浏览器的 user-agent 字符串。

浏览器	user-agent 字符串
Internet Explorer 6.0 (Windows XP)	Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)
Mozilla 1.5 (Windows XP)	Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.5) Gecko/20031007
Firefox 0.92 (Windows XP)	Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.7) Gecko/20040707 Firefox/0.8
Opera 7.54 (Windows XP)	Opera/7.54 (Windows NT 5.1; U)
Safari 1.25 (MacOS X)	Mozilla/5.0 (Macintosh; U; PPC Mac OS X; en) AppleWebKit/124 (KHTML, like Gecko) Safari/125.1

只需快速看一下这些 user-agent 字符串，就能从中了解很多关于生成它们的浏览器的信息。同时，你可能注意到，不同浏览器之间的 user-agent 字符串的区别。Opera 的 user-agent 字符串较短而 Safari 的特别长。而且你可能还注意到 IE 的 user-agent 字符串看上去类似 Mozilla 的，而且 Safari 的还类似 Gecko。从 user-agent 字符串的发展历史也可以看出这些年浏览器的发展历程。

8.3 user-agent 字符串简史

在研究 user-agent 字符串检测法之前，应该首先理解，为什么检测脚本要查找 user-agent 字符串的特定部分。如果不了解 user-agent 字符串为何以及怎样发展，是很难理解 user-agent 字符串的。这一节将介绍 user-agent 字符串从早期的浏览器（如 Netscape Navigator 3.0）到今天流行的浏览器（如 Safari）的演化过程。

8.3.1 Netscape Navigator 3.0 与 IE3.0

使 Web 开始普及的浏览器先锋正是 Netscape Navigator 3.0，它大约在 1996 年发布。这个版本的 Netscape 引擎的开发代号是 Mozilla，那时 user-agent 字符串的格式十分简单：

`Mozilla/AppVersion (Platform; Security [: OS-or-CPU-Description])`

例如，运行于 Windows 95 上的 Netscape Navigator 3.0 会有以下 user-agent 字符串：

`Mozilla/3.0 (Win95; I)`

I 表示这个浏览器安全性较低，相对应的是 N 表示没有安全性，U 代表强 128 位加密安全性（在美国大部分现在的浏览器都有 128 位安全性）。当运行于 Windows 时，Netscape 没有给出最后一个包含操作系统或者 CPU 的描述。

那之后，很快，微软又引入了 IE 3.0，并提供了一个和 Netscape Navigator 完全兼容的 user-agent 字符串。为此，IE 的 user-agent 字符串同样以字符串 Mozilla 开头，这样任何检查它（当时针对 Netscape 进行检查是一种标准）的服务器就会允许 IE 浏览页面。

IE 3.0 的 user-agent 字符串有如下格式：

`Mozilla/2.0 (compatible; MSIE [IEVersion]; [OS])`

例如，运行于 Windows 95 上的 IE 3.02 有如下 user-agent 字符串：

`Mozilla/2.0 (compatible; MSIE 3.02; Windows 95)`

在这个例子中，IEVersion 是 3.02，OS 是 Windows 95。出于某些原因，微软使用了 Mozilla/2.0 而没有使用 Mozilla/3.0。历史也不确定为什么会发生这种情况，尽管看上去很像是个疏忽。不幸的是，这个错误造成一连串的 user-agent 字符串混淆。

要理解这个问题，先考虑 navigator 对象的 appVersion 属性，它会返回 user-agent 字符串在第一个斜杠后的所有内容。对于 Netscape Navigator 3.0，appVersion 返回 3.0 (Win95; I)。这个值可以被传送到 parseFloat() 中以获取浏览器的版本号。然而，对于 IE 3.0，appVersion

返回的是 2.0 (compatible; MSIE 3.02; Windows 95)。传送给 parseFloat() 函数将返回 2.0，这是不正确的。

本来，开发人员希望只用一个算法就能检测 3.0 的浏览器，例如：

```
if (parseFloat(navigator.appVersion) >= 3) {
    //do 3.0-level stuff here
}
```

因为 IE 的格式，这个算法必须改成：

```
if (navigator.userAgent.indexOf("MSIE") > -1) {

    //IE, now check the version
    if (navigator.userAgent.indexOf("MSIE 3.") > -1) {
        //do IE 3.0 browser stuff here
    }

} else if (parseFloat(navigator.appVersion) >= 3) {
    //do other 3.0 browser stuff here
}
```

当根据 user-agent 字符串判断操作系统的时侯，另一个问题又出现了。因为 Netscape 和微软用了不同的字符串来表示同一个操作系统，所以对每个操作系统必须进行两次检验，例如：

```
var isWin95 = navigator.userAgent.indexOf("Win95") > -1 ||
    navigator.userAgent.indexOf("Windows 95") > -1;
```

但这仅仅是麻烦的开始。

228

8.3.2 Netscape Communicator 4.0 与 IE 4.0

Netscape Communicator 4.0 于 1997 年 8 月发布（从这个版本起名字从 Navigator 变成 Communicator）。Netscape 仍然保持原来的 user-agent 字符串格式：

Mozilla/AppVersion (Platform; Security [; OS-or-CPU-Description])

如果 4.0 运行在 Windows 98 的机器上，user-agent 字符串应该像这样：

Mozilla/4.0 (Win98; I)

而且随着 Netscape 发布一些补丁修补它的浏览器，AppVersion（可以通过 navigator.appVersion 来访问）也相应增加，4.79 版的 user-agent 字符串：

Mozilla/4.79 (Win98; I)

由于 Netscape 的信用，检测 Netscape Communicator 版本的方法依然同原来一样。

IE 4.0 在很短的时间后发布了，这次微软帮了开发人员一个大忙，更新了 user-agent 字符串，将 Mozilla 的版本改为 4.0（这之后都和 Netscape 的最新浏览器相对应）。除了这个小的改动外，IE 都和原来的 user-agent 字符串格式保持一致：

Mozilla/4.0 (compatible; MSIE [IEVersion]; [OS])

例如，运行在 Windows 98 上的 IE 4.0 返回如下 user-agent 字符串：

```
Mozilla/4.0 (compatible; MSIE 4.0; Windows 98)
```

这个更改可以用一个很简单的算法来确定这个浏览器版本是否为 4.0：

```
if (parseFloat(navigator.appVersion) >= 4) {
    //do 4.0-level stuff here
}
```

虽然 IE 4.0 是唯一在 Windows 平台上发布的 4.0 系列的浏览器，但是之后不久微软却为 MacOS 发布了 IE 4.5。这让我们得以一窥 IE 未来的 user-agent 字符串格式。

MacOS 的 IE 4.5 和 IE 4.0 的 user-agent 字符串格式保持一致，但是更新了 IE 的版本号：

```
Mozilla/4.0 (compatible; MSIE 4.5; Mac_PPC)
```

浏览器版本是 4.5，但是 Mozilla 版本却是 4.0，这又迫使开发人员调整它们的算法来检测 IE/Mac。这一点很重要，必须记住。
229

8.3.3 IE 5.0 及更高版本

微软于 1999 年发布了它的下一个 IE 版本，IE 5.0。同预期一样，user-agent 字符串还是存在同目前一样的问题。例如，在 Windows NT 4.0 上运行的 IE 5.0 返回这个 user-agent 字符串：

```
Mozilla/4.0 (compatible; MSIE 5.0; Windows NT)
```

这次还是，IE 版本更新了，但是 Mozilla 版本仍为 4.0。

这个问题在 5.5 和 6.0 版本发布时依然如故，最后让 6.0 的 user-agent 字符串变成这样：

```
Mozilla/4.0 (compatible; MSIE 6.0; Windows NT)
```

正因为如此，必须对 IE 进行单独的检测。

8.3.4 Mozilla

作为 Netscape 6 (Mozilla) 开发的一部分，开发人员起草了一份简短的文档作为 user-agent 字符串的标准。新的格式代表了它首次同 Netscape 原来的 user-agent 字符串格式的分离。

```
Mozilla/MozillaVersion ( Platform ; Security ; OS-or-CPU ; Localization information
? [; PrereleaseVersion] *[; Optional Other Comments] ) Gecko/GeckoVersion
[ApplicationProduct/ApplicationProductVersion]
```

很明显，很多方面考虑到了 user-agent 字符串格式。user-agent 字符串中每一个部分的解释都列在了下表中：

字符串	是否必须	描述
Mozilla Version	是	Mozilla 的版本
Platform	是	使用的操作系统的类型。可能的值是：Windows、Macintosh、X11（Unix 上的）

(续)

字符串	是否必须	描述	
Security	是	浏览器的安全性。可能的值: N (没有安全性) , U (高度安全性) 、 I (弱安全性)	
OS-or-CPU	是	浏览器运行的操作系统或者是运行浏览器的计算机的处理器类型。如果 Platform 是 Windows, 那么这是 Windows 的版本 (如 WinNT、Win95 等等)。如果 Platform 是 Macintosh, 那么这是 CPU 的类型 (68k 或者 PPC 代表 PowerPC)。如果 Platform 是 X11, 这是 Unix 操作系统的名称从 Unix 命令 uname-sm 中获得	
Localization information	是	浏览器用的语言。典型的是美国使用 en-US	230
Prerelease Version	否	用于这个浏览器的开放源代码的 Mozilla 代码的基础版本。注意: 这个一直未被使用, 直到 Mozilla 0.9.2 (Netscape 6.1)	
Optional Other Comments	否	这个是留给各自的 Mozilla 实现添加额外信息的空间	
GeckoVersion	是	使用的 Gecko 渲染引擎的版本。这是按照 yyyyymmdd 格式的日期	
Application Product	否	使用 Mozilla 代码的品牌浏览器的名称。在 Netscape 6 发行版中, 这个是 Netscape6; Netscape 7 改成 Netscape	
Application Product Version	否	使用 Mozilla 代码的品牌浏览器的版本	

为了能完全理解具体的内容, 现看一下运行于 Windows XP 的 Netscape 6.2.1 的例子:

Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:0.9.4) Gecko/20011128
Netscape6/6.2.1

匹配每一个不同部分的信息:

- MozillaVersion 是 5.0;
- Platform 是 Windows;
- Security 是 U;
- OS-or-CPU 是 Windows NT 5.1;
- Localization Information 是 en-US;
- PrereleaseVersion 是 rv:0.9.4;
- GeckoVersion 是 20011128;
- ApplicationProduct 是 Netscape 6;
- ApplicationProductVersion 是 6.2.1。

这看上去都很明显。那么为什么 Netscape 6.0 描述的 MozillaVersion 是 5.0 呢? 好像没人清楚为什么会出现这个情况, 最安全的推测是这是延期, 因为下一个计划发布的 Netscape 版本是 5.0。

Netscape 7.1 是 Netscape 系列浏览器的最后一个版本。AOL 更新了它与微软的许可证条款来使 IE 作为 AOL 软件内置的浏览器, 然后就解散了 Netscape 小组。现在 Mozilla 项目依然在发布自己新版本的浏览器, 同时还有一个更易用的版本, 叫做 Firefox。

230

231

8.3.5 Opera

在大多数操作系统上，除了 IE 和 Mozilla 之外最强大的选择就是 Opera 了。

Opera 的 user-agent 字符串的格式很独特。基本的 user-agent 格式如下：

```
Opera/AppVersion (OS; Security) [Language]
```

在 Windows XP 计算机上使用 Opera 7.54，user-agent 字符串如下：

```
Opera/7.54 (Windows NT 5.1; U) [en]
```

Opera 用一个独特的 user-agent 字符串来正确地（并简单地）确定它的 Web 浏览器。但问题也出现在另一个独特的浏览器功能中：可以将其伪装成另一种浏览器的能力。

在 7.0 版之前，Opera 可以解释 Windows 操作系统字符串的意思。例如，Windows NT 5.1 实际上就是 Windows XP，所以在 Opera 6.0 中，user-agent 字符串包含 Windows XP 而不是 Windows NT 5.1。为了能更加和标准兼容，7.0 版开始包含官方报告的操作系统版本而不是解释过的版本。

仅仅使用菜单中的功能，Opera 用户可以选择将浏览器伪装成某种 IE 和 Mozilla 的版本，包括旧一点的 Netscape 版本。Opera 只要更改它所报告的 user-agent 字符串以及调整它的一些其他特征（包括 `navigator` 对象的值）就可以办到这一点。然而，它并不是完全去模拟它伪装的浏览器，所以判断一个浏览器是不是由 Opera 伪装的也相当重要。

当 Opera 伪装成 Mozilla 5.0 时，它会返回这样的 user-agent 字符串：

```
Mozilla/5.0 (Windows NT 5.1; U) Opera 7.54
```

可以看到，应用程序的名字被换成 Mozilla，版本现在是 5.0，就和 Mozilla 的 user-agent 字符串一样。注意它在最后添加了字符串“Opera 7.54”，这就可以确定浏览器是 Opera。

如果 Opera 伪装成 Mozilla 4.78，user-agent 字符串如下：

```
Mozilla/4.78 (Windows NT 5.1; U) Opera 7.54
```

这和上面 Mozilla 5.0 的伪装看上去差不多只是 Mozilla 的版本更改了。类似的规则也适用于 Mozilla 3.0，它的 user-agent 字符串如下：

```
Mozilla/3.0 (Windows NT 5.1; U) Opera 7.54
```

如果 Opera 伪装成 IE 6.0，user-agent 字符串就变成这样：

```
Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1) Opera 7.54
```

在这个字符串中有重要的几项需要注意，如下：

- (1) Mozilla 的版本设为 4.0，就和 IE 6.0 一样。
- (2) 出现了字符串“compatible”。
- (3) 出现了字符串“MSIE”。

就是这三条模拟了 IE 5.0 的 user-agent 字符串。

8.3.6 Safari

2004 年, Apple 推出自己的浏览器 Safari。Safari 基于另一个叫做 KHTML 的开源项目, 这个项目基于 Unix 的 Konqueror 浏览器的主要组件。Apple 从 KHTML 创建了 Apple Web Kit, 给 Macintosh 的开发人员提供了他们的第一个官方 Web 技术平台。Safari 作为 Apple Web Kit 的一个应用程序而创建, 目前已经作为所有 MacOS X 的默认 Web 浏览器搭载在所有的零售版中。通过这个手段, Apple 立即占据了市场的一角, 而这本来需要花很长时间才能获得。

Safari 的 user-agent 字符串的基本格式如下:

```
Mozilla/5.0 (Platform; Security; OS-or-CPU; Language)
AppleWebKit/AppleWebKitVersion (KHTML, like Gecko) Safari/SafariVersion
```

例如:

```
Mozilla/5.0 (Macintosh; U; PPC Mac OS X; en) AppleWebKit/124 (KHTML, like Gecko)
Safari/125.1
```

可以看到, 这是一个很长的 user-agent 字符串。它不仅考虑了 AppleWebKit 的版本, 还加入了 Safari 的版本。关于是否将浏览器确定为 Mozilla 的争论焦点, 很快就因兼容性而解决。现在, 所有的 Safari 浏览器自称为 Mozilla 5.0, 和所有的 Mozilla 浏览器一样。Safari 版本一般是浏览器的构建号, 并不一定代表发布的版本号。所以尽管 Safari 1.25 在 user-agent 字符串中有一个数字 125.1, 但并不代表它们是一一对应的。

这个 user-agent 字符串最有趣的和最受争议的部分是, 在 Safari 的 pre-1.0 版本中加入了字符串 "(KHTML, like Gecko)"。Apple 遭到了很多开发者的反对, 他们认为这明显是要欺骗客户端和服务器, 让其认为 Safari 实际上是 Mozilla (好像添加 Mozilla/5.0 还不够)¹。Apple 的回应和当年 Microsoft 因 IE 的 user-agent 字符串备受攻击时的反应一样: Safari 和 Mozilla 是兼容的, 网站不应该因为 Safari 用户好像用的是不支持的浏览器就抵制它们。

8.3.7 结语

尽管 user-agent 字符串检测法在检测使用的浏览器上十分有效, 但它还需要进一步研究来得到更精确的结果。由于这个动荡的历史, 使得开发人员更喜欢对象/特征检测法, 而非 user-agent 字符串检测法。不过, user-agent 字符串检测法有很实用的方法, 确保你学习之后就能有效地运用它。

233

8.4 浏览器检测脚本

在本节中描述的浏览器检测脚本, 使用 user-agent 字符串检测法来确定以下一些浏览器:

1. Gecko 是 Mozilla 系列浏览器的引擎。——译者注

- Opera 4.0 及更高版本;
- IE 4.0 及更高版本;
- Mozilla 0.9.2 及更高版本;
- Safari 1.0 及更高版本;
- Netscape Navigator 4.0 – 4.8.x。

另外，本章中所使用的方法可能不完全支持 ECMAScript Edition 3 的老的浏览器，如果失败会直接跳过，不会产生 JavaScript 错误。

8.4.1 方法学

实际的做法是，最好检测最小的版本而不是直接检查准确的版本。例如，这段代码直接检测准确的版本：

```
if (isIE5 || isIE6) {
    //code
}
```

现在可能还看不出问题，但如果 IE 升级到了版本 10 呢？可能就要不断地给代码添加内容：

```
if (isIE5 || isIE6 || isIE7 || isIE8 || isIE9 || isIE10) {
    //code
}
```

显然这种做法是不理想的。然而，如果测试浏览器的最小版本，那么无论未来版本怎么发布，测试结果都一样：

```
if (isMinIE5) {
    //code
}
```

这个算法基本不用改变，它代表了本章中开发的浏览器检测代码所使用的方法。

8.4.2 第一步

浏览器检测必要的前两步是，将 user-agent 字符串保存到本地变量中，并获得浏览器版本：

```
var sUserAgent = navigator.userAgent;
var fAppVersion = parseFloat(navigator.appVersion);
```

在 user-agent 字符串中常出现的问题是，版本号有多个小数点（例如，Mozilla 0.9.2）。这就给比较浏览器版本制造了麻烦。我们知道 `parseFloat()` 函数可以用于将字符串转换成浮点数。另外，`parseFloat()` 是这样运行的：逐个读取字符串中的字符，当它发现一个非数字符时就停止。如果出现版本号有多个小数点的情况，非数字符就是第二个小数点。也就是说，使用 `parseFloat()` 解析字符串 "0.9.2" 返回的是浮点数 0.9，.2 的部分完全没有了，这并不好。

比较这种类型的两个版本字符串的最好的方法是，分别比较小数点分开的各部分。例如，假设要确定 0.9.2 是不是比 0.9.1 高。正确的做法是分别比较 0 和 0、9 和 9 以及 2 和 1。因为 2 比 1

大，所以版本 0.9.2 高于 0.9.1。因为检测浏览器和操作系统版本时，要经常进行这个操作，所以将这个逻辑封装到一个函数中是合理的。

函数 compareVersions() 接受两个版本字符串作为参数，如果相等，则返回 0；如果第一个大于第二个，则返回 1；如果第一个小于第二个，则返回 -1。（同本书前面一样，这是表示两个版本之间关系的最常见方法。）

函数的第一步是将两个版本号转换成数组。最快的方法就是使用 split() 方法，并将小数点作为字符的分割符：

```
function compareVersions(sVersion1, sVersion2) {
    var aVersion1 = sVersion1.split(".");
    var aVersion2 = sVersion2.split(".");
}
```

这时，aVersion1 包含传入的第一个版本的各个数字，aVersion2 包含第二个版本中的各个数字。下面，必须保证数组中包含的都是数字，且个数一样；否则，很难将 0.8.4 和 0.9 进行比较。这样，首先决定哪个数组中的数字更多，然后再给另外一个添加 0。这样 0.9 就会变成 0.9.0：

```
function compareVersions(sVersion1, sVersion2) {
    var aVersion1 = sVersion1.split(".");
    var aVersion2 = sVersion2.split(".");

    if (aVersion1.length > aVersion2.length) {
        for (var i=0; i < aVersion1.length - aVersion2.length; i++) {
            aVersion2.push("0");
        }
    } else if (aVersion1.length < aVersion2.length) {
        for (var i=0; i < aVersion2.length - aVersion1.length; i++) {
            aVersion1.push("0");
        }
    }
}
```

235

突出显示的代码包含一个 if 语句，用于测试哪个数组中的项目更多（如果数量相等，就不需要做任何改动）。两个 if 语句是对不同的数组做相同的事情。第一个分支给 aVersion2 添加 0，第二个分支给 aVersion1 添加 0。之后，两个数组都具有相等数目的数字。

最后一步是对数组迭代，比较每个数组中对应的数字：

```
function compareVersions(sVersion1, sVersion2) {
    var aVersion1 = sVersion1.split(".");
    var aVersion2 = sVersion2.split(".");

    if (aVersion1.length > aVersion2.length) {
        for (var i=0; i < aVersion1.length - aVersion2.length; i++) {
            aVersion2.push("0");
        }
    }
```

```

        }
    } else if (aVersion1.length < aVersion2.length) {
        for (var i=0; i < aVersion2.length - aVersion1.length; i++) {
            aVersion1.push("0");
        }
    }

    for (var i=0; i < aVersion1.length; i++) {
        var iVal1 = parseInt(aVersion1[i], 10);
        var iVal2 = parseInt(aVersion2[i], 10);

        if (iVal1 < iVal2) {
            return -1;
        } else if (iVal1 > iVal2) {
            return 1;
        }
    }

    return 0;
}

```

每个数组的值首先转换为整数以进行比较。在这部分中，`for` 循环用来比较数组。如果在 `aVersion1` 中的一个数字小于在 `aVersion2` 中相应的数字，函数就直接退出返回-1。类似的，如果 `aVersion1` 中的数字大于 `aVersion2` 中对应的，那么函数直接退出返回 1。如果测试了所有的数字，没有返回值，那么函数自动返回一个 0，表示两个版本相等。

可以这样使用函数：

```

alert(compareVersions("0.9.2", "0.9"));           //returns 1
alert(compareVersions("1.13.2", "1.14"));          //returns -1
alert(compareVersions("5.5", "5.5"));              //returns 0

```

第一行代码返回 1 因为 0.9.2 大于 0.9；第二行代码返回-1，因为 1.13.2 小于 1.14；第三行代

236 码返回 0，因为两个版本是相等的。在本章中，会经常用到这个函数。

8.4.3 检测 Opera

开始浏览器检测的最简单且最好的方法是从问题浏览器着手，比如 Opera 和 Safari。如果判断出浏览器不是这其中之一，接下来就很容易判断浏览器是 IE 还是 Mozilla 了。

先考虑一下可能的 Opera 的 user-agent 字符串：

```

Opera/7.54 (Windows NT 5.1; U)
Mozilla/5.0 (Windows NT 5.1; U) Opera 7.54
Mozilla/4.78 (Windows NT 5.1; U) Opera 7.54
Mozilla/3.0 (Windows NT 5.1; U) Opera 7.54
Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1) Opera 7.54

```

立刻发现每个字符串中都有单词 "Opera"。那么最方便的判断浏览器是否为 Opera 的方法是在 user-agent 字符串中搜索有没有 Opera：

```
var isOpera = sUserAgent.indexOf("Opera") > -1;
```

也可以使用正则表达式来进行检测；然而，一些早期的浏览器并不支持正则表达式，所以执行时可能会产生错误。使用 `indexOf()` 保证了这一行代码可以正常运行，因为第一个版本的 JavaScript 中就包含了这个方法。

知道用的是 Opera 浏览器后，就可以继续判断实际使用的版本。第一步是定义几个变量，用于测试不同版本的 Opera。

要确定使用的 Opera 是什么版本，可以定义一些变量：

```
var isMinOpera4 = isMinOpera5 = isMinOpera6 = isMinOpera7 = isMinOpera7_5 = false;
```

这一句代码使用组合赋值来给每个变量都设一个初值 `false`，保证如果浏览器是 Netscape 或其他，这些值能正确返回 `false`。

自然地，除非浏览器被鉴定为 Opera，否则不用再去设置这些值，所以任何接下来的浏览器版本的计算过程就发生在 `if` 语句中：

```
if (isOpera) {
    //version detection here
}
```

由于 Opera 可以有伪装，有两种不同的方法来确定浏览器的版本。如果 Opera 使用自己的 `user-agent` 字符串，那么版本已经包含在 `fAppVersion` 中了，这前面已经定义过。可以通过检查 `navigator.appName` 来判断 Opera 是否使用了伪装；如果这个值等于 "Opera"，那么浏览器并没有使用伪装。

先定义一个变量来保存 Opera 的版本，命名为 `fOperaVersion`。然后，要检查 Opera 是不是使用了伪装。如果没有，直接将 `fAppVersion` 的值赋给 `fOperaVersion`：

```
if (isOpera) {
    var fOperaVersion;
    if (navigator.appName == "Opera") {
        fOperaVersion = fAppVersion;
    }
}
```

当 Opera 使用了伪装，情况就有点复杂。这样的话，就需要使用 `user-agent` 字符串并使用正则表达式来抽取版本号：

```
var reOperaVersion = new RegExp("Opera (\d+\.\d+)");
```

这个正则表达式捕获了 Opera 的版本，它应该是一个或多个数字，并跟着小数点，后面还有一个或多个数字。注意，这个正则表达式使用了构造函数方法，所以 `\d` 和 `.` 都必须使用双重转义。使用构造函数方法是为了保持向下兼容。即便浏览器不是 Opera，这段代码不会被执行，但是浏览器可能并不支持正则表达式字面量格式（它会破坏 ECMAScript Edition 1 的语法），这就会产生错误。

使用正则表达式的 `test()` 方法测试并将版本号存储在 `RegExp.$1` 中，最好使用 `RegExp["$1"]` 来表示，确保不会破坏旧的 JavaScript 语法。

```

if (isOpera) {
    var fOperaVersion;
    if(navigator.appName == "Opera") {
        fOperaVersion = fAppVersion;
    } else {
        var reOperaVersion = new RegExp("Opera (\d+\.\d+)");
        reOperaVersion.test(sUserAgent);
        fOperaVersion = parseFloat(RegExp["$1"]);
    }
}

```

因为 Opera 几乎在 4.0 (这是这里我们所需检测的最早版本) 之前就支持了正则表达式, 所以我们可以在 `if` 语句中使用正则表达式。

这时, Opera 的版本已存放到 `fOperaVersion` 中。唯一剩下的事情是给变量赋值:

```

if (isOpera) {
    var fOperaVersion;
    if(navigator.appName == "Opera") {
        fOperaVersion = fAppVersion;
    } else {
        var reOperaVersion = new RegExp("Opera (\d+\.\d+)");
        reOperaVersion.test(sUserAgent);
        fOperaVersion = parseFloat(RegExp["$1"]);
    }
    isMinOpera4 = fOperaVersion >= 4;
    isMinOpera5 = fOperaVersion >= 5;
    isMinOpera6 = fOperaVersion >= 6;
    isMinOpera7 = fOperaVersion >= 7;
    isMinOpera7_5 = fOperaVersion >= 7.5;
}

```

238

这就完成了浏览器检测代码的第一部分。有了这部分代码后, 就可以判断浏览器是否为 Opera, 如果是, 那么又是什么版本。接下来的是另外一个问题浏览器: Safari。

8.4.4 检测 Konqueror/Safari

Konqueror 和 Safari 都是基于 KHTML 项目的, 所以可以认为它们是一样的。问题是, 无法获知浏览器到底使用的是什么版本的 KHTML。于是, 只能检测是否使用了 KHTML, 但这还是需要依赖浏览器的版本号来判断浏览器的功能。

先看一些基于 KHTML 的 user-agent 字符串:

```

Mozilla/5.0 (compatible; Konqueror/2.2.2; SunOS)
Mozilla/5.0 (compatible; Konqueror/3; Linux; de, en_US, de_DE)
Mozilla/5.0 (compatible; Konqueror/3.1; Linux 2.4.20)
Mozilla/5.0 (compatible; Konqueror/3.2; FreeBSD) (KHTML, like Gecko)
Mozilla/5.0 (Macintosh; U; PPC Mac OS X; en) AppleWebKit/51 (KHTML, like Gecko) Safari/51
Mozilla/5.0 (Macintosh; U; PPC Mac OS X; es-es) AppleWebKit/106.2 (KHTML, like
Gecko) Safari/100.1
Mozilla/5.0 (Macintosh; U; PPC Mac OS X; en) AppleWebKit/124 (KHTML, like Gecko)
Safari/125.1

```

前四个字符串是出自 Konqueror；最后三个是来自 Safari。注意里面的一些细节。首先，不是所有的 user-agent 字符串都包含 "KHTML"，所以必须查找 "Konqueror" 和 "AppleWebKit" 或者 "Safari" 以及 "KHTML" 字符串。Apple 建议查找 "AppleWebKit" 而不是 "Safari"，因为可能有一些开发人员会将 Apple Web Kit 嵌入到他们的程序中，来创建其他的浏览器。第二，Konqueror 版本号和 Apple Web Kit 或者 Safari 版本号没有任何关系。

所以，必须首先判断浏览器是否基于 KHTML 的：

```
var isKHTML = sUserAgent.indexOf("KHTML") > -1
    || sUserAgent.indexOf("Konqueror") > -1
    || sUserAgent.indexOf("AppleWebKit") > -1;
```

在设置了 isKHTML 后，接下来就可以判断，使用的是何种 KHTML 浏览器。

```
if (isKHTML) {
    isSafari = sUserAgent.indexOf("AppleWebKit") > -1;
    isKong = sUserAgent.indexOf("Konqueror") > -1;
}
```

239

下面，为不同版本的浏览器设置变量：

```
var isMinSafari1 = isMinSafari1_2 = false;
var isMinKong2_2 = isMinKong3 = isMinKong3_1 = isMinKong3_2 = false;
```

可以通过解释 Safari 的构建号或者 Apple Web Kit 的版本号，来判断 Safari 的版本号。正如前面提到的，Apple 建议你只使用 Apple Web Kit 的信息。Safari 1.0 使用 Apple Web Kit 85 版，Safari 1.2 使用 124 版。要获得这个信息，还是必须使用正则表达式。

回顾一下本节开始时介绍的 user-agent 字符串，可以看到 Apple Web Kit 版本可以有小数，但不全是。这就让正则表达式变得有些复杂：

```
var reAppleWebKit = new RegExp("AppleWebKit\\/(\\d+(?:\\.\\d*)?)");
```

这个表达式使用非捕获性分组来包含小数点和后面的数字。而捕获性分组返回版本号：

```
if (isKHTML) {
    isSafari = sUserAgent.indexOf("AppleWebKit") > -1;
    isKong = sUserAgent.indexOf("Konqueror") > -1;

    if (isSafari) {
        var reAppleWebKit = new RegExp("AppleWebKit\\/(\\d+(?:\\.\\d*)?)");
        reAppleWebKit.test(sUserAgent);
        var fAppleWebKitVersion = parseFloat(RegExp["$1"]);

        isMinSafari1 = fAppleWebKitVersion >= 85;
        isMinSafari1_2 = fAppleWebKitVersion >= 124;
    }
}
```

要判断 Konqueror 的版本，用正则表达式也有点复杂，因为 Konqueror 使用的版本号中，可能有一个或两个或没有小数点。正因如此，必须使用多个非捕获性分组才能匹配所有的可能性：

```
var reKong = new RegExp("Konqueror\\/(\\d+(?:\\.\\d+(?:\\.\\d)?)?)");
```

这个正则表达式表示匹配字符串 "Konqueror"，跟着一个斜杠，再跟着至少一个数字，它后面可以（也可以不）跟着小数点和一个或多个数字，而这之后也可以（或没有）跟着小数点和多个数字。

在获取这个值后，必须使用 `compareVersion()` 函数来测试，它才能判断最小的浏览器版本号。

```
if (isKHTML) {
    isSafari = sUserAgent.indexOf("AppleWebKit") > -1;
    isKonq = sUserAgent.indexOf("Konqueror") > -1;

    if (isSafari) {
        var reAppleWebKit = new RegExp("AppleWebKit\\/(\\d+(?:\\.\\d*)?)");
        reAppleWebKit.test(sUserAgent);
        var fAppleWebKitVersion = parseFloat(RegExp["$1"]);

        isMinSafari1 = fAppleWebKitVersion >= 85;
        isMinSafari1_2 = fAppleWebKitVersion >= 124;
    } else if (isKonq) {

        var reKonq = new RegExp("Konqueror\\/(\\d+(?:\\.\\d+(?:\\.\\d*)?)?)");
        reKonq.test(sUserAgent);
        isMinKonq2_2 = compareVersions(RegExp["$1"], "2.2") >= 0;
        isMinKonq3 = compareVersions(RegExp["$1"], "3.0") >= 0;
        isMinKonq3_1 = compareVersions(RegExp["$1"], "3.1") >= 0;
        isMinKonq3_2 = compareVersions(RegExp["$1"], "3.2") >= 0;
    }
}
```

在这部分代码中，检查 `compareVersions()` 返回的值是否为大于或等于 0，以表示第一个版本等于或大于第二个。

对基于 KHTML 的浏览器的检测就完成了。如果无所谓使用的是哪个浏览器，也可以使用 `isKHTML`，或者使用更加准确的变量来检测浏览器及其版本。

8.4.5 检测 IE

正如前面所讨论的，IE 的 user-agent 字符串也很独特。回想一下 IE 6.0 的 user-agent 字符串：

Mozilla/4.0 (compatible; MSIE 6.0; Windows NT)

和其他浏览器对比，就发现两个部分十分特别：“`compatible`”和“`MSIE`”。这是检测 IE 的基础：

```
var isIE = sUserAgent.indexOf("compatible") > -1
&& sUserAgent.indexOf("MSIE") > -1;
```

看上去很明显，但是还存在一个问题。再看一下 Opera 伪装成 IE 6.0 时的 user-agent 字符串：

Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1) Opera 7.54

看出问题了吗？如果只检查“`compatible`”和“`MSIE`”，那么 Opera 伪装成的 IE 就会通过检验。解决方法是使用 `isOpera` 变量（已经在前面解释过）来确保检测的准确：

```
var isIE = sUserAgent.indexOf("compatible") > -1
&& sUserAgent.indexOf("MSIE") > -1
&& !isOpera;
```

下面，定义几个 IE 不同版本的变量：

```
var isMinIE4 = isMinIE5 = isMinIE5_5 = isMinIE6 = false;
```

就像判断伪装的 Opera 版本一样，我们用正则表达式从 user-agent 字符串中取出 IE 的版本：

```
var reIE = new RegExp("MSIE (\d+\.\d+);");
```

这个模式查找一个或多个数字，跟着一个小数点，然后跟着另一些数字。把表达式放到函数中，获得以下代码：

```
if (isIE) {
    var reIE = new RegExp("MSIE (\d+\.\d+);");
    reIE.test(sUserAgent);
    var fIEVersion = parseFloat(RegExp["$1"]);

    isMinIE4 = fIEVersion >= 4;
    isMinIE5 = fIEVersion >= 5;
    isMinIE5_5 = fIEVersion >= 5.5;
    isMinIE6 = fIEVersion >= 6.0;
}
```

这就完成了 IE 的检测。这段代码在 Windows 上和 Macintosh 上都可以有效地运行。下面来讨论 IE 的主要竞争对手——Mozilla。

8.4.6 检测 Mozilla

到目前为止，你应该知道该怎么做了。回顾一下 Mozilla 的 user-agent 字符串：

```
Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:0.9.4) Gecko/20011128
Netscape6/6.2.1
```

为了全面考虑，先看一下 Opera 伪装成 Mozilla 5.0 时的 user-agent 字符串：

```
Mozilla/5.0 (Windows NT 5.1; U) Opera 7.54
```

幸好，有很多方法来判断是否为 Mozilla。最突出的东西很明显地摆在眼前：Mozilla 的 user-agent 字符串中有 "Gecko"。再看一下 Opera 伪装的 Mozilla 5.0，Gecko 并没有出现。有了！这就让事情变得很简单了：

```
var isMoz = sUserAgent.indexOf("Gecko") > -1;
```

直到最近，这对判断浏览器是否为 Mozilla 已经足够了。然而，正如前面所见，基于 KHTML 的浏览器的 user-agent 字符串包含 "like Gecko"，这样在前面的测试就会返回 true。所以必须保证浏览器包含 "Gecko" 但又不是基于 KHTML 的。

```
var isMoz = sUserAgent.indexOf("Gecko") > -1
&& !isKHTML;
```

242 现在 isMoz 变量变得更加精确，所以，我们继续来查看如何获取版本。

根据计划支持的浏览器，可以有多种不同的 Mozilla 版本。例如，Netscape 7 基于 Mozilla 1.0，Netscape 7.1 基于 Mozilla 1.4。所以，最好也测试这些版本，以防万一有些用户仍然在使用 Netscape 系列的浏览器。Mozilla 1.5 也比较流行，所以最好也算在其中：

```
var isMinMoz1 = sMinMoz1_4 = isMinMoz1_5 = false;
```

然后，必须从 user-agent 字符串中取出实际的版本号。在 Mozilla 的 user-agent 字符串中，Mozilla 的版本位于 "rv：" 文本后，且可以包含一个或两个小数点，所以要用到非捕获性分组：

```
var reMoz = new RegExp("rv:(\\d+\\.\\d+(?:\\.\\d+)?)");
```

然后再使用 compareVersions() 函数就可以很容易地检测 Mozilla 版本号了：

```
if (isMoz) {
    var reMoz = new RegExp("rv:(\\d+\\.\\d+(?:\\.\\d+)?)");
    reMoz.test(sUserAgent);
    isMinMoz1 = compareVersions(RegExp["$1"], "1.0") >= 0;
    isMinMoz1_4 = compareVersions(RegExp["$1"], "1.4") >= 0;
    isMinMoz1_5 = compareVersions(RegExp["$1"], "1.5") >= 0;
}
```

最后的任务是检测 Mozilla 的祖先：原始的 Netscape 浏览器。

8.4.7 检测 Netscape Communicator 4.x

尽管 Netscape Communicator 已成为昨日的恐龙，但仍然存在一些它的用户。

我们先回忆一下 Netscape Communicator 4.79 的 user-agent 字符串。

Mozilla/4.79 (Win98; I)

可以看到，user-agent 字符串并没有明确说明这是 Netscape Communicator。而且其他的浏览器都在它们的 user-agent 字符串中包含 "Mozilla"，所以不能仅仅靠这个进行判断。检测 Netscape Communicator 4.x 的方法是曾由福尔摩斯使用过的：如果排出所有的不可能，那么剩下的，尽管令人难以置信，但一定是真的。根据浏览器检测的目的，这意味着，你必须首先判断用户用的不是其他浏览器：

```
var isNS4 = !isIE && !isOpera && !isMoz && !isKHTML;
```

目前为止，很好。下面还有三个额外的内容要做检查：

(1) "Mozilla" 字符串位于 user-agent 字符串的开头（位置 0）。

(2) Navigator.appName 的值是 "Netscape"。

(3) Navigator.appVersion 的值大于等于 4.0，但小于 5.0（这个值已经存入变量

243 fAppVersion 中，该变量在代码的一开始就创建了）。

把这些检查工作加入到代码中，就得到以下代码：

```

var isNS4 = !isIE && !isOpera && !isMoz && !isKHTML
&& (sUserAgent.indexOf("Mozilla") == 0)
&& (navigator.appName == "Netscape")
&& (fAppVersion >= 4.0 && fAppVersion < 5.0);

```

通过这个变量，就可以准确判断使用的浏览器是否为 Netscape Communicator。下面拟要判断最小版本。这次，要检查版本 4.0、4.5（主要的发布版，代码做了大量改进）4.7（另一个主要发布版）和 4.8（最后的发布版）。

```
var isMinNS4 = isMinNS4_5 = isMinNS4_7 = isMinNS4_8 = false;
```

同时，因为 Netscape Communicator 存储版本号的方法比较合理，可以很容易地进行判断：

```

if (isNS4) {
    isMinNS4 = true;
    isMinNS4_5 = fAppVersion >= 4.5;
    isMinNS4_7 = fAppVersion >= 4.7;
    isMinNS4_8 = fAppVersion >= 4.8;
}

```

第一个变量，`isMinNS4`，被自动设为 `true`，因为在计算 `isNS4` 时，这就已经确定了。其他最小版本的检测方式很正常。

这就完成了脚本的浏览器检测。下面就是平台和操作系统检测。

8.5 平台/操作系统检测脚本

之前已经在浏览器检测上钻研到很高的程度了，现在必须面对另一个挑战：指出客户端机器的操作系统。即使浏览器公司说，它们的浏览器在不同的平台和操作系统上运行得都一样，但实际上还是有差别的。

例如，IE。在 Windows 上，有很强大的接口可以让 Microsoft ActiveX 控件嵌入到页面中或者在 JavaScript 中使用。问题就在于，ActiveX 控件只能在 Windows 上运行。所以，即便微软说它的 IE 能运行在 Macintosh 上，甚至 Unix 上，而且与在 Windows 上运行的一样，但这是不可能的。于是，必须至少能分辨出你面对的是什么操作系统以便判断是否该做些特殊的调整。

8.5.1 方法学

判断操作系统应该首先从查找平台开始。针对本书的目的，平台分成三大类：Windows、Macintosh 和 Unix。

在确定平台后，就可以判断一些操作系统信息了。对于 Windows 和 Unix，可以获得操作系统的版本。然而，对于 Macintosh 就不行。Macintosh 平台提供的信息只有处理器是 68000 或者是 PowerPC，尽管 Safari 包含字符串“MacOS X”（不过，Safari 只能运行在 Mac OS X 上，这还有什么帮助？）。

一般来说，合适的 JavaScript 判断对于单独判断平台已经足够。然而，有时候，额外的操作系统信息也很重要，脚本就必须提供那些信息。

8.5.2 第一步

那么如何判断客户端用户的平台呢？这次，还是由 `navigator` 对象以及它的 `platform` 属性出面来解围。与平时一样，事情不会像看上去那样简单。实际上，每个浏览器对 `navigator.platform` 提供了不同的信息。例如，IE 和 Netscape Communicator 对于 Windows 32 位操作系统返回 "Win32"，对 Macintosh 系统返回 "Mac68k" 或者 "MacPPC"（根据处理器的型号）。对 Unix 操作系统它返回操作系统实际的名字。从另一方面来说，Mozilla 对所有的 Windows 系统都返回 "Windows"，对所有的 Macintosh 系统都返回 "Macintosh"，对所有的 Unix 系统都返回 "X11"。所以，要检查用户平台，还需要检查很多选项。

检查 Windows 和 Macintosh 系统很简单，你只需检查不同的字符串：

```
var isWin = (navigator.platform == "Win32") || (navigator.platform == "Windows");
var isMac = (navigator.platform == "Mac68K") || (navigator.platform == "MacPPC")
           || (navigator.platform == "Macintosh");
```

因为浏览器对于使用的 Unix 返回的信息因人而异，所以首先必须保证平台不是 Windows，也不是 Macintosh，然后检查是不是 "X11"：

```
var isUnix = (navigator.platform == "X11") && !isWin && !isMac;
```

在知道面对的是什么平台后，就可以开始判断使用的是什么操作系统了。

8.5.3 检测 Windows 操作系统

似乎每隔几年就会发布新的 Windows 版本。很久以前，微软有两个独立的 Windows 版本：一个针对家庭使用，另一个针对商业使用。家用版就简单的叫做 Windows。而商业版叫做 Windows NT。在两个版本之间有些重叠。不过，用户界面不同。2001 年，微软决定将两者整合成一个新的产品，Windows XP。这个新的操作系统结合了 Windows NT 的安全性和稳定性以及传统 Windows 友好的用户界面。

245

简单介绍历史之后，有这些不同版本的 Windows 需要检测：

- Windows 95;
- Windows 98;
- Windows NT 4.0;
- Windows 2000;
- Windows ME;
- Windows XP。

幸好，操作系统信息已被包含在 `user-agent` 字符串中，也就是说由浏览器来决定要显示什么。下面的表格显示，根据使用的操作系统，不同的浏览器包含在 `user-agent` 字符串中的信息。

	IE4+	NS4x	Mozilla	Opera pre-6	Opera 7+
Windows 95	"Windows 95"	"Win95"	"Win95"	"Windows 95"	"Windows 95"

(续)

	IE4+	NS4x	Mozilla	Opera pre-6	Opera 7+
Windows 98	"Windows 98"	"Win98"	"Win98"	"Windows 98"	"Windows 98"
Windows NT 4.0	"Windows NT"	"WinNT"	"WinNT4.0"	"Windows NT 4.0"	"Windows NT 4.0"
Windows 2000	"Windows NT 5.0"	"Windows NT 5.0"	"Windows NT 5.0"	"Windows 2000"	"Windows NT 5.0"
Windows ME	"Win 9x 4.90"	"Win 9x 4.90"	"Win 9x 4.90"	"Windows ME"	"Win 9x 4.90"
Windows XP	"Windows NT 5.1"	"Windows NT 5.1"	"Windows NT 5.1"	"Windows XP"	"Windows NT 5.1"

现在的任务就同检测浏览器版本一样，先定义一些变量：

```
var isWin95 = isWin98 = isWinNT4 = isWin2K = isWinME = isWinXP = false;
```

为了判断每个 Windows 版本，必须根据前面的表格逐个检查 user-agent 字符串。例如，想检测 Windows98，则必须在 user-agent 字符串中查找 "Windows 98" 或者 "Win98"，这样就涵盖了四个浏览器。

先完成对 Windows 95、98、ME、2000 和 XP 的检查，因为它们都只有两种值：

```
if (isWin) {
    isWin95 = sUserAgent.indexOf("Win95") > -1
        || sUserAgent.indexOf("Windows 95") > -1;
    isWin98 = sUserAgent.indexOf("Win98") > -1
        || sUserAgent.indexOf("Windows 98") > -1;
    isWinME = sUserAgent.indexOf("Win 9x 4.90") > -1
        || sUserAgent.indexOf("Windows ME") > -1;
    isWin2K = sUserAgent.indexOf("Windows NT 5.0") > -1
        || sUserAgent.indexOf("Windows 2000") > -1;
    isWinXP = sUserAgent.indexOf("Windows NT 5.1") > -1
        || sUserAgent.indexOf("Windows XP") > -1;
}
```

246

Windows NT 4.0 就有点复杂，这是 IE 的错。在它的 user-agent 字符串中只包含了 "Windows NT"，这意味着不能仅仅搜索它。为什么？因为 IE 认为 Windows 2000 是 "Windows NT 5.0" 而 Windows XP 是 "Windows NT 5.1"。如果简单搜索 "Windows NT"，那么结果对这三者均为真，这可不是预期的结果。所以对于 Windows NT 4.0，必须搜索 "Windows NT"、"WinNT"、"WinNT4.0" 以及 "Windows NT 4.0" 并且确保它不是其他的 Windows 版本：

```
if (isWin) {
    isWin95 = sUserAgent.indexOf("Win95") > -1
        || sUserAgent.indexOf("Windows 95") > -1;
    isWin98 = sUserAgent.indexOf("Win98") > -1
        || sUserAgent.indexOf("Windows 98") > -1;
    isWinME = sUserAgent.indexOf("Win 9x 4.90") > -1
        || sUserAgent.indexOf("Windows ME") > -1;
    isWin2K = sUserAgent.indexOf("Windows NT 5.0") > -1
        || sUserAgent.indexOf("Windows 2000") > -1;
}
```

```

isWinXP = sUserAgent.indexOf("Windows NT 5.1") > -1
    || sUserAgent.indexOf("Windows XP") > -1;
isWinNT4 = sUserAgent.indexOf("WinNT") > -1
    || sUserAgent.indexOf("Windows NT") > -1
    || sUserAgent.indexOf("WinNT4.0") > -1
    || sUserAgent.indexOf("Windows NT 4.0") > -1
    && (!isWinME && !isWin2K && !isWinXP);
}

```

这样就可以成功检测不同的 Windows 操作系统了。

8.5.4 检测 Macintosh 操作系统

无论相信与否，这部分是我们在客户端机器的艰苦历程中最简单的部分了。以前，Macintosh 浏览器不会告诉你使用的操作系统；唯一提供的信息是，Macintosh 使用的是 68000 处理器还是 PowerPC 处理器。直到最近才有浏览器开始报告操作系统是 MacOS X，也就是说如果在 user-agent 字符串中找到 MacOS X，就可以确信这是 MacOS X 系统。但是，如果没有这个，也不能说明用户就没有使用 MacOS X，可能浏览器并没有报告它。因此，最好还是利用检测处理器的老办法。

下面的表格列出每个浏览器对不同的处理器包含在 user-agent 字符串中的字符串。

	IE	NS4	Mozilla	Opera
MacOS(68k)	"Mac_68000"	"68K"	"68K"	N/A
MacOS(PPC)	"Mac_PowerPC"	"PPC"	"PPC"	"Mac_PowerPC"

247

首先是定义变量：

```
var isMac68K = isMacPPC = false;
```

下面，检查 user-agent 字符串中不同的字符串：

```

if (isMac) {
    isMac68K = sUserAgent.indexOf("Mac_68000") > -1
        || sUserAgent.indexOf("68K") > -1;
    isMacPPC = sUserAgent.indexOf("Mac_PowerPC") > -1
        || sUserAgent.indexOf("PPC") > -1;
}

```

之后，就可以判断 Macintosh 平台。还需检测的平台就是 Unix。

8.5.5 检测 Unix 操作系统

某些情况下这是最好处理的平台；而另一些情况下，又是最棘手的。你可能已经知道，Unix 有相当多的款式。有 SunOS、HP-UX、AIX、Linux、IRIX，甚至更多。而每种又有不同的版本及不同的 user-agent 字符串表示。为避免啰嗦，本节将只关注检测 SunOS 及其一些版本。使用在这里和前面得到的信息，就应该有足够的知识来将脚本改造成检测其他 Unix 平台。

要判断一个指定 Unix 平台，例如 SunOS，只需搜索 user-agent 字符串中的一些特定子串。

实际上，这方面在 Unix 上来说，是很简单的，因为浏览器在 user-agent 字符串中包含 Unix 命令 uname -sm 的结果。这样，每一个浏览器对于同一个操作系统都返回相同的字符串。下面是关于 SunOS 的一些例子：

```
Mozilla/4.0 (compatible; MSIE 6.0; SunOS 5.6 sun4u)
Mozilla/5.0 (X11; U; SunOS 5.6 sun4u; en-US; rv:0.9.4) Gecko/20011128 Netscape6/6.2
Mozilla/4.7 [en] (X11; U; SunOS 5.6 sun4u)
Opera/6.0 (SunOS 5.6 sun4u; U) [en]
```

我们先定义代表要查找的不同版本的变量：

```
var isSunOS = isMinSunOS4 = isMinSunOS5 = isMinSunOS5_5 = false;
```

对于 SunOS，需要查找的字符串就是 "SunOS"：

```
if (isUnix) {
    isSunOS = sUserAgent.indexOf("SunOS") > -1;
}
```

下面，使用正则表达式来取出操作系统版本。因为 SunOS 使用两部分数字，表达式和 Mozilla 使用的类似：

```
var reSunOS = new RegExp("SunOS (\d+\.\d+(?:\.\d+)?)");
```

248

在获得版本号后，再使用 compareVersions() 函数（在前一节中解释过）来判断最小的操作系统版本：

```
if (isUnix) {
    isSunOS = sUserAgent.indexOf("SunOS") > -1;

    if (isSunOS) {
        var reSunOS = new RegExp("SunOS (\d+\.\d+(?:\.\d+)?)");
        reSunOS.test(sUserAgent);
        isMinSunOS4 = compareVersions(RegExp["$1"], "4.0") >= 0;
        isMinSunOS5 = compareVersions(RegExp["$1"], "5.0") >= 0;
        isMinSunOS5_5 = compareVersions(RegExp["$1"], "5.5") >= 0;
    }
}
```

之后，就可以使用 isSunOS、isMinSunOS4、isMinSunOS5 和 isMinSunOS5_5 了。

当然，除非遇到实际要用到它们的地方，这些浏览器和操作系统的知识也没什么用。

8.6 全部脚本

为了方便，在这里列出了全部的脚本。注意不同的检测过程出现的顺序是十分重要的。全部的脚本应该存放在一个 JavaScript 文件中，比如 detect.js。

```
var sUserAgent = navigator.userAgent;
var fAppVersion = parseFloat(navigator.appVersion);

function compareVersions(sVersion1, sVersion2) {
    var aVersion1 = sVersion1.split(".");
    var aVersion2 = sVersion2.split(".");
    var i = 0;
    while (i < aVersion1.length && i < aVersion2.length) {
        if (aVersion1[i] > aVersion2[i]) return 1;
        if (aVersion1[i] < aVersion2[i]) return -1;
        i++;
    }
    if (aVersion1.length > aVersion2.length) return 1;
    if (aVersion1.length < aVersion2.length) return -1;
    return 0;
}
```

```

var aVersion2 = sVersion2.split(".");
if (aVersion1.length > aVersion2.length) {
    for (var i=0; i < aVersion1.length - aVersion2.length; i++) {
        aVersion2.push("0");
    }
} else if (aVersion1.length < aVersion2.length) {
    for (var i=0; i < aVersion2.length - aVersion1.length; i++) {
        aVersion1.push("0");
    }
}
for (var i=0; i < aVersion1.length; i++) {
    if (aVersion1[i] < aVersion2[i]) {
        return -1;
    } else if (aVersion1[i] > aVersion2[i]) {
        return 1;
    }
}
return 0;
}

var isOpera = sUserAgent.indexOf("Opera") > -1;
var isMinOpera4 = isMinOpera5 = isMinOpera6 = isMinOpera7 = isMinOpera7_5 = false;

if (isOpera) {
    var fOperaVersion;
    if(navigator.appName == "Opera") {
        fOperaVersion = fAppVersion;
    } else {
        var reOperaVersion = new RegExp("Opera (\\"d+\\".\\"d+)");
        reOperaVersion.test(sUserAgent);
        fOperaVersion = parseFloat(RegExp["$1"]);
    }

    isMinOpera4 = fOperaVersion >= 4;
    isMinOpera5 = fOperaVersion >= 5;
    isMinOpera6 = fOperaVersion >= 6;
    isMinOpera7 = fOperaVersion >= 7;
    isMinOpera7_5 = fOperaVersion >= 7.5;
}

var isKHTML = sUserAgent.indexOf("KHTML") > -1
    || sUserAgent.indexOf("Konqueror") > -1
    || sUserAgent.indexOf("AppleWebKit") > -1;

var isMinSafari1 = isMinSafari1_2 = false;
var isMinKonq2_2 = isMinKonq3 = isMinKonq3_1 = isMinKonq3_2 = false;

if (isKHTML) {
    isSafari = sUserAgent.indexOf("AppleWebKit") > -1;
}

```

```

isKonq = sUserAgent.indexOf("Konqueror") > -1;

if (isSafari) {
    var reAppleWebKit = new RegExp("AppleWebKit\\/(\\d+(?:\\.\\d*)?)");
    reAppleWebKit.test(sUserAgent);
    var fAppleWebKitVersion = parseFloat(RegExp["$1"]);

    isMinSafari1 = fAppleWebKitVersion >= 85;
    isMinSafari1_2 = fAppleWebKitVersion >= 124;
} else if (isKonq) {

    var reKonq = new RegExp("Konqueror\\/(\\d+(?:\\.\\d+(?:\\.\\d*)?)?)");
    reKonq.test(sUserAgent);
    isMinKonq2_2 = compareVersions(RegExp["$1"], "2.2") >= 0;
    isMinKonq3 = compareVersions(RegExp["$1"], "3.0") >= 0;
    isMinKonq3_1 = compareVersions(RegExp["$1"], "3.1") >= 0;
    isMinKonq3_2 = compareVersions(RegExp["$1"], "3.2") >= 0;
}

var isIE = sUserAgent.indexOf("compatible") > -1
    && sUserAgent.indexOf("MSIE") > -1
    && !isOpera;

var isMinIE4 = isMinIE5 = isMinIE5_5 = isMinIE6 = false;

if (isIE) {
    var reIE = new RegExp("MSIE (\\d+\\.\\d+);");
    reIE.test(sUserAgent);
    var fIEVersion = parseFloat(RegExp["$1"]);

    isMinIE4 = fIEVersion >= 4;
    isMinIE5 = fIEVersion >= 5;
    isMinIE5_5 = fIEVersion >= 5.5;
    isMinIE6 = fIEVersion >= 6.0;
}

var isMoz = sUserAgent.indexOf("Gecko") > -1
    && !isKHTML;

var isMinMoz1 = isMinMoz1_4 = isMinMoz1_5 = false;

if (isMoz) {
    var reMoz = new RegExp("rv:(\\d+\\.\\d+(?:\\.\\d*)?)");
    reMoz.test(sUserAgent);
    isMinMoz1 = compareVersions(RegExp["$1"], "1.0") >= 0;
    isMinMoz1_4 = compareVersions(RegExp["$1"], "1.4") >= 0;
    isMinMoz1_5 = compareVersions(RegExp["$1"], "1.5") >= 0;
}

var isNS4 = !isIE && !isOpera && !isMoz && !isKHTML
    && (sUserAgent.indexOf("Mozilla") == 0)
    && (navigator.appName == "Netscape")
    && (fAppVersion >= 4.0 && fAppVersion < 5.0);

```

```

var isMinNS4 = isMinNS4_5 = isMinNS4_7 = isMinNS4_8 = false;

if (isNS4) {
    isMinNS4 = true;
    isMinNS4_5 = fAppVersion >= 4.5;
    isMinNS4_7 = fAppVersion >= 4.7;
    isMinNS4_8 = fAppVersion >= 4.8;
}

var isWin = (navigator.platform == "Win32") || (navigator.platform == "Windows");
var isMac = (navigator.platform == "Mac68K") || (navigator.platform == "MacPPC")
    || (navigator.platform == "Macintosh");

var isUnix = (navigator.platform == "X11") && !isWin && !isMac;

var isWin95 = isWin98 = isWinNT4 = isWin2K = isWinME = isWinXP = false;
var isMac68K = isMacPPC = false;
251 var isSunOS = isMinSunOS4 = isMinSunOS5 = isMinSunOS5_5 = false;
if (isWin) {
    isWin95 = sUserAgent.indexOf("Win95") > -1
        || sUserAgent.indexOf("Windows 95") > -1;
    isWin98 = sUserAgent.indexOf("Win98") > -1
        || sUserAgent.indexOf("Windows 98") > -1;
    isWinME = sUserAgent.indexOf("Win 9x 4.90") > -1
        || sUserAgent.indexOf("Windows ME") > -1;
    isWin2K = sUserAgent.indexOf("Windows NT 5.0") > -1
        || sUserAgent.indexOf("Windows 2000") > -1;
    isWinXP = sUserAgent.indexOf("Windows NT 5.1") > -1
        || sUserAgent.indexOf("Windows XP") > -1;
    isWinNT4 = sUserAgent.indexOf("WinNT") > -1
        || sUserAgent.indexOf("Windows NT") > -1
        || sUserAgent.indexOf("WinNT4.0") > -1
        || sUserAgent.indexOf("Windows NT 4.0") > -1
        && (!isWinME && !isWin2K && !isWinXP);
}

if (isMac) {
    isMac68K = sUserAgent.indexOf("Mac_68000") > -1
        || sUserAgent.indexOf("68K") > -1;
    isMacPPC = sUserAgent.indexOf("Mac_PowerPC") > -1
        || sUserAgent.indexOf("PPC") > -1;
}

if (isUnix) {
    isSunOS = sUserAgent.indexOf("SunOS") > -1;

    if (isSunOS) {
        var reSunOS = new RegExp("SunOS (\d+\.\d+(?:\.\d+)?)");
        reSunOS.test(sUserAgent);
        isMinSunOS4 = compareVersions(RegExp["$1"], "4.0") >= 0;
        isMinSunOS5 = compareVersions(RegExp["$1"], "5.0") >= 0;
        isMinSunOS5_5 = compareVersions(RegExp["$1"], "5.5") >= 0;
    }
}

```

8.7 例子：登录页面

当创建 Web 应用时，用户看到的第一个页面往往是登录页面。大多数登录页面至少有两个字段：用户名和密码。目的当然是防止未经认证的用户登录。但那些还不能满足应用程序对系统和浏览器的最低要求的用户呢？也不能让他们登录。很多开发人员选用的方法是，在登录页面中加入浏览器检测，在用户还未能输入用户名和密码之前就进行检测。为此，浏览器和操作系统检测脚本就可以大显身手了。

首先，确定 Web 应用的最小需求。例如，假设 Web 应用限制在 Windows 平台上的 IE 5.5 及更高版本和 Unix 上的 Mozilla 1.0 及更高版本，以及 Macintosh 上的 Safari 1.0 或更高版本（这些需求并不太理想，不过可以作为很好的例子）。记住没有说明的一些特定要求：浏览器必须支持 JavaScript，这一点也必须进行检查。

252

因为只对两种浏览器进行编程，默认情况下，当用户使用了不正确的浏览器时，应该出现错误信息。错误信息，在没有 JavaScript 支持以及使用了错误的浏览器或错误的操作系统时，都要显示。下面是一个例子：

```
<html>
    <head>
        <title>Login</title>
        <script type="text/javascript" src="detect.js"></script>
    </head>
    <body>
        <form method="post" action="DoLogin.jsp">
            <div style="border: 2px dashed blue; background-color: #dedede; height: 300px; padding: 10px">
                <div id="divError" style="position: absolute; left: 20px; top: 100px; ">
                    This Web application requires one of the following:
                    <ul>
                        <li>Internet Explorer 5.5 or higher for Windows</li>
                        <li>Mozilla 1.0 or higher for Unix</li>
                        <li>Safari 1.0 or higher for Macintosh</li>
                    </ul>
                </div>
            </div>
        </form>
    </body>
</html>
```

代码的突出显示部分包含实际的错误信息。注意全部的错误信息都包含在叫做 `divError` 的 `<div>` 中，还要注意 `divError` 使用绝对定位。这很重要，因为登录表单直接位于错误信息之上。不过，登录表单在载入时是不可见的，只有在适当的时候才出现。在处理它之前及错误信息之后，给登录表单添加代码：

```
<html>
    <head>
        <title>Login</title>
        <script type="text/javascript" src="detect.js"></script>
```

```

</head>
<body>
    <form method="post" action="DoLogin.jsp">
        <div style="border: 2px dashed blue; background-color: #dedede; height: 300px; padding: 10px">
            <div id="divError" style="position: absolute; left: 20px; top: 100px; ">
                This Web application requires one of the following:
                <ul>
                    <li>Internet Explorer 5.5 or higher for Windows</li>
                    <li>Mozilla 1.0 or higher for Unix</li>
                    <li>Safari 1.0 or higher for Macintosh</li>
                </ul>
            </div>
            <div id="divLogin" style="position: absolute; left: 20px; top: 100px; visibility: hidden">
                <table border="0" width="100%" height="100%"><tr><td align="center">
                    <table border="0">
                        <tr>
                            <td>Username:</td><td><input type="text" name="txtUsername" /></td>
                        </tr>
                        <tr>
                            <td>Password:</td><td><input type="password" name="txtPassword" /></td>
                        </tr>
                        <tr>
                            <td>&ampnbsp</td><td><input type="Submit" value="Login" /></td>
                        </tr>
                    </table>
                </td></tr></table>
            </div>
        </div>
    </form>
</body>
</html>

```

253

现在这一部分添好后，就可以使用检测脚本检查浏览器和操作系统了。如果用户满足要求，代码应该显示登录表单，隐藏错误信息：

```

if ((isMinIE5_5 && isWin) || (isMinMoz1 && isUnix) || (isMinSafari1 && isMac)) {
    document.getElementById("divLogin").style.visibility = "visible";
    document.getElementById("divError").style.visibility = "hidden"
}

```

这个代码片断用 DOM 的样式扩展来设置每个<div/>的 CSS 的可见性 (visibility) 属性。关于如何使用脚本来访问元素的 CSS 样式将会在第 10 章中介绍。

这段代码应该在文档载入后调用，所以应该将其分配给 window.onload 事件处理函数。(不用太担心这部分，事件和事件处理函数将在下一章讨论。)

```

<html>
    <head>
        <title>Login</title>
        <script type="text/javascript" src="detect.js"></script>
        <script type="text/javascript">
            window.onload = function () {
                if ((isMinIE5_5 && isWin) || (isMinMoz1 && isUnix)
                    || (isMinSafari1 && isMac)) {

                    document.getElementById("divLogin").style.visibility = "visible";
                    document.getElementById("divError").style.visibility = "hidden"
                }
            };
        </script>
    </head>
    <body>
        <form method="post" action="DoLogin.jsp">
            <div style="border: 2px dashed blue; background-color: #dedede; height:
300px; padding: 10px">
                <div id="divError" style="position: absolute; left: 20px; top:
100px; ">
                    This Web application requires one of the following:
                    <ul>
                        <li>Internet Explorer 5.5 or higher for Windows</li>
                        <li>Mozilla 1.0 or higher for Unix</li>
                        <li>Safari 1.0 or higher for Macintosh</li>
                    </ul>
                </div>
                <div id="divLogin" style="position: absolute; left: 20px; top:
100px; visibility: hidden">
                    <table border="0" width="100%" height="100%"><tr><td
align="center">
                        <table border="0">
                            <tr>
                                <td>Username:</td><td><input type="text"
name="txtUsername" /></td>
                            </tr>
                            <tr>
                                <td>Password:</td><td><input type="password"
name="txtPassword" /></td>
                            </tr>
                            <tr>
                                <td>&ampnbsp</td><td><input type="Submit" value="Login"
/></td>
                            </tr>
                        </table>
                    </td></tr></table>
                </div>
            </div>
        </form>
    </body>
</html>

```

254

最后一步，万一用户没有 JavaScript 或者禁用了它，也可以包含特别的注意信息。只要使用

<noscript>标签。如果浏览器支持 JavaScript，那么在<noscript>标签中的文本都会被忽略。如果浏览器不支持(或者禁用了它)，那么就会正常显示这段文本。这应该放在 divError 的元素后面：

```

<html>
    <head>
        <title>Login</title>
        <script type="text/javascript" src="detect.js"></script>
        <script type="text/javascript">
            window.onload = function () {
                if ((isMinIE5_5 && isWin) || (isMinMoz1 && isUnix)
                    || (isMinSafari1 && isMac)) {

                    document.getElementById("divLogin").style.visibility = "visible";
                    document.getElementById("divError").style.visibility = "hidden"
                }
            };
        </script>
    </head>
    <body>
        <form method="post" action="DoLogin.jsp">
            <div style="border: 2px dashed blue; background-color: #dedede; height:
300px; padding: 10px">
                <div id="divError" style="position: absolute; left: 20px; top:
100px; ">
                    This Web application requires one of the following:
                    <ul>
                        <li>Internet Explorer 5.5 or higher for Windows</li>
                        <li>Mozilla 1.0 or higher for Unix</li>
                        <li>Safari 1.0 or higher for Macintosh</li>
                    </ul>
                    <noscript>
                        <p>This Web application also requires JavaScript (if you are
using one of the above browsers, make sure that JavaScript is enabled).</p>
                    </noscript>
                </div>
                <div id="divLogin" style="position: absolute; left: 20px; top:
100px; visibility: hidden">
                    <table border="0" width="100%" height="100%"><tr><td
align="center">
                        <table border="0">
                            <tr>
                                <td>Username:</td><td><input type="text"
name="txtUsername" /></td>
                            </tr>
                            <tr>
                                <td>Password:</td><td><input type="password"
name="txtPassword" /></td>
                            </tr>
                            <tr>
                                <td>&nbsp;</td><td><input type="Submit" value="Login"
/></td>
                            </tr>
                        </table>
                    </div>
                </div>
            </div>
        </form>
    </body>

```

```

        </table>
    </td></tr></table>
</div>
</div>
</form>
</body>
</html>

```

256

现在，所有问题都已解决。如果访问页面的用户没有正确的浏览器，就会显示出错误消息（图 8-1），因为脚本不会隐藏错误消息，并显示登录表单。

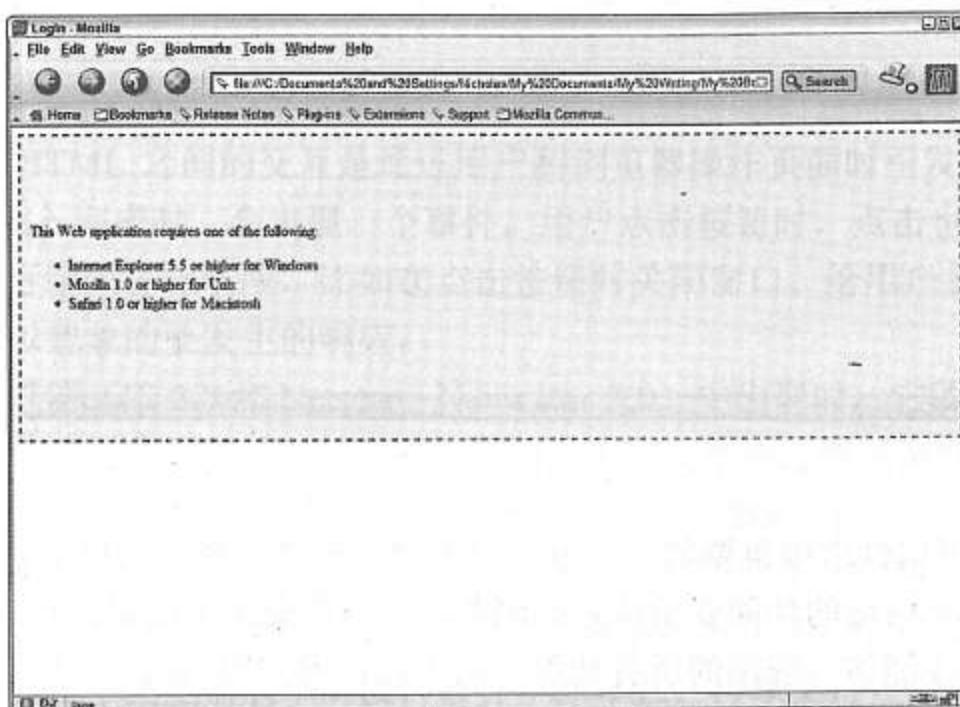


图 8-1

如果浏览器不支持 JavaScript，这段代码就不会运行，就会显示额外的信息（图 8-2）。

257

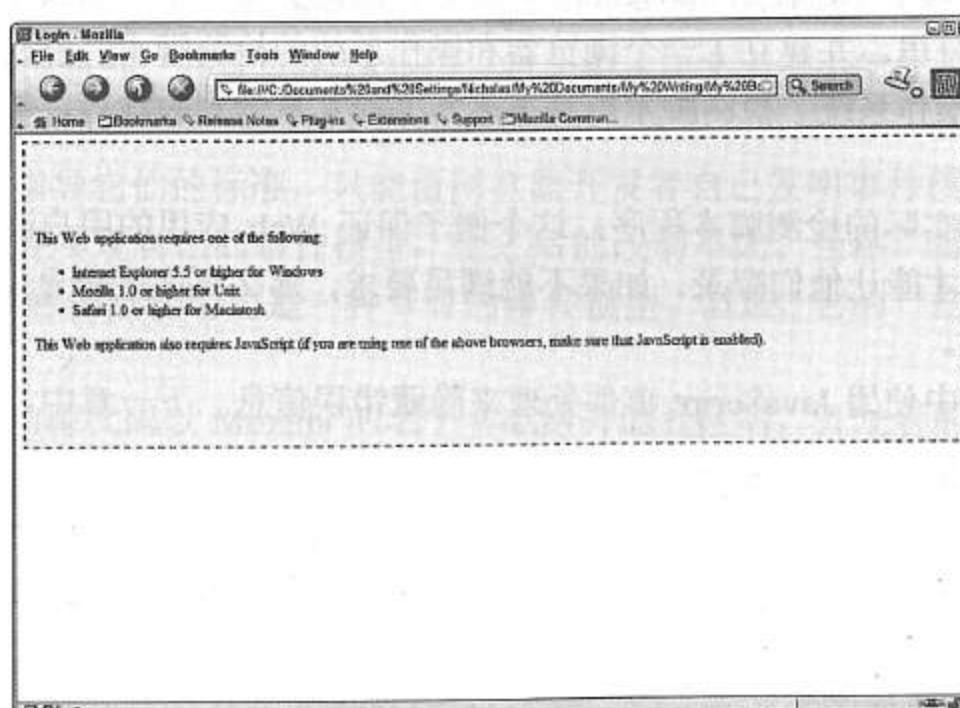


图 8-2

如果使用正确的浏览器和平台，页面载入的时候就执行脚本，隐藏错误信息（这样用户就看不到了）并显示登录表单（图 8-3）。

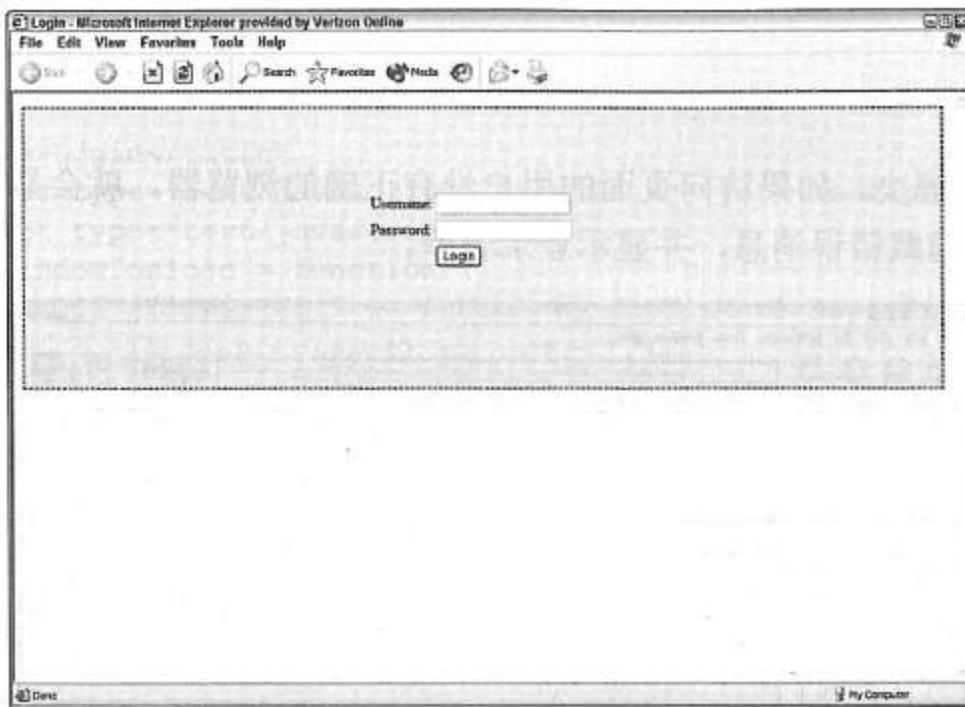


图 8-3

这是 Web 应用可用性中很重要的一部分。很多网站提供很普通的登录表单，任何浏览器都可以使用它，即便网站内部的功能要求特定的浏览器。这个登录页面保证该 Web 应用的用户满足最小的必要需求时才能访问应用的所有功能，而无需再返回服务器。

8.8 小结

在这一章里，学习了如何使用 `navigator` 对象来检测浏览器、平台和操作系统，并且特别地学习了 `user-agent` 字符串。并建立了一个浏览器和操作系统的检测脚本，能精确地检测大部分常见的 Web 浏览器和操作系统。检测脚本用到很多本书前几章学到的技巧，例如字符串处理和正则表达式。

本章最后以一个实际的检测脚本程序：这个例子保证 Web 应用的用户满足最低的浏览器和平台的需求时，页面才能让他们登录。如果不能满足要求，那么页面就会显示错误；否则，页面将正常显示登录表单。

登录表单的例子中使用 JavaScript 事件处理来隐藏错误信息。下一章中，将学习所有关于事件和事件处理的内容。



JavaScript 和 HTML 之间的交互是通过用户和浏览器操作页面时引发的事件（event）来处理的。页面载入完成时，会出现一个事件。用户点击按钮时，点击也是一个事件。开发人员用这些事件来执行编码进行响应，比如在点击按钮时关闭窗口、给用户显示消息、验证日期，以及基本上任何可以想象的要发生的响应。

事件首次在浏览器（IE 3.0 和 Netscape Navigator 3.0）中出现时，它的作用主要是将一些服务器功能转移到客户端完成。那时，访问因特网的常用方法是通过解调器和拨号连接。速度最高只有 56kbit/s，每次返回服务器都可能花费好几分钟的下载时间。

JavaScript 就是为解决这个问题而设计的，通过将一些功能在客户端实现来节省返回服务器的时间。因此，大部分早期的事件都集中在使用表单和表单元素上，可对它们进行简单有效的验证。经过多年发展和多个浏览器版本阶段，事件不断增强，可以支持更多的页面内容。

9.1 今天的事件

正如以前所说，事件是 DOM（文档对象模型）的一部分。不幸的是，也在前面提过，在 DOM Level 1 中未定义任何事件，并且只在 DOM Level 2 中才定义了一小部分子集。完整的事件是在 DOM Level 3 中规定的，该标准在 2004 年最终定案。

由于早前没有指导它们的标准，只能由浏览器开发者自己发明事件模型。IE 首先在 4.0 版（1995 年）中创建并实现自己的事件模型，但之后的改动不大。当然，那时还没有任何 DOM 标准，这也意味着 IE 仍然使用的是一种专有的事件模型。但是，它的一些设计最后被 DOM 吸收到。

261

Netscape 将它的源代码以 Mozilla 的名字贡献给开源社区后，开发者的关键的目标是尽可能地跟随标准。与标准之间存在差距时，Mozilla 小组就会仔细研究标准草案并即时实践。因此，Mozilla 的事件模式和 DOM 标准最为接近。

后来者 Opera 和 Safari 最近成功支持了 DOM 标准的事件模型，最后 IE 成为唯一个对 DOM 事件模型缺乏良好支持的主流浏览器。

尽管在不同的浏览器之间存在不同的 DOM 实现，但一些基本的性质是相同的。

9.2 事件流

IE 4.0 和 Netscape Navigator 4.0 的开发小组都认为仅支持事件是不够的，于是他们各自发明了自己的事件流 (event flow) 形式。事件流意味着在页面上可有不仅一个，甚至多个元素响应同一个事件。点击页面上的按钮时，会发生什么？实际上，是点击了按钮、它的容器及整个页面。从逻辑上说，每一个元素都按照特定的顺序响应那个事件。事件发生顺序（也就是事件流）是 IE 4.0 和 Netscape 4.0 在事件支持上的主要差别。

9.2.1 冒泡型事件

IE 上的解决方案是绰号为冒泡 (dubbed bubbling) 的技术。冒泡型事件的基本思想是，事件按照从最特定的事件目标到最不特定的事件目标 (document 对象) 的顺序触发。例如，如果有下面的页面：

```
<html>
  <head>
    <title>Example</title>
  </head>
  <body onclick="handleClick()">
    <div onclick="handleClick()">Click Me</div>
  </body>
</html>
```

如果用户使用 IE5.5 点击

元素，则事件按以下顺序“冒泡”：

- (1) <div>;
- (2) <body>;
- (3) document.

从逻辑上说，可像图 9-1 中那样形象地思考例子中的冒泡型事件：

262 流程的方法之所以称为冒泡，如图中所示，因为事件按照 DOM 的层次结构像水泡一样不断上升至顶端。

在 IE 6.0 中稍微修改了冒泡型事件，这样<html/>元素也可接收冒泡的事件，则可出现以下代码：

```
<html onclick="handleClick()">
  <head>
    <title>Example</title>
  </head>
  <body onclick="handleClick()">
    <div onclick="handleClick()">Click Me</div>
  </body>
</html>
```

在此例子中，点击页面就冒出一个事件泡上升到<html/>元素，前面的图就要改成图 9-2 中这样：

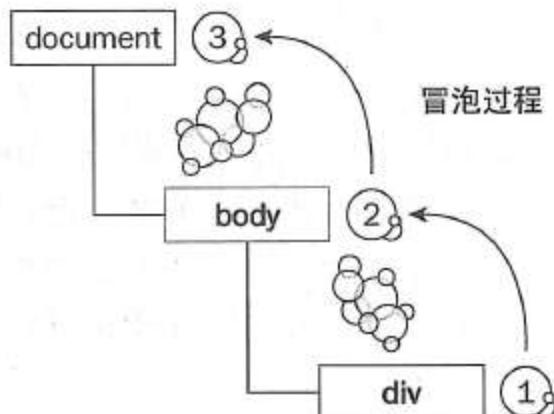


图 9-1

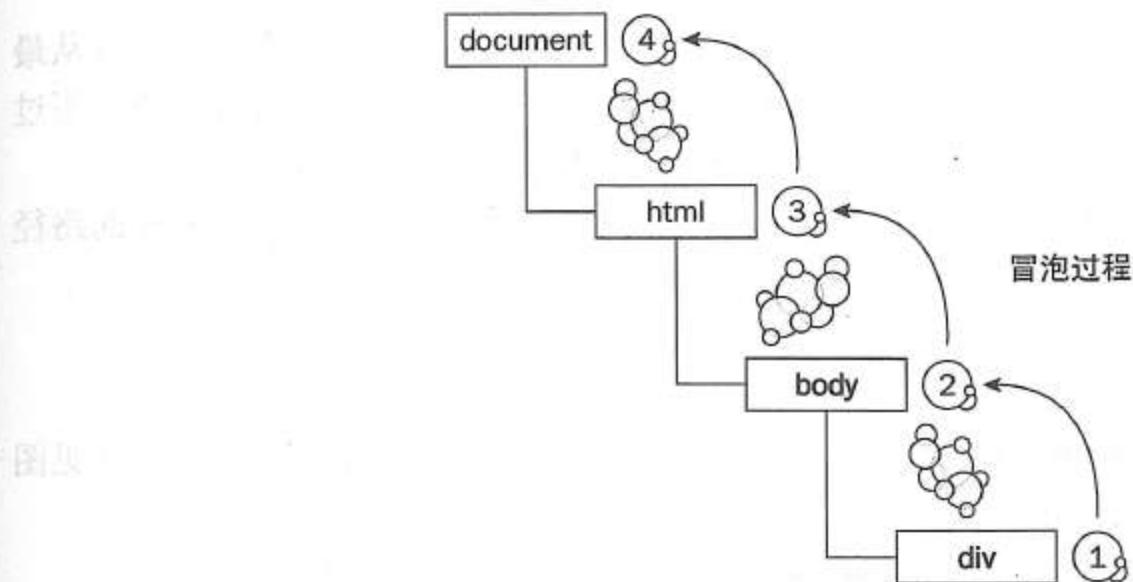


图 9-2

263

如果不确定用户用的是 IE 5.5 还是 IE 6.0，最好避免在<html>的元素级别上处理事件。

Mozilla 1.0 及更高版本的也支持冒泡型事件，但到达了另一层次。类似 IE 6.0，它也支持<html>元素级别的事件。不过，事件“起泡”一直到上升到 windows 窗口对象（这并不是 DOM 标准的一部分）。例如在 Mozilla 中使用前面的例子，点击<div>元素将造成如图 9-3 所示的事件冒泡。

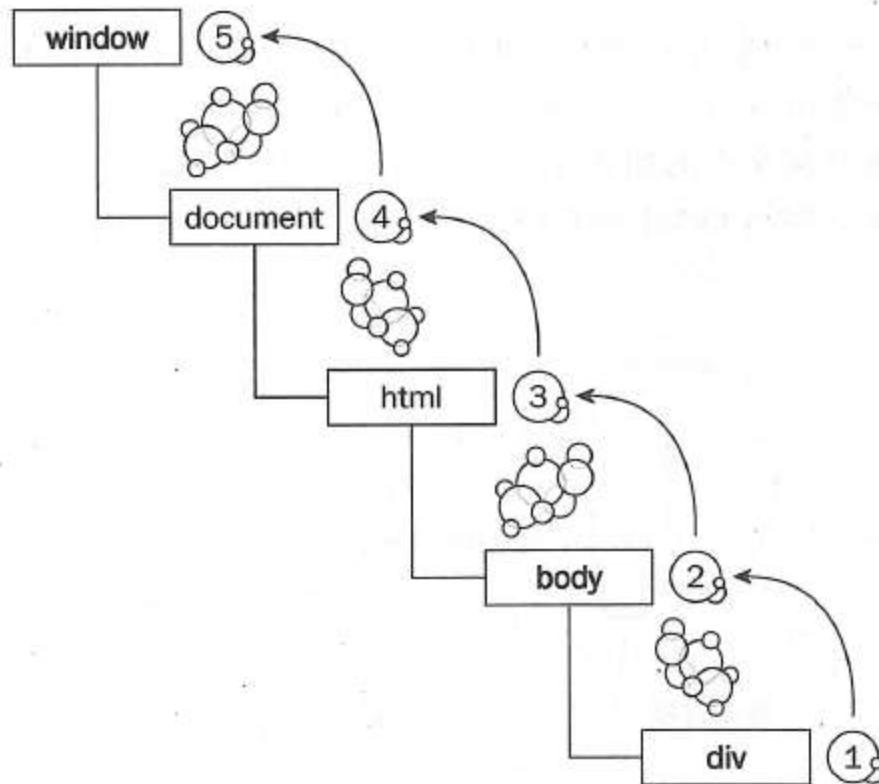


图 9-3

9.2.2 捕获型事件

IE 4.0 使用冒泡型事件，相对地，Netscape Navigator 4.0 使用了另一种称为捕获型事件 (event

capturing) 的解决方案。事件的捕获和冒泡刚好是相反的两种过程——捕获型事件中，事件从最不精确的对象（document 对象）开始触发，然后到最精确（也可以在窗口级别捕获事件，不过必须由开发人员特别指定）。Netscape Navigator 不会将页面上的很多元素暴露给事件。

再次借用前面的例子，如果用户使用 Netscape 4.0 点击<div>元素，事件按照下面的路径传播：

(1) document

264

(2) <div/>

有些人也称之为自顶向下的事件模型，因为它是从 DOM 层次的顶端开始向下延伸的（见图 9-4）。

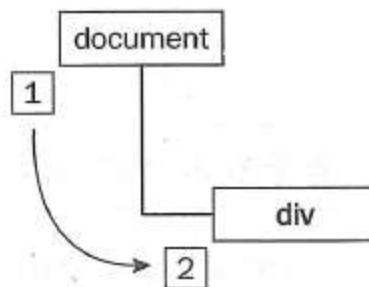


图 9-4

9.2.3 DOM 事件流

DOM 同时支持两种事件模型：捕获型事件和冒泡型事件，但是，捕获型事件先发生。两种事件流会触及 DOM 中的所有对象，从 document 对象开始，也在 document 对象结束（大部分兼容标准的浏览器会继续将事件的捕获/冒泡延续至 window 对象）。

再考虑前面的例子。在与 DOM 兼容的浏览器中点击<div>元素时，事件流的进行如图 9-5 所示。

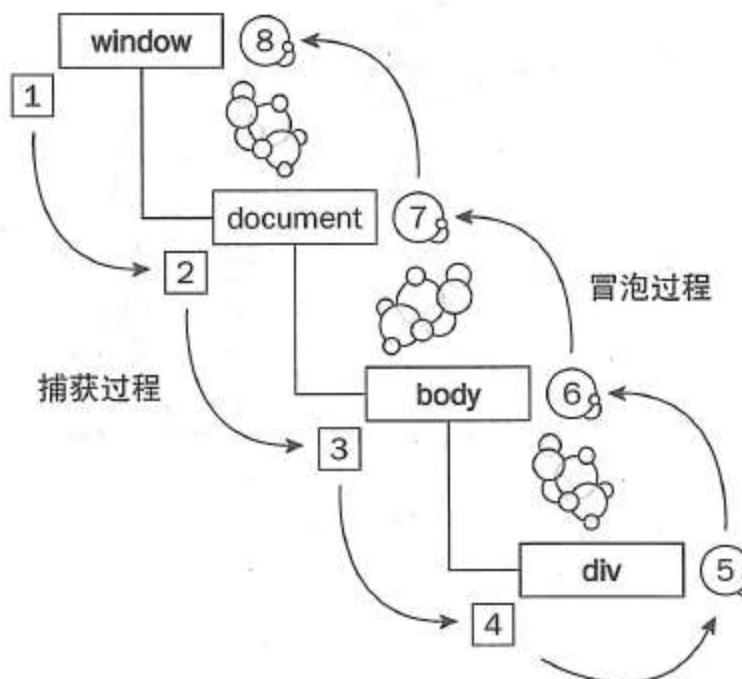


图 9-5

注意因为事件的目标（`<div>`元素）是最精确的元素（于是，在 DOM 树中最深），实际上它会连续接收两次事件，一次在捕获过程中，另一次在冒泡过程中。

DOM 事件模型最独特的性质是，文本节点也触发事件（在 IE 中不会）。所以如果点击例子中的文本 Click Me，实际的事件流应该像图 9-6：

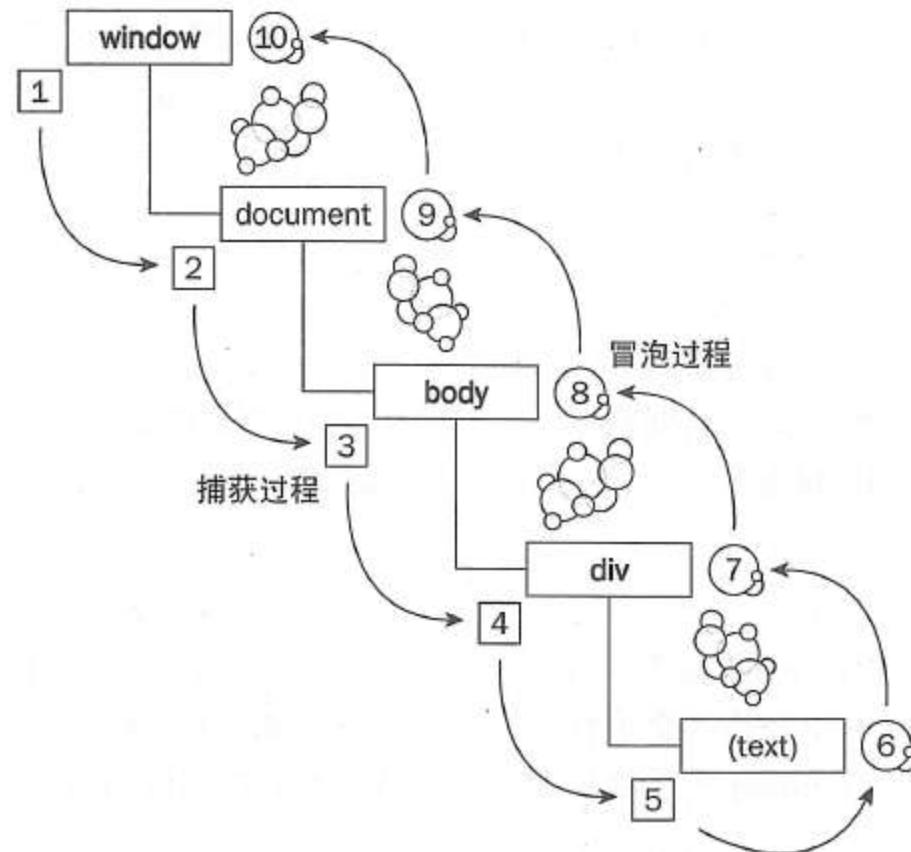


图 9-6

如果要对兼容 DOM 标准的浏览器进行开发，这是非常重要的概念。忘记文本节点也会触发 DOM 中的事件，是使在较新的浏览器上进行开发的人员头疼的头号原因。

9.3 事件处理函数/监听函数

事件是用户或浏览器自身进行的特定行为。这些事件都有自己的名字，如 `click`（点击）、`load`（载入）和 `mouseover`（鼠标经过）。用于响应某个事件而调用的函数称为事件处理函数（event handler），或者 DOM 称之为事件监听函数（event listener）。我们认为响应点击事件的函数是 `onclick`（点击时）事件处理函数。以前，事件处理函数有两种分配方式：在 JavaScript 中或者在 HTML 中。

如果在 JavaScript 中分配事件处理函数，则需要首先获得要处理的对象的引用，然后将函数赋值给对应的事件处理函数属性，像这样：

```
var oDiv = document.getElementById("div1");
oDiv.onclick = function () {
    alert("I was clicked");
};
```

用这个分配方法，事件处理函数名称必须小写，才能正确响应事件。

如果在 HTML 中分配事件处理函数，则只要在 HTML 标签中添加事件处理函数的特性，并在其中包含合适的脚本作为特性值就可以了，如下：

```
<div onclick="alert('I was clicked')"></div>
```

用这种方法，事件处理函数的大小写可任意，所以 `onclick` 等同于 `onClick`、`OnClick` 或 `ONCLICK`（不过，如果使用标准 XHTML 代码，则事件处理函数应该全部用小写定义）。

在 HTML 中分配事件处理函数时，记住特性值中的代码（双引号之间的）是被包装在匿名函数中的，所以 HTML 代码实际上是执行了 JavaScript 代码：

```
oDiv.onclick = function () {
    alert("I was clicked");
};
```

看着很熟悉吧？对，它是与前面 JavaScript 例子中相同的代码。

这两个方法在当前所有流行的浏览器中都可使用，但传统的做法能为每个可用事件分配多个事件处理函数。于是，IE 包含了自己独有的方法，而 DOM 却规定了另一个方法。

9.3.1 IE

在 IE 中，每个元素和 `window` 对象都有两个方法：`attachEvent()` 和 `detachEvent()`。顾名思义，`attachEvent()` 用来给一个事件附加事件处理函数，而 `detachEvent()` 用来将事件处理函数分离出来。每个方法都有两个参数：要分配的事件处理函数的名字（例如：`onclick`）及一个函数。

```
[Object].attachEvent("name_of_event_handler", fnHandler);
[Object].detachEvent("name_of_event_handler", fnHandler);
```

在 `attachEvent()` 中，函数被作为事件处理函数添加，`detachEvent()` 在事件处理函数列表中查找制定的函数，并移除它。例如：

```
var fnClick = function () {
    alert("Clicked!");
};

var oDiv = document.getElementById("div");
oDiv.attachEvent("onclick", fnClick);           //add the event handler
//do some other stuff here
oDiv.detachEvent("onclick", fnClick);           //remove the event handler
```

正如前面所描述的，这种方法可用来添加多个事件处理函数：

```
var fnClick1 = function () {
    alert("Clicked!");
};

var fnClick2 = function () {
    alert("Also clicked! ");
};
```

```
var oDiv = document.getElementById("div");
oDiv.attachEvent("onclick", fnClick1);
oDiv.attachEvent("onclick", fnClick2);
```

这段代码会在点击<div>元素时，显示出两个警告窗口：第一个是“Clicked!”，接着是“Also Clicked!”。事件处理函数总是按照添加它们的顺序进行调用。

也可使用传统的 JavaScript 方法来分配事件处理函数：

```
var fnClick1 = function () {
    alert("Clicked!");
};

var fnClick2 = function () {
    alert("Also clicked!");
};

var oDiv = document.getElementById("div");
oDiv.onclick = fnClick1;
oDiv.attachEvent("onclick", fnClick2);
```

这段代码与前面的例子一样，警告窗口显示的顺序也一样。传统方法上对事件处理函数的赋值其实被看成是另一种 attachEvent() 的调用，所以事件处理函数仍按照定义它们的顺序执行。

9.3.2 DOM

DOM 方法 addEventListener() 和 removeEventListener() 用来分配和移除事件处理函数。与 IE 不同，这些方法需要三个参数：事件名称，要分配的函数和处理函数是用于冒泡阶段还是捕获阶段。如果事件处理函数是用于捕获阶段，第三个参数为 true；用于冒泡阶段，则为 false。下面是语法：

```
[Object].addEventListener("name_of_event", fnHandler, bCapture);
[Object].removeEventListener("name_of_event", fnHandler, bCapture);
```

要使用这些方法，首先获取要处理的对象的引用，然后分配或删除事件处理函数：

```
var fnClick = function () {
    alert("Clicked!");
};

var oDiv = document.getElementById("div1");
oDiv.addEventListener("click", fnClick, false);           //add the event handler
//do some other stuff here
oDiv.removeEventListener("click", fnClick, false);        //remove the event
handler
```

268

与 IE 中一样，可附加多个事件处理函数：

```
var fnClick1 = function () {
    alert("Clicked!");
};

var fnClick2 = function () {
    alert("Also clicked!");
};
```

```
var oDiv = document.getElementById("div1");
oDiv.addEventListener("onclick", fnClick1);
oDiv.addEventListener("onclick", fnClick2);
```

用户点击<div>时，这段代码显示“Clicked! ”，然后是“Also Clicked! ”。与 IE 类似，事件处理函数按照定义它们的顺序执行。

如果使用 `addEventListener()` 将事件处理函数加入到捕获阶段，则必须在 `removeEventListener()` 中指明是捕获阶段，才能正确地将这个事件处理函数删除。例如，不要这样做：

```
var fnClick = function () {
    alert("Clicked!");
};

var oDiv = document.getElementById("div1");

//add the event handler in the bubbling phase
oDiv.addEventListener("click", fnClick, false);

//do some other stuff here

//try to remove the event handler, but the third parameter is true
//instead false...this will fail, though it won't cause an error.
oDiv.removeEventListener("click", fnClick, true);
```

这里，函数 `fnClick()` 被添加到冒泡阶段中，然后尝试在捕获阶段删除它。这不会产生错误，但是函数也不会被删除。

如果使用传统方法直接给事件处理函数属性赋值，事件处理函数将被添加到事件的冒泡阶段。例如，下面两行的效果一样：

```
oDiv.onclick = fnClick;
oDiv.addEventListener("click", fnClick, false);
```

对事件处理函数的直接赋值被认为是另一种对 `addEventListener()` 的调用，所以事件处理

269

函数还是会按照指定它们的顺序进行调用。

直接赋值的一个很重要的不同点是，后续对事件处理函数的赋值清除前面的赋值：

```
oDiv.onclick = fnClick;
oDiv.onclick = fnDifferentClick;
```

在此例子中，起初 `fnClick` 被赋给 `onclick` 事件处理函数，然后又被 `fnDifferentClick` 替换了。

9.4 事件对象

基于不同浏览器开发的开发人员都知道，获取事件信息是很重要的。所以，会创建包含关于刚刚发生的事件的信息的事件对象，包含的信息如下：

- 引起事件的对象；

- 事件发生时鼠标的信息;
- 事件发生时键盘的信息。

事件对象只在发生事件时才被创建，且只有事件处理函数才能访问。所有事件处理函数执行完毕后，事件对象就被销毁。

你也许已经猜到，IE 和 DOM 是用两种不同的方法实现事件对象的。

9.4.1 定位

在 IE 中，事件对象是 window 对象的一个属性 event。也就是说，事件处理函数必须这样访问事件对象：

```
oDiv.onclick = function () {
    var oEvent = window.event;
}
```

尽管它是 window 对象的属性，event 对象还是只能在事件发生时访问。所有的事件处理函数执行完毕后，事件对象就被销毁。

DOM 标准则说，event 对象必须作为唯一的参数传给事件处理函数。所以，在 DOM 兼容的浏览器（如 Mozilla、Safari 和 Opera）中访问事件对象，要这么做：

```
oDiv.onclick = function () {
    var oEvent = arguments[0];
}
```

当然，可直接命名参数，访问就更方便：

```
oDiv.onclick = function (oEvent) {
```

270

9.4.2 属性/方法

1. IE

下面是 IE 中事件的属性和方法(注意：仅应用于 IE 技术和特征的事件的属性和方法未列出来)：

特性/方法	类 型	可读/可写	描 述
altKey	Boolean	R/W	true 表示按下 ALT 键； false 表示没有
button	Integer	R/W	对于特定的鼠标事件，表示按下的鼠标按钮： 0-未按下按钮 1-按下左键 2-按下右键 3-同时按下左右按钮 4-按下中键 5-按下左键和中键 6-按下右键和中键 7-同时按下左中右键

(续)

特性/方法	类 型	可读/可写	描 述
cancelBubble	Boolean	R/W	开发人员将其设为 true 时，将会停止事件向上冒泡
clientX	Integer	R/W	事件发生时，鼠标在客户端区域（不包含工具栏、滚动条等）的 x 坐标
clientY	Integer	R/W	事件发生时，鼠标在客户端区域（不包含工具栏、滚动条等）的 y 坐标
ctrlKey	Boolean	R/W	true 表示按下 CTRL 键，false 表示没有
fromElement	Element	R/W	某些鼠标事件中，鼠标所离开的元素
keyCode	Integer	R/W	对于 keypress 事件，表示按下按钮的 Unicode 字符；对于 keydown/keyup 事件，表示按下按钮的数字代号
offsetX	Integer	R/W	鼠标相对于引起事件的对象的 x 坐标
offsetY	Integer	R/W	鼠标相对于引起事件的对象的 y 坐标
repeat	Boolean	R	如果不断触发 keydown 事件，则为 true，否则为 false
returnValue	Boolean	R/W	开发人员将其设置为 false 以取消事件的默认动作
screenX	Integer	R/W	相对于整个计算机屏幕的鼠标 x 坐标
screenY	Integer	R/W	相对于整个计算机屏幕的鼠标 y 坐标
shiftKey	Boolean	R/W	true 表示按下 Shift 键；否则为 false
srcElement	Element	R/W	引起事件的元素
toElement	Element	R/W	在鼠标事件中，鼠标正在进入的元素
type	String	R/W	事件的名称
x	Integer	R/W	鼠标相对于引起事件的元素的父元素的 x 坐标
y	Integer	R/W	鼠标相对于引起事件的元素的父元素的 y 坐标

271

2. DOM

DOM 事件对象包含了相似的核心属性和方法，但也有很大的不同。下表将逐个列出：

特性/方法	类 型	可读/可写	描 述
altKey	Boolean	R/W	true 表示按下 ALT 键；false 表示没有
bubbles	Boolean	R	表示事件是否正在冒泡阶段中
button	Integer	R/W	对于特定的鼠标事件，表示按下的鼠标按钮： 0-未按下按钮 1-按下左键 2-按下右键 3-同时按下左右按键 4-按下中键 5-按下左键和中键 6-按下右键和中键 7-同时按下左中右键

(续)

特性/方法	类 型	可读/可写	描 述
cancelable	Boolean	R	表示事件能否取消
cancelBubble	Boolean	R	表示事件冒泡是否已被取消
charCode	Integer	R	按下的按键的 Unicode 值
clientX	Integer	R	事件发生时，鼠标在客户端区域（不包含工具栏、滚动条等）的 x 坐标
clientY	Integer	R	事件发生时，鼠标在客户端区域（不包含工具栏、滚动条等）的 y 坐标
ctrlKey	Boolean	R	true 表示按下 CTRL 键，false 表示没有
currentTarget	Element	R	事件目前所指向的元素
detail	Integer	R	鼠标按钮点击的次数
eventPhase	Integer	R	事件的阶段，可能是以下的值中的一个： 0-捕获阶段 1-在目标上 2-冒泡阶段
isChar	Boolean	R	表示按下的按键是否有字符与之相关
keyCode	Integer	R/W	表示按下按键的数字代号
metaKey	Integer	R	表示 META 键是否被按下
pageX	Integer	R	鼠标相对于页面的 x 坐标
pageY	Integer	R	鼠标相对于页面的 y 坐标
preventDefault()	Function	N/A	可以调用这个方法来阻止事件的默认行为
relatedTarget	Element	R	事件的第二目标，经常用于鼠标事件
screenX	Integer	R	相对于整个计算机屏幕的鼠标 x 坐标
screenY	Integer	R	相对于整个计算机屏幕的鼠标 y 坐标
shiftKey	Boolean	R	true 表示按下 Shift 键；否则为 false
stopPropagation()	Function	N/A	可调用这个方法阻止将来事件的冒泡
target	Element	R	引起事件的元素/对象
timestamp	Long	R	事件发生的时间，从 1970 年 1 月 1 日 0:00 起的毫秒数
type	String	R	事件的名称

272

273

9.4.3 相似性

下面是对两种事件对象相似方面的总结。

1. 获取事件类型

这样可在任何一种浏览器中获取事件的类型：

```
var sType = oEvent.type;
```

它返回类似“click”或“mouseover”之类的值，当某个函数同时为两个事件的处理函数时，这很有用。

```
function handleEvent(oEvent) {
    if (oEvent.type == "click") {
        alert("Clicked!");
    } else if (oEvent.type == "mouseover") {
        alert("Mouse Over!");
    }
}

oDiv.onclick = handleEvent;
oDiv.onmouseover = handleEvent;
```

在这段代码中，将函数 handleEvent() 分配给 click 和 mouseover 事件，作为它们的事件处理函数。在函数中，type 属性可用来判断该采取何种行动。

注意这个例子用 DOM 的方法来传递事件对象，但是函数内部的代码也可用于 IE，只要将 event 对象赋值给 oEvent。

2. 获取按键代码 (keydown/keyup事件)

在 keydown 或者 keyup 事件中，可使用 keyCode 属性获取按下的按键的数值代码：

```
var iKeyCode = oEvent.keyCode;
```

keyCode 属性总包含代表按下按键的一个代码，它可能代表一个字符，也可能不是。例如，Enter (回车) 键的 keyCode 为 13，空格键的 keyCode 为 32，回退 (BackSpace) 键的 keyCode 为 8。

3. 检测 Shift、Alt、Ctrl 键

要检测 Shift、Alt、Ctrl 键是否被按下，IE 和 DOM 都可以使用以下代码：

```
var bShift = oEvent.shiftKey;
var bAlt = oEvent.altKey;
var bCtrl = oEvent.ctrlKey;
```

这里面每个属性都包含一个表示按键是否被按下的 Boolean 值（这几个按键都会触发 keydown 事件，然后即可获取它的 keyCode）。

4. 获取客户端坐标

在鼠标事件中，可用 clientX 和 clientY 属性获取鼠标指针在客户端区域的位置：

```
var iClientX = oEvent.clientX;
var iClientY = oEvent.clientY;
```

客户端区域是显示网页的窗口部分（见图 9-7）。这些属性描述鼠标在该区域内的位置离边界有多远（单位是像素）。

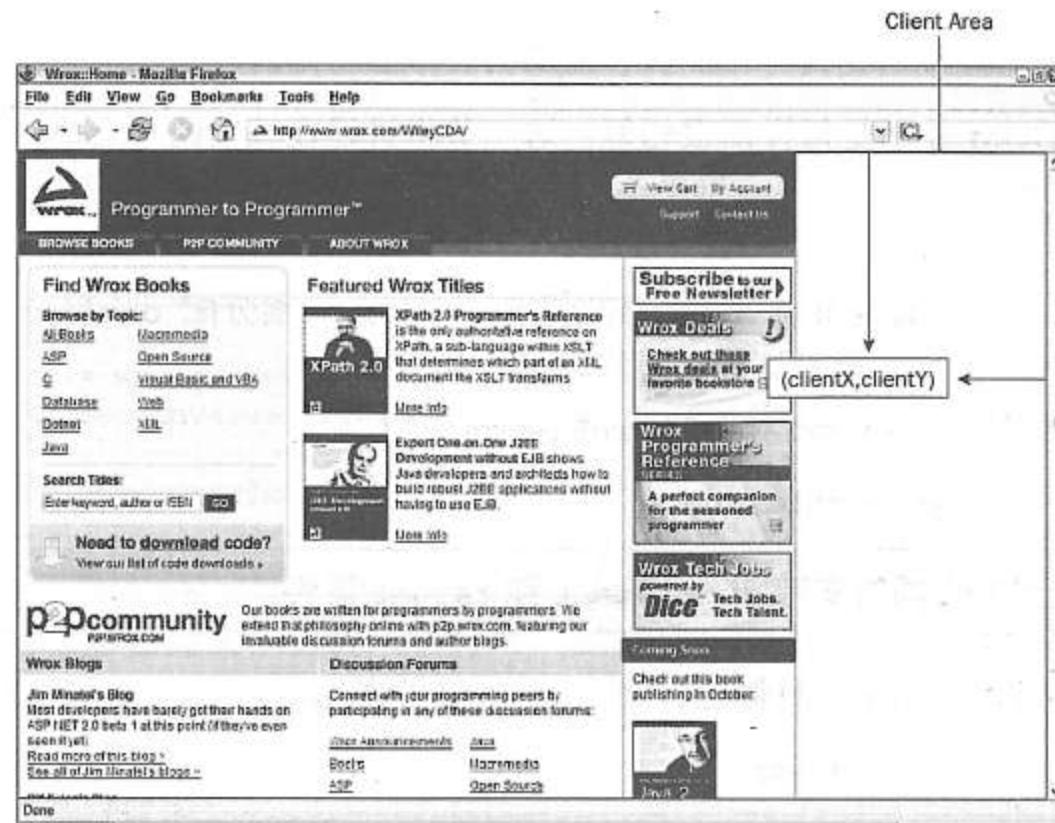


图 9-7

275

5. 获取屏幕坐标

在鼠标事件中，可用 `screenX` 和 `screenY` 属性来获取鼠标指针在计算机屏幕中的位置：

```
var iScreenX = oEvent.screenX;
var iScreenY = oEvent.screenY;
```

这两个属性都返回表示离用户屏幕的边界有多少个像素的整数（见图 9-8）。

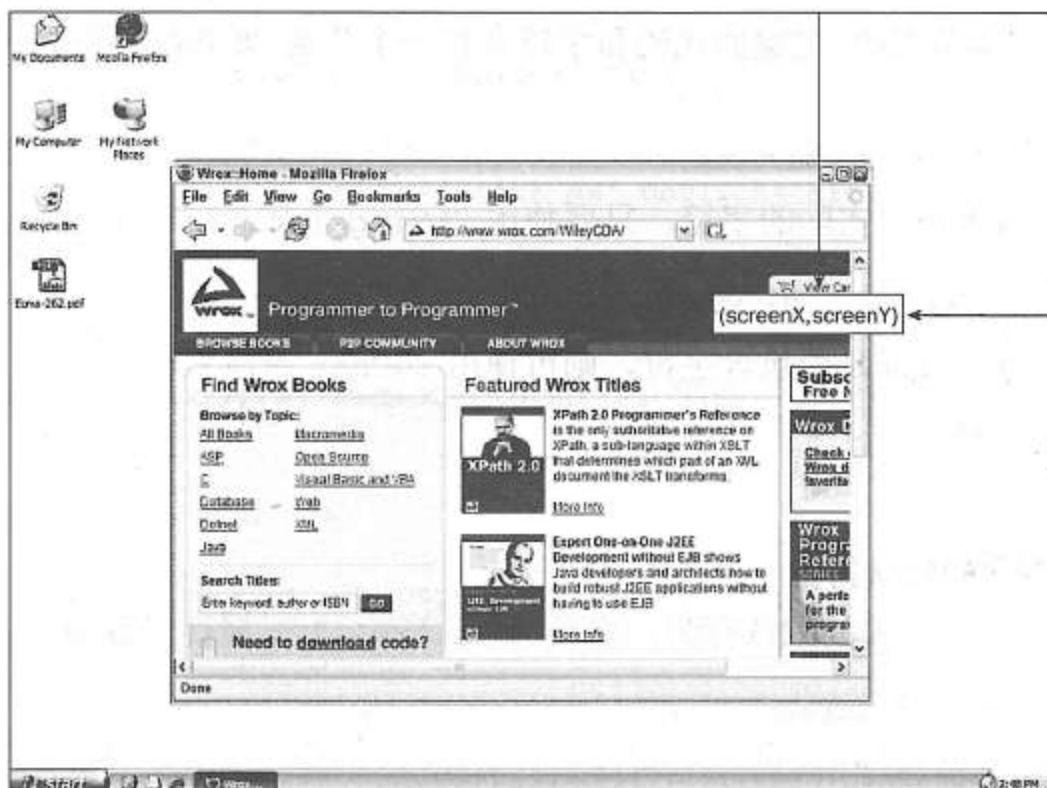


图 9-8

9.4.4 区别

当然, IE 和 DOM 并不是每样都很相似。这一节将讲述在编写跨平台脚本时必须注意的一些区别。

276

1. 获取目标

位于事件中心的对象称为目标 (target)。假设为<div/>元素分配 onclick 事件处理函数。触发 click 事件时, <div/>就被认为是目标。

在 IE 中, 目标包含在 event 对象的 srcElement 属性中:

```
var oTarget = oEvent.srcElement;
```

Macintosh 上的 IE 同时支持 srcElement 和 target 属性。

在 DOM 兼容的浏览器中, 目标包含在 target 属性中:

```
var oTarget = oEvent.target;
```

IE 目标只能是元素、文档或者窗口; DOM 兼容的浏览器也允许把文本节点作为目标。

2. 获取字符代码

你看到了 IE 和 DOM 都支持 event 对象的 keyCode 属性, 它会返回按下按键的数值代码。如果按键代表一个字符 (非 Shift、Ctrl、Alt 等), IE 的 keyCode 将返回字符的代码 (等于它的 Unicode 值):

```
var iCharCode = oEvent.keyCode;
```

在 DOM 兼容的浏览器中, 按键的代码和字符会有一个分离。要获取字符代码, 使用 charCode 属性:

```
var iCharCode = oEvent.charCode;
```

然后可用这个值来获取实际的字符, 只要使用 String.fromCharCode() 方法:

```
var sChar = String.fromCharCode(oEvent.charCode);
```

如果不确定按下的按键是否包含字符, 则可使用 isChar 属性来进行判断:

```
if (oEvent.isChar) {
    var iCharCode = oEvent.charCodeAt();
}
```

277

3. 阻止某个事件的默认行为

在 IE 中要阻止某个事件的默认行为, 必须将 returnValue 属性设置为 false:

```
oEvent.returnValue = false;
```

而在 Mozilla 中, 只要调用 preventDefault() 方法:

```
oEvent.preventDefault();
```

你也许会想，“什么时候才需要阻止事件的默认行为呢？”其实，某些情况下阻止事件的默认行为是十分有用的。

第一，当用户右键点击页面时，阻止他使用上下文菜单。你只要阻止 contextmenu 事件的默认行为就可以了，这样做：

```
document.body.oncontextmenu = function (oEvent) {
    if (isIE) {
        oEvent = window.event;
        oEvent.returnValue = false;
    } else {
        oEvent.preventDefault();
    }
};
```

第二，你可能还想在为文本框键入字符时，阻止它的默认行为，以禁止特定字符的输入或者抢先阻止鼠标动作，除非满足特定的条件。这是非常强大的功能，我将在本书后面详细介绍。

4. 停止事件复制（冒泡）

在 IE 中，要阻止事件进一步冒泡，必须设置 cancelBubble 属性为 true：

```
oEvent.cancelBubble = true;
```

在 mozilla 中，只需调用 stopPropagation() 方法：

```
oEvent.stopPropagation();
```

停止事件复制可以阻止事件流中的其他对象的事件处理函数的执行。考虑这个例子：

```
<html onclick="alert('html')">
    <head>
        <title>Event Propagation Example</title>
    </head>
    <body onclick="alert('body')">
        <input type="button" value="Click Me" onclick="alert('input')"/>
    </body>
</html>
```

278

点击页面上的按钮，会顺次出现三个警告框：“input”、“body”和“html”。这是因为事件先从<input/>元素冒泡到<body/>，然后又到<html/>。然而，如果在按钮处停止事件复制，情况就变了：

```
<html onclick="alert('html')">
    <head>
        <title>Stopping Event Propagation Example</title>
        <script type="text/javascript" src="detect.js"></script>
        <script type="text/javascript">
            function handleClick(oEvent) {
                alert("input");
                if (isIE) {
                    oEvent.cancelBubble = true;
                } else {
```

```

        oEvent.stopPropagation();
    }
}
</script>
</head>
<body onclick="alert('body')">
    <input type="button" value="Click Me" onclick="handleClick(event)" />
</body>
</html>

```

执行此例子后，点击按钮，你将只能看到“input”警告框，其他的都没有，因为事件的复制被停止。为能正确实现，还需使用前一章的浏览器检测代码。

你可能还注意到，`<input/>`元素将 `event` 对象作为参数传送给 `handleClick()` 函数。这在任何浏览器中都是可行的，因为一旦发生事件，就会创建 `event` 对象，且这时是一个全局变量。

9.5 事件的类型

根据触发事件的事物和事件发生的对象，可将浏览器中发生的事件分成几个类型。DOM 标准定义了以下几组事件：

- 鼠标事件：用户使用鼠标进行特定操作时触发；
- 键盘事件：用户在键盘上敲击、输入时触发；
- HTML 事件：窗口发生变动或者发生特定的客户端-服务器端交互时触发；
- 突变事件：底层的 DOM 结构发生改变时触发。

279

9.5.1 鼠标事件

鼠标事件是 Web 上最常用的事件类型。其中包含以下事件：

- `click`——用户点击鼠标左键时发生（如果右键也按下则不会发生）。当用户的焦点在按钮上，并按了回车键，同样会触发这个事件。
- `dblclick`——用户双击鼠标左键时发生（如果右键也按下则不会发生）。
- `mousedown`——用户点击任意一个鼠标按钮时发生。
- `mouseout`——鼠标指针在某个元素上，且用户正要将其移出元素的边界时发生。
- `mouseover`——鼠标移出某个元素，到另一个元素上时发生。
- `mouseup`——用户松开任意一个按钮时发生。
- `mousemove`——鼠标在某个元素上时持续发生。

页面上的所有元素都支持鼠标事件。下面的例子简单地描述了所有的鼠标事件：

```

<html>
    <head>
        <title>Mouse Events Example</title>
        <script type="text/javascript">
            function handleEvent(oEvent) {
                var oTextbox = document.getElementById("txt1");
                oTextbox.value += "\n" + oEvent.type;
            }
        </script>
    </head>
    <body>
        <input type="text" id="txt1" value="Initial Value" />
        <input type="button" value="Click Me" onclick="handleEvent(event)" />
    </body>
</html>

```

```

        }
    </script>
</head>
<body>
    <p>Use your mouse to click and double click the red square.</p>
    <div style="width: 100px; height: 100px; background-color: red"
        onmouseover="handleEvent(event)"
        onmouseout="handleEvent(event)"
        onmousedown="handleEvent(event)"
        onmouseup="handleEvent(event)"
        onclick="handleEvent(event)"
        ondblclick="handleEvent(event)" id="div1"></div>
    <p><textarea id="txt1" rows="15" cols="50"></textarea></p>
</body>
</html>

```

- 这段代码将在页面上显示一个红色方框和一个文本框。在使用鼠标与红色方框交互时，文本框会显示触发的事件。注意对所有的事件都只用一个函数作为事件处理函数。它仅向文本框输出事件的类型。

280

使用 `mouseover` 和 `mouseout` 事件是改变页面上某个东西的外观的一个很流行的方法，比如某个图片。这是个非常简单的技巧，但经常用到：

```

```

这段代码段中，`onmouseover` 和 `onmouseout` 事件处理函数仅用了一行代码。注意为什么可用 `this` 对象可以改变图像的 `src` 属性？这是事件处理函数一个隐藏的真相：事件处理函数会被认为是分配给对象的方法。于是，即可用 `this` 对象来访问事件对象（在此例子中，它十分有用，因为它避免了进行浏览器检测）。`mouseover` 事件触发后，图像的 `src` 属性就被设置为 `image2.gif`，它看起来与 `image1.gif` 不同；`mouseout` 事件触发时，`src` 属性就设回 `image1.gif`。

1. 事件的属性

每个鼠标事件都会给以下 `event` 对象的属性填入值：

- 坐标属性（例如 `clientX` 和 `clientY` 等）；
- `type` 属性；
- `target` (DOM) 或 `srcElement` (IE) 属性；
- `shiftKey`、`ctrlKey`、`altKey` 和 `metaKey` (DOM) 属性；
- `button` 属性（只在 `mousedown`、`mouseover`、`mouseout`、`mousemove` 和 `mouseup` 事件中）。

这些属性都给出刚刚发生的鼠标事件的一些信息。你应该已经熟悉 `type` 属性了，但如果能将其他属性也加入到事件处理中去，就可以描绘出对刚刚发生的事件的更加完整的图像：

```
<html>
    <head>
        <title>Mouse Events Example</title>
        <script type="text/javascript">
```

```

        function handleEvent(oEvent) {
            var oTextbox = document.getElementById("txt1");
            oTextbox.value += "\n> " + oEvent.type;
            oTextbox.value += "\n      target is " + (oEvent.target ||
oEvent.srcElement).id;
            oTextbox.value += "\n      at (" + oEvent.clientX + "," +
oEvent.clientY + ") in the client";
            oTextbox.value += "\n      at (" + oEvent.screenX + "," +
oEvent.screenY + ") on the screen";
            oTextbox.value += "\n      button down is " + oEvent.button;

            var arrKeys = [];
            if (oEvent.shiftKey) {
                arrKeys.push("Shift");
            }
            if (oEvent.ctrlKey) {
                arrKeys.push("Ctrl");
            }

            if (oEvent.altKey) {
                arrKeys.push("Alt");
            }

            oTextbox.value += "\n      keys down are " + arrKeys;
        }
    </script>
</head>
<body>
    <p>Use your mouse to click and double click the red square.</p>
    <div style="width: 100px; height: 100px; background-color: red"
        onmouseover="handleEvent(event)"
        onmouseout="handleEvent(event)"
        onmousedown="handleEvent(event)"
        onmouseup="handleEvent(event)"
        onclick="handleEvent(event)"
        ondblclick="handleEvent(event)" id="div1"></div>
    <p><textarea id="txt1" rows="15" cols="50"></textarea></p>
</body>
</html>

```

此例子是前一个的更新，文本框中可显示更多信息。这里，除事件类型外，还输出前面提到的属性。代码中需要注意的一行是(oEvent.target || oEvent.srcElement).id。记住，逻辑或操作符作用于两个对象时，第一个对象非空，返回第一个对象，否则返回第二个对象。这样，就可利用它来判断哪个属性保存了事件目标，以便获取 id 特性。

Opera 7.5 在检测不同的按键（Shift、Ctrl 和 Alt）时有 Bug。它错误地将 Shift 键报告为 Ctrl 键。另外，它根本不会检测 Alt 键。如果打算支持 Opera，则要对这些属性极为小心。

对于 mouseover 和 mouseout 事件，事件对象还有额外的属性。在 IE 中，fromElement 属性

性包含鼠标指针来自的元素，同时 toElement 包含鼠标指针去往的元素。对于 mouseover 事件，toElement 总是等于 srcElement，而对于 mouseout 事件，fromElement 总是等于 srcElement。你可以自己尝试一下：

```

<html>
  <head>
    <title>IE Mouse Events Example</title>
    <script type="text/javascript">
      function handleEvent(oEvent) {
        var oTextbox = document.getElementById("txt1");
        oTextbox.value += "\n>" + oEvent.type;
        oTextbox.value += "\n    target is " + oEvent.srcElement.tagName;
        if (oEvent.fromElement) {
          oTextbox.value += "\n    fromElement is " +
            oEvent.fromElement.tagName;
        }
        if (oEvent.toElement) {
          oTextbox.value += "\n    toElement is " +
            oEvent.toElement.tagName;
        }
      }
    </script>
  </head>
  <body>
    <p>Use your mouse to click and double click the red square.</p>
    <div style="width: 100px; height: 100px; background-color: red"
      onmouseover="handleEvent(event)"
      onmouseout="handleEvent(event)"
      onmousedown="handleEvent(event)"
      onmouseup="handleEvent(event)"
      onclick="handleEvent(event)"
      ondblclick="handleEvent(event)" id="div1"></div>
    <p><textarea id="txt1" rows="15" cols="50"></textarea></p>
  </body>
</html>
```

282

因为这种有冗余，所以 DOM 对 mouseover 和 mouseout 只支持一个 event 属性 relatedTarget。在 mouseover 事件中，relatedTarget 指出鼠标指针来自何处。在 mouseout 事件中，relatedTarget 指出鼠标指针去往何方。可修改前面的例子来进行测试：

```

<html>
  <head>
    <title>DOM Mouse Events Example</title>
    <script type="text/javascript">
      function handleEvent(oEvent) {
        var oTextbox = document.getElementById("txt1");
        oTextbox.value += "\n>" + oEvent.type;
        oTextbox.value += "\n    target is " + oEvent.target.tagName;
        oTextbox.value += "\n    relatedTarget is " +
          oEvent.relatedTarget.tagName;
      }
    </script>
```

```

</head>
<body>
    <p>Use your mouse to click and double click the red square.</p>
    <div style="width: 100px; height: 100px; background-color: red"
        onmouseover="handleEvent(event)"
        onmouseout="handleEvent(event)"
        onmousedown="handleEvent(event)"
        onmouseup="handleEvent(event)"
        onclick="handleEvent(event)"
        ondblclick="handleEvent(event)" id="div1"></div>
    <p><textarea id="txt1" rows="15" cols="50"></textarea></p>
</body>
</html>

```

283

2. 顺序

`click` 事件触发前，会先发生 `mousedown` 事件，然后发生 `mouseup` 事件。类似的，要触发 `dblclick` 事件，在同一个目标上要按顺序发生以下事件：

- (1) `mousedown`;
- (2) `mouseup`;
- (3) `click`;
- (4) `mousedown`;
- (5) `mouseup`;
- (6) `click`;
- (7) `dblclick`。

移动鼠标从一个对象进入另一个对象时，先发生的事件是 `mouseout`（发生在鼠标移出的那个对象）。接着，在这两个对象上都触发 `mousemove` 事件。最后，在鼠标进入的那个对象上触发 `mouseover` 事件。

9.5.2 键盘事件

键盘事件由用户对键盘的动作引发。有以下键盘事件：

- `keydown`——用户在键盘上按下某按键时发生。一直按着某按键，它则会不断触发（Opera 浏览器除外）。
- `keypress`——用户按下一个按键，并产生一个字符时发生（也就是不管类似 Shift、Alt 之类的键）。一直按下按键时，它则会持续发生。
- `keyup`——用户释放按着的按键时发生。

尽管所有的元素都支持键盘事件但在文本框中输入文字时，可最容易看见这些事件。

```

<html>
    <head>
        <title>Key Events Example</title>
        <script type="text/javascript">
            function handleEvent(oEvent) {
                var oTextbox = document.getElementById("txt1");
                oTextbox.value += "\n>" + oEvent.type;
            }
        </script>
    </head>
    <body>
        <input type="text" id="txt1" value="Type something...">
        <span>Press a key to see what happens!</span>
    </body>
</html>

```

```

</script>
</head>
<body>
    <p>Type some characters into the first textbox.</p>
    <p><textarea id="txtInput" rows="15" cols="50">
        onkeydown="handleEvent(event)"
        onkeyup="handleEvent(event)"
        onkeypress="handleEvent(event)"></textarea></p>
    <p><textarea id="txt1" rows="15" cols="50"></textarea></p>
</body>
</html>

```

284

1. 事件的属性

对每个键盘事件，会填入以下的事件属性：

- keyCode 属性；
- charCode 属性（仅 DOM）；
- target(DOM)或者 srcElement (IE) 属性；
- shiftKey、ctrlKey、altKey 和 metaKey (DOM) 属性。

注意按下 Shift、Ctrl、Alt 或者 Meta 键时，除设置对应的属性为 true 外，还都会引发 keydown 事件。以下的例子测试了这些属性：

```

<html>
    <head>
        <title>Key Events Example</title>
        <script type="text/javascript">
            function handleEvent(oEvent) {
                var oTextbox = document.getElementById("txt1");
                oTextbox.value += "\n" + oEvent.type;
                oTextbox.value += "\n      target is " + (oEvent.target ||
oEvent.srcElement).id;
                oTextbox.value += "\n      keyCode is " + oEvent.keyCode;
                oTextbox.value += "\n      charCode is " + oEvent.charCode;

                var arrKeys = [];
                if (oEvent.shiftKey) {
                    arrKeys.push("Shift");
                }

                if (oEvent.ctrlKey) {
                    arrKeys.push("Ctrl");
                }

                if (oEvent.altKey) {
                    arrKeys.push("Alt");
                }

                oTextbox.value += "\n      keys down are " + arrKeys;
            }
        </script>
    </head>

```

```

<body>
  <p>Type some characters into the first textbox.</p>
  <p><textarea id="txtInput" rows="15" cols="50"
    onkeydown="handleEvent(event)"
    onkeyup="handleEvent(event)"
    onkeypress="handleEvent(event)"></textarea></p>
  <p><textarea id="txt1" rows="15" cols="50"></textarea></p>
</body>
</html>

```

285

2. 顺序

用户按一次某字符按键时，会按以下顺序发生事件：

- (1) keydown;
- (1) keypress;
- (2) keyup。

如果用户按一次某非字符按键（例如 Shift），会按以下顺序发生事件：

- (1) keydown;
- (2) keyup。

如果用户按下一个字符按键且不放，keydown 和 keypress 事件将逐个持续触发，直到松开按键；如果用户按下非字符按键且不放，将只有 keydown 事件持续触发。可使用前面的例子进行测试。

9.5.3 HTML 事件

HTML 事件分类是由原来从 IE 4.0 和 Netscape 4.0 的开发人员创建的事件模型中遗留下来了很多事件组成的。

- load 事件——页面完全载入后，在 window 对象上触发；所有的框架都载入后，在框架集上触发；完全载入后，在其上触发；或者对于<object/>元素，当其完全载入后在其上触发。
- unload 事件——页面完全卸载后，在 window 对象上触发；所有的框架都卸载后，在框架集上触发；完全卸载后，在其上触发；或者对于<object/>元素，其完全卸载，在其上触发。
- abort 事件——用户停止下载过程时，如果<object/>对象还未完全载入，就在其上触发。
- error 事件——JavaScript 脚本出错时，在 window 对象上触发；某个的指定图像无法载入时，在其上触发；或<object/>元素无法载入时触发；或者框架集中的一或多个框架无法载入时触发。这个事件将在第 14 章详细讨论。
- select 事件——用户选择了文本框中的一个或多个字符时触发（<input/>或者<textarea/>）。这个事件将在第 11 章进一步讨论。
- change 事件——文本框（<input/>或者<textarea/>）失去焦点时并且在它获取焦点后内容发生过改变时触发；某个<select/>元素的值发生改变时触发。这个事件将在第 11

章进一步讨论。

- submit 事件——点击提交按钮 (`<input type="submit"/>`) 时，在`<form/>`上触发。这个事件将在第 11 章进一步讨论。
- reset 事件——点击重置按钮 (`<input type="reset"/>`) 时，在`<form/>`上触发。这个事件将在第 11 章进一步讨论。
- resize 事件——窗口或者框架的大小发生改变时触发。
- scroll 事件——用户在任何带滚动条的元素上卷动它时触发。`<body/>`元素包含载入页面的滚动条。
- focus 事件——任何元素或者窗口本身获取焦点时（用户点击它、Tab 键切换到它或者任何其他与它交互的手段）触发。
- blur 事件——任何元素或者窗口本身失去焦点时触发。

286

1. load 和 unload 事件

载入 `load` 事件也许是最常用的事件，因为在全部的页面载入完之前，任何 DOM 操作都不能发生。对于 `window` 对象可用两种方法定义 `onload` 事件处理函数。首先，可使用 JavaScript 直接将其分配给 `window` 对象：

```
<html>
  <head>
    <title>Onload Example</title>
    <script type="text/javascript">
      window.onload = function () {
        alert("Loaded");
      };
    </script>
  <body>
  </body>
</html>
```

第二种方法是在 HTML 中的`<body/>`元素上分配：

```
<html>
  <head>
    <title>Onload Example</title>
    <body onload="alert('Loaded')">
    </body>
</html>
```

糊涂了？这里的问题是，`load` 事件实际发生在窗口上，这也是用 JavaScript 在 `window` 对象上定义事件处理函数的原因。但在 HTML 中，没有任何代码可表示 `window` 对象，所以 HTML 权威们就决定处理函数要在`<body/>`元素上进行分配，然后再在后台放入 `window` 对象。所以，如果在`<body/>`元素上设置了 `onload` 事件处理函数，然后检查 `window.onload` 属性（如下例所示），你就会发现事件处理函数的代码是一样的：

```
<html>
  <head>
    <title>Onload Example</title>
```

```

<script type="text/javascript">
    function handleLoad() {
        alert(window.onload);
    }
</script>
</head>
<body onload="handleLoad()">
</body>
</html>

```

287

那么还能否给 `document.body.onload` 分配事件处理函数呢？当然可以。问题是，在页面还未载入完`<body>`标签前，`document.body` 是不存在的。这意味着，如果尝试在`<head>`元素中分配事件处理函数，按理应该是可完成的，但却出现了错误。自己尝试一下：

```

<html>
    <head>
        <title>Onload Example</title>
        <script type="text/javascript">
            document.body.onload = function () {
                alert("loaded");
            }
        </script>
        <body>
        </body>
    </html>

```

如果运行这段代码，你会看到一个错误，显示 `document.body` 未定义。所以，最好总是给 `window` 对象分配 `onload` 事件处理函数。

`unload` 事件也可用同样方式进行处理，直接给 `window` 对象分配事件处理函数或者在`<body>` 元素中进行分配。卸载事件在你从一个页面浏览到了另外一个页面时（通过点击一个链接或者使用前进/后退按钮）或者关闭浏览器窗口时触发：

```

<html>
    <head>
        <title>OnUnload Example</title>
    </head>
    <body onunload="alert('Goodbye')">
    </body>
</html>

```

在窗口关闭或者下一个页面获取控制之前，只有很短的时间来执行事件处理函数的代码，所以最好避免使用 `onunload` 事件处理函数。使用 `onunload` 的最佳方式是取消该页面上的对象引用；任何比这更复杂的功能都应该避免。

2. `resize` 事件

网页常常需要根据浏览器窗口的大小进行调整。这时，可用 `resize` 事件来判断何时动态地改变某些元素。

与 `load` 和 `unload` 事件类似，`resize` 事件的处理函数必须使用 JavaScript 代码分配给 `window` 对象或者在 HTML 中分配给`<body>` 元素。

```
<html>
  <head>
    <title>OnResize Example</title>
  </head>
  <body onresize="alert('Resizing')">
  </body>
</html>
```

根据使用的浏览器，实际的 `resize` 事件会发生在不同时刻。在 IE 和 Opera 中，一旦浏览器的大小发生改变，就触发 `resize` 事件。只要窗口的边框移动一个像素，事件就触发。而在 Mozilla 中，`resize` 事件只在停止对窗口大小的改变后，才会触发。你可在不同的浏览器中试验前面的例子。

最大化或者最小化窗口时，也会触发 `resize` 事件。

3. scroll 事件

你可能会希望在用户卷动窗口（或其他元素）时，跟踪变化来确保某些内容一直在屏幕上可见。通过使用 `scroll` 事件即可实现：

```
<html>
  <head>
    <title>OnScroll Example</title>
  </head>
  <body onscroll="alert('Scrolling')">
    <p>Try scrolling this window.</p>
    <p>&nbsp;</p>
    <p>&nbsp;</p>
  </body>
</html>
```

也可以把事件处理函数分配给 `window.onscroll` 属性：

```
<html>
  <head>
    <title>OnScroll Example</title>
    <script type="text/javascript">
      window.onscroll = function () {
        alert("scrolling");
      }
    </script>
  </head>
  <body>
    <p>Try scrolling this window.</p>
```

289

```

<p>&nbsp;</p>
</body>
</html>

```

也可和<body>元素的一些属性协调使用这个事件，如 scrollLeft，它保存窗口在水平方向上卷动的距离，以及 scrollTop，它保存窗口在垂直方向上卷动的距离：

```

<html>
    <head>
        <title>OnScroll Example</title>
        <script type="text/javascript">
            window.onscroll = function () {
                var oTextbox = document.getElementById("txt1");
                oTextbox.value += "\nscroll is at " + document.body.scrollLeft +
horizontally and " + document.body.scrollTop + " vertically.";
            }
        </script>
    </head>
    <body>
        <p>Try scrolling this window.</p>
        <p><textarea rows="15" cols="50" id="txt1"></textarea>
        <p>&nbsp;</p>
        </body>
    </html>

```

在此例子中，用一个文本框来跟踪 scrollLeft 和 scrollTop 属性，这样能精确地看到它们的变化。这可在主流的浏览器上运行，并且可用来创建一些很酷的特效，例如始终在页面的顶部显示的水印：

```

<html>
    <head>
        <title>OnScroll Example</title>

```

290

```

<script type="text/javascript">
    window.onscroll = function () {
        var oWatermark = document.getElementById("divWatermark");
        oWatermark.style.top = document.body.scrollTop;
    }
</script>
</head>
<body>
    <p>Try scrolling this window.</p>
    <div id="divWatermark" style="position: absolute; top: 0px; right: 0px;
color: #cccccc; width: 150px; height: 30px; background-color: navy">Watermark</div>
    <p>Line 1</p>
    <p>Line 2</p>
    <p>Line 3</p>
    <p>Line 4</p>
    <p>Line 5</p>
    <p>Line 6</p>
    <p>Line 7</p>
    <p>Line 8</p>
    <p>Line 9</p>
    <p>Line 10</p>
    <p>Line 11</p>
    <p>Line 12</p>
</body>
</html>

```

在此例子中，指定了绝对定位的<div>就是水印。它从页面顶部开始，在窗口卷动时，它仍然在原地。这里在 onscroll 事件处理函数中添加了一小段简单的代码，用来移动水印使之始终与 scrollTop 属性相等，这就产生了总是停留在窗口右上角的效果。

9.5.4 变化事件

虽然变化事件已经是 DOM 标准的一部分，但是目前还没有任何主流的浏览器实现了它。因此，下面的信息仅仅是为提供对标准所定义的内容的清晰的描述，在这里我们不讨论如何使用这些事件。

变化事件包括以下内容：

- DOMSubtreeModified——当文档或者元素的子树因为添加或者删除节点而改变时触发。
- DOMNodeInserted——当一个节点作为另一个节点的子节点插入时触发。
- DOMNodeRemoved——当一个节点作为另一个节点的子节点被删除时触发。
- DOMNodeRemovedFromDocument——当一个节点从文档中删除时触发。
- DOMNodeInsertedIntoDocument——当一个节点插入到文档中时触发。

这些事件的目的是，提供一个独立于语言的事件范例，使其可使用在所有基于 XML 的语言中（诸如 XHTML、SVG 和一些更新的语言如 MathML 等）。

9.6 跨平台的事件

至此，你已在各个例子中看到了多种不同类型浏览器以及特征检测。在实际的代码中，你会希望尽可能减少在主要代码中使用这种检测的次数。这样，大部分开发人员需要有一个跨浏览器的事件的处理方式，以使所有的浏览器和特征检测就可在后台完成。

这一节中的跨浏览器的代码的目的是，尽可能地弱化 IE 事件模型和 DOM 事件模型之间的区别，使一部分代码能在所有主流浏览器上几乎完全相同的运行。当然，还是存在一些局限性的，例如 IE 缺乏对双向事件流的支持，但仍然可以涵盖 80%~90% 的应用。

9.6.1 EventUtil 对象

无论何时，要创建可用在同一个任务中的多个函数，最好创建一个管理他们的容器对象。这样，调试时就可以容易地指出函数是在哪里定义的。

在此例子中，EventUtil 对象是这节中所有与事件相关的函数定义的容器。因为没有属性，且只需这个对象的一个实例，所以没有必要定义一个类：

```
var EventUtil = new Object;
```

9.6.2 添加/删除事件处理函数

你在前面已看到，IE 用 attachEvent() 方法为元素分配任意数目的事件处理函数，而 DOM 则用 addEventListener() 方法。所以，第一个 EventUtil 对象的方法就是创建一个统一的分配事件处理函数的方式，我们称其为 addEventHandler()（这样不会与任何浏览器的实现相混淆）。这个方法有三个参数：要分配事件处理函数的对象，要处理的事件的名称以及要分配的函数。因为 IE 不支持事件捕获，所以这个方法只会在冒泡阶段分配事件处理函数。在方法的内部，使用了一个简单的检测算法以在正确的时间地点使用正确的功能。

```
EventUtil.addHandler = function (oTarget, sEventType, fnHandler) {
    if (oTarget.addEventListener) { //for DOM-compliant browsers
        oTarget.addEventListener(sEventType, fnHandler, false);
    } else if (oTarget.attachEvent) { //for IE
        oTarget.attachEvent("on" + sEventType, fnHandler);
    } else { //for all others
        oTarget["on" + sEventType] = fnHandler;
    }
};
```

这个方法中的这段代码，用特征检测来判断要用何种方式来添加事件处理函数。if 语句的第一个分支是针对支持 addEventListener() 方法的 DOM 兼容的浏览器。浏览器是 DOM 兼容的时，就使用 addEventListener() 方法来添加事件处理函数，并且最后一个参数是 false，指定在冒泡阶段。

在 if 语句的第二部分中，进行对 IE 的 attachEvent() 方法的特征检测。注意，为了能正常工作，你必须在事件类型前加上字符串 "on"（记住，attachEvent() 方法的第一个参数接受

的是事件处理函数的名字而不是事件类型的名字)。

最后的 else 子句针对既不兼容 DOM 又不兼容 IE 的其他浏览器。还没有太多浏览器满足这个条件，所以这个分支的运行几率不是很高。

当然，不能仅添加事件处理函数；还必须找到一个方法来删除它们。在这点上，EventUtil 对象有另外一个方法 removeEventHandler()。正如你期望的那样，这个函数接受和 addEventHandler()一样的参数，用的也是差不多的算法：

```
EventUtil.removeEventHandler = function (oTarget, sEventType, fnHandler) {
    if (oTarget.removeEventListener) { //for DOM-compliant browsers
        oTarget.removeEventListener(sEventType, fnHandler, false);
    } else if (oTarget.detachEvent) { //for IE
        oTarget.detachEvent("on" + sEventType, fnHandler);
    } else { //for all others
        oTarget["on" + sEventType] = null;
    }
};
```

可以看到，这段代码就像是 addEventHandler() 代码的镜像，几乎完全一样，只不过在最后的 else 语句中，将事件处理函数设置为 null，不再使用 fnHandler。

我们在以下的例子中展示一下这些方法：

```
<html>
<head>
    <title>Add/Remove Event Handlers Example</title>
    <script type="text/javascript">
        var EventUtil = new Object;
        EventUtil.addHandler = function (oTarget, sEventType,
fnHandler) {
            if (oTarget.addEventListener) {
                oTarget.addEventListener(sEventType, fnHandler, false);
            } else if (oTarget.attachEvent) {
                oTarget.attachEvent("on" + sEventType, fnHandler);
            } else {
                oTarget["on" + sEventType] = fnHandler;
            }
        };

        EventUtil.removeEventHandler = function (oTarget, sEventType,
fnHandler) {
            if (oTarget.removeEventListener) {
                oTarget.removeEventListener(sEventType, fnHandler, false);
            } else if (oTarget.detachEvent) {
                oTarget.detachEvent("on" + sEventType, fnHandler);
            } else {
                oTarget["on" + sEventType] = null;
            }
        };
};

function handleClick() {
    alert("Click!");
}
```

```

        var oDiv = document.getElementById("div1");
        EventUtil.removeEventHandler(oDiv, "click", handleClick);
    }

    window.onload = function() {
        var oDiv = document.getElementById("div1");
        EventUtil.addHandler(oDiv, "click", handleClick);
    }
</script>
</head>
<body>
    <div id="div1" style="background-color: red; width: 100px; height:
100px"></div>
</body>
</html>

```

在这段代码中，`onload` 事件为 ID 是“div1”的`<div>`分配一个 `onclick` 事件处理函数。点击`<div>`时，会出现警告框，显示“Click！”，然后删除事件处理函数。然后再点击这个`<div>`，就不会出现警告框。

9.6.3 格式化 `event` 对象

一种对付 IE 和 DOM 中的 `event` 对象之间区别的最佳手段是，调整它们使之尽可能地表现相似。因为更多的浏览器使用的是 DOM 的事件模型，所以将 IE 的事件模型调整为接近于 DOM 事件模型就可以了。

下表是 DOM 和 IE 的 `event` 对象属性和方法之间的一个对比。大部分情况下，两者的事对象和方法都有能力完成同样的事情（例如阻止默认行为），只不过使用了不同的实现方式。这个表格显示了 IE 如何完成 DOM 的行为。虽然无法完全精确地将所有属性复制到 IE 中（例如 `bubbles` 和 `cancelBubble`），但是还是可以得到在大多数情况下大体上相同的事件对象。

DOM 属性/方法	IE 属性/方法
<code>altKey</code>	<code>altKey</code>
<code>bubbles</code>	-
<code>button</code>	<code>button</code>
<code>cancelBubble</code>	<code>cancelBubble</code>
<code>charCode</code>	<code>keyCode</code>
<code>clientX</code>	<code>clientX</code>
<code>clientY</code>	<code>clientY</code>
<code>ctrlKey</code>	<code>ctrlKey</code>
<code>currentTarget</code>	-
<code>detail</code>	-
<code>eventPhase</code>	-
<code>isChar</code>	-
<code>keyCode</code>	<code>keyCode</code>
<code>metaKey</code>	-
<code>pageX</code>	-
<code>pageY</code>	-
<code>preventDefault()</code>	<code>returnValue = false;</code>
<code>relatedTarget</code>	<code>fromElement</code> <code>toElement</code>

(续)

DOM 属性/方法	IE 属性/方法
screenX	screenX
screenY	screenY
shiftKey	shiftKey
stopPropagation()	cancelBubble = true;
target	srcElement
timeStamp	-
type	type

首先，给 EventUtil 定义一个新方法 formatEvent()，它可以接受一个参数，也就是 event 对象：

```
EventUtil.formatEvent = function (oEvent) {
    return oEvent;
}
```

第一件要做的事情就是，用前面一章中的浏览器检测脚本检查是否为 Windows 上的 IE。这里，因为这个脚本主要的目的是修复只存在于 Windows 上的 IE 中的问题，所以只检查这个特定的浏览器。

```
EventUtil.formatEvent = function (oEvent) {
    if (isIE && isWin) {

    }
    return oEvent;
};
```

295

为使事情简单些，只需直接照前面表中的属性和方法来对 IE 的 event 对象进行调整，使之与 DOM 的模型接近。其中 altKey 属性已有，而不能重新创建 bubbles 属性，button 属性已有，cancelBubble 属性已有、不能创建 cancelable 属性，然后要处理的就是 charCode 属性。

前面也说过，IE 中的字符代码是在 keypress 事件发生时包含在 keyCode 属性中的。所以，如果事件类型是 keypress，需要创建 charCode 属性，值等于 keyCode；否则，将其设置为 0：

```
EventUtil.formatEvent = function (oEvent) {
    if (isIE && isWin) {
        oEvent.charCode = (oEvent.type == "keypress") ? oEvent.keyCode : 0;
    }
    return oEvent;
};
```

继续看表，clientX、clientY 以及 ctrlKey 在 IE 和 DOM 中都一样。然后，我们不能精确地自己创建 currentTarget 或者 detail 属性，所以就不管它们吧。然而，还是可以给 eventPhase 设置一个值。这个属性始终等于 2 代表冒泡阶段，因为 IE 仅支持这个阶段：

```
EventUtil.formatEvent = function (oEvent) {
    if (isIE && isWin) {
        oEvent.charCode = (oEvent.type == "keypress") ? oEvent.keyCode : 0;
        oEvent.eventPhase = 2;
    }
    return oEvent;
};
```

};

表中接下来是 isChar 属性，当 charCode 不为 0 时为 true。

```
EventUtil.formatEvent = function (oEvent) {
    if (isIE && isWin) {
        oEvent.charCode = (oEvent.type == "keypress") ? oEvent.keyCode : 0;
        oEvent.eventPhase = 2;
        oEvent.isChar = (oEvent.charCode > 0);
    }
    return oEvent;
};
```

然后，keyCode 属性在两种浏览器中都一样，metaKey 属性在 IE 中也不适用，那么下面就是 pageX 和 pageY 了。尽管 IE 的 event 对象中没有等同的属性，不过可以通过 clientX 和 clientY 的值以及文档主体的 scrollLeft 和 scrollTop 的值来计算出它们。

```
EventUtil.formatEvent = function (oEvent) {
    if (isIE && isWin) {
        oEvent.charCode = (oEvent.type == "keypress") ? oEvent.keyCode : 0;
        oEvent.eventPhase = 2;
        oEvent.isChar = (oEvent.charCode > 0);
        oEvent.pageX = oEvent.clientX + document.body.scrollLeft;
        oEvent.pageY = oEvent.clientY + document.body.scrollTop;
    }
    return oEvent;
};
```

下面是 preventDefault() 方法。只要给 event 对象定义一个将 returnValue 设置为 false 的方法就行了：

```
EventUtil.formatEvent = function (oEvent) {
    if (isIE && isWin) {
        oEvent.charCode = (oEvent.type == "keypress") ? oEvent.keyCode : 0;
        oEvent.eventPhase = 2;
        oEvent.isChar = (oEvent.charCode > 0);
        oEvent.pageX = oEvent.clientX + document.body.scrollLeft;
        oEvent.pageY = oEvent.clientY + document.body.scrollTop;
        oEvent.preventDefault = function () {
            this.returnValue = false;
        };
    }
    return oEvent;
};
```

注意 this 对象的使用。在 event 对象的上下文中，this 指代的是 event 对象。

然后，relateTarget 属性即可能是 fromElement 也可能 is toElement 属性，这由事件的类型决定：

```
EventUtil.formatEvent = function (oEvent) {
    if (isIE && isWin) {
        oEvent.charCode = (oEvent.type == "keypress") ? oEvent.keyCode : 0;
        oEvent.eventPhase = 2;
```

```

oEvent.isChar = (oEvent.charCode > 0);
oEvent.pageX = oEvent.clientX + document.body.scrollLeft;
oEvent.pageY = oEvent.clientY + document.body.scrollTop;
oEvent.preventDefault = function () {
    this.returnValue = false;
};

if (oEvent.type == "mouseout") {
    oEvent.relatedTarget = oEvent.toElement;
} else if (oEvent.type == "mouseover") {
    oEvent.relatedTarget = oEvent.fromElement;
}
}

return oEvent;
};

```

下面的 screenX、screenY 和 shiftKey 属性都一样，可以直接用。然后又是 stopPropagation() 方法，这个方法只需直接将 cancelBubble 设置为 true：

```

EventUtil.formatEvent = function (oEvent) {
    if (isIE && isWin) {
        oEvent.charCode = (oEvent.type == "keypress") ? oEvent.keyCode : 0;
        oEvent.eventPhase = 2;
        oEvent.isChar = (oEvent.charCode > 0);
        oEvent.pageX = oEvent.clientX + document.body.scrollLeft;
        oEvent.pageY = oEvent.clientY + document.body.scrollTop;
        oEvent.preventDefault = function () {
            this.returnValue = false;
        };

        if (oEvent.type == "mouseout") {
            oEvent.relatedTarget = oEvent.toElement;
        } else if (oEvent.type == "mouseover") {
            oEvent.relatedTarget = oEvent.fromElement;
        }

        oEvent.stopPropagation = function () {
            this.cancelBubble = true;
        };
    }
    return oEvent;
};

```

297

然后是 target 属性，它确切地等同于 IE 的 srcElement 属性：

```

EventUtil.formatEvent = function (oEvent) {
    if (isIE && isWin) {
        oEvent.charCode = (oEvent.type == "keypress") ? oEvent.keyCode : 0;
        oEvent.eventPhase = 2;
        oEvent.isChar = (oEvent.charCode > 0);
        oEvent.pageX = oEvent.clientX + document.body.scrollLeft;
        oEvent.pageY = oEvent.clientY + document.body.scrollTop;
        oEvent.preventDefault = function () {
            this.returnValue = false;
        };
    }
}

```

```

    );
    if (oEvent.type == "mouseout") {
        oEvent.relatedTarget = oEvent.toElement;
    } else if (oEvent.type == "mouseover") {
        oEvent.relatedTarget = oEvent.fromElement;
    }

    oEvent.stopPropagation = function () {
        this.cancelBubble = true;
    };

    oEvent.target = oEvent.srcElement;
}
return oEvent;
};

```

对于 timestamp 属性，只需创建一个当前时间的 Date 对象，并从中获得毫秒数就行了：

```

298 EventUtil.formatEvent = function (oEvent) {
    if (isIE && isWin) {
        oEvent.charCode = (oEvent.type == "keypress") ? oEvent.keyCode : 0;
        oEvent.eventPhase = 2;
        oEvent.isChar = (oEvent.charCode > 0);
        oEvent.pageX = oEvent.clientX + document.body.scrollLeft;
        oEvent.pageY = oEvent.clientY + document.body.scrollTop;
        oEvent.preventDefault = function () {
            this.returnValue = false;
        };

        if (oEvent.type == "mouseout") {
            oEvent.relatedTarget = oEvent.toElement;
        } else if (oEvent.type == "mouseover") {
            oEvent.relatedTarget = oEvent.fromElement;
        }

        oEvent.stopPropagation = function () {
            this.cancelBubble = true;
        };

        oEvent.target = oEvent.srcElement;
        oEvent.time = (new Date()).getTime();
    }
}

return oEvent;
};

```

最后，type 属性在 IE 和 DOM 中也一样，这样 formatEvent 方法就完成了。然而，这个方法并不是单独使用的。必须在另一个能获得 event 对象引用的方法中使用。

9.6.4 获取事件对象

不幸的是，IE 和 DOM 使用不同的方法来获取 event 对象。在 IE 中，event 对象是与 window

对象相关的，而在 DOM 中，它独立于任何其他对象，并且是作为参数传递的。因此，很难使 IE 的事件模型表现成 Mozilla 那样。所以我们不会尝试改变其中一个使它像另一个，而是创建一个新的方法，使其可同时在两种浏览器中使用并获取 event 对象——getEvent() 方法。

这个 getEvent() 方法不接受任何参数，它唯一的目的是返回 event 对象。首先要处理的是 IE，我们检查 window.event 是否存在，然后在返回事件对象前用 formatEvent() 方法格式化它。

```
EventUtil.getEvent = function() {
    if (window.event) {
        return this.formatEvent(window.event);
    }
};
```

下面要处理 DOM 的情况。记住，DOM 兼容的浏览器是将 event 对象作为参数传递给事件处理函数的。还要记住，函数其实也是一个对象，也有属性。在这里，很有意思的，我们需要一个属性 caller。299

每个函数都有一个 caller 属性，它包含了指向调用它的方法的引用。例如，如果 funcA() 调用了 funcB()，那么 funcB.caller 就等于 funcA。假设某个事件处理函数调用了 EventUtil.getEvent()，那么 EventUtil.getEvent.caller 就指向这个事件处理函数本身。

回忆在第 2 章中学到的函数的 arguments 属性。因为 caller 属性是指向函数的，所以可访问事件处理函数的 arguments 属性。而 event 对象总是事件处理函数的第一个参数，也就是说可以访问事件处理函数的 argument[0] 来获取 event 对象：

```
EventUtil.getEvent = function() {
    if (window.event) {
        return this.formatEvent(window.event);
    } else {
        return EventUtil.getEvent.caller.arguments[0];
    }
};
```

这个方法可在事件处理函数中使用，如下：

```
oDiv.onclick = function () {
    var oEvent = EventUtil.getEvent();
};
```

最好将这几节中定义的 EventUtil 的代码单独放入一个 eventutil.js 文件中，这样就能在任何页面中调用它。

9.6.5 示例

这段代码是从鼠标事件一节中的例子修改过来的：

```
<html>
<head>
    <title>Mouse Events Example</title>
    <script type="text/javascript" src="detect.js"></script>
    <script type="text/javascript" src="eventutil.js"></script>
```

```

<script type="text/javascript">

    EventUtil.addHandler(window, "load", function () {
        var oDiv = document.getElementById("div1");

        EventUtil.addHandler(oDiv, "mouseover", handleEvent);
        EventUtil.addHandler(oDiv, "mouseout", handleEvent);
        EventUtil.addHandler(oDiv, "mousedown", handleEvent);
        EventUtil.addHandler(oDiv, "mouseup", handleEvent);
        EventUtil.addHandler(oDiv, "click", handleEvent);
        EventUtil.addHandler(oDiv, "dblclick", handleEvent);

    });

    function handleEvent() {
        var oEvent = EventUtil.getEvent();
        var oTextbox = document.getElementById("txt1");
        oTextbox.value += "\n> " + oEvent.type;
        oTextbox.value += "\n    target is " + oEvent.target.tagName;
        if (oEvent.relatedTarget) {
            oTextbox.value += "\n    relatedTarget is "
                + oEvent.relatedTarget.tagName;
        }
    }

</script>
</head>
<body>
    <p>Use your mouse to click and double click the red square.</p>
    <div style="width: 100px; height: 100px; background-color: red"
id="div1"></div>
    <p><textarea id="txt1" rows="15" cols="50"></textarea></p>
</body>
</html>

```

此例子可在所有的 DOM 兼容的浏览器中运行，也可以在 IE 兼容的浏览器中运行，使用了在格式化过的 event 对象中的 target 和 relatedTarget 属性。

9.7 小结

这一章介绍了 JavaScript 中的事件的概念。学习了事件（它代表一个动作的发生）以及事件处理函数（它表示指定的某个事件发生时执行的函数）之间的区别。还学了分配事件处理函数的不同方式以及在 IE 和 DOM 标准中，为同一事件分配多个事件处理函数的不同的方法。这一章还介绍了事件流的概念，以及两种不同的事件流，冒泡型和捕获型，并都做了详细介绍。

然后详细研究了 event 对象，它用来为开发人员提供关于特定事件的详细信息。这一章还对 IE 以及 DOM 所支持的不同 event 对象做了比较。还学习了不同的事件分类：鼠标事件、键盘事件、HTML 事件以及变化事件。

本章最后一节带你创建了一个跨浏览器事件库，可让你使用同一套方法来访问事件对象，添加/删除事件处理函数，而无需考虑浏览器检测的问题。



虽然基本的 DOM 相当简单，但是可以应用不同方式来处理文档底层的 DOM 树。首先，可利用现行浏览器中非标准的属性和方法，以及鲜为人知且有待进一步利用的 DOM 标准接口。

本章中讨论的接口中，有些是 DOM 定义的，其他的不是，不过它们都能提高处理 DOM 文档和节点的能力。

10.1 样式编程

当 CSS（层叠样式表）于 1996 年被提出后，它颠覆了开发人员格式化 HTML 页面的方式。抛弃使用诸如``和``之类的标签，页面开始使用 CSS 来定义字体以及其他内容的外观。很自然，支持 CSS 的下一步就是让 JavaScript 能够访问样式表。

IE 4.0 为每一个页面上的元素都引入了一个 `style` 对象来管理元素的 CSS 样式。DOM 最后也采用这个方式，并将其作为访问元素的样式信息的标准手段。

如今，`style` 对象包含与每个 CSS 样式对应的特性，虽然格式不同。单个单词的 CSS 样式，以相同名字的特性来表示（例如，`color` 样式通过 `style.color` 来表示）；但是 `style` 对象中，两个单词的样式的表示方式是通过将第一个单词加上首字母大写的第二个单词，且单词间没有横线（例如，`background-color` 样式对应 `style.backgroudColor`）来表示的。下面的表格列出一些常用的 CSS 特性及它们对应的 JavaScript 中 `style` 对象的表示：

303

CSS 样式特性	JavaScript 样式属性
<code>background-color</code>	<code>style.backgroundColor</code>
<code>color</code>	<code>style.color</code>
<code>font</code>	<code>style.font</code>
<code>font-family</code>	<code>style.fontFamily</code>
<code>font-weight</code>	<code>style.fontWeight</code>

要用 JavaScript 来更改样式的值，只需将 CSS 的字符串分配给它们的样式对象的特性就行。例如，下面的代码将

的 CSS border 特性更改为“1px solid black”：

```
var oDiv = document.getElementById("div1");
oDiv.style.border = "1px solid black";
```

也可以使用 style 对象来获取任何内联样式(直接通过 HTML 的 style 特性来分配的样式)的值。例如，下面的页面当点击

按钮时，显示它们的背景色：

```
<html>
  <head>
    <title>Style Example</title>
    <script type="text/javascript">
      function sayStyle() {
        var oDiv = document.getElementById("div1");
        alert(oDiv.style.backgroundColor);
      }
    </script>
  </head>
  <body>
    <div id="div1" style="background-color: red; height: 50px; width: 50px"></div><br />
    <input type="button" value="Get Background Color" onclick="sayStyle()" />
  </body>
</html>
```

同样的技巧也可应用于用户将鼠标移动到页面上指定元素时的翻转效果。尽管 CSS Level 2 提供 hover 伪类来为所有元素实现翻转效果，但并不是所有浏览器都支持所有元素。为克服这个支持上的缺陷，就要用到 style 对象。

```
304 <html>
  <head>
    <title>Style Example</title>
  </head>
  <body>
    <div id="div1"
      style="background-color: red; height: 50px; width: 50px"
      onmouseover="this.style.backgroundColor = 'blue'"
      onmouseout="this.style.backgroundColor = 'red'"></div>
  </body>
</html>
```

当把鼠标移动到红色

上时，它就会变成蓝色；当把鼠标移走，它又变回红色。注意这个事件处理函数用 this 关键字来指代

本身，并通过它获取对 style 对象的访问。

style 对象还包含 cssText 特性，这个特性包含了所有描述元素样式的 CSS 字符串：

```
<html>
  <head>
    <title>Style Example</title>
  </head>
  <body>
    <div id="div1"
      style="background-color: red; height: 50px; width: 50px"
      ></div>
```

```

        onclick="alert(this.style.cssText)"></div>
    </body>
</html>

```

当点击例子中的<div>时，就会出现文本"background-color: red; height: 50px; width: 50px"。

10.1.1 DOM 样式的方法

DOM 还描述了一些样式对象的方法，以达到与单独的 CSS 样式的定义部分进行交互的目的：

- `getPropertyValue(propertyName)`——返回 CSS 特性 *propertyName* 的字符串值。特性必须按照 CSS 样式定义，例如"background-color"而不是"backgroundColor"。
- `getPropertyPriority()`——如果在规则中指定 CSS 特性 "important"，则返回 "important"；否则返回空字符串。
- `item(index)`——返回在给定索引 *index* 处的 CSS 特性的名称，例如"background-color"。
- `removeProperty(propertyName)`——从 CSS 定义中删除 *propertyName*。
- `setProperty(propertyName, value, priority)`——按照指定的优先级 *priority* 来设置 CSS 特性 *propertyName* 的 *value* 值 ("important"或者为空字符串)。

这里有个简单的例子：

```

<html>
    <head>
        <title>Style Example</title>
        <script type="text/javascript">
            function useMethods() {
                var oDiv = document.getElementById("div1");
                alert(oDiv.style.item(0));      //outputs "background-color"
                alert(oDiv.style.getPropertyValue("background-color"));
                oDiv.style.removeProperty("background-color");
            }
        </script>
    </head>
    <body>
        <div id="div1" style="background-color: red; height: 50px; width:
50px"></div><br />
        <input type="button" value="Use Methods" onclick="useMethods()" />
    </body>
</html>

```

305

当点击页面上的按钮时，会发生三件事。首先，在第一个位置（位置 0）上的条目出现"background-color"，因为这是<div>的第一个 style 特性。然后，background-color 的当前的值（red，红色）出现。最后 background-color 属性被一起删除，使得<div>不可见。

这些方法可以用在不同的 style 对象属性上，来完成同样的实现：例如，它可以返回<div>的背景色。

```

<html>
  <head>
    <title>Style Example</title>
    <script type="text/javascript">
      function sayStyle() {
        var oDiv = document.getElementById("div1");
        alert(oDiv.style.getPropertyValue("background-color"));
      }
    </script>
  </head>
  <body>
    <div id="div1" style="background-color: red; height: 50px; width: 50px"></div><br />
    <input type="button" value="Get Background Color" onclick="sayStyle()" />
  </body>
</html>

```

这是用 `style` 方法重写过的翻转效果：

```

<html>
  <head>
    <title>Style Example</title>
  </head>
  <body>
    <div id="div1"
      style="background-color: red; height: 50px; width: 50px"
      onmouseover="this.style.setProperty('background-color', 'blue', '')"
      onmouseout="this.style.setProperty('background-color', 'red', '')">
    </div>
  </body>
</html>

```

306

IE 不支持 DOM 的 `style` 方法，因此，最好直接使用 `style` 对象的特性来获取和设置 CSS 的特性。

10.1.2 自定义鼠标提示

另一个有趣的 `style` 对象的用法是创建自定义的鼠标提示，所谓鼠标提示，就是把鼠标移动到图片按钮上时，出现的帮助性黄色方框。通过使用 `title` 特性，HTML 元素也可提供普通的文本鼠标提示，例如：

```
<a href="http://www.wrox.com" title="Wrox Site">Wrox</a>
```

然而，仅有这些文本的鼠标提示，可能还不足够。如果想创建包含粗体或者斜体的鼠标提示，甚至包含图片的鼠标提示，可以通过创建隐藏的`<div>`来实现，然后当鼠标移到指定目标上时，就是显示这个隐藏的`<div>`。这其实与翻转的代码在本质上是一样的，只涉及一个额外步骤：将`<div>`移到靠近鼠标的某个位置。为能修正`<div>`的位置，可以利用 `event` 对象的 `clientX` 和 `clientY` 特性。注意因为这些特性存在于所有的 `event` 对象的实例中，所以没必要使用本书前面介绍的 `EventUtil` 对象：

```

<html>
  <head>
    <title>Style Example</title>
    <script type="text/javascript">
      function showTip(oEvent) {
        var oDiv = document.getElementById("divTip1");
        oDiv.style.visibility = "visible";
        oDiv.style.left = oEvent.clientX + 5;
        oDiv.style.top = oEvent.clientY + 5;
      }

      function hideTip(oEvent) {
        var oDiv = document.getElementById("divTip1");
        oDiv.style.visibility = "hidden";
      }
    </script>
  </head>
  <body>
    <p>Move your mouse over the red square.</p>
    <div id="div1"
         style="background-color: red; height: 50px; width: 50px"
         onmouseover="showTip(event)" onmouseout="hideTip(event)"></div>

    <div id="divTip1"
         style="background-color: yellow; position: absolute; visibility:
         hidden; padding: 5px">
      <span style="font-weight: bold">Custom Tooltip</span><br />
      More details can go here.
    </div>
  </body>
</html>

```

307

这个例子中，直接给 `showTip()` 和 `hideTip()` 方法传送了 `event` 对象（与我们在前一章讨论的一样）。当调用 `showTip()` 时，首先通过将 `divTip1` 的 `style.visibility` 设置为 "visible" 将其显示出来。然后，函数通过设置 `style.left` 和 `style.top` 等于 `event.clientX` 和 `event.clientY` 将 `divTip1` 移到相应位置。为保证提示不会直接出现在鼠标下面，左端和顶端的坐标各增加了 5 个像素。而 `hideTip()` 函数仅将 `style.visibility` 设置回 "hidden"，提示就不可见了。

10.1.3 可折叠区域

使用简单的技术，就可在网页上创建可折叠区域。利用这种功能，可以在不必要的时候隐藏某些设置和字段。这种手段在过去几年中越来越流行。它现在都已被加入到 Windows XP 文件系统的外壳中了。

可折叠区域的基本思想是，可以通过点击某个地方来显示或者隐藏屏幕中的某个区域。当某个区域折叠起来，其他区域就会移动位置到空余的地方。使用 CSS 的 `display` 特性就可以完成同样的操作。当 `display` 设置为 `none` 时，元素就被从页面流中移除，页面重绘时就像这个元素不存在一样。这与将 `visibility` 设置为 `hidden` 不同，它仅仅隐藏元素，以留下一个元素所占

区域的空白。

一般来说，可折叠区域会有个标题栏，这部分总是可见的，还有个内容区域，这部分可以折叠或展开。要在 Web 上模拟这个效果，可以用一对元素：一个作为标题，另一个用来存放内容。还需要一个小函数来展开/折叠内容。将这些整合在一起，就形成了下面的例子：

```

<html>
    <head>
        <title>Style Example</title>
        <script type="text/javascript">
            function toggle(sDivId) {
                var oDiv = document.getElementById(sDivId);
                oDiv.style.display = (oDiv.style.display == "none") ? "block" :
                "none";
            }
        </script>
    </head>
    <body>
        <div style="background-color: blue; color: white; font-weight: bold;
padding: 10px; cursor: pointer"
            onclick="toggle('divContent1')">Click Here</div>
        <div style="border: 3px solid blue; height: 100px; padding: 10px"
            id="divContent1">This is some content
            to show and hide.</div>
        <p>&nbsp;</p>
        <div style="background-color: blue; color: white; font-weight: bold;
padding: 10px; cursor: pointer"
            onclick="toggle('divContent2')">Click Here</div>
        <div style="border: 3px solid blue; height: 100px; padding: 10px"
            id="divContent2">This is some content
            to show and hide.</div>
    </body>
</html>

```

308

这个页面显示了两个可折叠区域。用来显示和隐藏的两个元素的名字是和。当调用 toggle() 函数时，将要执行动作的的 ID 作为参数传给它。如果的 style.display 等于 none（也就是说它没有被显示），那么值就切换为 block（元素的默认值）；否则，将 style.display 设置为 none。这样就可以在网页上创建可折叠区域了。

10.1.4 访问样式表

使用 style 对象可以方便地获取某个有 style 特性的元素的 CSS 样式。但它无法表示由 CSS 规则或者在 style 特性外部定义的类定义的元素的 CSS 样式。例如，在<style/>元素中或在外部的样式表中定义的 CSS 样式。下面的例子将描述这个问题：

```

<html>
    <head>
        <title>Runtime Style Example</title>
        <style type="text/css">

```

```


}
</style>
<script type="text/javascript">
function getBackgroundColor() {
    var oDiv = document.getElementById("div1");
    alert(oDiv.style.backgroundColor);
}
</script>
</head>
<body>
<div id="div1" class="special"></div>
<input type="button" value="Get Background Color"
onclick="getBackgroundColor()" />
</body>
</html>


```

在这段代码中，`<div/>`的样式定义在类 `special` 中。当点击按钮并调用 `getBackgroundColor()` 时，警告框显示的是空字符串，因为 CSS 数据并非存储在 `style` 属性中；它是存储在类中的。所以问题出现了，怎样才能访问 CSS 类？

第一步，获取定义类的样式表的引用。使用 `document.styleSheets` 集合实现这个目的，这个集合中包含了 HTML 页面中所有样式表的引用，包含了所有的`<style/>`元素（JavaScript 认为这是全功能的样式表）。DOM 指定样式表对象有以下特性：

- `disabled`——表示样式表是否被禁用。
- `href`——用于外部引用文件样式表的 URL；对于`<style/>`元素，这个值应该是 `null`，不过 Mozilla 返回的是当前 HTML 页面的 URL。
- `media`——可以使用该样式表的媒体类型，由 HTML 的 `media` 特性指定。IE 在实现这个属性上有错误，返回的是与 `media` 特性包含相同内容的字符串。
- `ownerNode`——指定样式表的 DOM 节点(`<link/>`或者`<style/>`元素)。IE 不支持这个特性。
- `parentStyleSheet`——如果样式表是通过 CSS 的`@import` 语句来加载的，那么这个特性将指出，出现`@import` 语句的样式表。
- `title`——通过 HTML `title` 特性分配给样式表的标题，可用在`<link/>`和`<style/>`上。
- `type`——样式表的 `mime` 类型；对于 CSS 通常是 `text/css`。

309

Opera 不支持 JavaScript 的样式表访问或者操作。Safari 提供有限的支持，但是不能访问 `rel` 特性为 "alternate stylesheet" 的禁用的样式表。

因为浏览器之间有很大不同，访问样式表中单独的规则是有技巧的。DOM 为每一个样式表指定一个 `cssRules` 的集合，它包含所有的定义在样式表中的 CSS 规则。Mozilla 和 Safari 正确实现了这个标准，不过 IE 称这个集合为 `rules`。因此，在操作样式表的规则前，应该检测一下

该使用哪个名称:

```
var oCSSRules = document.styleSheets[0].cssRules || document.styleSheets[0].rules;
```

每个规则都包含 selectorText 特性, 可以返回在左花括号之前某一条 CSS 规则的全部文本。回顾前面例子中的 CSS 规则:

```
div.special {
    background-color: red;
    height: 10px;
    width: 10px;
    margin: 10px;
}
```

这条规则的 selectorText 特性是 div.special (不过, 实际上 IE 会将所有的标签名大写, 所以它最后应该是 DIV.special)

规则还包含 style 特性, 就像元素中的那样它是个 style 对象。因此, 前面的例子可以改进为, 使用 CSS 规则的 style 对象来报告正确的背景色, 而不是使用<div/>本身的 style 对象:

```
<html>
<head>
    <title>Accessing Style Sheets Example</title>
    <style type="text/css">
        div.special {
            background-color: red;
            height: 10px;
            width: 10px;
            margin: 10px;
        }
    </style>
    <script type="text/javascript">
        function getBackgroundColor() {
            var oCSSRules = document.styleSheets[0].cssRules ||
document.styleSheets[0].rules;
            alert(oCSSRules[0].style.backgroundColor);
        }
    </script>
</head>
<body>
    <div id="div1" class="special"></div>
    <input type="button" value="Get Background Color"
onclick="getBackgroundColor()" />
</body>
</html>
```

当点击本例中的按钮时, 警告框将根据 div.special 的规则正确显示出背景色。

规则上的 style 对象并不是只读的; 也可以修改它。但是在这里必须小心, 因为修改 CSS 的规则会影响到页面上所有使用它的元素。考虑这个例子:

```

<html>
  <head>
    <title>Accessing Style Sheets Example</title>
    <style type="text/css">
      div.special {
        background-color: red;
        height: 10px;
        width: 10px;
        margin: 10px;
      }
    </style>
    <script type="text/javascript">
      function changeBackgroundColor() {
        var oCSSRules = document.styleSheets[0].cssRules ||
document.styleSheets[0].rules;
        oCSSRules[0].style.backgroundColor = "blue";
      }
    </script>
  </head>
  <body>
    <div id="div1" class="special"></div>
    <div id="div2" class="special"></div>
    <div id="div3" class="special"></div>
    <input type="button" value="Change Background Color"
onclick="changeBackgroundColor()" />
  </body>
</html>

```

311

在这个例子中，三个<div/>元素都有"special"的 CSS 类。点击按钮后，style.backgroundColor 就被设置为"blue"，因此三个元素的背景色都改变了。因此，最好只修改单独的元素的 style 对象，而不要更改 CSS 规则的。改变某个元素的 style 对象将覆盖相应的 CSS 规则的设置：

```

<html>
  <head>
    <title>Accessing Style Sheets Example</title>
    <style type="text/css">
      div.special {
        background-color: red;
        height: 10px;
        width: 10px;
        margin: 10px;
      }
    </style>
    <script type="text/javascript">
      function changeBackgroundColor() {
        var oDiv = document.getElementById("div1");
        oDiv.style.backgroundColor = "blue";
      }
    </script>
  </head>
  <body>
    <div id="div1" class="special"></div>
    <div id="div2" class="special"></div>
    <div id="div3" class="special"></div>
  
```

```

<input type="button" value="Change Background Color"
onclick="changeBackgroundColor()" />
</body>
</html>

```

这个例子中，通过改变第一个的 style 对象，改变了它的背景色，而对其他元素无影响。

10.1.5 最终样式

除元素的 style 对象和 CSS 规则之外，还有元素的最终样式。最终样式由所有从内联样式和 CSS 规则计算得来的样式信息组成，用来告诉我们元素最后是如何显示在屏幕上的。和以前一样，IE 和 DOM 在它们各自的实现上存在不同。

1. IE 中的最终样式

微软为每个元素提供一个 currentStyle 对象，它包含了所有元素的 style 对象的特性和任何未被覆盖的 CSS 规则的 style 特性。currentStyle 对象与 style 对象的使用方式完全一样，特性和方法都一样。这表示即使背景色是在 CSS 规则中定义的，currentStyle. backgroundColor 也能包含正确的值：

```

<html>
    <head>
        <title>Computed Style Example</title>
        <style type="text/css">
            div.special {
                background-color: red;
                height: 10px;
                width: 10px;
                margin: 10px;
            }
        </style>
        <script type="text/javascript">
            function getBackgroundColor() {
                var oDiv = document.getElementById("div1");
                alert(oDiv.currentStyle.backgroundColor);
            }
        </script>
    </head>
    <body>
        <div id="div1" class="special"></div>
        <input type="button" value="Get Background Color"
onclick="getBackgroundColor()" />
    </body>
</html>

```

在这个例子中，点击按钮，显示出最后的背景颜色（red 红色），虽然它是定义在了 div.special 规则中。

记住，所有 currentStyle 对象的特性都是只读的，如果给它们赋值将会引发错误。因为，

`currentStyle` 对象是对所有采用了的样式（元素的、CSS 规则中的）的合计；它并非一个活生生的对象。如果要动态改变样式，必须使用前面讨论过的 `style` 对象。

2. DOM 中的最终样式

DOM 提供一个 `getComputedStyle()` 方法，它根据给定的元素创建类似 `style` 的对象。这个方法接受两个参数，需要获取样式的元素和诸如 `:hover` 或者 `:first-letter` 之类的伪元素（如果不需要，也可以为 `null`）。可以从 `document.defaultView` 对象中访问这个方法，这个对象代表文档当前渲染的视图（IE 和 Safari 不支持 `document.defaultView`）。

使用 DOM 的方法，可以这样改写前面的例子：

```
<html>
  <head>
    <title>Computed Style Example</title>
    <style type="text/css">
      div.special {
        background-color: red;
        height: 10px;
        width: 10px;
        margin: 10px;
      }
    </style>
    <script type="text/javascript">
      function getBackgroundColor() {
        var oDiv = document.getElementById("div1");
        alert(document.defaultView.getComputedStyle(oDiv,
null).backgroundColor);
      }
    </script>
  </head>
  <body>
    <div id="div1" class="special"></div>
    <input type="button" value="Get Background Color"
onclick="getBackgroundColor()" />
  </body>
</html>
```

313

使用 DOM 兼容的浏览器运行这个例子，当点击按钮时，将在警告框中显示背景色。

注意尽管某些浏览器支持这种功能，但是值得表现方式可能不同。例如，Mozilla 将所有的色彩都表示成 RGB 格式（例如，红色是 `rgb(255, 0, 0)`），而 Opera 将所有的颜色都转化为十六进制表示（例如，红色是 `#ff0000`）。所以，当使用 `getComputedStyle()` 方法时，最好要在多种不同的浏览器上进行测试。

10.2 innerText 和 innerHTML

尽管 DOM 带来了动态修改文档的能力，但对微软的开发人员来说，这还不够。IE 4.0 为所有的元素引入两个特性，以更方便的进行文档操作，这两个特性是 `innerText` 和 `innerHTML`。

其中，`innerText` 特性用来修改起始标签和结束标签之间的文本的。例如，假设有个空的`<div>`元素，希望将其变成`<div>New text for the div.</div>`。用 DOM 实现时，要这么做：

```
oDiv.appendChild(document.createTextNode("New text for the div."));
```

这段代码并不难读，但是很冗长。如果使用 `innerText`，只要这么做：

```
oDiv.innerText = "New text for the div.";
```

使用 `innerText`，代码更加简洁，且更容易理解。另外，`innerText` 会自动将小于号、大于号、引号和&符号进行 HTML 编码，所以丝毫不需担心这些特殊字符：

314

```
oDiv.innerText = "New text for the <div/>.";
```

这一行代码的执行结果是`<div>New text for the <div/>.</div>`。但如何一定要在元素中包含 HTML 标签呢？这就是 `innerHTML` 所要解决的问题。

应用 `innerHTML` 特性，可以直接给元素分配 HTML 字符串，而不需考虑使用 DOM 方法来创建元素。例如，假设一个空`<div>`要变成`<div>Hello World</div>`。使用 DOM，要用下面的代码：

```
var oStrong = document.createElement("strong");
oStrong.appendChild(document.createTextNode("Hello"));
var oEm = document.createElement("em");
oEm.appendChild(document.createTextNode("World"));
oDiv.appendChild(oStrong);
oDiv.appendChild(document.createTextNode(" ")); //space between "Hello" and "World"
oDiv.appendChild(oEm);
```

而使用 `innerHTML`，代码就变成：

```
oDiv.innerHTML = "<strong>Hello</strong> <em>World</em>";
```

七行代码一下就变成一行，这就是 `innerHTML` 的威力！

还可以使用 `innerText` 和 `innerHTML` 来获取元素的内容。如果元素只包含文本，那么 `innerText` 和 `innerHTML` 返回相同的值。但是，如果同时包含文本和其他元素，`innerText` 将只返回文本的表示，而 `innerHTML`，将返回所有元素和文本的 HTML 代码。下面的表格列出了根据特定代码 `innerText` 和 `innerHTML` 返回的不同值。

代 码	<code>innerText</code>	<code>innerHTML</code>
<code><div>Hello world</div></code>	"Hello world"	"Hello world"
<code><div>Hello world</div></code>	"Hello world"	"Hello world"
<code><div></div></code>	" "	""

最后，通过将 `innerText` 赋值给它自身，表示从指定的元素中删除所有的 HTML 标签。

```
oDiv.innerText = oDiv.innerText;
```

虽然它们并非 DOM 标准的一部分，但事实上现在大部分的浏览器，包括 IE、Opera 和 Safari，都支持 `innerText` 和 `innerHTML`；而 Mozilla，仅支持 `innerHTML`。

10.3 outerText 和 outerHTML

伴随 `innerText` 和 `innerHTML`，IE 4.0 还引入了 `outerText` 和 `outerHTML`，后两者的情况和前两者十分相似，只不过替换的是整个目标节点。例如，设置`<div>`的 `outerText`，将删除标签本身并把它替换成文本节点，考虑下面这行代码：

```
oDiv.outerText = "Hello world!";
```

这一行代码等同于下面一系列 DOM 操作：

```
var oText = document.createTextNode("Hello world! ");
var oDivParent = oDiv.parentNode;
oDivParent.replaceChild(oText, oDiv);
```

`outerText` 特性和 `innerText` 特性有个相同的规则，就是用相应的 HTML 实体替换所有的小于号、大于号、引号以及&号。类似的，`outerHTML` 和 `innerHTML` 的行为也差不多，创建用 HTML 字符串表示的所有必要的 DOM 节点。

```
oDiv.outerHTML = "<p>This is a paragraph.</p>";
```

这一行代码执行了以下一系列的 DOM 修改：

```
var oP = document.createElement("p");
oP.appendChild(document.createTextNode("This is a paragraph. "));
var oDivParent = oDiv.parentNode;
oDivParent.replaceChild(oP, oDiv);
```

虽然 `outerText` 和 `outerHTML` 为开发人员提供了强大的功能，但是它们并未清楚地表示出到底发生了什么（从代码中看不出来）。很多开发人员都避免使用 `outerText` 和 `outerHTML`，因为如果哪部分出了错，它们会让人相当头疼。一般来说，使用 DOM 方法更加安全，因为它的表述更清晰。

这两个特性都可用来获取某个元素的内容。不管元素的内容是什么，`outerText` 特性总是返回和 `innerText` 一样的内容。而另一方面，`outerHTML` 是返回元素完整的 HTML 代码，包括元素本身。下面的表格列出了根据特定代码 `outerText` 和 `outerHTML` 返回的值：

代 码	<code>outerText</code>	<code>outerHTML</code>
<code><div>Hello world</div></code>	"Hello world"	"<div>Hello world</div>"
<code><div>Hello world</div></code>	"Hello world"	"<div>Hello world</div>"
<code><div></div></code>	""	"<div></div>"

类似 `innerText`，可以用某种独特的方式使用 `outerText`。通过将 `outerText` 赋值给其自

身，就可以删除元素，并把它替换成包含元素中所有文本的一个文本节点。

```

316 <html>
      <head>
          <title>OuterText Example</title>
          <style type="text/css">
              div.special {
                  background-color: red;
                  padding: 10px;
              }
          </style>
          <script type="text/javascript">
              function useOuterText() {
                  var oDiv = document.getElementById("div1");
                  oDiv.outerText = oDiv.outerText;
                  alert(document.getElementById("div1"));
              }
          </script>

      </head>
      <body>
          <div id="div1" class="special">This is my original text</div>
          <input type="button" value="Use OuterText" onclick="useOuterText()" />
      </body>
</html>

```

当点击例子中的按钮时，`<div>`就被替换成包含“`This is my original text`”的文本节点。使用 `document.getElementById()` 来寻找 `div1`，可以发现`<div>`已不存在。在这个例子中，将在警告框中显示函数的结果（`null`）。

只有在 IE 和 Opera 中才支持 `outerText` 和 `outerHTML` 特性。

10.4 范围

为了能在较大尺度上控制页面，可以使用范围（range）。范围可以用来选择文档的某个部分而不管节点的边界（注意这个选取是在后台发生的，对用户不可见）。

当一般的 DOM 操作不足以用来改变文档时，范围将非常有用。通常，有两种不同的范围实现：一种是 DOM 的，另一种是 IE 的。

10.4.1 DOM 中的范围

DOM Level 2 定义了方法 `createRange()` 来创建范围。在 DOM 兼容的浏览器中的，这个方法属于 `document` 对象，所以可以这样创建一个范围：

```
var oRange = document.createRange();
```

类似于节点，范围是直接与文档相关的。要判断文档是否支持 DOM 风格的范围，可以使用在第 6 章中介绍的 `hasFeature()` 方法。

```
var supportsDOMRanges = document.implementation.hasFeature("Range", "2.0");
```

如果打算使用 DOM 的范围功能，最好先进行这个检测，然后将代码放在判断语句中：

```
if (supportsDOMRange) {
    var oRange = document.createRange();
    //range code here
}
```

1. DOM 范围中的简单选区

选择使用范围的文档的某部分的最简单的方法是用 `selectNode()` 或 `selectNodeContents()`。这些方法都只接受一个参数（DOM 节点），然后它们用节点中的信息来填充范围。

其中，`selectNode()` 方法将选择整个节点，包括它的子节点，而 `selectNodeContents()` 则选择节点所有的子节点。例如，考虑下面代码：

```
<p id="p1"><b>Hello</b> World</p>
```

这段代码可以用以下 JavaScript 代码来访问：

```
var oRange1 = document.createRange();
var oRange2 = document.createRange();
var oP1 = document.getElementById("p1");
oRange1.selectNode(oP1);
oRange2.selectNodeContents(oP1);
```

这个例子中的两个范围包含文档的不同选区：`oRange1` 包含`<p>`元素及所有子节点，而 `oRange2` 包含``元素及文本节点 `World`（见图 10-1）。

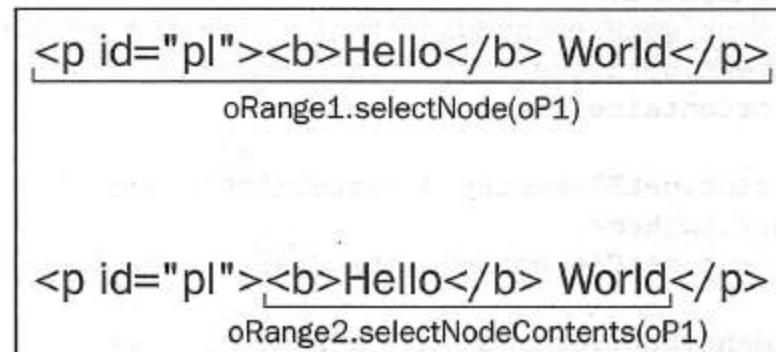


图 10-1

创建范围时，会分配给它一些特性：

- `startContainer`——范围是从哪个节点中开始的（选区中第一个节点的父节点）。
- `startOffset`——在 `startContainer` 中范围开始的偏移位置。如果 `startContainer` 是个文本节点、注释节点或者是 CData 节点，`startOffset` 是指范围开始前的字符数；否则，偏移是范围中第一个子节点的索引。
- `endContainer`——范围是在哪个节点中结束的（选区中最后一个节点的父节点）。
- `endOffset`——在 `endContainer` 中范围结束的偏移位置（与 `startOffset` 的规则一样）。
- `commonAncestorContainer`——`startContainer` 和 `endContainer` 都处于哪个最小的节点。

这些特性都是只读的，用来提供范围的额外信息。

当使用 `selectNode()` 时，`startContainer`、`endContainer` 和 `commonAncestorContainer` 均等同于传入的节点的父节点；`startOffset` 等同于给定节点在父节点的 `childNodes` 集合中的索引，而 `endOffset` 等同于 `startOffset` 加 1（因为只选择了一个节点）。

当使用 `selectNodeContents()` 时，`startContainer`、`endContainer` 和 `commonAncestorContainer` 等同于传入的节点，`startOffset` 等于 0；`endOffset` 等于子节点的数量 (`node.childNodes.length`)。

下面的例子将具体描述这些特性：

```

<html>
  <head>
    <title>DOM Range Example</title>
    <script type="text/javascript">
      function useRanges() {
        var oRange1 = document.createRange();
        var oRange2 = document.createRange();
        var oP1 = document.getElementById("p1");
        oRange1.selectNode(oP1);
        oRange2.selectNodeContents(oP1);

        document.getElementById("txtStartContainer1").value =
        oRange1.startContainer.tagName;
        document.getElementById("txtStartOffset1").value =
        oRange1.startOffset;
        document.getElementById("txtEndContainer1").value =
        oRange1.endContainer.tagName;
        document.getElementById("txtEndOffset1").value = oRange1.endOffset;
        document.getElementById("txtCommonAncestor1").value =
        oRange1.commonAncestorContainer.tagName;

        document.getElementById("txtStartContainer2").value =
        oRange2.startContainer.tagName;
        document.getElementById("txtStartOffset2").value =
        oRange2.startOffset;
        document.getElementById("txtEndContainer2").value =
        oRange2.endContainer.tagName;
        document.getElementById("txtEndOffset2").value = oRange2.endOffset;
        document.getElementById("txtCommonAncestor2").value =
        oRange2.commonAncestorContainer.tagName;
      }
    </script>
  </head>
  <body><p id="p1"><b>Hello</b> World</p>
    <input type="button" value="Use Ranges" onclick="useRanges()" />
    <table border="0">
      <tr>
        <td>
          <fieldset>
            <legend>oRange1</legend>
            Start Container: <input type="text" id="txtStartContainer1"

```

```

/><br />
    Start Offset: <input type="text" id="txtStartOffset1" /><br />
    End Container: <input type="text" id="txtEndContainer1" /><br />
/>
    End Offset: <input type="text" id="txtEndOffset1" /><br />
    Common Ancestor: <input type="text" id="txtCommonAncestor1" /><br />
/><br />
        </fieldset>
    </td>
    <td>
        <fieldset>
            <legend>oRange2</legend>
            Start Container: <input type="text" id="txtStartContainer2" /><br />
            Start Offset: <input type="text" id="txtStartOffset2" /><br />
            End Container: <input type="text" id="txtEndContainer2" /><br />
/>
            End Offset: <input type="text" id="txtEndOffset2" /><br />
            Common Ancestor: <input type="text" id="txtCommonAncestor2" /><br />
/><br />
        </fieldset>
    </td>
</tr>
</table>
</body>
</html>

```

图 10-2 显示了这个例子在 DOM 兼容的浏览器，如 Mozilla 中的运行结果。

320

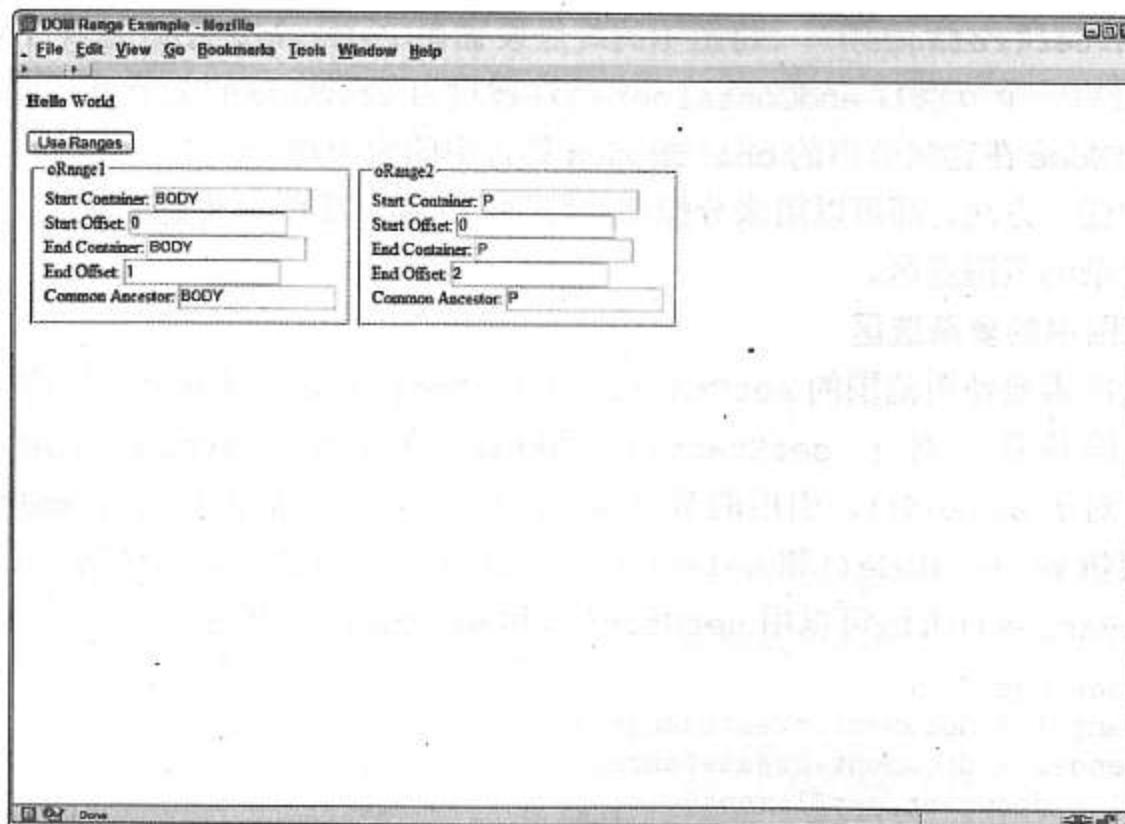


图 10-2

可以看到，oRange1 的 startContainer、endContainer 和 commonAncestorContainer

等于`<body>`元素，因为`<p>`元素完全包含在其中。同时，`startOffset` 等于 0，因为`<p>`是`<body>`的第一个子节点，`endOffset` 等于 1，意思是范围结束于第二个子节点（它的索引是 1）之前。

再看由`selectNodeContents()`收集的`oRange2`的信息，`startContainer`、`endContainer` 和`commonAncestorContainer` 都等于`<p>`元素本身，因为我们选择的是它的子节点。`startOffset` 等于 0，因为选区开始于`<p>`的第一个子节点。`endOffset` 等于 2，因为`<p>`有两个子节点：``和文本节点`World`。

还有一些方法可以直接设置这些特性：

- `setStartBefore(refNode)`——将范围的起点设在`refNode`之前（这样`refNode`就是选区的第一个节点）。同时，`startContainer` 特性被设置成`refNode`的父节点，`startOffset` 特性被设置为`refNode`在其父节点的`childNodes`集合中的索引。
- `setStartAfter(refNode)`——将范围的起点设在`refNode`之后（这样，`refNode`就不是选区的一部分，而它的下一个兄弟节点则是选区的第一个节点）。`startContainer` 特性被设为`refNode`的父节点，同时`startOffset` 特性被设为`refNode`在它的父节点的`childNodes`集合中的索引加一。
- `setEndBefore(refNode)`——将范围的终点设在`refNode`之前（这样，`refNode`就不是选区的一部分，而它的前一个兄弟节点则是选区的最后一个节点）。`endContainer` 特性被设置为`refNode`的父节点，并将`endOffset` 特性设置为`refNode`在父节点的`childNodes`集合中的索引。
- `setEndAfter(refNode)`——将范围的终点设置在`refNode`之后（这样`refNode`将成为选区的最后一个节点）。`endContainer` 特性被设为`refNode`的父节点，而`endOffset` 被设为`refNode`在它父节点的`childNodes`集合中的索引加一。

321

使用其中的任一方法，都可以用来分配那些特性。然而，还可用其他方法来更直接地分配特性值以创建更复杂的范围选区。

2. DOM 范围中的复杂选区

创建复杂选区需要使用范围的`setStart()`和`setEnd()`方法。这两个方法均接受两个参数：节点的引用和偏移量。对于`setStart()`，引用的节点为`startContainer`，偏移量为`startOffset`；对于`setEnd()`，引用的节点为`endContainer`，而偏移量为`endOffset`。

可以直接模仿`selectNode()`和`selectNodeContents()`方法来使用这两个方法。例如，前面例子中的`useRanges()`方法可以用`setStart()`和`setEnd()`来重写。

```
function useRanges() {
    var oRange1 = document.createRange();
    var oRange2 = document.createRange();
    var oP1 = document.getElementById("p1");

    var iP1Index = -1;
    for (var i=0; i < oP1.parentNode.childNodes.length; i++) {
        if (oP1.parentNode.childNodes[i] == oP1) {
```

```

    iP1Index = i;
    break;
}

oRange1.setStart(oP1.parentNode, iP1Index);
oRange1.setEnd(oP1.parentNode, iP1Index + 1);
oRange2.setStart(oP1, 0);
oRange2.setEnd(oP1, oP1.childNodes.length);

//textbox assignments here
}

```

注意，为了选择节点（使用 `oRange1`），必须首先判断给定节点（`oP1`）在其父节点的 `childNodes` 集合中的索引；为选择节点内容（使用 `oRange2`），则不需任何计算。但是现在已经知道了一种更加简单的选择节点和节点内容的方法；其实这个方法的真正能力在于只选择部分节点。

回忆本节中所讲的第一个例子，在 HTML 代码 `<p id="p1">Hello World</p>` 中，从 Hello 中选择 llo 和从 World 中选择 wo。通过使用 `setStart()` 和 `setEnd()`，可以很容易地完成这个任务。

322

第一步是，用一般的 DOM 方法获取包含 Hello 和 World 的文本节点：

```

var oP1 = document.getElementById("p1");
var oHello = oP1.firstChild.firstChild;
var oWorld = oP1.lastChild;

```

Hello 文本节点实际上是 `<p/>` 的孙子节点，因为它上面还有一个 ``，所以可以用 `oP1.firstChild` 来获取 ``，用 `oP1.firstChild.firstChild` 来获取文本节点。World 文本节点是 `<p/>` 的第二个（也是最后一个）子节点，所以可以用 `oP1.lastChild` 来获取它。

下面，创建范围对象，并设置合适的偏移量：

```

var oP1 = document.getElementById("p1");
var oHello = oP1.firstChild.firstChild;
var oWorld = oP1.lastChild;
var oRange = document.createRange();

oRange.setStart(oHello, 2);
oRange.setEnd(oWorld, 3);

```

对于 `setStart()`，偏移量是 2，因为第一个 l 在 Hello 中是在位置 2（从 H 位置 0 开始）上。对于 `setEnd()`，偏移量是 3，表示第一个字符不应被选中，这个字符是 r 在位置 3 上（其实在位置 0 上有一个空格，见图 10-3）

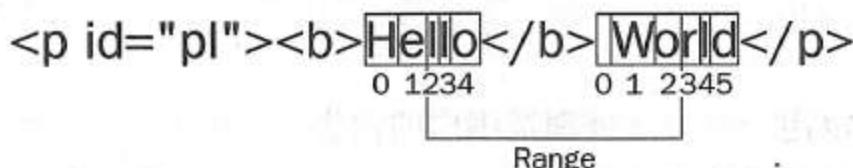


图 10-3

因为 `oHello` 和 `oWorld` 都是文本节点，所以它们将变成范围的 `startContainer` 和 `endContainer`，这样 `startOffset` 和 `endOffset` 将查找包含在其中的文本而不是子节点，这发生在传入元素时。`commonAncestorContainer` 是`<p>`元素，是同时包含两个文本节点的第一个祖先节点。

当然，除非可以与选区进行交互，否则仅仅选择文档部分还不是十分有用。

在 mozilla 的 DOM 范围对象的实现中有一个错误 (bug #135928)，当对同一个文本节点尝试使用 `setStart()` 和 `setEnd()` 时就会出现这个错误。这个错误在新的 Mozilla 发行版中已经修复。

323

3. 与DOM范围对象的内容进行交互

当创建对象时，内部将创建文档碎片节点，在选区中的所有节点都将被附加其上。然而，在附加之前，范围对象必须保证所选的内容的格式是正确的。

前面已经学习了选择从 Hello 的第一个 l 开始到 World 中的 o 结束区域，其中包括``结束标签（见图 10-4）。而使用本书前面描述过的普通的 DOM 方法是不可能实现的。

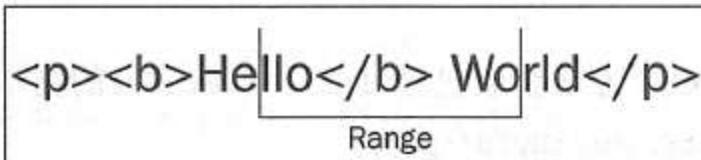


图 10-4

范围可以绕开这个障碍是因为它能识别缺失起始和结束标记。在前面的例子中，范围计算出选区中缺少``起始标签，所以范围对象在后台动态地添加了它，并提供了``结束标签来关闭 He，这样 DOM 被转换成下面这样：

```
<p><b>He</b><b>llo</b> World</p>
```

包含在范围中的文档碎片如图 10-5 所示：

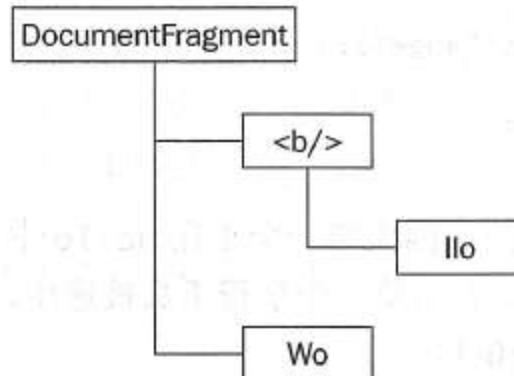


图 10-5

创建文档碎片后，就可以用一些方法处理范围内的内容。

第一个，最易理解和使用的方法是：`deleteContents()`。这个方法会从文档中将范围内的内容全部删除。前面的例子里，对范围调用 `deleteContents()` 会使以下 HTML 留在页面中：

```
<p><b>He</b>rld</p>
```

因为整个文档碎片都已被删除，所以范围对象必须将缺失的标签放入文档中，以保持文档格式的正确。

`extractContents()`与`deleteContents()`方法相似。它也可以从文档中删除选区范围，并将范围内的文档碎片作为函数返回值返回。这样就可将范围的内容插入到其他位置去：

```
var oP1 = document.getElementById("p1");
var oHello = oP1.firstChild.firstChild;
var oWorld = oP1.lastChild;
var oRange = document.createRange();

oRange.setStart(oHello, 2);
oRange.setEnd(oWorld, 3);
var oFragment = oRange.extractContents();

document.body.appendChild(oFragment);
```

这个例子中，碎片被取出并加到文档中<body>元素的结尾（记住，当文档碎片传到`appendChild()`中时，只有碎片的子节点会被插入，而碎片本身不会被插入）。这个例子中可以看到代码Herld在页面的顶部，而llo Wo在页面的底部。

还有种选择是将碎片留在原地，但是创建可以用来插入到文档的任何其他位置的副本，这可以用`cloneContents()`方法实现：

```
var oP1 = document.getElementById("p1");
var oHello = oP1.firstChild.firstChild;
var oWorld = oP1.lastChild;
var oRange = document.createRange();

oRange.setStart(oHello, 2);
oRange.setEnd(oWorld, 3);
var oFragment = oRange.cloneContents();

document.body.appendChild(oFragment);
```

这个方法类似于`extractContents()`，它们均返回范围中的文档碎片。代码的运行结果是llo Wo被添加到页面的底部，而原来的HTML留在原地。

创建文档碎片和对选区范围的更改，在调用这些方法之前都不会发生。在此之前，原来的HTML代码都将原封不动。

4. 插入DOM范围的内容

前面的三个方法以不同的方式删除范围中的信息。同时，我们还可以用其他方法向范围内添加内容。

`insertNode()`方法用来在选区的开头插入节点。如果要将以下HTML代码插入到上节例子中定义的范围内：

```
<span style="color: red">Inserted text</span>
```

可以用以下代码来完成：

```
var oP1 = document.getElementById("p1");
var oHello = oP1.firstChild.firstChild;
var oWorld = oP1.lastChild;
var oRange = document.createRange();

var oSpan = document.createElement("span");
oSpan.style.color = "red";
oSpan.appendChild(document.createTextNode("Inserted text"));

oRange.setStart(oHello, 2);
oRange.setEnd(oWorld, 3);
oRange.insertNode(oSpan);
```

运行这段 JavaScript 可以产生如下 HTML 代码：

```
<p id="p1"><b>He<span style="color: red">Inserted text</span>llo</b> World</p>
```

注意``是直接插在`Hello`中`llo`之前的，这是选区范围的开头部分。还要注意没有对原来的 HTML 添加或删除``元素，因为没有用到在上节中所讲的任何方法。你可以用这个技术来插入很有用的信息，比如在链接旁边加上用于在新窗口中打开链接的图像。

除了可以将内容插到范围，还可用`surroundContents()`方法插入包围范围的内容。这个方法接受一个参数（要包围范围的内容的节点）。在后台会执行以下几步：

- (1) 抽取出范围中的内容（类似于`extractContents()`）。
- (2) 将给定节点插入到原来范围所在的位置。
- (3) 将文档碎片的内容添加到给定节点中。

这类功能在网页上也很有用，可以用来突出显示页面上的特定单词，如下：

```
var oP1 = document.getElementById("p1");
var oHello = oP1.firstChild.firstChild;
var oWorld = oP1.lastChild;
var oRange = document.createRange();

var oSpan = document.createElement("span");
oSpan.style.backgroundColor = "yellow";

oRange.setStart(oHello, 2);
oRange.setEnd(oWorld, 3);
oRange.surroundContents(oSpan);
```

326

前面的代码用黄色的背景突出显示范围选区。

5. 折叠DOM范围

要清空范围（也就是，不选择文档的任何部分），可以折叠它。折叠范围类似文本框的一种行为。当文本框中有文字时，可以用鼠标突出全部文字。但是，只要再点击一下鼠标左键，选区就消失了，光标将停留在两个字母之间。折叠范围，是将范围的位置设置在文档中间，要么是原来选区的开头，要么是结尾。图 10-6 描述在范围折叠时发生的情况。

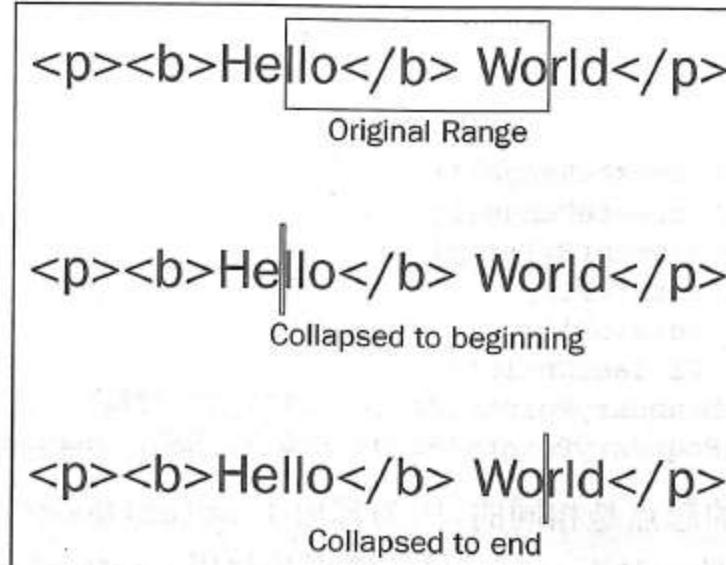


图 10-6

可以用 `collapse()` 方法来折叠范围，这个方法接受一个参数：用来表示是折叠到范围的哪一端的 Boolean 值。如果参数是 `true`，范围就折叠到开头；如果是 `false`，范围就折叠到末尾。要判断范围是否已被折叠，可以用 `collapsed` 特性：

```
oRange.collapse(true);           //collapse to the starting point
alert(oRange.collapsed);        //outputs "true"
```

如果不确定范围中的两个节点是否相邻，则测试范围是否已被折叠是很有用的。例如，考虑这段 HTML 代码：

DO <p id="p1">Paragraph 1</p><p id="p2">Paragraph 2</p>

如果知道这段代码的确切组成（因为可能是自动生成的），可以这样创建范围：

```
var oP1 = document.getElementById("p1");
var oP2 = document.getElementById("p2");
var oRange = document.createRange();
oRange.setStartAfter(oP1);
oRange.setStartBefore(oP2);
alert(oRange.collapsed);      //outputs "true"
```

这个例子中，被创建的范围是被折叠的，因为在 p1 的末尾到 p2 的开头之间无任何内容。

327

6. 比较DOM范围

如果有几个范围，则可以使用 `compareBoundaryPoints()` 方法来判断范围是否有相同的边界（开头或者结尾）。这个方法接受两个参数：要进行比较的范围及如何进行比较，后者是个常量值：

- `START_START(0)`——比较两个范围的起点。
- `START_TO_END(1)`——比较第一个范围的起点和第二个范围的终点。
- `END_TO_END(2)`——比较两个范围的终点。
- `END_TO_START(3)`——比较第一个范围的终点和第二个范围的起点。

如果第一个范围的被测点在第二个范围的被测点之前，`compareBoundaryPoints()` 方法返回 -1；如果两个点一样，返回 0；如果第一个范围的被测点在第二个范围的被测点的后面，

则返回 1。

例如：

```
var oRange1 = document.createRange();
var oRange2 = document.createRange();
var oP1 = document.getElementById("p1");
oRange1.selectNodeContents(oP1);
oRange2.selectNodeContents(oP1);
oRange2.setEndBefore(oP1.lastChild);
alert(oRange1.compareBoundaryPoints(Range.START_TO_START, oRange2)); //outputs 0
alert(oRange1.compareBoundaryPoints(Range.END_TO_END, oRange2)); //outputs 1;
```

这段代码中，两个范围的起点是相同的，因为都用了 `selectNodeContents()` 方法的默认值；因此，这个方法返回 0。但是，对于 `oRange2`，终点是使用 `setEndBefore()` 设置的，使得它在 `oRange1` 的终点之前（见图 10-7），所以方法返回 1。

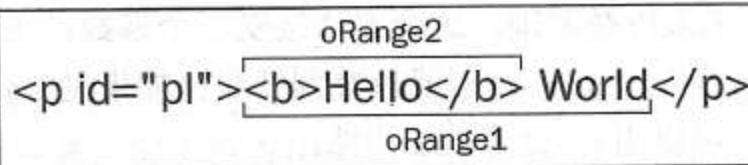


图 10-7

7. 复制DOM范围

如果需要，可通过 `cloneRange()` 方法来创建范围的副本。它可以创建与被调用的范围对象完全一致的副本：

```
var oNewRange = oRange.cloneRange();
```

328 新的范围包含的所有特性与原范围的特性均相同，并且能够丝毫不会影响原范围地修改它。

8. 清理

当用完范围后，最好用 `detach()` 方法释放系统资源。这不是必需的，因为不再被引用的范围最后总是会被垃圾收集器清理掉。然而，如果范围已用完且不再需要了，那么调用 `detach()` 方法可以保证它不会一直占用不必要的内存空间：

```
oRange.detach();
```

10.4.2 IE 中的范围

IE 处理范围的方法并不标准，虽然如此，只要理解了两者之间的区别，还是可以发挥它的功能。

首先，IE 中的范围称为文本范围（text range），它们主要用来处理文本（并不是针对 DOM 节点）。要创建范围，要调用 `<body/>`、`<button/>`、`<input/>` 或者 `<textarea/>` 元素上的 `createTextRange()` 方法（而不是 `document` 对象）：

```
var oRange = document.body.createTextRange();
```

用这种方法创建的范围可以在页面上的任何位置上使用（如上面所说，其他元素创建的范围

只允许范围在其内部使用)。

1. IE 范围中的简单选区

选择页面的某个区域的最简单的方法是用范围的 `findText()` 方法。这个方法可找到给定文本字符串的第一个实例并用范围包含它。再来考虑以下的 HTML 代码:

```
<p id="p1"><b>Hello</b> World</p>
```

要选择 Hello, 可以用以下代码:

```
var oRange = document.body.createTextRange();
var bFound = oRange.findText("Hello");
```

代码执行后, 文本 Hello 就被包含在范围之内。可用范围的 `text` (可以返回包含在范围之内的文本) 特性来检测它, 或者也可以检测 `findText()` 的返回值, 如果为 `true`, 表示找到文本:

```
alert(bFound);
alert(oRange.text);
```

要在文档中移动范围, 可用 `findText()` 方法的第二个参数, 这是个表示继续搜索的方向的数字: 负数表示搜索向回查找, 正数表示搜索继续向前。所以, 要找到文档中 Hello 的前两个实例, 可以用这段代码:

```
var bFound = oRange.findText("Hello");
var bFoundAgain = oRange.findText("Hello", 1);
```

与 DOM 的 `selectNode()` 方法最相近的是 IE 的 `moveToElementText()` 方法, 它可接受 DOM 元素作为参数, 并选取元素的所有文本, 包括 HTML 标签:

```
var oRange = document.body.createTextRange();
var oP1 = document.getElementById("p1");
oRange.moveToElementText(oP1);
```

要测试这段代码是否正常运行, 可使用 `htmlText` 特性, 它将返回包含在范围中的所有 HTML 代码:

```
alert(oRange.htmlText);
```

尽管有一个 `parentElement()` 方法的行为与 DOM 的 `commonAncestorContainer` 特性相同, 但 IE 中的范围没有其他会随着选区的变化而动态更新的特性:

```
var oCommonAncestor = oRange.parentElement();
```

2. IE 范围中的复杂选区

IE 中的复杂范围选区的复杂的部分是必须先使用前面的简单选区方法。当范围在相对较为正确的位置上后, 即可以使用 `move()`、`moveStart()`、`moveEnd()` 和 `expand()` 来进一步给范围定位。

这些方法都接受两个参数: 移动的单位和要移动的单位的个数。移动的单位可以是下面字符串值之一:

- "character"——移动一个字符。

- "word"——移动一个单词（非空字符的序列）。
- "sentence"——移动一个句子（一系列以句号、问号、感叹号结尾的字符）。
- "textedit"——移动到当前选区的开头或者是结尾。

`moveStart()`方法将范围的起点移动给定数目的单位，`moveEnd()`方法将范围的终点移动给定数目的单位：

```
oRange.moveStart("word", 2);           //move the start point by two words
oRange.moveEnd("character", 1);        //move the ending point by two words
```

也可以用 `expand()`方法使范围规范化。它可以确保将被部分选中的单位变成完全选中。例如，如果只选中了某个单词的中间两个字母，则可调用 `expand('word')` 来确保整个单词都被包含在范围内。

`move()`方法首先折叠范围（令起点和终点一样），然后将范围移动指定数目的单位：

```
oRange.move("character", 5);          //move over five characters
```

使用 `move()`方法后，必须用 `moveStart()`或者 `moveEnd()`来建立一个选区。

3. 与IE的范围内容进行交互

与 IE 的范围内容进行的交互是通过 `text` 特性或者是 `pasteHTML()` 方法来完成的。

`text` 特性，在前面用来获取范围的文本内容，也可用来设置范围的文本内容。例如：

```
var oRange = document.body.createTextRange();
oRange.findText("Hello");
oRange.text = "Howdy";
```

如果对前面的 Hello World 运行这段代码，则执行结果如下：

```
<p id="p1"><b>Howdy</b> World</p>
```

注意，当设置 `text` 特性时，所有 HTML 标签均保持原样。如果想插入更多的内容，而不仅是文本内容，则可用 `pasteHTML()` 方法插入 HTML 代码。例如：

```
var oRange = document.body.createTextRange();
oRange.findText("Hello");
oRange.pasteHTML("<em>Howdy</em>");
```

如果运行这段代码，结果如下：

```
<p id="p1"><b><em>Howdy</em></b> World</p>
```

当范围包含 HTML 代码时，不推荐使用 `pasteHTML()`，因为这会造成不可预料的后果，往往致使 HTML 格式有误。

4. 折叠IE的范围

IE 中的范围有个 `collapse()`方法，与 DOM 方法的行为完全一样：传入 `true` 则折叠范围到起点，`false` 则折叠范围到终点。

```
oRange.collapse(true);
```



不幸的是，没有对应的 collapsed 特性来判断范围是否已经被折叠。相反，必须用 boundingWidth 特性，它将返回范围的宽度（以像素为单位）。如果 boundingWidth 值为 0，则范围被折叠了：

```
var bIsCollapsed = (oRange.boundingWidth == 0);
```

还有 boundingHeight、boundingLeft 和 boundingTop 特性也会给出一些范围的位置信息，尽管它们不如 boundingWidth 有用。

331

5. 比较IE的范围

IE 中的范围有个与 DOM 范围的 compareBoundaryPoints() 方法类似的方法 compareEndPoints()。这个方法接受两个参数，比较的类型和要比较的范围。与 DOM 的实现不同，IE 中的比较类型是这些值："StartToStart"、"StartToEnd"、"EndToEnd" 和 "EndToStart"。这些比较类型对应 DOM 中的比较类型。

返回值也和 DOM 的类似，如果第一个范围的边界在第二个范围的边界前面，那么 compareEndPoints() 返回 -1，如果在后面返回 1，如果相同返回 0。再来考虑前面例子中用到的 Hello World 的 HTML 代码。下面的代码将创建两个范围：一个选择"Hello World"（包含 **>** 标签），另一个选择"Hello"（也包含 **>** 标签，见图 10-7）：

```
var oRange1 = document.body.createTextRange();
var oRange2 = document.body.createTextRange();
oRange1.findText("Hello World");
oRange2.findText("Hello");
alert(oRange1.compareEndPoints("StartToStart", oRange2)); //outputs 0
alert(oRange1.compareEndPoints("EndToEnd", oRange2)); //outputs 1;
```

类似于 10.4.1 第 6 小节中的例子，两个范围的起点相同，所以 compareEndPoints() 返回 0；oRange1 的终点在 oRange2 的终点后面，所以 compareEndPoints() 返回 1。

IE 还有另外两个比较范围的方法：isEqual()，判断两个范围是否完全一样；inRange()，判断一个范围是否出现在另一个范围的内部：

```
var oRange1 = document.body.createTextRange();
var oRange2 = document.body.createTextRange();
oRange1.findText("Hello World");
oRange2.findText("Hello");
alert("oRange1.isEqual(oRange2): " + oRange1.isEqual(oRange2)); //outputs "false"
alert("oRange1.inRange(oRange2): " + oRange1.inRange(oRange2)); //outputs "true"
```

这个例子用前面例子中的范围来说明这两个方法。现已知道这两个范围因终点不同而不同；如果相等，范围的起点和终点必须都相同。所以第一个警告框显示 "false"。然而，oRange2 其实在 oRange1 的内部，因为 oRange2 的终点在 oRange1 的前面，起点却在 oRange1 的起点后面。因此，第二个警告框显示 "true"，通告 oRange2 在 oRange1 里面。

6. 复制IE的范围

类似 DOM，在 IE 中可用 duplicate() 方法来创建给定范围的精确副本：

```
var oNewRange = oRange.duplicate();
```

332

所有原来范围的特性都被复制到新创建的范围中。

10.4.3 范围在实际中的应用

DOM 和 IE 的范围实现上的不同，造成创建跨浏览器的解决方案十分困难，这也许就是很多开发人员都不愿使用范围的原因。当评估自己的用法时，了解目标人群是十分重要，他们会使用哪种浏览器，以及是否能够不使用范围而达到相同的效果。

范围可以在 Web 网页上提供十分强大的功能。根据一系列搜索词汇，用范围来突出页面上特定单词的页面，可以使用户很方便地找到他们在查找的单词。另一种用途，在广告商中经常使用，将特定单词变成链接（例如，将单词 computer 变成链接到计算机制造商的网站或将单词 JavaScript 指向关于它的说明的页面）。

10.5 小结

本章介绍了一些新的操作文档的 DOM 树的方法。

首先，学会了如何影响网页上元素的 CSS 样式。讨论了一些例子，包括鼠标悬停特效和自定义鼠标提示。还学习了如何访问元素的样式定义和 CSS 规则，以及将它们与最终样式进行比较。

接下来的一节介绍了 innerText、innerHTML、outerText 和 outerHTML。学习了用 innerText 改变 DOM 元素的文本内容以及用 innerHTML 来改变 DOM 元素的 HTML 内容。类似的，还学习了用 outerText 和 outerHTML 来完全替换 DOM 元素（对应地使用纯文本或者 HTML 代码）。

333

334

最后，学习了范围的有关知识。还对 DOM 范围和 IE 中范围的区别进行了讨论，并给出一些例子来展示两者异同。

创建表单元素是为了满足用户向服务器发送数据的需求。Web 表单使用 HTML 的 `<form/>`、`<input/>`、`<select/>` 和 `<textarea/>` 等元素。用这些元素，浏览器可以渲染文本框、组合框以及其他输入控件，来实现客户端与服务器端之间的通信。

Web 已经发展得很快了，而 Web 表单基本上没有变化。尽管一种新的 XForm 的标准正在兴起，但还没有任何浏览器去进一步支持它，所以现在的 Web 表单要依赖 JavaScript 来增加其内置的行为。

在这一章中，你将学会用 JavaScript 来扩展普通表单的行为和可用性，以便满足用户所需求的功能。

11.1 表单基础

HTML 表单是通过 `<form/>` 元素来定义的，它有以下特性：

- `method`——表示浏览器发送 GET 请求或是发送 POST 请求。
- `action`——表示表单所要提交到的地址 URL。
- `enctype`——当向服务器端发送数据时，数据应该使用的编码方法。默认是 `application/x-www-url-encoded`，不过，如果要上传文件，可以设置成 `multipart/form-data`。
- `accept`——当上传文件时，列出服务器能正确处理的 mime 类型。
- `accept-charset`——当提交数据时，列出服务器接受的字符编码。

表单可以包含任意数目的输入元素：

- `<input/>`——主要的 HTML 输入元素。通过 `type` 特性来判断是哪种输入控件：
 - "text"——单行文本框；
 - "radio"——单选按钮；
 - "checkbox"——复选框；
 - "file"——文件上传文本框；
 - "password"——密码输入框（输入时不会显示字符）；
 - "button"——可用来产生自定义动作的一般按钮；

- "submit"——提交按钮，其唯一的目的是提交表单；
 - "reset"——重置按钮，其唯一的目的是将表单中的所有字段都重置为默认值；
 - "hidden"——不会出现在屏幕上的输入字段；
 - "image"——图像，与提交按钮功能一样。
- <select/>——渲染组合框或者下拉列表框，里面的值由<option/>元素定义。
- <textarea/>——渲染多行文本框，尺寸由 rows 和 cols 特性来确定。

下面是使用以上这些输入元素的简单表单：

```

<html>
  <head>
    <title>Sample Form</title>
  </head>
  <body>
    <form method="post" action="handlepost.jsp">
      <!-- regular textbox -->
      <label for="txtName">Name:</label><br />
      <input type="text" id="txtName" name="txtName" /><br />

      <!-- password textbox -->
      <label for="txtPassword">Password:</label><br />
      <input type="password" id="txtPassword" name="txtPassword" /><br />

      <!-- age combobox (drop-down) -->
      <label for="selAge">Age:</label><br />
      <select name="selAge" id="selAge">
        <option>18-21</option>
        <option>22-25</option>
        <option>26-29</option>
        <option>30-35</option>
        <option>Over 35</option>
      </select><br />
      <!-- multiline textbox -->
      <label for="txtComments">Comments:</label><br />
      <textarea rows="10" cols="50" id="txtComments"
      name="txtComments"></textarea><br />

      <!-- submit button -->
      <input type="submit" value="Submit Form" />

    </form>
  </body>
</html>

```

336

这个例子中，描述了五个字段：普通文本框、密码输入框、组合框、多行文本框和提交按钮。注意除提交按钮外，每个字段前都有个<label/>元素。这个元素用来在后台将标签绑定到特定的表单字段。这个功能对于那些因视力缺陷而使用屏幕阅读器的用户来说很有用。for 特性指定它绑定的表单的 ID。正因如此，每个表单字段都应包含 id 和 name 特性，且值相等（name 用于将数据提交到服务器，而 id 用于在客户端确定元素）。

每种表单字段都可使用 JavaScript 进行操作。`<form/>`元素本身也可使用 JavaScript 来控制，以便进一步控制数据的传输。

11.2 对`<form/>`元素进行脚本编写

JavaScript 使用`<form/>`元素的方式与使用其他 HTML 元素的方式有所不同。不仅可用核心的 DOM 方法来访问表单；还可用其他方法来访问`<form/>`元素及其中的表单字段。这一节涵盖了必需的基本知识。

11.2.1 获取表单的引用

开始对表单进行编程前，必须先获取`<form/>`元素的引用。有多种方法可以用来完成这一操作。

首先，可使用典型的 DOM 树中的定位元素的方法 `getElementById()`，只需传入表单的 ID：

```
var oForm = document.getElementById("form1");
```

另外，还可用 `document` 的 `form` 集合，并通过表单在 `form` 集合中的位置或者表单的 `name` 特性来进行引用：

```
var oForm = document.forms[0];           //get the first form
var oOtherForm = document.forms["formZ"]; //get the form whose name is "formZ"
```

使用这些方法来获取某个表单的引用都是可行的（至少它们返回的都是相同的内容）。

337

11.2.2 访问表单字段

每个表单字段，不论它是按钮、文本框或者其他的内容，均包含在表单的 `elements` 集合中。可以用它们的 `name` 特性或是它们在集合中的位置来访问不同的字段：

```
var oFirstField = oForm.elements[0]; //get the first form field
var oTextbox1 = oForm.elements["textbox1"]; //get the field with the name "textbox1"
```

还有一种通过名字来直接访问字段的一种方法，每个表单字段都是表单的特性，可以直接通过它的 `name` 来访问：

```
var oTextbox1 = oForm.textbox1; //get the field with the name "textbox1"
```

在名字中有空格，则可以使用方括号标记：

```
var oTextbox1 = oForm["text box 1"]; //get the field with the name "text box 1"
```

当然，也可以用 `document.getElementById()` 和表单字段的 `ID` 来直接获取这个元素。本节中讨论的方法在需要迭代单个表单的所有字段时十分有用。

11.2.3 表单字段的共性

所有的表单字段（除了隐藏字段）都包含同样的特性、方法和事件：

- `disabled` 特性可用来获取或设置表单控件是否被禁用（被禁用的控件不允许用户输入，如果控件被禁用也会在外观上反应出来）。
- `form` 特性用来指向字段所在的表单。
- `blur()` 方法可以使表单字段失去焦点（将焦点移动到别处）。
- `focus()` 方法会让表单字段获取焦点（控件被选中以便进行键盘交互）。
- 当字段失去焦点时，发生 `blur` 事件，执行 `onblur` 事件处理函数。
- 当字段获取焦点时，发生 `focus` 事件，执行 `onfocus` 事件处理函数。

例如：

```
var oField1 = oForm.elements[0];
var oField2 = oForm.elements[1];

//set the first field to be disabled
oField1.disabled = true;

//set the focus to the second field
oField2.focus();

//is the form property equal to oForm?
alert(oField1.form == oForm); //outputs "true"
```

338

当需要用到高级功能时，这些特性、方法和事件就可以派上用场了。比如，当要把焦点移动到第一个字段上时。

隐藏字段只支持 `form` 特性，其他没有同样的方法或者事件。

11.2.4 聚焦于第一个字段

当页面上显示了一个表单后，通常焦点并不在第一个控件上。只需用一个通常的脚本即可改变这种情况，而且可用在任何表单页面上。

很多开发人员仅将下面一段代码放到页面的 `.onload` 事件处理函数中：

```
document.forms[0].elements[0].focus();
```

在大部分的情况下它都可以使用，但是考虑一下，如果表单的某个元素是隐藏字段，这个字段是不支持 `focus()` 方法的。这种情况下，就会出现 JavaScript 错误。这里的关键是将焦点设在第一个可见的表单字段上，可以为此写个简短的方法。

本章中所有属于表单的方法都会放入 `FormUtil` 对象，以便能进行简单的封装：

```
var FormUtil = new Object;
```

`FormUtil` 对象仅用来集中类似的函数；可以将这些方法放在单独的对象中或者是直接放在外面。

将焦点放在第一个字段的方法首先是检验并保证在页面上存在一个表单。只需检测 `document.forms.length`:

```
FormUtil.focusOnFirst = function () {
    if (document.forms.length > 0) {
        //...
    }
};
```

当知道至少存在一个表单后，就开始遍历表单字段直到找到第一个非隐藏的字段。

```
FormUtil.focusOnFirst = function () {
    if (document.forms.length > 0) {
        for (var i=0; i < document.forms[0].elements.length; i++) {
            var oField = document.forms[0].elements[i];
            if (oField.type != "hidden") {
                oField.focus();
                return;
            }
        }
    }
};
```

339

这个方法可以在 `onload` 事件处理函数中调用:

```
<body onload="FormUtil.focusOnFirst()">
```

使用这个方法要小心。在载入很慢的页面中，在页面完全载入完之前用户可能就开始进行输入了。这时如果再把焦点设置到第一个字段上，就会中断用户当前的输入。为处理这个问题，应该先检查是否在第一个字段中已经有值；如果有，则不改变焦点。

11.2.5 提交表单

在普通的 HTML 中，必须使用提交按钮或是扮演提交按钮角色的图像来提交表单：

```
<input type="submit" value="Submit" />      <!-- submit button -->
<input type="image" src="submit.gif" />      <!-- image button -->
```

当用户点击其中一个按钮，无需其他编码，就可以提交表单。如果按回车键，并存在这些按钮，浏览器就会认为是点击了按钮。

可以通过在 `action` 特性中加入警告来检测表单是否已被提交：

```
<form method="post" action="javascript:alert('Submitted')">
```

这将会把表单提交给 JavaScript 函数，然后弹出警告框“Submitted”。这对测试表单提交与否十分有用，因为实际上它不会返回到服务器。

如果想提交表单，又不想使用前面提到的按钮，可使用 `submit()` 方法。`submit()` 方法是 DOM 定义的`<form>`元素的一部分，可在页面的任意位置使用。要用这个方法，首先要通过 `getElementById()` 或者 `document.forms` 集合来获取`<form>`元素的引用。下面三行代码都可

用来引用表单：

```
oForm = document.getElementById("form1");
oForm = document.forms["form1"];
oForm = document.forms[0];
```

在获取表单引用后，即可以直接调用 `submit()` 方法：

340 oForm.submit();

实际上，也可以通过创建一般按钮，并给它分配 `onclick` 事件处理函数来提交表单来模仿提交按钮的行为：

```
<input type="button" value="Submit Form" onclick="document.forms[0].submit()" />
```

在表单提交前，按下提交按钮后，会触发 `submit` 事件，并执行 `onsubmit` 事件处理函数。使用 `onsubmit` 事件处理函数，可以停止表单提交过程，如果需要在提交表单前进行客户端验证，这将十分有用。可以使用本书前面提到的事件方法，取消事件，阻止表单提交：

```
function handleSubmit() {
    var oEvent = EventUtil.getEvent();
    oEvent.preventDefault();
}
```

这个方法可在表单 `onsubmit` 事件处理函数中调用：

```
<form method="post" action="javascript:alert('Submitted')"
onsubmit="handleSubmit()">
```

当尝试使用提交按钮或者图像按钮提交表单时，表单不会被提交。

`onsubmit` 事件处理函数可以用来在提交过程前验证表单，但这仅限于使用提交按钮和图片按钮，如果使用的是 `submit()` 方法，不会触发 `submit` 事件，所以所有的验证过程应该在调用它之前完成。

11.2.6 仅提交一次

Web 表单的一个很大问题是用户在提交表单的时候会很不耐烦。如果点击提交按钮后，表单没有立即消失，用户则常会点好多次。对于这个问题，简单的会创建多次重复的请求，糟糕的情况，可能会多次从信用卡中扣钱。解决方法十分简单：在用户点击提交按钮后，将其禁用。这里介绍实现方法：选用一个普通按钮，而不是提交按钮，然后在用户点击按钮后，将其禁用。将以下代码：

```
<input type="submit" value="Submit" />
```

替换成：

```
<input type="button" value="Submit"
      onclick="this.disabled=true; this.form.submit()" />
```

当点击按钮后，就通过将 `disabled` 特性设为 `true` 来禁用这个按钮。然后，提交表单（注

意代码使用 `this` 关键字来引用按钮，并通过 `form` 特性来引用所属的表单）。这段代码也可封装在一个函数中。

你可能在想为什么不使用提交按钮并用 `onclick` 来禁用它。这是因为按钮其实在表单提交前就已被禁用，这将导致表单不被提交。

11.2.7 重置表单

若要给用户提供将所有表单字段设回默认值的功能，可以使用 HTML Reset 按钮：

```
<input type="reset" value="Reset Values" />
```

类似提交按钮，点击重置按钮时，它并不需要编写任何脚本，浏览器自然知道应该做什么。同样类似于提交按钮，当按下重置按钮后，会触发 `reset` 事件。

```
<form method="post" action="javascript:alert('Submitted') " onreset="alert('I am resetting') ">
```

当然，可用 `onreset` 事件处理函数来取消表单重置。

同时表单也有 `reset()` 方法，可以直接从脚本中重置表单而无需使用重置按钮：

```
<input type="button" value="Reset" onclick="document.forms[0].reset()" />
```

与 `submit()` 不同的是，使用 `reset()` 方法仍会触发 `reset` 事件并执行 `onreset` 事件处理函数。

重置表单在 Web 开发人员中已不再受欢迎，因为常常会有用户不小心点击重置按钮而不是提交按钮（因为提交和重置按钮往往靠在一起）。如果表单第一次载入时，在字段中已包含一些默认信息，那么重置按钮还有些作用，因为可恢复到初始值。但对载入时字段中没有任何信息的表单，最好避免使用重置按钮。

11.3 文本框

在 HTML 中有两种不同类型的文本框：单行的，如`<input type="text" />`和多行的，如`<textarea>`。

`<input />`元素必须将其 `type` 特性设为“`text`”才能显示为文本框。然后使用 `size` 特性指定文本框的宽度（根据可见字符的数目，例如，将 `size` 设置为“`10`”表示一次只能看到 10 个字符）。`value` 特性用来指定文本框中的初始内容，`maxlength` 特性指定在文本框中允许的最大字符数。那么要创建同时可以显示 25 个字符、最多容纳 50 个字符的文本框，可用以下代码：

```
<input type="text" size="25" maxlength="50" value="initial value" />
```

`<textarea>`元素用来渲染多行文本框。可使用 `rows` 和 `cols` 特性指定文本框的大小。`rows` 特性指定以字符为单位文本框的高度，`cols` 指定也以字符为单位文本框的宽度（类似于`<input />` 元素的 `size` 特性）。但与`<input />` 不同的是，`<textarea />` 的初始值必须包含在`<textarea>`

和</textarea>之间，例如：

```
<textarea rows="25" cols="5">initial value</textarea>
```

另外，<textarea/>不能指定允许的最大字符数。

11.3.1 获取/更改文本框的值

尽管<input type="text" />和<textarea />是不同元素，但它们均支持同样的特性来获

343 取包含在文本框内的文本。考虑以下例子：

```
<html>
  <head>
    <title>Retrieving a Textbox Value Example</title>
    <script type="text/javascript">
      function getValues() {
        var oTextbox1 = document.getElementById("txt1");
        var oTextbox2 = document.getElementById("txt2");
        alert("The value of txt1 is \"" + oTextbox1.value + "\"\n" +
              "The value of txt2 is \"" + oTextbox2.value + "\"");
      }
    </script>
  </head>
  <body>
    <input type="text" size="12" id="txt1" /><br />
    <textarea rows="5" cols="25" id="txt2"></textarea><br />
    <input type="button" value="Get Values" onclick="getValues()" />
  </body>
</html>
```

这个例子显示了两个文本框，一个是单行的另一个是多行的，还有一个按钮。当点击按钮时，会出现警告框显示各个文本框中的内容。也可以在两个文本框中输入一些内容，然后点击按钮。

因为 value 特性是个字符串，可以使用任何字符串的特性和方法。例如，可以使用 length 特性来获取文本框中的文本长度：

```
<html>
  <head>
    <title>Retrieving a Textbox Length Example</title>
    <script type="text/javascript">
      function getLengths() {
        var oTextbox1 = document.getElementById("txt1");
        var oTextbox2 = document.getElementById("txt2");
        alert("The length of txt1 is " + oTextbox1.value.length + "\n" +
              "The length of txt2 is " + oTextbox2.value.length);
      }
    </script>
  </head>
  <body>
    <input type="text" size="12" id="txt1" /><br />
```

```

<textarea rows="5" cols="25" id="txt2"></textarea><br />
<input type="button" value="Get Lengths" onclick="getLengths()" />
</body>
</html>

```

这个例子中使用 `value` 的 `length` 特性来判断每个文本框中的字符数。

这个 `value` 特性也可用于给文本框设置新内容：

```

<html>
  <head>
    <title>Changing a Textbox Value Example</title>
    <script type="text/javascript">
      function setValues() {
        var oTextbox1 = document.getElementById("txt1");
        var oTextbox2 = document.getElementById("txt2");
        oTextbox1.value = "first textbox";
        oTextbox2.value = "second textbox";
      }
    </script>
  </head>
  <body>
    <input type="text" size="12" id="txt1" /><br />
    <textarea rows="5" cols="25" id="txt2"></textarea><br />
    <input type="button" value="Set Values" onclick="setValues()" />
  </body>
</html>

```

这个例子中，点击按钮可将第一个文本框设置为 "first textbox"，将第二个文本框设置为 "second textbox"。

11.3.2 选择文本

两种类型的文本框都支持 `select()` 方法，这个方法可选中文本框中的所有文本。为了达到这个功能，文本框必须已经获得焦点。为保证文本框已获取焦点，必须在调用 `select()` 前调用另外一个方法 `focus()`。（并不是所有的浏览器都要求这样，但安全起见，最好每次都先调用 `focus()`。）例如：

```

<html>
  <head>
    <title>Select Text Example</title>
    <script type="text/javascript">
      function selectText() {
        var oTextbox1 = document.getElementById("txt1");
        oTextbox1.focus();
        oTextbox1.select();
      }
    </script>
  </head>
  <body>
    <input type="text" size="12" id="txt1" value="initial value" /><br />

```

```

<input type="button" value="Select Text" onclick="selectText()" />
</body>
</html>

```

这个例子显示一个文本框和一个按钮。当点击按钮时，文本框中的文本全被选中。

11.3.3 文本框事件

两种类型的文本框都支持以前提到的表单字段事件 blur 和 focus，同时还支持事件：change 和 select。

- change——当用户更改内容后文本框失去焦点时发生（如果是通过 value 特性来更改内容，则不会触发）。
- select——当一个或多个字符被选中时发生，无论是手工选中还是用 select() 方法。

注意 change 事件和 blur 事件的区别。只要文本框失去焦点，就触发 blur 事件，而 change 事件只有当文本框内的文本发生改变后，失去焦点时才触发。如果文本不变，但文本框失去焦点，那么只有 blur 事件被触发；如果文本发生了变化，则先触发 change 事件，然后触发 blur 事件。最好亲自尝试一下以获得更加深刻的认识：

```
<input type="text" name="textbox1" value=""  
onblur="alert('Blur')" onchange="alert('Change')"/>
```

另外，select 事件和文本框的焦点无关。当用户选中一个或多个字符时或调用 select() 方法时触发这个事件。可以用下面的代码进行试验：

```
<input type="text" name="textbox1" value="" onselect="alert('Select')"/>
```

11.3.4 自动选择文本

当用户在传统的桌面应用中输入信息时，常是用户切换到文本框时其中的全部内容即已被选中。这个功能在 HTML 中很容易实现，无论是 input 的还是 textarea 的文本框。只要在控件的 [345] onfocus 事件处理函数中加入代码 "this.select()"。

```
<input type="text" onfocus="this.select()" />  
<textarea onfocus="this.select()"></textarea>
```

这只是个小小的改动，但它为用户提供了极大的可用性。要将这个行为自动添加到页面上所有的文本框中，可以使用下面的函数：

```
FormUtil.setTextboxes = function() {
    var colInputs = document.getElementsByTagName("input");
    var colTextAreas = document.getElementsByTagName("textarea");

    for (var i=0; i < colInputs.length; i++) {
        if (colInputs[i].type == "text" || colInputs[i].type == "password") {
            colInputs[i].onfocus = function () { this.select(); };
        }
    }
}
```

```

for (var i=0; i < colTextAreas.length; i++){
    colTextAreas[i].onfocus = function () { this.select(); };
}
);

```

这个函数首先获取文档中所有的<input>和<textarea>实例。第一个 for 循环迭代所有的<input>标签并寻找文本框和密码字段（其实密码字段也是文本框）。然后函数将包含选择代码的匿名函数加到字段的 onfocus 事件处理函数上（当然，也可以用 EventUtil.addHandler()方法）。第二个 for 循环对页面上所有的 textarea 做同样的 onfocus 赋值。

11.3.5 自动切换到下一个

当某个文本框只允许接受指定数量的字符串时，如果能直接切换到下一个字段，岂不是很好？当输入社会安全码或是产品 ID 数字等，常会用到这个功能。用 JavaScript 也可很容易地模仿这种行为。这个脚本要求文本框设置 maxlength 特性，例如：

```
<input type="text" maxlength="4" />
```

基本思想是判断输入文本框的字符是否已达到最大数量，如果是，则调用下一个字段的 focus() 方法。要实现这个功能还要用到另一个方法：

```

FormUtil.tabForward = function(oTextbox) {
    var oForm = oTextbox.form;
    //make sure the textbox is not the last field in the form
    if (oForm.elements[oForm.elements.length-1] != oTextbox
        && oTextbox.value.length == oTextbox.getAttribute("maxlength")) {
        for (var i=0; i < oForm.elements.length; i++) {
            if (oForm.elements[i] == oTextbox) {
                for(var j=i+1; j < oForm.elements.length; j++) {
                    if (oForm.elements[j].type != "hidden") {
                        oForm.elements[j].focus();
                        return;
                    }
                }
            }
            return;
        }
    }
};

```

346

FormUtil.tabForward() 方法接受一个参数，要检查的文本框。这个方法中，通过使用文本框的 form 特性获得其所属表单的引用。然后，它检查文本框是否为表单中的最后一个字段。如果测试通过，则使用文本框的 maxlength 特性来检测内容是否已达到最大的字符串长度。如果还未达到上限，则直接退出函数，否则，进入第一个循环。

第一个 for 循环唯一的目的是在 form.elements 集合中为文本框定位。定位后，会出现类

似乎于前一节所遇到的问题：如果下一个元素是隐藏字段该怎么办？这时就用到第二个循环。第二个循环迭代剩下的元素直到发现第一个非隐藏字段，然后为其设置焦点，然后通过 `return` 语句退出方法。

这个方法必须在输入每一个字符后调用，所以需要使用 `onkeyup` 事件处理函数（`keyup` 事件在字符被放入文本框后触发，而 `keypress` 事件在之前触发）：

```
<input type="text" maxlength="4" onkeyup="FormUtil.tabForward(this) " />
```

注意使用 `this` 关键字将文本框的引用传给这个方法。假设想让用户输入 U.S. 格式的电话号码（3 个数字-3 个数字-4 个数字）。可以这样创建三个文本框：

```
<input type="text" id="txtAreaCode" maxlength="3"
       onkeyup="FormUtil.tabForward(this) " />
```

```
<input type="text" id="txtExchange" maxlength="3"
       onkeyup="FormUtil.tabForward(this) " />
```

```
<input type="text" id="txtNumber" maxlength="4"
       onkeyup="FormUtil.tabForward(this) " />
```

一旦用户在文本框中输入了一个数字，焦点就移到下一个上，这样用户就不需要用 Tab 按键或鼠标在字段之间来切换。

11.3.6 限制 `textarea` 的字符数

虽然 `<input>` 元素可以很容易地限制允许的字符数量，但 `<textarea>` 元素却不能，因为它没有 `maxlength` 特性。解决方法是用 JavaScript 来模仿 `maxlength` 特性。例如：

347

```
<textarea rows="10" cols="25" maxlength="150"></textarea>
```

这一章中常要添加额外的特性，当使用 XHTML 的严格版本时，如果页面包含未预期的特性，它将会被认为无效。根据对页面的要求，可能需要通过 JavaScript 为 DOM 节点添加特性或者直接为函数传递额外的信息，而不能使用 HTML 特性。

这将是本章第一个专门针对于文本框的方法（前面几节的方法要处理文本框和其他表单元），所以现在需要新的包装对象来封装将要用到的方法：

```
var TextUtil = new Object();
```

这个对象的第一个方法是 `isNotMax()`，当未达到最大字符数量时返回 `true`，否则，返回 `false`。首先，看下面代码：

```
TextUtil.isNotMax = function(oTextArea) {
    return oTextArea.value.length != oTextArea.getAttribute("maxlength");
}
```

可以看到，这个方法十分简单：只需将文本框内的文本长度和 `maxlength` 特性进行比较即

可。注意，虽然 `maxlength` 并非`<textarea />`有效的 HTML 特性，仍可以用 `getAttribute()` 来获取它的值。

下面，方法的调用必须放到文本框的 `onkeypress` 事件处理函数中。记住，`keypress` 事件是在文本被插入到文本框之前触发的，如果达到最大的字符数，则必须阻止字符的插入。代码如下：

```
<textarea rows="10" cols="25" maxlength="150"
    onkeypress="return TextUtil.isNotMax(this)"></textarea>
```

注意 `isNotMax()` 的返回值被传给事件处理函数。这是一种阻止事件默认行为发生的原始的方法。当文本长度小于 `maxlength` 时，方法返回 `true`，表示 `keypress` 事件可以正常继续。一旦达到最大值，方法返回 `false`，就阻止字符添加到文本框中。

这个方法可以和 `FormUtil.tabForward()` 一起使用，来为`<textarea />`元素添加直接跳到下一个字段的功能。

你也许会问为何不用事件的标准的 `preventDefault()` 方法来阻止 `keypress` 事件。答案很简单，在 Mozilla 对 `keypress` 事件的处理上有个错误，会造成 `preventDefault()` 功能异常。为使这个方法真正成为跨浏览器的解决方法，就未使用这个有缺陷的函数。这个例子中的代码可以在所有的 DOM 兼容的浏览器中运行，包括 IE、Safari、Opera 以及 Mozilla。

348

11.3.7 允许/阻止文本框中的字符

处理数据时，必须限制可以输入的数据。例如，如果字段要求的是一个数字，但用户输入的是字符，则应该视为无效。在 JavaScript 出现前，要返回服务器端来进行这类验证。有了 JavaScript，不仅可以直接在客户端验证用户数据，还可以直接阻止用户输入无效的数据。

1. 阻止无效的字符

第一个方法仅阻止无效字符。因为表单中的很多字段都要求输入不同类型的字符，必须能针对字段指定不同的要阻止的字符。最理想的方法是给 HTML `<input />`元素添加指定无效字符的特性，像这样：

```
<input type="text" invalidchars="0123456789" />
```

理想状态下，前面的例子可直接阻止向文本框输入 0~9 这几个数字。当然，为达到这个目的，还需使用另一个方法。

`TextUtil.block()` 方法接受两个参数：要处理的文本框和 `event` 对象。类似于 `TextUtil.isNotMax()`，要在 `onkeypress` 事件处理函数中调用这个方法，当字符被允许时返回 `true`，不允许时，返回 `false`。方法的主体只有四行代码：

```
TextUtil.blockChars = function (oTextbox, oEvent) {
    oEvent = EventUtil.formatEvent(oEvent);

    var sInvalidChars = oTextbox.getAttribute("invalidchars");
    var sChar = String.fromCharCode(oEvent.charCode);

    var bIsValidChar = sInvalidChars.indexOf(sChar) == -1;

    return bIsValidChar || oEvent.ctrlKey;
};
```

注意对本书前面定义的 `Event.formatEvent()` 方法的使用，当 `event` 对象是直接传递给方法而未使用 `EventUtil.getEvent()` 方法时，这种使用是必要的。事件对象经过合理地格式化后，方法将 `invalidchars` 特性存储到一个变量中，然后用事件的 `charCode` 特性和 `String.fromCharCode()` 取出将被输入到文本框中的字符。这时，无效的字符被存储在 `sInvalidChars` 中，将被插入的字符存储在 `sChar` 中。然后，用 `indexOf()` 方法来判断字符是否在 `sInvalidChars` 字符串中。如果子串（在这个例子中，就是输入的字符）不在字符串中，那么 `indexOf()` 返回`-1`。所以当字符不存在于 `sInvalidChars` 中时，`bIsValidChar` 为 `true`。返回语句返回 `bIsValidChar` 和 `oEvent.ctrlKey` 的逻辑或值。这个运算是必要的，如果没有它，当按下 `Ctrl` 键后，再按其他字符（例如 `Ctrl+C` 复制），这个方法就会将其阻止。所以，如果字符是有效的，方法返回 `true`，而且如果按下 `Ctrl` 键，也应该返回 `true`。

349

要用这个方法，将其插入到带有 `invalidchars` 特性的文本框的 HTML 的代码中（`<input />` 或者 `<textarea />` 中）。

```
<input type="text" invalidchars="0123456789"
       onkeypress="return TextUtil.blockChars(this, event)" />

<textarea rows="10" cols="25" invalidchars="0123456789"
           onkeypress="return TextUtil.blockChars(this, event)" />
```

这个函数可以用来在多种情况下阻止字符：

```
<!-- block all numbers -->
<input type="text" onkeypress="return FormUtil.block(this, event)"
       invalidchars="0123456789" />

<!-- block all uppercase letters -->
<input type="text" onkeypress="return FormUtil.block(this, event)"
       invalidchars="ABCDEFGHIJKLMNPQRSTUVWXYZ" />

<!-- block all lowercase letters -->
<input type="text" onkeypress="return FormUtil.block(this, event)"
       invalidchars="abcdefghijklmnopqrstuvwxyz" />

<!-- block spaces -->
<input type="text" onkeypress="return FormUtil.block(this, event)"
       invalidchars=" " />
```

2. 允许有效的字符

很自然，另外一种限制用户输入的方法是在文本框中只允许输入特定字符。我们还是为文本框加上 HTML 特性：

```
<input type="text" validchars="0123456789" />
```

这个例子将只允许 0~9 的数字，不允许其他字符。我们还需要 `allowChars()` 方法，实现和 `blockChars()` 方法相反的功能：

```
TextUtil.allowChars = function (oTextbox, oEvent) {
    oEvent = EventUtil.formatEvent(oEvent);

    var sValidChars = oTextbox.getAttribute("validchars");
    var sChar = String.fromCharCode(oEvent.charCode);

    var bIsValidChar = sValidChars.indexOf(sChar) > -1;

    return bIsValidChar || oEvent.ctrlKey;
};
```

正如你所见，`allowChars()` 与 `blockChars()` 之间有很多共同点：接受文本框和事件对象作为参数；使用 `EventUtil.formatEvent()` 来格式化事件对象；用 `String.fromCharCode()` 将要输入到文本框中的字符储存在变量中。它们的主要区别是，`allowChars()` 是查找 `validchars` 特性，如果 `sValidChars` 中包含 `sChar`，则返回 `true`。还需注意的是，如果按下 Ctrl 键，说明用户在对文本框进行某种特殊操作，则方法也要返回 `true`。甚至，这个方法的用法和前面的也类似，通过给 `onkeypress` 事件处理函数返回某值：

```
<input type="text" validchars="0123456789"
       onkeypress="return TextUtil.allowChars(this, event)" />

<textarea rows="10" cols="25" validchars="0123456789"
           onkeypress="return TextUtil.allowChars(this, event)" />
```

这两个文本框只允许输入数字（不允许空格、字符或其他符号）。还可以有多种新奇的用法：

```
<!-- only allow positive integers -->
<input type="text" validchars="0123456789"
       onkeypress="return FormUtil.allow(this, event)" />

<!-- only allow "Y" or "N" -->
<input type="text" validchars="YN"
       onkeypress="return FormUtil.allow(this, event)" />
```

3. 不要忘记粘贴

很多开发人员会遗忘的文本验证的一个重要方面就是用户可以直接向文本框粘贴内容。在 `blockChars()` 和 `allowChars()` 方法中，假设用户逐个输入字符，所以它们在字符进入时逐个检查。但当用户粘贴内容时，整个字符串就被放到文本框中。现在有两种方法用来处理粘贴内容的验证：禁止粘贴或者当文本框失去焦点时验证。

● 禁止粘贴

阻止用户粘贴内容很容易办到，但必须涵盖各种方法。用户可以通过两种方法粘贴：通过点击文本框上下文菜单（右键点击它）的粘贴选项或者按下 `Ctrl + V`。

在 IE 中，解决方法十分简单，因为有 `paste` 事件。如果 `onpaste` 事件处理函数阻止默认行为，就不会进行任何粘贴操作，不管用户怎么尝试：

```
<input type="text" onkeypress="return TextUtil.allowChars(this, event)" 
       validchars="0123456789"
       onpaste="return false" />
```

关于其他浏览器，这个过程要复杂些。首先要做的是阻止上下文菜单，这可通过在 `oncontextmenu` 事件处理函数中返回 `false` 来实现。

```
<input type="text" onkeypress="return TextUtil.allowChars(this, event)" 
       validchars="0123456789"
       onpaste="return false" oncontextmenu="return false" />
```

下面，要阻止用户按下 `Ctrl + V` 进行的粘贴。这个问题，其实就是要按下 `Ctrl` 和 `V`，触发 `keypress` 事件，所以只要修改 `allowChars()` 和 `blockChars()` 方法：

```
TextUtil.blockChars = function (oTextbox, oEvent, bBlockPaste) {
    oEvent = EventUtil.formatEvent(oEvent);

    var sInvalidChars = oTextbox.getAttribute("invalidchars");
    var sChar = String.fromCharCode(oEvent.charCode);

    var bIsValidChar = sInvalidChars.indexOf(sChar) == -1;

    if (bBlockPaste) {
        return bIsValidChar && !(oEvent.ctrlKey && sChar == "v");
    } else {
        return bIsValidChar || oEvent.ctrlKey;
    }
};

TextUtil.allowChars = function (oTextbox, oEvent, bBlockPaste) {
    oEvent = EventUtil.formatEvent(oEvent);

    var sValidChars = oTextbox.getAttribute("validchars");
    var sChar = String.fromCharCode(oEvent.charCode);

    var bIsValidChar = sValidChars.indexOf(sChar) > -1;

    if (bBlockPaste) {
        return bIsValidChar && !(oEvent.ctrlKey && sChar == "v");
    } else {
        return bIsValidChar || oEvent.ctrlKey;
    }
};
```

注意给两个方法添加的是相同的代码。首先，添加第三个参数，`bBlockPaste`，要阻止文本框粘贴，这个参数必须设置为 `true`。然后，在每个方法的末尾添加 `if` 语句，来检查是否要阻止粘贴。如果要阻止，只有当字符是有效的并且按下的不是 `Ctrl + V` 时，返回 `true`。如果不要求阻止粘贴，则 `return` 语句使用原来的方法。

要使用新方法，只需加入第三个参数：

```
<input type="text" validchars="0123456789"
       onpaste="return false" oncontextmenu="return false"
       onkeypress="return TextUtil.allowChars(this, event, true)" />

<textarea rows="10" cols="25" validchars="0123456789"
            onpaste="return false" oncontextmenu="return false"
            onkeypress="return TextUtil.allowChars(this, event, true)" />
```

尽管定义了第三个参数，仍然可只给出前两个参数，因为 `undefined` 值在 `if` 语句中也认为是 `false`。

352

必须同时使用 `onpaste="return false"` 以及 `allowChars()` 和 `blockChars()` 方法，因为在 IE 中无法阻止 `Ctrl + V` 按键组合。

● 失去焦点时验证

如果不禁止用户粘贴，则必须用另一种方法进行验证。最简单的方式是创建 `onblur` 事件处理函数，在用户未输入正确的值之前，不让用户移动到另一个字段。这需要用到两种方法：一个与 `blockChars()` 一起用，另一个与 `allowChars()` 一起用。

与 `blockChars()` 相关的函数是 `blurBlock()`，它接受文本框作为唯一的参数。代码如下：

```
TextUtil.blurBlock = function(oTextbox) {

    var sInvalidChars = oTextbox.getAttribute("invalidchars");
    var arrInvalidChars = sInvalidChars.split("");

    for (var i=0; i< arrInvalidChars.length; i++) {
        if (oTextbox.value.indexOf(arrInvalidChars[i]) > -1) {
            alert("Character '" + arrInvalidChars[i] + "' not allowed.");
            oTextbox.focus();
            oTextbox.select();
            return;
        }
    }
};
```

这个方法中的第一步与 `blockChars()` 的一样：从 `invalidchars` 特性中获取无效字符。因为必须验证文本框中所有的文本（不是单个字符），所以必须对每个无效字符分别进行检测，于是方法中的下一步是将无效的字符串分割成字符数组（回想一下，以空字符串为参数，以字符数组为返回值的函数 `split()` 的用法）。然后，进行循环，检测每个无效的字符是否出现在文本框中。当发现一个无效字符，就显示警告框通告用户给定的字符是无效的。然后焦点切回文本框，

并将其内容选中。然后方法结束，因为只要存在一个无效的字符，就不需要继续检测了。

这个方法要插到 `onblur` 事件处理函数中：

```
<input type="text" onkeypress="return TextUtil.blockChars(this, event)"
       invalidchars="0123456789" onblur="TextUtil.blurBlock(this)" />
```

与 `allowChars()` 一起用的另一个类似的方法是 `blurAllow()`，它基本上和 `blurBlock()` 相同，不过它的功能是保证文本框中的每一个字符都是有效的：

```
TextUtil.blurAllow = function(oTextbox) {
    var sValidChars = oTextbox.getAttribute("validchars");
    var arrTextChars = oTextbox.value.split("");
    for (var i=0; i< arrTextChars.length; i++) {
        if (sValidChars.indexOf(arrTextChars[i]) == -1) {
            alert("Character '" + arrTextChars[i] + "' not allowed.");
            oTextbox.focus();
            oTextbox.select();
            return;
        }
    }
};
```

注意，这个方法先从 `validchars` 特性中获取有效的字符，与 `allowChars()` 一样。下一步是，将文本框中的文字分割成字符的数组，因为必须对每个字符验证有效性。方法对数组进行循环，并检测是否每个字符都包含在 `sValidChars` 字符串里。当遇到一个无效字符时（也就是说 `indexOf()` 返回-1，表示给定的字符不在 `sValidChars` 中），就出现警告框显示非法的字符。然后，类似 `blurBlock()`，焦点切回文本框，同时选中所有文本，然后方法退出。这个方法可这样使用：

```
<input type="text" onkeypress="return TextUtil.allowChars(this, event)"
       validchars="0123456789" onblur="TextUtil.blurAllow(this)" />
```

这样做就不会阻止用户粘贴非法的内容，但是确保了如果粘贴非法的内容，可立刻警告用户。

你也许会问为什么代码使用 `onblur` 事件处理函数而不是 `onchange`。理论上讲，因为 `change` 事件只有在文本框的内容改变后，文本框失去焦点时才触发，看上去这好像才是最完美的检测粘贴内容的方法。那么试想一下这个代码执行的情况。首先，用户向文本框粘贴了非法内容，然后尝试切换到下个字段。首先触发了 `change` 事件，然后是 `blur` 事件。用户被告知发现非法字符，同时焦点被切回这个文本框。但因某种原因，用户并未修改非法的字符，而直接向前切换。这次，并没有触发 `change` 事件，因为自上次获取焦点以来，文本框中的内容并未发生改变。然而，`blur` 事件仍会触发。

11.3.8 使用上下按键操作数字文本

假设已使用 `TextUtil.allowChars()` 方法实现了仅接受数字的文本框，但对用户来说这往

往还不够。有时候他们还想能够通过按下上下按键来增加或者减少数字。为实现它，需要使用 onkeydown 事件处理函数。

你可能会问，为什么不继续使用 onkeypress 事件处理函数呢？答案是，keypress 事件仅对那些能表示文本框内出现的字符的按键才触发。因为方向键不会产生能输入到文本框中的字符，所以不会触发 keypress 事件。然而，keydown 事件是不管哪个按键按下都会触发的。

为确保处理的只是上下按键，要使用 event 的 keyCode 特性。上键的代码是 38，下键的是 40。其他的按键都被忽略：

```
TextUtil.numericScroll = function (oTextbox, oEvent) {
    oEvent = EventUtil.formatEvent(oEvent);
    var iValue = oTextbox.value.length == 0 ? 0 : parseInt(oTextbox.value);

    if (oEvent.keyCode == 38) {
        oTextbox.value = (iValue + 1);
    } else if (oEvent.keyCode == 40) {
        oTextbox.value = (iValue - 1);
    }
};
```

这一次，又使用 EventUtil.formatEvent() 方法来对 event 对象进行格式化。下一步是，判断文本的数值。如果文本框中有内容，则使用 parseInt() 函数来转换值；否则，就假设为零。然后，测试事件的 keyCode 特性是上还是下。根据 keyCode 来增加或者减少整数值，然后再放入文本框内。该方法必须和 allowChars() 一起使用（以及任意一种处理粘贴的方法）来确保文本框中只出现整数。

这个方法这样使用：

```
<input type="text" onkeypress="return TextUtil.allowChars(this, event)"
       validchars="0123456789" onblur="TextUtil.blurAllow(this)"
       onkeydown="TextUtil.numericScroll(this, event)" />
```

简单地加入这个方法就可使文本框通过上下按键来更改数值。那么用户还可能需要什么呢？加入最小和最大值如何？只需加入两个自定义特性 min 和 max，然后修改一下方法，就可以加入这个限制数值范围的功能：

```
TextUtil.numericScroll = function (oTextbox, oEvent) {
    oEvent = EventUtil.formatEvent(oEvent);
    var iValue = oTextbox.value.length == 0 ? 0 : parseInt(oTextbox.value);

    var iMax = oTextbox.getAttribute("max");
    var iMin = oTextbox.getAttribute("min");

    if (oEvent.keyCode == 38) {
        if (iMax == null || iValue < parseInt(iMax)) {
            oTextbox.value = (iValue + 1);
        }
    } else if (oEvent.keyCode == 40) {
```

```

        if (iMin == null || iValue > parseInt(iMin)) {
            oTextbox.value = (iValue - 1);
        }
    }
};

```

为完成一些特定的任务给方法添加几行代码。首先，从自定义特性中获取最小值和最大值。然后测试按键时，检测最小值和最大值是否被指定。如果未指定这两个特性，`iMax` 和 `iMin` 则等于 `null`。如果它们不为 `null`，则调用 `parseInt()` 来获取它们的整数值。然后将这个值和文本框中的值进行比较来判断是否要更改（增加或者减少）文本框中的值。

要使用这个功能，只需给 `<input>` 标签添加 `min` 或 `max` 特性（或者两个都添加）：

```

<input type="text" onkeypress="return TextUtil.allowChars(this, event)"
       validchars="0123456789" onblur="TextUtil.blurAllow(this)"
       onkeydown="TextUtil.numericScroll(this, event)"
       max="100" min="0" />

```

使用这段代码，当数值达到两个界限时，就会停止增加或者减少。

11.4 列表框和组合框

列表框和组合框是通过 HTML 的 `<select>` 元素来创建的。默认情况下，浏览器会将 `<select>` 元素渲染成组合框：

```

<select name="selAge" id="selAge">
    <option value="1">18-21</option>
    <option value="2">22-25</option>
    <option value="3">26-29</option>
    <option value="4">30-35</option>
    <option value="5">Over 35</option>
</select>

```

其中 `<option>` 的 `value` 特性用来确定控件所有可能的值；所选的选项可将它的 `value` 指定给控件（这样这个值就可以发送到服务器了）。

要将同样的代码渲染成列表框，只要添加 `size` 特性用来指示同时可见的条目的个数。例如，下面的代码将显示有三个条目同时可见的列表框。

```

<select name="selAge" id="selAge" size="3">
    <option value="1">18-21</option>
    <option value="2">22-25</option>
    <option value="3">26-29</option>
    <option value="4">30-35</option>
    <option value="5">Over 35</option>
</select>

```

因为两种控件都使用同样的 HTML 代码，所以可用同样的 JavaScript 代码对它们进行处理。第一步，通过 `document.getElementById()` 来获取控件的引用或者直接通过 `document.forms` 集合进行访问：

```

oListbox = document.getElementById("selAge");
oListbox = document.forms["form1"].selAge;
oListbox = document.forms[0].selAge;

```

这一节中，我们创建的所有方法都隶属于 ListUtil 对象，以便保持直观（类似于前面创建的 EventUtil 对象）。ListUtil 仅仅是用于放置各个方法的简单对象：

```
var ListUtil = new Object();
```

11.4.1 访问选项

HTML DOM 为每个`<select>`元素定义了 `options` 集合，它是控件的所有`<option>`元素的列表。要获取`<option>`的显示文本和值，可以使用一般的 DOM 方法：

```

alert(oListbox.options[1].firstChild.nodeValue);      //output display text
alert(oListbox.options[1].getAttribute("value"));    //output value

```

不过，使用两个 HTML DOM 中定义的`<option>`特性会更加简单：`text`，返回显示文本；`value`，返回值特性。提供这两个特性是为了保持和旧的 BOM 功能的向下兼容。

```

alert(oListbox.options[1].text);      //output display text
alert(oListbox.options[1].value);    //output value

```

每个`<option>`也有个 `index` 特性，表示它在 `options` 集合中的位置：

```
alert(oListbox.options[1].index);      //outputs "1"
```

当然，因为 `options` 是一个集合，可以使用它的 `length` 特性来判断有多少个选项：

```
alert("There are " + oListbox.options.length + " in the list.");
```

但是如何才能知道目前选中的是哪个选项呢？

11.4.2 获取/更改选中项

`<select>`元素有一个 `selectedIndex` 特性，它总是包含目前选中的选项的索引（如果没有选中任何选项，那么为-1）。

```
alert("The index of the selected option is " + oListbox.selectedIndex);
```

不过，可以在列表框中选择多个选项（组合框不能），只要将`<select>`元素的 `multiple` 特性设置为 "multiple"。

```

<select name="selAge" id="selAge" size="3" multiple="multiple">
  <option value="1">18-21</option>
  <option value="2">22-25</option>
  <option value="3">26-29</option>
  <option value="4">30-35</option>
  <option value="5">Over 35</option>
</select>

```

如果选中多个选项，`selectedIndex` 将包含第一个选中选项的索引，但这没有多大用途。

我们需要的是能够获取所有选中选项索引的方式。为此，需要自己创建方法，这就是 ListUtil 对象的第一个方法。

getSelectedIndexes() 方法利用了<option/>元素的另一个特殊的特性：selected 特性。HTML DOM 定义的 selected 特性是个 Boolean 值，表示这个选项是否被选中。接下来，就是对列表框中所有的选项进行循环，测试它们是否被选中。如果选中了，就要把索引存到数组中。

这个方法只有一个参数，需要进行检查的列表框：

```
ListUtil.getSelectedIndexes = function (oListbox) {
    var arrIndexes = new Array;

    for (var i=0; i < oListbox.options.length; i++) {
        if (oListbox.options[i].selected) {
            arrIndexes.push(i);
        }
    }

    return arrIndexes;
};
```

然后 getSelectedIndexes() 方法可以用来获取所有选中选项的索引或者使用返回的数组的长度，来判断选中的选项个数：

```
var oListbox = document.getElementById("selListbox");
var arrIndexes = ListUtil.getSelectedIndexes(oListbox);

alert("There are " + arrIndexes.length + " option selected."
    + "The options have the indexes " + arrIndexes + ".");
```

这段代码首先获取 ID 为“selListbox”的列表框的引用，然后获取选中的索引，并将它们存储在 arrIndexes 中。警告框显示选中选项的数量，以及它们的索引（记住，Array 对象的 toString() 方法将返回逗号分隔的字符串）。

358 尽管针对的是多选框，getSelectedIndexes() 方法也可以用于单选框和组合框，返回的是只有一个条目的数组：selectedIndex 的值。

11.4.3 添加选项

如果不使用 HTML 给列表框载入任何选项，还可用 JavaScript 添加。

我们先定义一个接受三个参数的方法：要处理的列表框，要添加的选项的名称以及要添加的选项的值。

```
ListUtil.add = function (oListbox, sName, sValue) {
    //...
}
```

下面，使用 DOM 方法创建<option/>元素，并通过创建一个文本节点来分配选项名称：

```
ListUtil.add = function (oListbox, sName, sValue) {
    var oOption = document.createElement("option");
    oOption.appendChild(document.createTextNode(sName));
    //...
}
```

选项值其实并非必要，只有传入这个参数时才将其添加进去。首先要确保 arguments.length 等于 3，如果是，则设置 value 特性：

```
ListUtil.add = function (oListbox, sName, sValue) {
    var oOption = document.createElement("option");
    oOption.appendChild(document.createTextNode(sName));

    if (arguments.length == 3) {
        oOption.setAttribute("value", sValue);
    }
    //...
}
```

最后一步是，使用 appendChild() 方法将新的选项添加到列表框中：

```
ListUtil.add = function (oListbox, sName, sValue) {
    var oOption = document.createElement("option");
    oOption.appendChild(document.createTextNode(sName));

    if (arguments.length == 3) {
        oOption.setAttribute("value", sValue);
    }
    oListbox.appendChild(oOption);
}
```

359

可以这样使用这个方法：

```
var oListbox = document.getElementById("selListbox");

ListUtil.add(oListbox, "New Display Text"); //add option with no value
ListUtil.add(oListbox, "New Display Text 2", "New value"); //add option with value
```

11.4.4 删除选项

JavaScript 不仅提供添加选项的能力，还可以删除。有种古老的从列表框中删除选项的方法，是用 options 集合将需要删除的选项设置为 null。

```
oListbox.options[1] = null;
```

这也是 BOM 的功能；使用 HTML DOM 可更有逻辑地删除。HTML DOM 为 <select /> 提供

`remove()`方法。只需将要删除的选项的索引传递给它：

```
var oListbox = document.getElementById("selListbox");
oListbox.remove(0); //remove the first option
```

也可以选择将其包装在 `ListUtil` 的方法中，这样使用添加和删除的方式就一致了：

```
ListUtil.remove = function (oListbox, iIndex) {
    oListbox.remove(iIndex);
}
```

前面的代码可以这样重写：

```
var oListbox = document.getElementById("selListbox");
ListUtil.remove(oListbox, 0); //remove the first option
```

如果想删除列表框中所有的选项，只要对每个选项调用 `remove()` 方法：

```
ListUtil.clear = function (oListbox) {
    for (var i=oListbox.options.length-1; i >= 0; i--) {
        ListUtil.remove(oListbox, i);
    }
};
```

360

这个方法按照逆序删除所有的选项。按照逆序是因为每删除一个选项后，每个选项的 `index` 特性就会被重置。因此，最好从最大的索引开始删除。

11.4.5 移动选项

在早期的 JavaScript 中，将选项从一个列表框中移动到另一个列表框中是很困难的，需要先从第一个列表框中删除该选项，然后创建一个相同名称和值的选项，再将其添加到第二个列表框。幸好，DOM 提供了更加方便的方法来完成这个任务。使用 DOM 方法，可以直接从第一个列表框将一个选项移动到另一个列表框中，利用 `appendChild()` 方法。如果将已经存在于文档中的元素传递给这个方法，那么会将这个元素从原来的父节点中删除，并放入指定的位置。

执行这个功能的方法接受三个参数：当前选项存在的列表框、要移入的列表框以及要移动的选项的索引。然后这个方法将给定索引上的选项（假定已经存在）移动到第二个列表框。

```
ListUtil.move = function (oListboxFrom, oListboxTo, iIndex) {
    var oOption = oListboxFrom.options[iIndex];

    if (oOption != null) {
        oListboxTo.appendChild(oOption);
    }
}
```

使用方法如下：

```
var oListbox1 = document.getElementById("selListbox1");
var oListbox2 = document.getElementById("selListbox2");
ListUtil.move(oListbox1, oListbox2, 0); //move the first option
```

这段代码将第一个选项从 `oListbox1` 移动到 `oListbox2`（新选项将出现在 `oListbox2` 的

底部)。

移动选项的情况和删除它们的情况一样，就是每个选项的 `index` 特性都会被重置为新的值，所以每次应该从最高的索引开始移动。

11.4.6 重新排序选项

要对列表框中的选项重新排序，首先要能将特定的选项向上或者向下移动，需要用到两个方法，一个用来将选项向上移动，另一个向下移动。每个方法都有两个参数：要处理的列表框和要移动的选项的索引。两个方法都要用 DOM 的 `insertBefore()` 方法对`<option/>`元素进行重新排序。

361

首先看 `shiftUp()` 方法，它可以将选项在列表框中向上移动一个位置：

```
ListUtil.shiftUp = function (oListbox, iIndex) {
    if (iIndex > 0) {
        var oOption = oListbox.options[iIndex];
        var oPrevOption = oListbox.options[iIndex-1];
        oListbox.insertBefore(oOption, oPrevOption);
    }
};
```

这个方法首先检查以确保要移动的选项的索引大于 0，因为无法将第一个选项向上移动。给定索引的选项保存在变量 `oOption` 中并且在它之前的选项存储在 `oPrevOption` 中。最后，调用 `insertBefore()` 方法将 `oOption` 选项移动到 `oPrevOption` 之前。将选项向下移动一个位置的方法也十分相似：

```
ListUtil.shiftDown = function (oListbox, iIndex) {
    if (iIndex < oListbox.options.length - 1) {
        var oOption = oListbox.options[iIndex];
        var oNextOption = oListbox.options[iIndex+1];
        oListbox.insertBefore(oNextOption, oOption);
    }
};
```

在这里，必须先获取选项的集合来确保 `iIndex` 不是列表中的最后一个选项（因为不能将最后一个选项再向下移动）。位置 `iIndex` 上的选项存储在 `oOption` 中；下一个位置上的选项存储在 `oNextOption` 中。使用 `insertBefore()`，将 `oNextOption` 放在列表框中的 `oOption` 之前。

这两个方法可以以如下方式使用：

```
var oListbox = document.getElementById("selListbox");
ListUtil.shiftUp(oListbox,1); //move the option in position 1 up one spot
ListUtil.shiftDown(oListbox,2); //move the option in position 2 down one spot
```

11.5 创建自动提示的文本框

我们必须面对这个事实，用户并不总愿意填写表单，尤其是需要用键盘输入内容时。这就是

为什么像 Microsoft Outlook 之类的应用会采用自动提示的文本框。这种文本框会检查用户输入的头几个字符，然后给出一个或多个可帮助用户输入的单词列表。在 Web 浏览器的地址栏中输入地址时也是这样。只要使用一些 JavaScript 技巧，就可在 Web 表单中创建类似的行为。

11.5.1 匹配

这个过程的第一步是，写一个方法来搜索字符串数组并返回以特定字符开头的所有值（例如，如果传入 a，那么这个方法要返回数组中所有以字母 a 开头的字符串）。这个方法称为 **TextUtil.autosuggestMatch()**，它需要两个参数：要匹配的文本以及要进行匹配的数组。

```
362 TextUtil.autosuggestMatch = function (sText, arrValues) {
    var arrResult = new Array;
    if (sText != "") {
        for (var i=0; i < arrValues.length; i++) {
            if (arrValues[i].indexOf(sText) == 0) {
                arrResult.push(arrValues[i]);
            }
        }
    }
    return arrResult;
};
```

这个方法的第一步是，创建用于存储所有匹配的值的数组。然后，方法要确保进行匹配的字符串非空（空字符串被认为存在于任何字符串中）。如果字符串非空，则用简单的 for 循环检查每个值，看其是否以这个字符串开头。使用 indexOf() 方法进行判断。如果 indexOf() 返回 0，则表示匹配，则将这个字符串添加到结果数组中。最后，返回包含匹配值的数组。

11.5.2 内部机制

完成匹配方法后，要创建脚本最重要的部分：**TextUtil.autosuggest()** 方法。这个方法接受三个参数：要处理的文本框，可能值的数组以及所要显示候选项的列表框的 ID。假设值数组为 **arrValues**，调用如下：

```
<input type="text"
       onkeyup="TextUtil.autosuggest(this, arrValues, 'lstSuggestions')"/>
```

其中使用了 **onkeyup** 事件处理函数，因为 **keyup** 事件是当字符输入到文本框中后才触发的，这可使每次文本框发生变化时都给出自动提示。这个方法定义如下：

```
TextUtil.autosuggest = function (oTextbox, arrValues, sListboxId) {
    var oListbox = document.getElementById(sListboxId);
    ListUtil.clear(oListbox);
```

```

var arrMatches = TextUtil.autosuggestMatch(oTextbox.value, arrValues);

for (var i=0; i < arrMatches.length; i++) {
    ListUtil.add(oListbox, arrMatches[i]);
}

};


```

这个方法首先获取 ID 为 sListboxId 的列表框的引用。然后使用 ListUtil.clear() 方法将列表框全部清空。

下面，这个方法调用 TextUtil.autosuggestMatch() 来获取匹配文本框中字符串的值。最后一步是，使用 ListUtil.add() 方法将所有匹配的值添加到列表框中。

要使用这个方法，页面上必须有一个文本框和一个列表框，并给出要使用的值的数组。这些值应按照字母顺序排列，这样当出现自动提示时它们是按照字母顺序的。下面是范例页面：

```

<html>
    <head>
        <title>Autosuggest Textbox Example</title>
        <script type="text/javascript" src="listutil.js"></script>
        <script type="text/javascript" src="textutil.js"></script>
        <script type="text/javascript">
            var arrColors = ["red", "orange", "yellow", "green", "blue", "indigo",
                "violet", "brown", "black", "tan", "ivory", "navy",
                "aqua", "white", "purple", "pink", "gray", "silver"];
            arrColors.sort();
        
```

```

            function setText(oListbox, sTextboxId) {
                var oTextbox = document.getElementById(sTextboxId);
                if (oListbox.selectedIndex > -1) {
                    oTextbox.value =
                        oListbox.options[oListbox.selectedIndex].text;
                }
            }
        </script>
    </head>
    <body>
        <p>Type in a color in lowercase:<br />

        <input type="text" value="" id="txtColor"
            onkeyup="TextUtil.autosuggest(this, arrColors, 'lstColors')"/><br />

        <select id="lstColors" size="5" style="width: 200px"
            onclick="setText(this, 'txtColor')"/></select>
    </p>
</body>
</html>

```

这个例子中，定义了 arrColors 颜色数组。因为这些不是按照字母表的顺序排列的，所以先调用 sort() 方法。TextUtil.autosuggest() 要引用的就是这个数组。ID 为 "lstColors" 的列表框也有个 onclick 事件处理函数，用于将文本框的内容设置为当前选中的选项（出于方便，

尽管不是自动提示功能所必需的)。`SetText()`方法有两个参数：列表框和文本框的 ID。然后这个方法获取文本框的引用，并将它的值设置为目前列表框中选中的值。

本节中描述的自动提示功能是区分大小写的。如果要进行不区分大小写的比较，需将数组中的所有值转换成小写并与转化为小写的文本框中的值进行比较(当然也可以全部转化成大写)。

11.6 小结

在本章中，我们探索了多种增强 Web 表单功能的途径。还学习了使用 JavaScript 来重置和提交表单。这包括确保表单仅提交一次。还探索了多种访问表单元素的途径。

随后，学习了关于文本框的很多知识，包括如何只允许或不允许特定的字符。接着，还探索了如何阻止用户向文本框中粘贴无效的值，以及如何使用 `onblur` 事件处理函数来验证文本框的值。

还介绍了列表框和组合框，讨论了各种不同的操作方法，包括添加新的选项、删除存在的选项，以及在单个列表框或者两个列表框中移动选项。

最后，我们使用这些知识来创建一个自动提示文本框，在用户输入一些字符后向他提示一些候选项。在下面几章中，我们将学到更多关于利用 JavaScript 来增强 Web 页面可用性的方法。

大多数应用中，对列表和表格进行排序是种常见的操作过程，甚至是日常使用的基本功能。查看电子邮件时，你可能将表格设置为按照日期栏进行降序排列，将最近的电子邮件放在最顶端。这种典型应用只用了极少的一段时间，就转到了 Web 上。

以前，在 Web 上进行排序要涉及到与服务器之间的交互，请求的过程中指出需要对哪个栏目进行排序，按照何种顺序。然而，JavaScript 让你可以直接在客户端创建类似的功能。

通过使用 JavaScript，创建一种可以进行排序的表格，就无需耗时的服务器端处理。

12.1 起点——数组

第 3 章介绍了 `Array` 对象和它的 `sort()` 方法。你可能还记得 `sort()` 方法是按照条目的 ASCII 字符代码的升序进行排列的，也就是说，数字也是按照它们的字符串形式排列的：

```
var arr = [3, 32, 2, 5]
arr.sort();
alert(arr.toString()); //outputs "2,3,32,5"
```

前面的例子在输出数组时，将输出“2,3,32,5”。幸好，JavaScript 并不会为难你。`sort()` 方法还可以仅接受一个参数，即比较函数，来告诉排序算法值与值之间是大于、小于还是等于关系。

比较函数是有着特定算法的函数。继续解释前，先看一个很基本的比较函数还是有益的：

```
function comparison_function(value1, value2) {
    if (value1 < value2) {
        return -1;
    } else if (value1 > value2) {
        return 1;
    } else {
        return 0;
    }
};
```

367

可以看到，比较函数将对两个值做比较，所以，比较函数总是有两个参数。如果第一个参数应该在第二个参数之前，函数返回-1；如果第一个参数应该在第二个参数之后，则返回 1。如果

两个参数相等，则返回 0。在 sort() 方法中，这样使用比较函数：

```
arr.sort(comparison_function);
```

前面介绍的基本的比较函数的模式可将数组按照从小到大的顺序进行排列。要按照从大到小的顺序进行排列，只需调换 -1 和 1 的位置：

```
function comparison_function_desc(value1, value2) {
    if (value1 < value2) {
        return 1;
    } else if (value1 > value2) {
        return -1;
    } else {
        return 0;
    }
};
```

你是否觉得这个模式看上去很熟悉，那是因为与 String 的 localeCompare() 方法一样。所以如果要对一个字符串数组进行排序，可以直接使用这个方法：

```
function compareStrings(string1, string2) {
    return string1.localeCompare(string2);
}
```

这个函数将令字符串数组按照从小到大的顺序排列。要将数组按照逆序排列，只需在调用 localeCompare 之前加个负号：

```
function compareStringsDesc(string1, string2) {
    return -string1.localeCompare(string2);
}
```

添加负号后，1 变成 -1，-1 变成 1，0 不变。

现在，回到前面数字排序错误的例子。只需写个比较函数就可轻易解决这个问题，这个比较函数要将参数转换成数字，然后进行比较：

```
function compareIntegers(vNum1, vNum2) {
    var iNum1 = parseInt(vNum1);
    var iNum2 = parseInt(vNum2);
    if (iNum1 < iNum2) {
        return -1;
    } else if (iNum1 > iNum2) {
        return 1;
    } else {
        return 0;
    }
}
```

368

如果将这个比较函数应用到前面的例子中，将返回正确的结果：

```
var arr = [3, 32, 2, 5]
arr.sort(compareIntegers);
alert(arr.toString()); //outputs "2,3,5,32"
```

这个例子现在输出正确顺序的数字(2, 3, 5, 32)。

reverse()方法

现在要介绍 reverse()方法，它可将数组中元素的顺序倒转。在这一章中，reverse()是排序的很基础的部分。

这样，如果使用比较函数是按照升序进行排序，则使用 reverse()方法可以很容易地将升序变成降序：

```
var arr = [3, 32, 2, 5]
arr.sort(compareIntegers);
alert(arr.toString());      //outputs "2,3,5,32"
arr.reverse();
alert(arr.toString());      //outputs "32,5,3,2"
```

当然，这仅给排序过程增加了一个额外的步骤，当排序是必要的时，创建两个比较函数也是完全无错的。不过记住，如果数组已经以一种顺序排列了，使用 reverse()将其按照相反的顺序排列，要比再次调用 sort()快得多。

12.2 对单列的表格排序

现在开始手头的任务——表格排序。最简单的情况是，对只有一列的表格进行排序，当然，也只有一种数据类型。建立要被排序的表格的最好的方法是，为表格头部创建`<thead>`元素，为存放数据的列创建`<tbody>`元素。

```
<table border="1" id="tblSort">
  <thead>
    <tr>
      <th>Last Name</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>Smith</td>
    </tr>
    <tr>
      <td>Johnson</td>
    </tr>
    <tr>
      <td>Henderson</td>
    </tr>
    <tr>
      <td>Williams</td>
    </tr>
    <tr>
      <td>Gilliam</td>
    </tr>
    <tr>
      <td>Walker</td>
    </tr>
  </tbody>
</table>
```

建立好表格后，就很容易区分表头行和数据行（显然，不能将表头行和数据放在一起排序，所以这个区分很重要）。使用表格的 `tBodies` 集合（本书前面提到过），就可以获取`<tbody>`元素和它所包含的行的引用：

```
var oTBody = oTable.tBodies[0];
var colDataRows = oTBody.rows;
```

要使用 DOM 从单元格中获取数据还要涉及一些其他内容，虽然不是很困难。`rows` 集合所包含的每个`<tr>`元素均包含一个子`<td>`元素。每个`<td>`元素又包含一个文本子节点，其中包含实际要进行排序的值。图 12-1 显示了这个 DOM 的层次结构：

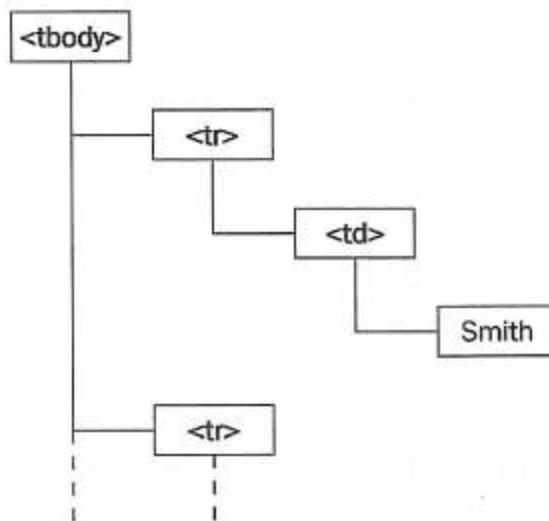


图 12-1

要从前面定义的表格中获取值 Smith，需要使用这个代码：

```
var sSmith = colDataRows[0].cells[0].firstChild.nodeValue;
```

这个方法可以用于获取每行包含的值，有了它才能创建用于排序的比较函数。

12.2.1 比较函数

比较函数很有意思的一点是它使用包含在行内的值对`<tr>`元素进行排序，也就是说必须在函数内获取这个值。在获取这些值后，可以使用 `localeCompare()` 对它们进行比较：

```
function compareTRs(oTR1, oTR2) {
    var sValue1 = oTR1.cells[0].firstChild.nodeValue;
    var sValue2 = oTR2.cells[0].firstChild.nodeValue;

    return sValue1.localeCompare(sValue2);
}
```

这个比较函数根据每行的第一个单元格（索引 0）中的值对表格行进行排序。下面，就可对表格使用这个比较函数了。

12.2.2 `sortTable()` 函数

这个 `sortTable()` 函数要完成大部分繁重的工作。它接受一个参数，要进行排序的表格的

ID。那么，函数第一步自然就是，获取表格的DOM引用来定位数据行：

```
function sortTable(sTableID) {
    var oTable = document.getElementById(sTableID);
    var oTBody = oTable.tBodies[0];
    var colDataRows = oTBody.rows;
    //...
}
```

这时的问题是，如何对 colDataRows 中的行进行排序。记住，rows 是个 DOM 集合，并非数组，因此，它并没有 sort() 方法。唯一的解决方案是创建一个数组，并将<tr>元素放入其中，对数组排序，最后使用 DOM 将行按顺序逐个放置。这样，首先要迭代<tr>元素并将它们添加到一个数组中：

```
function sortTable(sTableID) {
    var oTable = document.getElementById(sTableID);
    var oTBody = oTable.tBodies[0];
    var colDataRows = oTBody.rows;
    var aTRs = new Array;

    for (var i=0; i < colDataRows.length; i++) {
        aTRs.push(colDataRows[i]);
    }

    //...
}
```

371

这一段代码创建了数组 aTRs 并将<tr>元素的引用放入其中。这样做并不会从表格中删除<tr>元素，因为仅仅存储了指针，并不是实际的元素。

下一步是使用 compareTRs() 函数对数组进行排序：

```
function sortTable(sTableID) {
    var oTable = document.getElementById(sTableID);
    var oTBody = oTable.tBodies[0];
    var colDataRows = oTBody.rows;
    var aTRs = new Array;

    for (var i=0; i < colDataRows.length; i++) {
        aTRs.push(colDataRows[i]);
    }

    aTRs.sort(compareTRs);

    //...
}
```

然后，所有的<tr>元素在数组中都被排列好，但页面上的顺序并没有改变。要改变页面上的顺序，还要将每一行按顺序放回。实现这个操作最快的方法是，创建文档碎片，并将所有的<tr>元素按照正确的顺序附加其上。然后，可以使用 appendChild() 将所有的子节点添加到<tbody>元素中。

```

function sortTable(sTableID) {
    var oTable = document.getElementById(sTableID);
    var oTBody = oTable.tBodies[0];
    var colDataRows = oTBody.rows;
    var aTRs = new Array;

    for (var i=0; i < colDataRows.length; i++) {
        aTRs[i] = colDataRows[i];
    }

    aTRs.sort(compareTRs);

    var oFragment = document.createDocumentFragment();
    for (var i=0; i < aTRs.length; i++) {
        oFragment.appendChild(aTRs[i]);
    }

    oTBody.appendChild(oFragment);
}

```

这段代码创建一个文档碎片，并将所有的`<tr>`元素添加进去，并将它们从原来的表格中删除（图 12-2）。然后，碎片的子节点被放回到`<tbody>`元素中。记住，当使用`appendChild()`并传给它一个文档碎片，最后添加的是碎片的所有子节点，而非碎片本身。

372

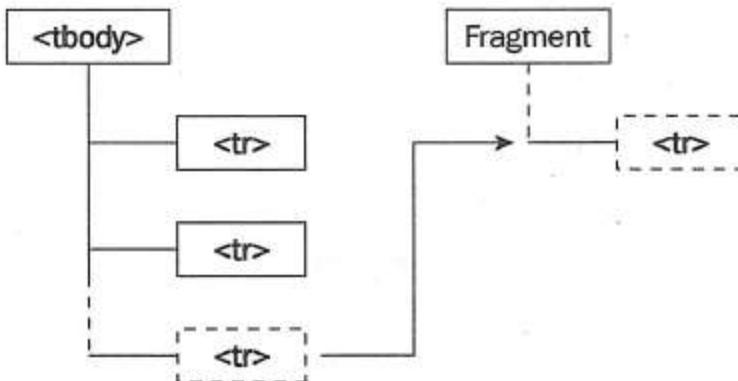


图 12-2

下面要做的就是，当用户点击表格列的头部时，调用这个函数。完成它的方式很多，最简单的就是，将函数调用放入`<th>`元素的`onclick`事件处理函数中：

```


| Last Name |
|-----------|
|-----------|


```

还要注意在`style`特性中设置了`cursor:pointer`，这可以令鼠标移动到表格头上时，鼠标

指针变成手的形状。

12.3 对多列表格进行排序

实际应用中，很少出现单列表格，所以下面的任务就是，对多列表格进行排序。假设给前面例子中的表格再添加一列，可能是用于显示某个人的名字，而不仅有姓。

```
<table border="1" id="tblSort">
    <thead>
        <tr>
            <th>Last Name</th>
            <th>First Name</th>
        </tr>
    </thead>
    <tbody>
        <tr>
            <td>Smith</td>
            <td>John</td>
        </tr>
        <tr>
            <td>Johnson</td>
            <td>Betty</td>
        </tr>
        <tr>
            <td>Henderson</td>
            <td>Nathan</td>
        </tr>
        <tr>
            <td>Williams</td>
            <td>James</td>
        </tr>
        <tr>
            <td>Gilliam</td>
            <td>Michael</td>
        </tr>
        <tr>
            <td>Walker</td>
            <td>Matthew</td>
        </tr>
    </tbody>
</table>
```

373

当然，前一节中的函数只能处理有一列的表格，必须对其进行改进。

12.3.1 比较函数生成器

在本书前面，曾经提到函数其实和 JavaScript 中其他数据类型一样，也就是说，函数也可以作为参数传递给其他函数或者作为函数的返回值。这一章中，我们已经看到将函数作为参数传递给另外一个函数（`sort()`方法）的情况；现在再来看将函数作为返回值的情况。

比较函数的很大的局限就是它要接受两个，也只能是两个参数，也就是说，不能传递额外的信息。为突破这个局限，可创建比较函数生成器，这是单独的返回一个比较函数的函数。

因为 compareTRs() 必须知道要对哪一列的值进行比较，所以需要传入额外的参数，列的索引。使用比较函数生成器，就可将额外的值传递给比较函数：

```
function generateCompareTRs(iCol) {
    return function compareTRs(oTR1, oTR2) {
        var sValue1 = oTR1.cells[iCol].firstChild.nodeValue;
        var sValue2 = oTR2.cells[iCol].firstChild.nodeValue;

        return sValue1.localeCompare(sValue2);
    };
}
```

374

generateCompareTRs() 函数只有一个参数，要处理的列的索引。返回的是和 compareTRs() 看上去几乎一样的函数。注意 iCol 参数，尽管没有在比较函数中定义，但在比较函数中用到了。这是个闭包(在第 2 章中讨论过)。变量 iCol 被比较函数捕获，因此，它被 generateCompareTRs() 返回后也可以使用。

定义好这个函数后，就可根据需要排序的列生成相应的比较函数了。

```
var compareTRs = generateCompareTRs(0);
var compareTRs1 = generateCompareTRs(1);
var compareTRs2 = generateCompareTRs(2);
```

上面的第一行代码生成的函数和前一节中定义的 compareTRs() 函数是完全一样的。第二和第三行代码生成的比较函数分别对第一和第三列进行排序。当然，其实无需将比较函数赋值给变量，可以直接传递给 sort() 方法：

```
aTRs.sort(generateCompareTRs(0));
```

实际上，这就是要修改 sortTable() 使之能在多列表格下工作的地方。

12.3.2 修改 sortTable() 方法

因为要处理多个列，sortTable() 函数必须要接受另一个表示要排序的列的索引的参数。然后，将这个值传递给 generateCompareTRs() 函数，以便对列进行正确的排序：

```
function sortTable(sTableID, iCol) {
    var oTable = document.getElementById(sTableID);
    var oTBody = oTable.tBodies[0];
    var colDataRows = oTBody.rows;
    var aTRs = new Array;

    for (var i=0; i < colDataRows.length; i++) {
        aTRs[i] = colDataRows[i];
    }

    aTRs.sort(generateCompareTRs(iCol));
```

```

var oFragment = document.createDocumentFragment();
for (var i=0; i < aTRs.length; i++) {
    oFragment.appendChild(aTRs[i]);
}

oTBody.appendChild(oFragment);
}

```

做了这两个改动后，即可给它传递要排序的列。不要忘记，表格头单元格也要进行这个改动。

375

```

<table border="1" id="tblSort">
    <thead>
        <tr>
            <th onclick="sortTable('tblSort', 0)"
                style="cursor:pointer">Last Name</th>
            <th onclick="sortTable('tblSort', 1)"
                style="cursor:pointer">First Name</th>
        </tr>
    </thead>
    <tbody>
        <!-- data rows -->
    </tbody>
</table>

```

当然，这个函数不局限于只有两列的表格。有任意数量列的表格都可以使用它，只需给函数传入正确的索引。

12.3.3 逆序排列

前两个例子中，学习了对单列和多列表格进行的排序。现在，我们要学习如何对表格列进行逆序的排序。

首先考虑对可排序表格所期望的行为。如在 Microsoft Outlook 中，点击列标题栏，就可以对某一列进行排序。如果再点击一次标题栏，就将这一列按照降序进行排列。在用户界面设计中，这个功能几乎是一个标准，所以也应该在表格中模仿这个功能。

现在已可以将每列按照升序进行排列，这就已经完成了一半。这个问题的关键在于第二次点击，这时应该按照降序排列。也就是说，为了能按照降序排列，必须已点过一次列标题栏（所以，这时列是已按照升序排列好的）。下面只要反转列的顺序（使用 `reverse()` 方法），就可以按照降序排列了。

这次，还需要修改 `sortTable()` 函数，来达到这个目的。

修改 `sortTable()` 函数

为创建降序排列，要将传递给函数的列索引保存起来以便后续使用。要实现这个目的，可以在表格上创建 `expando` 特性。`expando` 特性是在运行时为对象添加的额外的 JavaScript 特性。这个例子中的 `expando` 特性称为 `sortCol`，它用来保存最后进行排列的列索引：

```

function sortTable(sTableID, iCol) {
    var oTable = document.getElementById(sTableID);
    var oTBody = oTable.tBodies[0];
    var colDataRows = oTBody.rows;
    var aTRs = new Array;

    for (var i=0; i < colDataRows.length; i++) {
        aTRs[i] = colDataRows[i];
    }
    aTRs.sort(generateCompareTRs(iCol));

    var oFragment = document.createDocumentFragment();
    for (var i=0; i < aTRs.length; i++) {
        oFragment.appendChild(aTRs[i]);
    }

    oTBody.appendChild(oFragment);
    oTable.sortCol = iCol;
}

```

376

下面，还要添加检测列索引是否与最后一次排序的列索引相同的代码。如果相同，则应该对数组进行反转，而不是排序：

```

function sortTable(sTableID, iCol) {
    var oTable = document.getElementById(sTableID);
    var oTBody = oTable.tBodies[0];
    var colDataRows = oTBody.rows;
    var aTRs = new Array;

    for (var i=0; i < colDataRows.length; i++) {
        aTRs[i] = colDataRows[i];
    }

    if (oTable.sortCol == iCol) {
        aTRs.reverse();
    } else {
        aTRs.sort(generateCompareTRs(iCol));
    }

    var oFragment = document.createDocumentFragment();
    for (var i=0; i < aTRs.length; i++) {
        oFragment.appendChild(aTRs[i]);
    }

    oTBody.appendChild(oFragment);
    oTable.sortCol = iCol;
}

```

这样更改的好处是不需对 HTML 进行任何改动。所以，当用户第一次点击列标题栏时，就像原来一样按照升序进行排列。当用户再次点击列标题栏时，就按照降序排列。如果用户第三次点击，则顺序再一次反转，变成升序排列。不过目前这只针对字符串，那么其他数据类型呢？

12.3.4 对不同的数据类型进行排序

虽然对字符串排序是个好开头，但很多情况下，需要进行排序的列是包含各种不同的数据类型的。因为 DOM 的文本节点只是字符串值，这意味着进行任何排序之前，必须对数据进行转换。所以，要创建转换函数来完成这个任务。

1. 创建转换函数

转换函数相对比较简单，需要两个参数：要转换的字符串及表示要转换成的类型。一般来说，有三种常用的转换：转换成整数、转换成浮点数和转换成日期。如果需要字符串，当然就无需转化了。

对于转换函数，第二个参数是表示要转换成的类型的字符串：

- "int"表示转换成整数；
- "float"表示转换成浮点数；
- "date"表示转换成日期；
- 任何其他值都返回字符串。

下面是函数：

```
function convert(sValue, sDataType) {
    switch(sDataType) {
        case "int":
            return parseInt(sValue);
        case "float":
            return parseFloat(sValue);
        case "date":
            return new Date(Date.parse(sValue));
        default:
            return sValue.toString();
    }
}
```

这个函数使用switch语句来判断sDataType的值（switch语句在所有类型的JavaScript上都可使用）。当sDataType是"int"时，对sValue调用parseInt()并将结果返回；当sDataType是"float"时，对sValue调用parseFloat()并将结果返回。如果sDataType是"date"，则使用Date.parse()以及Date()构造器来创建和返回新的Date对象。如果sDataType是其他值，那么函数返回sValue.toString()，以确保返回的是字符串值。也就是说，如果sDataType是"string"、null或其他值，convert()总是返回一个字符串，例如：

```
var sValue = "25";
var iValue = convert(sValue, "int");
alert(typeof iValue); //outputs "number"
var sValue2 = convert(sValue, "string");
alert(typeof sValue2); //outputs "string"
var sValue3 = convert(sValue);
alert(typeof sValue3); //outputs "string"
var sValue4 = convert(sValue, "football");
alert(typeof sValue4); //outputs "string"
```

在这个例子中，使用 `convert()` 将字符串 "25" 转换成整数，意味着，当使用 `typeof` 时，返回值返回 "number"。但是，如果第二个参数是 "string"，返回的值的 `typeof` 则是 "string"。当第二个参数是 "football"，那么这个参数就被忽略，直接返回字符串。

完成转换函数后，必须修改其他代码来使用它。

虽然在排序时区分整数和浮点数不是必需的。大多数情况下，将所有数字转换成浮点数就已足够。本章的代码使用两种类型的数字只为达到举一反三的目的。

2. 修改代码

第一步，修改 `generateCompareTRs()` 函数，现在必须接受额外的代表进行比较的数据类型的参数。然后，从表格中取出的数值，必须在比较函数中转换成相应的数据类型。

```
function generateCompareTRs(iCol, sDataType) {
    return function compareTRs(oTR1, oTR2) {
        var vValue1 = convert(oTR1.cells[iCol].firstChild.nodeValue,
            sDataType);
        var vValue2 = convert(oTR2.cells[iCol].firstChild.nodeValue,
            sDataType);

        //...
    };
}
```

这次对 `generateCompareTRs()` 所做的修改再次利用了 JavaScript 对闭包的支持，可以直接将 `sDataType` 参数传递给比较函数。不幸的是，这次不能使用 `localeCompare()` 方法来返回正确的函数值，因为数字和日期不支持它。因为不确定存储的是何种数据类型，而且对各种数据类型进行不同的比较也不太好，所以最好使用小于和大于来判断要返回的值：

```
function generateCompareTRs(iCol, sDataType) {
    return function compareTRs(oTR1, oTR2) {
        var vValue1 = convert(oTR1.cells[iCol].firstChild.nodeValue,
            sDataType);
        var vValue2 = convert(oTR2.cells[iCol].firstChild.nodeValue,
            sDataType);

        if (vValue1 < vValue2) {
            return -1;
        } else if (vValue1 > vValue2) {
            return 1;
        } else {
            return 0;
        }
    };
}
```

使用这种方法，不管使用的是哪种数据类型，比较函数都能返回正确的值。

在这个例子中，不能使用等于符号（==）。虽然数字（整数和浮点数）和字符串都能使用。但是日期是不能使用的。记住，日期是对象，不是基本类型。也就是说等于运算符比较对象是看它们是否为同一个对象；而不是比较 Date 对象的值。然而，使用 Date 对象的 valueOf() 以及小于和大于号就可以比较它们的毫秒表示形式。

下面，修改 sortTable() 函数以使用新的比较函数生成器。函数必须接受额外的表示用于进行比较的数据类型的参数。然后，将数据类型传递给 generateCompareTRs() 函数。

```
function sortTable(sTableID, iCol, sDataType) {
    var oTable = document.getElementById(sTableID);
    var oTBody = oTable.tBodies[0];
    var colDataRows = oTBody.rows;
    var aTRs = new Array;

    for (var i=0; i < colDataRows.length; i++) {
        aTRs[i] = colDataRows[i];
    }

    aTRs.sort(generateCompareTRs(iCol, sDataType));

    var oFragment = document.createDocumentFragment();
    for (var i=0; i < aTRs.length; i++) {
        oFragment.appendChild(aTRs[i]);
    }

    oTBody.appendChild(oFragment);
    oTable.sortCol = iCol;
}
```

完成所有的 JavaScript 代码后，就要给表格添加额外的各种数据类型的数据。

```
<table border="1" id="tblSort">
    <thead>
        <tr>
            <th onclick="sortTable('tblSort', 0)"
                style="cursor:pointer">Last Name</th>
            <th onclick="sortTable('tblSort', 1)"
                style="cursor:pointer">First Name</th>
            <th onclick="sortTable('tblSort', 2, 'date')"
                style="cursor:pointer">Birthday</th>
            <th onclick="sortTable('tblSort', 3, 'int')"
                style="cursor:pointer">Siblings</th>
        </tr>
    </thead>
    <tbody>
        <tr>
            <td>Smith</td>
            <td>John</td>
            <td>7/12/1978</td>
            <td>2</td>
        </tr>
```

```

<tr>
    <td>Johnson</td>
    <td>Betty</td>
    <td>10/15/1977</td>
    <td>4</td>
</tr>
<tr>
    <td>Henderson</td>
    <td>Nathan</td>
    <td>2/25/1949</td>
    <td>1</td>
</tr>
<tr>
    <td>Williams</td>
    <td>James</td>
    <td>7/8/1980</td>
    <td>4</td>
</tr>
<tr>
    <td>Gilliam</td>
    <td>Michael</td>
    <td>7/22/1949</td>
    <td>1</td>
</tr>
<tr>
    <td>Walker</td>
    <td>Matthew</td>
    <td>1/14/2000</td>
    <td>3</td>
</tr>
</tbody>
</table>

```

注意列标题栏中的代码也被修改过，包含新的数据类型参数。对于前两列，不用修改函数调用，因为两列都只包含字符串。但是第三和第四列包含日期和整数。对于这两个列标题栏，必须加入数据类型参数。

12.3.5 高级排序

现在，已经了解如何在同一个表格中对不同的数据类型按照升序或降序进行排列。不幸的是，很少有表格只包含普通的数据类型。现实是表格中常会出现链接、图像或某种 HTML 内容；而用户仍希望对其进行排序。最常见的情况也许就是包含图标的列。图标究竟暗示什么内容（例如，电子邮件中的附件）还是仅仅用于装饰，人们希望能对它进行排序。前面的代码不支持这些内容，
[381] 不过仍可以改进。

1. 概念

注意表格中的每个单元格必须包含可排序的值，也就是说至少有一个值属于以下数据类型：字符串、整数、浮点数和日期。因为 HTML 代码不能直接被转化为这些数据类型，所以需要指定一个候选值。这可以通过给每个包含 HTML 的 `<td>` 添加额外的特性实现，像这样：

```
<td value="blue"></td>
```

因为这个单元格包含图像，一般情况下不能对其进行排序。然而，额外的 value 特性指出了要进行排序的 "blue"，而不是<td/>元素的内容。

前面也学习过，可以使用 DOM 的 getAttribute() 方法访问新特性：

```
var sValue = oTD.getAttribute("value");
```

现在，没必要对表格中的每个单元格都添加 value 特性，因为这样会多出很多冗余信息。只需给那些包含 HTML 代码的单元格添加这个特性。例如，下面的表格列出了文件名以及相关的图标，注意只有第一列才有 value 特性：

```
<table border="1" id="tblSort">
  <thead>
    <tr>
      <th>Type</th>
      <th>Filename</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td value="doc"></td>
      <td>My Resume.doc</td>
    </tr>
    <tr>
      <td value="xls"></td>
      <td>Fall Budget.xls</td>
    </tr>
    <tr>
      <td value="pdf"></td>
      <td>How to be a better programmer.pdf</td>
    </tr>
    <tr>
      <td value="doc"></td>
      <td>My Old Resume.doc</td>
    </tr>
    <tr>
      <td value="txt"></td>
      <td>Notes from Meeting.txt</td>
    </tr>
    <tr>
      <td value="zip"></td>
      <td>Backups.zip</td>
    </tr>
    <tr>
      <td value="xls"></td>
      <td>Spring Budget.xls</td>
    </tr>
    <tr>
      <td value="doc"></td>
      <td>Job Description - Web Designer.doc</td>
    </tr>
```

```

<tr>
    <td value="pdf"></td>
    <td>Saved Web Page.pdf</td>
</tr>
<tr>
    <td value="doc"></td>
    <td>Chapter 1.doc</td>
</tr>
</tbody>
</table>

```

然而，单独使用新特性不能解决这个问题。还必须更新 JavaScript 代码来利用这个 value 特性。

正如本书前面提到的，为 HTML 标签添加自定义特性在严格的 XHTML 代码表示中是不允许的。你可以通过单元格的 title 特性来提供这个值（如果这个值对客户有意义的话）或者通过在单元格内部放一不可见的<div/>来包含这个值。

2. 修改代码

对代码最后的修改是，判断是从<td/>元素的文本中读取可排序的值还是从 value 特性中读取。下面是改过的代码：

```

function generateCompareTRs(iCol, sDataType) {
    return function compareTRs(oTR1, oTR2) {
        var vValue1, vValue2;

        if (oTR1.cells[iCol].getAttribute("value")) {
            vValue1 = convert(oTR1.cells[iCol].getAttribute("value"),
                sDataType);
            vValue2 = convert(oTR2.cells[iCol].getAttribute("value"),
                sDataType);
        } else {
            vValue1 = convert(oTR1.cells[iCol].firstChild.nodeValue,
                sDataType);
            vValue2 = convert(oTR2.cells[iCol].firstChild.nodeValue,
                sDataType);
        }

        if (vValue1 < vValue2) {
            return -1;
        } else if (vValue1 > vValue2) {
            return 1;
        } else {
            return 0;
        }
    };
}

```

基本上，vValue1 和 vValue2 是没有初始值的。然后，使用 getAttribute() 检查第一行的

单元格是否有 value 特性，如果这个特性不存在，则返回 null。放在 if 语句中时，null 等同于 false，非 null 值则等同于 true。因此，如果 value 特性存在，则 vValue1 和 vValue2 都被相应分配为 oTR1 和 oTR2 的特性值。如果 value 特性不存在，则 vValue1 和 vValue2 都被分配为在表格单元格中包含的值。

最后，将排序的调用添加到 HTML 代码中：

```
<table border="1" id="tblSort">
    <thead>
        <tr>
            <th onclick="sortTable('tblSort', 0)"
                style="cursor:pointer">Type</th>
            <th onclick="sortTable('tblSort', 1)"
                style="cursor:pointer">Filename</th>
        </tr>
    </thead>
    <tbody>
        <tr>
            <td value="doc"></td>
            <td>My Resume.doc</td>
        </tr>
        <tr>
            <td value="xls"></td>
            <td>Fall Budget.xls</td>
        </tr>
        <tr>
            <td value="pdf"></td>
            <td>How to be a better programmer.pdf</td>
        </tr>
        <tr>
            <td value="doc"></td>
            <td>My Old Resume.doc</td>
        </tr>
        <tr>
            <td value="txt"></td>
            <td>Notes from Meeting.txt</td>
        </tr>
        <tr>
            <td value="zip"></td>
            <td>Backups.zip</td>
        </tr>
        <tr>
            <td value="xls"></td>
            <td>Spring Budget.xls</td>
        </tr>
        <tr>
            <td value="doc"></td>
            <td>Job Description - Web Designer.doc</td>
        </tr>
        <tr>
            <td value="pdf"></td>
            <td>Saved Web Page.pdf</td>
        </tr>
    </tbody>

```

```
</tr>
<tr>
    <td value="doc"></td>
    <td>Chapter 1.doc</td>
</tr>
</tbody>
</table>
```

这段 HTML 代码使用字符串对两列进行排序；因此，调用 `sortTable()` 时，第三个参数不是必需的。虽然第一列中包含图像，代码还是可以使用 `value` 特性来进行升序和降序排列。

12.4 小结

这一章探索了如何使用 JavaScript 将很多基于服务器端的功能转移到客户端，即对 HTML 表格进行排序。还学习了通过自定义比较函数，来自定义 `Array` 的 `sort()` 方法的排列顺序。之后，继续学习了如何对包含字符串的表格列按照升序进行排序。

下面，又学习了使用 `Array` 的 `reverse()` 方法进行逆序的排列。然后，还介绍了对包含不同值的列进行排序。在写了个小小的转化函数后，就可对表格列中的整数、浮点数和日期进行排序。最后，学会如何处理包含 HTML 而非简单文本值的单元格。

在这其中，还学习了 JavaScript 中的闭包，这可令被生成的函数包含看似不在范围内的变量。使用闭包，可以了解到看似有限的比较函数定义如何扩展成允许使用额外的数据来判断两个数中的哪个值应该首先出现。

计算机可用性的一个重大改进是拖放行为，这让用户从屏幕上的某一点拖动某个东西，然后在其他某个位置将其放下，这就创建了一种动作，例如移动对象。这种方式，最早由施乐公司开发，然后首先由 Mac OS 1.0 用于消费品技术，从此，这个功能出现在几乎所有的个人电脑的操作系统中（包括 Windows）。当首次引入动态 HTML 时，世界上的开发者们都开始尝试使用 JavaScript 来实现拖放功能。

拖放（Drag and Drop）现在可以说是可用性上的时髦词，所以将这个功能添加到网站或者是 Web 应用中就可极大地取悦消费者和客户。现在，有两种使用 JavaScript 的方法来实现拖放功能：系统拖放和模拟拖放。

13.1 系统拖放

系统拖放是操作 Windows、Macintosh 或其他图形操作系统都要用到的一个基本操作。从屏幕的某个区域将某个东西拖到另一个地方并放下。比如，要删除文件，可将其拖到垃圾箱（Trash, Macintosh 称呼）或是回收站（Recycle bin, Windows 称呼）；要将几个文件从一个文件夹移动到另一个文件夹中，只要将文件从原来的位置拖到新的位置，并将其放下。我们可使用系统拖放，因为它很好地利用了操作系统来完成任务。目前，只有一种平台上的一种 Web 浏览器支持网页上的系统拖动，那就是 Windows 上的 IE（不过 Mozilla 在基于 XUL 的页面上支持它）。

系统拖放可以在窗口和框架页之间移动，因为这个拖动行为是由操作系统处理的。还可以从 Web 浏览器中将图像拖动到桌面或另一个浏览器上。当将图像拖到桌面上后，图像就被下载了；当将其拖到另外一个浏览器中时，这个浏览器会显示它。浏览器和桌面（或者另一个浏览器）之间的通讯是由操作系统处理的。

在 IE 4.0 版中，网页上只有两个对象可以初始化系统拖放，图像或是文本。当拖动图像时，只要按住鼠标按钮然后移动；对于文本，首先要选中一些文本然后和拖动图像一样进行拖动。在 IE 4.0 中，唯一有效的放置对象就是文本框。

在 5.0 版中，IE 扩展了它的拖放功能：添加了新的事件以及让网页上几乎所有的东西都可以成为放置对象。5.5 版稍微更进一步，让几乎所有内容都可被拖动（IE 6.0 也支持这个功能）。

本节所讨论的系统拖放功能只针对 Windows 上的 IE; Macintosh 版的 IE 从未具备拖放功能, 因为它和操作系统分离了。

13.1.1 拖放事件

微软为 IE 添加的拖放事件几乎可以令你控制系统拖放操作的所有方面。比较取巧的部分是判断每一个事件在哪里触发的。有些是在被拖动的项上触发, 有些是在放置目标上触发。

1. 拖动项事件

当某个项被拖动时, 以下事件会被触发 (按顺序):

- (1) dragstart;
- (2) drag;
- (3) dragend.

当按下鼠标按钮, 开始移动鼠标时, 在被拖动的项上触发 dragstart 事件。默认情况下, 这个事件是在图像或者被选中的文本上触发。鼠标形状变成不能放置符号 (圆圈中有条线), 表示这一项不能放在自己身上。你可以使用 ondragstart 事件处理函数来在拖动开始时运行 JavaScript 代码。

在 dragstart 事件触发后, drag 事件会一直触发, 只要对象还在被拖动, 就一直触发。你可以认为这个事件和 mousemove 类似 (当鼠标移动的时候这个事件也会重复触发)。当拖动停止时 (在有效的或无效的放置对象上放下该项), 则触发 dragend 事件。

下面的例子将演示如何使用 ondragstart、ondrag 和 ondragend 事件处理函数:

```

<html>
    <head>
        <title>System Drag And Drop Example</title>
        <script type="text/javascript">
            function handleDragDropEvent(oEvent) {
                var oTextbox = document.getElementById("txt1");
                oTextbox.value += oEvent.type + "\n";
            }
        </script>
    </head>
    <body>
        <form>
            <p>Try dragging the image.</p>
            <p>
                ondragstart="handleDragDropEvent(event)"
                ondrag="handleDragDropEvent(event)"
                ondragend="handleDragDropEvent(event)" /></p>
            <p><textarea rows="10" cols="25" readonly="readonly"
                id="txt1"></textarea></p>
        </form>
    </body>
</html>

```

这个例子为图像分配了 `ondragstart`、`ondrag` 和 `ondragend` 事件处理函数。当拖动图像时，下面的文本框将在每次事件触发时显示事件。最后可以在文本框中看到如下的内容：

```
dragstart
drag
drag
drag
drag
drag
drag
dragend
```

你可以再试试这个例子，以便掌握这些事件。

除 Windows 上的 IE 外的任何浏览器上，前面这个例子中的事件处理函数都会被忽略。如果要实现系统拖放的解决方案，就要记住它。

2. 放置目标事件

当对象被拖到有效的放置目标上，会触发 `dragenter` 事件（类似于 `mouseover` 事件）。在 `dragenter` 事件发生后，将立即触发 `dragover` 事件，当被拖动的项在放置目标的边界内时，将一直触发。当项被拖出放置目标后，`dragover` 事件停止触发，并且触发 `dragleave` 事件（类似于 `mouseout`）。如果被拖动的项正是在这个目标上放下的，则会触发 `drop` 事件来替代 `dragleave` 事件。

这个例子演示了放置对象事件：

```
<html>
    <head>
        <title>System Drag And Drop Example</title>
        <script type="text/javascript">
            function handleDragDropEvent(oEvent) {
                var oTextbox = document.getElementById("txt1");
                oTextbox.value += oEvent.type + "\n";
            }
        </script>
    </head>
    <body>
        <form>
            <p>Try dragging the text from the left textbox to the right one.</p>
            <p><input type="text" value="drag this text" />
            <input type="text" ondragenter="handleDragDropEvent(event)"
                   ondragover="handleDragDropEvent(event)"
                   ondragleave="handleDragDropEvent(event)"
                   ondrop="handleDragDropEvent(event)" /></p>
            <p><textarea rows="10" cols="25" readonly="readonly"
id="txt1"></textarea></p>
        </form>
    </body>
</html>
```

前面的例子给出两个文本框来和事件进行交互，一个用来在事件发生时告诉我们是什么事件。当你将文本从左边的文本框拖到右边的文本框时，`<textarea>`事件将填入被触发的事件。如果将文本拖入文本框然后又将其拖出，则能看到如下的事件：

```
dragenter
dragover
dragover
dragover
dragover
dragover
dragleave
```

另外一种情况是，在第二个文本框中放下文本，就可看到类似于下面的事件：

```
dragenter
dragover
dragover
dragover
dragover
dragover
drop
```

注意当在第二个文本框中放下文本时，其实被选中的文本已移动到其中了。

3. 使用所有的拖放事件

处理系统拖放中比较有技巧的部分是，理解被拖动项的事件和放置目标的事件之间的关系。一般来说，被拖动项的事件总是先发生，除了 `drop` 事件要比 `dragend` 先触发。下面的例子可以让你了解这些事件之间的关系：

```
390
<html>
    <head>
        <title>System Drag And Drop Example</title>
        <script type="text/javascript">
            function handleDragDropEvent(oEvent) {
                var oTextbox = document.getElementById("txt1");
                oTextbox.value += oEvent.type + "\n";
            }
        </script>
    </head>
    <body>
        <p>Try dragging the text from the left textbox to the right one.</p>
        <form>
            <p><input type="text" value="drag this text"
                    ondragstart="handleDragDropEvent(event)"
                    ondrag="handleDragDropEvent(event)"
                    ondragend="handleDragDropEvent(event)" />
                <input type="text" ondragenter="handleDragDropEvent(event)"
                    ondragover="handleDragDropEvent(event)"
                    ondragleave="handleDragDropEvent(event)"
                    ondrop="handleDragDropEvent(event)" /></p>
            <p><textarea rows="10" cols="25" readonly="readonly"
                    id="txt1"></textarea></p>
        </form>
    </body>
</html>
```

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

你一看即知，这个例子组合了前面两个例子，同时监视被拖动项事件和放置目标事件。当从左边的文本框中拖动一些文本到右边的文本框中时，就可以看到事件像下面这样列出：

```
dragstart
drag
drag
drag
dragenter
drag
dragover
drag
dragover
drag
drop
dragend
```

注意，因为不是在放置目标上开始拖动的，所以开始只有被拖动项的事件触发了。当文本被拖到放置目标上时，dragenter 事件触发，然后是 drag，再然后是 dragover。当项始终被拖在放置目标上时，这两个事件不断被触发。当把文本放到目标上时，drop 事件触发，紧接着是 dragend 事件。这就完成了拖放过程。

如果没有把项放到放置目标上，可以看到类似下面的一系列事件：

```
dragstart
drag
drag
drag
dragenter
drag
dragover
drag
dragover
drag
dragleave
drag
drag
drag
drag
dragend
```

391

在这个例子中，将一些文本拖到右边的文本框上，然后又将其拖回，所以 dragleave 事件触发了，然后紧接着是 drag 事件，最后停止拖动文本时，dragend 事件触发。

默认情况下，文本框（`<input/>`或者`<textarea/>`）是网页上唯一有效的放置目标，不过通过改变 `dragover` 和 `dragenter` 事件的行为，可以在任何对象创建放置目标。

4. 创建自己的放置目标

当尝试将一些文本（或图像）拖动到无效的放置对象上时，会看到一个特殊的鼠标形状（圆圈中有条线），告诉你这里不能放置。尽管所有的元素都支持放置目标的事件，但默认情况下，它们的行为是不允许放置的。例如：

```

<html>
  <head>
    <title>System Drag And Drop Example</title>
    <script type="text/javascript">
      function handleDragDropEvent(oEvent) {
        var oTextbox = document.getElementById("txt1");
        oTextbox.value += oEvent.type + "\n";
      }
    </script>
  </head>
  <body>
    <p>Try dragging the text from the textbox to the red square.  

      No drop target events fire.</p>
    <form>
      <p><input type="text" value="drag this text"
          ondragstart="handleDragDropEvent(event)"
          ondrag="handleDragDropEvent(event)"
          ondragend="handleDragDropEvent(event)" />
        <div style="background-color: red; height: 100px; width: 100px"
            ondragenter="handleDragDropEvent(event)"
            ondragover="handleDragDropEvent(event)"
            ondragleave="handleDragDropEvent(event)"
            ondrop="handleDragDropEvent(event)"></div></p>
      <p><textarea rows="10" cols="25" readonly="readonly"
          id="txt1"></textarea></p>
    </form>
  </body>
</html>

```

392

在这个例子中，所有的被拖动项的事件都触发，但当文本被拖到红色的

上时，并没有触发任何放置目标事件。为能将

变成有效的放置对象，必须覆盖默认的dragenter和dragover行为。因为这些是特定于IE的，所以可直接将oEvent.returnValue特性设置为false：

```

<html>
  <head>
    <title>System Drag And Drop Example</title>
    <script type="text/javascript">
      function handleDragDropEvent(oEvent) {
        var oTextbox = document.getElementById("txt1");
        oTextbox.value += oEvent.type + "\n";

        switch(oEvent.type) {
          case "dragover":
          case "dragenter":
            oEvent.returnValue = false;
        }
      }
    </script>
  </head>
  <body>
    <p>Try dragging the text from the textbox to the red square.  

      Drop target events fire now.</p>

```

```

<form>
<p><input type="text" value="drag this text"
   ondragstart="handleDragDropEvent(event)"
   ondrag="handleDragDropEvent(event)"
   ondragend="handleDragDropEvent(event) />
<div style="background-color: red; height: 100px; width: 100px"
   ondragenter="handleDragDropEvent(event)"
   ondragover="handleDragDropEvent(event)"
   ondragleave="handleDragDropEvent(event)"
   ondrop="handleDragDropEvent(event)"></div></p>
<p><textarea rows="10" cols="25" readonly="readonly"
   id="txt1"></textarea></p>
</form>
</body>
</html>

```

在这个例子中，将文本拖到红色的

上时，鼠标形状变成带加号的指针，表示这是个有效的放置目标。默认情况下，

的 dragenter 和 dragover 事件是不允许放置的，所以如果阻止了默认的行为，就能让

变成放置目标。然后，dragenter 和 dragover 事件即被触发，dragleave 和 drop 事件也可用了。

13.1.2 数据传输对象 dataTransfer

仅仅拖放是毫无用途，除非数据能真正受到影响。为帮助数据通过拖放进行传递，IE 5.0 引入 dataTransfer 数据传输对象，它是作为 event 的一个特性出现的，是用来从被拖动的项中将字符串数据传输给放置目标的（IE 6.0 中也可以使用 dataTransfer 对象）。

因为它是 event 的一个特性，所以除了在事件处理函数的范围，dataTransfer 对象是不存在的，更明确一些，是拖放事件的事件处理函数。在事件处理函数中，可用这个对象的特性和方法来完成你要实现的拖放功能。

1. 方法

数据传输对象有两个方法：getData() 和 setData()。你可能已经猜到，getData() 用来获取由 setData() 存储的值。可以设置两种类型的数据：普通文本和 URL。setData 的第一个参数，也是 getData() 的唯一的参数，一个字符串，表示要设置哪个类型的数据，可以是 "text" 或者 "URL"。例如：

```

oEvent.dataTransfer.setData("text", "some text");
var sData = oEvent.dataTransfer.getData("text");
oEvent.dataTransfer.setData("URL", "http://www.wrox.com/");
var sURL = oEvent.dataTransfer.getData("URL");

```

393

注意，其实有两个空间可以存储数据：一个放文本，另一个放 URL。如果重复调用 setData()，会覆盖指定空间内已经存储的数据。

存储在 dataTransfer 对象中的数据在 drop 事件发生前可用。如果在 ondrop 事件处理函数中没有去获取其中的数据，那么 dataTransfer 就会被销毁，数据丢失。

当从文本框中拖动文本时，系统调用 setData() 将被拖动的文本存储在 "text" 格式中。当

它被放到目标上时，可以获取这个值。考虑以下例子：

```

<html>
  <head>
    <title>System Drag And Drop Example</title>
    <script type="text/javascript">
      function handleDragDropEvent(oEvent) {

        switch(oEvent.type) {
          case "dragover":
          case "dragenter":
            oEvent.returnValue = false;
            break;
          case "drop":
            alert(oEvent.dataTransfer.getData("text"));
        }
      }
    </script>
  </head>
  <body>
    <p>Try dragging the text from the textbox to the red square.  

    It will show you the selected text when dropped.</p>
    <p><input type="text" value="drag this text" />
    <div style="background-color: red; height: 100px; width: 100px"
      ondragenter="handleDragDropEvent(event)"
      ondragover="handleDragDropEvent(event)"
      ondrop="handleDragDropEvent(event)"></div></p>

    </body>
  </html>

```

394

除了调用了 `dataTransfer.getData()` 方法来获取正在拖动的文本，它基本上和前一个例子一样，将文本放到红色的`<div>`上时，这个页面将弹出警告框，显示正在拖动的文本。

你可能会想，如果使用参数"URL"而非"text"调用 `getData()` 会发生什么。在这个例子中，它会返回 `null` 值，因为这个数据是作为文本存储的，所以必须以文本格式获取。

如果使拖动超链接到红色的`<div>`中，那么就要使用"URL"参数来调用 `getData()` 才能正确地获取这个链接：

```

<html>
  <head>
    <title>System Drag And Drop Example</title>
    <script type="text/javascript">
      function handleDragDropEvent(oEvent) {

        switch(oEvent.type) {
          case "dragover":
          case "dragenter":
            oEvent.returnValue = false;
            break;
          case "drop":
            alert(oEvent.dataTransfer.getData("URL"));
        }
      }
    </script>
  </head>
  <body>
    <p>Try dragging the link from the textbox to the red square.  

    It will show you the selected URL when dropped.</p>
    <p><input type="text" value="drag this link" />
    <div style="background-color: red; height: 100px; width: 100px"
      ondragenter="handleDragDropEvent(event)"
      ondragover="handleDragDropEvent(event)"
      ondrop="handleDragDropEvent(event)"></div></p>

    </body>
  </html>

```

```

    }
</script>
</head>
<body>
<p>Try dragging the link to the red square.  
It will show you the URL when dropped.</p>
<p><a href="http://www.wrox.com" target="_blank">Wrox Home Page</a>
<div style="background-color: red; height: 100px; width: 100px"
     ondragenter="handleDragDropEvent(event)"
     ondragover="handleDragDropEvent(event)"
     ondrop="handleDragDropEvent(event)"></div></p>

</body>
</html>

```

当开始拖动链接时，浏览器会调用 `setData()` 并将 `href` 特性作为 URL 存储。然后用 `getData()` 获取 URL 的格式的内容，即可获取这个值。那么究竟文本格式和 URL 格式之间有何区别呢？

当指定要存储的数据是文本时，不会对其有什么特别的处理，也没有特殊的待遇。但是，如果指定要存储的数据是 URL，它会被看作是网页的一个链接，也就是说当你将它放到另一个浏览器窗口上时，浏览器会转到这个 URL。这将在后面讨论。

2. `dropEffect` 和 `effectAllowed`

`dataTransfer` 对象可实现的功能不仅仅是传输数据；它还可以用来判断，对被拖动的物体及放置目标可以做哪种类型的动作。这要通过两个特性来完成：`dropEffect` 和 `effectAllowed`。

395

`dropEffect` 特性是设置在放置目标上的，用来确定允许哪种类型的放置行为，有四种可能的值：

- "none"——被拖动的项不能放在这个地方。这是除文本框之外的所有对象的默认值。
- "move"——表示被拖动的项应该移动到放置对象上。
- "copy"——表示被拖动的项应该复制到放置对象上。
- "link"——表示放置对象将会浏览到被拖动的项（是 URL 时才有效）。

当某个项拖到放置目标上时，这其中每个值都会显示不同的鼠标形状。然而，真正产生什么行为，还是由你决定。换句话说，没有你的指示，不会有任何东西会自动移动、复制或链接。现在唯一能做的就是自由改变鼠标的外观。如果使用 `dropEffect` 特性，必须在放置对象的 `ondragenter` 事件处理函数中对其进行设置。

除非还设置了被拖放项的 `effectAllowed` 特性，否则 `dropEffect` 也没什么用。这个特性表示对被拖动的项，允许哪种 `dropEffect`。可能的值如下：

- "uninitialized"——没有为被拖动的项设置任何动作。
- "none"——在被拖动的项上不允许有任何动作。
- "copy"——只允许 `dropEffect` "copy"。
- "link"——只允许 `dropEffect` "link"。
- "move"——只允许 `dropEffect` "move"。
- "copyLink"——只允许 `dropEffect` "copy" 和 "link"。

- "copyMove"—只允许 dropEffect "copy" 和 "move"。
- "linkMove"—只允许 dropEffect "link" 和 "move"。
- "all"—允许所有的 dropEffect。

这个特性必须在 ondragstart 事件处理函数中设置。

假设让用户将文本从一个文本框拖入

中。必须同时设置 dropEffect 和 effectAllowed 为 "move"。不过，文本也不会自动移动自己，因为

的放置事件的默认行为是什么也不做。如果覆盖了默认的行为，文本会自动从原来的文本框中删除。将文本插入到

中的事件要由你来完成，这可以使用 innerHTML 特性：

```

<html>
  <head>
    <title>System Drag And Drop Example</title>
    <script type="text/javascript">
      function handleDragDropEvent(oEvent) {

        switch(oEvent.type) {
          case "dragstart":
            oEvent.dataTransfer.effectAllowed = "move";
            break;

          case "dragenter":
            oEvent.dataTransfer.dropEffect = "move";
            oEvent.returnValue = false;
            break;

          case "dragover":
            oEvent.returnValue = false;
            break;

          case "drop":
            oEvent.returnValue = false;
            oEvent.srcElement.innerHTML =
              oEvent.dataTransfer.getData("text");
        }
      }
    </script>
  </head>
  <body>
    <p>Try dragging the text in the textbox to the red square.
    The text will be "moved" to the red square.</p>
    <p><input type="text" value="drag this text"
      ondragstart="handleDragDropEvent(event)" />
    <div style="background-color: red; height: 100px; width: 100px"
      ondragenter="handleDragDropEvent(event)"
      ondragover="handleDragDropEvent(event)"
      ondrop="handleDragDropEvent(event)"></div>
    </p>

  </body>
</html>

```

在这个例子中，可以从文本框中拖动文本，并将放入红色的

中。文本从原来的文本框中删除，因为放置事件的默认行为被覆盖了。然后使用 innerHTML 特性，将文本插入到

中。

如果将 dropEffect 和 effectAllowed 改成 "copy"，文本框中的文本不变，在

中产生一个副本。

13.1.3 dragDrop()方法

已经知道了如何创建自己的放置目标，那么现在应该学习创建自己的可拖动对象。在 IE 5.5 中，dragDrop()方法可以用于几乎所有的 HTML 元素。可以通过调用 dragDrop()来初始化系统拖动事件，这样就可让一般情况下不能拖动的项触发 dragStart、drag 和 dragend 事件。

关键问题是调用 dragDrop()的时机。最好使用 onmousemove 事件处理函数来初始化拖动，像这样：

```
oElement.onmousemove = function (oEvent) {
    if (oEvent.button == 1) {
        oElement.dragDrop();
    }
};
```

397

使用 event.button 特性，可确保在鼠标移动的时候按下左键，通常情况下这时开始拖动对象。

下一步是在元素的 ondragstart 事件处理函数中使用 dataTransfer 对象来确定被拖动项的动作。例如，当将一个

拖动到文本框时，插入文本：

```
<html>
    <head>
        <title>System Drag And Drop Example</title>
        <script type="text/javascript">
            function handleMouseMove(oEvent) {
                if (oEvent.button == 1) {
                    oEvent.srcElement.dragDrop();
                }
            }

            function handleDragDropEvent(oEvent) {
                oEvent.dataTransfer.setData("text", "This is a red square");
            }
        </script>
    </head>
    <body>
        <p>Try dragging the red square into the textbox.</p>
        <p><div style="background-color: red; height: 100px; width: 100px"
            onmousemove="handleMouseMove(event)"
            ondragstart="handleDragDropEvent(event)">This is a red square</div>
        </p>
        <form>
            <p><input type="text" value="" /></p>
        </form>
    </body>
</html>
```

当拖动例子中红色的<div>时，文本"This is a red square"将放入 dataTransfer 对象中。这样如果方块放入文本框中，马上就可插入了这段文字，就像从另一个文本框中拖过来的一样。甚至可以拖动红色的<div>到另一个浏览器窗口的文本框中，那时会发生同样的事情。

如果还存储了 URL 而不仅是文本，还可以将红色方块拖到另一个浏览器窗口中，进而让浏览器转到指定的页面：

```

398 <html>
      <head>
        <title>System Drag And Drop Example</title>
        <script type="text/javascript">

          function handleMouseMove(oEvent) {
            if (oEvent.button == 1) {
              oEvent.srcElement.dragDrop();
            }
          }

          function handleDragDropEvent(oEvent) {
            oEvent.dataTransfer.setData("URL", "http://www.wrox.com/");
          }
        </script>
      </head>
      <body>
        <p>Try dragging the red square into another browser window.</p>
        <p><div style="background-color: red; height: 100px; width: 100px"
               onmousemove="handleMouseMove(event)"
               ondragstart="handleDragDropEvent(event)">http://www.wrox.com</div>
        </p>
      </body>
    </html>

```

这个例子中，将红色的<div>拖动到另一个浏览器窗口中，就会让这个浏览器跳转到 Wrox 的主页 <http://www.wrox.com>。

13.1.4 优点及缺点

显然，IE 的系统拖动是个很强大的功能。在框架、浏览器窗口之间拖动信息的能力，为 JavaScript 开发人员开启了一扇通向充满无限可能的新世界的大门。然而，必须使用 Windows 上的 IE 5.0 或者更高版本。

如果需要开发可以在多个浏览器上运行的拖放程序，系统拖放的路子就走不通了。尽管这个方法易于使用，但没有任何其他浏览器开发了这种功能，而且貌似在近期也不会有所改变。对于那些只针对 IE 开发的人来说，这个解决方案用起来很好；但对于其他的人，答案就是模拟鼠标拖放功能。

13.2 模拟拖放

模拟拖放早在 IE 4.0 和 Netscape Navigator 4.0 引入对动态 HTML 的支持时就出现了。基本

思想很简单：创建可以跟着鼠标移动的绝对定位层。在实际应用中，要令用户获得完全真实的拖动感觉还是有些复杂的。

这类拖放方法是对经典的鼠标拖尾脚本的扩展。你可能还记得，打开某些页面，鼠标指针后面会跟着一个图像。这个很容易实现：

```
<html>
  <head>
    <title>Simulated Drag And Drop Example</title>
    <script type="text/javascript">
      function handleMouseMove(oEvent) {
        var oDiv = document.getElementById("div1");
        oDiv.style.left = oEvent.clientX;
        oDiv.style.top = oEvent.clientY;
      }
    </script>
    <style type="text/css">
      #div1 {
        background-color: red;
        height: 100px;
        width: 100px;
        position: absolute;
      }
    </style>
  </head>
  <body onmousemove="handleMouseMove(event)">
    <p>Try moving your mouse around.</p>
    <p><div id="div1"></div> </p>
  </body>
</html>
```

这个例子中，红色的

随着鼠标指针移动。一旦鼠标移动，就将

定位在同样的位置，让它的左上角坐标等于鼠标指针的坐标。注意 onmousemove 事件处理函数是分配给<body>元素的，而不是<div>元素。这是因为必须在整个页面上跟踪鼠标的移动，而不仅在<div>内。

要模拟拖放，<div>仅移动一下是不够的；必须可以初始化和停止拖动过程。这就是为何代码有些复杂，同时还需要用到本书前面的 EventUtil 对象。

13.2.1 代码

第一步是创建用来处理三个鼠标事件（mousemove，mousedown 和 mouseup）的三个函数。处理 mousedown 的函数是分配给<div>的（或者是要被拖动的元素）。当用户的鼠标在<div>上按下时，函数将mousemove 和 mouseup 的两个事件处理函数分配给 document.body。当用户松开鼠标时，拖动停止，同时删除mousemove 和 mouseup 的事件处理函数。代码如下：

```
<html>
  <head>
    <title>Simulated Drag And Drop Example</title>
    <script type="text/javascript" src="eventutil.js"></script>
```

400

```

<script type="text/javascript">

    function handleMouseMove() {
        var oEvent = EventUtil.getEvent();
        var oDiv = document.getElementById("div1");
        oDiv.style.left = oEvent.clientX;
        oDiv.style.top = oEvent.clientY;
    }

    function handleMouseDown() {
        EventUtil.addHandler(document.body, "mousemove",
            handleMouseMove);
        EventUtil.addHandler(document.body, "mouseup",
            handleMouseUp);
    }

    function handleMouseUp() {
        EventUtil.removeEventHandler(document.body, "mousemove",
            handleMouseMove);
        EventUtil.removeEventHandler(document.body, "mouseup",
            handleMouseUp);
    }

</script>
<style type="text/css">
    #div1 {
        background-color: red;
        height: 100px;
        width: 100px;
        position: absolute;
    }
</style>
</head>
<body>
    <p>Try dragging the red square.</p>
    <p><div id="div1" onmousedown="handleMouseDown()"></div> </p>
</body>
</html>

```

可以看到, handleMouseDown() 仅是添加了 onmousemove 和 onmouseup 事件处理函数, 而 handleMouseUp() 则是删除这些事件处理函数。这样做可防止拖动功能工作错误。在这个例子中, 可以初始化拖动的唯一事件是用户在<div/>上按住鼠标按键。

401

当尝试运行这段代码时, 注意<div/>的左上角总是和鼠标指针平齐, 这没什么大碍, 但会对用户产生一些困扰。理想状态下, <div/>应该看上去是被鼠标指针抓起一样, 也就是说鼠标指针应该一直在用户按下鼠标的那个点上, 不管<div/>怎么移动(见图 13-1)。

要让鼠标显示在正确的位置上只需稍微做些计算就行了。首先点击时(在 handleMouseDown() 函数中)取得<div/>的位置和鼠标位置之间的差值。将事件对象的 clientX 和 clientY 特性与<div/>的 offsetLeft 和 offsetTop 特性进行比较(见图 13-2)。

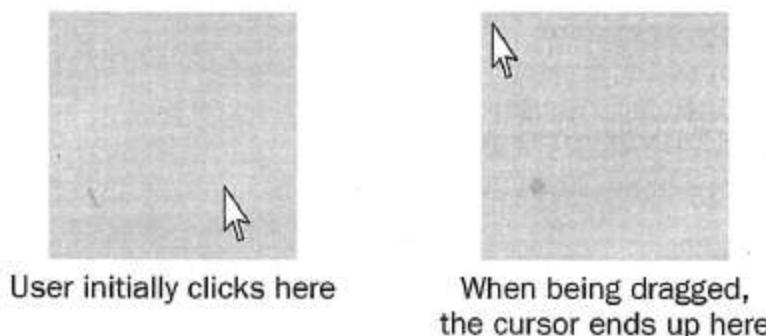


图 13-1

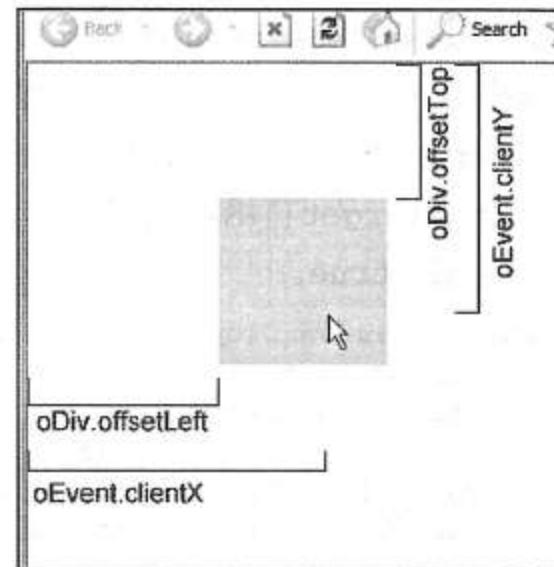


图 13-2

x 和 y 坐标的差值必须存储在函数之外，这样其他的函数才能访问到这些值。存储这两个差值的变量分别称为 iDiffX 和 iDiffY，它们在 handleMouseDown() 函数中初始化，在其他两个事件处理函数添加到 document.body 之前。然后，在 handleMouseMove() 中，将 clientX 和 clientY 相应减去这些值，就可让<div>移动到正确的位置上了。

```

var iDiffX = 0;
var iDiffY = 0;

function handleMouseMove() {
    var oEvent = EventUtil.getEvent();
    var oDiv = document.getElementById("div1");
    oDiv.style.left = oEvent.clientX - iDiffX;
    oDiv.style.top = oEvent.clientY - iDiffY;
}

function handleMouseDown() {
    var oEvent = EventUtil.getEvent();
    var oDiv = document.getElementById("div1");
    iDiffX = oEvent.clientX - oDiv.offsetLeft;
    iDiffY = oEvent.clientY - oDiv.offsetTop;

    EventUtil.addHandler(document.body, "mousemove", handleMouseMove);
    EventUtil.addHandler(document.body, "mouseup", handleMouseUp);
}

function handleMouseUp() {
    EventUtil.removeHandler(document.body, "mousemove", handleMouseMove);
    EventUtil.removeHandler(document.body, "mouseup", handleMouseUp);
}

```

402

做这些改动后就可以让<div>按照比较好看的方式进行拖动。

13.2.2 创建放置目标

现在已知道如何在屏幕上拖动元素，还需要一个放置它的地方。为模拟拖放创建放置目标，

需要检测鼠标的坐标是否位于放置目标的边界内。

与被拖动的项一样，放置对象也必须是绝对定位的，这样才能使用元素的 offsetLeft、offsetTop、offsetHeight 和 offsetWidth 特性来判断每个角的 x 和 y 坐标。我们设计一个 isOverDropTarget() 函数来封装这个计算过程，它接受一个坐标作为参数，如果坐标在放置目标内，则返回 true。

```
function isOverDropTarget(iX, iY) {
    var oTarget = document.getElementById("divDropTarget");
    var iX1 = oTarget.offsetLeft;
    var iX2 = iX1 + oTarget.offsetWidth;
    var iY1 = oTarget.offsetTop;
    var iY2 = iY1 + oTarget.offsetHeight;

    return (iX >= iX1 && iX <= iX2 && iY >= iY1 && iY <= iY2);
}
```

函数的第一行代码获取 ID 为 "divDropTarget" 元素的引用。接下来一行代码判断两个 x 坐标和两个 y 坐标。最后一行检查坐标参数是否位于放置目标内，如果是返回 true，不是则返回 false。

```
function handleMouseUp() {
    var oEvent = EventUtil.getEvent();
    EventUtil.removeEventHandler(document.body, "mousemove", handleMouseMove);
    EventUtil.removeEventHandler(document.body, "mouseup", handleMouseUp);

    if (isOverDropTarget(oEvent.clientX, oEvent.clientY)) {
        alert("dropped!");
        var oDiv = document.getElementById("divDropTarget");
        var oTarget = document.getElementById("div2");
        oDiv.style.left = oTarget.offsetLeft;
        oDiv.style.top = oTarget.offsetTop;
    }
}
```

代码中突出显示的部分表示如果鼠标在放置目标上则弹出警告框。然后，被拖动的元素就被定位于放置目标的左上角，这样看上去好像它停靠在其中了。当然，如果有多个内容要拖到放置目标上，必须创建某种逻辑来确保对象被合理地排列起来，不过这些已经足够我们上手了。下面是完整的代码：

```
<html>
<head>
    <title>Simulated Drag And Drop Example</title>
    <script type="text/javascript" src="eventutil.js"></script>
    <script type="text/javascript">

        var iDiffX = 0;
        var iDiffY = 0;

        function handleMouseMove() {
            var oEvent = EventUtil.getEvent();
            var oDiv = document.getElementById("div1");

```

```

        oDiv.style.left = oEvent.clientX - iDiffX;
        oDiv.style.top = oEvent.clientY - iDiffY;
    }

    function handleMouseDown() {
        var oEvent = EventUtil.getEvent();
        var oDiv = document.getElementById("div1");
        iDiffX = oEvent.clientX - oDiv.offsetLeft;
        iDiffY = oEvent.clientY - oDiv.offsetTop;

        EventUtil.addHandler(document.body, "mousemove",
handleMouseMove);
        EventUtil.addHandler(document.body, "mouseup", handleMouseUp);
    }

    function handleMouseUp() {
        var oEvent = EventUtil.getEvent();
        EventUtil.removeHandler(document.body, "mousemove",
handleMouseMove);
        EventUtil.removeHandler(document.body, "mouseup",
handleMouseUp);

        if (isOverDropTarget(oEvent.clientX, oEvent.clientY)) {
            alert("dropped!");
            var oDiv = document.getElementById("div1");
            var oTarget = document.getElementById("divDropTarget");
            oDiv.style.left = oTarget.offsetLeft;
            oDiv.style.top = oTarget.offsetTop;
        }
    }

    function isOverDropTarget(iX, iY) {
        var oTarget = document.getElementById("divDropTarget");
        var iX1 = oTarget.offsetLeft;
        var iX2 = iX1 + oTarget.offsetWidth;
        var iY1 = oTarget.offsetTop;
        var iY2 = iY1 + oTarget.offsetHeight;
        return (iX >= iX1 && iX <= iX2 && iY >= iY1 && iY <= iY2);
    }

</script>
<style type="text/css">
    #div1 {
        background-color: red;
        height: 100px;
        width: 100px;
        position: absolute;
        z-index: 10;
    }

    #divDropTarget {
        background-color: blue;
        height: 200px;
        width: 200px;
    }

```

```

        position: absolute;
        left: 300px;
    }

```

```
</style>
```

```
</head>
```

```
<body>
```

```
    <p>Try dragging the red square onto the blue square.</p>
```

```
    <div id="div1" onmousedown="handleMouseDown(event)"></div>
```

```
    <div id="divDropTarget"></div>
```

```
</body>
```

```
</html>
```

13.2.3 优点及缺点

模拟拖放的最大的优点是可在所有的 DOM 兼容的浏览器中运行，如 IE 5.0+、Mozilla 1.0+、Safari 1.0+ 和 Opera 7.0+。因为这个策略仅使用了基本的 DOM 功能，所以它可在很多平台上运行。

当然，模拟拖放不能与操作系统挂钩，而系统拖放则可以。我们不能影响用户拖动的文本和链接，被拖动的元素也只能在指定的窗口和框架中拖动。然而，对于大部分使用情况来说，模拟拖放已经可以完成这个使命了。

13.3 zDragDrop

可以看到，模拟拖放需要写不少 JavaScript 代码。有读者可能已经在想，是否已经有某些 JavaScript 库提供了对它的处理？答案是肯定的。有一个 zDragDrop 库，可以从 <http://www.nczonline.net/downloads/> 上免费下载。这个库提供一些封装了很多模拟拖放过程的对象。只要将 zdragdroplib.js 包含在页面中，就可以使用这个功能。

405

13.3.1 创建可拖动元素

zDragDrop 类可用于使任何绝对定位的 DOM 元素变得可拖动。这个构造器接受两个参数，要处理的 DOM 元素和一系列决定元素如何被拖动的约束条件。第二个参数可由一个或多个特殊值组成。

要让元素只能在水平方向拖动，使用 zDraggable.DRAG_X 作为第二个参数：

```
var oDiv = document.getElementById("divToDrag");
var oDraggable = new zDraggable(oDiv, zDraggable.DRAG_X);
```

要让元素只能在垂直方向拖动，使用 zDraggable.DRAG_Y 作为第二个参数：

```
var oDiv = document.getElementById("divToDrag");
var oDraggable = new zDraggable(oDiv, zDraggable.DRAG_Y);
```

如果要两元素在两个方向都可以拖动，使用二进制 OR 运算符来组合两个值：

```
var oDiv = document.getElementById("divToDrag");
var oDraggable = new zDraggable(oDiv, zDraggable.DRAG_X | zDraggable.DRAG_Y);
```

因为 zDraggable 构造器期望将 DOM 元素作为第一个参数，所以只有在页面完全载入后才能使用这些代码。

13.3.2 创建放置目标

使用 zDragDrop 库，必须为可拖动元素明确设置放置目标。首先，这样创建一个 zDropTarget 对象：

```
var oDivTarget = document.getElementById("divDropTarget");
var oDropTarget = new zDropTarget(oDiv);
```

在创建放置目标后，可以使用 addDropTarget() 方法将其添加到可拖动元素中。例如：

```
var oDivToDrag = document.getElementById("divToDrag");
var oDivTarget = document.getElementById("divDropTarget");

var oDraggable = new zDraggable(oDiv, zDraggable.DRAG_X | zDraggable.DRAG_Y);
var oDropTarget = new zDropTarget(oDiv);

oDraggable.addDropTarget(oDropTarget);
```

有了这段代码，现在即可使用 zDragDrop 库的内置事件来管理拖动和释放。

13.3.3 事件

虽然不能与 IE 的拖放事件的健壮性相比，zDragDrop 库确实提供了一些基本的可以令处理模拟拖放更容易的事件。

406

zDraggable 对象支持以下三种事件：

- dragstart——在开始拖动元素之前立刻发生。
- drag——在拖动元素时持续发生。
- dragend——在停止拖动元素之后触发，不管有没有放在有效的放置目标上。

zDropTarget 对象只支持一个事件 drop，当可拖动对象放置其上时发生。

为了使用这些事件，zDragDrop 库使用了 DOM 风格的事件处理函数分配方式，使用 addEventListener() 方法。而与 DOM 不同的是，它只有两个参数：要处理的事件的类型和事件处理函数。例如下面的代码将事件处理函数分配给一个 zDropTarget 对象，当放入一个物体时，显示信息 "dropped"：

```
oDropTarget.addEventListener("drop", function () {
    alert("dropped");
});
```

也可以使用同样的参数调用 removeEventListener() 来删除事件处理函数：

```
function handleDragEnd() {
    alert("drag end");
}
```

```

oDraggable.addEventListener("dragend", handleDragEnd);

//other code here

```

```

oDraggable.removeEventListener("dragend", handleDragEnd);

```

zDragDrop 库也支持 `event` 对象包含一些额外的信息。这个 `event` 对象的特性是：

- `type`——发生的事件的类型（如“`drag`”或者“`dragend`”）。
- `target`——引发事件的对象（`zDraggable` 或者 `zDropTarget`）。
- `timestamp`——事件发生时的时间和日期，以毫秒为单位。
- `relatedTarget`——和事件相关的另一个对象。当在 `zDropTarget` 上触发 `drop` 事件时，这个特性始终等于放入的 `zDraggable` 对象。
- `cancelable`——表示事件是否可被取消。

另外，`event` 对象还支持方法 `preventDefault()`，用来阻止事件的默认行为。目前唯一可以被阻止的事件是 `dragstart`。

407

这个事件对象作为唯一的参数传给事件处理函数，可以这样访问它：

```

oDraggable.addEventListener("dragstart", function (oEvent) {
    alert(oEvent.type + " occurred at " + oEvent.timeStamp);
    oEvent.preventDefault();
});

```

13.3.4 例子

为将前面一节中的例子重写，要使用 `zDragDrop` 库中的一些其他方法：

- `zDraggable.moveTo(x, y)`——将 `zDraggable` 元素移动到位置 `x, y`。
- `zDropTarget.getLeft()`——返回放置目标的左坐标。
- `zDropTarget.getTop()`——返回放置坐标的顶坐标。

如果使用这些方法以及 `zDragDrop` 的事件处理功能，那么代码会变得更加简洁：

```

<html>
    <head>
        <title>Simulated Drag And Drop Example</title>
        <script type="text/javascript" src="zdragdroplib.js"></script>
        <script type="text/javascript">

            function doLoad() {
                var oDraggable = new zDraggable(document.getElementById("div1"),
                    zDraggable.DRAG_X | zDraggable.DRAG_Y);
                var oDropTarget =
                    new zDropTarget(document.getElementById("divDropTarget"));

                oDraggable.addDropTarget(oDropTarget);

                oDropTarget.addEventListener("drop", function (oEvent) {
                    oEvent.relatedTarget.moveTo(oDropTarget.getLeft(),
                        oDropTarget.getTop());
                });
            }
        </script>
    </head>
    <body>
        <div id="div1" style="border: 1px solid black; width: 100px; height: 100px; margin: 10px; float: left; position: relative; background-color: #f0f0f0;"></div>
        <div id="divDropTarget" style="border: 1px solid black; width: 100px; height: 100px; margin: 10px; float: left; position: relative; background-color: #e0e0e0;"></div>
        <div style="clear: both; margin-top: 10px; font-size: 1.2em; font-weight: bold; color: #0000ff; text-align: center; position: absolute; top: 0; left: 0; width: 100%; height: 100%; z-index: 1; background-color: black; opacity: 0.5; filter: alpha(opacity=50);">Simulated Drag And Drop Example</div>
    </body>
</html>

```

```

        });
    }

</script>
<style type="text/css">
    #div1 {
        background-color: red;
        height: 100px;
        width: 100px;
        position: absolute;
        z-index: 10;
    }

    #divDropTarget {
        background-color: blue;
        height: 200px;
        width: 200px;
        position: absolute;
        left: 300px;
    }
</style>
</head>
<body onload="doLoad()">
    <p>Try dragging the red square onto the blue square.</p>
    <div id="div1"></div>
    <div id="divDropTarget"></div>
</body>
</html>

```

408

可以看到，这段代码中的 JavaScript 部分明显比前一节中的要少很多。前两行 JavaScript 创建 zDraggable 和 zDropTarget 对象。然后，使用 addDropTarget 将放置目标注册到 zDraggable 对象上。最后，将 drop 事件的处理函数添加到放置目标上。这个事件处理函数用前面提到的方法，将可拖动对象移动到放置目标的左上角。记住，在 drop 事件中，event 对象的 relatedTarget 特性等于可拖动元素。

当然，这些代码必须在页面完全载入后才能调用，为此使用了 onload 事件处理函数。

13.4 小结

这一章介绍了 Web 浏览器中拖放的概念，并解释了系统拖放和模拟拖放之间的区别。

还学习了 IE 内建的系统拖放功能，而且 IE 是唯一支持网页上系统拖放的浏览器。IE 提供的用于处理系统拖放的各种事件和方法也做了一一讨论，同时还讲到拖动文本和链接的策略。

接下来学习了模拟拖放，一种用 DOM 移动元素的方式，模拟了拖放过程。同时还演示了如何构建一个简单的拖放例子。

最后，我们介绍了 zDragDop 库，一个免费的 JavaScript 库，封装了大量的模拟拖放功能。还学了如何使用这个库来创建一个拖放的例子。

409

410

以前, JavaScript 总是因为会出现一些令人困惑的错误信息(例如 Object Expected 和 Illegal Syntax, 外加一个指定出处的行号, 除此之外就没有其他信息了)而知名。调试这种信息确实是一种痛苦的经历, 因此, 第四版的浏览器(IE 4.0 和 Netscape Navigator 4.0)包含了一些基本的错误处理功能。不久之后, ECMA 在 ECMAScript 第三版中提出了新的解决方案。

最新的 ECMAScript 添加了异常处理机制, 采用了从 Java 中移植过来的模型。ECMAScript 第三版用 ECMAScript 第二版中的一些保留字实现了 try...catch...finally 结构以及 throw 操作符。

本章将探索基于浏览器的错误处理机制以及 ECMAScript 的异常处理特性。

14.1 错误处理的重要性

早期的浏览器(比如 IE 3.0 和 Netscape Navigator 3.0 中)没有错误处理。函数通过返回一个无效值(一般是 null、false 或者是 -1, 根据不同的函数不同)来表示发生了错误, 考虑以下代码:

```
var iLoc = findItem(colorarray, "blue");
if (iLoc == -1) {
    alert("The item doesn't exist.");
} else {
    alert("The item is in location " + iLoc + ".");
}
```

在此例子中, 给定的字符串不存在时, 函数 findItem() 返回一个 -1(一个无效值)。这么看来, 有效值应该是大于或等于 0 的数字。但是函数为什么要返回一个无效值呢? 是否只是字符串 "blue" 不在 colorarray 中? 或者还有其他原因, 也许数组中一个项也没有? 无法知道为何返回这个无效值。错误和错误处理将帮助我们解决这个难题。

在 JavaScript 中引入了错误处理后, Web 开发人员就可以更好地对代码进行控制了。好的错误处理技术也可以让脚本的开发、调试和部署更流畅。在很多情况下, 这类技术实际上在 JavaScript 中要比在其他编程语言更加重要, 因为它缺乏标准的开发环境来指导开发人员, 不像.NET Framework 开发有 Visual Studio.NET, Java 开发有 NetBeans。

14.2 错误和异常

论及编程中的错误时，需要关心两类主要错误：语法错误和运行时错误。

语法错误，也称为解析错误，发生在传统编程语言的编译时，在JavaScript中发生在解释时。这些错误是由代码中的意外字符直接造成的，然后就不能完全编译/解释。例如，下面一行代码因缺少一个右括号，产生了语法错误。

```
document.write("test";
```

发生语法错误时，就不能继续执行代码。在JavaScript中，只有在同一个线程中的代码会受语法错误的影响。在其他线程中的代码和其他外部引用的文件中的代码，如果不依赖于包含错误的代码，则可以继续执行。

运行时错误，也称为异常（exception，在编译期/解释期后）。此时，问题并不出在代码的语法上。而是，尝试完成的一个操作，在某些情况下是非法的。例如：

```
window.openMySpecialWindow();
```

在上面的代码中，尝试访问 window 对象的 openMySpecialWindow 方法。语法上这行代码是正确的，但是并不存在这个方法。浏览器就会返回一个异常。

异常只影响它们发生的线程，其他 JavaScript 线程则可继续正常的执行。考虑下面的 HTML 页面：

```
<html>
    <head>
        <title>Exception Test</title>
        <script type="text/javascript">
            function handleLoad() {
                window.openMySpecialWindow();
                alert("Loaded");
            }

            function handleClick() {
                alert("Clicked");
            }
        </script>
    </head>
    <body onload="handleLoad()">
        <input type="button" value="Click Me" onclick="handleClick()" />
    </body>
</html>
```

412

这个页面载入时，尝试调用 handleLoad() 函数，当尝试调用 window 对象的一个不存在的方法时，就发生一个异常。然后该线程退出，所以 alert("Loaded"); 不会被执行。然而，用户点击按钮时，还可以调用 handleClick() 函数，并出现警告框，显示"Clicked"。

14.3 错误报告

因为每个浏览器都有自己的内置 JavaScript 解释程序（包含它自己的错误跟踪机制），所以每种浏览器报告错误的方式都不同：有些是弹出错误信息，有些是仅把信息记录在 JavaScript 控制台中。像 IE、Mozilla、Safari 和 Opera 这些浏览器，都有自己独特的显示错误数据的方式。这一节将指导你如何在各个浏览器上找到错误信息。

14.3.1 IE (Windows)

微软的旗舰浏览器，Windows 上的 IE，可以以几种方式来报告错误。默认情况下，IE 弹出包含错误细节的对话框，并询问是否继续执行页面上的脚本。其实这很误导人，因为不管点击 Yes 还是 No，都会允许其他脚本继续执行。

如果浏览器有个调试器（比如 Microsoft Script Debugger，本章后面会介绍），这个对话框会提供一个是调试还是忽略的选项（图 14-2）。点击 Yes 会打开调试器；点击 No 就简单地关闭对话框。



图 14-1

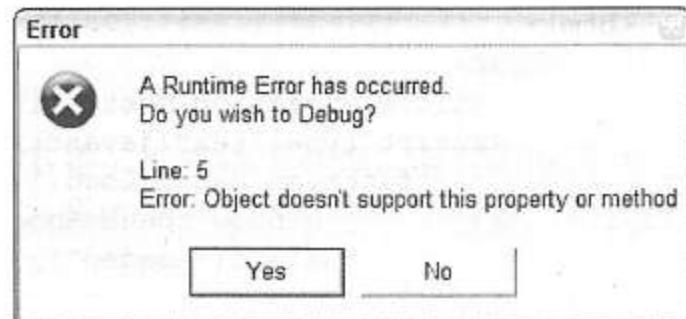


图 14-2

还有一个不要弹出错误对话框的选项。这个设置可在 Internet Options (Internet 选项) 对话框中启用，点击 Advanced (高级) 标签页，取消勾选“Display a notification about every script error”（图 14-3）。

采用这个设置后，一旦出现错误，IE 将在左下角显示一个黄色的图标，上面有一个 X（图 14-4）。双击这个图标就会弹出显示典型错误信息的对话框（消息、URL 和行号）。这个对话框也可用于（通过使用 Previous 按钮）查看显示的错误之前发生的错误。

要小心，它所给出的行号不是绝对精确的。如果产生错误的代码在 HTML 页面内，则这个行号可以正确对应发生错误的那一行。但是如果发生错误的代码在外部文件中，则行号往往要差一行，也就是如果一个错误说是发生在第 5 行其实是发生在第 4 行的。

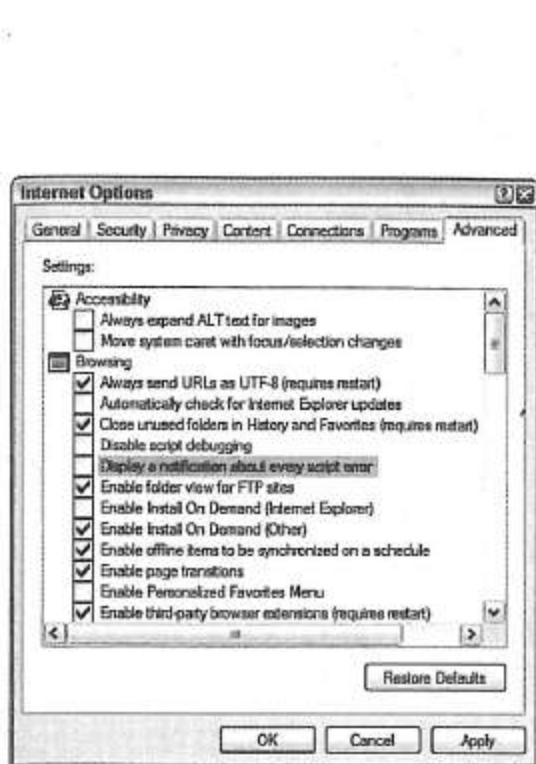


图 14-3

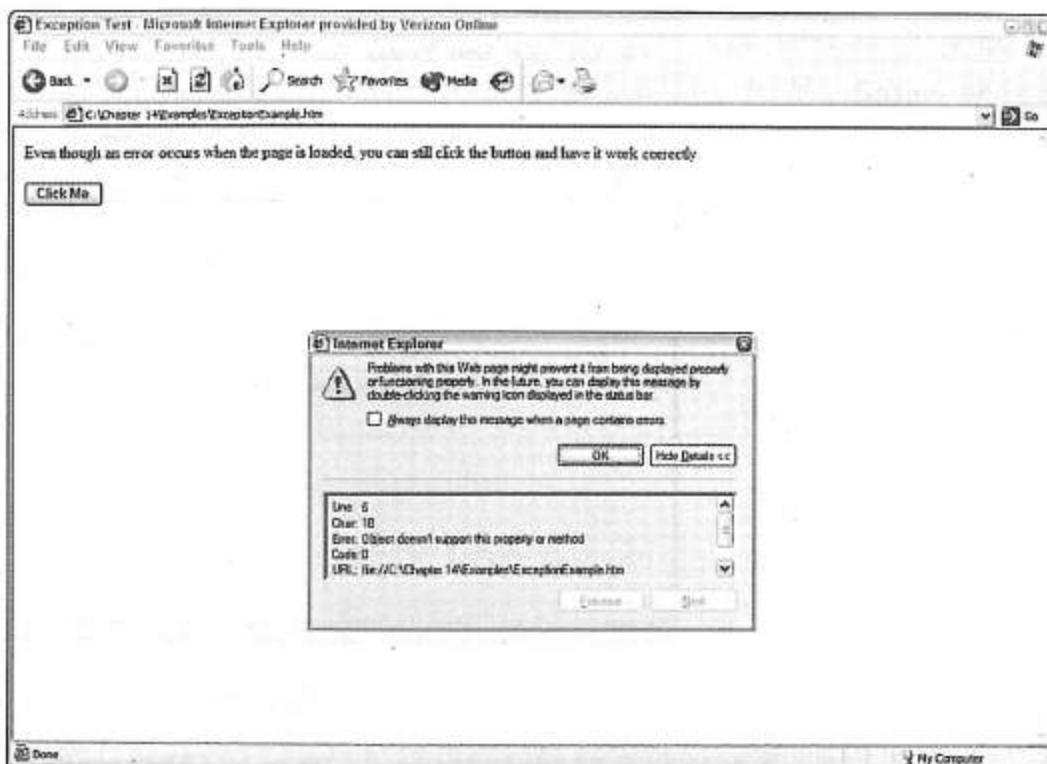


图 14-4

14.3.2 IE (MacOS)

默认情况下，MacOS 上的 IE 不会显示 JavaScript 错误。可在 Preference 对话框中 (Edit → Preferences) 改变这一设置。在 Web Browser 下的窗口的左边，点击 Web Content。在窗口下面的部分中，可看到一个标题为 Show Scripting Error Alerts 的复选框。勾选这个框可令浏览器在出现 JavaScript 错误时弹出错误信息。

IE/Mac 的错误提示的问题是，它不能给出精确的外部脚本的行号（行号一般是 HTML 中引用脚本的那一行）。所以，有些情况下直接在页面中插入代码进行测试更加有用。

14.3.3 Mozilla (所有平台)

Mozilla 特有的 JavaScript 控制台，不仅可以记录错误，还可以发出警告。要访问这个 JavaScript 控制台，Mozilla 1.0+ 查找 Tools→Web Development→JavaScript（在 1.0 之前的版本中，查找 Tasks→Tools→JavaScript Console）。

与它的祖先 Netscape Navigator 一样，Mozilla 也允许在地址栏中输入 javascript: 来打开 JavaScript 控制台，而无须访问菜单项。

JavaScript 控制台报告三种类型的消息：错误，严格警告和消息。错误是语法错误或者运行时错误（任何停止脚本运行的东西），同时会报告错误信息——文件名和发生错误的行号（图 14-5）。严格警告在代码执行了非法的操作时发生，但是解释程序会跳过并让脚本继续执行（这类问题包括重定义已经出现的变量）。消息就纯粹是信息，常常是 Mozilla 内部处理的结果，而非 JavaScript 造成的。

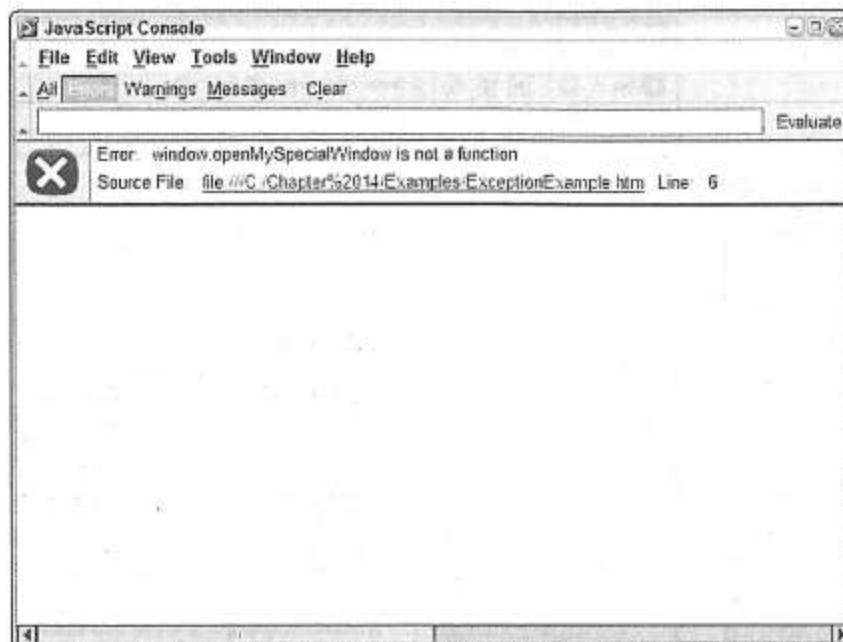


图 14-5

416

14.3.4 Safari (MacOS)

Macintosh 的 Safari 浏览器也许是对 JavaScript 错误和调试的支持最差的浏览器了。默认情况下，它对终端用户不提供任何 JavaScript 错误报告。可按照以下步骤启用错误报告：

- (1) 打开命令行。
- (2) 执行以下命令：defaults write com.apple.Safari IncludeDebugMenu 1。
- (3) 重新启动 Safari。
- (4) 在 debug 菜单下，选择 Log JavaScript Exception（图 14-6）。
- (5) 在 application/Utilities 下启动 Console.app。

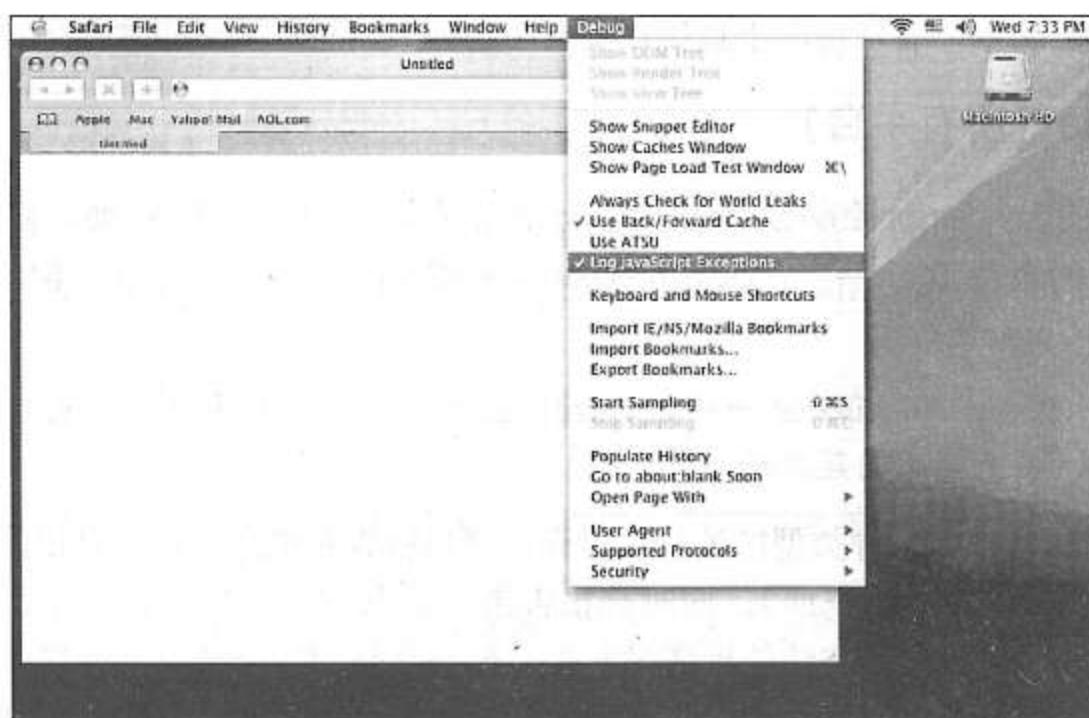


图 14-6

完成这些步骤后，JavaScript 错误就可以输出到 Console.app 窗口中了（图 14-7）。不幸的是，如果没有 URL 或者行号来帮助定位指定的错误，这些消息也是没太大用处。但是，Safari 就只能做到这些了。

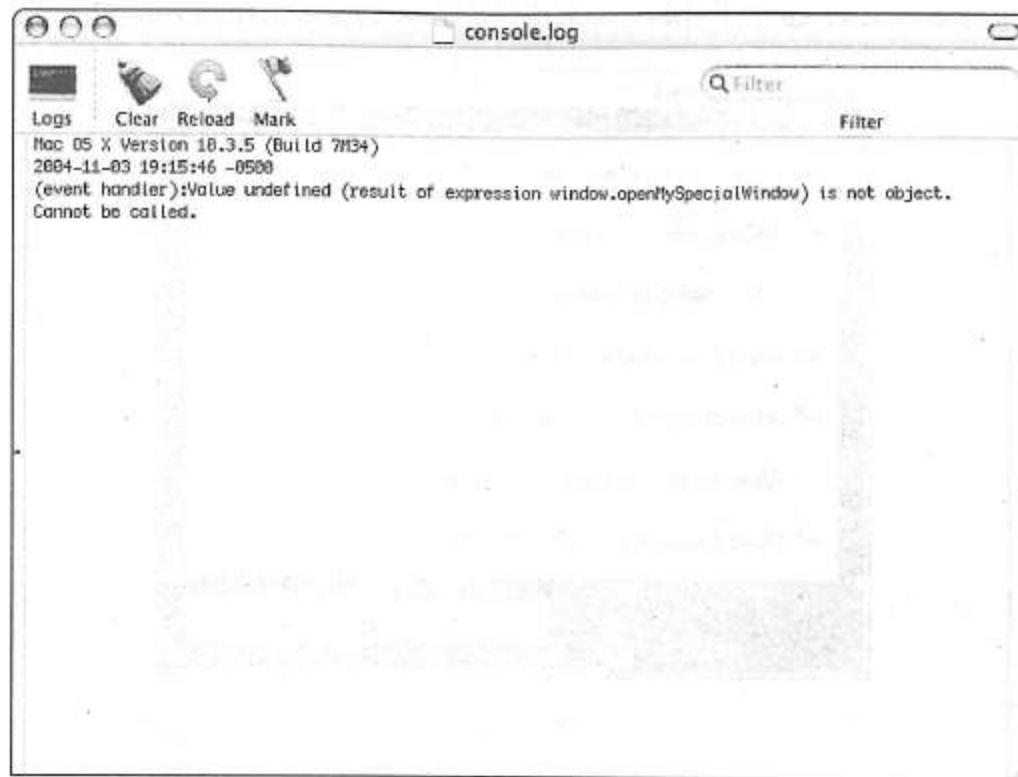


图 14-7

14.3.5 Opera 7 (所有平台)

类似于 Mozilla，Opera 7 也有个 JavaScript 控制台来帮助调试。控制台可以在 Windows→Special→JavaScript Console 中访问。控制台（图 14-8）提供了可用浏览器中大部分的数据，包括错误类型、错误代码以及发生错误的线程和指示错误来源的调用堆栈（在窗口中的标题是 Backtrace）。

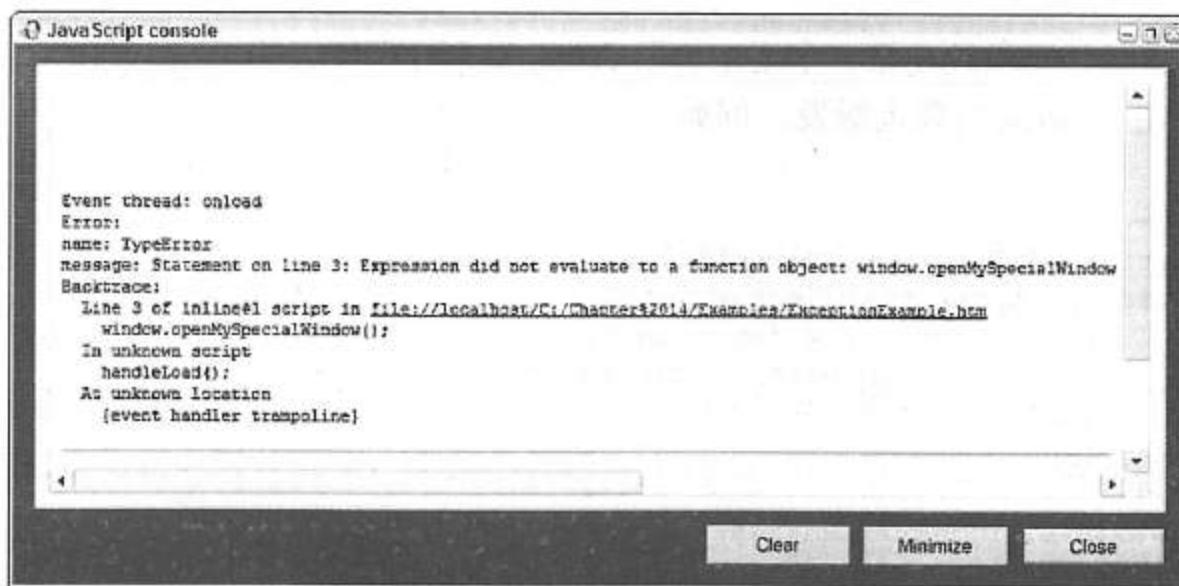


图 14-8

也可以令 JavaScript 控制台一有错误就弹出来，进入 Tools→Preferences，点击 Multimedia。你将看到标题为 Enabled JavaScript 的复选框，以及标题为 JavaScript Options...的按钮，点击这个按钮，会弹出新的对话框（见图 14-9），其中一个选项是 Open JavaScript Console on Error。选中这个复选框（默认情况下没有选中）后，一旦出现错误就会弹出 JavaScript 控制台。

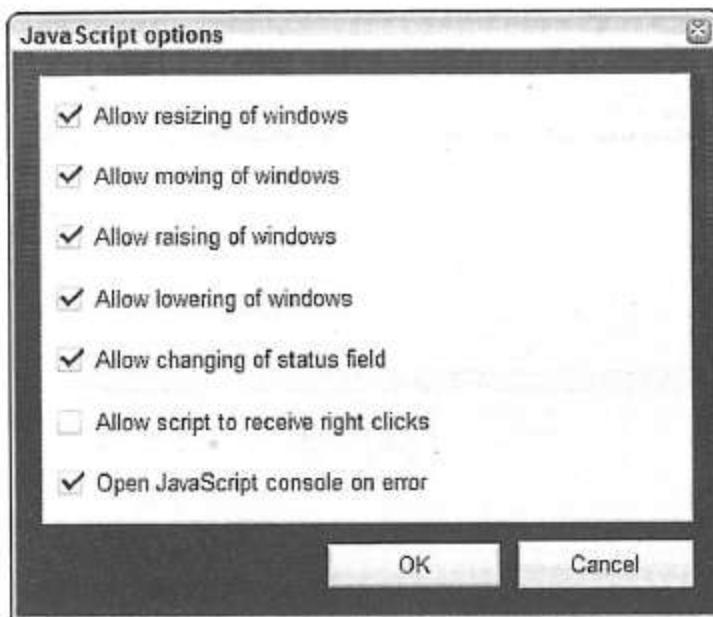


图 14-9

14.4 处理错误

理解错误仅是解决方案的一部分；理解如何处理这些问题则是剩下的很重要的一部分。JavaScript 提供了两种特别的处理错误的方式来。BOM 包含一个 `onerror` 事件处理函数，这在 `window` 对象和图像对象上都有，同时 ECMAScript 定义了另一个从 Java 中借过来的 `try...catch` 结构，来处理异常。这一节将描述两种方式的优点和缺点。

14.4.1 `onerror` 事件处理函数

`onerror` 事件处理函数是第一个用来协助 JavaScript 处理错误的机制。页面上出现异常时，`error` 事件便在 `window` 对象上触发。例如：

```
<html>
  <head>
    <title>OnError Example</title>
    <script type="text/javascript">
      window.onerror = function () {
        alert("An error occurred. ");
      }
    </script>
  </head>
  <body onload="nonExistentFunction()">
  </body>
</html>
```

在这个例子中，尝试调用不存在的函数 `nonExistentFunction()` 时，就会引发一个异常。这时，会弹出警告框显示“An error occurred.”。不幸的是，浏览器的错误信息也显示出来了。要从浏览器中隐藏它（这样就不会报告这个错误了），`onerror` 事件处理函数必须返回 `true` 值：

```
<html>
  <head>
    <title>OnError Example</title>
    <script type="text/javascript">
      window.onerror = function () {
        alert("An error occurred. ");
        return true;
      }
    </script>
  </head>
  <body onload="nonExistentFunction()">
  </body>
</html>
```

1. 取出错误信息

仅知道发生了错误，对于程序员来说没什么用。还好，`onerror` 事件处理函数提供了三种信息来确定错误确切的性质：

- 错误信息——对于给定错误，浏览器会显示同样的信息。
- URL——在哪个文件中发生了错误。
- 行号——给定 URL 中发生错误的行号。

这个信息将作为三个参数传递给 `onerror` 事件处理函数，可以像这样访问：

```
<html>
  <head>
    <title>OnError Example</title>
    <script type="text/javascript">
      window.onerror = function (sMessage, sUrl, sLine) {
        alert("An error occurred:\n" + sMessage + "\nURL: " + sUrl +
"\nLine Number: " + sLine);
        return true;
      }
    </script>
  </head>
  <body onload="nonExistentFunction()">
  </body>
</html>
```

使用这段代码，就可创建自己的 JavaScript 错误对话框来模仿浏览器出错对话框的功能。

2. 图像载入错误

`window` 对象并非唯一支持 `onerror` 事件处理函数的对象；图像对象也支持。当一个图像由于某种原因未能成功载入时（例如，文件不存在），`error` 事件便在这个图像上触发。你可以为图像设置一个 `onerror` 事件处理函数，这可以直接在 HTML 中设置，也可通过脚本进行设置。例如：

```
<html>
  <head>
    <title>Image Error Test</title>
  </head>
  <body>
    <p>The image below attempts to load a file that doesn't exist.</p>
    
  </body>
</html>
```

这个例子直接在 HTML 中分配 onerror 事件处理函数。因为图像 "blue.gif" 不存在，所以才弹出一个警告框让用户知道图像没有完全载入。如果要通过脚本来分配事件处理函数，在设置图像的 src 特性前，必须等待页面完全载入：

```
<html>
  <head>
    <title>Image Error Test</title>
    <script type="text/javascript">
      function handleLoad() {
        document.images[0].onerror = function () {
          alert("An error occurred loading the image.");
        };

        document.images[0].src = "blue.gif";
      }
    </script>
  </head>
  <body onload="handleLoad()">
    <p>The image below attempts to load a file that doesn't exist.</p>
    <img />
  </body>
</html>
```

在此例子中，第一个图像没有 src 特性。页面载入后，先将 onerror 事件处理函数分配给图像，然后将 src 设置为 "blue.gif"，但这个图像并不存在。然后就出现了警告框，显示图像没有载入。

与 window 对象的 onerror 事件处理函数不同，image 的 onerror 事件处理函数没有任何关于额外信息的参数。

421

3. 处理语法错误

onerror 事件处理函数不仅可以处理异常，它还能处理语法错误，也只有它才能处理。

首先，事件处理函数必须是页面中第一个出现的代码。为什么要是第一个呢？如果语法错误出现在设置事件处理函数之前就出现，事件处理函数就没用了。记住，语法错误会完全停止代码的执行。考虑以下例子：

```
<html>
  <head>
```

```

<title>OnError Example</title>
<script type="text/javascript">
    alert("Syntax error. ";
    window.onerror = function (sMessage, sUrl, sLine) {
        alert("An error occurred:\n" + sMessage + "\nURL: " + sUrl +
"\nLine Number: " + sLine);
        return true;
    }
</script>
</head>
<body onload="nonExistentFunction()">
</body>
</html>

```

因为突出显示的那一行代码（里面有语法错误）在分配 `onerror` 事件处理函数之前就出现了，所以浏览器直接报告这个错误。在错误之后的所有代码就不再被解释（因为这个线程已经退出了），所以 `load` 事件触发时调用 `nonExistentFunction()`，浏览器也报告这个错误。如果重写这个页面，将 `onerror` 事件处理函数的分配放在语法错误之前，那么会出现两个警告框：一个显示语法错误，另一个显示异常。

```

<html>
<head>
    <title>OnError Example</title>
    <script type="text/javascript">
        window.onerror = function (sMessage, sUrl, sLine) {
            alert("An error occurred:\n" + sMessage + "\nURL: " + sUrl +
"\nLine Number: " + sLine);
            return true;
        }
        alert("Syntax error. ");
    </script>
</head>
<body onload="nonExistentFunction()">
</body>
</html>

```

422

使用 `onerror` 事件处理函数的主要的问题是，它是 BOM 的一部分，所以，没有任何标准能控制它的行为。因此，不同的浏览器使用这个事件处理函数处理错误的方式有明显的不同。例如，在 IE 中发生 `error` 事件时，正常的代码会继续执行：所有的变量和数据都保留下来，并可通过 `onerror` 事件处理函数访问。然而在 Mozilla 中，正常的代码执行都会结束，同时所有错误发生之前的变量和数据都被销毁。

Safari 和 Konqueror 不支持 `window` 对象上的 `onerror` 事件处理函数，但是它们支持图像上的 `onerror` 事件处理函数。

14.4.2 `try...catch` 语句

ECMAScript 第三版，从 Java 中引入了另外一个功能，`try...catch` 语句，支持 ECMAScript 第三版的浏览器中便可使用（见第 1 章）。基本语法如下：

```

try {
    //code to run
    [break;]
} catch ([exception]) {
    //code to run if an exception occurs and the expression is matched
    [break;]
} [finally {
    //code that is always executed regardless of an exception occurring
}]

```

例如：

```

try {
    window.nonExistentFunction();
    alert("Method completed.");
} catch (exception) {
    alert("An exception occurred.");
} finally {
    alert("End of try...catch test.");
}

```

运行 try...catch 语句时，解释程序首先进入 try 关键词后的代码块。在上面的例子中，执行 window.nonExistentFunction(); 时，会产生错误（因为 window 对象中没有 nonExistentFunction 函数）。这时，解释程序立刻从 try 子句中退出，然后进入 catch 子句，跳过剩下的代码（alert("Method completed."); 这一行未被执行）。在 catch 中的 alert 被执行并弹出警告框，然后继续执行 finally 子句中的代码并显示另一个警告框。

与 Java 不同，ECMAScript 标准在 try...catch 语句中指定只能有一个 catch 子句。因为 JavaScript 是弱类型的语言，没办法指明 catch 子句中异常的特定类型。不管错误是什么类型，都由同一个 catch 子句处理。

423

Mozilla 对 ECMAScript 进行了扩展，可为 try...catch 语句添加多个 catch 子句。当然，因为只有 Mozilla 才能使用这个形式，所以不推荐使用。

在 finally 子句中的代码与 Java 中的 finally 子句的行为一样，用于包含无论是否有异常发生都要执行的代码。这对关闭打开的链接和释放资源很有用。例如：

```

connection.open();
try {
    connection.send(data);
} catch (exception) {
    alert("An exception occurred.");
} finally {
    connection.close();
}

```

1. 嵌套try...catch语句

在 try...catch 语句中的 catch 子句中，也会发生错误。此时，就可以使用嵌套的 try...catch 语句。考虑以下例子：

```

try {
    eval("a ++ b");           //causes error
} catch (oException) {
    alert("An exception occurred. ");
    try {
        var aErrors = new Array(10000000000000000000000000000000);
        aErrors.push(exception);
    } catch (oException2) {
        alert("Another exception occurred.");
    }
} finally {
    alert("All done.");
}

```

在这个例子中，抛出第一个错误后，立刻出现第一个警告框。继续执行第一个 catch 子句时，由于尝试创建一个超大的数组，出现了另一个错误。于是进入第二个 catch 子句，并显示第二个警告框，最后进入 finally 子句，执行完毕。

2. Error对象

那么 catch 语句捕获的到底是什么呢？类似于 Java 有个可用于抛出的基类 Exception，JavaScript 有个 Error 基类用于抛出。Error 对象有以下特性：

- name——表示错误类型的字符串。
- message——实际的错误信息。

Error 对象的名称对应于它的类（因为 Error 只是一个基类），可以是以下值之一：

类	发生原因
EvalError	错误发生在 eval() 函数中
RangeError	数字的值超出 JavaScript 可表示的范围 (Number.MIN_VALUE 和 Number.MAX_VALUE)
ReferenceError	使用了非法的引用
SyntaxError	在 eval() 函数调用中发生了语法错误。其他的语法错误由浏览器报告，无法通过 try...catch 语句处理
TypeError	变量的类型不是预期所需的
URIError	在 encodeURI() 或者 decodeURI() 函数中发生了错误

Error 对象的 message 特性是浏览器生成的用于表示错误性质的信息。因为这个特性是特定于浏览器的，同样的错误在不同的浏览器上可产生不同的错误信息。考虑以下代码：

```
eval("a ++ b");
```

这一行将产生 SyntaxError，因为在这里，++ 操作是错误的。IE 6.0 中给出的错误是 "Exception"，而 Mozilla 1.5 给出的是 "missing; before statement"。

message 特性可为用户提供更加有意义的消息，同时可阻止浏览器直接报告错误：

```

try {
    window.nonExistentFunction();
    alert("Method completed.");
} catch (oException) {
    alert("An exception occurred: " + oException.message);
} finally {
    alert("End of try...catch test.");
}

```

Mozilla 和 IE 都扩展了 `Error` 对象以适应各自的需求。Mozilla 提供一个 `fileName` 特性来表示错误发生在哪一个文件中，以及一个 `stack` 特性以包含到错误发生时的调用堆栈；IE 提供一个 `number` 特性来表示错误代号。

3. 判断错误类型

尽管每个 `try...catch` 语句只能有一个 `catch` 子句，但判断抛出的错误的类型的方法却有几种。第一种方法是使用 `Error` 对象的 `name` 特性。

```

try {
    eval("a ++ b");           //causes SyntaxError
} catch (oException) {
    if (oException.name == "SyntaxError") {
        alert("Syntax Error: " + oException.message);
    } else {
        alert("An unexpected error occurred: " + oException.message);
    }
}

```

第二种方法是使用 `instanceof` 操作符，并使用不同错误的类名：

```

try {
    eval("a ++ b");           //causes SyntaxError
} catch (oException) {
    if (oException instanceof SyntaxError) {
        alert("Syntax Error: " + oException.message);
    } else {
        alert("An unexpected error occurred: " + oException.message);
    }
}

```

4. 抛出异常

ECMAScript 第三版还引入了 `throw` 语句，用于有目的地抛出异常。语法如下：

```
throw error_object;
```

`error_object` 可以是字符串、数字、布尔值或者是实际的对象。下面的代码都是有效的：

```

throw "An error occurred.";
throw 50067;
throw true;
throw new Object();

```

也可以抛出一个 `Error` 对象。`Error` 对象的构造函数只有一个参数，错误信息，如下：

```
throw new Error("You tried to do something bad.");
```

其他的 Error 的子类也都是可以使用的:

```
throw new SyntaxError("I don't like your syntax.");
throw new TypeError("What type of variable do you take me for?");
throw new RangeError("Sorry, you just don't have the range.");
throw new EvalError("That doesn't evaluate.");
throw new URIError("Uri, is that you?");
throw new ReferenceError("You didn't cite your references properly.");
```

426

实际地说，正常的执行不能继续时，应该抛出一个异常，比如这个:

```
function addTwoNumbers(a, b) {
    if (arguments.length < 2) {
        throw new Error("Two numbers are required.");
    } else {
        return a + b;
    }
}
```

在上面的代码中，函数要有两个数字作为参数才能正常工作。如果没有传入两个参数，那么函数就抛出表示计算不能完成的异常。

开发人员抛出的异常和由浏览器自身抛出的错误都在 try...catch 语句中捕获。考虑以下代码，可以捕获由开发人员抛出的异常。

```
function addTwoNumbers(a, b) {
    if (arguments.length < 2) {
        throw new Error("Two numbers are required.");
    } else {
        return a + b;
    }
}

try {
    result = addTwoNumbers(90);
} catch (oException) {
    alert(oException.message);           //outputs "Two numbers are required."
}
```

另外，因为浏览器不生成 Error 对象（它总是生成一个较精确的 Error 对象，比如 RangeError），所以区分浏览器抛出的错误和开发人员抛出的错误很简单，使用前面所讨论过的技术就行了:

```
function addTwoNumbers(a, b) {
    if (arguments.length < 2) {
        throw new Error("Two numbers are required.");
    } else {
        return a + b;
    }
}

try {
    result = addTwoNumbers(90, parseInt("z"));
}
```

```

} catch (oException) {
    if (oException instanceof SyntaxError) {
        alert("Syntax Error: " + oException.message);
    } else if (oException instanceof Error) {
        alert(oException.message);
    }
}

```

427

注意检查 `instanceof Error` 必须是 `if` 语句中的最后一个条件，因为所有其他的错误类都继承于它（所以 `SyntaxError` 在检测 `instanceof Error` 时返回 `true`）。

所有的浏览器都可以报告开发人员抛出的错误，但是错误消息的显示方式可能有所不同。IE 6 只会在抛出 `Error` 对象时，才显示错误信息；其他情况下，它就只显示是 `Exception thrown and not caught`，而没有任何详细内容；Mozilla 则会报告 `Error:uncaught exception:` 然后调用所抛出的对象的 `toString()` 方法。

14.5 调试技巧

在 JavaScript 调试器出现之前，开发人员绞尽脑汁地通过各种途径来调试代码。产生放置 `alert` 的方法、使用 LiveConnect 访问 Java 控制台的方法，使用 JavaScript 控制台的方法，以及抛出自定义错误的方法。这些技巧都各有优缺点。应该使用哪一种呢？这一节将讲解哪种调试方法更适合你。

14.5.1 使用警告框

最流行（也是最笨拙的）的调试方法就算是在代码中放置警告框的方法了。例如：

```

function test_function() {
    alert("Entering function.");

    var iNumber1 = 5;
    var iNumber2 = 10;

    alert("Before calculation.");
    var iResult = iNumber1 + iNumber2;
    alert("After calculation.");

    alert("Leaving function.");
}

```

全世界的 JavaScript 开发人员都认为警告框是一种快捷但很土的调试方式。正如前面例子中那样，不时地弹出描述运行的警告框。有些人甚至会使用计数的方法，第一个警告框为 0，第二个警告框为 1，如此继续，来看代码在哪里停止。

这个方法要求在最后要对代码进行大量的清扫工作，因为调试结束后，必须删除额外的警告框。另一个问题发生在处理死循环上：如果你的脚本造成了死循环或者是长循环，就会不断弹出警告框，最后连关闭浏览器都很困难。因此，最好对小代码段使用警告框进行调试。

428

14.5.2 使用 Java 控制台

在支持 LiveConnect（在 Java 和 JavaScript 之间进行交互的一种机制，将在后面进行讨论）的浏览器中，与在 Java 中一样可以使用 Java 控制台来记录信息。

例如，在 Java 中可以这样记录日志信息：

```
System.out.println("Message");
```

在 JavaScript 中，可以将上面的 System 扩展到完整的 java.lang.System 记号：

```
java.lang.System.out.println("Message");
```

将这样的调用放入函数就可跟踪代码执行了：

```
function test_function() {
    java.lang.System.out.println("Entering function.");

    var iNumber1 = 5;
    var iNumber2 = 10;

    java.lang.System.out.println("Before calculation.");
    var iResult = iNumber1 + iNumber2;
    java.lang.System.out.println("After calculation.");

    java.lang.System.out.println("Leaving function.");
}
```

要查看输出，选择 Tools→Java Console。即可在默认的 Java 控制台输出后看到这些输出（图 14-10）。

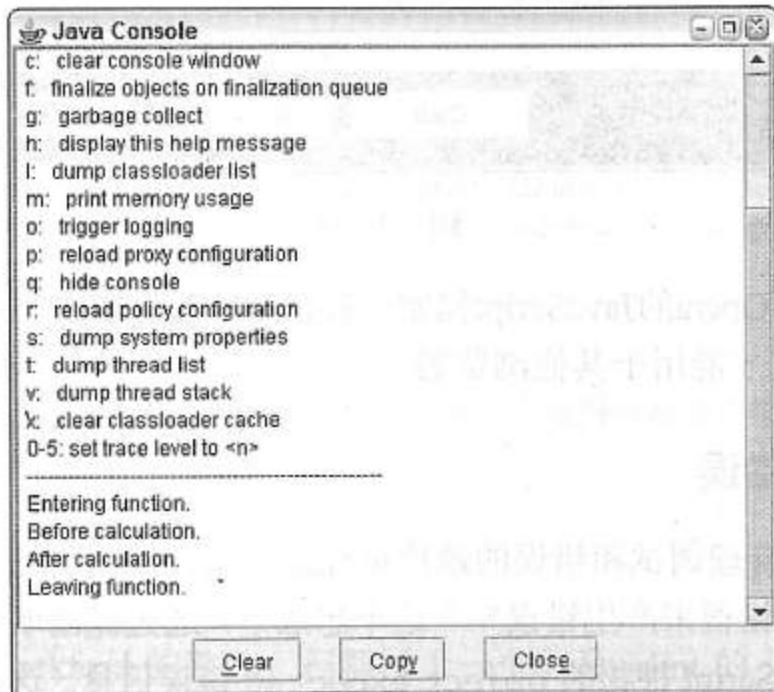


图 14-10

Mozilla、Safari 和 Opera 都支持 LiveConnect；IE 不支持。

14.5.3 将消息写入 JavaScript 控制台（仅限 Opera 7+）

在Opera 7+中，可以通过`opera.postError()`方法直接将消息写入JavaScript控制台中：

```
function test_function() {
    opera.postError("Entering function.");

    var iNumber1 = 5;
    var iNumber2 = 10;

    opera.postError ("Before calculation.");
    var iResult = iNumber1 + iNumber2;
    opera.postError ("After calculation.");

    opera.postError ("Leaving function.");
}
```

尽管这个方法名为`postError`，其实它可用于贴任何消息（见图14-11）。

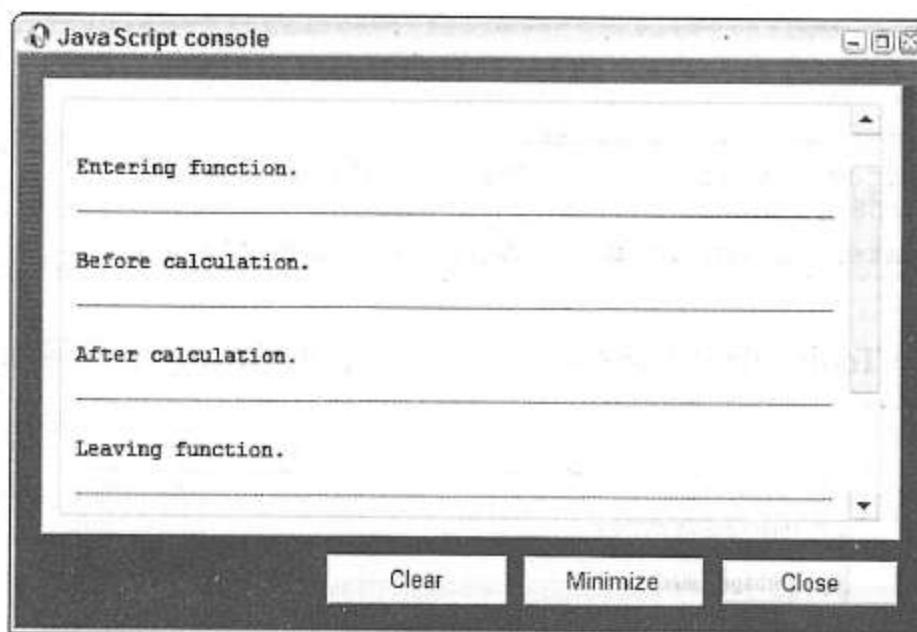


图 14-11

当用于调试目的时，Opera的JavaScript控制台和Java控制台的工作方式一样。当然，唯一的问题是必须删除这些代码才能用于其他浏览器。

14.5.4 抛出自定义错误

在JavaScript代码中管理调试和错误的最佳途径之一是，抛出自己的错误。现在，你可能会在想“怎样才能通过制造错误来产生错误？”这个想法是，通过抛出明确的可以给出错误是为何产生的错误，来替代JavaScript神秘的`object expected`错误信息。这样做可以减少无意义的调试时间。考虑用于将一个数除以另一个数的函数：

```
function divide(iNum1, iNum2) {
    return iNum1.valueOf() / iNum2.valueOf();
}
```

这个函数是基于一些假设的。首先，它假设传入了两个参数；第二，它假设两个参数都是数字。但是如果调用时并没有遵守这两条假设，比如 `divide("a")`，最后就会出现类似 `undefined is not an object` 或者 `iNum2 has no properties` 之类的错误。加入一些更确切的错误信息，问题就会变得更加清晰：

```
function divide(iNum1, iNum2) {
    if (arguments.length != 2) {
        throw new Error("divide() requires two arguments.");
    } else if (typeof iNum1 != "number" || typeof iNum2 != "number") {
        throw new Error("divide() requires two numbers for arguments.");
    }

    return iNum1.valueOf() / iNum2.valueOf();
}
```

在此例中，如果调用 `divide("a")`，会出现错误 `divide() requires two arguments`；如果调用 `divide("a", "b")`，则会出现错误说 `divide() requires two numbers for arguments`。两种情况下，这些错误信息都可以给你足够的信息以使调试更加简单。

因为这些代码总是要先检查错误，然后抛出异常，所以很多开发人员会创建自己的 `assert()` 函数。很多编程语言中都内置了 `assert()` 方法，其实自己创建也很方便：

```
function assert(bCondition, sErrorMessage) {
    if (!bCondition) {
        throw new Error(sErrorMessage);
    }
}
```

其中 `assert()` 函数就是测试第一个参数是否等于 `false`，如果为 `false`，就抛出给定的错误信息的错误。可以这样使用这个 `assert()`：

```
function divide(iNum1, iNum2) {
    assert(arguments.length == 2, "divide() requires two arguments.");
    assert(typeof iNum1 == "number" && typeof iNum2 == "number",
           "divide() requires two numbers for arguments.");

    return iNum1.valueOf() / iNum2.valueOf();
}
```

可以看到，这减少了在 `divide()` 函数中的代码量，也使得开发者在编写该函数时更清楚应该思考些什么。

431

14.5.5 JavaScript 校验器

一位名叫 Douglas Crockford 的软件工程师写了一个称为 `jslint` 的小工具——JavaScript 检验器。`jslint` 的主要目的是指出不合规范的 JavaScript 语法和可能的语法错误。只要将 JavaScript 代码（可以包含 `<script>` 标签）粘贴到文本框中，并点击 `jslint` 按钮。工具就会输出你的代码中的警告和潜在的错误。

`jslint` 给出的警告的类型与本书中提到的很多专业的标准做法是一致的。如果代码包含一些

如下不太合适的编码方式，就会给出一个警告：

- 语句（if、for、while 或其他）未使用块标记（见第 2 章）。
- 一行的结尾未以分号结束。
- 用 var 声明一个已在使用的变量。
- with 语句（避免使用 with 语句的原因请见第 2 章）

JavaScript 校验器可以在 <http://www.crockford.com/javascript/jslint.html> 上找到，也可以直接下载它的源代码。

14.6 调试器

熟练使用 C/C++、Java 和 Perl 之类语言进行开发的开发人员都知道，调试是开发过程中很重要的一个过程。这些开发人员很大程度上依赖于针对这种语言的调试器来协助追踪代码中的错误。大部分语言都提供某种调试器，从最简单的命令行程序到有 GUI 布局的。尽管 JavaScript 自己并不具备调试器，但 IE 和 Mozilla 都有可以使用的调试器。

14.6.1 Microsoft Script Debugger

Microsoft Script Debugger 是个免费工具，可在微软的网站上下载。进入 <http://www.microsoft.com/downloads/search.aspx?categoryid=10> 并在搜索框中输入 script debugger。最后会出现一条记录 Script Debugger for Windows NT, 2000, XP。点击这个连接，并点击 Download 按钮。运行安装程序，然后启动 IE。注意在 View 菜单下出现了 Script Debugger 菜单项。这里面提供了一些运行调试器的选项。

1. 运行

可以以多种方式运行 Microsoft Script Debugger。首先，可直接通过选择 View→Script Debugger→Open，打开没有载入任何信息的调试器。第二种方法是，选择 View→Script Debugger→Break at Next Statement，这可在页面上的下一个 JavaScript 执行前打开调试器。打开调试器后会打开包含正在执行的 JavaScript 的文件。使用这种方法，即将运行的那一行代码会以黄色标亮。最后，还可在代码的任意位置使用 JavaScript 的 debugger 命令来打开调试器。

```
var iSum = 1 + 2;
debugger;
var iProduct = iSum * 10;
```

代码遇到 debugger 命令时，代码执行被中断，并打开调试器（和 Break At Next Statement 选项的功能很像）。

2. 窗口

Microsoft Script Debugger 由一个窗口和三个小的工具窗口组成（见图 14-12）。

第一个工具窗口的标题为 Running Document。这个窗口显示所有正在运行的 IE 的实例，以及在每个实例中载入的文档。通过点击图标前的加号，可以看到会有不止一个 HTML 文件被载入，还有所有相关的 JavaScript 文件。然后就可以通过双击这个窗口中的文件来打开它们。

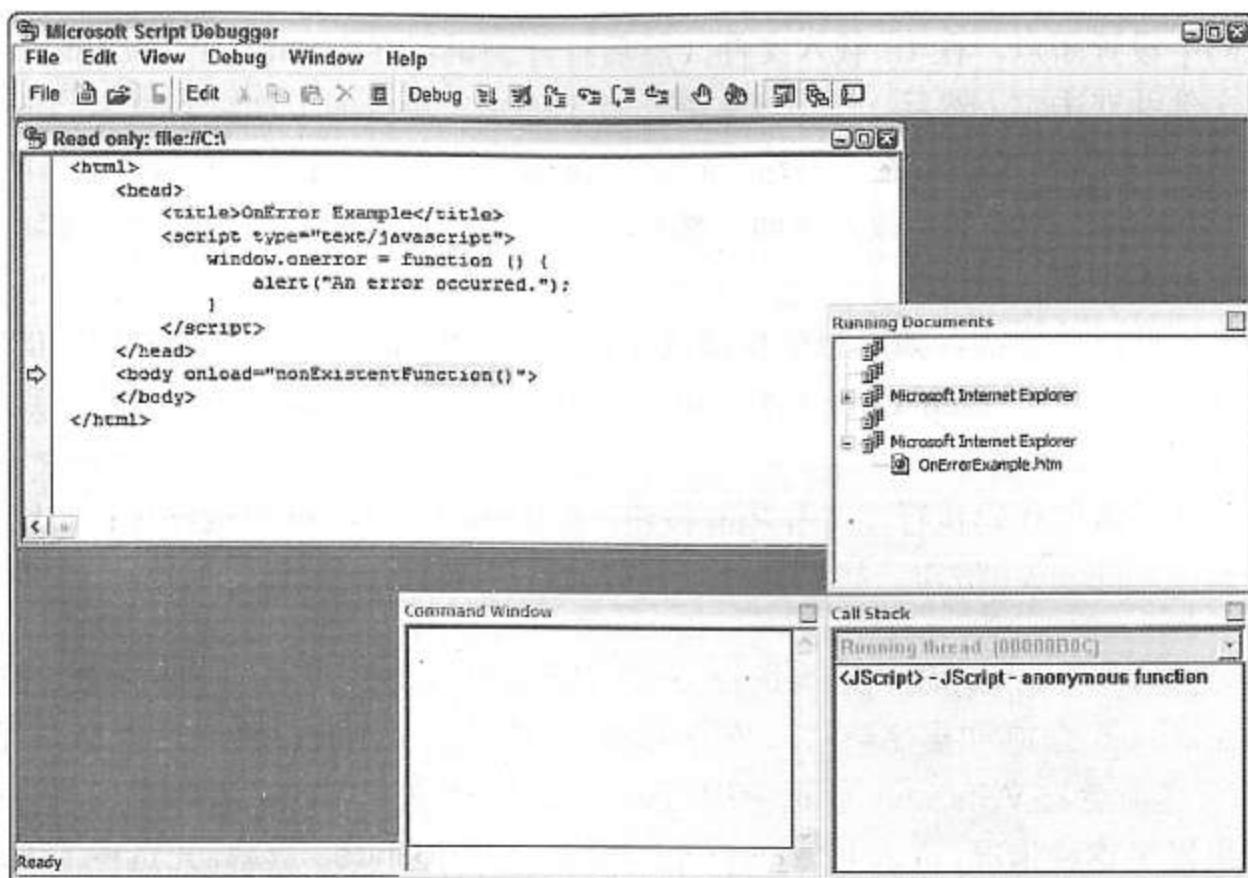


图 14-12

第二个工具窗口的标题为 Call Stack，顾名思义，它显示的是到代码当前断点的调用堆栈。双击 Call Stack 窗口中的某个条目则可以得到这个函数的源代码。

最后一个工具窗口是 Command Window，它与 Visual Studio 中的 Immediate 窗口十分相似。这里，可在正在执行的代码的上下文中输入 JavaScript 命令来检查变量的值。只要在其中输入需要查看的变量的名称，并以分号结尾，就会输出其值。例如，要获取窗口的 URL，输入：

```
window.location.href;
```

按下 Enter 键，就会执行这一行代码并显示出结果。

3. 断点和单步调试

Debug 工具条（图 14-13）包含了所有用于单步调试 JavaScript 代码和设置断点的选项。



图 14-13

要在代码中设置断点，在IE载入文件，然后打开调试器。在Running Documents窗口中找到要调试的文件并双击它。然后，找到要在哪一行上停止，并点击工具栏上的Set Breakpoint（设置断点）按钮（白色手状图标）。然后这一行会用黄颜色标亮，并在前面空白处加上一个黄色的箭头。然后，回到IE中并重新载入页面，然后运行需要进行的命令；代码会在指定的位置停止执行，这时调试器接手工作。

这时，你可做这些事情。如果想单步调试代码，可使用Step Into（按照代码的顺序逐行执行，在每一行之后停止），Step Over（不会进入函数中执行）和Step Out（将执行点移动到函数被调用的位置）。

如果要继续正常的代码执行，点击Run按钮，但代码会在下个断点处停止；如果要完全停止调试，点击Stop Debugger按钮，接下来任何代码的执行都不会停止。
434

很讽刺的是，Microsoft Script Debugger这个调试器本身还需要进一步调试，它的毛病太多了。在Windows 2000和更高版本系统的机器上，这个调试器在保持激活状态方面似乎有很多问题。而且常常会在Windows会话中用过调试器后，并关机重启后，会发现IE中的菜单项不见了。如果发生这种情况，进入Internet Options点击Advanced选单。先选中Disable Script Debugging的复选框，点击应用，然后再取消这个选择，再点击OK。大部分情况下可以重新启用调试器。

14.6.2 Venkman

作为Mozilla开发的调试器，Venkman（名字出自影片《捉鬼敢死队》中的一个角色Peter Venkman）是针对基于Mozilla的浏览器（包括Firefox）的自由工具。安装如下：打开Mozilla浏览器，进入`http://www.hacksrus.com/~ginda/venkman/`。在此，你会发现一个Venkman的列表。点击最新版本旁边的Install链接。会提示是否要继续安装，因为Venkman是个未签署的Mozilla的插件。现在点击Install，然后重启浏览器，你会发现在Tools→Web Development菜单下多了一个新的条目。

1. 运行

要运行Venkman，可点击Tools→Web Development→JavaScript Debugger来手工打开窗口。此后，Venkman的窗口会将所有包含JavaScript的文件自动载入调试器窗口。

也可在代码中使用debugger命令。一旦遇到debugger命令，调试器会自动开启并在这一行代码上停止执行。

与IE不同，Mozilla不会询问你是否希望调试某个错误。

2. 窗口

Venkman的窗口（图14-14）要比Microsoft Script Debugger复杂得多，当然也强大得多。

它的窗口由一些包含关于正在检查的脚本的不同信息的小窗口（也叫视图）组成。每个视图都由一些共同的组件组成，如图14-15所示。
435

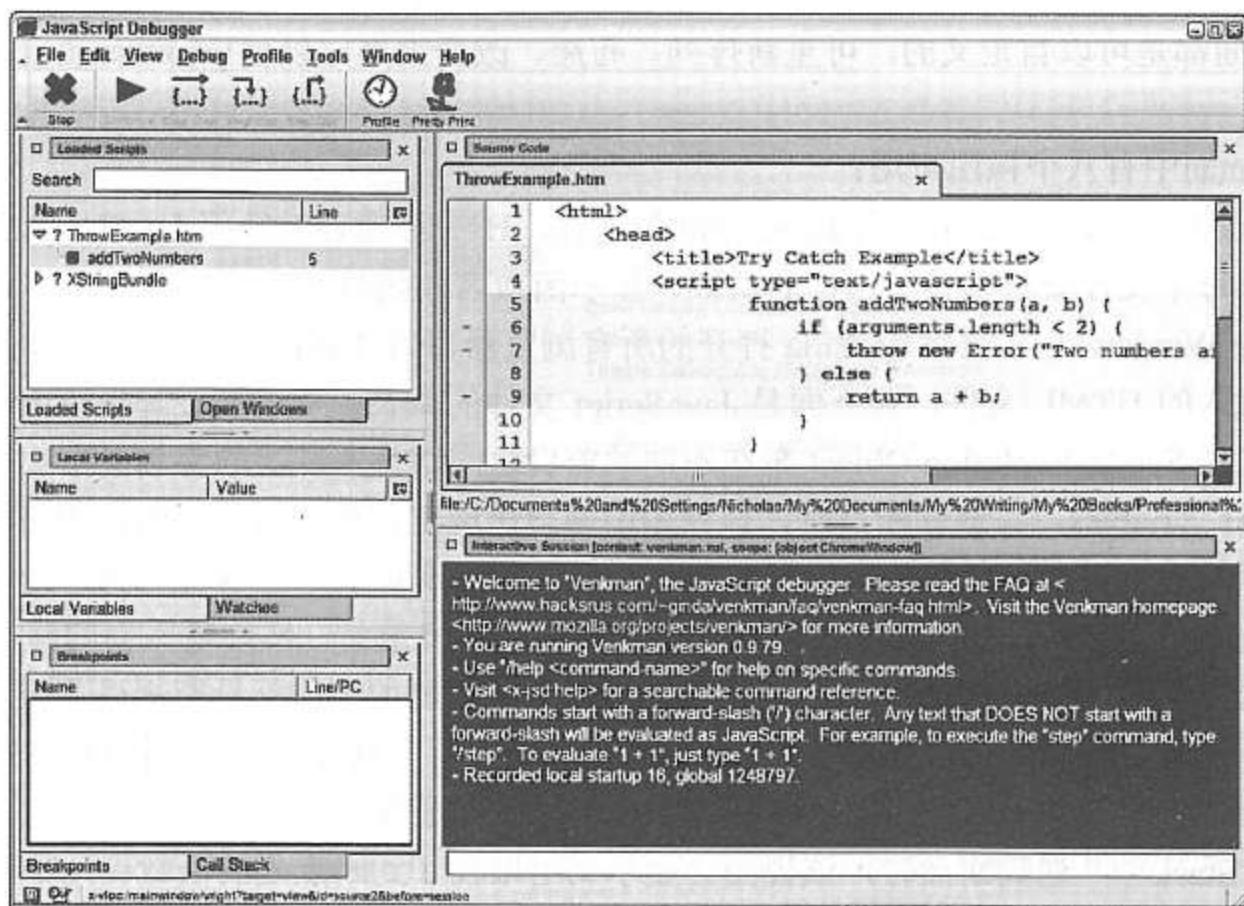


图 14-14



图 14-15

每一个视图由以下部分组成：

- 左上角的小方块。点击这个方块可令视图停靠或者浮动在主窗口上。也可将这个方块拖动到另一个视图的方块上来将这两个视图组合为一个标签页视图。
- 视图的标题。
- 右上角的 X，用于关闭视图（可以在 View→Show/Hide 中选择需要的视图）。
- 内容区域，显示列表或者其他关于脚本的信息。
- 可选的标签组。如果在同一个区域有多个视图，那么调试器会在这里添加一个标签组用于在视图中进行切换。

整个界面都是可以自定义的，可重新排列、布局、改变大小，以使Venkman更好地适应你的要求。

在Venkman中有八个视图可用：

- (1) Loaded Script——显示包含 JavaScript 的文件，HTML 或者是外部 JavaScript 文件。然后展开每个文件会出现其中包含的函数，显示函数名和函数开始的行号。
- (2) Open Windows——显示 Mozilla 打开的所有浏览器窗口（和标签页）。在每个窗口下是已经载入的 HTML 文件，再下面是 JavaScript 文件的列表。可以通过邮件点击给定文件然后选择 Set As Evalution Object 来在不同的窗口之间切换调试器的焦点。
- (3) Local Variables——遇到断点时，该视图中便会出现正在执行的代码的范围内可用的所有变量的列表。如果变量包含的是对象，也可以展开变量名来查看对象所有的特性。在断点处停止执行后，如果要更改变量的值，只需双击变量名，然后输入新的值。
- (4) Watches——显示调试器会话的监视器的列表。监视器的工作就是监视变量值的变化。一旦变量的值发生变化，Watches 视图中就会自动更新（监视器将在后面详细讨论）。
- (5) Breakpoints——显示调试器会话中已经注册的断点的列表。
- (6) Call Stack——遇到断点时，该视图会显示调用堆栈（到断点处的函数调用的序列）。
- (7) Source Code——显示任何包含 JavaScript 的文件的源代码。
- (8) Interactive——一个传统风格的调试器的命令行界面。在这个视图中，利用文本命令几乎可以控制调试器任何方面。

3. Loaded Script面板

默认情况下，Loaded Script 面板显示在调试器窗口的左上角。它显示当前载入调试器中的脚本的位置。面板中包含所有包含在 HTML 中的任何脚本和外部的 JavaScript 文件。

在每个文件（JavaScript 或者 HTML）下面是文件中包含的函数的列表。在图 14-16 中，载入了使用了 throw 操作符的例子，所以可以看到函数 addThrowNumbers() 以及这个函数在文件中开始的行号。如果双击其中一个函数（或者右键点击并选择 Find Function），那么这个函数就会在源代码面板中被突出显示。

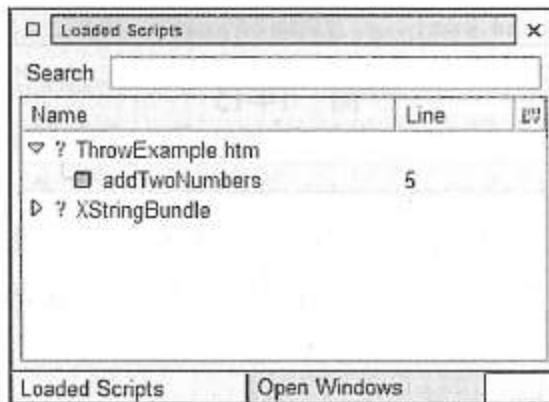


图 14-16

对于 loaded Script 面板中的每个文件，都可由你决定包含在其中的脚本是否该被调试。右键点击这个文件并选择 File Options，会出现一个子菜单（图 14-17），从中可以控制调试以及 profiling

(在本章后面进行讨论)。

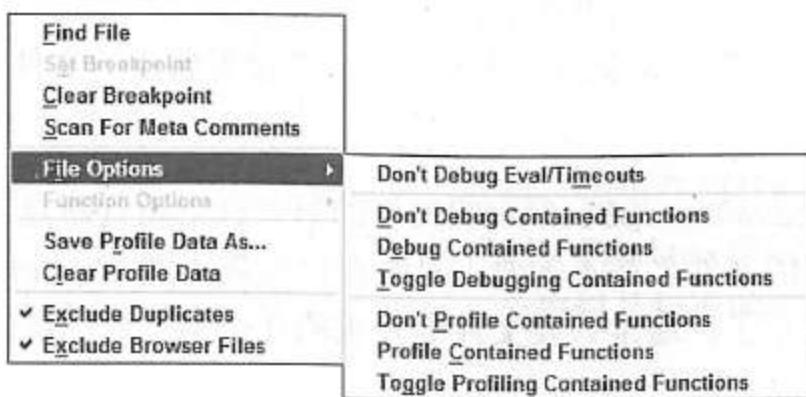


图 14-17

如果不调试文件中的任何代码，点击 Don't Debug Contained Functions (默认情况下，所有的函数都会被调试)。这个菜单还可以防止调试 eval() 或者时段 (timeout) 函数的代码，选择 Don't Debug Eval/Timeout。

默认情况下，Venkman 会只显示已经载入到浏览器中的文件；不过，它也可以载入浏览器在后台额外载入的所有文件。取消上下文菜单中的 Exclude Browser Files 上的勾选就可以看到所有载入的文件。同样，在默认情况下，该面板只显示每个文件被载入的一个实例；取消上下文菜单中的 Exclude Duplicates 选项，就可以看到重复的文件实例。

在 Loaded Scripts 面板中的每个函数都有各自的某种程度的控制权。在函数上点击右键，会出现一个与文件上出现的上下文菜单类似的菜单。在 Function Options 下面，可以点击 Don't Debug 来强迫调试器仅忽略这个函数而非整个文件。这在很大程度上给了开发人员更加灵活的控制权。438

4. 断点

在 Venkman 的各种设置断点的方法中，最快最简单的方法就是双击包含脚本的文件。在 Source Code 视图中显示出的代码中，找到需要设置断点的那一行代码，并点击行左边的空白处（在可以设置断点的行旁边有一个横线）。现在看到一个 B，表示设置了一个硬断点；或者是一个 F，表示调试器只能设置一个未来断点（图 14-18）。未来断点是在脚本已经完全从内存中卸载时创建的，一旦下次这个脚本被载入（例如，刷新浏览器），这个断点就会变成硬断点。

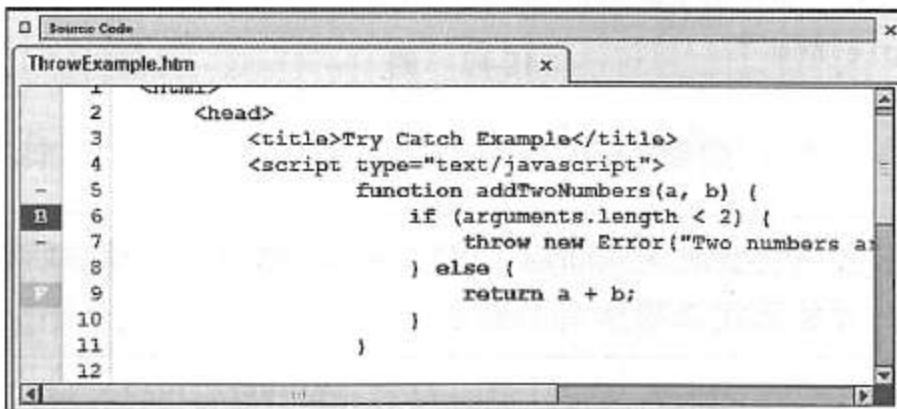


图 14-18

可通过双击（不是单击）左边的空白来创建未来断点，但在单击后，就不能强制已经设置好

的未来断点变成硬断点。

第二个设置断点或者未来断点的方法是，用命令行界面和`/break` 命令（设置硬断点）或者`/fbreak`（设置未来断点）。两个命令都有两个参数：要设置断点的文件名和行号。例如：

```
/break ThrowExample.htm 7
```

这个命令将在匹配`ThrowExample.htm`的文件的第 7 行上设置断点。对于文件名，不需要输入完整的路径甚至连完整的文件名都不需要。只需在已经载入到 Venkman 中的文件中能唯一确定一个文件的几个字符就行了。通常只要文件名开头的几个字符就已足够：

```
/break Thr 7
```

在创建断点后（任意一种方法），断点会被存放在 Breakpoints 视图中的文件名下面。硬断点和未来断点是分开存放的，所以你可能会在同一个文件下看到两种不同的条目（如图 14-19 所示）。
439 只有硬断点可以按照函数名和行号查看，未来断点只显示其文件名和行号。

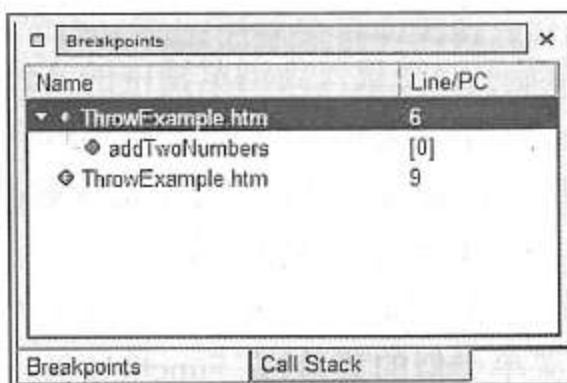


图 14-19

还有多种清除断点的选项。首先，可以点击左边空白处的 B 或者 F，令其变成横线。也可以在 Breakpoints 视图上点击右键，显示上下文菜单，从中可以选择清除一个硬断点、一个未来断点、所有的硬断点或者所有的未来断点。

最后一种方法还是使用命令行界面，使用`/clear` 命令（清除硬断点）或者`/fclear` 命令（清除未来断点）。这些命令接受与`/break` 以及`/fbreak` 同样的参数：文件名和行号。同样，不需要输入完整的文件名，所以下面两行是一样的：

```
/clear ThrowExample.htm 7
/clear Thr 7
```

还可用`/clear-all` 命令来清除所有的硬断点，以及用`/fclear-all` 来清除所有的未来断点。

清除硬断点时，它会自动变成未来断点。这意味着，你必须先清除所有的硬断点，然后再清除所有的未来断点，才能真正消除所有的断点。

5. 单步跟踪

与断点一样，也有多种方法可用于单步调试源代码。最简单的方法是，使用调试工具栏（图 14-20），它总是出现在 Venkman 窗口的顶部。



图 14-20

调试工具栏由五个按钮组成：Stop、Continue、Step Over、Step Into 和 Step Out。Stop 按钮可以停止当前活动的脚本，而不再继续执行以后的代码。如果任何代码正在执行，在 Stop 按钮上面会显示三个白点来表示调试器目前未运行任何代码。Continue 按钮可以恢复脚本的执行，脚本会继续执行到其结束或者遇到下一个断点。最后三个按钮是标准的单步跳过、单步进入和单步跳出命令。

440

这些动作也可以在 Interactive 视图中使用下面表格中所列出的命令来完成：

调试按钮	文本命令
Stop	/stop
Continue	/cont
Step Over	/next
Step Into	/step
Step Out	/finish

代码执行停止在某一行时，则这一行会在 Source Code 视图中以黄颜色突出显示。另外，包含这一行代码的函数在 Call Stack 视图中会有一个黄颜色的箭头（图 14-21）。

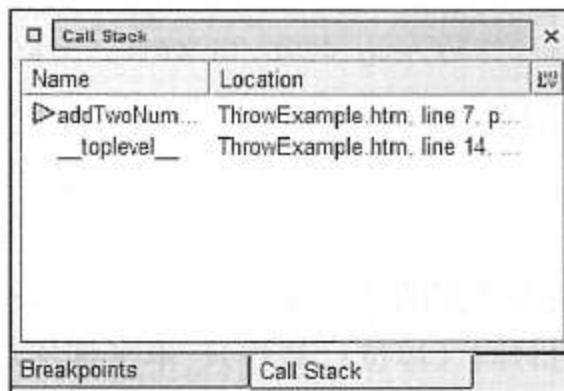


图 14-21

Call Stack 视图中总有个 `_toplevel_` 条目来表示全局范围，这是第一个函数被调用之外。

6. Watches

Venkman 其中一个独特的功能是它可以设置变量监视器。监视器可以监视变量内容的改动，并及时在 Watches 视图中反映出来。

要添加监视器，可在 Local Variables 视图中选择一个变量，右键点击，并选择 Add Watch Expression。也可以在 Interactive 视图中使用 `/watch-expr` 命令来完成同样的工作：

```
/watch-expr variable_name
```

441

将变量加入 Watches 视图后，Watches 视图的行为就与 Local Variables 视图一样了，可以显示每个变量当可用时的值以及对象的特性。

Venkman 中的监视器都与变量名相关，但不是直接对应，所以如果在不同的范围中有两个相同的变量名，会适时显示它们的值。

7. 剖析

Venkman 另外一个独特的功能是，对执行进行剖析的能力。如果执行剖析（profiling），Venkman 就会跟踪每个函数的情况，记录它的调用次数，每次调用花费的时间。

你可以点击工具栏上的 Profile 按钮来开关执行剖析功能。Venkman 正在剖析时，在 Profile 按钮上会出现一个绿色的标志：如果未出现标志，说明它不在进行剖析。当 Venkman 处于剖析模式时，可以运行你的脚本。在对测试满意后，可以在 Profile→Save Profile Data As 中保存测试结果。这时会出现 Save File 对话框，用于选择保存文件的位置。

默认情况下，对话框会建议将文件保存为 HTML 格式。但这是错误的，应该将文件保存为纯文本。

对脚本执行数据剖析后，每个函数在文件中都会有单独的一节，如下：

```
<file:/C:/Chapter%2014/Examples/ThrowExample.htm>

ThrowExample.htm: 1000 - 5000 milliseconds

Function Name: addTwoNumbers (Lines 5 - 10)
Total Calls: 1 (max recurse 0)
Total Time: 4696.75 (min/max/avg 4696.75/4696.75/4696.75)
```

每一节都以包含函数的文件的位置开头，接下来是包含在文件中的每一个函数。每一个函数会显示以下内容：

- 出现的行号；
- 对函数调用的总次数和最大达到的递归层次（max recurse 后的数字）；
- 运行这个函数总共使用的时间（单位：毫秒）、单个调用最短的时间和最长的时间，以及平均每次调用的时间。

这些信息对于找出代码中的瓶颈十分有用。

遗憾地是，剖析的数据包含了浏览器和调试器自身的信息，所以需要通读文件才能找到需要的信息。

也可以指定要进行剖析的函数。右键点击 Loaded Script 视图，并选择 File Options→Don't Profile Contained Function，就可以设置整个文件不被剖析。如果要针对某个单独的函数禁用 profiling，可以在 Loaded Scripts 视图中右键点击这个函数并选择 Function Options→Don't Profile。

14.7 小结

这一章介绍了 JavaScript 中错误及错误处理的方方面面。也介绍了 `onerror` 事件处理函数，包括如何创建自己的错误信息窗口。还学习了如何使用 `try...catch` 语句来逐步捕获错误。之后，针对 `Error` 对象进行了详细的讨论，向你介绍了如何抛出自己的 JavaScript 错误。

接下来，讨论了一些调试技巧，从使用警告框和 Java 控制台信息到抛出自己的错误。还学习了如何创建自己的 `assert()` 函数以使抛出自己的错误更加方便。

最后，向你介绍了针对 IE 和 Mozilla 的 JavaScript 调试器。你学会了如何下载、安装及在各自的浏览器中运行不同的调试器，以及如何使用调试器设置断点和进行单步跟踪。

443
444

随着 XML 的流行, JavaScript 开发者也迫切地希望有可以在客户端 Web 开发中应用 XML 的方法。第四代浏览器出现后,很多开发人员开始使用 JavaScript 编写一些自己的用于处理 XML 的对象。一些浏览器开发人员在感受到这些召唤后,开始大胆地向在客户端支持 XML 和 XML 相关语言的方向推进。

15.1 浏览器中的 XML DOM 支持

虽然 XML 和 DOM 已经变成 Web 开发的重要组成部分,但是还仅有两个浏览器可以支持客户端的 XML 处理。无疑,它们就是世界上最流行的两个浏览器:IE 和 Mozilla。

15.1.1 IE 中的 XML DOM 支持

在为 IE 添加 XML 支持时,微软在 JavaScript 之外另寻了个方案:基于 ActiveX 的 MSXML 库。MSXML 是为向开发人员提供 Windows 平台上首个公用 DOM 实现而开发的。作为一个 ActiveX 控件,MSXML 可以用在 Visual Basic、C++和其他基于 Windows 的开发平台上。所以,有理由直接把这个公司所提供的作为客户端 XML 支持的基础。

微软在 JavaScript 中引入了用于创建 ActiveX 对象的 ActiveXObject 类。ActiveXObject 的构造函数只有一个参数——要进行实例化的 ActiveX 对象的字符串代号。例如,XML DOM 对象的第一个版本称为 Microsoft.XmlDom。所以,要创建这个对象的实例,使用以下代码:

445 var oXmlDom = new ActiveXObject("Microsoft.XmlDom");

执行这行代码后, oXmlDom 对象就同其他 DOM Document 的对象行为完全一样了,包括在本书前面讨论过的各种方法和特性。

开发人员首次使用这个 XML 处理方法时,经常会出现问题,因为用户常常未安装 MSXML。绝大多数情况,开发人员必须直接从微软下载这个库。不过,IE 5.0 修复了这个问题,它直接搭载了 MSXML,这就确保了用 IE 5.0 或更高版本的用户一定可以使用这个功能。

1. DOM 创建

对每个新版本的 MSXML，都会创建出不同的 XML DOM 对象，而它们各自的名称也不同。MSXML 最新的版本是 5.0，也就是说，存在以下 XML DOM 实现：

- Microsoft.XmlDom (最原始的);
- MSXML2.DOMDocument;
- MSXML2.DOMDocument.3.0;
- MSXML2.DOMDocument.4.0;
- MSXML2.DOMDocument.5.0。

自然地，只要可能，大家都会选择最新的 XML DOM 的版本，因为新版会提高速度，添加一些新的功能如验证等。但是，如果尝试创建不存在于客户端机器上的 ActiveX 对象，IE 就会抛出错误并停止所有执行。所以，为确保使用了正确的 XML DOM 版本，也为避免任何其他错误，我们可以创建一个函数来测试每个 XML DOM 字符串，出现错误即捕获：

```
function createXMLDOM() {
    var arrSignatures = ["MSXML2.DOMDocument.5.0", "MSXML2.DOMDocument.4.0",
                         "MSXML2.DOMDocument.3.0", "MSXML2.DOMDocument",
                         "Microsoft.XmlDom"];

    for (var i=0; i < arrSignatures.length; i++) {
        try {
            var oXmlDom = new ActiveXObject(arrSignatures[i]);
            return oXmlDom;
        } catch (oError) {
            //ignore
        }
    }

    throw new Error("MSXML is not installed on your system.");
}
```

这个函数有个 arrSignatures 的数组，包含了所有可能的 XML DOM 字符串，且按照最新到最旧降序排列。for 循环尝试创建位置 i 上的字符串表示的 XML DOM 对象，并存储到变量 oXmlDom 中。如果不存在 XML DOM 的这个版本，那会产生错误，这个错误会被 try...catch 语句捕获，然后忽略。发生错误时，就跳过 return oXmlDom 这一行，进入 for 循环的下一次循环。一旦成功创建了 XML DOM 对象，就立刻将其作为函数值返回。另外，如果测试过 XML DOM 的所有版本，而没有任何版本可用，函数就会抛出错误告诉用户系统上没有安装 MSXML，处理无法继续。

通过使用这个函数，就可确保使用了 XML DOM 的最新版本。

```
var oXmlDom = createXMLDOM();
```

这段在 IE 中创建 XML DOM 的代码在其他浏览器中会出现错误。因此，必须在其之前先进行浏览器检测。

2. 载入XML

现在已有个可用的 XML DOM 对象，就可以载入一些 XML 了。微软的 XML DOM 有两种载入 XML 的方法：`loadXML()` 和 `load()`。

`loadXML()` 方法可直接向 XML DOM 输入 XML 字符串：

```
oXmlDom.loadXML("<root><child/></root>");
```

`load()` 方法用于从服务器上载入 XML 文件。不过，`load()` 方法只可以载入与包含 JavaScript 的页面存储于同一服务器上的文件，也就是说，不可以通过其他人的服务器载入 XML 文件。

还有两种载入文件的模式：同步和异步。以同步模式载入文件时，JavaScript 代码会等待文件完全载入后才继续执行代码；而以异步模式载入时，不会等待，可以使用事件处理函数来判断文件是否完全载入了。

默认情况下，文件按照异步模式载入。要进行同步载入，只需设置 `async` 特性为 `false`：

```
oXmlDom.async = false;
```

然后使用 `load()` 方法，并给出要载入的文件名：

```
oXmlDom.load("test.xml");
```

执行这一行代码后，`oXmlDom` 会包含能表示 XML 文件结构的一个 DOM 文档，这样就可以使用 DOM 所有的特性和方法了：

```
447 alert("Tag name of the root element is " + oXmlDom.documentElement.tagName);
alert("The root element has this many children: " +
      oXmlDom.documentElement.childNodes.length);
```

异步载入文件时，要使用 `readyState` 特性和 `onreadystatechange` 事件处理函数：

`readyState` 特性有五种可能的值：

- 0——DOM 尚未初始化任何信息；
- 1——DOM 正在载入数据；
- 2——DOM 完成了数据载入；
- 3——DOM 已经可用，不过某些部分可能还不能用；
- 4——DOM 已经完全被载入，可以使用了。

一旦 `readyState` 特性的值发生变化，就会触发 `readystatechange` 事件。如果使用 `onreadystatechange` 事件处理函数，就可以在 DOM 完全载入时，发出通知：

```
oXmlDom.onreadystatechange = function () {
    if (oXmlDom.readyState == 4) {
        alert("Done");
    }
};
```

必须在调用 `load()` 方法前分配好 `onreadystatechange` 事件处理函数，就像下面代码中那样：

```

oXmlDom.onreadystatechange = function () {
    if (oXmlDom.readyState == 4) {
        alert("Done");
    }
};

oXmlDom.load("test.xml");

```

现在当文件完全载入后，就会出现警告框显示 Done。

注意，事件处理函数代码使用 `oXmlDom` 而不是 `this` 关键字。这是 JavaScript 中 ActiveX 对象的特殊之处：使用 `this` 关键词可能会出现不可预期的情况。为避免出现问题，最好在事件处理函数中使用完整变量名称。

无论同步或者异步地载入文件，`load()`方法都可以接受部分的、相对的或者完整的 XML 文件路径，如下：

```

oXmlDom.load("test.xml");
oXmlDom.load("../test.xml");
oXmlDom.load("http://www.mydomain.com/test.xml");

```

对部分的及相对的路径的计算总是从使用 XML DOM 的页面起算的，与超链接、图像一样。 448

3. 获取XML

把 XML 载入到 DOM 中后，肯定还需将 XML 取出来。微软为每个节点（包括文档节点）都添加了一个 `xml` 特性，使得这个操作十分方便，它会将 XML 表现形式作为字符串返回。所以要获取载入后的 XML 十分简单：

```

oXmlDom.load("test.xml");
alert(oXmlDom.xml);

```

也可以仅获取某个特定节点的 XML：

```

var oNode = oXmlDom.documentElement.childNodes[1];
alert(oNode.xml);

```

`xml` 特性是只读的，如果尝试直接对其赋值会产生错误。

4. 解释错误

在尝试将 XML 载入到 XML DOM 对象中时，无论使用 `loadXML()`方法还是 `load()`方法，都有可能出现 XML 格式不正确的情况。为解决这个问题，微软的 XML DOM 的 `parseError` 的特性包含了关于解析 XML 代码时所遇到的问题的所有信息。

`parseError` 特性实际上是包含以下特性的对象：

- `errorCode`——表示所发生的错误类型的数字代号（当没有错误时为 0）；
- `filePos`——错误发生在文件中的位置；
- `line`——遇到错误的行号；
- `linepos`——在遇到错误的那一行上的字符的位置；
- `reason`——对错误的一个解释；

- `srcText`——造成错误的代码；
- `url`——造成错误的文件的 URL（如果可用）。

当直接对 `parseError` 自身取值，它会返回 `errorCode` 的值，也就是说可以这样进行检查：

```
if (oXmlDom.parseError != 0) {
    //there were errors, do something about it here
}
```

检查时应该检查错误代码是否不等于 0，因为错误代码可能为正也可能为负。

可以使用 `parseError` 对象来创建自己的错误对话框：

```
if (oXmlDom.parseError != 0) {
    var oError = oXmlDom.parseError;

    449 alert("An error occurred:\nError Code: "
        + oError.errorCode + "\n"
        + "Line: " + oError.line + "\n"
        + "Line Pos: " + oError.linepos + "\n"
        + "Reason: " + oError.reason);

}
```

另一个选项是抛出自己的错误：

```
if (oXmlDom.parseError != 0) {
    var oError = oXmlDom.parseError;

    throw new Error(oError.reason + " (at line " + oError.line
        + ", position " + oError.linepos + ")");
}
```

不管最终如何显示错误，最好在 XML DOM 载入完毕后就立即检查错误。

MSXML ActiveX 控件只能在 Windows 上使用，因此，Macintosh 上的 IE 不能利用这个功能。Windows XP Service Pack 2 对很多 ActiveX 控件引入了新的安全限制，但是 MSXML 不在其中（所有 MSXML 中的控件都认为是安全的）。

15.1.2 Mozilla 中 XML DOM 支持

与 Mozilla 其他方面一样，它提供的 XML DOM 版本要比 IE 的更加标准。Mozilla 中的 XML DOM 实际上是它的 JavaScript 实现，也就是说它不仅与浏览器一起衍化，同时它能可靠地在 Mozilla 支持的所有平台上使用。因此与不能在 Macintosh 上使用 XML DOM 的 IE 不同，Mozilla 的支持跨越了平台的界限。另外，Mozilla 的 XML DOM 实现了支持 DOM Level 2 的功能，而微软的，仅支持 DOM Level 1。

1. 创建DOM

DOM 标准指出，`document.implementation` 对象有个可用的 `createDocument()` 方法。Mozilla 严格遵循了这个标准，可以这样创建 XML DOM：

```
var oXmlDom = document.implementation.createDocument("", "", null);
```

`createDocument()` 的三个参数分别是文档的命名空间 URL, 文档元素的标签名以及一个文档类型对象 (总是为 `null`, 因为在 Mozilla 中还没有对文档类型对象的支持)。前面这行代码创建一个空的 XML DOM。要创建包含一个文档元素的 XML DOM, 只需将标签名作为第二个参数:

```
var oXmlDom = document.implementation.createDocument("", "root", null);
```

450

这行代码创建了代表 XML 代码`<root/>`的 XML DOM。如果在第一个参数中指定了命名空间 URL, 可进一步定义文档元素:

```
var oXmlDom = document.implementation.createDocument("http://www.wrox.com",
    "root", null);
```

这行代码创建了表示`<a0:root xmlns:a0="http://www.wrox.com"/>`的 XML DOM。Mozilla 自动创建一个称为 `a0` 的命名空间来表示输入的命名空间 URL。

2. 载入 XML

与微软的 XML DOM 不同, Mozilla 只支持一个载入数据的方法: `load()`。Mozilla 中的 `load()` 方法和 IE 中的 `load()` 工作方式一样。只要指定要载入的 XML 文件, 以及同步还是异步(默认)载入。

如果同步载入 XML 文件, 代码基本上 IE 差不多:

```
oXmlDom.async = false;
oXmlDom.load("test.xml");
```

如果进行异步载入, 情况就有些不同了。

Mozilla 的 XML DOM 并不支持微软的 `readyState` 特性(实际上, `readyState` 并非 Mozilla 的实现所遵循的 DOM Level 3 载入和保存规范的一部份)。相反, Mozilla 的 XML DOM 会在文件完全载入后触发 `load` 事件, 也就是说必须使用 `onload` 事件处理函数来判断 DOM 何时可用:

```
oXmlDom.onload = function () {
    alert("Done");
};

oXmlDom.load("test.xml");
```

在 1.4 版之前, Mozilla 仅支持外部文件的异步载入。在 1.4 中, Mozilla 引入了 `async` 特性以及同步载入的能力。

不幸的是, Mozilla 的 XML DOM 不支持 `loadXML()` 方法。要将 XML 字符串解析为 DOM, 必须使用 `DOMParser` 对象:

```
var oParser = new DOMParser();
var oXmlDom = oParser.parseFromString("<root/>", "text/xml");
```

这段代码创建了代表`<root/>`的 XML DOM。第一行创建 `DOMParser` 对象, 第二行用它唯一的方法 `parseFromString()` 来创建 XML DOM。这个方法接受两个参数, 要解析的 XML 字符串以及字符串的内容类型。要解析 XML 代码, 内容类型可以是 "`text/xml`" 或者 "`application/`

xml"; 任何其他内容类型都被忽略（它还可以使用内容类型 "application/xhtml+xml" 来解析 XHTML 代码）。

因为 XML DOM 只是 Mozilla 的 JavaScript 实现的一部分，所以它可自己添加 loadXML() 方法。XML DOM 实际的类称为 Document，所以添加新方法同使用 prototype 对象一样容易：

```
Document.prototype.loadXML = function (sXml) {
    //function body
};
```

然后，用 DOMParser 创建新的 XML DOM：

```
Document.prototype.loadXML = function (sXml) {

    var oParser = new DOMParser();
    var oXmlDom = oParser.parseFromString(sXml, "text/xml");

    //...

};
```

下面，原来的文档必须清空其内容。可用 while 循环来删除所有的文档的子节点：

```
Document.prototype.loadXML = function (sXml) {

    var oParser = new DOMParser();
    var oXmlDom = oParser.parseFromString(sXml, "text/xml");

    while (this.firstChild) {
        this.removeChild(this.firstChild);
    }

    //...

};
```

记住，因为这个函数是一个方法，所以 this 关键词指向 XML DOM 对象。在删除所有的子节点后，所有的 oXmlDom 的子节点必须导入到文档中（使用 importNode() 方法）并作为子节点添加（使用 appendChild()）：

```
Document.prototype.loadXML = function (sXml) {

    var oParser = new DOMParser();
    var oXmlDom = oParser.parseFromString(sXml, "text/xml");

    while (this.firstChild) {
        this.removeChild(this.firstChild);
    }

    for (var i=0; i < oXmlDom.childNodes.length; i++) {
        var oNewNode = this.importNode(oXmlDom.childNodes[i], true);
        this.appendChild(oNewNode);
    }

};
```

只需包含这段代码，在 Mozilla 中就可以与在 IE 中一样地使用 `loadXML()` 方法了。

这段代码会在 IE 中造成错误，因为不存在 `Document` 对象。因此，在代码外要使用浏览器检测来避免这个问题。

3. 获取XML

微软的 XML DOM 提供了 `xml` 特性，可以很方便地访问下面的 XML 代码。因为这个特性并非标准的一部分，Mozilla 也不支持它。但是，Mozilla 提供了可以用于同样的目的的 `XMLSerializer` 对象：

```
var oSerializer = new XMLSerializer();
var sXml = oSerializer.serializeToString(oXmlDom, "text/xml");
```

这段简单的代码段用 `XMLSerializer` 唯一的方法——`serializeToString()` 为 `oXmlDom` 创建了 XML 代码。`serializeToString()` 接受要进行序列化的节点和内容类型作为参数。这次，内容类型可以是 "text/xml" 或者 "application/xml"。使用这个对象，就可以自己为 Mozilla 合成 `xml` 特性，只需使用一个鲜为人知的 `defineGetter()` 方法。

`defineGetter()` 方法只存在于 Mozilla 中，用于为某个特性定义获取函数，也就是说，读取特性时，就会调用这个函数并返回它的结果。例如：

```
var sValue = oXmlNode.nodeValue;      //read mode
oXmlNode.nodeValue = "New value";    //write mode
```

第一行代码读取 `nodeValue` 特性，即解释器读取特性的值。如果定义了获取函数，就会运行这个函数并将函数值作为特性的值返回。第二行代码对 `nodeValue` 特性进行设置，即为其赋值。如果定义设置函数（与获取函数相对），则调用这个函数，并以 `New value` 作为参数值。当然，还有个 `defineSetter()` 方法，但在这里用不到。

方法 `defineGetter()` 是按照 JavaScript 对于私有特性和方法的标准隐藏的——在名字前后加上两个下划线：

```
oObject.__defineGetter__("propertyName", function() { return "PropertyValue"; });
```

可以看见，`defineGetter()` 需要两个参数：特性的名称和要调用的函数。指定的特性名不能再用作一般的特性名。例如，不能这样：

```
oObject.propertyName = "blue";
oObject.__defineGetter__("propertyName", function() { return "PropertyValue"; });
```

一般获取函数和设置函数是成对定义的，但是也可以只分配一个获取函数来创建只读特性。这样就可创建 `xml` 特性了。

因为文档中每种类型的节点都有 `xml` 特性。所以最好将其添加到 `Node` 类中（其他的节点类型都是继承 `Node` 的）：

```
Node.prototype.__defineGetter__("xml", function () {
    var oSerializer = new XMLSerializer();
    return oSerializer.serializeToString(this, "text/xml");
});
```

分配给 `xml` 特性的函数十分简单, 对前面的例子的唯一改变是 `this` 是 `serializeToString()` 方法的第一个参数 (在此上下文中, `this` 指向节点自身)。如果将这段代码包含在页面中, 就可以与使用微软的 `xml` 特性方式一样地使用自定义的 `xml` 特性:

```
oXmlDom.load("test.xml");
alert(oXmlDom.xml);

var oNode = oXmlDom.documentElement.childNodes[1];
alert(oNode.xml);
```

这段代码必须包含在浏览器检测中, 因为它只能运行在 Mozilla 中。

4. 解析错误

在 XML 文件的解析过程中发生错误时, XML DOM 会创建文档来解释这个错误。假设运行以下代码:

```
var oParser = new DOMParser();
var oXmlDom = oParser.parseFromString("<root><child></root>");
```

尽管未抛出任何错误, 但是 `oXmlDom` 会显示出错误。在此例子中, 它会呈现出这段代码:¹

```
<parsererror xmlns="http://www.mozilla.org/newlayout/xml/parsererror.xml">
XML Parsing Error: not well-formed
Location: file:///c:/Chapter 15/examples/MozillaXmlDomExample.htm
Line Number 5, Column 1:<sourcetext>&lt;root&gt;&lt;child&gt;&lt;/root&gt;
-----^</sourcetext>
</parsererror>
```

所以要判断是否在 XML 代码的解析过程中有错误, 必须测试文档元素的标签名:

```
if (oXmlDom.documentElement.tagName != "parsererror") {
    //continue on, no errors
} else {
    //do something else, there was an error
}
```

454

可惜, 唯一能获取确切的错误信息的方法是, 解析错误信息文本。最简单的方法是使用正则表达式:

```
var reError = />([\s\S]*?)Location:([\s\S]*?)Line Number (\d+), Column
(\d+):<sourcetext>([\s\S]*?)(?:\-*\^)/;
```

这段正则表达式抓取了所有 XML 代码中潜在的信息。第一个捕获性分组获取错误信息, 第二个获取文件名, 第三个获取行号, 第四个获取列号, 第五个获取造成错误的源代码 (不包含最后的横线和脱字符号)。只需使用 `test()` 方法就可创建错误信息了:

```
var reError = />([\s\S]*?)Location:([\s\S]*?)Line Number (\d+), Column
(\d+):<sourcetext>([\s\S]*?)(?:\-*\^)/;

if (oXmlDom.documentElement.tagName == "parsererror") {
    reError.test(oXmlDom.xml);
    alert("An error occurred:\nDescription: "
```

1. 实际的输出会根据系统语言的不同而有所不同。——译者注

```

+ RegExp.$1 + "\n"
+ "File: " + RegExp.$2 + "\n"
+ "Line: " + RegExp.$3 + "\n"
+ "Line Pos: " + RegExp.$4 + "\n"
+ "Source: " + RegExp.$5);
}

```

15.1.3 通用接口

有了跨浏览器的解决方案时，使用 XML DOM 进行开发才真正有用。IE 和 Mozilla 的实现之间有着十分明显的区别，开发时会造成很严重的问题。所以必须找出一套能在两种浏览器中都可使用 XML DOM 的通用办法。

1. 修改DOM创建

第一步是创建 IE 和 Mozilla 通用的创建 XML DOM 对象的方法。最简单的方法是创建伪类，使之可以这样创建 XML DOM：

```
var oXmlDom = new XmlDom();
```

当然，在 XmlDom 构造函数中必须进行浏览器检测：

```

function XmlDom() {
    if (window.ActiveXObject) {
        //IE-specific code
    } else if (document.implementation && document.implementation.createDocument) {
        //DOM-specific code
    } else {
        throw new Error("Your browser doesn't support an XML DOM object.");
    }
}

```

455

这段代码使用了对象/特性检测法来判断要走哪条路。因为 Windows 上的 IE 是唯一支持 ActiveXObject 类的浏览器，所以这样测试 IE 是可以的。第二种测试比较通用，用于判断浏览器是否支持 DOM 标准的 createDocument() 方法。虽然目前只有 Mozilla 浏览器支持这个方法，但可以预见将来会有其他的浏览器采纳这个功能，所以这样测试确保了代码是经得起未来考验的。如果两个方法都不等于 true，那么构造函数将抛出错误，表示没有可用的 XML DOM 对象。

2. IE 分支

对于构造函数的 IE 部分，只需把本书前面的 createXMLDOM() 函数中的代码插进来即可：

```

function XmlDom() {
    if (window.ActiveXObject) {
        var arrSignatures = ["MSXML2.DOMDocument.5.0", "MSXML2.DOMDocument.4.0",
                             "MSXML2.DOMDocument.3.0", "MSXML2.DOMDocument",
                             "Microsoft.XmlDom"];
        for (var i=0; i < arrSignatures.length; i++) {
            try {
                var oXmlDom = new ActiveXObject(arrSignatures[i]);

```

```

        return oXmlDom;

    } catch (oError) {
        //ignore
    }
}

throw new Error("MSXML is not installed on your system.");

} else if (document.implementation && document.implementation.createDocument) {
    //DOM-specific code
} else {
    throw new Error("Your browser doesn't support an XML DOM object.");
}
}

```

这就是你要对 IE 所做的。对 Mozilla 则要做较复杂的事情。

3. Mozilla分支

Mozilla 分支的第一步是，用 `createDocument()`方法创建 XML DOM 对象。

```

function XmlDom() {
    if (window.ActiveXObject) {
        var arrSignatures = ["MSXML2.DOMDocument.5.0", "MSXML2.DOMDocument.4.0",
                            "MSXML2.DOMDocument.3.0", "MSXML2.DOMDocument",
                            "Microsoft.XmlDom"];

456

        for (var i=0; i < arrSignatures.length; i++) {
            try {

                var oXmlDom = new ActiveXObject(arrSignatures[i]);

                return oXmlDom;

            } catch (oError) {
                //ignore
            }
        }

        throw new Error("MSXML is not installed on your system.");
    } else if (document.implementation && document.implementation.createDocument) {
        var oXmlDom = document.implementation.createDocument("", "", null);
        return oXmlDom;
    } else {
        throw new Error("Your browser doesn't support an XML DOM object.");
    }
}

```

下面的任务是让 Mozilla 支持 `readyState` 特性以及 `onreadystatechange` 事件处理函数。这要求必须对 `Document` 类再进行一些修改。

首先，添加 readyState 特性，并初始化为 0。

```
Document.prototype.readyState = 0;
```

下面，创建 onreadystatechange 特性，并初始化为 null：

```
Document.prototype.onreadystatechange = null;
```

一旦 readyState 特性发生变化，必须调用 onreadystatechange 函数。为达到这个目的，最好创建一个方法：

```
Document.prototype.__changeReadyState__ = function (iReadyState) {
    this.readyState = iReadyState;

    if (typeof this.onreadystatechange == "function") {
        this.onreadystatechange();
    }
};
```

这个方法将新状态作为参数并将其分配给 readyState 特性。注意调用前要先检测 onreadystatechange 确实是个函数（否则会出现错误）。因为这个方法不能在 Document 对象外调用，所以使用了 JavaScript 对于私有方法的标记（前后加上两个下划线）。

IE 的 XML DOM 支持有五种状态，但不可能每种都能在 Mozilla 中进行模拟。其实，readyState 唯一重要的值是 4，它表示 XML DOM 已完全载入并可使用了。这个值可通过 onload 事件处理函数来模拟。模拟 readyState 1 也很容易，它表示 XML DOM 即将开始载入。其他状态不是特别重要，且模拟起来比较困难。457

影响到 readyState 特性的两个方法是，loadXML() 和 load()。loadXML() 方法比较容易更新，因为这是自己创建的方法。只需添加两行代码：

```
Document.prototype.loadXML = function (sXml) {

    this.__changeReadyState__(1);

    var oParser = new DOMParser();
    var oXmlDom = oParser.parseFromString(sXml, "text/xml");

    while (this.firstChild) {
        this.removeChild(this.firstChild);
    }

    for (var i=0; i < oXmlDom.childNodes.length; i++) {
        var oNewNode = this.importNode(oXmlDom.childNodes[i], true);
        this.appendChild(oNewNode);
    }

    this.__changeReadyState__(4);
};
```

更新后的 loadXML() 方法最初将 readyState 设置为 1，结束时将其设置为 4。

要更新 load() 方法，首先要创建指向原来方法的引用：

```
Document.prototype.__load__ = Document.prototype.load;
```

下面，定义新的 load()方法，它先将 readyState 特性设置为 1，然后调用原来的 load() 方法：

```
Document.prototype.load = function (sURL) {
    this.__changeReadyState__(1);
    this.__load__(sURL);
};
```

要使用 onload 事件处理函数适时地将 readyState 设置为 4。因为只能在 XML DOM 对象被实例化之后才能分配事件处理函数，所以必须放在 XmlDom 构造函数中：

```
function XmlDom() {
    if (window.ActiveXObject) {
        var arrSignatures = ["MSXML2.DOMDocument.5.0", "MSXML2.DOMDocument.4.0",
            "MSXML2.DOMDocument.3.0", "MSXML2.DOMDocument",
            "Microsoft.XmlDom"];

        for (var i=0; i < arrSignatures.length; i++) {
            try {
                var oXmlDom = new ActiveXObject(arrSignatures[i]);

                return oXmlDom;
            } catch (oError) {
                //ignore
            }
        }

        throw new Error("MSXML is not installed on your system.");
    } else if (document.implementation && document.implementation.createDocument) {
        var oXmlDom = document.implementation.createDocument("", "", null);

        oXmlDom.addEventListener("load", function () {
            this.__changeReadyState__(4);
        }, false);

        return oXmlDom;
    } else {
        throw new Error("Your browser doesn't support an XML DOM object.");
    }
}
```

458

现在，Mozilla 的 XML DOM 就能较好地支持 readyState 特性（仅对值 0、1 和 4），onreadystatechange 事件处理函数，loadXML()方法以及 xml 特性了。下面的例子可以在 IE 和 Mozilla 中使用：

```
var oXmlDom = new XmlDom();
oXmlDom.onreadystatechange = function () {
    if (oXmlDom.readyState == 4) {
```

```

        alert(oXmlDom.xml);
    }
};

oXmlDom.load("test.xml");

```

其实，两种 XML DOM 处理错误的方式才是主要的区别。不过这个也可以处理好。

4. 错误处理

最后一步是为 Mozilla 创建 parseError 对象。不过这次在 Mozilla 解析器中依然不能模拟所有的 IE 特性，但是却有足够的信息来解析 XML 以模拟大部分的 IE 特性。

首先，必须在 XmlDom 构造函数中创建这个对象并初始化所有的值：

```

function XmlDom() {
    if (window.ActiveXObject) {
        var arrSignatures = ["MSXML2.DOMDocument.5.0", "MSXML2.DOMDocument.4.0",
            "MSXML2.DOMDocument.3.0", "MSXML2.DOMDocument",
            "Microsoft.XmlDom"];
        for (var i=0; i < arrSignatures.length; i++) {
            try {

                var oXmlDom = new ActiveXObject(arrSignatures[i]);

                return oXmlDom;

            } catch (oError) {
                //ignore
            }
        }

        throw new Error("MSXML is not installed on your system.");
    } else if (document.implementation && document.implementation.createDocument) {

        var oXmlDom = document.implementation.createDocument("", "", null);

        oXmlDom.parseError = {
            valueOf: function () { return this.errorCode; },
            toString: function () { return this.errorCode.toString() }
        };

        oXmlDom.__initError__();

        oXmlDom.addEventListener("load", function () {
            this.__changeReadyState__(4);
        }, false);

        return oXmlDom;
    } else {
        throw new Error("Your browser doesn't support an XML DOM object.");
    }
}

```

这段代码使用对象字面量记号来创建 `parseError` 对象，节省了空间。并定义 `valueOf()` 方法返回 `errorCode` 特性，这与 IE 的实现一样；`toString()` 方法也返回 `errorCode` 特性，不过是以字符串形式。`initError()` 方法初始化所有 `parseError` 对象的特性。代码如下：

```
Document.prototype.__initError__ = function () {
    this.parseError.errorCode = 0;
    this.parseError.filepos = -1;
    this.parseError.line = -1;
    this.parseError.linepos = -1;
    this.parseError.reason = null;
    this.parseError.srcText = null;
    this.parseError.url = null;
};
```

下一步是检测解析错误。必须在 `load()` 和 `loadXML()` 方法中完成，因为解析错误可能发生在其中任何一个中。因为在两处重复相同的代码是种不好的编码实践，最好创建函数来处理解析错误：

```
460 Document.prototype.__checkForErrors__ = function () {
    if (this.documentElement.tagName == "parsererror") {
        var reError = />([\s\S]*?)Location:([\s\S]*?)Line Number (\d+), Column (\d+):<sourcetext>([\s\S]*?)(?:\-*\^)/;
        reError.test(this.xml);
        this.parseError.errorCode = -999999;
        this.parseError.reason = RegExp.$1;
        this.parseError.url = RegExp.$2;
        this.parseError.line = parseInt(RegExp.$3);
        this.parseError.linepos = parseInt(RegExp.$4);
        this.parseError.srcText = RegExp.$5;
    }
};
```

注意，不管发生什么错误，`errorCode` 都设为 -999999。映射微软所有的错误代码是一项繁琐的任务，且没有什么必要。大部分情况下，只需要检查 `parseError` 是否为 0，而不是某个特定的值。

下面，`load()` 和 `loadXML()` 方法必须进行更新以使用 `initError()`（进行解析前清除所有的错误值）和 `checkForErrors()`（解析完毕时检测有没有解析错误）：

```
function XmlDom() {
    if (window.ActiveXObject) {
        var arrSignatures = ["MSXML2.DOMDocument.5.0", "MSXML2.DOMDocument.4.0",
            "MSXML2.DOMDocument.3.0", "MSXML2.DOMDocument",
            "Microsoft.XmlDom"];
        for (var i=0; i < arrSignatures.length; i++) {
            try {
```

```
var oXmlDom = new ActiveXObject(arrSignatures[i]);

return oXmlDom;

} catch (oError) {
    //ignore
}

}

throw new Error("MSXML is not installed on your system.");

} else if (document.implementation && document.implementation.createDocument) {

    var oXmlDom = document.implementation.createDocument("", "", null);

    oXmlDom.parseError = {
        valueOf: function () { return this.errorCode; },
        toString: function () { return this.errorCode.toString() }
    };

    oXmlDom.__initError__();

    oXmlDom.addEventListener("load", function () {
        this.__checkForError__();
        this.__changeReadyState__(4);
    }, false);

    return oXmlDom;

} else {
    throw new Error("Your browser doesn't support an XML DOM object.");
}

}

Document.prototype.load = function (sURL) {
    this.__initError__();
    this.__changeReadyState__(1);
    this.__load__(sURL);
};

Document.prototype.loadXML = function (sXml) {

    this.__initError__();
    this.__changeReadyState__(1);

    var oParser = new DOMParser();
    var oXmlDom = oParser.parseFromString(sXml, "text/xml");

    while (this.firstChild) {
        this.removeChild(this.firstChild);
    }

    for (var i=0; i < oXmlDom.childNodes.length; i++) {
```

```

        var oNewNode = this.importNode(oXmlDom.childNodes[i], true);
        this.appendChild(oNewNode);
    }

    this.__checkForErrors__();
    this.__changeReadyState__(4);
};


```

注意, initError()必须在 readyState 设为 1 之前调用。类似的, checkForErrors()也必须在 readyState 设为 4 (这就是必须在 onload 事件处理函数中调用的原因) 之前调用。这些方法必须按顺序调用, 因为每次 readyState 更改时都要调用 onreadystatechange。如果在 parseError 对象中存在旧的数据, 必须在调用 onreadystatechange 前重置, 否则可能会造成混淆。在同一行上, parseError 对象必须在 readyState 变成 4 后包含正确的数据, 因为那时所有的处理都应该已完成了。

462

加入这段代码后, 就可以写一段完整的可以同时在 IE 和 Mozilla 中处理解析错误的代码了:

```

var oXmlDom = new XmlDom();
oXmlDom.onreadystatechange = function () {
    if (oXmlDom.readyState == 4) {

        if (oXmlDom.parseError != 0) {
            var oError = oXmlDom.parseError;
            alert("An error occurred:\nError Code: "
                + oError.errorCode + "\n"
                + "Line: " + oError.line + "\n"
                + "Line Pos: " + oError.linepos + "\n"
                + "Reason: " + oError.reason);

        }
    }
};

oXmlDom.load("errors.xml");

```

这个例子载入了有错误的 XML 文件。readyState 特性设成 4 时(文件载入并进行了解析)。检验 parseError 的值是否等于 0 (代表是否有错误)。如果发生了错误, 出现一个警告框, 显示错误代码、行号、行位置(列号)以及错误原因。

5. 完整的代码

在这一章中, 开发代码时, 我跳过了许多内容。下面看一下完整的代码(注意它使用了本书前面创建的浏览器检测代码):

```

function XmlDom() {
    if (window.ActiveXObject) {
        var arrSignatures = ["MSXML2.DOMDocument.5.0", "MSXML2.DOMDocument.4.0",
            "MSXML2.DOMDocument.3.0", "MSXML2.DOMDocument",
            "Microsoft.XmlDom"];
    }

    for (var i=0; i < arrSignatures.length; i++) {
        try {

```

```
var oXmlDom = new ActiveXObject(arrSignatures[i]);

return oXmlDom;

} catch (oError) {
    //ignore
}

}

throw new Error("MSXML is not installed on your system.");

} else if (document.implementation && document.implementation.createDocument) { 463
    var oXmlDom = document.implementation.createDocument("", "", null);

    oXmlDom.parseError = {
        valueOf: function () { return this.errorCode; },
        toString: function () { return this.errorCode.toString() }
    };

    oXmlDom.__initError__();

    oXmlDom.addEventListener("load", function () {
        this.__checkForErrors__();
        this.__changeReadyState__(4);
    }, false);

    return oXmlDom;

} else {
    throw new Error("Your browser doesn't support an XML DOM object.");
}

}

if (isMoz) {

    Document.prototype.readyState = 0;
    Document.prototype.onreadystatechange = null;

    Document.prototype.__changeReadyState__ = function (iReadyState) {
        this.readyState = iReadyState;

        if (typeof this.onreadystatechange == "function") {
            this.onreadystatechange();
        }
    };

    Document.prototype.__initError__ = function () {
        this.parseError.errorCode = 0;
        this.parseError.filepos = -1;
        this.parseError.line = -1;
        this.parseError.linepos = -1;
        this.parseError.reason = null;
        this.parseError.srcText = null;
        this.parseError.url = null;
    };
}
```

```

};

Document.prototype.__checkForErrors__ = function () {
    if (this.documentElement.tagName == "parsererror") {
        var reError = />([\s\S]*?)Location:([\s\S]*?)Line Number (\d+), Column (\d+):<sourcetext>([\s\S]*?)(?:\-*\^)/;
        reError.test(this.xml);

        this.parseError.errorCode = -999999;
        this.parseError.reason = RegExp.$1;
        this.parseError.url = RegExp.$2;
        this.parseError.line = parseInt(RegExp.$3);
        this.parseError.linepos = parseInt(RegExp.$4);
        this.parseError.srcText = RegExp.$5;
    }
};

464 Document.prototype.loadXML = function (sXml) {
    this.__initError__();
    this.__changeReadyState__(1);

    var oParser = new DOMParser();
    var oXmlDom = oParser.parseFromString(sXml, "text/xml");

    while (this.firstChild) {
        this.removeChild(this.firstChild);
    }

    for (var i=0; i < oXmlDom.childNodes.length; i++) {
        var oNewNode = this.importNode(oXmlDom.childNodes[i], true);
        this.appendChild(oNewNode);
    }

    this.__checkForErrors__();
    this.__changeReadyState__(4);
};

Document.prototype.__load__ = Document.prototype.load;

Document.prototype.load = function (sURL) {
    this.__initError__();
    this.__changeReadyState__(1);
    this.__load__(sURL);
};

Node.prototype.__defineGetter__("xml", function () {

```

```

        var oSerializer = new XMLSerializer();
        return oSerializer.serializeToString(this, "text/xml");
    });

}

```

15.2 浏览器中的 XPath 支持

因为 XML 用于处理多种数据，所以必须有一种可以在 XML 代码中定位数据的方式。这个问题的答案就是 XPath，它是专门用于定位匹配模式的一个或多个节点的小语言。尽管关于 XPath 深入的讨论已经超出本书的范围，不过还是要进行一些简单的介绍。

465

15.2.1 XPath 简介

每个 XPath 表达式都有两部分：一个上下文节点和一个节点模式。上下文节点提供了节点模式起始的位置。节点模式是由一个或多个节点选择器组成的字符串。

例如，考虑以下 XML 文档：

```

<?xml version="1.0"?>
<employees>
    <employee title="Software Engineer">
        <name>Nicholas C. Zakas</name>
    </employee>
    <employee title="Salesperson">
        <name>Jim Smith</name>
    </employee>
</employees>

```

同时考虑这个 XPath 表达式：

```
employee/name
```

如果上下文节点是`<employees>`，则前面的 XPath 表达式就匹配了`<name>Nicholas C. Zakas</name>`和`<name>Jim Smith</name>`。在这个表达式中，`employee` 和 `name` 都表示 XML 元素的标签名，按照它们在上下文节点中出现的顺序；斜杠表示从父节点到子节点的关系。这个 XPath 表达式表示：从`<employees>`起，匹配位于`<employee>`元素下的子节点`<name>`元素。要选择地`<employee>`元素的第一个`<name>`元素，表达式要变成：

```
employee[position() = 1]/name
```

在 XPath 中，方框记号用于为某个节点提供更加确切的信息。这个例子使用了 XPath 的 `position()` 函数，它用于返回元素在父节点下的位置。第一个子节点的位置为 1，所以将 `position()` 和 1 进行比较则只能匹配第一个`<employee>`元素。然后，斜杠和 `name` 匹配在第一个`<employee>`元素下的`<name>`元素。

除位置和名称外，还可以使用不同的方法来匹配元素。假设要选择所有 `title` 特性等于 "Salesperson" 的`<employee>`元素，则 XPath 表达式变成：

```
employee[@title = "Salesperson"]
```

在表达式中，@是 attribute 的缩写。

XPath 是一种十分强大的表达式可以令在 DOM 文档中查找指定节点变得很容易。因此，IE 和 Mozilla 都在 DOM 实现中引入了 XPath 支持。

如果想了解更多关于 XPath 的信息，请参阅 *XPath 2.0: Programmer's Reference* (Wiley 出版社，ISBN 0-7645-6910-4)。

15.2.2 IE 中的 XPath 支持

看来，微软对直接在 XML DOM 对象中建立 XPath 支持感觉很惬意。每个节点都有两个可用于获取匹配 XPath 模式的节点的方法：`selectNodes()`，用于返回匹配某个模式的节点的集合；`selectSingleNode()`，用于返回匹配给定模式的第一个节点。

使用前面一节中的数据，可用下面的代码选择所有`<employee/>`元素下的`<name/>`元素：

```
var lstNodes = oXmlDom.documentElement.selectNodes("employee/name");
```

因为 `selectNodes()` 是作为 `oXmlDom.documentElement` 的方法调用的，所以文档元素被看作 XPath 表达式的上下文节点。方法返回包含所有匹配给定模式的节点的 `NodeList`，也就是说，可以这样迭代所有的元素：

```
for (var i=0; i < lstNodes.length; i++) {
    alert(lstNodes[i]);
}
```

如果没有匹配给定模式的节点，还是会返回 `NodeList`。如果为空，则它的 `length` 属性值等于 0。

`selectNodes()` 的返回结果是活的列表，所以如果用另外一个匹配 XPath 表达式的节点更新该文档，则这个元素会自动被添加到 `NodeList` 的合适位置。

如果只需要匹配模式的第一个元素，则可以使用 `selectSingleNode()`：

```
var oElement = oXmlDom.documentElement.selectSingleNode("employee/name");
```

如果发现了节点，则 `selectSingleNode()` 方法返回一个 `Element`，否则它返回 `null`。

15.2.3 Mozilla 中的 XPath 支持

Mozilla 是根据 DOM 标准来实现对 XPath 的支持的。DOM Level 3 附加标准 DOM Level 3 XPath 定义了用于在 DOM 中计算 XPath 表达式的接口。遗憾的是，这个标准要比微软直观的方式复杂得多。

虽然有好多与 XPath 相关的对象，最重要的两个是：`XPathEvaluator` 和 `XPathResult`。`XpathEvaluator` 利用方法 `evaluate()` 计算 XPath 表达式。

`evaluate()`方法有五个参数：XPath 表达式、上下文节点、命名空间解释程序和返回的结果的类型，同时，在 `XPathResult` 中存放结果（通常为 `null`）。

命名空间解释程序，只有在 XML 代码用到了 XML 命名空间时才是必要的，所以通常留空，置为 `null`。返回结果的类型，可以是以下十个常量值之一：

- `XPathResult.ANY_TYPE`——返回符合 XPath 表达式类型的数据；
- `XPathResult.ANY_UNORDERED_NODE_TYPE`——返回匹配节点的节点集合，但顺序可能与文档中的节点的顺序不匹配；
- `XPathResult.BOOLEAN_TYPE`——返回布尔值；
- `XPathResult.FIRST_ORDERED_NODE_TYPE`——返回只包含一个节点的节点集合，且这个节点是在文档中第一个匹配的节点；
- `XPathResult.NUMBER_TYPE`——返回数字值；
- `XPathResult.ORDERED_NODE_ITERATOR_TYPE`——返回匹配节点的节点集合，顺序为节点在文档中出现的顺序。这是最常用到的结果类型；
- `XPathResult.ORDERED_NODE_SNAPSHOT_TYPE`——返回节点集合快照，在文档外捕获节点，这样将来对文档的任何修改都不会影响这个节点列表。节点集合中的节点与它们出现在文档中的顺序一样；
- `XPathResult.STRING_TYPE`——返回字符串值；
- `XPathResult.UNORDERED_NODE_ITERATOR_TYPE`——返回匹配节点的节点集合，不过顺序可能不会按照节点在文档中出现的顺序排列；
- `XPathResult.UNORDERED_NODE_SNAPSHOT_TYPE`——返回节点集合快照，在文档外捕获节点，这样将来对文档的任何修改都不会影响这个节点列表。节点集合中的节点和文档中原来的顺序不一定一样。

指定的结果类型决定了如何获取结果的值。下面是个典型的例子：

```
var oEvaluator = new XPathEvaluator();
var oResult = oEvaluator.evaluate("employee/name", oXmlDom.documentElement, null,
                                  XPathResult.ORDERED_NODE_ITERATOR_TYPE, null);

if (oResult != null) {
    var oElement = oResult.iterateNext();
    while(oElement) {
        alert(oElement.tagName);
        oElement = oResult.iterateNext();
    }
}
```

这个例子使用了 `XPathResult.ORDERED_NODE_ITERATOR_TYPE` 结果，它是最常用到的结果类型。如果没有节点匹配 XPath 表达式，`evaluate()` 返回 `null`；否则，它返回一个 `XPathResult` 对象。如果结果是个节点迭代子，不管它是有序的还是无序的，都可以不断用 `iterateNext()` 方法获取在结果中的每一个匹配的结果。如果没有更多匹配的节点，`iterateNext()` 返回 `null`。可以用节点迭代子为 Mozilla 创建一个 `selectNodes()` 方法：

468

```

Element.prototype.selectNodes = function (sXPath) [
    var oEvaluator = new XPathEvaluator();
    var oResult = oEvaluator.evaluate(sXPath, this, null,
        XPathResult.ORDERED_NODE_ITERATOR_TYPE,
        null);

    var aNodes = new Array;

    if (oResult != null) {
        var oElement = oResult.iterateNext();
        while(oElement) {
            aNodes.push(oElement);
            oElement = oResult.iterateNext();
        }
    }

    return aNodes;
];

```

给 Element 类添加 selectNodes() 方法以模仿 IE 中的行为。调用 evaluate() 时，把用 this 关键字作为上下文节点（这也是 IE 的工作方式）。然后，在结果数组 (aNodes) 中放入匹配的节点。新方法用法如下：

```

var aNodes = oXmlDom.documentElement.selectNodes("employee/name");
for (var i=0; i < aNodes.length; i++) {
    alert(aNodes[i].xml);
}

```

如果指定了快照结果类型（不管是有序的还是无序的），都可使用 snapshotItem() 以及 snapshotLength() 方法，如下例所示：

```

var oEvaluator = new XPathEvaluator();
var oResult = oEvaluator.evaluate("employee/name", oXmlDom.documentElement, null,
    XPathResult.ORDERED_NODE_SNAPSHOT_TYPE, null);

if (oResult != null) {
    for (var i=0; i < oResult.snapshotLength; i++) {
        alert(oResult.snapshotItem(i).tagName);
    }
}

```

在这个例子中，snapshotLength 返回节点的数量，snapshotItem() 返回快照中给定位置上的节点（类似于 NodeList 的 length 和 item()）。

XPathResult.FIRST_ORDERED_NODE_TYPE 结果返回第一个匹配的节点，可通过 singleNodeValue 特性来访问：

```

var oEvaluator = new XPathEvaluator();
var oResult = oEvaluator.evaluate("employee/name", oXmlDom.documentElement, null,
    XPathResult.FIRST_ORDERED_NODE_TYPE, null);

alert(oResult.singleNodeValue.xml);

```

你肯定已猜到，这段代码可用来模仿 IE 的 selectSingleNode() 方法：

469

```

Element.prototype.selectSingleNode = function (sXPath) {
    var oEvaluator = new XPathEvaluator();
    var oResult = oEvaluator.evaluate(sXPath, this, null,
        XPathResult.FIRST_ORDERED_NODE_TYPE, null);

    if (oResult != null) {
        return oResult.singleNodeValue;
    } else {
        return null;
    }
}

```

这个方法的用法与在 IE 中的一样：

```

var oNode = oXmlDom.documentElement.selectSingleNode("employee/name");
alert(oNode);

```

XpathResult 类型最后部分就是布尔类型、数字类型和字符串类型。其中，每个结果类型中都用相应的 booleanValue、numberValue 和 stringValue 来返回单个值。对于布尔类型，一般返回 true，只需有一个节点匹配了 XPath 表达式，否则返回 false：

```

var oEvaluator = new XPathEvaluator();
var oResult = oEvaluator.evaluate("employee/name", oXmlDom.documentElement, null,
    XPathResult.BOOLEAN_TYPE, null);
alert(oResult.booleanValue);

```

在此例中，如果任何节点匹配了 "employee/name"，booleanValue 特性都等于 true。

对于数字类型，XPath 表达式必须使用返回数字的 XPath 函数，诸如 count()，它可用于计算匹配给定模式的节点的个数：

```

var oEvaluator = new XPathEvaluator();
var oResult = oEvaluator.evaluate("count(employee/name)", oXmlDom.documentElement,
    null, XPathResult.NUMBER_TYPE, null);
alert(oResult.numberValue);

```

这段代码输出了匹配模式 "employee/name" (2 个)。如果不使用这种特殊的 XPath 函数便使用这个方法，numberValue 则等于 NaN。

对于字符串类型，evaluate() 方法查找第一个匹配 XPath 表达式的节点，然后返回第一个子节点的值（假设第一个子节点是文本节点）。否则，结果是空字符串。例：

```

var oEvaluator = new XPathEvaluator();
var oResult = oEvaluator.evaluate("employee/name", oXmlDom.documentElement, null,
    XPathResult.STRING_TYPE, null);

alert(oResult.stringValue);

```

前面的代码将输出 "Nicholas C. Zakas"，因为这是在 <employee> 元素下 <name> 元素中的第一个文本节点。

如果觉得这比较危险，可以使用 XPathResult.ANY_TYPE。通过指定结果类型，使 evaluate() 根据 XPath 表达式返回最合适的结果类型。一般来说，这个结果类型是个布尔值，字符串值或者是无序的节点迭代子。要判断返回的结果类型，可使用 resultType 特性：

```

var oEvaluator = new XPathEvaluator();
var oResult = oEvaluator.evaluate("employee/name", oXmlDom.documentElement, null,
                                  XPathResult.STRING_TYPE, null);

if (oResult != null) {
    switch(oResult.resultType) {
        case XPath.STRING_TYPE:
            //handle string type
            break;
        case XPath.NUMBER_TYPE:
            //handle number type
            break;
        case XPath.BOOLEAN_TYPE:
            //handle boolean type
            break;
        case XPath.UNORDERED_NODE_ITERATOR_TYPE:
            //handle unordered node iterator type
            break;
        default:
            //handle other possible result types
    }
}

```

你已经看出来了，Mozilla 中的 XPath 计算远比 IE 的复杂，当然也更加强大。通过使用自定义的 `selectNodes()` 和 `selectSingleNode()` 方法，可以在两种浏览器上使用同样的代码进行 XPath 计算。

15.3 浏览器中的 XSLT 支持

XML 的姊妹语言——XSLT（可扩展样式表语言转换）可以对 XML 进行操作，将其转换成任何基于文本的形式。目前，很多开发人员都用 XSLT 将 XML 转换成 HTML，当然，这只是其中一种用途（见图 15-1）。

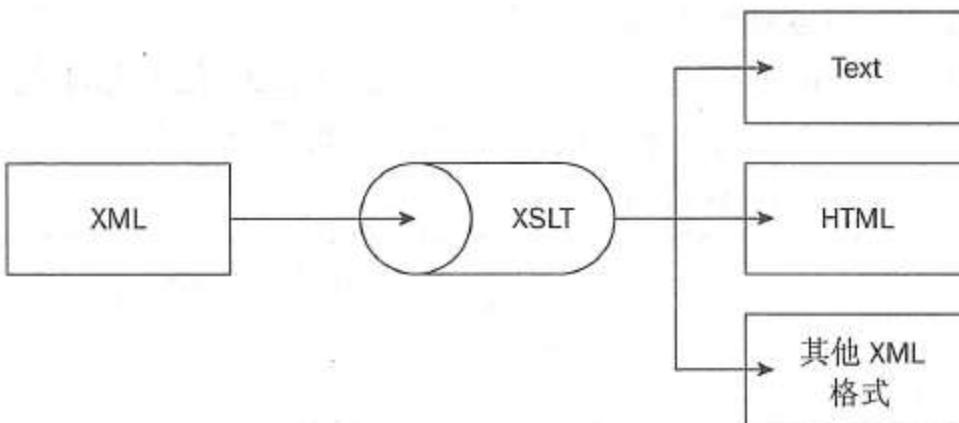


图 15-1

XSLT 文件称为样式表，由一些模板组成。模板属于 XML 的某个特性的一部分（使用 XPath 指定），它可以决定为这一部分输出什么文本。通过为不同的元素和条件定义模板，XSLT 样式表变成了一种 XML 解析器。例如，考虑前面用过的 XML：

```
<?xml version="1.0"?>
<employees>
    <employee title="Software Engineer">
        <name>Nicholas C. Zakas</name>
    </employee>
    <employee title="Salesperson">
        <name>Jim Smith</name>
    </employee>
</employees>
```

现在假设你想将这个雇员列表按照以下的 HTML 格式显示：

```
<html>
    <head>
        <title>Employees</title>
    </head>
    <body>
        <ul>
            <li>Nicholas C. Zakas, <em>Software Engineer</em></li>
            <li>Jim Smith, <em>Salesperson</em></li>
        </ul>
    </body>
</html>
```

其实，就是从内容中拉出<name/>元素的内容，然后将其放入一个无序列表中。然如要将<employee/>的 title 特性取出，并放在元素内的名字旁边，则可以创建一个 XSLT 样式表：

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

    <xsl:output method="html" />

    <xsl:template match="/">
        <html>
            <head>
                <title>Employees</title>
            </head>
            <body>
                <ul>
                    <xsl:apply-templates select="*" />
                </ul>
            </body>
        </html>
    </xsl:template>

    <xsl:template match="employee">
        <li><xsl:value-of select="name" />, <em><xsl:value-of select="@title" /></em></li>
    </xsl:template>

</xsl:stylesheet>
```

可以看到，XSLT 其实就是另一个基于 XML 的语言。文档元素是<xsl:stylesheet/>，它还指定了 XSLT 使用的版本（1.0）以及命名空间 URL。缺少这个信息，XSLT 处理器则无法正确使用样式表。

下面一行包含<xsl:output/>元素，它指定了输出处理的规则。对于 method 特性，有三种可能的值：html、xml 和 text。使用 html 时，解析器将输出作为 HTML 对待，也就是说不会应用严格的 XML 规则；xml 强制对输出应用所有的 XML 规则，同时 text 只输出包含在元素之外的内容。

下面讨论模板。第一个模板匹配了文档元素，由 match=“/”.; 制定；/ XPath 表达式总是指向文档元素。在模板中还有 HTML 代码，直到<xsl:apply-templates/>元素，让解析器为子节点（这是匹配所有子节点的 XPath 表达式）应用任何匹配模板。因为已定义了匹配这个模式的模板，所以处理继续进行。

在第二个模板中，注意元素。后面紧跟着<xsl:value-of/>元素，它用于从源 XML 中输出一个值。select 特性是另一个 XPath 表达式——name，指示输出<name/>元素的值（包含其中的文本）。然后，是一个逗号，然后是起始标签，后面是另一个<xsl:value-of/>元素。这次，select 特性指向<employee/>的 title 特性，这样转换器就输出了 title 特性值。

XSLT 样式表应用于 XML 文件时，就可以出现前面的 HTML 结果。尽管例子很简单，不过还是显示了 XSLT 的一些独特能力。

如果想学习 XSLT，请参阅 *XSLT 2.0: Programmer's Reference, 3rd Edition*(Wiley 出版社 ISBN 0-7645-6909-0)。

15.3.1 IE 中的 XSLT 支持

从 MSXML 3.0 开始，IE 就完全支持 XSLT 1.0 了。如果你还在使用 IE 5.0 或者 5.5，必须手工安装新版的 MSXML；如果在使用 IE 6.0，那么就表示你至少已经有了 MSXML 3.0。

最简单的进行 XSLT 转换的方法是，分别将 XML 源代码和 XSLT 文件载入各自的 DOM，并使用特有的 transformNode() 方法：

```
oXmlDom.load("employees.xml");
oXslDom.load("employees.xslt");
var sResult = oXmlDom.transformNode(oXslDom);
```

这个例子中的一个 DOM 载入了 XML，而另一个 DOM 载入了 XSLT 样式表（注意 XSLT 也可以载入 XML DOM 因为它也是一种 XML 格式）。然后，第三行调用文档的 transformNode() 方法，并将包含 XSLT 代码的 DOM 作为唯一的参数传给它。然后在变量 sResult 中放入转换后的文本结果。

你无需从文档层次开始转换；每一个节点都有个 transformNode() 方法。下面的均有效：

```
sResult = oXmlDom.documentElement.transformNode(oXslDom);
sResult = oXmlDom.documentElement.childNodes[1].transformNode(oXslDom);
sResult = oXmlDom.getElementsByTagName("name")[0].transformNode(oXslDom);
sResult = oXmlDom.documentElement.firstChild.lastChild.transformnode(oXslDom);
```

如果在非文档元素中调用 transformNode()，则从这个点开始转换，但是 XSLT 样式表能访问包含这个节点的整个 XML 文档。

在 IE 中使用 XSLT 的另外一个比较复杂的方法是，使用 XSL 模板和处理器。这种方法必须

使用 MSXML 的其他几个 ActiveX 控件。首先，XSLT 文件必须载入到一个自由线程 DOM 文档中（行为和普通 DOM 一样，但是线程是安全的）。

```
var oXslDom = new ActiveXObject("MSXML2.FreeThreadedDOMDocument");
oXslDom.async = false;
oXslDom.load("employees.xsl");
```

自由线程 DOM 文档建立并加载好后，必须将其分配到 XSL 模板中，这是另一个 ActiveX 对象：

```
var oTemplate = new ActiveXObject("MSXML2.XSLTemplate");
oTemplate.stylesheet = oXslDom;
```

然后，可以用 XSL 模板来创建一个 XSL 处理器（当然也是另一个 ActiveX 对象）：

```
var oProcessor = oTemplate.createProcessor();
```

创建处理器后，将 `input` 特性设置为要进行转换的 XML DOM 节点，然后调用 `transform()` 方法：

```
oProcessor.input = oXmlDom;
oProcessor.transform();
```

然后从 `output` 特性中访问结果字符串：

```
var sResult = oProcessor.output;
```

这些代码模仿了 `transformNode()` 的功能。你肯定在想，如果两种方法都一样，为什么还会有人使用 XSL 模板/处理器的方法。答案是处理器允许你更多地控制 XSLT。

例如，XSLT 样式表可以接受传入的参数，并将其用作局部变量。考虑下面的样式表：

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

    <xsl:output method="html" />
    <xsl:param name="message" />

    <xsl:template match="/">
        <html>
            <head>
                <title>Employees</title>
            </head>
            <body>
                <ul>
                    <xsl:apply-templates select="*" />
                </ul>
                <p>Message: <xsl:value-of select="$message" /></p>
            </body>
        </html>
    </xsl:template>

    <xsl:template match="employee">
        <li><xsl:value-of select="name" />, <em><xsl:value-of select="@title"
/></em></li>
    </xsl:template>
</xsl:stylesheet>
```

这个样式表添加了两行代码。第一行是<xsl:param>元素，它定义了 message 的参数。第二行通过<xsl:value-of>元素输出 message（美元符号表示这个是局部变量，而不是特性）。

要设置 message 的值，可在调用 transform() 前使用 addParameter() 方法。AddParameter() 方法有两个参数，要设置的参数的名称（与<xsl:param>中指定的一样）以及分配给它的值（一般是字符串，不过也可以是数字或者是布尔值）：

```
oProcessor.input = oXmlDom.documentElement;
oProcessor.addParameter("message", "Hello World!");
oProcessor.transform();
```

为参数设置一个值后，输出就变成这样了：

```
<html>
  <head>
    <title>Employees</title>
  </head>
  <body>
    <ul>
      <li>Nicholas C. Zakas, <em>Software Engineer</em></li>
      <li>Jim Smith, <em>Salesperson</em></li>
    </ul>
    <p>Message: Hello World!</p>
  </body>
</html>
```

475

可以看到，通过 JavaScript 传入的值正确地输出了 HTML 结果。这样，就可以通过加入基于参数的不同行为使样式表更加有弹性。

另一个 XSL 处理器的高级特性是，设置操作的模式的能力。在 XSLT 中，可以定义模板的模式。定义 mode 后，除非<xsl:apply-template />指定按照 mode 特性进行调用，否则模板不会运行。例如：

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="html" />

  <xsl:param name="message" />

  <xsl:template match="/">
    <html>
      <head>
        <title>Employees</title>
      </head>
      <body>
        <ul>
          <xsl:apply-templates select="*" />
        </ul>
        <p>Message: <xsl:value-of select="$message" /></p>
      </body>
    </html>
  </xsl:template>
```

```

<xsl:template match="employee">
    <li><xsl:value-of select="name" />, <em><xsl:value-of select="@title" /></em></li>
/></xsl:template>

<xsl:template match="employee" mode="position-first">
    <li><em><xsl:value-of select="@title" /></em>, <xsl:value-of select="name" /></li>
/></xsl:template>

</xsl:stylesheet>

```

这个样式表定义了 mode 特性为 "position-first" 的模板（注意可以任意地命名；无任何预定义的模式）。在这个模板中，首先输出雇员的位置，然后输出雇员的名称。为使用这个模板，`<xsl:apply-templates/>` 元素必须也将其 mode 设置为 "position-first"。如果使用这个样式表，它的输出也与前面一样，首先显示雇员名称，然后显示位置。但是，如果用这个样式表，并使用 JavaScript 将模式设置为 "position-first"，则先输出雇员的位置：

```

oProcessor.input = oXmlDom;
oProcessor.addParameter("message", "Hello World!");
oProcessor.setStartMode("position-first");
oProcessor.transform();

```

476

`setStartMode()` 方法只接受一个参数，要设置的模式。与 `addParameter()` 一样，必须在 `transform()` 之前调用。

如果要使用同一个样式表进行多次转换，则可在每次转换后重置处理器。调用 `reset()` 方法，输入和输出特性则会被清除，处理器又可以使用了。

```

oProcessor.reset();

```

因为处理器已经编译过 XSLT 样式表，这要比重复调用 `transformNode()` 快得多。

MSXML 只支持 XSLT 1.0。MSXML 的开发自从转到.NET Framework 后就停止了。可能，在未来的某天，JavaScript 会允许访问 XML 和 XSL 的.NET 对象。

15.3.2 Mozilla 中 XSLT 支持

从 Mozilla 1.2 开始，JavaScript 开发人员就可用 `XSLTProcessor` 对象来进行客户端的 XSLT 转换了。这个对象使用 Mozilla 内置的 XSLT 处理器——Transformiix。

转换过程的第一步是将 XML 和 XSLT 载入 DOM：

```

oXmlDom.load("employees.xml");
oXslDom.load("employees.xslt");

```

然后，创建 `XSLTProcessor` 并使用 `importStylesheet()` 方法来分配 XSLT DOM：

```

var oProcessor = new XSLTProcessor();
oProcessor.importStylesheet(oXslDom);

```

最后一步是调用 `transformToDocument()` 或者 `transformToFragment()`，并以 `XML DOM` 为参数，最后返回结果。`transformToDocument()` 返回的是新的 DOM 文档，`transformToFragment()` 返回新的文档碎片，一般来说，应该使用 `transformToDocument()`，除非你想直接将结果添加到某个已经存在的文档中，后一种情况才使用 `transformToFragment()`。

使用 `transformToFragment()` 时，仅传入 XML DOM 并将结果作为另一个完整不同的 DOM。

```
var oResultDom = oProcessor.transformToDocument(oXmlDom);
alert(oResultDom.xml);
```

使用 `transformToFragment()` 时，传入 XML DOM 和要添加到结果的文档。这确保了新的
477 文档碎片在目标文档中是有效的。

```
var oResultFragment = oProcessor.transformToDocument(oXmlDom, document);
var oDiv = document.getElementById("divResult");
oDiv.appendChild(oResultFragment);
```

在前面的例子中，处理器创建了由 `document` 对象所有的碎片。这使得碎片添加到了页面中的`<div>`元素中。

在 XSLT 的输出方法是 HTML 或者 XML 时，这些当然很有意义，但如果输出的是文本方式呢？要解决这个问题，Mozilla 创建了单个元素的 XML 文档`<transformiix:result/>`，包含了所有输出的文本。所以，从 XSLT 文件中使用文本输出也产生有效的文档或者文档碎片。

记住这一点，就可为 Mozilla 创建 `transformNode()` 方法了：

```
Node.prototype.transformNode = function (oXslDom) {
    var oProcessor = new XSLTProcessor();
    oProcessor.importStylesheet(oXslDom);

    var oResultDom = oProcessor.transformToDocument(this);
    var sResult = oResultDom.xml;

    if (sResult.indexOf("<transformiix:result") > -1) {
        sResult = sResult.substring(sResult.indexOf(">") + 1,
                                    sResult.lastIndexOf("<"));
    }

    return sResult;
};
```

这个方法是用给定的 XML DOM 创建结果文档。然后使用 `xml` 特性（本章前面定义的）将结果 XML 代码存储到 `sResult` 中。然后检查代码是否包含`<transformiix:result/>`。如果有，则 XML 部分被去除（只要取出第一个大于号和最后一个小于号之间的文本）。最后，返回 `sResult`。用这个方法，你可以创建在 Mozilla 和 IE 中都能使用的代码：

```
var oXmlDom = new XmlDom();
var oXslDom = new XmlDom();

oXmlDom.async = false;
oXslDom.async = false;
```

```

oXmlDom.load("employees.xml");
oXslDom.load("employees.xslt");

alert(oXmlDom.transformNode(oXslDom));

```

也可以为 Mozilla 中的 XSLTProcessor 设置 XSLT 参数。`setParameter()`方法接受三个参数：命名空间 URI，参数本地名称及要设置的值。一般来说，命名空间 URI 为 null，本地名称就是参数的名称，这个方法必须在 `transformToDocument()` 或者 `transformToFragment()` 方法之前调用。

478

```

var oProcessor = new XSLTProcessor()
oProcessor.importStylesheet(oXslDom);
oProcessor.setParameter(null, "message", "Hello World!");
var oResultDom = oProcessor.transformToDocument(oXmlDom);

```

还有另外两个方法与参数有关，`getParameter()` 和 `removeParameter()`，分别用于获取某个参数当前的值和删除参数值。它们都以一个命名空间 URI（也可以为 null）和参数的本地名称为参数：

```

var oProcessor = new XSLTProcessor()
oProcessor.importStylesheet(oXslDom);
oProcessor.setParameter(null, "message", "Hello World!");

alert(oProcessor.getParameter(null, "message")); //outputs "Hello World!"
oProcessor.removeParameter(null, "message");

var oResultDom = oProcessor.transformToDocument(oXmlDom);

```

这些方法不常用，主要是为了方便才提供的。

15.4 小结

这一章向你介绍了 IE 和 Mozilla 的客户端 XML 处理。第一个讨论的主题是 XML DOM 在客户端的使用，IE 中使用 MSXML 而在 Mozilla 中使用本地的 DOM 兼容的接口。还学习了两个模型的差别，并建立了减少两者之间差别的方法，以使代码更加直观。

接下来，学习了每个浏览器对 XPath 的支持，这是为在 XML 文档中为特定部分定位而设计的语言。这一节讨论了 IE 和 Mozilla 的不同实现。了解到 IE 选择了非标准的 API，而 Mozilla 选择遵循 DOM Level 3 XPath 规范。也讨论了创建标准的跨平台的方法。

最后一个讨论的主题是，使用 XSLT 进行 JavaScript XML 处理和转换的概念。学习了两种在 IE 中使用 JavaScript 完成 XSLT 转换的方法，通过 `transformNode()` 方法和 `XSLTProcessor` 对象实现。还学习了 Mozilla 的 `XSLTProcessor` 对象以及与 IE 的实现之间的比较。使用 `XSLTProcessor`，你还学习了如何在 Mozilla 创建 `transformNode()`。

记住，本章中介绍的资料只能在 IE 和 Mozilla 中运行，因为其他的浏览器尚未实现任何 JavaScript 对 XML、XPath 和 XSLT 的实现。

479

480

以前，JavaScript 完全不与服务器进行任何交互；它有时在客户端进行少量操作，然后让服务器来完成其他任务。然而，随着 Web 的发展，也要求 JavaScript 能向服务器发送数据，接受响应。这种需求下产生了几种建立通信的方法。

16.1 cookie

先不管报纸、杂志上关于它的安全性的种种说法，cookie 其实就是网站放在用户机器上的一小块信息。cookie 以前用于保存登录信息，这样用户就无需每次从同一台机器上访问受限制页面时都登录了（到处都可以看到“记住我”的复选框）。

因为 cookie 对于用户来说是唯一的，网站可以判断用户上次访问网站的时间，以及他访问了哪些页面；这就是出现隐私问题的地方。确实，cookie 可用于在某个网站跟踪你访问的页面；但无法用它来获取个人信息（如信用卡号，电子邮件地址等等），但有些菜鸟可能就会这么认为。

cookie 是第一个 JavaScript 可以利用的客户端-服务器端之间的交互手段。浏览器向服务器发送请求时，为这个服务器存储的 cookie 会与其他信息一起发送到服务器。这使得 JavaScript 可以在客户端设置一个 cookie，之后服务器端就能够读取它了。

16.1.1 cookie 的成分

呵呵，这个 cookie（曲奇）里可没有巧克力和糖。它由以下小段信息组成：

□ 名称——每一个 cookie 由一个唯一的名称代表。这个名称可以包含字母、数字和下划线。

与 JavaScript 的变量不同，cookie 的名称是不区分大小写的，所以 myCookie 和 MyCookie 是一样的。然而，实际上，最好将 cookie 名字认为是区分大小写的，因为有些服务器端软件是这样的。

□ 值——保存在 cookie 中的字符串值。这个值在存储之前必须用 encodeURIComponent() 对其进行编码，以免丢失数据或占用了 cookie。名称和值加起来的字节数不能超过 4095 字节，也就是 4KB。

□ 域——出于安全考虑，网站不能访问由其他域创建的 cookie。创建 cookie 后，域的信息会作为 cookie 的一部分存储起来。不过，虽然这不常见，还是可以覆盖这个设置以允许

另一个网站访问这个 cookie 的。

- 路径——另一个 cookie 的安全特征，路径限制了对 Web 服务器上的特定目录的访问。例如，可指定 cookie 只能从 `http://www.wrox.com/books` 中访问，这样就不能访问 `http://www.wrox.com/` 上的网页了，尽管都在同一个域中。
- 失效日期——cookie 何时应该被删除。默认情况下，关闭浏览器时，即将 cookie 删除；不过，也可以自己设置删除时间。这个值是个 GMT 格式的日期（可以使用 `Date` 对象的 `toGMTString()` 方法），用于指定应该删除 cookie 的准确时间。因此，cookie 可在浏览器关闭后依然保存在用户的机器上。如果你设置的失效日期是个以前的时间，则 cookie 被立刻删除。
- 安全标志——一个 `true/false` 值，用于表示 cookie 是否只能从安全网站（使用 SSL 和 https 协议的网站）中访问。可以将这个值设置为 `true` 以提供加强的保护，进而确保 cookie 不被其他网站访问。

16.1.2 其他安全限制

为确保 cookie 不被恶意使用，浏览器还对 cookie 的使用进行了一些限制：

- 每个域最多只能在一台用户的机器上存储 20 个 cookie；
- 每个 cookie 的总尺寸不能超过 4096 字节；
- 一台用户的机器上的 cookie 的总数不能超过 300 个。

另外，一些新的浏览器还对 cookie 进行了严格控制，可以让用户阻止所有的 cookie、阻止某些未知的网站的 cookie 或者在创建 cookie 时进行提示。

482

16.1.3 JavaScript 中的 cookie

在 JavaScript 中处理 cookie 有些复杂，因为其众所周知的蹩脚的接口。`document` 对象有个 `cookie` 特性，是包含所有给定页面可访问的 cookie 的字符串。`Cookie` 特性也很特别，因为将这个 `cookie` 特性设置为新值只会改变对页面可访问的 cookie，并不会真正改变 `cookie`（特性）本身。这是 BOM 功能的一部份，但没有任何可作为指导的规范（所以缺乏逻辑）。

要创建一个 cookie，必须按照下面的格式创建字符串：

```
cookie_name=cookie_value; expires=expiration_time; path=domain_path;
domain=domain_name; secure
```

只有字符串的第一部分（指定名称和值的字符串）是对设置 cookie 必需的，其他部分都是可选的。然后将这个字符串复制给 `document.cookie` 特性，即可创建 cookie。例如，可用以下代码设置简单的 cookie：

```
document.cookie = "name=Nicholas";
document.cookie = "book=" + encodeURIComponent("Professional JavaScript");
```

读取 `document.cookie` 的值即可访问这些 cookie，以及所有其他可以从给定页面访问的 cookie。如果在运行上面两行代码后显示 `document.cookie` 的值，则出现 "name=Nicholas;

book= Professional%20JavaScript”。即使制定了其他的 cookie 特性，如失效时间，document.cookie 也只返回每个 cookie 的名称和值，并用分号来分隔这些 cookie。

因为创建和读取 cookie 均需记住它的格式，大部分开发人员用函数来处理这些细节。创建 cookie 的函数很简单：

```
function setCookie(sName, sValue, oExpires, sPath, sDomain, bSecure) {
    var sCookie = sName + "=" + encodeURIComponent(sValue);

    if (oExpires) {
        sCookie += "; expires=" + oExpires.toGMTString();
    }

    if (sPath) {
        sCookie += "; path=" + sPath;
    }

    if (sDomain) {
        sCookie += "; domain=" + sDomain;
    }

    if (bSecure) {
        sCookie += "; secure";
    }

    document.cookie = sCookie;
}
```

483

这个 setCookie() 函数可以根据传入的参数建立 cookie 字符串。因为只有前两个参数是必需的，所以函数在把参数传给 cookie 字符串前，要对参数进行检测，以确保前两个参数是存在的。第三个参数应该是 Date 对象，这样才能调用 toGMTString() 方法。在函数的尾部，将 document.cookie 设置为这个 cookie 字符串。函数的调用方法如下：

```
setCookie("name", "Nicholas");
setCookie("book", "Professional JavaScript", new Date(Date.parse("Jan 1, 2006")));
setCookie("message", "Hello World!", new Date(Date.parse("Jan 1, 2006")),
          "/books", "http://www.wrox.com", true);
```

下一个函数 getCookie()，通过传入的名称获取 cookie 的值：

```
function getCookie(sName) {

    var sRE = "(?:; )?" + sName + "=([^\;]*);?";
    var oRE = new RegExp(sRE);

    if (oRE.test(document.cookie)) {
        return decodeURIComponent(oRE["$1"]);
    } else {
        return null;
    }
}
```

这个函数使用了通过 cookie 名称建立的正则表达式。由于 cookie 字符串的格式，正则表达式是从 `document.cookie` 中抽取特定值最方便的方法。如果只有一个 cookie，字符串就只有一个名称和值，而值就是等于号后面的所有字符。如果后面还有 cookie，则用分号进行分隔，也就是说 cookie 的值（除最后一个外）包含等于号之后，分号之前的所有字符。正则表达式使得使用一个捕获性分组来获取 cookie 值变得很容易。可这样获取 cookie 值：

```
var sName = getCookie("name");
var sBook = getCookie("book");
var sMessage = getCookie("message");
```

最后一个函数是 `deleteCookie()`，用于从系统中立即删除一个 cookie。前面提到，将 cookie 的失效时间设置为过去的一个时间即可实现。但是，要删除 cookie 必须给出与创建它时一样的路径和域信息，所以参数中也必须有：

```
function deleteCookie(sName, sPath, sDomain) {
    setCookie(sName, "", new Date(0), sPath, sDomain);
}
```

因为 `setCookie()` 设置的信息和 `deleteCookie()` 需要的信息一样，可以直接使用 `setCookie()` 并传入一个过去的失效时间（在这里是 1970 年 1 月 1 日）。

使用这些函数，就能很方便地使用 JavaScript 操作 cookie。即使是服务器创建的 cookie，JavaScript 也可以读取它们，这才是真正最有威力之处。

484

16.1.4 服务器端的 cookie

当然，使用 cookie 进行客户端-服务器端的通信还需要服务器端的额外逻辑。诸如 JSP、ASP.NET 和 PHP 之类的服务器端技术都提供了内置的读取、写入以及其他处理 cookie 的功能。使用 JavaScript 和这些服务器端语言之一，就可以进行通信了。

1. JSP

Java Server Pages (JSP) 提供了非常简单的处理 cookie 的接口。`request` 对象——会在执行 JSP 时自动初始化，有一个可以返回一个 `Cookie` 对象的数组的 `getCookies()` 方法。每个 `Cookie` 对象有以下方法（摘自 Javadoc 文档）：

- `getComment()`——返回描述 cookie 目的的注释，如果 cookie 没有注释则为 null；
- `getDomain()`——返回为 cookie 设置的域名称；
- `getMaxAge()`——返回 cookie 的最大寿命，单位为秒；默认情况下为 -1，表示 cookie 会持续到浏览器关闭；
- `getName()`——返回 cookie 的名称；
- `getPath()`——返回服务器为浏览器提供的返回 cookie 的路径；
- `getSecure()`——如果浏览器只通过安全协议发送 cookie 则为 true，否则为 false；
- `getValue()`——返回 cookie 的值；
- `getVersion()`——返回与 cookie 相关的协议的版本；

- `setComment(String purpose)`——指定描述 cookie 目的的注释;
- `setDomain(String pattern)`——指定给出 cookie 的域;
- `setMaxAge(int expiry)`——指定 cookie 的最大寿命, 单位为秒;
- `setPath(String uri)`——指定浏览器返回 cookie 的路径;
- `setSecure(boolean flag)`——指示浏览器 cookie 是否只能通过安全协议, 如 HTTP 或者 SSL 来传送;
- `setValue(String newValue)`——创建 cookie 后, 为 cookie 指定一个新值;
- `setVersion(int v)`——指定与 cookie 相关的协议的版本。

要获取指定的 cookie, 也需创建一个函数, 并用它迭代每个 cookie 来判断最需要哪个:

```
<%!
public static Cookie getCookie(HttpServletRequest request, String name) {
    Cookie[] cookies = request.getCookies();

    if (cookies != null) {
        for (int i=0; i < cookies.length; i++) {
            if (cookies[i].getName().equals(name)) {
                return cookies[i];
            }
        }
    } else {
        return null;
    }
}
%>
```

把这段代码插到 JSP 后, 即可通过以下代码来获取 cookie 的值:

```
<%
Cookie nameCookie = getCookie(request, "name");
System.out.println("Name is " + nameCookie.getValue());
%>
```

在 JSP 中创建 cookie 也相当简单。要创建新的 cookie, 只需实例化一个新的 `Cookie` 对象并为其传入名称和值。然后, 用 `response` 对象的 `addCookie()` 方法将其添加到用户的系统中:

```
<%
Cookie nameCookie = new Cookie("name", "Nicholas");
response.addCookie(nameCookie);
%>
```

另外, 还可以在存储之前为 cookie 添加额外的信息:

```
<%
Cookie nameCookie = new Cookie("name", "Nicholas");
nameCookie.setDomain("http://www.wrox.com");
nameCookie.setPath("/books");
response.addCookie(nameCookie);
%>
```

要删除指定的 cookie, 首先需获取它, 然后将其失效日期设置为 0 (JSP 的 cookie 用毫秒数作为失效时间值):

```
<%
Cookie cookieToDelete = getCookie("name");
cookieToDelete.setMaxAge(0);
response.addCookie(cookieToDelete);
%>
```

2. ASP.NET

ASP.NET 处理 cookie 的方式与 JSP 很相似。Request 对象（为每个 ASP.NET 页面自动创建的）包含一个 Cookies 集合，可以从中读出所有的 cookie。每个 cookie 都是 HttpCookie 的一个实例，它包含以下特性：

- Name——cookie 的名称；
- Value——cookie 的值；
- Expires——cookie 失效的日期；
- Path——cookie 的路径；
- Domain——cookie 的域；
- Secure——表示 cookie 是否为加强安全的布尔值。

486

其他的 HttpCookie 的特性是在 cookie 存储多个值时使用的，但是这些 cookie 不能在本章关注的 JavaScript 应用中正常工作，所以在此就不列出了。

可以通过 Cookies 集合中的 cookie 名称来读取一个 cookie：

```
Dim cookie as HttpCookie
Dim cookieValue as String
cookie = Request.Cookies("name")
cookieValue = cookie.Value
```

在此例中，读取了名为 "name" 的 cookie，并将其值分配给 cookieValue。

可通过创建一个 HttpCookie 的新实例并将 cookie 的名称和值传递给其构造函数来创建新的 cookie。然后，在调用 Response.SetCookie() 进行保存之前，修改 cookie 的其他特性。

```
Dim cookie as HttpCookie = New HttpCookie("name", "Nicholas");
cookie.Expires = #1/1/2006#;
Response.SetCookie(cookie);
```

注意，Expires 特性是个 DateTime 对象，而在 JSP 中是个数值。

要删除 cookie，只需将 Expires 特性设置为以前的时间：

```
cookie.Expires = DateTime.Now.AddDays(-1);
```

3. PHP

PHP 的 cookie 函数十分直观。PHP 用 setcookie() 函数来创建 cookie，它的参数与前面定义的 JavaScript 的 setCookie() 函数的一样。主要的区别是失效日期是个数值，与 JSP 一样：

```
bool setcookie ( string name [, string value [, int expire [, string path
[, string domain [, bool secure]]]]])
```

调用函数也很简单：

487

```
<?php
    setcookie("name", "Nicholas");
    setcookie("book", "Professional JavaScript");
?>
```

在 PHP 中设置 cookie 的一个限制是，必须在向客户端输出内容前完成 cookie 操作。这与 header() 函数类似。例如，下面的代码无效：

```
<html>
    <head>
<?php
    setcookie("name", "Nicholas");
    setcookie("book", "Professional JavaScript");
?>
```

在 PHP 中，用以 cookie 的名称作为键的\$_COOKIE 结合数组来获取 cookie 的值：

```
<?php
    $name = $_COOKIE["name"];
    $book = $_COOKIE["book"];
?>
```

注意，在下一个页面载入前不能获取使用 setcookie() 创建的 cookie 的值，也就是说，下面的代码是无效的：

```
<?php
    setcookie("name", "Nicholas");
    setcookie("book", "Professional JavaScript");
    $name = $_COOKIE["name"];
    $book = $_COOKIE["book"];
?>
```

要在 PHP 中删除 cookie，只需使用失效时间被设置为 0 的 setcookie() 方法：

```
<?php
    setcookie("name", "", 0);
?>
```

16.1.5 在客户端与服务器端之间传递 cookie

与 JavaScript 的交互，常常是服务器在发送响应前先创建一个 cookie（也许在 servlet 或者其他运行在服务器上的应用中）。然后带有用于获取 cookie 值的 JavaScript 的页面就会载入。

例如，假设你在网站上创建一个反馈表单，上面要求用户输入名字、电子邮件地址及反馈消息。你可以让用户将他们的名字和电子邮件地址保存下来，以便他们下次访问并要留下反馈时，可以无需再输入这两个字段。一般来说，常常会放“记住我”（Remember Me）复选框，如下：

```
<form name="feedbackForm" method="post" action="submitfeedback.php">
    <p>Name: <input type="text" name="personName" /><br />
    E-mail Address: <input type="text" name="personEmail" /><br />
    Feedback: <textarea rows="10" cols="50" name="feedbackText">
        </textarea><br />
    <input type="checkbox" name="rememberMe" value="yes" /> Remember Me<br />
    <input type="submit" value="Submit Feedback" />
</form>
```

488

服务器端代码（这个例子是用 PHP 写的）如下：

```
<?php
    //send e-mail
    mail("you@yourdomain.com", "User Feedback", $feedbackText,
        "From: feedback@{$_SERVER['SERVER_NAME']}");

    //if flag is set, set cookies
    if ($rememberMe == "yes") {
        setcookie("personName", $personName, time() + 1000 * 60 * 60 * 24 * 365);
        setcookie("personEmail", $personEmail, time() + 1000 * 60 * 60 * 24 * 365);
    }
?>
<!-- thank you message goes here -->
```

PHP 代码首先将反馈邮件用 PHP 的 `mail()` 函数发送出去，然后检查用户是否希望记住自己的信息，也就是说 `$remember` 是否等于 `yes`。如果是，就用 cookie 来保存用户的名字和邮件地址。两个 cookie 都设置为当前时间的一年之后（使用 PHP 的 `time()` 函数并加上 365 天的毫秒数， $1000 \text{ 毫秒} \times 60 \text{ 秒} \times 60 \text{ 分钟} \times 24 \text{ 小时} \times 365 \text{ 天}$ ）。

在 PHP 中，表单字段的值可以直接通过与表单字段名称相同的变量来访问，所以名为 `personName` 的文本框的值可通过字符串变量 `$personName` 来访问¹。

记住，还需在返回表单的页面上包含 `getCookie()` 函数来获取每个 cookie 的值。然后，在 `onload` 事件处理函数中查找是否有包含用户信息 cookie，如果找到了，则将信息填到表单中。

```
window.onload = function () {

    var sName = getCookie("personName");
    var sEmail = getCookie("personEmail");

    if (sName && sEmail) {
        var oForm = document.forms["feedbackForm"];
        oForm.personName.value = sName;
        oForm.personEmail.value = sEmail;
    }
};
```

这段 JavaScript 代码用于检查用户的名字和邮件地址是否已存储在 cookie 中。如果是，就将两个值分别填到表单中相应的字段中。现在，用户就无需每次发送反馈时都填写联系信息了（至少一年）。

489

16.2 隐藏框架

长久以来，开发人员经常使用的一个小技巧就是隐藏框架方法。基本的概念，创建一个可用 JavaScript 与服务器进行通信的 0 像素高的框架（这样，就隐藏了）。这种通信方式要求两部分内容：用于处理客户端通信的 JavaScript 对象和在服务器端处理通信的特殊页面。

1. 这是 PHP 配置中的 `register_global` 设置项为 `On` 的效果，高版本的 PHP 中默认不打开这个功能。

——译者注

一个很简单的例子，首先是框架集：

```
<html>
  <head>
    <title>Hidden Frame Example</title>
  </head>
  <frameset rows="*,0">
    <frame src="HiddenFrameExampleMain.htm" name="mainFrame" />
    <frame src="HiddenFrameExampleBlank.htm" name="hiddenFrame" />
  </frameset>
</html>
```

这个框架集由两行组成，第二个框架高度为 0（在 Netscape 4.x 中，框架不能为 0 像素高，所以框架还是可见的）。第一个框架用于用户进行交互；第二个框架用于与服务器交互。默认情况下，第二个框架载入的是空白页面。

在第一个框架中，定义了两个函数：一个用于向服务器发送请求，另一个用于用户处理响应。发送请求的函数称为 `getServerInfo()`，它为隐藏框架分配 URL：

```
function getServerInfo() {
  parent.frames["hiddenFrame"].location.href = "HiddenFrameExampleCom.htm"
}
```

函数当然也可以给请求添加额外的查询字符串。

第二个函数 `handleResponse()`，在隐藏框架从服务器返回页面之后调用。函数可以任意处理返回的数据，在此例中，只是将数据显示在警告框中：

```
function handleResponse(sResponseText) {
  alert("The server returned: " + sResponseText);
}
```

处理隐藏请求的页面必须输出一个普通的 HTML 页面，其中有个`<textarea/>`元素，包含返回的数据。使用`<textarea/>`可以方便地处理多行数据。这个页面必须在主框架中调用 `handleResponse()` 函数：

```
490
<html>
  <head>
    <title>Hidden Frame Example (Response)</title>
    <script type="text/javascript">
      window.onload = function () {
        parent.frames[0].handleResponse(
          document.forms["formResponse"].result.value);
      };
    </script>

  </head>
  <body>
    <form name="formResponse">
      <textarea name="result">This is some data coming from the
server.</textarea>
    </form>
  </body>
</html>
```

这个窗口的一个重要部分是 `onload` 事件处理函数，它在第一个框架中调用 `handleResponse()` 函数，并将包含在 `<textarea>` 元素内的值传给这个函数。而且，即使出现错误，也必须保证调用了 `handleResponse()` 函数，否则主框架中的 JavaScript 会挂起并一直等待回复。

为了展示，前面的代码已经直接将数据插入到 `<textarea>` 中；实际上，数据应该是由某种服务器端逻辑输出的。

在主框架中调用 `getServerInfo()` 函数时，通过隐藏框架发送请求，数据是通过 `handleResponse()` 函数传送回来并在警告框中显示的。这是个十分简单的例子，不过已经可以勾勒出大体的思想了。另外，这种客户端-服务器端通信可以在任何支持框架和 JavaScript 的浏览器上运行（包括较老的像 Netscape Navigator 4.x 的浏览器）。

使用 `iframe`

隐藏框架方法的演变为 HTML 引入了 `iframe`。`iframe` 是可以直接插在 HTML 文档的任意位置的一个框架，完全与框架集无关。此后，开发人员更改了隐藏框架方法，使用即时创建隐藏 `iframe` 来达到与服务器进行通信的目的。

要利用 `iframe`，必须对 `getServerInfo()` 函数做一些修改：

```
var oHiddenFrame = null;

function getServerInfo() {
    if (oHiddenFrame == null) {
        oHiddenFrame = document.createElement("iframe");
        oHiddenFrame.name = "hiddenFrame";
        oHiddenFrame.id = "hiddenFrame";
        oHiddenFrame.style.height = "0px";
        oHiddenFrame.style.width = "0px";
        oHiddenFrame.style.position = "absolute";
        oHiddenFrame.style.visibility = "hidden";
        document.body.appendChild(oHiddenFrame);
    }
    setTimeout(function () {
        frames["hiddenFrame"].location.href = "HiddenFrameExampleCom2.htm";
    }, 10);
}
```

491

第一个修改，添加名为 `oHiddenFrame` 的全局变量。因为同一个框架可以用来重复地进行请求，所以没有必要在每次请求时都创建新的 `iframe`。创建 `iframe` 后，使用这个全局变量来保存对 `iframe` 的引用。调用 `getServerInfo()` 时，首先通过检查 `oHiddenFrame` 的值来检查是否已创建了 `iframe`。如果不存在，则用 DOM 的 `createElement()` 方法来创建这个框架。

用 DOM 创建 `iframe` 还要注意，必须同时将 `name` 和 `id` 设置为 "hiddenFrame" 以保证在大部分浏览器中都能正常工作（有些使用 `name`，有些使用 `id`）。下面，框架的宽和高必须指定为 0，定位方式为绝对，可见性设置为 "hidden"。这些设置是为了确保新添加的框架不会影响文档的显示。最后，将 `iframe` 添加到文档主体中。

iframe 创建好并添加后，大部分浏览器（尤其是 Mozilla 和 Opera）会需要一小段时间（毫秒级）来认出新的框架并将其添加到帧集合中。考虑到这个问题，应该用 `setTimeout()` 函数使发送请求的操作等待 10 毫秒。到请求执行时，浏览器已经认出了新的框架，就可以毫无障碍地将请求发送出去了。

必须对提供响应的页面所做的唯一改动是，使用 `parent` 而非 `parent.frames[0]` 来调用 `handleResponse()`：

```
<html>
  <head>
    <title>Hidden Frame Example (Response)</title>
    <script type="text/javascript">
      window.onload = function () {
        parent.handleResponse(document.forms["formResponse"].result.value);
      };
    </script>

  </head>
  <body>
    <form name="formResponse">
      <textarea name="result">This is some data coming from the
server.</textarea>
    </form>
  </body>
</html>
```

现在调用 `getServerInfo()` 即可以获得与前面使用原始的隐藏框架技术的例子完全一样的效果。当然，这个技术要求浏览器首先支持 `iframe`，这就使得比较老的浏览器，如 Netscape

492

Navigator 4.x 不能被列入考虑范围。

16.3 HTTP 请求

现在，很多浏览器都可以直接从 JavaScript 中初始化 HTTP 请求并获取结果，完全不用隐藏框架和其他取巧的小技巧。

这个令人振奋的新功能的核心是，微软创建的 XML HTTP 请求的对象。这个对象是与 MSXML 一起出现的，直到最近它的能力才被完全发掘。XML HTTP 请求本质上是添加了额外的用于发送和接收 XML 代码的功能的普通 HTTP 请求。

要在 IE 中重新创建 XML HTTP 请求对象，还是要使用 `ActiveXObject`：

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

与 IE 中的 XML DOM 一样，XML HTTP 请求对象有多种版本，所以需要一个用于确保使用的是最新版本的函数：

```
function createXMLHTTP() {
  var arrSignatures = ["MSXML2.XMLHTTP.5.0", "MSXML2.XMLHTTP.4.0",
    "MSXML2.XMLHTTP.3.0", "MSXML2.XMLHTTP",
```

```

    "Microsoft.XMLHTTP"];

for (var i=0; i < arrSignatures.length; i++) {
    try {

        var oRequest = new ActiveXObject(arrSignatures[i]);

        return oRequest;

    } catch (oError) {
        //ignore
    }
}

throw new Error("MSXML is not installed on your system.");
}

```

创建好 XML HTTP 请求对象后，可用 `open()` 方法来指定要发送的请求。这个方法有三个参数：要发送的请求的类型（GET、POST 或者其他受服务器支持的 HTTP 方法）、请求的 URL 以及表示请求是否应该以异步方式发送的布尔值（与 XML DOM 的 `load()` 方法一样）。例如：

```
oRequest.open("get", "example.txt", false);
```

打开这个请求后，还要用 `send()` 方法将其发送出去。这个方法一定要有一个参数，不过大多数情况下都可以用 `null`：

```
oRequest.send(null);
```

如果使用同步调用（将第三个参数设置为 `false`），则 JavaScript 解释程序就会等待请求返回。493 返回响应后，`status` 特性中会放入请求的 HTTP 状态（200 是正常，404 表示没有找到指定页面等等）。同时在 `statusText` 特性中放入描述状态的信息，在 `responseText` 特性放入从服务器上接收到的文本。另外，如果返回的是 XML，还要在 `responseXML` 特性中放入一个由返回的文本构造出来的 XML DOM 对象。例如：

```

var oRequest = createXMLHTTP();
oRequest.open("get", "example.txt", false);
oRequest.send(null);
alert("Status is " + oRequest.status + " (" + oRequest.statusText + ")");
alert("Response text is: " + oRequest.responseText);

```

在此例中，从服务器获取一个纯文本文件，然后显示其内容。同时，还显示了 `status` 和 `statusText` 特性的内容。

如果在本地运行这个例子，`status` 为 0，而 `statusText` 为 "Unknown"，因为本地文件的读取并非真正的 HTTP 请求。

请求 XML 文件还有个 `responseXML` 特性：

```

var oRequest = createXMLHTTP();
oRequest.open("get", "example.xml", false);

```

```

oRequest.send(null);
alert("Status is " + oRequest.status + " (" + oRequest.statusText + ")");
alert("Response text is: " + oRequest.responseText);
alert("Tag name of document element is: " +
    oRequest.responseXML.documentElement.tagName);

```

这个例子显示了载到 responseXML 特性中的文档元素的标签名。

这个例子也不能直接在本地运行，必须在服务器上运行，因为 XML HTTP 对象通过服务器通报的 MIME 类型来判断返回的内容是否为 XML 文档。

如果要发送异步请求，必须使用 onreadystatechange 事件处理函数，并检查 readyState 特性是否等于 4（与 XML DOM 一样）。response 特性在请求完毕前是不可用的：

```

var oRequest = createXMLHTTP();
oRequest.open("get", "example.txt", true);
oRequest.onreadystatechange = function () {
    if (oRequest.readyState == 4) {
        alert("Status is " + oRequest.status + " (" + oRequest.statusText + ")");
        alert("Response text is: " + oRequest.responseText);
    }
}
oRequest.send(null);

```

494

与在同步调用中一样，status、statusText 和 responseText 特性都会被填入相应的数据。而对于异步调用，可以通过调用 abort() 方法在 readyState 变成 4 之前取消请求的操作：

```

var oRequest = createXMLHTTP();
oRequest.open("get", "example.txt", true);
oRequest.onreadystatechange = function () {
    if (oRequest.readyState == 3) {
        oRequest.abort();
    } else if (oRequest.readyState == 4) {
        alert("Status is " + oRequest.status + " (" + oRequest.statusText + ")");
        alert("Response text is: " + oRequest.responseText);
    }
}
oRequest.send(null);

```

在这个例子中，不会出现警告框，因为请求在 readyState 为 3 时被中止了。

16.3.1 使用 HTTP 首部

每个 HTTP 请求发送时都包含一组带有额外信息的首部。在我们使用浏览器时，这些首部被隐藏了，因为对于终端用户来说，这些信息是没用的。然而，这些首部信息对开发人员来说可能是很重要的，所以 XML HTTP 请求对象提供了获取和设置它们的方法。

第一个方法——getAllResponseHeaders() 方法，可以返回包含所有响应的 HTTP 首部信息的字符串。下面是由 getAllResponseHeaders() 方法返回的样例：

```
Date: Sun, 14 Nov 2004 18:04:03 GMT
Server: Apache/1.3.29 (Unix)
Vary: Accept
X-Powered-By: PHP/4.3.8
Connection: close
Content-Type: text/html; charset=iso-8859-1
```

从这段首部信息可以看出，服务器是在 Unix 上运行的 Apache，且带有 PHP 支持，文件作为 HTML 文件返回。如果要获取指定的某个首部，可以使用 `getResponseHeader()` 方法，参数为要获取的首部的名称。例如，要获取 "Server" 首部的值，可以这样做：

```
var sValue = oRequest.getResponseHeader("Server");
```

除了读取请求首部信息，还需要在发送请求前设置自己的首部信息。

可以使用 `setRequestHeader()` 方法来设置 XML HTTP 请求的首部信息。例如：

```
oRequest.setRequestHeader("myheader", "yippee");
oRequest.setRequestHeader("weather", "warm");
```

495

这里假设你已经设计了一些服务器端逻辑来根据这些首部信息提供额外的功能或者对请求的计算。

16.3.2 实现的复制品

由于 XML HTTP 请求对象已获得广大 Web 开发人员的青睐，所以其他浏览器的开发人员也做了类似的实现。Mozilla 第一个复制了这个实现，创建了名为 `XMLHttpRequest` 的 JavaScript，行为完全与微软的版本相同。Safari (1.2) 和 Opera (7.6) 也复制了 Mozilla 的实现，创建了自己的 `XMLHttpRequest` 对象。

若要创建一种统一的 XML HTTP 请求对象的创建方法，只需在页面中添加下面这个简单的包装类：¹

```
if (typeof XMLHttpRequest == "undefined" && window.ActiveXObject) {
    function XMLHttpRequest() {

        var arrSignatures = ["MSXML2.XMLHTTP.5.0", "MSXML2.XMLHTTP.4.0",
                            "MSXML2.XMLHTTP.3.0", "MSXML2.XMLHTTP",
                            "Microsoft.XMLHTTP"];

        for (var i=0; i < arrSignatures.length; i++) {
            try {

                var oRequest = new ActiveXObject(arrSignatures[i]);

                return oRequest;

            } catch (oError) {
                //ignore
            }
        }
    }
}
```

1. 这段代码在 Opera 8 及以前版本中会出现问题，可以考虑改成 `XMLHttpRequest = function()....`。——译者注

```

        throw new Error("MSXML is not installed on your system.");
    }
}

```

可以使用下面一行代码在所有支持 XML HTTP 请求对象的浏览器中创建这个对象：

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

此后，就可以像前面所描述的那样在受支持的浏览器中使用 XML HTTP 请求对象了。

16.3.3 进行 GET 请求

Web 上最常见的请求类型就是 GET 请求。每次在浏览器中输入 URL 并打开页面时，就是在向服务器发送一个 GET 请求。

496

GET 请求的参数是用问号追加到 URL 的结尾，后面跟着用&号连接起来的名称/值。例如：

```
http://www.somewhere.com/page.php?name1=value1&name2=value2&name3=value3
```

每个名称和值都必须在编码后才能用在 URL 中（在 JavaScript 中可以用 encodeURIComponent() 进行编码）。URL 最大长度为 2048 字符（2KB）。问号后面的内容称为查询字符串，这些参数可以在服务器端的页面中读取。

要是用 XML HTTP 请求对象发送一个 GET 请求，只需将 URL（包含所有的参数）传入 open() 方法，同时第一个参数设为 "get"：

```
oRequest.open("get", "http://www.somewhere.com/page.php?name1=value1", false);
```

因为参数必须追加到 URL 的末尾，所以最好用个函数来处理这个细节：

```

function addURLParam(sURL, sParamName, sParamValue) {
    sURL += (sURL.indexOf("?") == -1 ? "?" : "&");
    sURL += encodeURIComponent(sParamName) + "=" + encodeURIComponent(sParamValue);
    return sURL;
}

```

addURLParam() 函数有三个参数：要添加参数的 URL，参数名称和参数值。首先，函数检查 URL 中是否已经存在一个问号（存在就表示已有参数了）。如果没有，函数就追加一个问号，否则，追加一个&号。下面，将名称和值进行编码并添加到 URL 的末尾。最后，返回新的 URL。

函数可以用来为请求构造一个 URL：

```

var sURL = "http://www.somewhere.com/page.php";
sURL = addURLParam(sURL, "name", "Nicholas");
sURL = addURLParam(sURL, "book", "Professional JavaScript");
oRequest.open("get", sURL, false);

```

然后处理请求的方式与以前一样。

16.3.4 进行 POST 请求

第二最常用的 HTTP 请求类型便是 POST 请求。一般来说，POST 请求用于在表单中输入数据后的提交过程，因为 POST 可以比 GET 方式发送更多数据（大约 2GB）。

与 GET 请求一样，POST 请求的参数也必须进行编码，并用&进行分割，尽管这些参数不会被附加到 URL 上。发送 POST 请求时，要将参数传入 send() 方法：

```
oRequest.open("post", "page.php", false);
oRequest.send("name1=value1&name2=value2");
```

497

也最好使用一个函数来格式化 POST 请求的参数：

```
function addPostParam(sParams, sParamName, sParamValue) {
    if (sParams.length > 0) {
        sParams += "&";
    }
    return sParams + encodeURIComponent(sParamName) + "="
        + encodeURIComponent(sParamValue);
}
```

除了 addPostParam() 针对的是参数字符串而不是 URL 外，这段函数类似于 addURLParam() 函数。第一个参数是已存在的参数列表，第二个参数是参数名称，第三个参数是参数值。函数先检查参数字符串的长度是否大于 0。如果是，追加一个&号来分隔新的参数。否则，返回只有新的名称和值的参数字符串。下面是这种使用方法的简单的例子：

```
var sParams = "";
sParams = addPostParam(sParams, "name", "Nicholas");
sParams = addPostParam(sParams, "book", "Professional JavaScript");
oRequest.open("post", "page.php", false);
oRequest.send(sParams);
```

尽管它看上去是个有效的 POST 请求，实际上需要 POST 请求的服务器端页面无法正确解释这段代码。因为所有由浏览器发送的 POST 请求都将 "Content-Type" 首部设置为 "application/x-www-form-urlencoded"，所以要使用 setRequestHeader() 方法设置这个首部：

```
var sParams = "";
sParams = addPostParam(sParams, "name", "Nicholas");
sParams = addPostParam(sParams, "book", "Professional JavaScript");
oRequest.open("post", "page.php", false);
oRequest.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
oRequest.send(sParams);
```

现在这个例子就可以像由浏览器中提交表单一样正常工作了。

16.4 LiveConnect 请求

Netscape Navigator 引入了 LiveConnect 概念，一种可以让 JavaScript 与 Java 类实现交互的能力。用户必须安装 Java 运行时环境（Java Runtime Environment, JRE）才能使用这个功能，同时在浏览器中必须启用 Java。几乎所有现在的浏览器（除 IE 之外）都支持 LiveConnect，它允许访问 Java 提供的所有 HTTP 相关的库。

16.4.1 进行 GET 请求

如果你知道如何使用 Java 进行 GET 请求，则将其转换成 LiveConnect 脚本也很简单。第一

498 步是创建 `java.net.URL` 的实例:

```
function httpGet(sURL) {
    var oURL = new java.net.URL(sURL);
    //...
}
```

注意, 使用 LiveConnect 时, 必须提供类的完整名称, 包括包名, 才能实例化一个 Java 对象。创建 URL 后, 就可以打开一个输入流并使用读取器来获取数据。最好的方法是创建一个 `InputStreamReader`, 然后再基于它创建一个 `BufferedReader`:

```
function httpGet(sURL) {
    var oURL = new java.net.URL(sURL);
    var oStream = oURL.openStream();
    var oReader = new java.io.BufferedReader(new
        java.io.InputStreamReader(oStream));
    //...
}
```

创建了缓冲阅读器后, 下面要做的就是从服务器中读取数据。缓冲阅读器是逐行获取数据的, 所以要创建一个变量来将响应组成完整的文本。这个响应文本的变量 (`sResponseText`) 必须为空字符串, 而不是 `null`, 这样才能正确地使用字符串联结来构建结果:

```
function httpGet(sURL) {
    var oURL = new java.net.URL(sURL);
    var oStream = oURL.openStream();
    var oReader = new java.io.BufferedReader(new
        java.io.InputStreamReader(oStream));
    var sResponseText = "";
    var sLine = oReader.readLine();
    while (sLine != null) {
        sResponseText += sLine + "\n";
        sLine = oReader.readLine();
    }
    //...
}
```

因为缓冲阅读器返回的是每行中的文本, 所以必须追加一个换行符来确保与响应的原先的内容一样。最后一步是关闭阅读器, 并返回响应文本:

```
function httpGet(sURL) {
    var oURL = new java.net.URL(sURL);
    var oStream = oURL.openStream();
```

```

var oReader = new java.io.BufferedReader(new
java.io.InputStreamReader(oStream));
var sResponseText = "";

var sLine = oReader.readLine();
while (sLine != null) {
    sResponseText += sLine + "\n";
    sLine = oReader.readLine();
}

oReader.close();
return sResponseText;
}

```

现在这个函数可以使用前面定义的 addURLParam() 函数来发送 GET 请求：

```

var sURL = "http://www.somewhere.com/page.php";
sURL = addURLParam(sURL, "name", "Nicholas");
sURL = addURLParam(sURL, "book", "Professional JavaScript");
var sData = httpGet(sURL);

```

不幸的是，使用 LiveConnect 时，不能获取完全相同的信息，比如状态。同时，这个函数只能进行同步调用，不能创建异步调用。但是它的好处是，可以用于 Netscape Navigator 4.x 和大部分 Opera 的版本中，以及其他支持 LiveConnect 的浏览器。

与 XML HTTP 请求对象不同，LiveConnect 要求输入完整的请求的 URL，从 `http://` 开始。这是因为，这个 Java 对象没有任何解释相对 URL 的上下文。

16.4.2 进行 POST 请求

与前面讨论过的一样，POST 请求和 GET 请求在格式和行为上略有不同。然而，使用 LiveConnect 发送 POST 请求也可以像发送 GET 请求一样简单。你可以提供一个 URL 和参数字符串（使用前面介绍的 addPostParam() 函数）。然后，创建另一个 `java.net.URL` 实例。与上次不同的是，这段代码使用 `Connection` 对象来协助进行请求：

```

function httpPost(sURL, sParams) {

    var oURL = new java.net.URL(sURL);
    var oConnection = oURL.openConnection();

    //...
}

```

下面，要确定连接对象的设置。因为 POST 请求可看作是双向的，所以必须使用 `setDoInput()` 和 `setDoOutput()` 方法将连接设成接受输入和输出。另外，连接不应该使用任何缓存数据，所以要调用 `setUseCaches(false)`。与 XML HTTP 请求对象一样，还必须用 `setRequestProperty()` 方法将 "Content-Type" 设置为相应的值：

```

function httpPost(sURL, sParams) {
    var oURL = new java.net.URL(sURL);
    var oConnection = oURL.openConnection();

    oConnection.setDoInput(true);
    oConnection.setDoOutput(true);
    oConnection.setUseCaches(false);
    oConnection.setRequestProperty("Content-Type",
        "application/x-www-form-urlencoded");

    //...
}

```

连接建立后，即可获取请求的输出流。而参数是用 `writeBytes()` 方法放到输出流中的。然后，调用 `flush()` 来立刻发送数据，最后关闭这个流：

```

function httpPost(sURL, sParams) {
    var oURL = new java.net.URL(sURL);
    var oConnection = oURL.openConnection();

    oConnection.setDoInput(true);
    oConnection.setDoOutput(true);
    oConnection.setUseCaches(false);
    oConnection.setRequestProperty("Content-Type",
        "application/x-www-form-urlencoded");

    var oOutput = new java.io.DataOutputStream(oConnection.getOutputStream());
    oOutput.writeBytes(sParams);
    oOutput.flush();
    oOutput.close();

    //...
}

```

函数的下一个部分就是，从连接中取得输入流并逐行读入数据，类似于 `httpGet()` 函数。然后，关闭输入流，同时将响应文本作为函数值返回：

```

function httpPost(sURL, sParams) {
    var oURL = new java.net.URL(sURL);
    var oConnection = oURL.openConnection();

    oConnection.setDoInput(true);
    oConnection.setDoOutput(true);
    oConnection.setUseCaches(false);
    oConnection.setRequestProperty("Content-Type",
        "application/x-www-form-urlencoded");

    var oOutput = new java.io.DataOutputStream(oConnection.getOutputStream());
    oOutput.writeBytes(sParams);
    oOutput.flush();
    oOutput.close();

```

```

var sLine = "", sResponseText = "";

var oInput = new java.io.DataInputStream(oConnection.getInputStream());
sLine = oInput.readLine();

while (sLine != null){
    sResponseText += sLine + "\n";
    sLine = oInput.readLine();
}

oInput.close();

return sResponseText;
}

```

使用这个函数，即可提交 POST 请求了，如下：

```

var sParams = "";
sParams = addPostParam(sParams, "name", "Nicholas");
sParams = addPostParam(sParams, "book", "Professional JavaScript");
var sData = httpPost("http://www.somewhere.com/reflectpost.php", sParams);

```

16.5 智能 HTTP 请求

对于两种完全不同的 HTTP 请求方法，有一系列通用的函数会对开发很有帮助。首先，检测是否要使用 XML HTTP 请求对象。检测的方法是判断 XMLHttpRequest 的类型是否等于 "object"，以及 window.ActiveXObject 是否有效：

```
var bXmlHttpSupport = (typeof XMLHttpRequest == "object" || window.ActiveXObject);
```

下面要创建对象 `Http` 来存放我们的方法：

```
var Http = new Object;
```

16.5.1 `get()` 方法

第一个方法称为 `get()`，它用于对指定的 URL 进行一个 GET 请求。这个方法有两个参数：发送请求的 URL 和一个回调函数。回调函数在很多编程语言中是用于在请求结束时通知开发者的。对于 `get()` 方法，回调函数格式如下：

```
function callback_function(sData) {
    //interpret data here
}
```

传给这个回调函数的唯一参数是从 HTTP 请求中获取的数据——`sData`。然后就可以任意处置该结果了。如果用回调函数，还需考虑一系列细节。

首先，用 XML HTTP 请求对象进行 GET 请求时，使用异步请求可以很方便地设置另一个回调函数，同时在 `readyState` 为 4 时，调用上面提到的回调函数：

```

Http.get = function (sURL, fnCallback) {

    if (bXmlHttpSupport) {

        var oRequest = new XMLHttpRequest();
        oRequest.open("get", sURL, true);
        oRequest.onreadystatechange = function () {
            if (oRequest.readyState == 4) {
                fnCallback(oRequest.responseText);
            }
        }
        oRequest.send(null);

    }

    //...
};


```

这段代码使用了 JavaScript 闭包，这允许在 `onreadystatechange` 事件处理函数中调用指定的回调函数 `fnCallback`。

如果浏览器不支持 XML HTTP 请求，就必须检查是否启用了 LiveConnect。不幸地是，没有任何特性或者设置能够表示 LiveConnect 是否可用。唯一的办法是用 `navigator.javaEnabled()` 方法来保证已经启用了浏览器中的 Java，并判断 `java` 和 `java.net` 是否已定义：

```

Http.get = function (sURL, fnCallback) {

    if (bXmlHttpSupport) {

        var oRequest = new XMLHttpRequest();
        oRequest.open("get", sURL, true);
        oRequest.onreadystatechange = function () {
            if (oRequest.readyState == 4) {
                fnCallback(oRequest.responseText);
            }
        }
        oRequest.send(null);

    } else if (navigator.javaEnabled() && typeof java != "undefined"
              && typeof java.net != "undefined") {

        //LiveConnect code here
    }

};


```

判断了是否可以使用 LiveConnect 后，即可使用 `httpGet()` 函数了。如果还要模仿异步调用，可用 `setTimeout()` 来延迟一段时间，然后再调用回调函数：

```

Http.get = function (sURL, fnCallback) {

    if (bXmlHttpSupport) {


```

```

var oRequest = new XMLHttpRequest();
oRequest.open("get", sURL, true);
oRequest.onreadystatechange = function () {
    if (oRequest.readyState == 4) {
        fnCallback(oRequest.responseText);
    }
}
oRequest.send(null);

} else if (navigator.javaEnabled() && typeof java != "undefined"
&& typeof java.net != "undefined") {

    setTimeout(function () {
        fnCallback(httpGet(sURL));
    }, 10);
}
//...
};

}

```

还需做的一件事情就是，为那些浏览器不能从 JavaScript HTTP 进行请求的用户提供一个信息：

```

Http.get = function (sURL, fnCallback) {

    if (bXmlHttpSupport) {

        var oRequest = new XMLHttpRequest();
        oRequest.open("get", sURL, true);
        oRequest.onreadystatechange = function () {
            if (oRequest.readyState == 4) {
                fnCallback(oRequest.responseText);
            }
        }
        oRequest.send(null);

    } else if (navigator.javaEnabled() && typeof java != "undefined"
        && typeof java.net != "undefined") {
        setTimeout(function () {
            fnCallback(httpGet(sURL));
        }, 10);
    } else {
        alert("Your browser doesn't support HTTP requests.");
    }
}

}

```

此后，就可以在不同的浏览器中使用统一的代码发送 GET 请求：

```

var sURL = "http://www.somewhere.com/page.php";
sURL = addURLParam(sURL, "name", "Nicholas");
sURL = addURLParam(sURL, "book", "Professional JavaScript");
Http.get(sURL, function(sData) {
    alert("Server sent back: " + sData);
});

```

记住，因为 LiveConnect 要求必须输入请求的完整的 URL，所以每次都必须提供完整的 URL 给 `Http.get()` 方法，以防万一浏览器只能使用 LiveConnect。

16.5.2 `post()`方法

除了需要三个参数（URL、参数字符串和回调函数）外，`post()`方法类似于 `get()`方法。为简便起见，`post()`的回调函数的格式与 `get()`使用的一样。

方法本身明显与 `get()`很相像，使用同样的 `if..else`语句。唯一的区别是，对请求首部的设置、参数的数量以及方法发送的是一个"POST"请求：

```
Http.post = function (sURL, sParams, fnCallback) {
    if (bXmlHttpSupport) {
        var oRequest = new XMLHttpRequest();
        oRequest.open("post", sURL, true);
        oRequest.setRequestHeader("Content-Type",
            "application/x-www-form-urlencoded");
        oRequest.onreadystatechange = function () {
            if (oRequest.readyState == 4) {
                fnCallback(oRequest.responseText);
            }
        }
        oRequest.send(sParams);
    } else if (navigator.javaEnabled() && typeof java != "undefined"
        && typeof java.net != "undefined") {
        setTimeout(function () {
            fnCallback(httpPost(sURL, sParams));
        }, 10);
    } else {
        alert("Your browser doesn't support HTTP requests.");
    }
};
```

505

使用这个方法，即可在不同的浏览器中使用同一段代码来进行 POST 请求：

```
var sURL = "http://www.somewhere.com/page.php";
var sParams = "";
sParams = addPostParam(sParams, "name", "Nicholas");
sParams = addPostParam(sParams, "book", "Professional JavaScript");
oHttp.post(sURL, function(sData) {
    alert("Server sent back: " + sData);
});
```

使用这个方法还是必须提供完整的 URL。

16.6 实际使用

尽管所有的关于客户端-服务器端通信的功能都很花哨，不过还是有很多实际应用的。例如，可以创建不会卸载的搜索页面。每次提交新的搜索时，它就向服务器发送请求然后获取新数据。这就是 Amazon.com 的新搜索引擎 A9 所使用的模型 (<http://www.a9.com>)。

在第一次用 A9 搜索后，你可以点击屏幕右边的选项选择添加图片、影片、图书及其他信息。每次点击，都会在页面的新区域中出现一个载入新的搜索结果的框，它使用的就是本章前面讨论的隐藏 iframe 技术。

有些网站用 JavaScript 客户端-服务器端通信来提供搜索结果。有个这样的网站，Bitflux 的 Blog (<http://blog.bitflux.ch/>)，它的 LiveSearch 功能用 XML HTTP 请求对象来获取搜索结果。这个很酷的技术用 `onkeypress` 事件处理函数对用户在搜索框内的输入进行检测。每当输入新的字母，网站就发送一个搜索请求，并将结果以一个层的形式显示在网页上，而无需重新载入页面。

另一种对这类通信的使用是，提供 JavaScript 中无法完成的功能。例如，你可以根据数据库中的内容来校验用户的输入。使用 `onblur` 事件处理函数，就可以发送数据库的请求来判断用户输入的值是否有效，然后，如果无效即给出错误信息。可以发挥的空间无穷无尽啊！

506

16.7 小结

在本章中，学习了关于 JavaScript 客户端-服务器端通信的所有内容。本章从最原始的使用 cookie 的客户端-服务器端通信方式开始讨论。你了解到 cookie 是存储在用户机器上的一小块数据。因为 cookie 可以同时在服务器上和通过 JavaScript 进行访问，所以可以提供独特的通信方式。

然后这章讨论了如何用隐藏框架来发送请求。还讨论了通过隐藏框架和隐藏 iframe 来实现这个技术。

然后，学习了新的浏览器是如何直接从 JavaScript 向服务器发送 HTTP 请求的。介绍了 XML HTTP 请求对象，又学习了如何向服务器发送 GET 和 POST 请求。对于这些不支持 XML HTTP 请求对象的浏览器，可以使用 LiveConnect——从 JavaScript 中访问 Java 的功能，来发送 GET 和 POST 请求。

最后，学习了如何创建一个跨浏览器的 GET 和 POST 请求的执行方法。这个方法利用浏览器内置的功能来判断是使用 XML HTTP 请求还是使用 LiveConnect 进行请求。

507

508

在过去的两年中，Web 服务逐渐成为一个热门话题。由于微软对.NET 的推动，开发人员现在已经可以快速、简便地创建、部署和访问 Web 服务。其实 Web 服务的思想很简单：服务器通过互联网提供（发布）Web 服务。开发人员可以在自己的程序中无缝地提供由 Web 服务器封装的功能。不过 Web 服务不是桌面程序的专利，使用 JavaScript 的网页也同样可以使用它们。

17.1 Web 服务快速入门

了解 Web 服务的含义是理解如何应用它们的关键。这一节将讲述 Web 服务的一些基础知识。

17.1.1 Web 服务是什么？

可以将 Web 服务器看成函数调用，只不过这个函数存在于某个服务器上，而调用在客户端上进行。这就要求要在客户端（也叫用户）和服务器端之间发送、接收消息。这些消息都采用 SOAP（Simple Object Access Protocol，简单对象访问协议）格式，这种格式是基于 XML 的对 Web 服务消息的包装器。SOAP 消息一般用标准的 HTTP 请求来进行传输（也可以使用其他的协议），同时加入了一些特殊的请求首部：

- **SOAPAction**——如果存在多种可能的动作，则给出指定的要进行的 SOAP 动作。如果只有一个动作可用，则这个首部通常为空。
- **Content-Type**——要设置为 `text/xml`。

SOAP 消息本身是包在信封中的，这个信封用于向服务器传输 Web 服务的调用，或从服务器传输 Web 服务调用的结果。典型的 SOAP 消息会看上去像这样：

```
<soap:Envelope xmlns:n="custom namespace goes here"
    xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

    <soap:Body soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
        <!-- method-specific XML goes here -->
```

```
</soap:Body>
</soap:Envelope>
```

客户端向服务器端发送消息时，`<soap:Body />`中的 XML 包含要调用的方法和参数。当服务器端发送消息时，`<soap:Body />`包含调用的结果。结果可以是单个值，也可以是有多个值的复杂数据类型。

客户端按照正确格式发送 SOAP 消息显然是很重要的，但是开发人员怎样才能知道如何格式化消息呢？这就涉及到了 WSDL。

17.1.2 WSDL

Web 服务描述语言（Web Service Description Language，简称 WSDL，读作 wizdel）用于描述 Web 服务的能力、格式和其他重要信息。WSDL 文件定义了 Web 服务提供的不同的操作。操作是指可以从给定服务中调用的特定的函数（一个 Web 服务可以有多个操作）。很容易就可以联想到 Web 服务是一个对象，并有一个或者多个方法；对象自身表示服务，而操作就是方法。

WSDL 文件的基本内容如下：

- **类型**——定义 Web 服务调用所使用的类型。这些类型基于 XML Schema 数据类型，可以表示简单类型——诸如数字和字符串，或者复杂类型，类似于包含特性的对象。
- **消息**——定义给定操作的输入参数和输出值。每个操作有两个消息定义，一个针对操作请求，另一个对应响应。
- **端口类型**——定义 Web 服务上可用的操作。
- **绑定**——定义用于每个操作的发送和接收的消息的格式。
- **服务**——定义访问 Web 服务操作的方式。

标准的 WSDL 文件有如下格式

```
<definitions name="name_of_service"
    targetNamespace="target_namespace"
    xmlns:tns="location_of_wsdl"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns="http://schemas.xmlsoap.org/wsdl/">
    <types>
        <!-- custom types defined here -->
    </types>
    <message name="request_name">
        <!-- parameters -->
    </message>
    <message name="response_name">
        <!-- return value(s) -->
    </message>
    <portType name="porttype_name">
        <operation name="method_name">
            <input message="tns:request_name" />
            <output message="tns:response_name" />
        </operation>
```

```

</portType>
<binding name="binding_name" type="tns:porttype_name">
    <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"
/>
    <operation name="method_name">
        <soap:operation soapAction="soap_action" />
        <input>
            <soap:body use="encoded" namespace="urn_string"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
        </input>
        <output>
            <soap:body use="encoded" namespace="urn_string"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
        </output>
    </operation>
</binding>
<service name="service_name">
    <documentation><!-- description of service --></documentation>
    <port name="port_name" binding="tns:binding_name">
        <soap:address location="webservice_url" />
    </port>
</service>
</definitions>

```

可以看到，WSDL 文件十分复杂，实际上，它们并不是给人看的。这些文件的预期应用是为程序（和它们的组件）提供调用 Web 服务的足够的信息。其实，完全可以利用当前大多数的 Web 服务工具包自动生成它，而无需编写 WSDL 文件。

然而，有时必须在 WSDL 文件中放入特定的信息才能让别人访问 Web 服务（根据开发的类型）。在 WSDL 文件中比较重要的一些信息如下：

- 要调用的方法的名称 (method_name);
- SOAP 动作 (soap_action);
- 要调用的方法的输入的目标命名空间 (urn_string);
- 端口名 (port_name);
- 服务的位置 (webservice_url)。

现在，最好看一下实际的 WSDL 文件来理解如何找到这些信息。

Temperature 服务是 XMethods (<http://www.xmethods.net>) 提供的一项 Web 服务。XMethods 是个公用的 Web 服务发行商，也是个公开可访问的 Web 服务的目录。这项服务接收五位的美国邮政编码，然后返回这个区域当前的温度。这个 Temperature 服务的 WSDL 文件如下：

```

<definitions name="TemperatureService"
    targetNamespace="http://www.xmethods.net/sd/TemperatureService.wsdl"
    xmlns:tns="http://www.xmethods.net/sd/TemperatureService.wsdl"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns="http://schemas.xmlsoap.org/wsdl/">

    <message name="getTempRequest">
        <part name="zipcode" type="xsd:string" />

```

```

</message>
<message name="getTempResponse">
    <part name="return" type="xsd:float" />
</message>
<portType name="TemperaturePortType">
    <operation name="getTemp">
        <input message="tns:getTempRequest" />
        <output message="tns:getTempResponse" />
    </operation>
</portType>
<binding name="TemperatureBinding" type="tns:TemperaturePortType">
    <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"
/>
    <operation name="getTemp">
        <soap:operation soapAction="" />
        <input>
            <soap:body use="encoded" namespace="urn:xmethods-Temperature"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
        </input>
        <output>
            <soap:body use="encoded" namespace="urn:xmethods-Temperature"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
        </output>
    </operation>
</binding>
<service name="TemperatureService">
    <documentation>Returns current temperature in a given U.S.
zipcode</documentation>
    <port name="TemperaturePort" binding="tns:TemperatureBinding">
        <soap:address
location="http://services.xmethods.net:80/soap/servlet/rpcrouter" />
    </port>
</service>
</definitions>

```

512

可从这个文件中获知的相关信息（突出标出的）如下：

- 有个唯一操作 getTemp。
- getTemp 对应的 SOAP 动作是 "" (空字符串)。
- getTemp 的输入的目标命名空间 urn:xmehtods-Temperature。
- 端口的名称 TemperaturePort。
- 服务的地址 http://services.xmethods.net:80/soap/servlet/rpcrouter。

根据访问 Web 服务的用户的类型，你需要了解其中一些或者全部的信息。

17.2 IE 中的 Web 服务

过去，微软的一些开发人员曾经开发了一个 HTML 组件（也称为行为），可以为使用 Web 服务的开发者隐藏起很多细节。HTML 组件其实是用 XML 和 JavaScript 定义的 COM 组件。HTML 组件包含属性、方法，支持自定义时间，这对创建默认浏览器没有的功能十分地理想。它的缺点

就是只有 IE 才支撑 HTML 组件，这样微软的 WebService 组件就不能在其他浏览器中使用了。WebService 组件可以从微软的网站上免费下载 (<http://msdn.microsoft.com/library/default.asp?url=/workshop/author/webservice/webservice.asp>)。

17.2.1 使用 WebService 组件

下载好 `webservice.htc` 文件后，将其放到 JavaScript 文件的目录中。然后，就可以通过将其应用到 HTML 元素上来访问它的功能，这使用了 `style` 特性和定制的 CSS 特性 `behavior`：

```
<div style="behavior:webservice.htc"></div>
```

然后，该 HTML 元素就可以从 WebService 组件中载入所有的属性、方法和事件了。如果将这个组件用于 JavaScript 对象，还要为元素分配一个 ID：

```
<div id="service" style="behavior:webservice.htc"></div>
```

然后再用 `document.getElementById()` 方法来获取一个引用：

```
var oService = document.getElementById("service");
```

下面，需要通过调用 `useService()` 来指定要使用的 Web 服务。`useService()` 方法接受两个参数：描述这个服务的 WSDL 文件和为服务所起的一个友好的名称。典型的调用如下：

513 `oService.useService(sUrl, "FriendlyName");`

调用这个方法时，该组件就下载 WSDL 文件并用其创建 JavaScript 对象和方法来访问 Web 服务。可以通过由在 `useService()` 方法中指定的友好名称所确定的对象来使用这些功能：

```
var oSpecificService = oService.FriendlyName;
```

这个对象有个可向服务器进行请求的方法——`callService()`。这个方法接受要调用的函数名和任意数量的参数。执行时，`callService()` 会返回一个调用 ID（从结果中获取值时将要用到它）。方法的调用如下：

```
iCallID = oService.FriendlyName.callService(sFuncName, sParam0, sParam1..sParamN);
```

然后进行异步的 Web 服务调用，这样就不会停止 JavaScript 的执行，然后等待服务器的响应。必须使用 `onresult` 事件处理函数来处理响应。可以直接在 HTML 中分配事件处理函数，也可以通过 JavaScript 来分配。如果使用 HTML，只需将 `onresult` 当成其他普通的事件处理函数：

```
<div id="service" style="behavior:webservice.htc" onresult="alert('Done') "></div>
```

要使用 JavaScript 分配事件处理函数，直接将函数赋值给 `onresult` 特性：

```
oService.onresult = function () {
    alert("Done");
};
```

`result` 事件被触发时，系统将创建包含 `result` 特性的 `event` 对象。这个特性包含一个对象，其中含有所有关于响应的细节内容。`result` 的特性列在下面的表格中：

特性名称	数据类型	描述
error	Boolean	如果在调用中出现了错误，则为 true
errorDetail	Object	包含错误信息的对象。主要用到的两个特性是 code（返回错误代码）和 string 特性（返回可理解的错误消息）
id	Number	由 callService() 创建的调用 ID
raw	String	从服务器发送的未经处理的 SOAP 代码
SOAPHeader	Array	用于调用的首部数组
value	Variant	由调用返回的值。可能是简单的数据类型，如数字或者字符串，也可能是对象

那么怎样使用这个对象呢？看下面这个简单的例子：

```
var iCallID = -1;

oService.onresult = function () {
    var oResult = window.event.result;

    if (oResult.id == iCallID) {
        if (oResult.error) {
            alert("An error occurred: " + oResult.errorDetail.string);
        } else {
            alert("Received back: " + oResult.value);
        }
    }
};

iCallID = oService.FriendlyName.callService(...);
```

514

在这段代码中，onresult 事件处理函数首先检查它处理的结果是否为相应的请求的响应（因为单个 WebService 可以处理多个请求）。如果结果的 ID 匹配调用的 ID，那么就处理结果。函数还要确保没有发生错误；如果发生了错误，则返回详细的错误信息；否则显示返回值。

当然，由你决定是直接使用 value 特性还是使用 raw 特性来获取所返回的 SOAPdaima。

17.2.2 WebService 组件例子

这个例子使用了本章前面所示的 WSDL 例子 Temperature 服务。只需为微软的 WebService 组件提供 WSDL 的位置要调用的操作的名称就能运行，并不一定是实际的文件。

这个例子中的页面由一个文本框（ID 为“txtZip”）和一个按钮（标题为“Get Temperature”）组成。用户在文本框中输入邮政编码，然后点击按钮获取该地区的温度（中间调用 Web 服务）。当然，你还需要一个可以用于分配 WebService 组件的元素。下面是 HTML 代码：

```
<html>
    <head>
        <title>IE Web Service Example</title>
        <script type="text/javascript">
            //...
        </script>
    </head>
```

```

<body>
    <p><input type="text" id="txtZip" size="10" /><input type="button"
    value="Get Temperature" onclick="callWebService()" />
        <div id="service" style="behavior:url(webService.htc)"
    onresult="onWebServiceResult()"></div>
    </body>
</html>

```

该页面运行的 JavaScript 也比较简单：

```

515
var iCallID = null;
var sWSDL = "http://www.xmethods.net/sd/2001/TemperatureService.wsdl";
function callWebService() {

    var sZip = document.getElementById("txtZip").value;
    var oService = document.getElementById("service");

    oService.useService(sWSDL, "Temperature");
    iCallID = oService.Temperature.callService("getTemp", sZip);
}

function onWebServiceResult() {
    var oResult = event.result;

    if (oResult.id == iCallID) {

        var oDiv = document.getElementById("divResult");

        if (oResult.error) {
            alert("An error occurred:" + oResult.errorDetail.string);
        } else {
            alert("It is currently " + oResult.value
                + " degrees in that zip code.");
        }
    }
}

```

第一个函数，callWebService()，先从文本框中获取邮政编码，然后调用 Web 服务。它首先载入 WSDL 文件，并加上友好名称 Temperature。下面一行使用 callService()方法，并传入操作名称和邮政编码。

另一个函数，onWebServiceResult()，用于显示调用的结果。这个函数使用了与前面例子中差不多的算法来检测结果 ID 是否等于调用 ID。然后它报告错误或者返回值。

Windows XP Service Pack 2 对 WebService 组件有一些重要的影响。首先，你必须指定以 .htc 结尾的文件名，同时要返回 MIME 类型为 `text/x-component`（这要在服务器端实现）。第二，WebService 组件被禁止连接到外部网站，也就是说，你只能访问和调用组件的页面在同一服务器上的 Web 服务。

17.3 Mozilla 中的 Web 服务

从 Mozilla 1.0 开始，Web 开发人员就已经可以访问 Web 服务了。Mozilla 目前支持两种处理 Web 服务的方法：使用低层 SOAP API 或者使用 WSDL 代理对象（在 Mozilla 1.4 中引入的）。在开始探索基于 WSDL 的功能之前，先理解基本的 SOAP API 是很重要的。

17.3.1 加强的特权

出于安全考虑，在 Mozilla 中使用 JavaScript 访问不同的服务器是明确禁止的。由于 Web 服务要求能访问不同服务器，这种安全错误造成了很多麻烦。不过，还是有办法让用户许可脚本并允许它进行跨域的访问。默认情况下，Mozilla 的安装设置不允许这种高级特权（还是由于安全问题）。但是，可以在 `all.js` 配置文件中覆盖这个设置（在 Windows 系统中位于 `Program Files\Mozilla\defaults\pref`）。

在文本编辑器中打开 `all.js`，找到以下一行：

```
pref("signed.applets.codebase_principal_support", false);
```

改成：

```
pref("signed.applets.codebase_principal_support", true);
```

更改这个设置后，必须关闭所有 Mozilla 窗口并重启浏览器。这只是第一步。第二步是请求通用浏览器读取（Universal Browser Read）特权，它允许跨域的通信。如下：

```
try {
    netscape.security.PrivilegeManager.enablePrivilege("UniversalBrowserRead");
} catch (e) {
    alert("Script not signed.");
}
```

执行到第二行代码时，会出现一个对话框说明脚本需要加强的特权。然后用户可以点击 Yes 来允许这个特权，或者点击 No 来禁止它（图 17-1）。

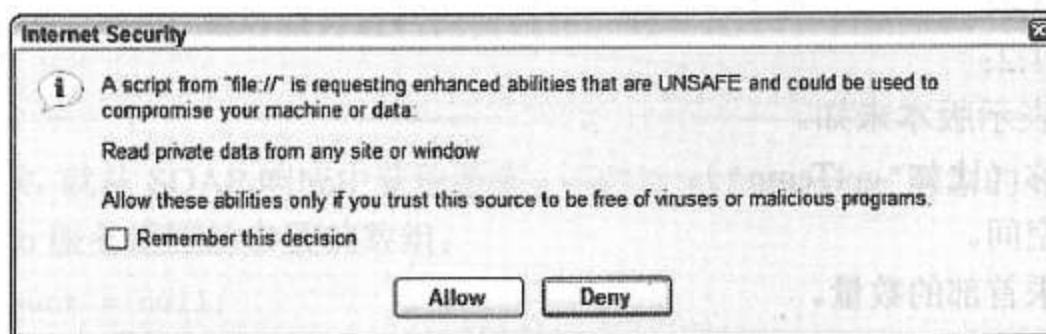


图 17-1

如果用户点击 Yes，代码便可继续运行；如果用户点击 No，会出现错误，同时在前面的例子中，会出现一个警告框说明脚本请求的特权已被否决。

注意，不能假设用户已经将浏览器设置为允许通用浏览器读取特权。如果打算使用这种 Web 服务的调用方式，必须创建一个已签署的脚本（在第 19 章中有更详细的介绍）。还要注意这是特定于 Mozilla 的代码，不能在 IE 中运行，所以在请求脚本签署前进行浏览器的检测。

17.3.2 使用 SOAP 方法

Mozilla 提供了很多 SOAP 功能，但是因为这是比较老的 API，所以我们只讨论最常用的。

Mozilla 的 SOAP 功能的基础是 SOAPCall 对象。这个对象包含和表示了整个 Web 服务的请求和响应。SOAPCall 对象的创建与其他对象一样：

```
var oSoapCall = new SOAPCall();
```

下一步是设置 Web 服务的位置，将位置赋值给 SOAPCall 对象的 transportURI 特性：

```
oSoapCall.transportURI = "http://address_of_service";
```

这个地址并非 WSDL 的地址，它其实是 Web 服务自身实际的 URL（与 WSDL 元素<soap:address location="http://address_of_service" />中定义的一样）。

设置了服务的位置后，可以创建一个用于存放 SOAP 的参数的数组。数组是个普通数组，其中存放了 SOAPPARAMETER 对象。SOAPPARAMETER 构造函数有两个参数（依次是）：参数的值和名称。数组可包含任意多个参数，如下创建：

```
var arrParams = new Array;
arrParams[0] = new SOAPPARAMETER("value", "name");
```

下面，对数组调用 SOAPCall 对象的 encode() 方法，它对参数进行编码以便于传输。对于这个方法，需要知道操作的名称和目标命名空间：

```
oSOAPCall.encode(0, "operation_name", "target_namespace", 0, null,
arrParams.length, arrParams);
```

其中参数是：

- 使用的 SOAP 的版本：
 - 0 表示 1.1；
 - 1 表示 1.2；
 - 65535 表示版本未知。
- 操作的名称（比如 "getTemp"）。
- 目标命名空间。
- 提供给请求首部的数量。
- SOAPHeaderBlock 对象的数组（对于大部分 Web 服务不是必需的）。
- 传递的参数的数量。
- SOAPPARAMETER 对象的数组。

最后，用 asyncInvoke() 方法向服务器发出调用。这个方法只有一个参数，这个参数是 Web 服务返回一个值时所调用的方法。如果你有个 onWebServiceResult 函数，则代码如下：

```
oSoapCall.asyncInvoke(onWebServiceResult);
```

回调函数本身格式如下：

```
function onWebServiceResult(oResponse, oSoapCall, iError) {
    //...
}
```

回调函数接受三个参数：SOAPResponse 对象，SOAPCall 对象和错误代码。SOAPResponse 对象包含成功请求后响应的所有信息。它的特性如下：

特 性	描 述
actionURI	SOAPAction 首部的字符串（可以是空字符串）
body	SOAP 响应中的<Body/>元素
encoding	用于表示响应的编码的 SOAPEncoding 对象
envelope	SOAP 响应中的<Envelope/>元素
fault	如果发生了错误，则是个 SOAPFault 对象；否则为空
header	SOAP 响应中的<Header/>元素，或者如果不存在则为 null
message	SOAP 响应所对应的 DOM 文档
methodName	调用的方法的名称。大部分情况下，是由 body 特性所表示的元素的标签名（"Body"）
targetObjectURI	响应的目标命名空间
version	SOAP 版本号（如果能从信封中获取的话），可能是以下常量中的一个：0 表示 1.1 版，1 表示 1.2 版，65535 表示版本未知

回调函数的第二个参数是用于进行请求的 SOAPCall 的实例。第三个参数是错误代码，用于表示在客户端到服务器端的通信中是否有问题出现；如果没有错误，它等于 0。如果 Web 服务自身造成了错误，将会创建一个 SOAPFault 对象并存储在 SOAPResponse 的 fault 特性中。

519

如果出现了通信错误，错误代码将是非 0 的数值。如果 Web 服务本身出错了，oResponse.fault 对象则不为 null。这个对象有 faultString 和 faultCode 两个特性，它们提供了关于错误的额外细节。

也可以用 invoke() 方法对服务进行同步调用，它将把一个 SOAPResponse 对象作为函数值返回。

如果没有错误，就从 SOAP 响应中获取数据。可用 SOAPResponse 对象的 getParameters() 方法来返回由 Web 服务返回的参数的数组：

```
var oParamCount = null;
var arrParams = oResponse.getParameters(false, oParamCount);
```

getParameters() 方法的第一个参数表示响应是否是 RPC 格式的。非 RPC 格式为 false，RPC 格式为 true（虽然大部分情况下使用 false 即可）。第二个参数只存放返回参数的数量的对象。大部分情况下，参数数量不是必需的，因为参数的数量可通过返回的数组的 length 特性来获取，所以典型的调用方式如下：

```
var arrParams = oResponse.getParameters(false, {});
```

注意，在这种情况下，第二个参数是没有引用的对象字面量。第二个参数即使没有用也必须包括在内，这样才可接受。

每个参数都包含一个 element 特性，可以用它来访问 SOAP 响应中对应的元素。这是一个 XML 元素，所以它有任何 XML 元素所具备的方法和特性。要获取参数的字符串值，一般像这样做：

```
var sValue = arrParams[0].element.firstChild.nodeValue;
```

当然，此后如何使用这些值就由你决定了。

要用 Mozilla 的 SOAP 对象来调用 Temperature 服务，首先需要一些信息：Web 服务的 URL 和目标命名空间。记住这个信息可以在 WSDL 文件中找到：

```
var sURL = "http://services.xmethods.net:80/soap/servlet/rpcrouter";
var sTargetNamespace = "urn:xmethods-Temperature";
```

除了没有额外的用于载入 WebService 组件的<div>元素外，这个例子中的 HTML 与 IE 例子
520 中的一样。

第一个函数仍称为 callWebService()。第一步必须是对加强特权的请求，如前面所述。如果没有给予特权，则函数就无需继续，直接返回。如果启用了特权，就获取邮政编码并将其放入 SOAPPARAMETER 数组中。最后，创建 SOAP 调用，并进行异步请求：

```
function callWebService() {
    try {
        netscape.security.PrivilegeManager.enablePrivilege("UniversalBrowserRead");
    } catch (e) {
        alert(e);
        return false;
    }

    var sZip = document.getElementById("txtZip").value;

    var arrParams = new Array;
    arrParams[0] = new SOAPPARAMETER(sZip, "zipcode");

    var oSoapCall = new SOAPCall();
    oSoapCall.transportURI = sURL;
    oSoapCall.encode(0, "getTemp", sTargetNamespace, 0, null,
                    arrParams.length, arrParams);
    oSoapCall.asyncInvoke(onWebServiceResult);
}
```

下面，通过 onWebServiceResult() 函数来处理响应。函数的大部分都着力于判断在 Web 服务调用过程中是否出现了错误：

```
function onWebServiceResult(oResponse, oSoapCall, iError) {
    if (iError != 0) {
        alert("An unspecified error occurred.");
```

```

    } else if (oResponse.fault != null) {
        alert("An error occurred (code=" + oResponse.fault.faultCode
              + ", string=" + oResponse.fault.faultString + ")");
    } else {
        var oParams = oResponse.getParameters(false, {});
        alert("It is currently " + oParams[0].element.firstChild.nodeValue
              + " degrees in that zip code.");
    }
}

```

该函数的第一步是检测错误代码 `iError`。如果它不等于 0，表示没有得出正确的响应，所以显示错误信息。如果错误代码是 0，下一步就判断有没有 SOAP 错误。确保 `fault` 特性是 `null`（如果不是，则显示详细的错误信息）。

这两个条件都不成立时，则说明响应是有效的。然后，显示第一个参数的值。

521

17.3.3 使用 WSDL 代理

使用 Mozilla 的 SOAP 功能需要一段过程，并且要进行一些研究（查看 WSDL 文件）才能正确使用。另外，为跨域通信签署脚本并非开发人员真正想处理的问题。作为一个有远见的组织，Mozilla 意识到了这一点，并给出了另一种方案：WSDL 代理。

WSDL 代理的基本概念是：只要给出了有效的 SOAP 请求，就允许其通过。在 Web 服务中，只有服务器在任何用户的范围内都是易受攻击的，服务器才是易受攻击的。因此，任何服务器都只会发布对其不会造成危害的功能。这样，如果 Web 服务的请求可以被标示为这类请求，则应该允许它和其他服务器进行通信。这最终使得在 Mozilla 1.4 中引入了 WSDL 代理。

WSDL 代理本质上是表示 Web 服务的对象，它将操作作为代理的方法。你可以指定方法进行同步或者异步调用，但是同一个代理的所有方法必须使用同一种同步性（要么都是同步的，要么都是异步的——不可同时存在）。

不幸的是，使用 WSDL 代理并不像使用底层 SOAP 功能和微软的 `WebService` 组件那样来得直观。

首先，创建 `WebServiceProxyFactory`：

```
var oFactory = new WebServiceProxyFactory();
```

工厂类的唯一实现了的方法（在文档中还有一些其他的方法）是 `createProxyAsync()`，可以创建一个异步调用的代理，以免与正常的 JavaScript 处理相冲突。这个方法接受以下参数：

- WSDL 文件的位置；
- 端口名称；
- 限定词，总为空字符串（这个参数只有在对象用于 Mozilla 的 C++ 层上时才会用到）；
- 表示被创建的代理的方法是否应该进行异步调用的布尔值；
- 回调对象，在代理的生存期内发生特定的事件时通知它相应的方法。

最后一个参数——回调对象，必须有两个方法：`onLoad()`，代理完全载入时调用；`onError()`，在代理的创建期内发生错误时调用。每个方法都接受一个参数。`onLoad()` 方法接受的是被

创建的代理对象；`onError()`方法接受的是错误信息。

回调对象一般使用对象直接表示法，如下：

```
var oCallbackObject = {
    onLoad: function (oCreatedProxy) {
        //...
    },
    onError: function (sMessage) {
        //...
    }
}
```

按照下面的方法为`createProxyAsync()`方法传递这个对象：

```
oFactory.createProxyAsync("wsdl_location", "port_name", "", true, oCallbackObject);
```

调用`createProxyAsync()`方法时，应该总是加上`try...catch`语句，那样如果方法出于某种原因未能执行完毕，它会抛出错误。

调用这个方法后，就会启动一个新的线程来载入 WSDL 代理。载入完毕时，会调用`onLoad()`方法，并传入新建的代理。如果只进行单个 Web 服务的调用，可以直接使用这个方法来调用特定的操作。否则，就要将代理的引用保存起来以便以后使用，如下：

```
var oProxy = null;

var oCallbackObject = {
    onLoad: function (oCreatedProxy) {
        oProxy = oCreatedProxy;
    },
    onError: function (sMessage) {
        alert(sMessage);
    }
}
```

在这个例子中，创建了一个用于保存新建的代理的全局变量`oProxy`。在`onLoad()`方法中，新建的代理被分配给`oProxy`，这样就可以在其他函数中访问它了。

尽管可以指定代理操作是同步或是异步调用，但实际上同步调用并不能总是正常工作。所以最好使用异步调用。

异步调用一个操作还需要另外一个回调对象。这个对象必须为每个要调用的操作准备一个回调方法。回调方法的名字总是通过在操作的名称后加上`Callback`来定义（例如，`getTemp`的回调方法的名称为`getTempCallback`）。这个方法把从 Web 服务调用返回的响应数据作为参数来接受。为了更好地理解这一点，我们来看一个例子。

这个例子还是使用 Temperature 服务。要了解的信息是 WSDL 文件的位置和端口名称。下面是一些全局变量：

```
var oProxy = null;
var sWSDL = "http://www.xmethods.net/sd/2001/TemperatureService.wsdl";
var sPort = "TemperaturePort";
```

下面，为代理的创建准备一个回调对象。它的任务就是将新建的代理存储在 oProxy 变量中，并使用 setListener 方法为操作分配一个回调对象：

```
var oProxyCreateCallback = {
    onLoad: function(oCreatedProxy) {
        oProxy = oCreatedProxy;
        oProxy.setListener(oGetTempCallback);
    },
    onError: function(sError) {
        alert(sError);
    }
};
```

操作的回调对象称为 oGetTempCallback，如下：

```
var oGetTempCallback = {
    getTempCallback: function (iDegrees) {
        alert("It is currently " + iDegrees + " degrees in that zip code.");
    }
};
```

可以看到，唯一的方法是 getTempCallback()。因为 Web 服务只是返回一个数字，第一个（也是唯一的）参数中包含一个简单的数字。参数名称为 iDegrees，但是，实际上，如何对参数进行命名是无所谓的。一旦 getTemp 返回一个值便调用这个方法，显示温度。

下面，需要用一个函数来创建 WSDL 代理。在很多情况下，最好在页面的 onload 事件处理函数中创建代理，这是由代理创建的异步性决定的，所以在这个例子中直接将函数分配给了 window.onload：

```
function createProxy() {
    try {
        var oFactory = new WebServiceProxyFactory();
        oFactory.createProxyAsync(sWSDL, sPort, "", true, oProxyCreateCallback);
    } catch (oError) {
        alert(oError.message);
    }
}

window.onload = createProxy;
```

该函数创建了 WebServiceProxyFactory 对象，并用它创建了代理。指定 WSDL 文件的位置、端口号、回调对象以及按照异步方式调用。

最后是 callWebService() 函数。这个函数在尝试进行调用前，首先检查代理是否已创建好。如果创建了，则调用代理的 getTemp() 方法，并传入邮政编码：

```
function callWebService() {
    if (oProxy) {
        var sZip = document.getElementById("txtZip").value;
        oProxy.getTemp(sZip);
```

```

} else {
    alert("Proxy not available.");
}

}

```

当在这个函数中调用 `getTemp()` 时，会启动另一个线程来进行 Web 服务的调用。结果会传到 `oGetTempCallback` 对象中。这个函数应该在用户点击 Get Temperature 按钮时调用（本例子使用的 HTML 页面与前面的例子中使用的一样）。

这段代码执行的方式并非完全线性的（见图 17-2），不过，还是能完成任务。

```

② var oProxyCreateCallback = {
    onLoad: function(...) { ←
        ...
        ...
    }
}

① function createProxy() {
    ...
}

④ var oGetTempCallback = {
    getTempCallback: function() { ←
        ...
    };
}

③ function callWebService() {
    ...
}

```

图 17-2

17.4 跨浏览器的方案

再有浏览器访问 Web 服务的区别如此之大，以致无法使用当前的浏览器功能来创建跨浏览器的方法。但是 Web 服务不过是特殊格式的标准 HTTP 请求，我们依然可以利用 XML HTTP 请求对象来创建跨浏览器的方案。这样，还要用到浏览器检测脚本以及创建 IE 的 `XMLHttpRequest` 构造函数（本书前面介绍过）。

17.4.1 WebService 对象

每个 Web 服务都是唯一的，所以如何才能开发出一种标准的使用途径呢？答案是创建普通的模板对象，且其定义可以被覆盖并使其变得更精确。这个模板对象为 `WebService`，定义如下：

```

function WebService() {
    this.action = "";
    this.url = "";
}

```

```

WebService.prototype.buildRequest = function () {
    return "";
};

WebService.prototype.handleResponse = function (sSOAP) {
};

WebService.prototype.send = function () {

    if (isMoz) {
        try {
            netscape.security.PrivilegeManager.enablePrivilege("UniversalBrowserRead");
        } catch (oError) {
            alert(oError);
            return false;
        }
    }

    var oRequest = new XMLHttpRequest;
    oRequest.open("post", this.url, false);
    oRequest.setRequestHeader("Content-Type", "text/xml");
    oRequest.setRequestHeader("SOAPAction", this.action);
    oRequest.send(this.buildRequest());

    if (oRequest.status == 200) {
        return this.handleResponse(oRequest.responseText);
    } else {
        throw new Error("Request did not complete, code " + oRequest.status);
    }
};

```

WebService 对象是跨浏览器 Web 服务支持的基础。它有两个特性：SOAP 请求的 URL(url) 和 SOAP 动作 (action)。两个特性在初始化时都被设为 null；子类按照需要填入这两个值。buildRequest() 方法用于构建 SOAP 消息字符串；但在这种情况下，它仅返回一个空字符串。handleRequest() 方法从响应中接受 SOAP 消息并返回合适的值。

send() 方法担任功能中最主要的部分。首先，如果浏览器是 Mozilla，必须请求通用浏览器读取特权。然后，创建 XMLHttpRequest 对象（注意，这个代码为 IE 使用了包装器），并构建 SOAP 请求。这个请求包括打开到指定 URL 的请求，将内容类型设置为 "text/xml"，并设置 SOAP 动作。然后，调用 buildRequest() 方法来获取 SOAP 消息，并用 XMLHttpRequest 的 send() 方法将其发送出去。

最后，如果请求返回状态 200，返回的文本就由 handleResponse() 进行解释并返回值。否则，抛出错误。最终的目的是可以这样使用 WebService 对象：

```

var oService = new WebService();
var vResult = oService.send();
alert("Service returned " + vResult);

```

当然，还没有足够的信息使得这个对象可以只使用自己。不过，它可以用来创建其他 WebService 的包装器。

17.4.2 Temperature 服务

这次，我们还是再以 Temperature 服务为例。首先，TemperatureService 对象继承 WebService 并定义 url 和 action 特性：

```
function TemperatureService() {
    WebService.apply(this);
    this.url = "http://services.xmethods.net:80/soap/servlet/rpcrouter";
    this.zipcode = "";
}

TemperatureService.prototype = new WebService();
```

记住 Temperature 服务的 SOAP 动作其实是个空字符串，所以默认值（从 WebService 中继承）可以直接用。新的特性 zipcode 用于保存要检查的邮政编码。

对于 buildRequest() 方法，首先确定 SOAP 请求的格式。这对于任意 Web 服务都是可以使用 WSDL 分析工具来轻易完成的，你可以在 XMethods 上下载到这个工具 (<http://www.xmethods.net/ve2/Tools.po>)。这些方法可以用 WSDL 文件看到任意 Web 服务的请求和响应的格式。

Temperature 服务的 SOAP 请求如下：

```
<soap:Envelope xmlns:n="urn:xmethods-Temperature"
    xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <soap:Body soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
        <n:getTemp>
            <zipcode xsi:type="xs:string"><!-- zipcode here --></zipcode>
        </n:getTemp>
    </soap:Body>
</soap:Envelope>
```

全部的信息都包含在这个简单的 SOAP 请求中，其中最重要的一行（突出标出的）就是输入邮政编码的地方。buildRequest() 方法必须在创建 SOAP 字符串时插入邮政编码：

```
TemperatureService.prototype.buildRequest = function () {
    var oBuffer = new StringBuffer();

    oBuffer.append("<soap:Envelope xmlns:n=\"urn:xmethods-Temperature\" ");
    oBuffer.append("xmlns:soap=\"http://schemas.xmlsoap.org/soap/envelope/\" ");
    oBuffer.append("xmlns:soapenc=\"http://schemas.xmlsoap.org/soap/encoding/\" ");
    oBuffer.append("xmlns:xs=\"http://www.w3.org/2001/XMLSchema\" ");
    oBuffer.append("xmlns:xsi=\"http://www.w3.org/2001/XMLSchema-instance\">");
    oBuffer.append("<soap:Body soap:encodingStyle=\"");
    oBuffer.append("\\"http://schemas.xmlsoap.org/soap/encoding/\\>\"");
    oBuffer.append("<n:getTemp><zipcode xsi:type=\"xs:string\\>\"");
    oBuffer.append(this.zipcode);
    oBuffer.append("</zipcode></n:getTemp></soap:Body></soap:Envelope>");

    return oBuffer.toString();
};
```

因为 SOAP 请求太长了，所以这个方法用本书前面创建的 `StringBuffer()` 对象来构造整个字符串。注意，这里用 `zipcode` 特性来插入邮政编码，而没有将其作为参数传入。如果将邮政编码作为参数传入，必须重写 `send()` 方法；而这种方法就可以直接使用原来的 `send()` 方法。

`handleResponse()` 方法期望接受一个 SOAP 响应字符串作为唯一的参数。格式依然可以用 XMethods 的 WSDL 分析工具来进行判断。

```
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">

    <SOAP-ENV:Body>
        <ns1:getTempResponse
            SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
            xmlns:ns1="urn:xmethods-Temperature">
            <return xsi:type="xsd:float"><!--returned value--></return>
        </ns1:getTempResponse>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

其他的代码都可以不管，唯一需要注意的是包含在 `<return>` 元素中的内容。从字符串中抽取数据的最方便的方法是使用正则表达式；在这个案例中，不需要使用昂贵的 DOM 解析和操作（不过，对于更加复杂的 Web 服务，也许可以考虑）。获取了值之后，可以用 `parseFloat()` 函数获取浮点数值。

```
TemperatureService.prototype.handleResponse = function (sResponse) {
    var oRE = /<return .*?>(.*)<\/return>/gi;
    oRE.test(sResponse);
    return parseFloat(RegExp["$1"]);
};
```

剩下的事情就是，修改 `send()` 方法以使它接受一个参数（邮政编码），因为必须设置 `zipcode` 特性才能令这个方法正确运行（注意，`send()` 方法会调用 `buildRequest()`，而这个方法就是用 `zipcode` 特性来创建 SOAP 字符串的）。

首先创建指向原来的 `send()` 方法的一个引用：

```
TemperatureService.prototype.webServiceSend = TemperatureService.prototype.send;
```

528

这一行代码创建的 `webServiceSend` 特性指向原来的 `send()` 函数。现在就可以在不丢失原来的功能的基础上重新定义 `send()` 了：

```
TemperatureService.prototype.send = function (sZipcode) {
    this.zipcode = sZipcode;
    return this.webServiceSend();
};
```

新的 `send()` 方法所做的第一件事情是分配 `zipcode` 特性。然后，它返回 `webServiceSend()` 方法的结果（是调用原来的 `send()` 方法的结果）。`TemperatureService` 对象可按如下方式使用：

```
var oService = new TemperatureService();
var fResult = oService.send("90210");
alert("Temperature is " + fResult + "degrees");
```

17.4.3 使用 TemperatureService 对象

再重建温度的例子，代码就更加简单了：

```
<html>
  <head>
    <title>Cross-Browser Web Service Example</title>
    <script type="text/javascript" src="detect.js"></script>
    <script type="text/javascript" src="stringbuffer.js"></script>
    <script type="text/javascript" src="http.js"></script>
    <script type="text/javascript" src="webservice.js"></script>
    <script>
      function callWebService() {
        var sZip = document.getElementById("txtZip").value;
        var oService = new TemperatureService();
        var fResult = oService.send(sZip);

        alert("It is currently " + fResult + " degrees in that zip code.");
      }
    </script>
  </head>
  <body>
    <p><input type="text" id="txtZip" size="10" />
    <input type="button" value="Get Temperature" onclick="callWebService()" />
    </p>
  </body>
</html>
```

可以看到，callWebService() 函数在这个代码中已经被极大地简化了。从文本框中获取邮政编码，然后创建 TemperatureService 对象，并将邮政编码传给 send() 方法，最后返回温度值。然后出现一个警告框显示温度。

529

17.5 小结

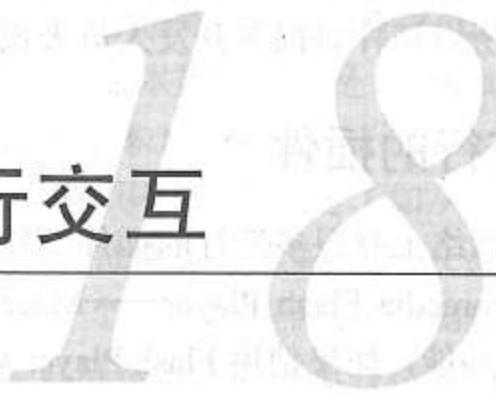
Web 服务是一种新技术，而且，对于 JavaScript 来说更是如此。在这一章中，学习了两个浏览器，IE 和 Mozilla 是如何利用 JavaScript 访问 Web 服务的功能的。

还学习了微软如何创建一个 HTML 组件来封装 Web 服务调用。这个 WebService 组件从 WSDL 文件中读取 Web 服务的信息，并开发一个友好的对象来处理请求和响应。

还学习了 Mozilla 是如何开发它的 Web 服务功能的。它提供了低层的 SOAP API 以及 WSDL 代理。跨域安全性的问题以及高级特权因为与 Web 服务相关，也对其进行了讨论。

最后，还向你介绍了一个跨浏览器的 Web 服务的解决方案，尽管还不够细致，但已经可以为 IE 和 Mozilla 提供一个统一的使用接口了。

530



如今，Web 上已不仅仅只有 HTML 和图像。世界上的网站还利用了各种各样的插件。插件能够在不妨碍底层的 HTML 的情况下，为 Web 浏览器页面嵌入 applet（有时候也可能比较大）或者对象。浏览器遇到<object/>元素中的内容时，它就将渲染的责任交给相关的插件。

最初，唯一可选择的插件是 Java applet。通过 applet Web 开发人员可以直接在 Web 页面中加入 Java 功能。从那之后，又不断地开发出许多新的插件。本章关注于今天最流行的插件，以及如何使用 JavaScript 与它们进行交互。

18.1 为何使用插件

最开始，网页上基本都是相对静态的东西。在 DOM 出现之前，页面载入后，它的外观在页面卸载前都不会改变。传统的开发人员都是与动态的界面打交道的，用户在屏幕上做的任何事情都会引起变动，但是转到 Web 后，发现这个环境极其缺乏动态性。然后就有了 Java。

Java 原本并非为在 Web 浏览器中使用而设计，因为它诞生的时候，万维网还仅仅只是一个概念。然而，随着因特网和 Web 的流行，原来的 Java 开发人员看到可以使用 Java 来丰富 Web 浏览的体验。于是，称为 HotJava 的试验性浏览器诞生了，它是由 Sun Microsystems 为证明 Java applet 是如何嵌入到网页中而开发的。网页自此就再也不是静态的了，而是无需页面重载就可以实现用户交互，这对 Web 而言将是个光明的未来。

同一时期，Netscape 着手开发一种针对助手应用的架构。基本思想是让浏览器能够辨认信息的 mime 类型，然后调用相应的应用来查看内容。Netscape 2.0 出现时，它还有一个特别的插件架构，本质上是提供了直接将助手应用嵌入到网页中的能力。

从此，浏览器又进入了一个漫长的旅程。现在，所有的浏览器都可以在页面中嵌入 Java applet，以及其他插件的宿主。当然，现在用 DOM 也可以创建动态页面，所以还有使用插件的理由吗？答案是肯定。

插件继续推动着网页交互的发展。虽然很多浏览器都内置 JavaScript 的 HTTP 请求功能，但是真正能使用套接字实现双向客户端-服务器端通信的还是插件。类似地，唯一能嵌入并动画演示矢量图形的方法使用的是特性的插件。作为额外的部分，很多插件可以在不同浏览器中正常运行。

插件可以提供浏览器现在或将来不能直接提供的功能。因为公司不能等浏览器更新功能后才更新插件，所以插件对很多开发人员来说仍然很有吸引力。

18.2 流行的插件

今天，网络上有很多流行的插件。最值得注意的是下面这些：

- Macromedia Flash Player——Macromedia Flash 提供了基于向量图形的动画和不断增加的复杂功能，如果使用 Flash Player 就可以将其嵌到页面中。毫无争议地，Flash Player 是最流行的插件。一些网站也完全是用 Flash 来写的。现在几乎所有的浏览器都搭载了 Flash Player，它在 Windows、MacOS、Linux 和大部分 Unix 系统上都是可用的。
- Java 插件——仍然是插件技术的领导者，Java 插件可以使你在页面中嵌入 Java Applet。它几乎可以用在所有的平台上。
- Applet Quicktime——用于在页面中插入 Apple Quicktime 视频。Quicktime 视频既可以是标准的视频，也可以是 Quicktime VR (Virtual Reality, 虚拟现实) 动画，使得用户可以在 3D 房间内进行观察。Quicktime 可以在 Windows 或者 MacOS 上使用。
- Real Player——Real Player 是早期通过因特网提供流媒体音乐和视频的先驱。今天，它正与 Quicktime 及 Windows Media Player 在流媒体的控制权方面进行激烈的竞争（见下面）。它做得很不错——对几乎所有的计算平台都有支持。
- Windows Media Player——为了不被打败，微软自己提供了可嵌入的视频播放器 Windows Media Player。虽然它可以处理 Quicktime 动画和自己的格式，但是 Windows Meida Player 只能在 Windows 系统上使用。
- Adobe SVG Viewer——可伸缩矢量图形 (Scalabe Vector Graphics, SVG) 是一种新的基于 XML 的语言，用于创建矢量图形。虽然浏览器仍然不能直接为这种语言提供支持，但是 Adobe 已经拉开了这场战斗的序幕，引入了 SVG Viewer，它可用于在页面中嵌入 SVG 图像。SVG Viewer 目前在大部分平台上都有可用的版本，并支持 IE 4.0 和 Netscape 4.x；目前还没有对 Mozilla 的支持。
- ActiveX 对象——IE 中到处都是 ActiveX 对象。你可以在 JavaScript 中使用它们，并可以将其嵌入到网页中。虽然技术上它并非插件，但因为在 IE 中存在对 ActiveX 的支持，在网页中嵌入 ActiveX 对象的方法与嵌入插件一样。
- Macromedia Shockwave——在 Flash 出现之前就有了 Shockwave。Shockwave 播放用 Macromedia Director 创建的文件。Director 是一种多媒体编程环境，可以让开发人员创建交互性演示动画、游戏和其他高级内容。虽然不如 Flash 出名。Shockwave 在 Windows 和 MacOS 上都有可用的版本。

18.3 MIME 类型

谈论插件时，MIME 类型的概念十分重要。MIME 代表 Multipurpose Internet Mail Extension

(多功能因特网邮件扩展), 原来是为判断电子邮件附件的格式而设计的一段字符串。从这个不起眼的起点出发, MIME 类型(也称之为内容类型)演变成了因特网上判断文件格式的事实标准。

每个 MIME 类型都由一个媒体类型和一个子类型组成。媒体类型可以是 application(应用程序)、image(图形)、audio(声音)、video(视频)、text(文本)、message(消息)、model(模型)和 multipart(多部分)。子类型通常是一个更加确切的标识符, 它指定了如从压缩格式到文件扩展名之类的东西。媒体类型和子类型用一个斜杠(/)分隔开来, 比如 text/css, 它会告诉浏览器(或者其他因特网应用)文件是纯文本文件, 也是一个 CSS 样式表。浏览器插件工作过程是, 通过将它们自己映射到特定的 MIME 类型, 告诉浏览器当处理这种特定类型时, 应该通过插件进行处理。

一般来说, 一些 MIME 类型是通过浏览器本身来处理的, 比如 text/javascript(JavaScript 文件)、text/css(CSS 样式表)、image/gif(GIF 图像)、image/jpeg(JPEG 格式图像)、text/xml(XML 文件) 和 text/html(HTML 文件)。但是, 还有很多其他类型都是由插件处理的。

18.4 嵌入插件

HTML 用<object>元素来将插件嵌入到页面中。<object>至少要求四个特性:

- type——嵌入的文件或对象的 MIME 类型;
- data——载入到对象中的文件的 URL;
- width——对象在页面上占据的水平空间;
- height——对象在页面上占据的垂直空间。

浏览器内部将文件的 MIME 类型与特定插件对应起来, 所以设置了 type 特性就可以告诉浏览器要载入哪个插件。例如, 如果 Flash 文件的内容类型是 application/x-shockwave-flash, 则下面的代码就可以嵌入一段 Flash 动画:

```
<object type="application/x-shockwave-flash" data="myflashmovie.swf"
        width="100" height="100"></object>
```

533

如果没有为这个 MIME 类型注册任何插件, 浏览器(根据使用的浏览器)可能会提示要为文件安装正确的插件。很多浏览器都会查看 pluginspage 特性, 这是<object>的一个非正式的特性(没有在规范中指定)。如果用户机器上没有安装嵌入的对象的插件, 这个特性就会指定到哪里去寻找。例如:

```
<object type="application/x-shockwave-flash" data="myflashmovie.swf"
pluginspage="http://www.macromedia.com/shockwave/download/download.cgi?P1_Prod_Version=ShockwaveFlash"
        width="100" height="100"></object>
```

18.4.1 加入参数

有时, 某个对象要有额外的参数才能正确运行。使用带有 name 特性以及 value 特性的<param>元素来指定某个嵌入的对象的参数:

```
<object type="application/x-shockwave-flash" data="myflashmovie.swf"
        width="100" height="100">
    <param name="message" value="Hello World!" />
</object>
```

每一个对象都可以有任意个参数。包含不必要的参数并不会产生任何负面的影响。

18.4.2 Netscape 4.x

老的 Netscape 4.x 浏览器并不支持`<object>`元素，所以要使用 Netscape 特有的`<embed>`元素。`<embed>`元素接受与`<object>`的特性类似的属性（事实上，`<object>`就是根据`<embed>`设计的），除了使用`src`特性来替代`data`外：

```
<embed type="application/x-shockwave-flash" src="myflashmovie.swf"
       width="100" height="100">
    <param name="message" value="Hello World!" />
</embed>
```

可以看到，这个方式与使用`<object>`极为相似。问题是新的浏览器并不支持这个元素（官方已经不推荐使用）。所以如果你打算支持 Netscape 4.x，最好的解决方法是使用`<object>`，并在里面添加一个`<embed>`元素，如下：

```
<object type="application/x-shockwave-flash" data="myflashmovie.swf"
        width="100" height="100">
    <param name="message" value="Hello World!" />
    <embed type="application/x-shockwave-flash" src="myflashmovie.swf"
           width="100" height="100">
        <param name="message" value="Hello World!" />
    </embed>
</object>
```

使用这个方法时，新的浏览器会忽略`<embed>`元素并使用`<object>`来嵌入文件；而 Netscape 4.x 就会忽略`<object>`并使用`<embed>`来嵌入文件。

18.5 检测插件

与大部分 Web 技术一样，有两种可用的插件：微软的解决方案和其他公司的。微软的方法是使用 ActiveX 技术，你应该记得这是创建与 XML 相关的对象所使用的方法；另一种方法为 Netscape 式插件，因为是 Netscape Navigator 将插件引入到 Web 中的。直到最近，IE 才支持 Netscape 式插件。但是，从 IE 5.5 Service Pack 2（只有 Windows 版）开始，微软又取消了对 Netscape 式插件的支持。

两种插件主要的区别在于架构的不同，ActiveX 插件是构建在微软的 Active X 平台上的，而 Netscape 式插件是构建在 Netscape Plugin API 上的。最初，每个浏览器（包括 IE）都被迫支持 Netscape 式的插件，因为当时 Netscape 是占据统治地位的浏览器，只有兼容它的浏览器才能与之竞争。尽管很多人会揣测为何微软要终止对 Netscape 式插件的支持，但却是它造成了插件世界中一个清晰的裂缝。很多插件开发人员都必须同时创建 Netscape 式插件和这些插件的 ActiveX 包装

器来支持 IE。

今天，浏览器被分成不支持 Netscape 式插件（如 Windows 上的 IE）和支持 Netscape 式插件的浏览器（如 Mozilla、Opera、Safari 和很多其他浏览器）。

因为这些区别，就有了不同的检测插件是否已安装在指定的浏览器中的方法。

18.5.1 检测 Netscape 式插件

从 Netscape 3.0 开始，很多浏览器（尤其是基于 Mozilla 的）都可通过 JavaScript 来判断哪些 MIME 类型映射到了插件上，这就可以判断是否已经安装了给定的插件。这是通过 window.mimeTypes 集合来实现的。

有两种不同的 Netscape 式插件：用于 Netscape 4.x 的和新的用在 Mozilla 中的 Gecko 式的。详细的区别对 JavaScript 开发人员没有意义，因为两种插件的访问方式一样。

每个注册到插件上的 MIME 类型都在 window.mimeTypes 中，按照数字和 MIME 类型进行索引，可以直接访问 MIME 类型，也可以迭代这个集合。代表每个 MIME 类型的对象有四个特性：

- description——由 MIME 类型代表的文件类型的描述；
- enabledPlugin——指向包含指定插件信息的插件对象；
- suffixes——与 MIME 类型相关的文件后缀名（如将.gif 映射到 image/gif）；
- type——MIME 类型。

可以用一个简单的脚本来输出所有可见的 MIME 类型和相关描述：

```
<html>
  <head>
    <title>MIME Types Example</title>
  </head>
  <body>
    <script type="text/javascript">

      if (navigator.mimeTypes) {
        document.writeln("<h3>Supported MIME Types:</h3>");
        document.writeln("<ul>");
        for (var i=0; i < navigator.mimeTypes.length; i++) {
          document.writeln("<li>" + navigator.mimeTypes[i].type + " (" +
            navigator.mimeTypes[i].description + ", "
            + navigator.mimeTypes[i].suffixes + "</li>");

        }
        document.writeln("</ul>");
      }
    </script>
  </body>
</html>
```

535

这个例子没有输出关于 enabledPlugin 的信息，enabledPlugin 是另一个对象，包含以下特性：

- `description`——关于插件的描述;
- `filename`——插件文件名;
- `length`——与插件相关的 MIME 类型的数量;
- `name`——插件的名称。

如果 MIME 类型没有任何插件与之相关，则 `enabledPlugin` 为 `null`。记住，可以将前面的例子修改一下以使其包含所有的插件信息：

```

<html>
  <head>
    <title>MIME Types Example</title>
  </head>
  <body>
    <script type="text/javascript">

      if (navigator.mimeTypes) {
        document.writeln("<h3>Supported MIME Types:</h3>");
        document.writeln("<ul>");
        for (var i=0; i < navigator.mimeTypes.length; i++) {
          document.writeln("<li>" + navigator.mimeTypes[i].type + " (" +
            + navigator.mimeTypes[i].description + ", "
            + navigator.mimeTypes[i].suffixes + "</li>");

          if (navigator.mimeTypes[i].enabledPlugin) {
            var oPlugin = navigator.mimeTypes[i].enabledPlugin;
            document.writeln("<ul>");
            document.writeln("<li>Name: " + oPlugin.name + "</li>");
            document.writeln("<li>" + oPlugin.description + "</li>");
            document.writeln("<li>MIME types supported: " +
              + oPlugin.length + "</li>");
            document.writeln("<li>Filename: " + oPlugin.filename +
              + "</li>");
```

}

document.writeln("");

}

</script>

</body>

</html>

这个例子输出了每个 MIME 类型以及相关的插件信息（如果有的话）。

与前面的例子一样，这个页面输出了按照数字索引的 MIME 类型。某些 MIME 类型只按照 MIME 类型字符串进行索引，不能通过数字访问。例如，`text/html` 是一个注册到浏览器的 MIME 类型，但是它并没有在前面代码中生成的列表中。但是，可以直接对其进行检测：

```
alert( navigator.mimeTypes["text/html"] != null);
```

这些不可见的 MIME 类型一般都是直接由浏览器自己处理的，通常也不会有插件注册到它上面。可以用一个简单的页面来测试你想到的任何 MIME 类型：

```

<html>
  <head>
    <title>MIME Types Example</title>
    <script type="text/javascript">

      function findPlugin() {
        var sType = document.getElementById("txtMimeType").value;

        if (navigator.mimeTypes) {
          if (navigator.mimeTypes[sType]) {
            if (navigator.mimeTypes[sType].enabledPlugin) {
              alert("The MIME type '" + sType
                  + "' uses the plugin '" +
                  navigator.mimeTypes[sType].enabledPlugin.name
                  + "'.");
            } else {
              alert("The MIME type '" + sType
                  + "' has no registered plugin.");
            }
          } else {
            alert("The MIME type '" + sType
                + "' is not registered.");
          }
        } else {
          alert("Browser doesn't support navigator.mimeTypes.");
        }
      }
    </script>
  </head>
  <body>
    <p>Type the name of the <acronym title="Multipurpose Internet Mail Extension">MIME</acronym>
       you want to check and click the Find Plugin button.</p>
    <p><input type="text" id="txtMimeType" />
       <input type="button" value="Find Plugin" onclick="findPlugin()" /></p>
  </body>
</html>

```

537

前面列出来的 HTML 页面允许你在文本框中输入一个 MIME 类型，点击按钮就可以判断这个 MIME 类型是否被注册了。如果有这个类型，还能告知是否有相应的插件。findPlugin() 函数从文本框中获取文本，然后检查 navigator.mimeTypes 集合是否存在。如果存在，则函数继续检查这个 MIME 类型是否存在。如果是，再检查是否安装了插件。

可以将这个方法抽取出来，并用它检查是否有处理特定的 MIME 类型的插件：

```

function hasPluginForMimeType(sMimeType) {
  if (navigator.mimeTypes) {
    return navigator.mimeTypes[sMimeType].enabledPlugin != null;
  } else {

```

```

        return false;
    }
}

```

然后，可以这样检查给定的插件是否存在：

```

if (hasPluginForMimeType("application/x-shockwave-flash")) {
    alert("The Flash plugin is installed.");
} else {
    alert("The Flash plugin is not installed.");
}

```

也可以不用 MIME 类型来获取所有注册的插件的列表。navigator.plugins 集合包含了所有可用的插件，按照名称和数字进行索引。集合中的每个条目都是一个插件对象，等同于每个 MIME 类型的 enabledPlugin。所以，如果知道插件的名称（等同于它的 name 特性），就可以直接访问：

```
var oFlashPlugin = navigator.plugins["Shockwave Flash"];
```

可以迭代所有的插件并输出它们的信息。用这样一个简单的页面即可实现：

```

<html>
    <head>
        <title>Plugins Example</title>
    </head>
    <body>
        <script type="text/javascript">
            if (navigator.mimeTypes) {
                document.writeln("<h3>Loaded Plugins:</h3>");
                document.writeln("<ul>");
                for (var i=0; i < navigator.plugins.length; i++) {
                    document.writeln("<li>" + navigator.plugins[i].name + " (" +
                        navigator.plugins[i].description + "</li>");

                }
                document.writeln("</ul>");
            }
        </script>
    </body>
</html>

```

这个页面输出了所有的已注册的插件的列表以及它们的描述。但是插件对象还有一个秘密：它实际上是一个 MIME 类型的集合，也就是说这个插件对应的所有 MIME 类型都可以这样访问：

```
var oMimeType = navigator.plugins[0][0];
```

这行代码访问了由第一个插件支持的第一个 MIME 类型。所以现在，可以更新前面的例子，将为每个插件注册的 MIME 类型都列出来：

```

<html>
    <head>
        <title>Plugins Example</title>
    </head>
    <body>

```

```

<script type="text/javascript">

    if (navigator.mimeTypes) {
        document.writeln("<h3>Loaded Plugins:</h3>");
        document.writeln("<ul>");
        for (var i=0; i < navigator.plugins.length; i++) {
            document.writeln("<li>" + navigator.plugins[i].name + " (" +
                navigator.plugins[i].description + ")</li>");

            document.writeln("<ul>");
            for (var j=0; j < navigator.plugins[i].length; j++) {
                document.writeln("<li>" + navigator.plugins[i][j].type +
                    "</li>");

                document.writeln("</ul>");
            }
            document.writeln("</ul>");
        }
    }
</script>
</body>
</html>

```

现在这个例子显示了每个插件的名称和描述，然后是所支持的 MIME 类型的列表。

最后要注意的是，用户下载了浏览所需的插件后，`navigator.plugins` 集合有可能会过期，也就是陈旧了。考虑到这种可能性，使用前应该调用 `refresh()` 方法来刷新 `navigator.plugins`。这个方法接受一个参数——一个布尔值，表示浏览器是否应该重新载入页面；需要重新载入页面，则传入 `true`，否则，传入 `false`。例如：

```
navigator.plugins.refresh(true); //reload all pages using plugins
```

进行这样一次调用即可避免以后的麻烦。

539

目前，Netscape Navigator 3.0+、Opera 5.0+、Safari 1.0、IE 5.0+（仅 Macintosh 版）、IE 3.0-5.5 SP1（Windows 版），以及所有基于 Mozilla 的浏览器都支持这个功能；但 Windows 上的 IE 5.5 SP2 不支持，尽管它也创建了 `navigator.plugins` 和 `navigator.mimeTypes`（这两个集合都是空的）。

18.5.2 检测 ActiveX 插件

因为 IE 的插件就是 ActiveX 控件，所要知道的就是要检测的控件的名称。在本书前面，已经出现过检测微软 XML DOM 最新版本的代码。同样的方法也可用于检测 IE 插件。但是如何才能知道你所要的 ActiveX 控件的名称呢？

微软有个 OLE/COM Object Viewer 工具，用于找到所有安装在电脑上的 ActiveX 控件的名称。你可以从 http://www.microsoft.com/downloads/details.aspx?FamilyID=5233b70d-d9b2-4cb5-aeb6-4566_4be858b6&displaylang=en 上免费下载这个工具。安装后，它会给出安

装在机器上的所有 OLE、COM 和 ActiveX 对象的列表及它们的一些重要信息。在所有对象中逐个查找要花费一些时间，找到所要的那个对象后，就可以显示出所有需要的信息（图 18-1）。

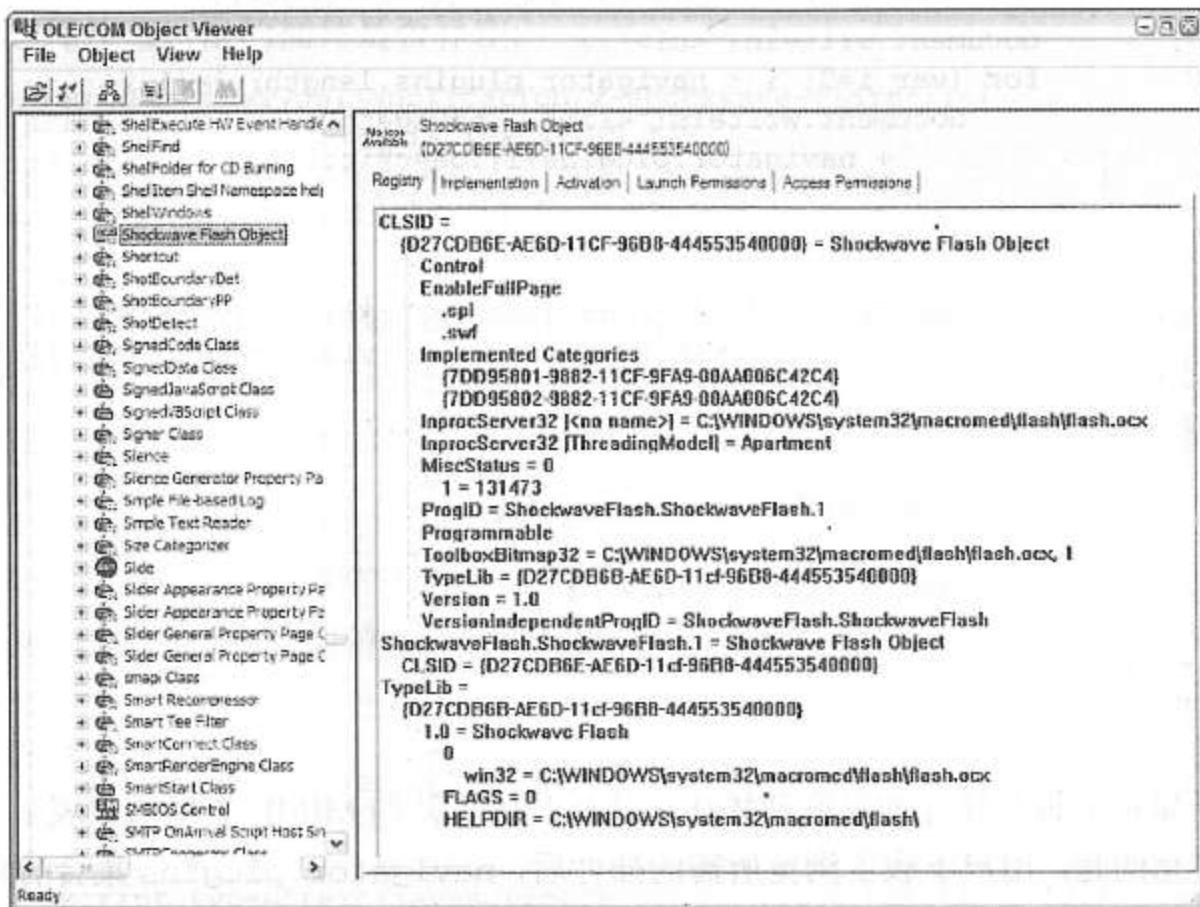


图 18-1

最重要的一部分信息是所列出的 VersionIndependentProgID，它给出了可以创建最新版控件的通用名称。ProgID 一般给出了特定版本的控件名称，它可用于帮助判断是否安装了特定版本的控件（不过并非总是列出可用的最高版本）。下面的表格列出了一些常见的插件的版本无关和版本特定的 ActiveX 控件的名称。

540

插件	版本无关的名称	特定版本的名称
Adobe (Acrobat) Reader	PDF.PdfCtrl	PDF.PdfCtrl.5 PDF.PdfCtrl.6
Adobe SVG Viewer	Adobe.SVGCtl	Adobe.SVGCtl.2 Adobe.SVGCtl.3
Macromedia Flash Player	ShockwaveFlash. ShockwaveFlash	ShockwaveFlash. ShockwaveFlash.6 ShockwaveFlash. ShockwaveFlash.7 ShockwaveFlash. ShockwaveFlash.8 ShockwaveFlash. ShockwaveFlash.9
Real Player 5	RealPlayer. RealPlayer(tm) ActiveX Control (32-bit)	无
Real Video 5	RealVideo. RealVideo(tm) ActiveX Control (32-bit)	无

541

(续)

插件	版本无关的名称	特定版本的名称
Real Player G2	rmocx.RealPlayer G2 Control	rmocx.RealPlayer G2 Control.1
Quicktime	Quicktime.Quicktime	Quicktime.Quicktime.1
Windows Media Player	WMPlayer.OCX	WMPlayer.OCX.7 WMPlayer.OCX.8

现在，尝试使用 ActiveXObject 创建指定名称的 ActiveX 控件对象。例如：

```
function detectFlashInIE() {
    try {
        new ActiveXObject("ShockwaveFlash.ShockwaveFlash");
        return true;
    } catch (oError) {
        return false;
    }
}
```

这个函数的目的是判断是否安装了 Flash Player。为此，它先尝试使用名称 ShockwaveFlash.ShockwaveFlash 来创建一个 ActiveXObject。如果成功了，就执行下一行并返回 true。如果无法创建这个对象，就捕获这个错误，并返回 false。同样的算法也可以用在任何有 ActiveX 包装器的插件上。

这个检测插件的方法只能在 Windows 上的 IE 5.0 或者更高版本中运行。

18.5.3 跨浏览器检测

可惜，没有用于建立一种通用的跨浏览器的插件检测的比较方便的方法。为了精确地检测某个插件是否可用，必须同时知道 MIME 类型和 ActiveX 控件名称。大部分情况下，开发人员只需创建包含两种检测手段的检测函数，如下：

```
function detectFlash() {
    if (navigator.mimeTypes.length > 0) {
        return navigator.mimeTypes["application/x-shockwave-flash"].enabledPlugin
    } != null;
    } else if (window.ActiveXObject) {
        try {
            new ActiveXObject("ShockwaveFlash.ShockwaveFlash");
            return true;
        } catch (oError) {
            return false;
        }
    } else {
        //no way to detect!
        return false;
    }
}
```

542

这个函数用对象检测法来判断要使用哪种方法。如果 navigator.mimeTypes 可用，则函数

使用 Netscape 式插件的检查方法来检查。否则，如果可以创建 ActiveXObject 对象，就使用 IE 的检测方法。如果两种都不行，函数就直接返回 `false`。

要检测指定的插件，可以自定义这个算法并创建一系列插件检测函数。

Macromedia 提供了一个 Flash 检测工具包（Flash Detection Kit），可以在 http://www.macromedia.com/software/download/detection_kit/ 获取，它可用于为 Flash 动画产生跨浏览器的 HTML 和 JavaScript。

18.6 Java applet

最老的插件形式——Java applet 最近被完全重定义了以使其工作于通用的浏览器插件框架中。以前，`applet` 必须用`<applet>`元素进行加载。HTML 4.0 不推荐使用`<applet>`，更倾向于把`<object>`作为嵌入插件的唯一手段。为此，Sun 公司创建了 Java 插件，已经作为 Java 运行时环境（Java Runtime Environment, JRE）的一部分包含在内了，可以在 <http://java.sun.com/>。

18.6.1 嵌入 applet

嵌入 applet 要求用`<object>`非标准的 `code` 特性来指定要载入的类。虽然大部分浏览器都能用 `data` 达到同样的效果，但是事实上这并不通用。所以，为实现跨浏览器，最好这样使用 `code`:

```
<object type="application/x-java-applet"
        code="ExampleApplet.class" width="100" height="100" id="ExampleApplet">
</object>
```

注意 Java applet 的 MIME 类型是 `application/x-java-applet`。通过明确指定它可以保证浏览器使用最恰当（可用）的 Java 插件的版本。可以在 MIME 类型的结尾指定确切的版本。例如，要指定 1.4.2 版（JRE 1.4.2），添加 "`jpi-version-1.4.2`"：

```
<object type="application/x-java-applet;jpi-version=1.4.2"
        code="ExampleApplet.class" width="100" height="100" id="ExampleApplet">
</object>
```

在这个 MIME 类型中，JPI 是 Java PlugIn 的缩写，它确保了除非用户机器上的插件版本为 1.4.2，否则不运行插件。所以即使用户有 1.4.3 版，`applet` 也不会运行。因此，最好省略插件版本以避免烦扰用户。

`Applet` 类可以包含在 JAR（Java Archive，Java 档案）文件中。这样，就要用 `archive` 特性来指定 JAR 文件：

```
<object type="application/x-java-applet"
        archive="ExampleArchive.jar"
        code="ExampleApplet.class" width="100" height="100" id="ExampleApplet">
</object>
```

为支持 Netscape Navigator 4.x，你需要使用原始的`<applet>`元素。为确保代码能在所有浏览器中使用，要将`<applet>`嵌入在`<object>`中（类似于将`<embed>`嵌在`<object>`中）。但

是，这么做还要求使用IE特有的`<comment/>`元素。虽然Mozilla和其他浏览器会忽略`<object/>`中的内容，但是IE不会，最后可能会渲染出两份同样的applet。加入`<comment/>`元素可以让IE忽略额外的内容：

```
<object type="application/x-java-applet"
        code="ExampleApplet.class" width="100" height="100" id="ExampleApplet">
<comment>
    <applet code="ExampleApplet.class" width="100" height="100"
            name="ExampleApplet"></applet>
</comment>
</object>
```

18.6.2 在 JavaScript 中引用 applet

把applet加入到HTML页面中后，还需要一种通过JavaScript来访问它的方法。以前，applet可以通过`document.applets`集合来引用，它通过`name`特性及在文档中的位置索引了所有的`<applet/>`元素（类似于`document.forms`和`document.frames`）。例如，要获取`name`特性为"ExampleApplet"的applet的引用，可以这样：

```
var oApplet = document.applets["ExampleApplet"];
```

但是，如果用`<object/>`元素来嵌入applet，`document.applets`集合就不会将其包含在内。使用`<object/>`时，你可以用`document.getElementById()`来访问该applet：

```
var oApplet = document.getElementById("ExampleApplet");
```

如果因为兼容性，在老的浏览器中同时使用`<object/>`和`<applet/>`，则需用一个函数来判断应该使用哪个方法：

```
function getApplet(sName) {
    if (document.getElementById) {
        return document.getElementById(sName);
    } else {
        return document.applets[sName];
    }
}
```

```
var oApplet = getApplet("ExampleApplet");
```

获取了applet的引用后，就可以从JavaScript直接访问applet所有的公用方法，如：

```
oApplet.appletPublicMethod();
```

这可以将Java applet作为宿主环境来打开所有JavaScript的功能，然后JavaScript就可以控制Applet来完成一些事情。

18.6.3 创建 applet

要自己写Java applet，则必须首先从Sun的网站上下载Java开发工具包（Java Development Kits，JDK，在`http://java.sun.com/j2se/`）。使用集成开发环境还是普通文本编辑器来写applet则由你自己决定。所有的applet都有一个共性：必须继承于`java.applet.Applet`。（不过，也可以创建基

于 Swing 的 Java applet——继承 javax.swing.JApplet，其实它也继承于 java.applet.Applet)。

下面是个简单的 applet 例子：

```
import java.applet.Applet;
import java.awt.Graphics;
import java.awt.HeadlessException;

public class ExampleApplet extends Applet {

    private String message = "Hello World!";

    public ExampleApplet() throws HeadlessException {
        super();
    }

    public void paint(Graphics g) {
        g.drawString(message, 20, 20);
    }

    public void setMessage(String message) {
        this.message = message;
        repaint();
    }
}
```

这个 applet 仅显示了文本 "Hello World!"。paint()方法控制了 applet 第一次载入时的显示，它接受一个 Graphics 对象作为唯一的参数。Graphics 对象代表 applet 的可视区域，它有一些绘制方法，如 drawString()，可以将指定文本绘制在给定的 x 和 y 坐标上。

545 前面定义的 applet 也有个私有特性 message，它首先初始化为 "Hello World!"。可用 setMessage() 来更改这个特性，因为这是个公用方法，所以也可以从 JavaScript 中访问它。这个方法就是将 message 特性设置为指定的字符串，然后调用 repaint() 来清除 applet 的显示，再调用 paint() 方法。

applet 只需要一个默认的构造函数（如果操作系统没有图形界面会抛出 HeadlessException 异常）。如果这个 applet 的目的仅是为 JavaScript 提供 Java 功能，则只需添加公用方法，同时在 HTML 中将 applet 的长宽设置为 0。但是，如果 applet 要显示某些内容，则最有可能需要定义 paint() 方法。

定义了 applet 后，将其保存为 .java 文件，使用 javac 工具对其进行编译：

```
javac ExampleApplet.java
```

这个命令会创建一个以 .class 为扩展名的文件（前面的例子会出现 ExampleApplet.class）。.class 文件必须放在 Web 服务器的目录中才能允许 HTML 页面访问它。

18.6.4 JavaScript 到 Java 的通信

现在，创建了一个公用方法的 applet，你可以将其放在页面中并通过 JavaScript 对其进行访问。下面的例子使用前面一节的 ExampleApplet，并演示了如何用 JavaScript 来更改显示在

applet 中的文本：

```

<html>
  <head>
    <title>Applet Example</title>
    <script>
      function changeAppletMessage() {
        var oApplet = document.getElementById("ExampleApplet");
        var oTextbox = document.getElementById("txtMessage");
        oApplet.setMessage(oTextbox.value);
      }
    </script>
  </head>
  <body>
    <p>Enter the message you want to see in the applet.</p>
    <p><input type="text" id="txtMessage" size="10" />
    <input type="button" value="Set Message" onclick="changeAppletMessage()" />
    </p>

    <object type="application/x-java-applet" code="ExampleApplet.class"
            width="100" height="100" border="1" id="ExampleApplet">
      <comment>
        <applet code="ExampleApplet.class" width="100" height="100"
                name="ExampleApplet"></applet>
      </comment>
    </object>
  </body>
</html>

```

546

这个例子向用户展示一个文本框，用于输入新的消息。用户点击 Set Message 按钮时，就调用 changeAppletMessage() 函数，它会获取 applet 的引用，并取出文本框中的文本，然后调用 applet 的 setMessage() 方法，传入输入的文本。显示在 Applet 中的消息从 "Hello World!" 变成用户输入的内容（有时很快，有时会很慢）。

1. 类型转换

尽管 JavaScript 与 Java 通信对开发者而言是个强大的工具，但是并不是没有一点问题。在前面的例子中，从 JavaScript 为 Java 方法传了一个字符串，没有出现问题。这是因为 JavaScript 的 String 对象映射到了 Java 的 String 对象。对于 JavaScript 中的简单类型，都是没有问题的，因为它们在 Java 中都有等同的类型。但如果要为 Java 方法传递对象就会遇到问题，因为没有等同的类。因此，要被 JavaScript 调用的方法最好只接受简单类型的值。

2. 处理 Java 异常

在 Java 中，异常要比在 JavaScript 中常见，所以必须小心可能会出现异常。访问可能会发生错误的 applet 时，要将其放到 try...catch 语句中。JavaScript 的 try...catch 语句也可以捕获 applet 抛出的异常。

假设将 ExampleApplet 改成，在 setMessage() 传入一个空字符串时，抛出一个错误，如：

```

import java.applet.Applet;
import java.awt.Graphics;
import java.awt.HeadlessException;

```

```

public class ExampleApplet2 extends Applet {

    private String message = "Hello World!";

    public ExampleApplet2() throws HeadlessException {
        super();
    }

    public void paint(Graphics g) {
        g.drawString(message, 20, 20);
    }

    public void setMessage(String message) throws Exception {
        if (message.length() == 0) {
            throw new Exception("Message must have at least one character.");
        }
        this.message = message;
        repaint();
    }
}

```

547

然后，就可以用 JavaScript 的 try...catch 语句来捕获这个错误：

```

var oApplet = document.getElementById("ExampleApplet");
var oTextbox = document.getElementById("txtMessage");

try {
    oApplet.setMessage(oTextbox.value);
} catch (oError) {
    alert("Error caught!");
}

```

因为浏览器的不同，很难说 oError 返回的是什么。IE 返回一个表示 Java 异常的 JavaScript 对象，Mozilla 则返回 Java 异常对象本身。两个浏览器也有不同的访问错误信息的方法：在 IE 中，oError.message 特性显示了 Java 异常信息；在 Mozilla 中，toString() 方法返回一个 Java 异常的字符串，但不包含产生异常的方法。不过，大部分情况下，这对已发生的错误已经足够了解。

3. 安全限制

尽管 Java 比 JavaScript 强大得多，但当它被包含在页面中时却不能自由操纵浏览器。Java applet 必须遵循浏览器设置的一系列严格的规则。（在不同浏览器中，规则也不同，不过有些规则是基本相同的）。我们将这个行为称为沙盒化（sandboxing）。

首先，applet 不允许访问用户的文件系统。这防止了一个主要的安全问题——如果某个没有任何防备的用户打开了包含恶意 applet 的页面。默认情况下，这是不可能的。

其次，applet 不允许访问跨域的资源。这是与前面讨论过的 XML HTTP 请求中一样的安全限制。也可以通过数字签署 applet 来绕开这些限制。applet 被签署后，会出现一个对话框询问签名是否有效，也就是 applet 是否应该获取尚不可用的加强特权。如果接受了这个签名，前面提到的限制就不复存在了。

你可以在 <http://java.sun.com/developer/technicalArticles/Security/Signed/> 获取更多关于 Applet 安全性的资料。

18.6.5 Java 到 JavaScript 的通信

不仅 JavaScript 可以访问包含在 Java applet 中的方法，applet 其实也可以通过 LiveConnect 来访问 JavaScript 对象和方法。在本书的前面提到过，LiveConnect 可作为 JavaScript 访问 Java 的一个途径。同时，它也可用于将 applet 和 JavaScript 更紧密地整合在一起，这要用到一个特殊的 Java 包：`netscape.javascript`。

这个包包含两个类：`JSObject`，它是 JavaScript 对象的 Java 表示形式；`JSEException`，代表一个 JavaScript 错误。不过，`JSObject` 才是 Java 到 JavaScript 的通信的焦点。548

`JSObject` 类有以下方法：

- `getMember(String name)`——获取某个对象的命名特性。等同于 JavaScript 中的 `oObject.property` 或者 `oObject["property"]`。返回值是一个 Java Object。
- `getSlot(int index)`——获取对象的数字索引特性（主要用于 JavaScript 数组）。等同于 `oObject[index]`。返回值是一个 Java Object。
- `setMember(String name, Object value)`——设置命名特性的值。
- `setSlot(int index, Object value)`——设置数字索引特性的值。
- `removeMember(String name)`——删除命名特性。
- `call(String methodName, Object args[])`——调用给定名称的方法，传入的参数包含在 Object 数组中。返回值是一个 Java Object。
- `eval(String code)`——在对象的上下文中计算一段 JavaScript 代码的字符串；类似于 JavaScript 的 `eval()` 函数，返回值是一个 Java Object。
- `equals(Object object)`——判断这个对象是否等于另一个对象。

`JSObject` 有个静态的方法 `getWindow()`，以 Java Applet 对象为参数，并返回 JavaScript `window` 对象的 `JSObject` 表示形式。

使用 `JSObject` 和 LiveConnect 需要一个适应过程。假设要获取当前载入的 URL。在 JavaScript，只需用一行代码：

```
var sURL = window.location.href
```

在 Java 中，涉及的内容要多一些：

```
JSObject window = JSObject.getWindow(this);
JSObject location = (JSObject) window.getMember("location");
String sURL = location.getMember("href").toString();
```

第一行代码获取 JavaScript `window` 对象的引用（参数 `this` 在被调用的函数中代表 applet 本身）。第二行代码通过使用 `getMember()` 获取 `location` 对象的引用。因为 `getMember()` 方法返回一个 `Object`，必须将其强制为 `JSObject` 才能访问 `JSObject` 的方法。最后一行代码用

`getMmeber()` 获取 `href` 的值，返回值仍是一个 `Object`，这意味着要调用 `toString()` 来得到字符串数值。显然，Java 访问 JavaScript 对象的代码比较冗长，不过还是把事情解决了。要在 applet 中使用 LiveConnect，必须导入包 `netscape.javascript`。没必要将这个包与 applet 一起发布，因为它被构建在 Java 插件中了。所以，只要在 `.java` 文件中加入下面这一行，并开始编码：

549 import `netscape.javascript.*`;

下面是一个使用 LiveConnect 包的 applet 例子：

```
import java.applet.Applet;
import java.awt.Graphics;
import java.awt.HeadlessException;
import netscape.javascript.*;

public class ExampleApplet3 extends Applet {

    public ExampleApplet3() throws HeadlessException {
        super();
    }

    public void paint(Graphics g) {
        JSObject window = JSObject.getWindow(this);
        JSObject document = (JSObject) window.getMember("document");
        JSObject location = (JSObject) window.getMember("location");

        g.drawString("Title: " + document.getMember("title"), 10, 20);
        g.drawString("URL: " + location.getMember("href"), 10, 40);

        window.eval("getMessageFromApplet(\"Hello from the Java applet!\")");
    }
}
```

这个 applet 使用 applet 的 `paint()` 方法与页面的 JavaScript 进行交互。这个方法先用前面讨论过的方法获取 `window`、`document` 和 `location` 对象的引用。然后，applet 将页面的标题（从 `document.title` 中获取）和页面的 URI 绘制到了 applet 的画布上。

最后，使用 `window` 的 `eval()` 方法调用 `getMessageFromApplet()`，这是定义在包含了 applet 的 HTML 页面中 JavaScript 函数。applet 初始化时，如果函数不存在，就会发生 `JSException`。

当包含了一个使用 LiveConnect 包的 applet 时，必须设置特殊的参数来允许其对 HTML 文档进行访问。参数名为 `mayscript`，应设为 `true`。

```
<object type="application/x-java-applet" code="ExampleApplet3.class"
        width="100" height="100" id="ExampleApplet">
    <param name="mayscript" value="true" />
</object>
```

如果用老的 `<applet>` 元素，要将 `mayscript` 作为一个特性包含进去：

```
<applet code="ExampleApplet3.class" mayscript="mayscript"
        width="100" height="100" name="ExampleApplet">
</applet>
```

回到前面的例子，`getMessageFromApplet()` 函数只显示了传入的参数。下面是页面上完整的 HTML 代码：

```
<html>
  <head>
    <title>Applet Example</title>
    <script>
      function getMessageFromApplet(sMessage) {
        alert("Applet says: " + sMessage);
      }
    </script>
  </head>
  <body>
    <p>This page defines a function that is called by the applet once
    it is loaded.</p>
    <object type="application/x-java-applet" code="ExampleApplet3.class"
            width="400" height="50" id="ExampleApplet">
      <param name="mayscript" value="true" />
    </object>
  </body>
</html>
```

载入这个页面后，applet 显示页面的标题（Applet Example）以及正在浏览的 URL。然后，应该看到一个警告框，显示“Applet says: Hello from the Java applet!”

当然也可以用 `call()` 来执行 `getMessageFromApplet()`：

```
Object[] args = new Object[1];
args[0] = "Hello from the Java applet!";
window.call("getMessageFromApplet", args);
```

因为 `getMessageFromApplet()` 是个全局函数，其实就是 `window` 的一个方法，因此可以通过调用 `window` 的 `call()` 方法来调用它。

18.7 Flash 动画

Macromedia Flash 起初是作为嵌入到网页上的基于矢量的小型动画而诞生的，现在已经成长为一个可以制作整个网站和 Web 应用的开发环境了。Macromedia Flash 已经不再是单纯的动画制作工具，而是完整的针对 Web 应用的开发环境。

Flash 动画，顾名思义，是使用 Macromedia 特有的 Flash 和 Flash MX 开发环境（不过，现在很多图形程序都可以基于 Macromedia 的 Open SWF initiative（开放 SWF 提议）来导出 Flash 动画）创建的。由于它的普遍性，现在大部分浏览器都搭载了 Flash 插件，也就是说大部分用户无需手工下载、安装插件就可以享受它的好处（当然，除非要进行升级）。

18.7.1 嵌入 Flash 动画

要在 HTML 页面中嵌入 Flash，还是要用 `<object>` 元素：

```
<object type="application/x-shockwave-flash" data="myflashmovie.swf"
        width="100" height="100" id="FlashMovie"></object>
```

基于 Mozilla 的浏览器不能正确显示，所以必须添加 movie 参数，并将其设置为和 data 特属性一样的 URL：

```
<object type="application/x-shockwave-flash" data="myflashmovie.swf"
        width="100" height="100" id="FlashMovie">
    <param name="movie" value="myflashmovie.swf" />
</object>
```

当然，如果还要支持 Netscape 4.x，就要包含<embed/>元素：

```
<object type="application/x-shockwave-flash" data="myflashmovie.swf"
        width="100" height="100" id="FlashMovie">
    <param name="movie" value="myflashmovie.swf" />
    <embed type="application/x-shockwave-flash" src="myflashmovie.swf"
           width="100" height="100" quality="high" name="FlashMovie">
    </embed>
</object>
```

18.7.2 引用 Flash 动画

与 Java applet 一样，根据嵌入的方式，Flash 动画也可以用多种方式引用（<object/>使用 document.getElementById() 或者<embed/>使用 document.embeds）。下面的函数可以用于获取 Flash 动画的引用而不管使用的是何种嵌入方法：

```
function getFlashMove(sName) {
    if (document.getElementById) {
        return document.getElementById(sName);
    } else {
        return document.embeds[sName];
    }
}
```

可以这样调用这个函数：

```
var oFlashMovie = getFlashMovie("FlashMovie");
```

获取动画的引用后，就可以用 JavaScript 进行通信了。

18.7.3 JavaScript 到 Flash 的通信

Flash 动画提供了一些 JavaScript 可以访问的标准方法：

- GetVariable(variable_name)——获取 Flash 动画变量的值；
- GotoFrame(frame_number)——将当前的 Flash 帧设置到指定的帧数；
- IsPlaying()——表示是否正在播放 Flash 动画；
- LoadMovie(layer_num, url)——将指定 URL 上的 Flash 动画载入到指定的 Flash 层上；
- Pan(x, y, mode)——将放大的动画平移到指定坐标。mode 参数是 0，表示坐标单位为像素，或者 1，表示坐标为百分比；

- PercentLoaded()——返回 Flash 动画已经载入的比例（0 到 100 的数字）；
- Play()——从当前位置开始播放动画；
- Rewind()——将动画重置到第一帧；
- SetVariable(variable_name, value)——设置 Flash 动画变量的值；
- SetZoomRect(left, top, right, bottom)——设置要放大的区域；
- StopPlay()——停止 Flash 动画的播放；
- TotalFrames——返回 Flash 动画中帧的总数；
- Zoom(percent)——放大给定的百分比。

这些方法可以直接用于动画对象本身，所以要停止动画 ExampleMovie，如下：

```
var oFlashMovie = getFlashMovie("ExampleMovie");
oFlashMovie.StopPlay();
```

要获取动画中的总帧数，可用 TotalFrames()。然而，必须小心某些跨浏览器的兼容性问题。在 Windows 上的 IE 中，有个错误——TotalFrames 是整数值而不是函数。所以跨浏览器的功能必须用 typeof 判断 TotalFrame 是数字还是函数：

```
function getTotalFrames(sName) {
    var oFlashMovie = document.getElementById(sName);

    if (typeof oFlashMovie == "function") {
        return oFlashMovie.TotalFrames();
    } else {
        return oFlashMovie.TotalFrames;
    }
}
```

553

可以用 SetVariable() 和 GetVariable() 向 Flash 动画传送信息和从中获取信息。若要这样，Flash 动画必须有个监视值变化的变量。最简单的方法就是创建一个文本字段并将其值绑定到一个变量上（从 Flash Properties 面板中选择 Dynamic Text，并输入变量的名称）。然后，用 GetVariable() 获取变量的值或者用 SetVariable() 改变它的值。两个方法都要求变量与它的时间线相关。所以，要访问在主时间线中的变量 message，两个方法的第一个参数都是 "/:message"，其中 "/" 代表默认时间线，冒号表示要访问的时间线的一部分，message 是变量名。例如：

```
var sOriginalMessage = oFlashMovie.GetVariable("/:message");
oFlashMovie.SetVariable("/:message", "my new message");
```

对于动画中特定的时间线，也可以使用某些方法：

- TCallFrame(timeline, frame_number)——执行在给定位置上的帧的动作；
- TCallLabel(timeline, frame_label)——执行由给定标签所代表的帧上的动作；
- TCurrentFrame(timeline)——返回在时间线中当前帧的位置；
- TCurrentLabel(timeline)——返回在时间线中当前帧的标签；
- TGetProperty(timeline, property_constant)——返回由 property_constant（以后讨论）表示的特性的值的字符串形式；

- TGetPropertyAsNumber(timeline, property_constant)——返回由 property_constant (以后讨论) 表示的特性的值的数字形式;
- TGotoFrame(timeline, frame_number)——将动画设置到时间线中给定位置的帧;
- TGotoLabel(timeline, frame_label)——将动画设置到时间线中给定标签的帧;
- TPlay(timeline)——播放在给定时间线上的动画;
- TSetProperty(timeline, property_constant, value)——设置由 property_constant 表示的特性的值;
- TStopPlay(timeline)——停止给定时间线上的动画。

TGetProperty()、TGetPropertyAsNumber() 和 TSetProperty() 方法都用常数来表示要获取或设置的特性。因为在 Flash 中只能访问常数，JavaScript 必须使用数值。这些常数列在了下表中：

Flash 常数	值	描述
X_POS	0	动画的 x 坐标
Y_POS	1	动画的 y 坐标
X_SCALE	2	水平放大系数
Y_SCALE	3	垂直放大系数
CURRENT_FRAME	4	动画中当前帧的位置
TOTAL_FRAMES	5	动画中总共的帧数
ALPHA	6	动画的透明度 (0 到 100 之间的数)
VISIBLE	7	动画是否可见
WIDTH	8	动画的宽度
HEIGHT	9	动画的高度
ROTATION	10	动画旋转的角度
TARGET	11	时间线名称 (和与时间线有关的方法的第一个参数一样)
FRAMES_LOADED	12	当前载入的帧的数量
NAME	13	动画的名称
DROP_TARGET	14	在动画中的放置目标的名称, 如果有的话
URL	15	动画的 URL
HIGH_QUALITY	16	动画是否以高质量渲染模式播放 (1 是, 0 否)
FOCUS_RECT	17	是否应该显示焦点方框 (1 是, 0 否)
SOUND_BUF_TIME	18	应该被缓冲来产生不间断回放的声音时间量

使用时间线特定的方法时, 第一个参数总是要操作的时间线的名称。默认的时间线由一个斜杠表示:

```
var iXPos = oFlashMovie.TGetProperty("/", 0);
```

第一个参数映射到 TARGET 特性:

```
var sTarget = oFlashMovie.TGetProperty("/", 11);
alert(sTarget == "/"); //outputs true
```

因为 Flash 使用了基于 ECMAScript 的脚本语言——ActionScript，所以它与 JavaScript 可以无缝交互。

18.7.4 Flash 到 JavaScript 通信

Flash 也可以与嵌入它的 HTML 页面上的 JavaScript 进行交互。用`<object>`元素嵌入 Flash 动画时，会默认启用这个功能。如果用`<embed>`元素，即使是向下兼容的，也必须添加一个特殊的`swLiveConnect`特性：

```
<object type="application/x-shockwave-flash" data="myflashmovie.swf"
        width="100" height="100">
    <param name="message" value="Hello World! " />
    <embed type="application/x-shockwave-flash" src="myflashmovie.swf"
           width="100" height="100" swLiveConnect="true">
        <param name="message" value="Hello World! " />
    </embed>
</object>
```

设置了这个特性后，就可以打开 Flash 到 JavaScript 的通信渠道了。

Flash 提供了两种不同的获取交互的方法：`getURL()` 和 `fsCommand()`。两者都只能给 JavaScript 发送简单类型值，两者都有各自的强项和弱点。

1. `getURL()`

`getURL()` 函数是通用的与浏览器交互的方法。它可用于在浏览器窗口（或在新的窗口中）中打开一个文档，类似于 JavaScript 中的 `window.open()`。例如，你可以在新的浏览器窗口中打开 `www.wrox.com`：

```
getURL("http://www.wrox.com", "_blank");
```

因为 `getURL()` 只是简单的把给定的 URL 传递给浏览器，所以它也可以接受 `javascript:` 的 URL。例如，假设你有个函数 `getMessageFromFlash()`，它以一个字符串为唯一的参数，然后将字符串显示在警告框中，如下：

```
function getMessageFromFlash(sMessage) {
    alert("Flash says: " + sMessage + ".");
}
```

在 Flash 动画中，创建一个按钮并为其添加以下 ActionScript：

```
on(release) {
    getURL("javascript:getMessageFromFlash(\"Hello from Flash!\")");
}
```

导出了这个动画并将其嵌在包含 `getMessageFromFlash()` 的 HTML 页面中后，点击按钮会弹出 JavaScript 警告框，并显示文本“Hello From Flash！”

`getURL()` 函数是从 Flash 中调用 JavaScript 的最简单的方法，不过还有另一种方法。

2. `fscommand()`

在 Flash 中使用 `fscommand()` 就像给 JavaScript 发送消息。消息由一个命令（表示动画所期

望的一个动作) 和一个参数(虽然 Flash 允许输入多个参数, 但无法被 JavaScript 正常处理) 组成。从 Flash 动画中进行的典型调用:

```
556    on(release) {
        fscommand("send_message", "the message");
    }
```

为了让 JavaScript 处理这个命令, 必须定义一个特殊的函数。这个函数必须以 Flash 动画对象的名称(name, 有时是 ID) 开始, 并跟着_DoSCommand, 如下:

```
function ExampleMovie_DoSCommand(sCommand, vArgument) {
    ...
}
```

该函数接受两个参数: 命令和命令的参数。命令必须是一个字符串, 但命令的参数可以是任意的基本类型。对 fscommand() 的调用会被转到这个函数中, 所以通过提供不同的命令, 就可以判断应该使用 JavaScript 进行什么动作。例如:

```
function ExampleMovie_DoSCommand(sCommand, vArgument) {
    switch(sCommand) {
        case "change_color":
            //change the color of something
            break;

        case "change_height":
            //change the height of something
            break;

        //etc.
    }
}
```

使用除 IE 之外的任何浏览器时, 执行 fscommand() 就会调用这个函数。对于 IE, 还需要另外一个函数。

出于某些未知的原因, Flash Player 在 IE 中将 fscommand() 调用导到了 VBScript, 而不是 JavaScript。VBScript 是只用于 IE 的技术, 它允许开发人员使用 Visual Basic 代码对网页进行编程。起初它打算与 JavaScript 进行竞争, 但 VBScript 并没有获得青睐, 其他浏览器也没有对其进行支持, 完全是一个鸡肋。然而, 仅在 IE 的解决方案下工作的开发人员, 可以考虑使用它。

为避免写很多 VBScript 代码, 你可以只写一个将命令和参数传递给 JavaScript 函数的简单的函数。这段代码必须包含在 language 特性为"VBScript"的<script>元素中。VBScript 函数的形式与前面的 JavaScript 函数类似, 但是以_FSCCommand 结尾, 而不是_DoSCommand:

```
<script language="VBScript">
    Sub ExampleMovie_FSCCommand(ByVal sCommand, ByVal vArgument)
        call ExampleMovie_DoSCommand(sCommand, vArgument)
    end sub
</script>
```

所有非 IE 的浏览器都会忽略这段代码, 因为"VBScript"是无法识别的脚本语言; 因此, 可以安全地将它放入跨浏览器的页面中(不过仍然建议将这段代码放入<head>元素中, 以免将代

码渲染成普通文本)。另一种方法是将 VBScript 放入外部文件,(以.vbs 结尾)然后将其加载到页面中:

```
<script language="VBScript" src="example.vbs"></script>
```

因为 fscommand()的这种复杂情况,很多开发人员都只选择 getURL()来进行 JavaScript 通信。

18.8 ActiveX 控件

在 Windows 的 IE 中,可以用<object/>元素在页面中嵌入 ActiveX 控件。为此你需要知道要嵌入的 ActiveX 控件的 class ID(可以使用 OLE/COM Object Viewer 来获取这个信息)并将其插入到 classid 特性中:

```
<object classid="activex_class_id" id="ActiveXControl"></object>
```

不是所有的 ActiveX 控件都能在嵌入到网页面中后仍然正常工作。其实,某些控件在 IE 和诸如杀毒软件之类的程序中引发安全警告。不过,还是有一些 ActiveX 控件是为能在网页中正常且安全的工作而设计的。比如表格数据控件。

基于 ActiveX 的插件不能在非 Windows 的浏览器上工作,因为 ActiveX 是 Windows 特有的技术。在 Windows XP Service Pack 2 中,任何试图访问服务器外部资源或者有访问本地文件能力的 ActiveX 控件,都会引发警告。

表格数据控件没有可视化的组件,所以它在页面上是不可见的对象。乍看上去它没啥意思,尤其是与 Java、Flash 的功能相比。不过,再看一下,就会发现表格数据控件的强大功能,它可以令 JavaScript 使用类似数据库的功能。

特别地,表格数据控件可以用普通文本文件来模拟数据库表。可以告诉控件分隔字符的含义,然后将文本文件解析成一系列行和值。

要在页面上创建表格数据控件,必须用 class ID "CLSID:333C7BC4-460F-11D0-BC04-0080C7055A83"(使用 OLE/COM Object Viewer 获取)并给对象设定一个 ID:

```
<object classid="CLSID:333C7BC4-460F-11D0-BC04-0080C7055A83" id="TextData">
</object>
```

下面,必须指定 DataURL 参数,它给出文本文件的位置(相对的或绝对的 URL 都可以),以及 FieldDelim 参数,它给出了同一行中的两个值之间的分隔符(通常会是一个逗号):

```
<object classid="CLSID:333C7BC4-460F-11D0-BC04-0080C7055A83" id="TextData">
  <param name="DataURL" value="Names.txt" />
  <param name="FieldDelim" value="," />
</object>
```

最后,如果要使用的文件以第一行作为表格头,还要将 UseHeader 参数设置为 true,它不会将第一行包含在数据值中,并且通过名称来引用列:

```
<object classid="CLSID:333C7BC4-460F-11D0-BC04-0080C7055A83" id="TextData">
  <param name="DataURL" value="Names.txt" />
  <param name="FieldDelim" value="," />
  <param name="UseHeader" value="true" />
</object>
```

这段代码的意思是，将 Names.txt 中的数据加载到表格数据控件中，文件以“,”为分隔符，并把表格头作为每一列的键。

下面是一个 Names.txt 文件的例子：

```
first_name,last_name
Nicholas,Zakas
Michael,Smith
Joyce,Anderson
Benjamin,Johnson
Amy,Jones
```

第一列称为 first_name，第二列称为 last_name。每行都用逗号（是用 FieldDelim 参数指定的）来分割值。

页面的 load 事件只在表格数据控件完全载入后才触发，所以用 onload 事件处理函数来判断数据是否可用是安全的。然后，通过访问控件的 recordset 特性来获取被创建的记录集：

```
window.onload = function () {
  oDataset = document.getElementById("TextData").recordset;
};
```

这段代码创建了全局变量 oDataset，它指向包含所有数据的记录集。

可用 moveFirst()、moveLast、moveNext() 和 movePrevious() 方法以及 EOF (end of file, 文件结束) 和 BOF (beginning of file, 文件开始) 标记来迭代这个记录集，下面是个简单的从第一个记录迭代到最后一个记录结束的代码框架：

```
oDataset.moveFirst();
while (!oDataset.EOF) {
  //do something with the data here
  oDataset.moveNext();
}
```

559

前面代码中的第一步是，将记录集设置到第一个记录。然后 while 循环检测 EOF，如果是 true，则表示到达了文件结尾。循环体内是数据处理的过程，然后移动到下一条记录。

也可以用相反的方法在记录集中反向移动：

```
oDataset.moveLast();
while (!oDataset.BOF) {
  //do something with the data here
  oDataset.movePrevious();
}
```

这段代码从最后一个记录开始并反向移动到第一个。接触到文件开始时，BOF 特性等于 true。

要获取数据，可以用记录集的 fields() 方法以及列名（在前面的例子中，是 first_name 和 last_name）或者列在表格中的位置，从 0 开始。这个方法返回代表字段的对象，它的 value 特性包含了实际的值。所以要显示 Names.txt 中每个人的 first_name，则使用下面的代码：

```
oDataset.moveFirst();
while (!oDataset.EOF) {
    alert(oDataset.fields(0).value);
    oDataset.moveToNext();
}
```

也可以用 first_name 来替代 0：

```
oDataset.moveFirst();
while (!oDataset.EOF) {
    alert(oDataset.fields("first_name").value);
    oDataset.moveToNext();
}
```

这样，就可以迭代整个记录集并获取需要的数据了。

如果你仅为 Windows 的 IE 进行开发，则表格数据控件是很有用的控件。它还有很多用途，在这里还可以讨论更多的内容。因为它被限定为以 IE 为浏览器、Windows 为平台，所以这也限制了它在 Web 开发中的使用和采纳。现在，这种解决方案最适于在内部网络应用，因为其中的用户群可以排他性地保持于 IE 和 Windows 上。

560

18.9 小结

本章介绍了插件的概念以及如何用 JavaScript 与它们进行通信。还学习了不同类型的插件，ActiveX 和 Netscape 式的，以及两者如何嵌入到网页中。

接下来讨论了 Java applet 及其在 Web 上的使用。学习了 JavaScript 如何通过 LiveConnect 来访问 Java applet 的公用方法，以及 Java applet 如何直接调用 JavaScript。

然后，学习了 Macromedia Flash 及如何在 HTML 页面中嵌入 Flash 动画。学习了大量的 JavaScript 控制 Flash 动画的方法以及如何传送和获取数据。另外，还学习了两种从 Flash 动画中调用 JavaScript 的方法——getURL() 和 fsCommand()。

本章最后讨论了嵌入的 ActiveX 控件，特别讨论了表格数据控件。学习了如何从普通文本文档中将数据载入到表格数据控件中，以及如何从 JavaScript 中访问这些数据。

561

562

现在，已经编好了 Web 应用或网站的 JavaScript 的代码，并且已对其进行了充分地调试，确保它在每个目标浏览器上都能正常运行。那么，现在就要着手部署工作了，这又会引出一些新的问题。虽然你可以在不同的操作系统上用不同的浏览器来测试你的代码，看上去几乎没有什会引发意料之外的事情的浏览器行为。可能是操作系统的一个升级包，也可能是浏览器的一个补丁或者也可能是跨平台行为上的一个小小的不同，都可能造成部署上的问题。你需要担心的问题就是如何在开发环境之外建立一个系统。

19.1 安全性

对于任何基于 Web 的系统（无论是纯信息性的还是一个网上商店）来说，最重要的问题就是安全性。JavaScript 中有各种各样的安全检查以防止恶意脚本攻击你的机器，其中一些特定的安全手段在各种浏览器中都有采用。例如，Mozilla 有个完全独特的安全模型，涉及到了签署脚本和加强特权。如果理解了哪些安全手段对所有的浏览器都适用，哪些是特定于浏览器的，你就能创建出更加安全的 JavaScript 脚本。

19.1.1 同源策略

在本书前面简单介绍过，JavaScript 只能与同一个域中的页面进行通讯。例如，运行在 Wrox 主页 (www.wrox.com) 上的脚本不能与任何包含 Mozilla 网站上页面 (www.mozilla.org) 的浏览器窗口或框架进行交互。这种安全手段叫做同源策略 (Same Origin Policy)。

两个脚本被认为是同源的条件是，包含它们的页面满足以下条件：

- 协议相同（比如都是 `http://`）；
- 端口相同（通常为 80 端口）；
- 域名相同。

如果这三个条件中有任何一条不满足，就不允许两个脚本进行交互。例如，运行在 www.wrox.com 上的脚本，不能访问 p2p.wrox.com 上的页面，因为两者的域名不同（尽管从技术上讲 p2p.wrox.com 是 www.wrox.com 的子域）。这个脚本也不能访问 www.wrox.com:8080

上的页面，因为端口不同，也不能访问 `about:blank`，因为协议不同（不是 `http://`）。

1. 对BOM和DOM编程的影响

这些规则也同样影响与 BOM 和 DOM 之间的交互。例如，不可以访问不同来源的任何页面的 `document` 对象，也就是说不能访问其中任何 DOM 结构。下面的两行代码解释了这个问题：

```
alert(frames[1].location.href);
alert(frames[1].document.location.href); //fails
```

前面的代码应该会出现两个警告框，都用来显示第二个框架页（索引为 1）的 URL。你应该还记得，`window` 和 `document` 对象都有一个 `location` 对象的特性。如果在与框架页不同源的页面中运行这两行代码，则代码的第二行会发生错误，因为脚本不能访问 `document.location` 对象或它的任何特性。不过，脚本可以访问 `window.location` 对象（由 `frames[1].location` 表示），也可以访问所有其他的窗口的特性。

你可能还记得本书前面说过的 XML HTTP Request 对象（所有浏览器中）以及 Web 服务功能都只能与同一个域下的资源交互；这又是一个受同源策略影响的例子。它对插件也起作用。

2. 规则特例

一般的逻辑认为，`www.wrox.com` 和 `p2p.wrox.com` 是属于同一个域的，所以它们应该可以相互通讯。是的，浏览器开发人员也认同这个看法并提供了允许这种通讯的方法。

只要在每个子域的页面中加入一行脚本就可以达到这个目的。设置 `document.domain` 特性，如下：

```
document.domain = "wrox.com";
```

这么简单的一行代码即可以消除 JavaScript 通讯的安全阻碍。不过，请注意，只能将 `domain` 设置为已经存在于 URL 中的一个值，所以 `www.wrox.com` 中的页面不能将 `domain` 设置为 `mozilla.org`，因为这与同源策略相违背。

19.1.2 窗口对象问题

还有很多方法用来保护终端用户，防止恶意脚本尝试使用窗口。

首先，也是最重要的，窗口不能在屏幕外打开，也不能小于 100×100 。如果指定了在屏幕外的坐标，窗口会自动放到屏幕中的离指定位置最近的地方，同时要留出足够的空间以能看到完整的窗口。同样的，如果尝试打开小于 100×100 的窗口，窗口会自动扩大到 100×100 。尽管这两条规则看上去没有实际意义，但是这保证用户总是能看到由脚本弹出的窗口。

同样，不能打开超过用户桌面大小的窗口。例如，不能在 1024×768 的桌面上打开 1600×1200 的窗口。

564

Windows XP Service Pack 2 上的 IE 的行为与上面所说的有些细微的差别。如果你在浏览可信的（在 Internet Options 中设置的）网站上，则可以打开偏离屏幕或者很小尺寸的窗口。可信网站（Trusted Sites）一般使用 `https://` 协议，但不一定必须这样。对于任何非信任网

站，如果窗口尝试打开窗口，你会看到安全警告。如果你允许其弹出，则前面所说的限制还存在（不能在屏幕外打开，不能用过小的尺寸打开）。

同样的窗口位置和尺寸规则对已创建的窗口也适用。不能用 `moveBy()` 或者 `moveTo()` 将弹出的窗口移到屏幕外，也不能用 `resizeBy()` 或者 `resizeTo()` 将窗口尺寸改成小于 100×100 或者超过桌面的大小。

某些浏览器（如 Mozilla）允许终端用户决定脚本是否可以移动窗口或者改变窗口大小。

还有，不能用 `close()` 将不是使用 `window.open()` 打开的窗口关闭。如果尝试打开不是由脚本打开的窗口，会出现询问用户是否同意关闭窗口的对话框。

大部分浏览器现在都已内置了弹出式窗口阻拦工具。很多非技术用户可能并不清楚他们是否已打开这个功能——在开发中必须记住这一点。

一般来说，弹出式窗口阻拦工具会将所有并非因与用户交互而出现的弹出式窗口阻拦，也就是说，不能在 `load` 和 `unload` 之类的事件中打开新的窗口；弹出窗口的操作只能放在 `click` 和 `keypress` 之类的事件中。然而，某些弹出式窗口阻拦工具不管有没有用户交互都会阻止弹出。那么如何判断弹出的窗口已被阻止呢？

`window.open()` 函数一般会返回新创建的窗口的引用。如果窗口被屏蔽了，`window.open()` 会返回 `null`：

```
var oWindow = window.open("page.htm", "mywindow");

if (oWindow == null) {
    alert("Your popup blocker won't allow you access to this window.");
} else {
    //continue on
}
```

565

当窗口被屏蔽时，前面的代码会为用户显示一个信息。一般来说，最好在打开窗口时确保这个方法返回的不是 `null`。

这个方法对于 Windows XP Service Pack 2 的 IE 弹出式窗口阻拦工具、Mozilla 的弹出式窗口阻拦工具和 Google 工具条的弹出式窗口阻拦工具都可以正常工作。对于其他的一些工具，最好在 `window.open()` 调用周围加上 `try...catch` 块（因为有些其他的弹出式窗口阻拦工具会造成 JavaScript 错误，而不会返回 `null`）。

19.1.3 Mozilla 特有的问题

作为 Netscape Communicator 代码的一部分，Mozilla 引入了一些新的安全机制以确保身份验证和嵌在网页内的脚本的安全性。前者涉及到了激活加强特权。

1. 特权

不同的安全相关的能力被安排在不同的特权中。要使用这些特权函数，必须先用 `netscape.security.PrivilegeManager.enablePrivilege()` 向用户请求许可。

Mozilla 为开发人员提供了指导对特权的合理使用的一份文档，名为 `JavaScript Security: Signed Scripts` (<http://www.mozilla.org/projects/security/components/signed-scripts.html>)。在这份文档中，Jesse Ruderman 列出了以下的在 Mozilla 中可用的特权：

特 权	描 述
<code>UniversalBrowserRead</code>	使浏览器忽略同源策略，可以读取当前域之外的资源
<code>UniversalBrowserWrite</code>	使浏览器忽略同源策略，可以写入当前域之外的资源
<code>UniversalXPConnect</code>	允许使用 XPConnect 对浏览器 API 进行访问
<code>UniversalPreferencesRead</code>	允许使用 <code>navigator.preferences</code> 读取用户设定
<code>UniversalPreferencesWrite</code>	允许使用 <code>navigator.preferences</code> 进行用户设定
<code>CapabilityPreferencesAccess</code>	允许读取/设置管理安全性的设定。要读取其中任何一个设定，同样需要 <code>UniversalBrowserRead</code> ；要进行设置，还需要 <code>UniversalBrowserWrite</code>
<code>UniversalFileRead</code>	允许使用 <code>file://</code> 协议打开浏览器窗口

要启用特权，必须将其中一个值传给 `enablePrivilege()` 方法：

```
netscape.security.PrivilegeManager.enablePrivilege("UniversalBrowserRead");
```

566

你可能知道这行代码出自第 17 章。这个特权为完成 Web 服务调用是必要的。`UniversalBrowserRead` 特权也允许访问浏览器历史记录中的 URL，如下：

```
netscape.security.PrivilegeManager.enablePrivilege("UniversalBrowserRead");
for (var i=0; i < history.length; i++){
    alert(history[i]);
}
```

这段代码输出浏览器历史记录中（存储在 `history` 对象中）的每个页面的 URL。

一旦完成特权动作的使用，最好禁止这个特权以确保没有其他恶意脚本可以篡夺这个特权：

```
netscape.security.PrivilegeManager.enablePrivilege("UniversalBrowserRead");
for (var i=0; i < history.length; i++){
    alert(history[i]);
}

netscape.security.PrivilegeManager.disablePrivilege("UniversalBrowserRead");
```

`UniversalBrowserWrite` 特权也许是最有趣的，因为它可以使你避免受到本章前面提到的窗口限制。启用这个特权后，可以：

- 将窗口大小设置为 100×100 以下，或者大于用户桌面的尺寸。
- 将窗口移到屏幕之外。
- 创建没有标题的窗口。
- 用 `close()` 关闭窗口，不管窗口是如何打开的。

如果请求了特权但未被允许，则 JavaScript 控制台中会显示消息——"User did not grant privilege"。

2. 签署脚本

为使用扩展的特权，必须签署脚本。JavaScript 可以与 Java applet 一样以同样的方式进行签署，以允许访问这些特权。签署脚本涉及到从安全机构获取数字证书，用于确认脚本的位置、发布者及用途。载入已签署的脚本后，浏览器会显示消息，询问是否允许签署的脚本访问扩展的特权。

Mozilla 基金会提供 SignTool 来帮助签署脚本。这个小工具将脚本及它的数字证书打包成一个 JAR 文件。为使用签署的脚本，必须用下面格式的 URL 访问其中的 HTML 页面：

```
jar:http://www.yourdomain.com/signedscripts.jar!/page.htm
```

脚本被正确签署并用正确的 URL 访问后，用户只需接受数字证书，而不需要再要求特权了，
567 因为已经启用了。

如果没有正确签署，脚本就不会运行。JavaScript 控制台会显示与用户不允许扩展特权时一样的消息 ("User did not grant privilege")。要获取更多关于 Mozilla 中签署脚本的信息，请查看 <http://www.mozilla.org/projects/security/components/signed-scripts.html>。

在 IE 或其他的浏览器中是不能签署脚本的。IE 用超文本应用程序(HyperText Application, HTA) 来提供一些高级的安全性。更多关于使用 HTA 的信息，请查看 <http://msdn.microsoft.com/workshop/author/hta/overview/htaoverview.asp>。

3. 代码库原则

另一种访问扩展特权的方法是启用代码库原则 (Codebase Principal)。这个策略根据脚本载入的位置（它的代码库），来判断指定脚本的安全性。它的逻辑是，如果脚本是从与 HTML 一样的服务器上载入的，则这个脚本是安全的。但这并非精确的安全假设，这也是 Mozilla 浏览器没有启用代码库的原因。启用这个选项还是有点复杂的，所以一般普通用户不会错误地打开代码库原则。

代码库策略只在测试和调试最后会被签署的脚本时使用。

19.1.4 资源限制

如果你是软件开发人员或者软件用户，你可能很关心程序是否会超过内存限制，使得机器越来越慢、不稳定甚至崩溃。其实是由浏览器公司来决定 JavaScript 如何运行才不会影响用户的机器。因此，浏览器在任何单个调用中最多运行一百万行 JavaScript 代码。

一百万行听起来好像很多（也许已多于你所需的了），但还是会遇到这种超出限制的情况的。发生这种情况时，浏览器会以某种方式提示你：IE 会弹出一个对话框，告诉你脚本使浏览器运行变慢了，是否继续运行脚本；Mozilla 则直接跳出当前操作，并在 JavaScript 控制台中输出一条消息。

这个一百万行限制指的并不是总合，所以不用担心用户点击页面不同部分时脚本的运行。这个限制只存在单个函数调用中，以防止诸如死循环和无限递归的情况发生。

19.2 国际化

如果你打算创建可令世界上各个地方的人都能看到的网站，或者可安装在世界任一角落的 Web 应用，国际化就是一个问题。在很多种编程语言中，都有一些库用于帮助实现软件的国际化，从典型的 C++ 应用程序到基于 Web 的系统。一些公司会花费数百小时的时间来针对国际化的目的对他们的网站和 Web 应用进行检验，但他们常常忘记检查 JavaScript 代码。

568

19.2.1 使用 JavaScript 检测语言

在第 5 章中介绍了 `navigator` 对象以及它的特性。其中没有进行详细讨论的一个特性是 `language` 特性，它返回浏览器的语言和国家（地区）代码（例如，“en-us”代表美国英语）：

```
var sLang = navigator.language; //won't work in IE
```

Mozilla、Opera 和 Safari/Konqueror 都支持这个特性，但 IE 不支持。

IE 用三个特性来替代这一个特性：`browserLanguage`（表示浏览器所用的语言），`userLanguage`（基本等同于 `browserLanguage`）和 `systemLanguage`（表示操作系统的语言）。`userLanguage` 特性和 `language` 本质上是一样的，所以对前面一行代码添加简单的内容即可检测任何浏览器的语言：

```
var sLang = navigator.language || navigator.browserLanguage;
```

用这段代码，就可以判断正在浏览页面的用户的浏览器是否有不支持的语言设置，并采取合适动作，比如将访客重定向到另一个合适的页面：

```
if (sLang.toLowerCase() == "fr") {
    document.location.replace("index_fr.htm");
}
```

这段代码检查，如果语言是法语（由 `fr` 表示），就重定向到另一个页面。

你可能已注意到，这段代码对语言字符串使用了 `toLowerCase()`。这是必要的，因为不同浏览器之间的大小写是不完全一致的。比如，有些将美国英语报告为“`en-us`”，有些则报告为“`en-US`”。

19.2.2 策略

在国际化 JavaScript 代码中，最重要的一步是避免硬编码的字符串。例如，不要这么做：

```
alert("The date you entered is incorrect.");
```

在此例中，字符串 "The date you entered is incorrect." 就是硬编码的。数值被硬编码后，如果不能修改使用它的代码，值就不发生变化。与下面的例子进行比较：

```
var sIncorrectDateMessage = "The date you entered is incorrect.";

//more code here

alert(sIncorrectDateMessage);
```

569

这个例子将消息字符串放到 `sIncorrectDateMessage` 变量中。其他的国际化字符串也应该与这个变量放在一起，这样即可在同一个地方修改任何字符串了。

处理国际化字符串的最佳做法是，将字符串放入不同的 JavaScript 代码中（类似于 JSP 应用使用属性文件的方式）。你支持的每种语言都应该有个单独的 JavaScript 文件。例如，假设要支持三种语言：英语（语言代码为 `en`）、德语（`de`）和法语（`fr`）。每种语言都应该有个自己的包含所有网站或者 Web 应用所需的字符串的 JavaScript 文件。最简单的做法是，为每个文件起一个包含语言代码的文件名。例如，以下的文件名会使正确地选择文件十分简单：

- `Strings_en.js`;
- `Strings_de.js`;
- `Strings_fr.js`。

然后，用一些服务器端逻辑来确保包含了正确的文件。在 PHP 中，可以这样做：

```
$supported = array("en", "de", "fr");

if (in_array($lang, $supported)) {
    $filename = "Strings_$lang.js";
} else {
    $filename = "Strings_en.js";
}

<script type="text/javascript"
src="scripts/<?php echo $filename ?>"></script>
```

这段代码假设，`$lang` 变量包含了要使用的语言，然后在受支持的语言的数组 (`$supported`) 中寻找与其匹配的项。如果支持这个语言，就载入相应的 JavaScript 文件。这确保为给定语言使用了正确的 JavaScript 字符串值。同时，如果遇到不支持的语言，则设置为默认的语言。

19.2.3 字符串的思考

ECMAScript 第一版引入了对 Unicode 字符（其字符数量有 65000 个，而 ASCII 则为 128 个）的支持，有效地保证 ECMAScript 可以处理任意一种语言的字符串，包括常常会遇到问题的双字节字符。

1. Unicode到底是什么？

根据 Unicode 官方网站介绍：“Unicode 为每个字符都提供了唯一的数值，不管是什平台、什么程序或是什么语言。”

开发 Unicode 是为了给处理世界上存在的所有字符提供统一的编码。在 Unicode 之前，每种语言都有自己的编码，也就是说，同样的代码在不同的语言中可能表示不同的字符。比如英语中的字母 A 的代码在另一个语言中可能代表另一个字母（当然，这也不一定）。

Unicode 将字符表示为一个 16 位数字，可容纳超过 65,000 个字符，这使其成为实现国际化的理想解决方案。另外，前 128 个 Unicode 字符其实就是原来的 128 个 ASCII 字符，这也使兼容老的英语应用更加简单。

2. 在JavaScript中的表示形式

所有的 Unicode 字符，包括 ASCII，在 Unicode 中都表示为四位十六进制数值，前面加上一个 \u 以表示这是一个 Unicode 字符。例如，\u0045 是 E 的 Unicode 形式（使用 ASCII 的方式表示为 \x45）。

这种字符表示方法可以用在 JavaScript 的注释及字符串中，就像使用特殊字符 \n 一样例如：

```
alert("\u0048\u0045\u004C\u004C\u004F \u0057\u004F\u0052\u004C\u0044");
```

这一行的运行结果是？它会弹出包含文字“HELLO WORLD”的警告框。使用 Unicode 字符集，就可以创建任何语言的消息了。尽管这种消息的纯文本形式对人来说并不是可读的，但它是唯一可以处理其他语言的多字节字符的方法。

3. 浏览器和操作系统支持

JavaScript 可以显示和理解 Unicode 字符并不能代表操作系统也可以。你可能会问，为什么只需关注浏览器支持的 Web 开发者，还需要关心这个问题？这是因为 JavaScript 使用了一些操作系统的功能来完成这个任务，尽管大部分开发人员从未发觉这一点。对于国际化来说，必须小心这个重要的界限。

使用 alert()、confirm() 或者 prompt() 时，实际上是在使用操作系统的对话框。除非客户端操作系统已安装了多语言支持，否则最后看到的将是内容乱七八糟的对话框。大部分情况下，浏览器可以正确地反映出操作系统，然而，却无法判断个人究竟对浏览器进行了怎样的设置。

对操作系统对话框使用国际化支持时，要当心这些问题。处理分布式 Web 应用时，最好能为用户提示这个限制；在公用网站上，最好完全避免使用这些对话框。

4. 防止错误的字符串

在国际化的 Web 页面中，开发人员常常会用这样的技术从服务器端变量直接将字符串传递给 JavaScript 变量：

```
<% String sJspHello = "Hello"; %>
<!-- more code here -->
<script type="text/javascript">
    var sJavaScriptHello = "<%= sJspHello %>";
```

570

571

```
    alert(sJavaScriptHello);
</script>
```

这个例子使用了 JSP，目的是将字符串 "Hello" 输出到 JavaScript 变量中。浏览器载入这个页面后，你可以从源代码中看到：

```
<script type="text/javascript">
    var sJavaScriptHello = "Hello";
    alert(sJavaScriptHello);
</script>
```

这个输出貌似正确，JavaScript 函数也是所期望的。但是考虑另一个例子：

```
<% String sJspHeSaidHi = "He said, \"hi.\""; %>

<!-- more code here -->

<script type="text/javascript">
    var sJavaScriptHeSaidHi = "<%= sJspHeSaidHi %>";
    alert(sJavaScriptHeSaidHi);
</script>
```

现在输出到浏览器中的结果变成：

```
<script type="text/javascript">
    var sJavaScriptHeSaidHi = "He said, "hi."'";
    alert(sJavaScriptHeSaidHi);
</script>
```

看出问题了吗？现在从 JSP 中输出的字符串中包含了引号，这在 JavaScript 中会导致错误。这是用 JavaScript 输出字符串的国际化网页中经常会出现的错误。如果字符串要被输入到 JavaScript 代码中，你必须当心包含其中的引号。处理该问题的最佳方法是，在输出到 JavaScript 之前，替换字符串中的引号，如下：

```
<% String sJspHeSaidHi = "He said, \"hi.\""; %>

<!-- more code here -->

<script type="text/javascript">
    var sJavaScriptHeSaidHi = "<%= sJspHeSaidHi.replaceAll(\"\\\\\"", "\\\\"") %>";
    alert(sJavaScriptHeSaidHi);
</script>
```

这个例子用 Java 的 `replaceAll()` 将所有的双引号替换成反斜杠加双引号。第一个参数是正则表达式的字符串表示形式（你应该还记得，正则表达式字符串必须进行双重转移，所以 \" 要变成 \\\"）；第二个参数虽然看起来一样，但它不是正则表达式而是普通的字符串。这就可以将下面的字符串：

572 "He said, \"hi.\""

变成

"He said, \\\\"hi.\\\\""

输出到 JavaScript 后，则会得到合法的字符串：

```
<script type="text/javascript">
    var sJavaScriptHeSaidHi = "He said, \"hi.\"";
    alert(sJavaScriptHeSaidHi);
</script>
```

这段代码的语法正确，运行也没有错误。

5. 使用双引号

另一种常见的错误是，用单引号表示字符串而不使用双引号。大家都知道，JavaScript 允许两种表示字符串的方式，所以下面两行是相等的：

```
sHello = "Hello";
sHello = 'Hello';
```

虽然 JavaScript 允许使用两种语法，但这并不表示可以随意地轮番使用，尤其是在要进行国际化时。实际上，因为单引号在现在的语言中（尤其是像法语之类的语言）单引号比双引号用的更多，所以你会遇到与我们在前面在双引号上遇到的相同的问题，甚至更频繁。因此，只使用双引号来表示字符串被认为是最佳的。

遵循这一节中的一些原则可以确保你的国际化的 JavaScript 代码能很好的运行。

19.3 优化 JavaScript

创建桌面应用时，大多数开发人员并不需要太关心优化。在极大程度上，编译时都会进行优化：所有的变量、函数、对象等等，都被替换成只有处理器才能理解的符号指针。

宏因为在编译时就已被替换，所以比函数调用更快。模板可用来提高对象创建的速度。但是 JavaScript 在这方面则很令人头疼，因为它是作为源代码下载，然后由浏览器对其进行解释的（不是进行编译）。因此，JavaScript 代码的速度被分割成两部分：下载时间和执行速度。

19.3.1 下载时间

使用 Java 或者其他类似的编程语言时，开发人员无需考虑就可以用 100 个字符长的变量名，因为编译后名称都会被替换掉；他们也不用担心写上长篇大论的注释，因为这些在编译时都会被删除。但作为 JavaScript 开发人员，就没这么爽了。

Web 浏览器下载的是 JavaScript 源代码，也就是说，所有的长变量名和注释都会包含在内。这个因素及其他因素会增加下载时间，这样就会增加脚本运行的总体时间。增加下载时间的关键因素是脚本所包含的字节数。

要记住的一个关键数字是 1160，这是能放入单个 TCP-IP 包中的字节数。最好能将每个 JavaScript 文件都保持在 1160 字节以下以获取最优的下载时间。

在 JavaScript 文件中，每个字符都是一个字节。因此，每个额外的字符（不管是变量名、函数名，或者注释）都会影响下载速度。部署任何 JavaScript 代码之前，都应该尽可能地优化下载速度。下面是一些可以用来减少脚本大小的方法。

1. 删除注释

虽然这似乎是废话，但是很多开发人员都会忘记，因为，以前这个事情都是由编译器来完成的。

脚本中的任何注释都应该在部署之前删除。进行开发时，注释十分重要，它可以帮助小组成员来理解代码。但是，要部署时，注释会明显使 JavaScript 代码体积变大。

不过，有时法律上需要将版权说明（或一些其他这类的注释）留在文件里。如果这样，则确保其他的注释都被删除了，同时尽量使这个法律声明简短。

删除注释是缩减 JavaScript 文件大小最方便的途径。即使不使用下面的建议，仅这一条就可以对缩减文件尺寸产生极大的效果了。

2. 删除制表符和空格

大部分优秀的程序员都会有规则地缩紧代码以增加其可阅读性。这样很好，但是浏览器不需要这些额外的制表符和空格；所以最好删掉它们。也不要忘记函数参数、赋值语句以及比较操作之间的空格：

```
function doSomething ( arg1, arg2, arg3 ) { alert(arg1 + arg2 + arg3); }

function doSomething(arg1,arg2,arg3){alert(arg1+arg2+arg3);}
```

对于 JavaScript 解释程序来说，这两行完全一样，虽然第一行比第二行多了 12 个字节。删除参数、括号和其他语言分隔符之间的空格也可以有效地减少文件的整体大小，这样就缩短了下载时间。

574 删除额外的制表符和空格时，在每行的结束使用分号有助于保持代码的语法含义。

3. 删除所有的换行

接下来，就要删除所有的换行符来减少脚本尺寸。只要你在程序的每行的结尾都正确地添加了分号，就不需要任何换行符了。

有许多关于在 JavaScript 中换行是否应该存在的思考，但底线都是换行要增加代码的可读性。将其删除也是一种最简便的使代码可以对抗反向工程的方法。

如果出于某种原因而不要删除换行，则要保证文件是 Unix 格式的，而非 Windows 格式的。Windows 用两个字符表示换行（回车和新行，ASCII 代码分别为 13 和 10）；Unix 仅使用一个。所以，将换行法从 Windows 格式转换成 Unix 格式也可以节约一些字节数。

4. 替换变量名

这是实现起来最无聊的一种优化方法。替换变量名通常不是手工完成的，因为这个过程并非简单地文本查找、替换操作。

基本的思想是所有变量名（或者对象的私有特性）都应被替换成无意义的变量名。毕竟变量的名称对解释程序来说毫无意义，只是对阅读代码的开发人员来说有意义。不过，部署字符串时，应该将描述性的变量名替换成更简单、更短的名称：

```
function doSomething(sName,sAge,sCity){alert(sName+sAge+sCity);}

function doSomething(a1,a2,a3){alert(a1+a2+a3);}
```

上面第一行代码是原先的；第二行将参数名称替换后的。这样，就减少了 16 个字节。想象一下，如果脚本中的变量名都被替换成一两个字符长，会节约多少长度啊！

要自己进行变量名替换时，你要极其小心。尤其不推荐使用文本编辑器一般的查找替换方法，因为编辑器并不区分变量名和其他匹配给定模式的文本。例如，你有个称为 `on` 的变量（也许是表示某个值是否有效的布尔值）。如果你尝试将 `on` 替换成别的文本，有可能将 `function` 结尾的 `on` 也给替换了，这样整个代码就完全错误了。

5. ECMAScript Cruncher

自己遵循前面的四步可能会很困难。因此，可以用外部程序来帮助你执行这些步骤。

575

为 JavaScript 进行文件最小化和变量名替换的最佳工具之一是 Thomas Loo 开发的 ECMAScript Cruncher (ESC，可在 <http://www.saltstorm.net/depo/esc/> 下载)。ESC 是一个小巧的 Windows Shell 脚本，可以替你完成本节中前面提到的优化操作。

要运行 ESC，必须使用 Windows 系统。打开一个控制台窗口，然后使用以下格式的命令：

```
cscript ESC.wsf -l [0-4] -ow outputFile.js inputFile1.js [inputFile2.js]
```

第一部分，`cscript`，是 Windows Shell 脚本解释程序。文件名 `ESC.wsf` 是 ESC 的程序本身。然后是压缩等级，一个 0 到 4 的数值，表示要进行的优化的等级。`-ow` 选项表示下一个参数是优化后输出的文件名。最后，剩下的参数是要进行优化的 JavaScript 文件。可以只给出一个要进行优化的文件，也可以有多个文件（多个文件在优化后会按顺序放到输出文件中）。

ESC 支持的四个优化等级如下：

等 级	描 述
0	不改变脚本。要将多个文件合到单个文件中时有用
1	删除所有的注释
2	除等级 1 外，再删除额外的制表符和空格
3	除等级 2 外，再删除换行
4	除等级 3 外，再进行变量名替换

ESC 擅长把变量名替换成无意义的名称。它还具有一定的智能，进行私有对象特性和方法名

的替换（由名称前后加上两个下划线表示），所以私有特性和方法依然保持其私有性。

ESC 不会更改构造函数名称、公用特性和公用方法名称，所以无需担心。使用 ESC 时要记住，如果某个 JavaScript 引用了另一个文件中的构造函数（比如本书前面讲到的 `StringBuffer` 类），4 级优化会把对构造函数的引用替换成无意义的名称。解决方法是将两个文件合成一个文件，这样就会保持构造函数的名称。

6. 其他减少字节数的方法

下面是一些不常用的优化脚本文件大小的手段，不过依然可以节省很多字节数。ESC 中没有处理这些建议，所以你必须手工进行。

- 替换布尔值

前面学过，对于比较来说，`true` 等于 1，`false` 等于 0。因此，脚本包含的字面量 `true` 都可以用 1 来替换，而 `false` 则可以用 0 来替换。`true` 节省了 3 个字节，`false` 则节省了 4 个字节，但是并没有改变布尔表达式的含义。
576

考虑这个例子

```
var bFound = false;

for (var i=0; i < aTest.length && !bFound; i++) {
    if (aTest[i] == vTest) {
        bFound = true;
    }
}
```

可以替换成：

```
var bFound = 0;

for (var i=0; i < aTest.length && !bFound; i++) {
    if (aTest[i] == vTest) {
        bFound = 1;
    }
}
```

这两段代码运行方式完全相同，而后者节省了 7 个字节。

- 缩短否定检测

代码中常常会出现检测某个值是否有效的语句。而大部分否定测试所做的就是判断某个变量是否为 `undefined`、`null` 或者 `false`，如下：

```
if (oTest != undefined) {
    //do something
}

if (oTest != null) {
    //do something
}

if (oTest != false) {
```

```
//do something
}
```

虽然这些都正确，但用逻辑非操作符来重写也有同样的效果：

```
if (!oTest) {
    //do something
}
```

为什么可以这样呢？回想一下本书开头关于自动类型转换的讨论。逻辑非操作符在操作数为 `undefined`、`null` 或者 `false` 时返回 `true`（操作数为 0 也返回 `true`）。这样的替换也可以节省很多字节。

577

7. 使用数组和对象字面量

在本书前面已经简单接触过数组字面量的概念了。现在回顾一下，以下两行是相同的：

```
var aTest = new Array;
var aTest = [];
```

第二行用了数组字面量，与第一行效果一样。很明显，第二行要短很多。创建数组时，使用数组字面量是个好选择。

类似的，对象字面量也可用于节省空间。以下两行是相等的，但是使用对象字面量的更加简短：

```
var oTest = new Object;
var oTest = {};
```

如果要创建具有一些特性的一般对象，也可以使用字面量，如下：

```
var oFruit = new Object;
oFruit.color = "red";
oFruit.name = "apple";
```

前面的代码可以用对象字面量来改写成这样：

```
var oFruit = { color: "red", name: "apple" };
```

这个例子用一个对象字面量来分配两个特性 `color` 和 `name`，这样，又节省了很多字节。

19.3.2 执行时间

JavaScript 整体性能的第二部分是运行脚本所需的时间。JavaScript 是一种解释性的语言，它执行的速度要大大慢于编译型语言。

Syracuse 大学的 Geoffrey Fox 写了一本在线课程——*JavaScript Performance Issues*（可以在 <http://www.npac.syr.edu/users/gcf/forcps616javascript/msrcobjectsapril99/tsld022.htm> 浏览），其中描述了 JavaScript 与其他知名语言相比较的性能。根据他的文章，JavaScript

- 比编译型的 C 程序慢 5000 倍；
- 比解释型的 Java 慢 100 倍；
- 比解释型的 Perl 慢 10 倍。

不过，我们还是可以做一些简单的事情来提高 JavaScript 代码的性能的。

578

1. 关注范围

在 JavaScript 中，范围很重要。范围可被认为是一个变量存在的空间。浏览器的 JavaScript 的默认的范围（全局）是 window。在 window 范围中创建的变量只在页面从浏览器中卸载后才会销毁。而你定义的每个函数都是在全局范围下的另一个范围中。在函数中创建的所有变量都只存在于函数范围内，函数执行完毕时就销毁。

可以认为 JavaScript 的范围是一个树状层次。引用变量时，JavaScript 解释程序先在最近的范围内查找其是否存在。如果没有，就到上一层次中查找，如此进行，直到 window 范围。如果在 window 范围内也没有发现变量，则在执行期间内就会出现错误。

每次解释程序到另一个范围内搜索变量，都会影响到执行速度。本地范围内的变量比全局变量执行起来更快。解释程序在树中要走的距离越短，脚本运行得越快。但这意味着什么呢？考虑以下例子：

```
var sMyFirstName = "Nicholas";

function fn1() {
    alert(sMyFirstName);
}

function fn2() {
    var sMyLastName = "Zakas";
    fn1();
}

function fn3() {
    var sMyMiddleInitial = "C";
    fn2();
}

fn3();
```

调用最后一行的 fn3() 时，创建了如下的范围树（见图 19-1）。

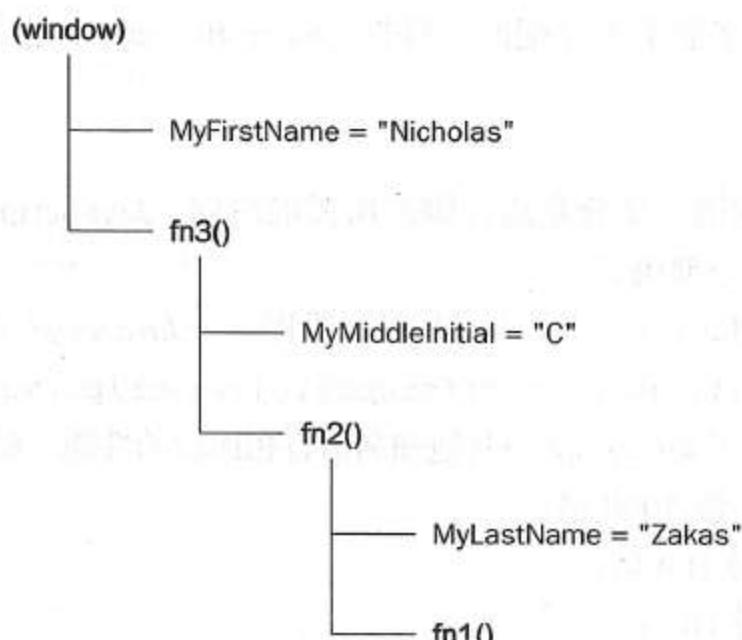


图 19-1

在这个例子中，理解范围的重要性十分明显。函数 fn3() 调用 fn2()，而 fn2() 调用了 fn1()。函数 fn1() 又访问了 window 级别的变量 sMyFirstName，为定位这个变量，解释程序必须不断向上查找整个范围树，直到 window 范围。这花费了很多时间。所以从 JavaScript 的范围内找到一个用来优化执行速度的方法，是很必要的。可以用一些简单的方法来帮助解释程序快速定位变量。

● 使用局部变量

在函数中，总是用 var 语句来定义变量。无论何时使用 var，都会在当前的范围内创建一个局部变量。如果直接使用变量而不事先用 var 进行声明，则变量会被创建在 window 范围内，也就是说每次使用这个变量，解释程序都要搜索整个范围树。例如，不要这样：

```
function sayFirstName() {
    sMyFirstName = "Nicholas";
    alert(sMyFirstName);
}
```

在这个函数中，对变量 sMyFirstName 的赋值没有使用 var；变量就被创建在了 window 范围内。可用下面的例子来证明：

```
function sayFirstName() {
    sMyFirstName = "Nicholas";
    alert(sMyFirstName);
}

function sayFirstNameToo() {
    alert(sMyFirstName);
}

sayFirstName();
sayFirstNameToo();
```

在这个例子中，你会看到两个显示“Nicholas”警告框。因为第一个函数调用 sayFirstName()，将变量 sMyFirstName 创建在 window 范围内了，这样，第二个函数 sayFirstNameToo() 也可以访问它。假设把这个例子用 var 进行修改，如下：

```
function sayFirstName() {
    var sMyFirstName = "Nicholas";
    alert(sMyFirstName);
}

function sayFirstNameToo() {
    alert(sMyFirstName);
}

sayFirstName();
sayFirstNameToo();
```

尝试运行这段代码，第一个警告框显示过后，会发生错误，因为第二个函数无法找到名为 sMyFirstName 的变量。这个变量是在 sayFirstName() 范围内创建的，函数完成执行后就被销毁了。

使用局部变量可带来更快的执行速度，因为解释程序无需因为搜索变量而离开当前范围。局部变量的效率也会更高，因为它们不会一直占用内存。

- 避免with语句

你可能已经明白：范围越少，速度越快。这也就是不使用 with 语句的原因。

with 语句会让你访问对象的特性就像访问变量一样，所以以下代码：

```
alert(document.title);
alert(document.body.tagName);
alert(document.location);
```

可以替换成：

```
with (document) {
    alert(title);
    alert(body.tagName);
    alert(location);
}
```

这确实消除了 with 语句中的三行代码对 document 对象的需要，从而也缩减了一些长度。但对于 with 语句的出现，要付出额外的代价：它是另一个范围。使用 with 语句时，要强制解释程序不仅在范围树内查找局部变量，还强制检测每个变量及指定的对象，看其是否为特性。因为，也可以在函数中定义变量 title 或者 location。

581 最好避免使用 with 语句。减少的代码长度并不能弥补损失的性能。

2. 计算机科学基础

很多其他编程语言中与代码优化相关的基本方法也可应用于 JavaScript。因为 JavaScript 的语法和语句很大程度上受到了 C、Java 和 Perl 的影响，所以这些语言中的技巧也可用于 JavaScript。本节中展示的技巧在很多书籍和文章中都有阐述，比如 Koushik Ghosh 的 *Writing Efficient C and C Code Optimization*（可在 http://www.codeproject.com/cpp/C__Code_Optimization.asp 上看到）。在这篇文章中，Ghosh 总结了一系列流行的 C 语言的代码优化技术，其中很多都可用于 JavaScript。

- 选择正确的算法

编程时，选择正确的算法也十分重要。算法越简单，代码运行越快。为衡量算法的复杂度，计算机科学中使用了大 O 记号。大 O 记号由定义为函数的字母 O 和特定的参数组成。

最简单的算法是常数值，表示为 O(1)。获取常数值是极快的操作过程。常数值包括真值常量，数值常量——数字 5 和保存在变量中的数值。考虑以下例子：

```
var iFive = 5;
var iSum = 10 + iFive;
alert(iSum);
```

这段代码获取了三个常数值。前两个是数字 10 和第二行的 iFive 变量。然后，在第三行上获取 iSum 的值；这也是常数值。这三个常数值的获取都只会花费极少的时间，因为它们非常简单。

存储在数组中的数值也是常数值，所以以下代码只用了常数值算法：

```
var aNumbers = [5, 10];
var iSum = aNumbers[0] + aNumbers[1];
alert(iSum);
```

在这段代码中，数组 `aNumbers` 用于存储要相加的数字。`aNumbers[0]` 和 `aNumbers[1]` 都进行常数值的获取，所以在这段代码执行了 3 次 $O(1)$ 算法。

下一个算法是线性的，表示为 $O(n)$ 。这个算法常用于简单的搜索，例如：遍历数组，直到找到结果。下面是线性算法的例子：

```
for (var i=0; i < aNumbers.length; i++) {
    if (aNumbers[i] == 5) {
        alert("Found 5");
        break;
    }
}
```

582

这种算法很常见，因为在数组中迭代是十分常见的技术。然而，还有一种线性算法在 JavaScript 也经常用：命名特性查找。

命名特性也就是对象的基本特性，如 `oDog.name`。尽管看上去它是位于给定对象中的变量。实际上，它是搜索对象中的所有特性以找到匹配名称的一个特性。因此，最好用局部变量或者数字索引的数组值来替代命名特性。

考虑这个例子：

```
var aValues = [1, 2, 3, 4, 5, 6, 7, 8];

function testFunc() {
    for (var i=0; i < aValues.length; i++) {
        alert(i + "/" + aValues.length + "=" + aValues[i]);
    }
}
```

下面是这段代码中用到的算法：

- 在 `i < aValues.length` 中的 `i`——常数($O(1)$)；
- 在 `i < aValues.length` 中的 `aValues.length`——线性 ($O(n)$)；
- `alert(i + "/" + aValues.length + "=" + aValues[i]);` 中的 `i`——常数 ($O(1)$)；
- `alert(i + "/" + aValues.length + "=" + aValues[i]);` 中的 `aValues.length`——线性 ($O(n)$)；
- `alert(i + "/" + aValues.length + "=" + aValues[i]);` 中的 `aValues[i]`——常数 ($O(1)$)。

这里的线性算法特别有意思，因为它们都可以由常数算法来替代。每次运行循环，所有的线性算法都运行两次：一次在 `i < aValues.length` 测试中，每次循环迭代都会执行，另一次在 `alert()` 调用中。也就是说，在整个函数执行期执行了 16 次线性算法。

下面代码运行的结果与前面的完全一样，但只用了一次线性算法：

```
var aValues = [1, 2, 3, 4, 5, 6, 7, 8];
```

```

function testFunc() {
    for (var i=0, iCount=aValues.length; i < iCount; i++) {
        alert(i + "/" + iCount + "=" + aValues[i]);
    }
}

```

在这个例子中，在`for`循环中初始化的第二个变量`iCount`，它被赋予`aValues.length`的值。然后，每次循环迭代后的测试变成了常数算法，因为它访问变量`i`和`iCount`，而不是`aValues.length`。同样，将`alert()`调用中的`aValues.length`替换成`iCount`，也消除了一个线性算法。最后，这段代码就仅执行一次线性算法，与原先的16次相比，大大减少了执行时间。

进行代码优化时，记住这些算法。下面是使用正确算法的基本规则：

- 只要有可能，就用局部变量或者数字索引的数组来替代命名特性。
- 如果命名特性要多次使用，就先将它的值存储在局部变量中，以避免多次使用线性算法请求命名特性的值。

关于大O记号及算法方面的更多信息，请参考Wikipedia上的Big O Notation, http://en.wikipedia.org/wiki/Big_O_notation。

● 反转循环

在大部分编程语言中，循环都要进行大量的处理，所以保持循环的高效性可极大地减少执行时间。一种知名的方式是按照反向的顺序进行循环迭代。下面是个普通的`for`循环：

```

for (var i=0; i < aValues.length; i++) {
    //do something here
}

```

如果要进行反转，应该将迭代子（`i`）从数组的最后一项（位置是`aValues.length - 1`）开始，然后逐渐减小`i`，而不是增加`i`，同时检查`i`是否大于等于0：

```

for (var i=aValues.length-1; i >= 0; i--) {
    //do something here
}

```

反转循环有助于降低算法的复杂度。它用常数(0)作为循环的控制语句以减小执行时间。

● 翻转循环

用`do..while`循环来替代`while`循环以进一步减少执行时间。假设有如下`while`循环：

```

var i=0;

while (i < aValues.length) {
    //do something here

    i++;
}

```

可用`do..while`循环重写上面的代码且不改变行为：

```

var i=0;

```

```

do {
    //do something here
    i++;
} while (i < aValues.length);

```

这段代码比用 while 循环的代码速度更快，因为它用循环反转来进行进一步地优化：

```

var i=aValues.length-1;

do {
    //do something here
    i--;
} while (i >= 0);

```

也可以将自减操作直接放入控制语句中，以减少额外的语句。

```

var i=aValues.length-1;

do {
    //do something here
} while (--i >= 0);

```

这个循环已被完全优化了。

● 展开循环

可以将循环展开，一次执行多个语句。考虑这个 for 循环的例子：

```

var aValues = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20];
var iSum = 0;

for (var i=0; i < aValues.length; i++) {
    iSum += aValues[i];
}

```

循环体执行了 20 次，每次都是对变量 iSum 进行增量操作。这是很简单的操作，但还可以展开循环，在 for 循环内进行多次操作：

```

var aValues = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20];
var iSum = 0;

for (var i=0; i < aValues.length; i++) {
    iSum += aValues[i];
    i++;
    iSum += aValues[i];
    i++;
}

```

在这个例子中，在循环体内做了五次增量。每次增量后，都对变量 i 加 1，所以它对数组的遍历与原来的 for 循环所做的一样。这样，控制语句就只能执行四次，减少了执行时间。

当然，把变量增量运算和加法结合在一起也可以进一步优化循环：

```
var aValues = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20];
var iSum = 0;

for (var i=0; i < aValues.length; i++) {
    iSum += aValues[i++];
    iSum += aValues[i++];
    iSum += aValues[i++];
    iSum += aValues[i++];
    iSum += aValues[i++];
}
```

这段代码又减少了五个语句，进一步减少了执行时间。是否有通用的展开循环的方法？答案是肯定的。用装置 Duff（根据它的创始人，Tom Duff 而命名）的技术，就可以无需了解需要进行多少次迭代而展开循环。

在前面的例子中，循环执行了 20 次，每次都处理数组中的一个项目。因此，展开后的循环必须包含十个、五个、四个或者两个（20 的因子）语句才能正常工作，Duff 装置用取模操作符来执行包含 8 个同样语句的展开循环。这个技术也可用于迭代任意次。

Duff 装置最初是用 C 写成的，由 Jeff Greenburg 移植到 JavaScript 中，他对 JavaScript 优化做了详尽的测试，并整理在其网站 http://home.earthlink.net/~kendrasg/info/js_opt/ 上。算法如下：

```
var iLoopCount = Math.ceil(iIterations / 8);
var iTestValue = iIterations % 8;

do {
    switch (iTestValue) {
        case 0: [execute statement];
        case 7: [execute statement];
        case 6: [execute statement];
        case 5: [execute statement];
        case 4: [execute statement];
        case 3: [execute statement];
        case 2: [execute statement];
        case 1: [execute statement];
    }
    iTestValue = 0;
} while (--iLoopCount > 0);
```

变量 iIterations 包含了循环要进行的迭代次数。变量 iLoopCount 包含了展开后 do..while 循环要执行的次数。iTestValue 变量表示第一次循环中 switch 语句中要执行 case 的数量，以后的循环都是从 case 0 开始的（注意，在 case 语句后没有停止 switch 执行的 break 语句）。如果对前面的例子应用 Duff 装置，结果如下：

```

var aValues = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20];

var iIterations = aValues.length;
var iLoopCount = Math.ceil(iIterations / 8);
var iTestValue = iIterations % 8;
var iSum = 0;
var i = 0;

do {

    switch (iTestValue) {
        case 0: iSum+= aValues[i++];
        case 7: iSum+= aValues[i++];
        case 6: iSum+= aValues[i++];
        case 5: iSum+= aValues[i++];
        case 4: iSum+= aValues[i++];
        case 3: iSum+= aValues[i++];
        case 2: iSum+= aValues[i++];
        case 1: iSum+= aValues[i++];
    }

    iTestValue = 0;

} while (--iLoopCount > 0);

```

注意，`iIteration` 被设为数组中的值的数量。`iSum` 变量用来存储数组中的数字的总和；而 `i` 变量是在数组中移动的迭代子（与 `for` 以及 `do..while` 循环中的一样）。

执行循环前，`iLoopCount` 等于 3，也就是说循环要执行 3 次。`iTestValue` 的值等于 4，所以循环进入第一个循环时，它会跳到 `switch` 语句中的 `case 4`，执行完其后的 4 行语句，`iTestValue` 被设置为 0。此后，每次执行循环，都从 `case 0` 开始执行。

Greenburg 进一步优化了 Duff 装置的 JavaScript 版本，将单个 `do..while` 循环分成了两个单独的循环。算法如下：

```

var iLoopCount = iIterations % 8;
while (iLoopCount--) {
    [execute statement]
}
iLoopCount = Math.floor(iIterations / 8);
while (iLoopCount--) {
    [execute statement]
    [execute statement]
}

```

587

这个算法的主要目的是，先在第一个循环中处理必须完成的额外迭代（即不能被 8 整除而剩下的迭代）。然后继续进入第二个循环，迭代剩下的 8 的倍数的循环。对前面的例子应用这个算

法，可以得到以下代码：

```

var iIterations = aValues.length;
var iLoopCount = iIterations % 8;
var iSum = 0;
var i = 0;

while (iLoopCount--) {
    iSum += aValues[i++];
}

iLoopCount = Math.floor(iIterations / 8);

while (iLoopCount--) {
    iSum += aValues[i++];
    iSum += aValues[i++];
}

```

这次又添加了一些变量，不过与原先的 JavaScript 的 Duff 装置的效果相同。

当然，这个算法还可通过将两个 `while` 循环翻转成 `do..while` 循环来进一步优化。不过，这未包含在 Greenburg 的算法中。

是否使用诸如 Duff 装置之类的算法，完全由你决定。你必须在代码加长（下载时间变长）
588 和速度优化之间进行权衡。

● 优化 `if` 语句

用 `if` 语句和多个 `else` 语句时，一定要把最有可能的情况放在第一个，然后是第二可能出
现的情况，如此排列。例如，如果预计某个数值一般在 0 到 10 之间，但是还需要为其他数值执
行指定操作，则可以这样安排代码：

```

if (iNum > 0 && iNum < 10) {
    alert("Between 0 and 10");
} else if (iNum > 9 && iNum < 20) {
    alert("Between 10 and 20");
} else if (iNum > 19 && iNum < 30) {
    alert("Between 20 and 30");
} else {
    alert("Less than or equal to 0 or greater than or equal to 30");
}

```

总是将最常见的情况放在第一个，这样就减少了要进行多次测试后才能遇到正确条件的情况。
也要尽量减少 `else if` 语句的数量，并且将条件按照二叉搜索树的方式进行排列。例如，
前面例子可重写为：

```

if (iNum > 0) {
    if (iNum < 10) {
        alert("Between 0 and 10");
    } else {
        if (iNum < 20) {
            alert("Between 10 and 20");
        } else {
            if (iNum < 30) {
                alert("Between 20 and 30");
            } else {
                alert("Greater than or equal to 30");
            }
        }
    }
} else {
    alert("Less than or equal to 0");
}

```

这看上去很复杂，但是它已考虑了很多代码潜在的如何执行的情况，所以执行得更快。因此，当测试数字的范围时，先在 `if` 条件中测试大于、小于或等于的情况，然后使用一个 `else` 语句来处理其他的可能情况。

● switch和If

使用 `switch` 还是 `if` 语句的这个经典问题也存在于 JavaScript 中。一般来说，超过两种情况时，最好使用 `switch` 语句。常用 `switch` 语句来替代 `if` 语句，最高可令执行快 10 倍。在 JavaScript 中就更加可以从中获益，因为 `case` 语句中可以使用任何类型的值。589

3. JavaScript陷阱

你应该已注意到了，JavaScript 在很多方面都与其他的编程语言不同。因此，最好记住它的一些容易出错的地方。

● 避免字符串连接

在本书前面介绍过用加号（+）操作符进行字符串连接的危险性。为避开这个问题，当时创建了 `StringBuffer` 对象，其中封装了用 `Array` 和 `join()` 方法进行连接的逻辑。

一旦一次要进行多个字符串的连接（比如，大于五个），最好使用 `StringBuffer` 对象。

● 优先使用内置方法

只要可能，就应该优先使用 JavaScript 对象的内置方法。因为内置方法是用 C++ 或者 C 之类 的语言编译的，运行起来要比必须实时编译的 JavaScript 快得多。例如，你可能会自己写一个用来计算某个数的阶乘的函数：

```

function power(iNum, n) {
    var iResult = iNum;

    for (var i=1; i < n; i++) {
        iResult *= iNum;
    }

    return iResult;
}

```

尽管这个函数结果完全正确，但是 JavaScript 已经提供了计算数字阶乘的方法——`Math.pow()`：

```
alert(Math.pow(3, 4)); //raise 3 to the 4th power
```

最大的区别就是 `Math.pow()` 是浏览器的一部分，是由 C++ 编译的，要比自己写的 `power()` 函数快很多很多。

对于这类常见操作，JavaScript 已有相当丰富的内置方法。所以，只要可以，最好使用内置操作，而不是自己定义的函数。

- 存储常用的值

当多次用到同一个值时，可先将其存储在局部变量中以便快速访问。尤其对通常使用对象的特性来进行访问的值更加重要。考虑以下代码：

590

```
oDiv1.style.left = document.body.clientWidth;
oDiv2.style.left = document.body.clientWidth;
```

在这个例子中，`document.body.clientWidth` 的值用到了两次，但它是用命名特性（比起访问单个变量，这是十分昂贵的操作）来获取的。可以用局部变量来重写这段代码：

```
var iClientWidth = document.body.clientWidth;
oDiv1.style.left = iClientWidth;
oDiv2.style.left = iClientWidth;
```

注意，这个例子同时也降低了前面代码的算法复杂度，将两个线性操作变成一个。

4. 最小化语句数量

我们有理由相信，脚本中的语句越少，执行所需的时间越短。很多种方法可用于将 JavaScript 代码中的语句数量减到最少，比如定义变量时，处理迭代数字时，使用数组和对象字面量时等等。

- 定义多个变量

本书前面提到，可用 `var` 语句一次定义多个变量。也可用同一个 `var` 语句定义不同类型的变量。例如，下面的代码块用四个单独的 `var` 语句来分别定义四个变量：

```
var iFive = 5;
var sColor = "blue";
var aValues = [1, 2, 3];
var oDate = new Date();
```

这四个语句可用一个 `var` 语句来替代：

```
var iFive = 5, sColor = "blue", aValues = [1, 2, 3], oDate = new Date();
```

消除了三个语句，这段代码运行也更快了。

- 插入迭代子

使用迭代子（在不同位置上加减的值）时，尽可能合并语句。考虑下面的代码段：

```
var sName = aValues[i];
i++;
```

前面两行语句其实只有一个目的：第一行语句从 `aValues` 中获取一个值，然后将其保存在变量 `sName` 中，第二行语句对 `i` 进行迭代。这两行语句可以合成一条语句：

```
var sName = aValues[i++];
```

这条语句完成了前面两条语句所完成的事情。因为其中增量操作是后置的，这个语句的其他部分执行完毕后才会执行 i 加一。出现类型情况时，就要将迭代值的操作插到最后使用它的语句中。

591

- 使用数组和对象字面量

在减少代码长度中，我提到过可以用字面量来替换数组和对象的定义，同时它们也可以减少运行时间。考虑以下对象定义过程：

```
var oFruit = new Object;
oFruit.color = "red";
oFruit.name = "apple";
```

与前面提过的一样，上面的代码可用对象字面量改写成：

```
var oFruit = { color: "red", name: "apple" };
```

除了可以减少字节数外，由于对象字面量表示法仅占用一条语句，而展开的语法用了 3 条语句，一条语句总是比三条语句执行得快。用数组字面量替代展开的数组语法也可达到同样的效果。

5. 节约使用DOM

DOM 处理，算是 JavaScript 最消耗时的操作之一。不管是添加、删除或者其他对页面的 DOM 内部结构的更改，都会导致明显的时间损耗。因为每次 DOM 操作都要更改页面对用户的表现，也就是说，必须对整个页面进行重新计算以保证正确地进行了渲染。解决这个问题的方法是尽可能对不在 DOM 文档中的元素进行 DOM 操作。

考虑以下为无序列表添加 10 个条目的例子：

```
var oUL = document.getElementById("ulItems");

for (var i=0; i < 10; i++) {
    var oLI = document.createElement("li");
    oUL.appendChild(oLI);
    oLI.appendChild(document.createTextNode("Item " + i));
}
```

在执行速度方面，这段代码有两大问题。首先是循环中的 oUL.appendChild() 的调用。每次执行过后，整个页面都要进行重新计算并重新渲染；第二个问题是，接下来的一行给列表添加了文本节点，这也会造成页面的重新计算。因此每次运行循环都会造成两次页面的重新计算，总计 20 次。

要修复这个问题，列表的项目应该在添加好文本节点后再添加。另外，可用文档碎片来保存所有创建的列表项，最后将其一次性添加到列表中：

```
var oUL = document.getElementById("ulItems");
var oFragment = document.createDocumentFragment();

for (var i=0; i < 10; i++) {
    var oLI = document.createElement("li");
    oLI.appendChild(document.createTextNode("Item " + i));
```

592

```

    oFragment.appendChild(oLI);
}

oUL.appendChild(oFragment);

```

在重写后的代码中，在循环开始之前创建了文档碎片。然后，在循环中创建列表项并添加了文本节点，在循环中的最后一步是将列表项添加到文档碎片中。循环执行完毕后，用 `appendChild()` 方法和文档碎片，一次性将所有的列表项添加到列表中（添加的是文档碎片中的子节点，而并非碎片本身）。

处理 DOM 文档时，记住这个技巧。如果要进行多次改动，最好在直接应用到文档上之前，用文档碎片来保存改动。

19.4 知识产权的问题

对文件尺寸和脚本速度进行完全的改进后，还必须考虑如何保护你的知识产权。这是传统程序员不用特别关心的问题，因为递交给客户的最终产品是已编译过的，对反向工程来说是比较安全的。但是，递交 JavaScript 代码其实就直接给出源代码，使其公开。尽管版权提示和其他一些法律声明可以在一定程度上提供法律上的保护，但是它不能保证代码的安全。那么开发人员要怎么做呢？

19.4.1 混淆

混淆是将源代码变得难以阅读，以对付喜欢窥视代码的人。前面说过的 ESC，通过将变量名和函数名替换成无意义的名称，在一定程度上进行了少量混淆。

Dithered JavaScript 压缩工具 (<http://www.dithered.com/javascript/compression/index.html>) 用独有的方法提供了更多的混淆：它将 JavaScript 代码中的字符序列抽取出来，用特别的记号进行替换。执行代码时，这些记号会用正则表达式进行替换，然后执行代码。例如，考虑前面一节中的 DOM 代码：

```

var oUL = document.getElementById("ulItems");
var oFragment = document.createDocumentFragment();

for (var i=0; i < 10; i++) {
    var oLI = document.createElement("li");
    oLI.appendChild(document.createTextNode("Item " + i));
    oFragment.appendChild(oLI);
}

oUL.appendChild(oFragment);

```

593

用 Dithered JavaScript 压缩工具进行压缩，脚本就会变成：

```

S='`4UL = docu`5ById(\"ulItems\");
`4`2`6Docu`5`2();
for (var i=0; i < 10; i++) {
`4LI`6Ele`5(`\"li\"`3LI`1`0eTextNode(`\"Item \" + i)`3`2`1oLI);

```

```

}oUL`1o`2);";for(I=6;I>=0;)S=S.replace(eval('`'+I+'`/g'),("document.create~.appendCh
ild(~Fragment~);
o~var o~ment~ = `0e~".split('~'))[I--]);eval(S);

```

虽然看上去没有变小，但是要记住原先的脚本并未删除所有的制表符、空格和换行符（建议在使用这个工具之前，先用其他工具对代码进行尺寸上的优化）。

这类混淆的缺点是它的启动速度很慢，因为需要额外地对代码进行解释。不过，可以极大地减小文件尺寸。

19.4.2 Microsoft Script Encoder (仅 IE)

如果你确定目标用户只使用 Windows 上的 IE，可利用 Microsoft Script Encoder 来对代码进行保护。Microsoft Script Encoder 是命令行工具，可以将 JavaScript 编码成完全无法阅读的代码。

首先，从微软的网站上下载这个工具 (<http://www.microsoft.com/downloads/details.aspx?FamilyId=E7877F67-C447-4873-B1B0-21F0626A6329&displaylang=en>)。安装之后，打开 DOS 命令窗口，并使用以下的语法：

```
screnc inputfile outfile
```

这个程序可以以多种不同的文件类型作为输入，但是我们最关注的是编码 HTML 文件和外部 JavaScript 文件 (.js)。指定输入文件为 HTML 文件时，包含在<script/>元素中的代码会被编码；对于.js 文件，则编码整个文件。

例如，前一节中的代码会编码成这段文字：

```

#@~^RAEAAA==-
mD~KjdP',NK^Es+UYconOAV+snxDAX&[cJ!V&Yn:dE*i@#@&7CD,Wo.mo:nUDPxP9G1Eh xDRmM+mO+GW^E
s+UOwDlTh+ Y`*I@#@#@#&6W.Pc-
mD~k{Ti,k~@!,F!I~b_Q#,`@#@&~,P,\lMPKJq,'~NKm;h xYc^D 1Y 3s+s+
YcJsrr#I@#@&~P,PGJ&R122 x[Z4r^Nc9W1E: xD
mM+CY KnaD1W9n`rqY h~J,_,kb#I@#@&P~P,GsM1Lh xY Cawnx9/4ks9`KSq*i@#@&8@#@#&KjJ
mww UN;tk^ [cWwDmoh+UO*i@#@&n18AAA==^#~@

```

如果指定 HTML 文件作为输入，<script/>元素会被更新，language 特性变成 "JScriptEncoded"；如果指定.js 文件作为输入，则必须手工在引用文件的地方添加这个 language 特性：

```
<script language="JScript.Encoded" src="encodedfile.js"></script>
```

594

即使某人下载了代码，目前对于做反向工程的人来说也很难获取实际的代码。所有同一个 HTML 页面中的 JavaScript 都可以使用同样的调用方法（也就是说，对 JavaScript 调用进行编码不会影响它们的运行）。

这个技术的缺点是，它只能在 Windows 上的 IE 中使用。代码被编码后，对于其他操作系统上的浏览器来说都是没有用的。如果你能确定用户只用 Windows 平台上的 IE（比如企业局域网），可能会很有用。否则，最好使用其他能在所有浏览器中使用的混淆工具。

可以在 <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/script56/html/seusingscriptencoder.asp> 上找到更多关于 Microsoft Script Debugger 的信息。

19.5 小结

在本章中，学习了一些与 JavaScript 部署（部署到公用网站或者 Web 应用）有关的问题。

首先讨论的问题是安全性。不同的安全性问题，从一般概念（如同源策略）到特定于浏览器的问题（如 Mozilla 的签署脚本），都一一进行了讨论。JavaScript 的安全性限制也涉及了 BOM 和 DOM。

下面，学习了如何优化 JavaScript 代码。两种优化 JavaScript 代码的方法是最小化代码的尺寸（删除额外的空白字符和注释）和减少脚本运行的时间（一些常用的编程技巧）。讨论了用于进行优化的不同的方法，还介绍了一些工具以协助进行优化。

最后讨论的主题是知识产权的问题。介绍了代码混淆和编码这两种方式，用来防止进行反向工程。还学习了混淆对跨浏览器的兼容性比用 Script Encoder 进行编码更好，后者只能用于 Windows 上的 IE。最后，你所使用的保护知识产权的方法很大程度上由开发的解决方案决定。



至此，你已经了解了 JavaScript 的起源以及目前使用中的各种实现。这一章将谈论 JavaScript 未来的发展。自从最初诞生以及 ECMAScript 标准的提出以来，JavaScript 及其派生语言已经在很多不同的编程环境中有了大量的应用。但是 JavaScript 还未达到其顶峰：它依然在成长，在发展，一些感兴趣的团体（比如微软和 Mozilla）也在不断地推动它的前进。在本章中，你将了解 JavaScript 的发展方向以及其对你的代码产生的意义。

20.1 ECMAScript 4

JavaScript 的未来不可避免地与 ECMAScript 4 联系在一起，自首次提案开始这一标准已经面对了各种潜在的问题。

第 39 技术委员会（TC39），ECMA 内部最初标准化 ECMAScript 的小组，现在仍在负责开发未来的版本。与 ECMAScript 1一样，ECMAScript 4 也是由 Netscape 公司提出的，TC39 原本的计划是在 2002 年将其发布。然而，在这个转折点浮现出一个重大的问题，自 ECMAScript 1 被标准化开始，微软就雄霸了 Web 的天下。微软向 TC39 提交了它自己对 ECMAScript 未来方向的提案。

TC39 把 ECMAScript 4 的发布时间改到了 2004 年第一季度，相比原本制定的计划推迟了近两年。然而，过了 2004 年 3 月仍然没有发布。到我写完这章时，还没有任何迹象显示下一个版本的 ECMAScript 究竟何时发布。所有的日程表记录写的还是 2004 年第一季度。

ECMA 未能给出一个明确的方向，唯一能了解 JavaScript 未来方向的办法就是查看提交给 TC39 的草案。

20.1.1 Netscape 的提案

Netscape 将它的提案提交给 TC39 时，Netscape 公司的未来可谓一片黯淡。之后，Netscape 被时代华纳公司（后来变成 AOL 时代华纳）收购，然后立刻被解散，而它的浏览器的未来就留给了开源的 Mozilla 基金会。Mozilla 基金会依然支撑着 Netscape 的提案（可在 <http://www.mozilla.org/js/language/es4/> 上查看）并对其进行定期更新，不过最后的更新是在 2003 年 6 月。

Netscape 的 ECMAScript 的草案是将 ECMAScript 变成 Java 语言的轻量级版本。草案中列出的一些目标如下：

- 使该语言适合书写模块化和面向对象的应用；
- 使其可以很容易地写出健壮、安全的应用；
- 增强 ECMAScript 与其他不同语言和环境交互的能力；
- 增强 ECMAScript 对性能优先的应用的适合性；
- 尽可能简化该语言；
- 保持语言实现的兼容和灵活性。

虽然这些目标看上去雄心勃勃，但是这份草案被很快指责背离了初衷——它的作者并未将 ECMAScript 看作是 C++ 或 Java 的替代品，他们也不想使其往这个方向发展。

为能完成前面列出的这些目标，Netscape 的草案提议了一些对语言的第 3 版的修改建议。这些修改包括：

- 可选的严格类型定义和值得类型检测。在本书中讨论过，ECMAScript 目前是弱类型的语言。这个提案想更改它以减少错误，并且可以使 ECMAScript 与其他面向对象语言保持一致。
- 更合理的类语法。在 ECMAScript 目前的版本中，定义全局函数和定义类之间没有严格的区分。这个草案建议用保留字 `class` 来使得定义类的语法更加合理（从而也就更像 Java）。这包括用 `extends` 保留字来实现直接的继承，以及引入 `private` 和 `protected` 作用域。
- 增加更多类型。这个草案的一个主要目的是为更方便地与其他语言进行交互，所以建议 ECMAScript 4 包含更多的类型，比如整型和长整型，其他语言都支持这些类型。

1. 关键字和保留字

Netscape 提议以下的关键字：

as	break	case	catch	class	const	continue
default	delete	do	else	export	extends	false
finally	for	function	if	import	in	instanceof
is	namespace	new	null	package	private	public
return	super	switch	this	throw	true	try
typeof	use	var	void	while	with	

你可能已发现，其中的一些关键字在 ECMAScript 3 中是保留字，这也正是它们被保留的原因。

另外，该提案还要求保留以下词：

abstract	debugger	enum	goto	implements	interface
native	protected	synchronized	throws	transient	volatile

可以看到，这个保留字的列表清晰地表达了 Netscape 的目的，要将 ECMAScript 向类 Java 的语法发展。

下面的单词在 ECMAScript 3 中是保留的，现在用作 Netscape 提案的语言的一部分：

boolean	byte	char	double	final	float	int	long	short	static
---------	------	------	--------	-------	-------	-----	------	-------	--------

最后，单词 `get` 和 `set` 也被赋予了特殊的含义，因此也不能用作变量名。

为与未来的 ECMAScript 实现兼容，应该避免将这些关键字和保留字作为函数或变量的名称。

2. 变量

根据 Netscape 的提案，变量可由明确的类型定义，在被定义的变量后面加上冒号和类型名称，像这样：

```
var color : String = "red";
```

前面的代码中，名为 `color` 的变量被创建为 `String` 类型。在这个草案中，定义类型是可选的。也可以用于自己定义的类：

```
var specialObject : MyClass = new MyClass();
```

在变量上还有一个小小的改变是，可以创建真正的无法改变的常量，用 `const` 关键字：

```
const age = 32;
```

这段代码把变量 `age` 定义为常量，并为其分配一个值 32。

3. 函数

函数也受益于新的类型声明，可以为函数的参数和返回值提供限定的类型。考虑以下例子：

```
function sum(num1 : Integer, num2 : Integer) : Integer {
    return num1 + num2;
}
```

599

这里，函数 `sum()` 有两个 `Integer` 型的参数，并返回一个 `Integer` 型的值（由右括号后面的冒号指定）。如果函数不返回任何值，可定义为 `Void` 型：

```
function doNothing(num1 : Integer, num2 : Integer) : Void {
    num1 + num2;
}
```

尽管也可以不出现类型，让函数的行为与今天的 ECMAScript 一样，但是定义了参数和返回值类型的函数是进行类型检测的。

Netscape 的提案允许出现函数的重载，方式类似于 Java 中的重载：只需参数列表不同的函数。例如，可以定义两个不同版本的 `sum()` 函数，一个有两个参数，另一个有三个：

```
function sum(num1 : Integer, num2 : Integer) : Integer {
    return num1 + num2;
}

function sum(num1 : Integer, num2 : Integer, num3 : Integer) : Integer {
    return num1 + num2 + num3;
}
```

另外一个对函数的改变是，可以定义命名参数并且它是可选的，而且只要使用了参数的名字它可以以任何顺序出现。例如，以下函数包含两个命名参数：

```
function sum(num1 : Integer, num2: Integer, named num3 : Integer = 7, named num4 : Integer = 10) {
    return num1 + num2 + num3 + num4;
}
```

要调用这个函数，可以只使用两个参数，或者三个，也可以四个都出现：

```
result = sum(10, 20);
result = sum(10, 20, num3: 10);
result = sum(10, 20, num4: 10, num3: 20);
```

注意最后两行中用到了命名参数，尤其注意最后一行，num4 出现在 num3 之前。

4. 数值字面量

为了支持更多类型的数字，Netscape 的提案中引入了三种新的数字类型：long、unsigned long(ulong)和 float。可以在数值前加上字母来指定数字的类型：l（或 L）表示长整型数值，ul（或者 uL、U1、UL）表示无符号长整型，以及 f（或者 F）表示浮点数值。例如，L25 是长整型，UL25 是无符号长整型，以及 F25 是浮点数。
600

5. 类型

Netscape 的草案还引入一些新的类型：

- Never——无值；
- Void——用于替代 ECMAScript 3 中的 Undefined 类型；
- Integer——表示整型数值；
- char——所有单个 16 位 Unicode 字符；
- Type——所有类型的超类型。

与 ECMAScript 3 不同，所有的类型都被认为是对象而非简单数值（primitive value）。

新的机器类型（machine type）代表简单数值：

- sbyte——有符号字节整型，-128~127；
- byte——字节类型，0~255；
- short——短整型，-32768~32767；
- ushort——无符号短整型，0~65535；
- int——整型，-2147483648~2143483647；
- uint——无符号整型，0~4294967295；
- long——长整型，-9223372036854775808~9223372036854775807；
- ulong——无符号长整型，0~18446744073709551615；
- float——浮点数，所有单精度 IEEE 浮点数字。

只要需要表示一个数字，即可以将机器类型当作一般类型用。例如：

```
var b : byte = 10;
var longnum : long = L100;
```

6. 类

在 ECMAScript 4（Netscape 提案）中的类的形式如下：

```

class MyClass {
    private var color : String = "red";

    public function MyClass(color : String) {
        this.color = color;
    }

    public function sayColor() : Void {
        alert(this.color);
    }
}

```

601

可以看到，`class` 关键字用于定义 `MyClass`。它有一个私有变量 `color`、一个构造函数及一个公用方法 `sayColor()`。在 ECMAScript 4 中，`public` 和 `private` 都被认为是命名空间，命名空间指定变量和方法可以访问的上下文。按照这种方式定义的类与 ECMAScript 3 中的实例化方式是一样的，通过 `new` 关键字来实现：

```
var myObject : MyClass = new MyClass();
```

除了变量和方法，还可以定义特性的获取函数（getter）和设置函数（setter）。当读取变量所包含的值时，调用获取函数，同时当给指定变量设置新值时，调用设置函数。

```

class MyClass {
    private var color : String = "red";

    public function get myColor () {
        return this.color;
    }

    public function set myColor (value : String) : Void {
        this.color = value;
    }
}

```

前面的代码创建了特性 `myColor`，可以像普通的特性（如 `color`）一样使用，但是它的行为是通过获取函数和设置函数定义的。例如：

```
var myObject : MyClass = new MyClass();
myObject.myColor = "blue";
```

一般来说，当你想在改变特性值后进行某些其他操作时，即可使用它（比如，在 `myColor` 特性改变时更改 UI 元素的颜色）。

也可以重定义操作符与对象交互时的行为。在 ECMAScript 3 中，所有的操作符都使用对象的 `valueOf()` 或者 `toString()` 方法。但是，在这里，你可以更精确地定义对特定类的实例进行加号或者减号操作时应该如何进行。

```

class MyClass {
    private var value : Integer = 25;

    public function MyClass(value : Integer) {
        this.value = value;
    }
}

```

```

        }

    public function getValue() : Integer {
        return this.value;
    }

    operator function "+" (anotherObject : MyClass) : MyClass {
        return new MyClass(this.value + anotherObject.getValue());
    }
}

```

这个类定义了私有特性 value，可用类构造函数对其进行初始化。最后一个函数定义了在对 MyClass 对象使用加号时该执行的动作。这个特殊的函数定义了这个行为：当第二个操作数也是 MyClass 对象时，将两个对象的 value 相加，并返回一个新的 MyClass 对象。有了这个定义，就可以使用以下代码：

```

var obj1 = new MyClass(20);
var obj2 = new MyClass(30);
var obj3 = obj1 + obj2;
alert(obj3.getValue()); //outputs "50"

```

最后，通过使用 static 关键字，类中还可以包含真正的静态特性和方法：

```

class MyClass {
    public static var value : Integer = 25;
}

```

你也肯定已经猜到了某个类的静态成员的访问方式：

```
var value : Integer = MyClass.value;
```

7. 继承

ECMAScript 4 中的继承与 Java 一样，使用 extends 关键字。例如：

```

class ClassA {
    private var value : Integer = 0;

    public function ClassA(value : Integer) {
        this.value = value;
    }
    public function getValue() : Integer {
        return this.value;
    }
}

class ClassB extends ClassA {

    public function ClassB(value : Integer) {
        super(value);
    }

    public function sayValue() : Integer {

```

```

        alert(this.getValue());
    }

}

```

首先，注意 `extends` 关键字用于创建继承于 `ClassA` 的 `ClassB`。这比 ECMAScript 原来建立继承关系的方式要更加简单、直观（为了向后兼容仍然可以使用对象伪装和原型链）。

再要注意的事情是：`super()` 方法是用来在 `ClassB` 中调用 `ClassA` 的构造函数的。这又与 Java 一样。

603

20.1.2 实现

尽管 ECMAScript 4 还没有官方发布的标准，但这并不影响微软和 Mozilla 部分地实现它。

1. Mozilla

Mozilla 选择在 Web 浏览器中实现一些简单功能。例如，可以在 Mozilla 中使用 `const` 关键字：

```
const message = "Hello World!";
```

在其他浏览器中，这一行代码会产生错误，因为 `const` 在 ECMAScript 3 中是保留字。然而，在 Mozilla 中，这一行是定义了常量字符串 `message`，不能改变它的值。

Mozilla 还实现了对象特性的获取函数和设置函数，但是语法不同。你可能在本书前面的章节中见过设置函数和获取函数，在这里有个简单的例子：

```

function MyClass() {
    this.__color__ = "red";
}

MyClass.prototype.color getter = function () {
    return this.__color__;
};

MyClass.prototype.color setter = function (value) {
    this.__color__ = value;
    alert("Color changed to " + value);
};

var obj = new MyClass();
alert(obj.color);      //outputs "red"
obj.color = "blue";    //outputs "Color change to blue"

```

这里还有另一种语法（这样就不会在非 Mozilla 的浏览器中致使语法不通过）：

```

function MyClass() {
    this.__color__ = "red";
}

MyClass.prototype.__defineGetter__("color", function () {
    ...
});

```

```

        return this.__color__;
    });

 MyClass.prototype.__defineSetter__("color", function (value) {
    this.__color__ = value;
    alert("Color changed to " + value);
});

var obj = new MyClass();
alert(obj.color);      //outputs "red"
obj.color = "blue";    //outputs "Color change to blue"

```

04 注意，尽管这个例子在语法上是正确的，但它在非 Mozilla 的其他浏览器中还是会产错误，因为两个函数都没有定义 (`defineGetter()` 和 `defineSetter()`)。

2. 微软

从 IE 5.5 发布开始，微软就没有更新过它基于浏览器的 JavaScript 实现。它所做的就是在.NET Framework 中包含了语言 JScript.NET。针对各种目的，JScript.NET 作为 ECMAScript 4 的实现，被添加了一些特定于微软的功能。然而，它不能作为客户端脚本语言在 IE 中使用，只能在 ASP.NET 中作为服务器端语言或者独立应用程序（确实是这样，JScript.NET 可以被编译）来用。

与其他的 JavaScript 版本不同，JScript.NET 是编译型语言，可以变成独立的可执行文件。它可以被编译成与 Visual Basic.NET 和 C#一样的.NET 机器代码，并可以用一样的公共语言运行库（Common Language Runtime, CLR）来执行。讨论 JScript.NET 的潜力和能力超出了本书的范围，你可以在微软的网站上找到更多的信息 <http://msdn.microsoft.com/library/en-us/dnclinic/html/scripting07142000.asp?frame=true>。

20.2 ECMAScript for XML

2002 年，由 BEA 公司领导的多家公司联合为 ECMAScript 起草了一个扩展，以为该语言添加最直接的 XML 支持。那时，XML 正逐渐受到青睐，而这些公司想使 ECMAScript 跟上一次技术浪潮。2004 年 6 月份，ECMAScript for XML (E4X) 作为 ECMA-357 标准发布。E4X 并非一种语言，而是对 ECMAScript 语言的一个可选的扩展。同样，E4X 引入了处理 XML 的新的语法，以及针对 XML 的对象。

20.2.1 途径

E4X 并不是简单地实现目前的 XML 标准，诸如 SAX、XPath 以及 DOM，相反，它展示了一种独特的创建和处理 XML 文档的方式。E4X 的方法源自多种技术，如 DOM、XPath 和 XSLT，尽管没有完全与其中任何一种相对应。

例如，假设要处理下面的 XML 节点：

```
<employees>
    <employee position="Software Engineer">
        <name>Nicholas C. Zakas</name>
```

```

</employee>
<employee position="Salesperson">
    <name>Jim Smith</name>
</employee>
</employees>

```

这段 XML 可通过以下代码赋给一个 ECMAScript 变量：

```

var oXml = <employees>
    <employee position="Software Engineer">
        <name>Nicholas C. Zakas</name>
    </employee>
    <employee position="Salesperson">
        <name>Jim Smith</name>
    </employee>
</employees>;

```

605

然后，在 XML 中的数据就可以用很有逻辑、很容易理解的方法进行引用了。在 E4X 中，所有的 XML 元素都完全被理解成对象（基于 XML 类，将在后面进行讨论）。这样，要获取第一个雇员的引用，代码如下：

```
var oFirstEmployee = oXml.employees.employee;
```

第一个雇员的名字可用下面的代码获取：

```
var sName = oXml.employees.employee.name;
```

第一个雇员的位置可用@符号来获取（它在 XPath 中也用来指示 XML 特性）：

```
var sPosition = oXml.employees.employee.@position;
```

要获取指定的雇员，使用与数组一样的方括号标记。下面的代码将返回第二个雇员：

```
var oSecondEmployee = oXml.employees.employee[1];
```

可用双点记号 (...) 来选择子孙节点。例如，要立刻找到第一个<name/>元素，可以使用以下代码：

```
var oFirstEmployeeName = oXml..name;
```

也可以用类似 XPath 的表达式来返回指定的对象。假设要获取第一个销售人的名字。代码如下：

```
var oFirstSalesperson = oXml..employee.(@position="salesperson").name;
```

除 XML 对象外，所有的值都是普通的 ECMAScript 字符串，处理方法也一样。可以改变前面的代码，以使更改第一个销售人员的名字更容易：

```
oXml..employee.(@position="salesperson").name = "Michael Anderson";
```

执行这段代码将自动更新后台存储的 XML 表示方式。

因为它的开发人员的思想十分超前，E4X 还提供了在 XML 中嵌入 ECMAScript 来创建新的 XML 对象的能力。此时，必须在变量两边加上花括号。例如：

606

```
var tagname = "color";
var value = "blue";
var oXml = <{tagname}>{value}</{tagname}>;
```

在这个例子中，oXml 的值变为<color>blue</color>。

可以看到，E4X 显示出了对传统 ECMAScript 在根本上的改变，来达到以一种简单又强大的形式支持 XML 的目的。

20.2.2 for each..in 循环

在本书中，曾用 for..in 循环来迭代对象的特性名。而在 E4X 中引入的 for each..in 循环，迭代的是数组中实际的对象。例如：

```
for each (var oItem in arrItems) {
    alert(oItem);
}
```

如果要用 for..in 循环来完成同样的任务，代码如下：

```
for (var sProperty in arrItems) {
    alert(arrItems[sProperty]);
}
```

可以看到，for each..in 循环更加有用，也更接近于其他语言的类似循环。

20.2.3 新的类

E4X 引入了一些新的类来特别处理 XML：

- Namespace 对象通过使用 URL 和前缀表示命名空间。
- QName 对象表示了由一个本地名称和一个可选的命名空间 URI 组成的 XML 正规名称。
- XML 对象代表了单独的 XML 元素。
- XMLList 对象包含了任意数量的 XML 对象。

1. Namespace类

在 E4X 中，Namespace 对象是引用命名空间的简便方法。直接使用它的构造函数（接受一个或两个参数）来创建 Namespace：

```
var oNamespace1 = new Namespace("http://www.wrox.com/");
var oNamespace2 = new Namespace("wrox", "http://www.wrox.com/");
```

在第一行代码中，调用构造函数时只给出了命名空间的 URI，用 XML 表达它时如下：

```
<root xmlns="http://www.wrox.com/">
    <message>Hello World!</message>
</root>
```

在例子的第二行代码上，构造函数调用时给出了命名空间前缀和 URI，用 XML 表达时如下：

```
<wrox:root xmlns:wrox="http://www.wrox.com/">
    <wrox:message>Hello World!</wrox:message>
</wrox:root>
```

然后，Namespace 对象可以用在选择语句中：

```
var oWroxNS = new Namespace("wrox", "http://www.wrox.com/");
var oXml = <wrox:root xmlns:wrox="http://www.wrox.com/">
    <wrox:message>Hello World!</wrox:message>
</wrox:root>;
var sMessage = oXml.oWroxNS::message;
```

突出显示的一行代码使用了 oWroxNS 这个 Namespace 对象来选择<wrox:message>的文本 Hello World!。如果 XML 代码指定了命名空间，那么在 E4X 中必须用这种方法来选择元素。

2. QName类

QName 类表示 XML 元素和特性的正规名称。正规名称是由命名空间前缀及本地名称组成的，比如下面这一行：

```
<wrox:message xmlns:wrox="http://www.wrox.com/">Hello World!</wrox:message>
```

在这段代码中，wrox:message 是正规名称，wrox 表示命名空间，message 代表本地名称（也叫标签名）。前缀 wrox 指向命名空间 URI http://www.wrox.com/。QName 对象可以这样表示 wrox:message 正规名称：

```
var oWroxNS = new Namespace("wrox", "http://www.wrox.com/");
var oQName = new QName(oWroxNS, "message");
```

使用这个版本的构造函数，必须提供一个 Namespace 对象来表达正规名称，不过，如果没有指定，QName 对象也可以不用命名空间进行创建：

```
var oQName = new QName("message");
```

创建了这个对象后，有两个特性可以访问：uri 和 localName。uri 特性返回命名空间在创建时指定的 URI（如果没有指定命名空间则为空字符串）；localName 特性返回正规名称的本地名称部分。例如：

```
var oWroxNS = new Namespace("wrox", "http://www.wrox.com/");
var oQName = new QName(oWroxNS, "message");
alert(oQName.uri);           //outputs "http://www.wrox.com/"
alert(oQName.localName);    //outputs "message"
```

在这个例子中，uri 特性返回 "http://www.wrox.com/"，localName 返回 "message"。这两个特性都是只读的，如果尝试改变它们的值就会发生错误。

Qname 对象覆盖了默认的 `toString()` 函数，会返回一个 `uri::localName` 格式的字符串，比如在前面的例子中会返回 "http://www.wrox.com/:message"。

3. XML类

前面已提到过，XML 对象代表了 XML 元素，也就是说，它表示带有名称的特性的顺序列表，父节点，一系列特性、子节点和命名空间集合。

可通过两种方式创建 XML 对象。第一种方法是使用它的构造函数，并为其传入一个 XML 字符串，如下：

```
var oXml = new XML("<person><name>Nicholas C. Zakas</name></person>");
```

第二种方法是使用前面提到的语法扩展，允许你直接在代码中输入 XML，而无需使用字符串：

```
var oXml = <person>
    <name>Nicholas C. Zakas</name>
</person>;
```

要根据 XML 构造函数的一些标示参数来解释 XML：

- `XML.ignoreComments`——如果设为 `true`，则解析器会忽略所有的注释。
- `XML.ignoreProcessingInstructions`——如果设为 `true`，则解析器会忽略所有的处理指令。
- `XML.ignoreWhitespace`——如果设为 `true`，则解析器会忽略元素之间所有的空白。

`toString()`方法和 `toXMLString()`方法要一起使用，两者都能返回 XML 的字符串表达形式。如果元素包含复杂的子节点（也就是，除了文本之外的其他东西），则 `toString()`方法返回一个 XML 编码的字符串；如果元素只包含简单内容（文本），则 `toString()`只返回文本而不包括起始和结束标签。另一方面，`toXMLString()`总是返回 XML 标签，不管元素有没有子节点。例如：

```
var oXml = <name>Nicholas C. Zakas</name>;
alert(oXml.toString());           //outputs "Nicholas C. Zakas"
alert(oXml.toXMLString());       //outputs "<name>Nicholas C. Zakas</name>;"
```

在前面的代码中，第一个警告框只显示 "Nicholas C. Zakas"，因为它调用的是 `toString()`。第二个警告框显示了完整的 XML 代码 "`<name>Nicholas C. Zakas</name>`"。然而，如果元素包含的是复杂的内容，则两个方法将返回同一个值：

```
var oXml = <name><first>Nicholas</first><last>Zakas</last></name>;
alert(oXml.toString());           //outputs
"<name><first>Nicholas</first><last>Zakas</last></name>"
alert(oXml.toXMLString());       //outputs
"<name><first>Nicholas</first><last>Zakas</last></name>;"
```

在这段代码中，`toString()` 和 `toXMLString()` 都返回 "`<name><first>Nicholas</first><last>Zakas</last></name>`"。

两个方法都用 XML 构造函数的几个设置来判断如何输出 XML 代码：

- `XML.prettyPrinting`——如果设为 `true`，方法将规范化元素之间的空格。
- `XML.prettyIndent`——指定方法中的行缩进量，默认为 2。

所有的控制器标志都保存在一个对象中，可以使用 `XML.settings()` 方法对其进行引用。使

用 `XML.setSettings()` 方法可以在以后恢复这个对象中的设置，如下所示：

```
var oXmlSettings = XML.settings();      //save these settings
XML.prettyIndent = 4;
XML.ignoreWhitespace = true;
//do something here
XML.setSettings(oXmlSettings);        //return to the original settings
```

这里，首先将设置在更改之前保存到 `oXmlSettings` 中。然后又用 `setSettings()` 方法重置了设置。也可用 `XML.defaultSettings()` 方法来获取解析器默认的设置：

```
XML.setSettings(XML.defaultSettings());
```

对于 XML 对象实例，也有很多可用于操作 XML 数据的方法。其中一些方法是从 DOM 中得到的灵感；有些则不是，不过都很实用。

第一组方法用于处理命名空间和 Namespace 对象。其中大部分都很直观：`addNamespace()` 将给定的命名空间添加到元素中，`removeNamespace()` 将给定的命名空间从元素上删除。

```
var oWroxNS = new Namespace("wrox", "http://www.wrox.com/");
var oXml = <message>Hello World!</message>;
oXml.addNamespace(oWroxNS);
//do something else
oXml.removeNamespace(oWroxNS);
```

也可以用 `inScopeNamespaces()` 和 `namespaceDeclarations()` 方法获取命名空间的数组。这两个方法都返回 Namespace 对象的数组，其中 `inScopeNamespaces()` 只返回当前元素下使用的命名空间，`namespaceDeclarations()` 则返回从根元素往下的所有的命名空间。

要处理单独的命名空间，可用 `namespace()` 方法来获取一个命名空间，还有 `setNamespace()`，顾名思义，是为当前的元素设置一个命名空间。

```
var oXml = <wrox:root xmlns:wrox="http://www.wrox.com/">
    <wrox:message>Hello World!</wrox:message>
</wrox:root>;
var oWroxNS = oXml.namespace();
var oNewNS = new Namespace("ncz", "http://www.nczonline.net/");
oXml.setNamespace(oNewNS);
```

在 E4X 中访问特性十分简单，只需使用 `attribute()` 和 `attributes()` 方法。`attribute()` 方法返回特性的值。`attributes()` 方法返回包含给定元素的所有特性的 XMLList（在下一节中介绍）。

```
var oXml = <value type="string">Hello World!</value>
var sType = oXml.attribute("type"); //set to "string"
var oAtts = oXml.attributes();
```

XML 元素的子节点可以用 `child()` 方法访问，它返回单个由子节点的索引和名称表示的 XML 对象，也可以用 `children()` 方法访问，它返回所有子节点的 XMLList。另外 `childIndex()` 方法可用于判断某个元素在兄弟元素中的位置。例如：

```
var oXml = <employees>
    <employee position="Software Engineer">
        <name>Nicholas C. Zakas</name>
    </employee>
    <employee position="Salesperson">
        <name>Jim Smith</name>
    </employee>
</employees>

var oFirstEmployee = oXml.child(0);
var oFirstEmployeeToo = oXml.child("employee");
var oAllEmployees = oXml.children();
var iFirstEmployeeIndex = oFirstEmployee.childIndex(); //0
```

这段代码展示了获取 XML 代码中第一个雇员的两种方法。第一种，用 `child()` 方法以及位置 0——第一个元素的位置。第二种，用 `child()` 方法和元素的标签名。这两种调用都返回同一个雇员元素的引用。要获取所有的雇员，可调用 `children()` 方法。最后，调用第一个雇员的 `childIndex()` 方法，返回 0。

前面讨论过的方法会返回给定元素的所有类型的子节点。然而，如果只想返回指定类型的子节点或者按照不同关系返回节点，可以使用以下方法：

- `comments()`——只返回注释子节点；
- `elements()`——只返回元素子节点；
- `processingInstructions()`——只返回处理指令子节点；
- `descendants()`——返回给定节点的所有子孙节点；
- `parent()`——返回元素的父节点。

还有一些方法可用于更改子节点：

- `appendChild(child)`——在所有子节点的末尾添加新的子节点；
- `prependChild(child)`——在所有子节点的开头添加新的子节点；
- `insertChild(child, refchild)`——在给定的节点引用前插入子节点；
- `insertChildAfter(child, refchild)`——在给定引用的节点后插入子节点；
- `replace(childname, newchild)`——用新节点替换指定名称（或者指定位置上）的节点；
- `setChildren(list)`——用给定 `XMLList` 中的子节点替换所有的子节点。

这些方法都极其有用，也很容易使用：

```
var oXml = <employees>
    <employee position="Software Engineer">
        <name>Nicholas C. Zakas</name>
    </employee>
    <employee position="Salesperson">
        <name>Jim Smith</name>
    </employee>
</employees>

oXml.appendChild(<employee position="Vice President">
    <name>Benjamin Anderson</name>
</employee>);

oXml.prependChild(<employee position="User Interface Designer">
    <name>Michael Johnson</name>
</employee>);

oXml.insertChildBefore(oXml.child(2), <employee position="Human Resources Manager">
    <name>Margaret Jones</name>
</employee>);

oXml.setChildren(<employee position="President">
    <name>Richard McMichael</name>
```

```
</employee> +
<employee position="Vice President">
    <name>Rebecca Smith</name>
</employee>);
```

这段代码演示了前面讨论的一些方法。注意可以在所有方法中用 XML 字面量来表示 XML 对象。首先，这个代码中添加称为 Benjamin Anderson 的 Vice President 到雇员列表的底部。其次，添加了一个称为 Michael Johnson 的 User Interface Designer 添加到雇员列表的顶端。再次，添加了称为 Margaret Jones 的 human Resource Manager 到位置第二的雇员（此时是 Jim Simth，因为 Michael Johnson 和 Nicholas C. Zakas 现在它前面）之前。最后，所有的子节点都被 President Richard McMichael 和 Vice President Rebecca Smith（可能出现了大裁员）替换了。注意，在这一行中，在两个雇员字面量之间的加号。它表示了包含在 XMLList 对象中的值。最后，XML 如下：

```
<employees>
    <employee position="President">
        <name>Richard McMichael</name>
    </employee>
    <employee position="Vice President">
        <name>Rebecca Smith</name>
    </employee>
</employees>
```

在这一节中，谈论了很多关于简单和复杂内容的重要概念。事实上，它们十分重要，因此 XML 对象有两个方法来帮助进行判断：`hasComplexContent()` 和 `hasSimpleContent()`。两个方法都返回表示元素中是否包含相应类型内容的布尔值。根据规则，对于任何给定的元素，两个方法中仅有一个能返回 `true`。

```
var oSimpleXml = <message>Hello World!</message>;
var oComplexXML =
<message><greeting>Hello</greeting><target>World!</target></message>
var bSimple = oSimpleXml.hasSimpleContent(); // returns true
var bComplex = oComplexXML.hasComplexContent(); // returns true
```

如果想创建某个 XML 的一个完整（深）副本，可用 `copy()` 方法。这个方法返回节点的一个精确副本和所有的子孙节点（不复制祖先节点）。

```
var oXml = <message>Hello World!</message>
var oNewXml = oXml.copy();
alert(oNewXml.toXMLString()); // outputs <message>Hello World!</message>
```

XML 对象也有个方法 `length()`，总是返回 1。看上去这个设定似乎很傻，但是它可用于模糊 XML 对象和 XMLList 对象之间的在写代码上的区别。另一个方法 `contains()`，在 XML 对象中基本是无用的，它也是为了与 XMLList 对象兼容而包含的。`contains()` 方法只在将调用该方法的 XML 对象本身传递进去，才会返回 `true`，如下：

```
oXml.contains(oXml);
```

这两个方法其实都是用在 XMLList 中的，在 XML 中基本没用。

用 nodeKind()方法可以判断 XML 对象代表哪种类型的节点，可以返回 "text"、"element"、"comment"、"processing-instruction" 或 "attribute"。考虑以下 XML 对象：

```
var oXml = <employees>
    <? Don't forget the donuts! ?>
    <employee position="President">
        <name>Richard McMichael</name>
    </employee>
    <!-- just added -->
    <employee position="Vice President">
        <name>Rebecca Smith</name>
    </employee>
</employees>
```

给出这个 XML，下面的表格显示了 nodeKind()对范围内不同元素返回的值：

语句	返回值
oXml.nodeKind()	"element"
oXml.child(0)	"processing-instruction"
oXml.employee.getAttribute("position").nodeKind()	"attribute"
oXml.employee.nodeKind()	"element"
oXml.child(2)	"comment"
oXml.employee.name.childNodes(0)	"text"

normalize()方法与在 DOM 中的运行方式类似。它可以规则化（组合）元素之间空格和文本，并创建单个文本节点而不是多个文本节点。听起来比较无趣，不过对处理 XML 代码还是很有用的。

有四个方法可用来处理 XML 节点的名称：

- name()——返回节点的正规名称，是一个 QName 对象；
- localName()——返回节点的本地名称，等同于 name().localName；
- setNames(qname)——设置节点的正规名称；
- setLocalName(localname)——设置节点的本地名称。

例子：

```
var oXml = <message>Hello World!</message>
oXml.setLocalName("msg");           //changes the code to <msg>Hello World!</msg>
oXml.setName(new QName("mess"));    //chnages the code to <mess>Hello World!</mess>
```

这里，XML 代码被改变了两次，先将<message/>变成<msg/>，然后<msg/>又变成<mess/>。

最后一个方法是 text()，它返回某个元素的文本（简单内容）：

```
var oXml = <name>Nicholas C. Zakas</name>;
var sName = oXml.text();           //returns "Nicholas C. Zakas";
```

4. XMLList 类

XMLList 类——已经在前面几节中简单地介绍过了，表示 XML 对象的一个数组。与 XML 对象一样，创建 XMLList 对象也有多种方法：

首先，可用构造函数，并为其传入包含一些元素（未包含在根节点中）的 XML 字符串。例如：

```
var oXmlList = new XMLElement("<name>Nicholas C. Zakas</name><name>Michael Smith</name>");
```

在这个例子中，将两个<name/>元素的字符串传递给 XMLElement，它会创建两个单独的 XML 对象并将它们存储起来。另外，可以对已存在的 XML 对象使用加号来创建 XMLElement：

```
var oXml1 = <name>Nicholas C. Zakas</name>;
var oXml2 = <name>Michael Smith</name>;
var oXmlList = oXml1 + oXml2;
```

这三行与上面的一行的运行结果一样，创建包含两个 XML 对象的 XMLElement 对象。不过，还有一种用来创建 XMLElement 对象的方法：

```
var oXmlList = <><name>Nicholas C. Zakas</name><name>Michael Smith</name></>;
```

这是 XMLElement 对象的 XML 字面量表示法。其中重要的语法是空打开和空关闭标签，表示这不是典型的 XML 对象（在 XML 中，空标签是非法的）。

在前面一节中提到过，XML 和 XMLElement 对象从设计目的上来说是有意设计得类似的。XMLElement 对象和 XML 对象的方法都一样，尽管它们的行为不同。

614

在 XMLElement 对象中，方法通常调用列表中 XML 对象的同名方法，并将结果存储在另一个 XMLElement 对象中并返回。例如，如果对 XMLElement 对象调用 attribute("id")，它会先对每个 XMLElement 对象调用 attribute("id")。如果 XML 对象返回一个特性，该特性被添加到结果 XMLElement 中。一旦对所有 XML 对象都操作后，就对返回结果 XMLElement 对象。下面列出了所有以这种方式执行的方法：

- attribute()——返回所有 XML 对象中的给定特性的 XMLElement；
- attributes()——返回所有 XML 对象中所有特性的 XMLElement；
- child()——返回所有 XML 对象中给定名称的所有子节点的 XMLElement；
- children()——返回所有 XML 对象中的所有子节点的 XMLElement；
- comments()——返回所有 XML 对象中的所有子评论节点的 XMLElement；
- descendants()——返回所有 XML 对象的所有子孙节点的 XMLElement；
- elements()——返回所有 XML 对象的所有子元素节点的 XMLElement；
- normalize()——规范化每个 XML 对象；
- processingInstructions()——返回所有 XML 对象中所有子处理指令节点的 XMLElement；
- text()——返回所有 XML 对象的所有子文本节点的 XMLElement。

还有两个对 XML 对象无意义的方法，length() 和 contains()，当用于 XMLElement 中就显然有意义得多。length() 方法用于返回 XMLElement 中的对象的数量；contains() 方法用于判断给定的 XML 对象是否包含在 XMLElement 中，例如：

```
var oXml1 = <name>Nicholas C. Zakas</name>;
var oXml2 = <name>Michael Smith</name>;
var oXmlList = oXml1 + oXml2;
alert(oXmlList.contains(oXml1)); //outputs "true"
```

在这个例子中，通过组合 oXml1 和 oXml2 创建 oXmlList，所以，用 oXml1 调用 contains() 方法时，结果为 true。

还有其他四个方法的行为与 XML 对象有一些不同：

- copy()——返回 XMLList 对象的一个精确副本；
- hasComplexContent()——XMLList 包含一个有复杂内容的元素或者 XMLList 包含多个 XML 对象时，返回 true；
- hasSimpleContent()——XMLList 为空或者 XMLList 包含一个有简单内容的 XML 对象时，返回 true；
- parent()——如果在 XMLList 中的所有对象的父节点相同，返回这个 XML 对象，其他情况返回 undefined。

615 XML 和 XMLList 类都提供了一个非常强大的处理 XML 数据的接口。

20.2.4 实现

目前只有 BEA Weblogic Workshop 实现了 E4X(这不奇怪，因为当初就是 BEA 定义了 E4X)。在 Weblogic Workshop 中，E4X 被称为 JavaScript for XML(JSX)或者 Native XML Scripting，是用在服务器端处理 XML 的。在 Weblogic Workshop 中，JSX 文件在执行期间会被编译成 Java 类，然后可以用在其他 Java 类中，包括 Web 服务。

如果要了解更多 Weblogic Workshop 中 JavaScript for XML 的信息，见 http://dev2dev.bea.com/products/wlworkshop/articles/JSchneider_XML.jsp。

根据 Mozilla 的路线图，Mozilla 2.0 会加入 E4X 的实现(其实，Rhino——Mozilla 的 JavaScript 解释程序，已经包含了与实现 E4X 相关的代码)，这将是第一个自由的 E4X 实现。

20.3 小结

这一章我们展望了 JavaScript 未来的方向。ECMAScript 4 已被提上日程(好几年了)，但是还没有发布。本章粗略地看了 Netscape 提出的 ECMAScript 4 的草案，并讨论了目前实现了哪些(如果有)。

本章还讨论了一个新的标准——ECMAScript for XML，它为 ECMAScript 添加了直接的 XML 支持。尽管这个标准对 Web 浏览器的客户端脚本编程还不可用，但 BEA Weblogic Workshop 能支持它，并且 Workshop 在今后几年中将继续推动它的应用。

最后要说的是，JavaScript 依然在发展，并且没有人能确定它的发展方向。然而，因为 JavaScript 被越来越多的平台和应用所采用，它的未来十分光明。例如，MacOS X Tiger 有一个称作 Dashboard 的新型开发平台，它用 JavaScript 来创建轻量级应用，可以在 MacOS 桌面环境中运行。有了这种支持，还有微软的 JScript.NET 的支持，即使看到 JavaScript 逐渐走出 Web 世界进入桌面领域，也不会令人惊讶。

站在巨人的肩上

Standing on Shoulders of Giants



www.turingbook.com

Professional JavaScript for Web Developers

JavaScript 高级程序设计

“如果你像我一样，想学习或者熟练掌握今天最热门的 Web 开发技术，本书是一个绝佳的起点，适合在所有 Ajax 图书之前阅读。”

—— J. Ambrose Little, Microsoft MVP

“本书作者显然非常了解读者的需要，切中要害，信息密集。单单是对客户端通信、Web 服务、正则表达式、DOM、XML 处理等现代 JavaScript 技术的详细讲解，就已经物超所值。”

—— JavaScriptKit.com

JavaScript 作为赋予网页活力与交互性的主要手段之一，早已经成为 Web 设计师和开发人员的必备技能。全世界无数网页每天都在依靠 JavaScript 完成各种关键任务。然而，JavaScript 可能也是被人误解和误用最多的主流编程语言。很多人将它看作 Java 等面向对象编程语言的功能不全的小兄弟，甚至贬为雕虫小技，对它不屑一顾。

如今，随着越来越多的程序员转向浏览器 / 服务器模式开发，更加上 Web 2.0 和 Ajax 的兴起，JavaScript 已经被推到了舞台中心。人们开始认识到，JavaScript 绝非一种容易学习和掌握的技术，它同时具有面向对象、过程和函数型语言三类语言的特性，将灵活性与强大功能完美结合。迄今为止，它的惊人潜力还远远没有真正释放出来。

本书针对开发人员和有经验的 Web 设计师撰写，在简明扼要地讲述了 JavaScript 的语言核心 ECMAScript，以及面向对象特性、BOM、DOM 之后，很快转向高级主题：正则表达式、事件、数据验证、表排序、拖放、错误处理、调试、XML、Web 服务、安全、国际化、优化和知识产权保护，能够解决 Web 开发者目前面对的各种迫切问题。

Nicholas C. Zakas 世界知名的 JavaScript 专家和 Web 开发人员。Nicholas 拥有丰富的 Web 开发和界面设计经验，曾经参与许多世界大公司的 Web 解决方案开发，并与他人合作撰写了畅销书《Ajax 高级程序设计》(中文版已由人民邮电出版社出版)。可以通过 www.nczonline.net 与他联系。

本书相关信息请访问：图灵网站 <http://www.turingbook.com>
读者 / 作者热线：(010)88593802
反馈 / 投稿 / 推荐信箱：contact@turingbook.com

上架建议

计算机 / 网络开发 / 程序设计

ISBN 978-7-115-15209-1



9 787115 152091 >



ISBN 978-7-115-15209-1/TP

定价：59.00 元