JavaScript 高级程序设计

序言

前言

虽然服务器端 的 Netscape Enterprise Server 和 Active Server Pages (ASP) 都曾经支持过 Javascript，但它主要还是 Web 浏览器使用的客户端脚本语言。目前它的重点是帮助开发者与 Web 页面和 Web 浏览器窗口本身进行交互。

Javascript 是一种不严格基于 Java 的面向对象程序设计语言，以嵌入式 Java 小程序的形式在 Web 上广为使用。虽然 Javascript 的语法和程序设计方法都与 Java 相似，但它并非 Java 语言的简化版本。相反的，Javascript 是一种独立的语言，在全世界的 Web 浏览器中都可以找到它，启用它可以增强用户与 Web 站点和 Web 应用程序之间的交互。

本书从最早期的 Netscape 浏览器中的 Javascript 开始讲起，直到当前它对 XML 和 Web 服务的具体支持。你将学到如何扩展这种语言，使它适应特殊的需求，还会学到如何在没有 Java 或隐藏框架这些媒介的情况下创建无缝的客户－服务器通信。简而言之，你将学到如何将 Javascript 解决方案应用到 Web 开发者面对的商业问题上。

本书涵盖的内容

本书提供的是开发者级别的 Javascript 介绍，包括很多高级的有用特性。

本书开头探讨了 Javascript 的起源以及迄今为止的发展。之后详细介绍了构成 Javascript 实现的各个组件，着重介绍了 ECMAScript 和文档对象模型 DOM 这样的标准。此外还讨论了不同 Web 浏览器中使用的 Javascript 实现的不同。

基于上述讨论，本书开始介绍 Javascript 的基本概念，包括面向对象的程序设计版本、继承性以及它在各种标记语言（如 HTML）中的用法。在探讨了浏览器

检测技术，介绍过在 Javascript 中使用正则表达式后，本书对事件和事件处理进行了深度考察。之后，它应用了所有这些知识，来创建动态用户界面。

本书最后一部分的重点是与在 Web 应用程序中部署 Javascript 解决方案有关的问题。这些主题包括错误处理、调试、安全性、优化/模糊化、XML 和 Web 服务器。

本书的适用对象

本书针对的读者群有三类：

- 熟悉面向对象程序设计方法，由于 Javascript 与传统的 OO 语言（如 Java 和 C++）相关所以想学习它的的有经验的开发者。
- 尝试提高 Web 站点和 Web 应用程序可用性的 Web 应用程序开发者。
- 目的在于更好理解 Javascript 语言的初学者。

此外，如果你熟悉下列相关技术，那么表明本书也适用于你：

- XML
- XSLT
- Java
- Web Services
- HTML
- CSS

本书针对的不是没有计算机科学的基础背景的初学者，也不是那些想在 Web 站点添加一些简单的用户交互特性的人。这些读者应该阅读 Wrox 编写的《Beginning Javascript》一书的第二版（Willey Publishing, Inc., ISBN 0-7645-5587-1）。

使用本书的前提需求

要运行本书中的示例，需要下列软件：

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

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

从本书的站点 http://www.wrox.com 可以下载书中示例的完整源代码。

本书的结构

1. Javascript 是什么？

这一章解释了 Javascript 的起源，它是怎样长生的，如何发展，现状如何。引入的概念包括 Javascript 和 ECMAScript、文档对象模型 DOM 以及浏览器对象模型 BOM 之间的关系。此外还有与欧洲计算机制造商协会 ECMA 和 W3C 有关的各项标准。

2. ECMAScript 基础

这一章分析了 Javascript 基于的核心技术 ECMAScript。从变量和函数的声明到使用和理解原始与引用值，它说明了编写 Javascript 代码必需的基础语法和概念。

3. 对象基础

这一章的重点是用 Javascript 进行面向对象的程序设计（OOP）的基础。涵盖的主题包括用各种方法定义定制的对象、创建对象实例以及了解 Javascript 和 Java 中的 OOP 的相同点和不同点。

4. 继承性

这一章继续解释 Javascript 中的 OOP，说明了继承机制是如何作用的，其中讨论了各种实现继承性的方法，并且还比较了它们与 Java 中的继承性的异同。

5. 浏览器中的 Javascript

这一章解释了如何把 Javascript 嵌入用各种语言（如 HTML、SVG 和 XUL）编写的 Web 页。此外还介绍了浏览器对象模型 BOM 及它的各种对象和接口。

6. DOM 基础

这一章介绍了 Javascript 中实现的 DOM，包括专门适用于 Web 开发者的 DOM 概念。后面用 HTML、SVG 和 XUL 编写的示例中使用了这些概念。

7. 正则表达式

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

8. 探测浏览器和操作系统

这一章解释了编写能在各种 Web 浏览器上运行的 Javascript 脚本的重要性。它讨论了两种探测浏览器的方法，即对象/特性探测法和用户－代理字符串探测法，每种方法的优点和缺点都被列了出来。

9. 事件

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

10. 高级 DOM 技术

这一章介绍了一些更高级的 DOM 特性，包括范围和样式表操作。我举了一个例子，说明如何使用这些技术，此外还讨论了如何实在跨浏览器的支持。

11. 表单和数据完整性

这一章讨论了使用表单时数据验证的重要性。在介绍处理验证的方法时，还应用了前面介绍过的概念，如正则表达式、事件和 DOM 操作。

12.表排序

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

13.拖放

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

14.错误处理

这一章通过讨论 try…catch 语句和 onerror 事件处理程序的用法介绍了 Javascript 中的事件处理概念。另一个主题是用 throw 语句创建定制的错误消息以及 Javascript 调试器的用法。

15.Javascript 中的 XML

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

16.客户－服务器通信

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

17.Web 服务

这一章介绍了如何用 Javascript 提供 Web 服务，其中讨论了 Internet Explorer 和 Mozilla 中使用的不同方法，还为原本没有 Web 服务支持的浏览器提供了一种基本的 Web 服务解决方案。

18. 用插件进行交互

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

19. 部署问题

这一章的重点是完成 Javascript 编码后的操作。它说明了在把 Javascript 解决方案部署到 Web 站点或 Web 应用程序之前要做哪些操作。其中的主题包括安全问题、国际化问题、优化、知识产权保护和 Section 508 Compliance。

20. Javascript 的发展

这一章探察了 Javascript 的未来，介绍了这种语言的发展方向。其中讨论了 ECMAScript 的 ECMAScript 4 和 XML。

规约

为了帮助你最大限度的利用本书，我在全书中使用了大量规约。

这样的矩形框中放置的是重要的、不容忘记的信息，它与周围的内容直接相关。

提示、暗示、小窍门和离题话都像这样用斜体显示，前面有缩进。

至于文本中的样式：

在介绍重要的单词时，高亮显示它们

用 Ctrl+A 这样的形式说明键盘按键

正文中的文件名、URL 和代码用 persistence.properties 这样的形式显示

代码有两种形式：

PXXV 代码

源代码

在练习本书中的示例时，可以选择手动输入代码，也可以使用本书附带的源代码文件。在 http://www.wrox.com 处可以下载到本书中使用的所有源代码。进入该站点后，只需要找到本书的名字（或者使用 Search 框，又或者点击列表中的一个名字），点击本书的细节页面中的 Download Code 链接，可以找到本书中的源代码。

由于许多数的名字相似，所以用 ISBN 号检索本书更容易找到它。本书的 ISBN 号是 0-7645-7908-8。

下载了代码后，用解压缩工具把它解压缩。此外，还可以在 Wrox 的主下载页面 http://www.wrox.com/dynamic/books/download.aspx 处找到本书和其他 Wrox 出版的书的代码。

勘误表

我们一直努力确保代码或正文中没有错误。不过，是人都会犯错误。如果你发现了我们出版的书中的错误，例如拼写错误或代码错，请告知我们，我们将会非常感谢。把勘误表发给我们，就能节省其他读者的时间，同时还能帮助我们提高信息的质量。

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

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

P2p.wrox.com

关于本书的讨论，请加入 P2P 论坛 p2p.wrox.com。该论坛是基于 Web 的系统，你可以在此发布与 Wrox 出版的书和相关的技术有关的消息，与其他读者和技术员进行交流。该论坛有预订功能，当你选择的感兴趣的主题有新帖子发布时，就会把它通过 email 发送给你。Wrox 的作者、编辑、业界的其他专家和像你一样的读者都会出现在这些论坛中。

在 http://p2p.wrox.com 处可以找到各种对你有用的论坛，不只是对你阅读本书有帮助，对你开发程序也有帮助。加入论坛的步骤如下：

1. 访问 p2p.wrox.com，点击 Register 链接。
2. 阅读使用条款，点击 Agree 链接。
3. 填写所有必需的信息以及你想提供的选填信息，点击 Submit 链接。
4. 你将收到一封 email，其中具有验证你的帐户的信息以及完成加入论坛的操作的信息。

即使不加入 P2P，也可以阅读论坛中的消息，不过要发布自己的消息，就必须加入论坛。

加入论坛后，可以发布新消息，回复其他用户发布的消息。可以随时在 Web 上阅读论坛上的消息。如果想让某个论坛的新消息以 email 的形式发送给你，可以点击 Subscribe to This Forum 图标，然后在论坛列表中选择你要预订的论坛的名字。

要了解更多如何使用 Wrox P2P 论坛的信息，请阅读 P2P FAQs，可以看到论坛软件是如何运行的，以及与 P2P 和 Wrox 出版的书相关的常见问题的答案。要阅读 FAQ，请点击 P2P 页面上的 FAQ 链接。

序言

译者序

亲爱的读者：

当您从书架上拿出这本书的时候，我想您肯定对 Ajax 技术有着浓厚的兴趣，而本书也正是您的正确选择。本书的作者 Nicholas C. Zakas 用通俗易懂的语言，将 JavaScript 的诞生、现在的状况、未来的发展和与其紧密相关的各种技术一一详尽地叙述出来，刚学 JavaScript 的朋友，可以按部就班成为高手，而已经是高手的朋友，则可以将本书作为参考手册。

第 1 章讲述了 JavaScript 的起源,给大家一个关于 JavaScript 正确的认知。

第 2~5 章详细介绍了 JavaScript 语言本身，揭示了一些 JavaScript 不为认知的语言特点。

第 6~9 章介绍了 JavaScript 和浏览器进行交互的一些基础知识和一些进阶知识，如 DOM 的基础、正则表达式。

第 10~13 章介绍了一些更加高级的 JavaScript 技巧，这些技巧可以构建良好的客户端逻辑，包括表格排序、拖动等。

第 14 章关于错误处理的内容，既有如何编程处理 JavaScript 错误，也包含了如何调试 JavaScript 的方法，而调试一直是 JavaScript 的弱项。

第 16、17 章讲述了利用 JavaScript 进行客户端到服务器的同学，不仅仅介绍了现在的 Ajax 技术的基础 XML HTTP Request，还介绍了曾经出现过的一些方法。第 17 章更明确的介绍了如何调用 Web 服务。

第 19 章，介绍了如何考虑生产环境中 JavaScript 所需要注意的一些事情，如安全性、性能等。

第 20 章，展望了 JavaScript 未来的发展。

本书除了介绍了 JavaScript 的各个方面外，更难得的是，作者更涵盖了现今各个流行浏览器之间在这些方面的区别，并帮助读者，解决这些问题。

本书第 1~5 章由张欣翻译，第 6~15 章由曹力翻译，第 16~20 章由王霄翻译，全书由张欣统稿、润色及审校。还要感谢全体工作人员的努力才将本书完成。

我们深深地感我们的家人和朋友。在翻译过程中，他们给予了我们莫大的关心、支持和帮助。

限于我们的水平，翻译过程中的疏漏和错误再作难免，请广大读者批评指正。

曹力

2006 年于东南大学

第 1 章 JavaScript 是什么

1.1 历史简述

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

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

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

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

1.1 历史简述

大概在 1992 年，一家称作 Nombas 的公司开始开发一种叫做 C 减减（C-minus-minus，简称 Cmm）的嵌入式脚本语言。Cmm 背后的理念很简单：一个足够强大可以替代宏操作（macro）的脚本语言，同时保持与 C（和 C++）足够的相似性，以便开发人员能很快学会。这个脚本语言捆绑在一个叫做 CEnvi 的共享软件产品中，它首次向开发人员展示了这种语

言的威力。Nombas 最终把 Cmm 的名字改成了 ScriptEase，原因是后面的部分（mm）听起来过于“消极”，同时字母 C“令人害怕”（http://www.nombas.com/us/scripting/history.htm）。现在 ScriptEase 已经成为了 Nombas 产品背后的主要驱动力。当 Netscape Navigator 崭露头角时，Nombas 开发了一个可以嵌入网页中的 CEnvi 的版本。这些早期的试验称为 Espresso Page（浓咖啡般的页面），它们代表了第一个在万维网上使用的客户端脚本语言。而 Nombas 丝毫没有料到它的理念将会成为因特网的一块重要基石。

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

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

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

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

1997 年，JavaScript 1.1 作为一个草案提交给欧洲计算机制造商协会（ECMA）。第 39 技术委员会（TC39）被委派来“标准化一个通用、跨平台、中立于厂商的脚本语言的语法和语

义”（http://www.ecma-international.org/memento/TC39.htm）。由来自 Netscape、Sun、微软、Borland 和其他一些对脚本编程感兴趣的公司的程序员组成的 TC39 锤炼出了 ECMA-262，该标准定义了叫做 ECMAScript 的全新脚本语言。

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

第1章 JavaScript 是什么

1.2 JavaScript 实现

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

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

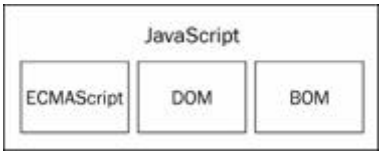

图 1-1

1.2.1 ECMAScript

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

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

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

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

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

图 1-2

每个浏览器都有它自己的 ECMAScript 接口的实现，然后这个实现又被扩展，包含了 DOM 和 BOM(在以下几节中再讨论)。当然还有其他实现并扩展了 ECMAScript

的语言，例如 Windows 脚本宿主（Windows Scripting Host，WSH）、Macromedia 的 Flash 与 Director MX 中的 ActionScript，以及 Nombas ScriptEase。

1. ECMAScript 的版本

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

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

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

2. 何谓 ECMAScript 符合性

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

- 符合的实现必须按照 ECMA-262 中所描述的支持所有的“类型、值、对象、属性、函数和程序语法及语义”（ECMA-262，第 1 页）；
- 符合的实现必须支持 Unicode 字符标准（UCS）；

- 符合的实现可以增加没有在 ECMA-262 中指定的“额外的类型、值、对象、属性和函数”。ECMA-262 将这些增加描述为规范中未给定的新对象或对象的新属性；

- 符合的实现可以支持没有在 ECMA-262 中定义的“程序和正则表达式语法”（意思是可以替换或者扩展内建的正则表达式支持）。

所有的 ECMAScript 实现必须符合以上标准。

3. Web 浏览器中的 ECMAScript 支持

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

在 ECMA-262 第一版定稿之前，发布含有 JavaScript 1.2 的 Netscape Navigator 4.0 是在 1997 年，在那年晚些时候，ECMA-262 标准被接受并标准化。因此，JavaScript 1.2 并不和 ECMAScript 的第一版兼容，虽然 ECMAScript 应该基于 JavaScript 1.1。

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

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

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

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

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

1.2.2 DOM

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

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

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

这段代码可以用 DOM 绘制成一个节点层次图（如图 1-3 所示）。

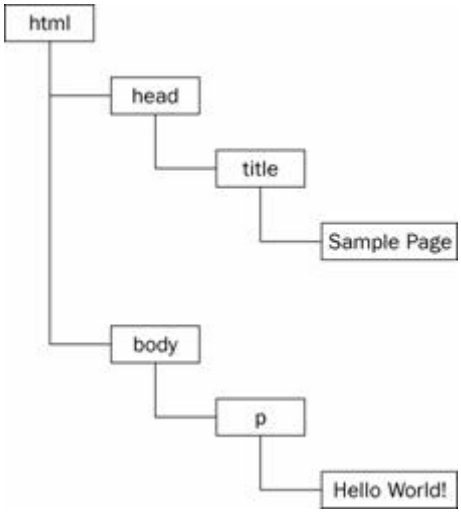

图 1-3

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

1．为什么 DOM 必不可少

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

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

2．DOM 的各个 Level

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

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

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

DOM Level 2 引入几种 DOM 新模块，用于处理新的接口类型：

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

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

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

3. **其他 DOM**

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

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

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

4. Web浏览器中的DOM支持

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

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

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

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

1.2.3 BOM

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

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

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

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

第 1 章 JavaScript 是什么

1.3 小结

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

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

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

第2章ECMAScript 基础

2.1 语法

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

ECMAScript 的基础概念如下：

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

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

- **每行结尾的分号可有可无**。Java、C 和 Perl 都要求每行代码以分号（;）结束才符合语法。ECMAScript 则允许开发者自行决定是否以分号结束一行代码。如果没有分号，ECMAScript 就把这行代码的结尾看作该语句的结尾（与 Visual Basic 和 VBScript 相似），前提是这样没有破坏代码的语义。最好的代码编写习惯是总

加入分号，因为没有分号，有些浏览器就不能正确运行，不过根据 ECMAScript 标准，下面两行代码的语法都是正确的：

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

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

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

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

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

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

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

第 2 章 ECMAScript 基础

2.2 变量

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

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

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

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

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

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

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

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

```
var test;
```

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

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

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

变量名需要遵守两条简单的规则：

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

下面的变量名都是合法的：

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

当然，只是因为变量名的语法正确并不意味着就该使用它们。变量还应遵守一条著名的命名规则：

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

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

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

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

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

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

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

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

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

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

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

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

第 2?ECMAScript 基础

2.3 关键字

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

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

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

第 2 章 ECMAScript 基础

2.4 保留字

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

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

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

第 2 章 ECMAScript 基础

2.5 原始值和引用值

在 ECMAScript 中，变量可以存放两种类型的值，即原始值和引用值。

- 贾?/SPAN>（primitive value）是存储在栈（stack）中的简单数据段，也就是说，它们的值直接存储在变量访问的位置。

- 引用值（reference value）是存储在堆（heap）中的对象，也就是说，存储在变量处的值是一个指针（point），指向存储对象的内存处。

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

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

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

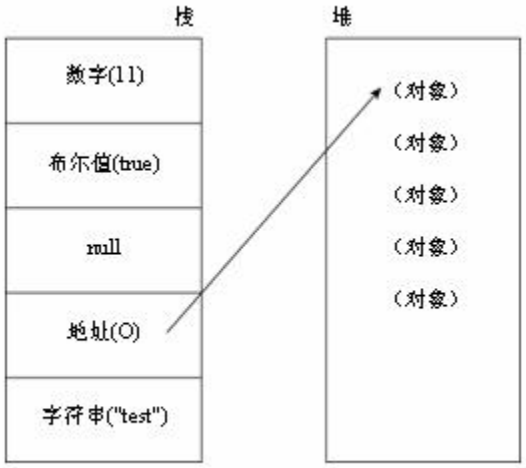

图 2-1

第3章ECMAScript 基础

2.6 原始类型

如前所述，ECMAScript 有 5 种原始类型(primitive type、Null、Boolean、Number 和 String。ECMA-262 把术语类型（type）定义为值的一个集合，每种原始类型定义了它包含的值的范围及其字面量表示形式。ECMAScript 提供了 typeof 运算符来判断一个值是否在某种类型的范围内。可以用这种运算符判断一个值是否表示一种原始类型；如果它是原始类型，还可以判断它表示哪种原始类型。

2.6.1 typeof 运算符

typeof 运算符有一个参数，即要检查的变量或值。例如：

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

对变量或值调用 typeof 运算符将返回下列值之一：

- "undefined"，如果变量是 Undefined 型的。
- "boolean"，如果变量是 Boolean 型的。
- "number"，如果变量是 Number 型的。

- "string"，如果变量是 String 型的。

- "object"，如果变量是一种引用类型或 Null 类型的。

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

2.6.2 Undefined 类型

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

```
var oTemp;
```

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

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

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

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

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

```
        var oTemp;

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

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

前面的代码对两个变量输出的都是"undefined"，即使只有变量 oTemp2 是未定义的。如果不用 typeof 运算符，就对 oTemp2 使用其他运算符，这将引起错误，因为那些运算符只能用于已定义的变量。例如，下面的代码将引发错误：

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

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

当函数无明确返回值时，返回的也是值 undefined，如下所示：

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

2.6.3 Null 类型

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

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

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

2.6.4 Boolean 类型

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

```
var bFound = true;
var bLost = false;
```

2.6.5 Number 类型

ECMA-262 中定义的最特殊的类型是 Number 型。这种类型既可以表示 32 位的整数，还可以表示 64 位的浮点数。直接输入的（而不是从另一个变量访问的）任何数字都被看作 Number 型的字面量。例如，下面的代码声明了存放整数值的变量，它的值由字面量 55 定义：

```
var iNum = 55;
```

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

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

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

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

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

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

```
var fNum = 5.0;
```

浮点字面量的有趣之处在于，用它进行计算前，真正存储的是字符串。

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

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

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

也可用科学记数法表示非常小的数，例如 0.00000000000000003 可以表示为 3-e17（这里，10 被升到-17 次幂，意味着需要被 10 除 17 次）。ECMAScript 默认把具有 6 个或 6 个以上前导 0 的浮点数转换成科学记数法。

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

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

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

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

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

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

最后一个特殊值是 NaN，表示非数（Not a Number）。NaN 是个奇怪的特殊值。一般说来，这种情况发生在类型（String、Boolean 等）转换失败时。例如，要把单词 blue 转换成数值就会失败，因为没有与之等价的数值。与无穷大值一样，

NaN 也不能用于算术计算。NaN 的另一个奇特之处在于，它与自身不相等，这意味着下面的代码将返回 false：

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

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

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

2.6.6 String 类型

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

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

图 2-2

字符串字面量是由双引号（″）或单引号（˜）声明的。与 Java 不同的是，双引号用于声明字符串，单引号用于声明字符。但是，由于 ECMAScript 没有字符类型，所以可使用这两种表示法中的任何一种。例如，下面的两行代码都有效：

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

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

字 面 量	含 义

\n	换行
\t	制表符
\b	空格
\r	回车
\f	换页符
\\	反斜杠
\ ˜	单引号
\ ″	双引号
\0nnn	八进制代码 *nnn*（*n* 是 0 到 7 中的一个八进制数字）表示的字符
\xnn	十六进制代码 *nn*（*n* 是 0 到 F 中的一个十六进制数字）表示的字符
\unnnn	十六进制代码 *nnnn*（*n* 是 0 到 F 中的一个十六进制数字）表示的 Unicode 字符

第 2 章 ECMAScript 基础

2.7 转换

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

2.7.1 转换成字符串

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

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

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

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

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

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

Number 类型的 toString()方法比较特殊，它有两种模式，即默认模式和基模式。采用默认模式，toString()方法只是用相应的字符串输出数字值（无论是整数、浮点数还是科学记数法），如下所示：

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

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

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

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

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

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

2.7.2 转换成数字

ECMAScript 提供了两种把非数字的原始值转换成数字的方法，即 parseInt()和 parseFloat()。正如你可能想到的，前者把值转换成整数，后者把值转换成浮点数。

只有对 String 类型调用这些方法，它们才能正确运行；对其他类型返回的都是 NaN。

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

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

parseInt()方法还有基模式，可以把二进制、八进制、十六进制或其他任何进制的字符串转换成整数。基是由 parseInt()方法的第二个参数指定的，所以要解析十六进制的值，需如下调用 parseInt()方法：

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

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

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

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

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

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

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

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

下面是使用 parseFloat()方法的示例：

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

2.7.3 强制类型转换

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

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

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

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

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

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

用　法	结　果
Number(false)	0
Number(true)	1
Number(undefined)	NaN
Number(null)	0
Number("5.5")	5.5
Number("56")	56
Number("5.6.7")	NaN
Number(new Object())	NaN
Number(100)	100

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

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

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

第 2 章 ECMAScript 基础

2.8 引用类型

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

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

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

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

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

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

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

尽管括号不是必需的，但为避免混乱，最好使用括号。

2.8.1 Object 类

Object 类自身用处不大，不过在了解其他类之前，还是应该先了解它。因为 ECMAScript 中的 Object 类与 Java 中的 java.lang.object 相似，ECMAScript 中的所

有类都由这个类继承而来，Object 类中的所有属性和方法都会出现在其他类中，所以理解了 Object 类，就可以更好地理解其他类。

Object 类具有下列属性：

- Constructor——对创建对象的函数的引用（指针）。对于 Object 类，该指针指向原始的 object()函数。

- Prototype——对该对象的对象原型的引用。第 3 章将进一步讨论原型。对于所有的类，它默认返回 Object 对象的一个实例。

Object 类还有几个方法：

- HasOwnProperty(property)——判断对象是否有某个特定的属性。必须用字符串指定该属性（例如，o.hasOwnProperty("name")）。

- IsPrototypeOf(object)——判断该对象是否为另一个对象的原型。

- PropertyIsEnumerable(property)——判断给定的属性是否可以用 for…in 语句（本章后面将讨论该语句）进行枚举。

- ToString()——返回对象的原始字符串表示。对于 Object 类，ECMA-262 没有定义这个值，所以不同的 ECMAScript 实现具有不同的值。

- ValueOf()——返回最适合该对象的原始值。对于许多类，该方法返回的值都与 toString()的返回值相同。

上面列出的每种属性和方法都会被其他类覆盖。

2.8.2 Boolean 类

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

```
var oBoolean Object = new Boolean(true);
```

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

问题通常出现在Boolean表达式中使用Boolean对象时。例如：

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

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

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

2.8.3 Number类

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

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

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

要得到数字对象的Number原始值，只需要使用valueOf()方法：

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

当然，Number类也有toString()方法，在讨论类型转换的小节中已经详细讨论过该方法。除从Object类继承的标准方法外，Number类还有几个处理数值的专用方法。

toFixed()方法返回的是具有指定位数小数的数字的字符串表示。例如：

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2.8.4 String 类

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

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

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

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

如果运行这段代码，输出是"true"，说明这些值真的相等。

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

String 类具有属性 length，它是字符串中的字符个数：

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

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

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

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

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

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

这个例子输出"101"，即小写字母"e"的字符代码。

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

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

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

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

迄今为止，已讨论过连接字符串的方法，访问字符串中的单个字符的方法。不过如果无法确定在某个字符串中是否确实存在一个字符,应该调用什么方法呢？这时，可调用 indexOf()和 lastIndexOf()方法。

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

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

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

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

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

示例如下：

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

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

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

采用这种结构，可以确保这段代码在所有实现中都能正确运行。

localCompare()的独特之处在于，实现所处的区域（locale，兼指国家/地区和语言）确切说明了这种方法运行的方式。在美国，英语是 ECMAScript 实现的标准语言，localCompare()是区分大小写的，大写字母在字母顺序上排在小写字母之后。不过，在其他区域情况可能并非如此。

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

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

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

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

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

这样即可看出 slice()和 substring()方法的主要不同。当只有参数-3 时，slice()返回"rld"，substring()则返回"hello world"。这是因为对于字符串"hello world"，slice(-3)将被转换成 slice(8)，而 substring(-3)则被转换成 substring(0)。同样，使用参数 3

和-4 时，差别也很明显。slice()方法将被转换成 slice(3,7)，与前面的例子相同，返回"lo w"。而 substring()方法则将这两个参数解释为 substring(3,0)，实际上即 substring(0,3)，因为 substring()总是把较小的数字作为起始位，较大的数字作为终止位。因此，substring(3,-4)返回的是"hel"。这里的最后一行代码用来说明如何使用这些方法。

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

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

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

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

2.8.5 instanceof 运算符

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

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

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

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

第 2 章 ECMAScript 基础

2.9 运算符（1）

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

2.9.1 一元运算符

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

1. delete

delete 运算符删除对以前定义的对象属性或方法的引用。例如：

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

这个例子中，删除了 name 属性，意味着强制解除对它的引用，将其设置为 undefined（即创建的未初始化的变量的值）。

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

```
delete o.toString;
```

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

2. void

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

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

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

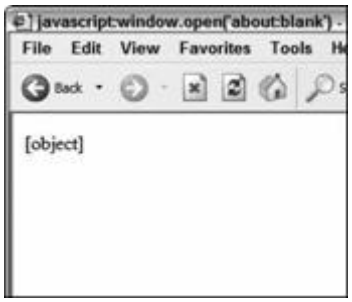

图 2-3

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

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

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

3. 前增量/前减量运算符

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

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

第二行代码把 iNum 增加到了 11，它实质上等价于：

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

同样，前减量运算符是从数值上减 1，形式是在变量前放两个减号（--）：

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

在这个例子中，第二行代码把 iNum 的值减到 9。

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

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

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

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

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

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

4. 后增量/后减量运算符

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

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

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

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

第二行代码把 iNum 的值减到 9。

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

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

与前缀式运算符的例子相似，第二行代码对 iNum 进行减量运算，第三行代码显示结果（“9”）。第四行代码再次显示 iNum 的值，不过这次是在同一个语句中应用后减量运算符。由于减量运算发生在计算过表达式之后，所以这条语句显示的数是"9"。执行了第五行代码后，alert 函数显示的是"8"，因为在执行第四行代码之后和执行第五行代码之前，执行了后减量运算。

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

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

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

5. 一元加法和一元减法

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

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

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

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

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

另一方面，一元减法就是对数值求负（例如把 25 转换成-25）：

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

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

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

在上面的代码中，一元减法运算符将把字符串"25"转换成-25（一元减法运算符对十六进制值和十进制的处理方式与一元加法运算符相似，只是它还会对该值求负）。

2.9.2 位运算符

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

1. 重温整数

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

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

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

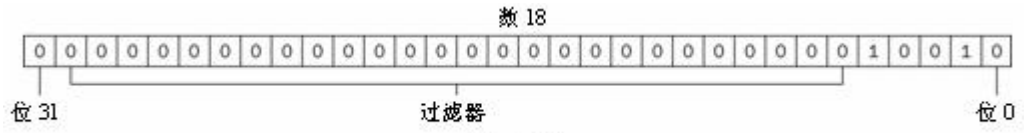

图 2-4

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

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

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

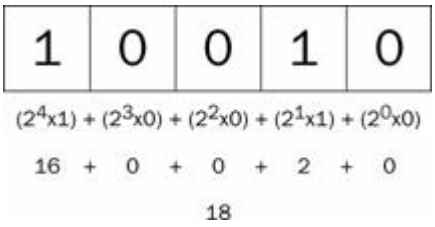

图　2-5

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

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

(2) 求得二进制反码，即要把 0 替换为 1，把 1 替换为 0；

(3) 在二进制反码上加 1。

要确定–18 的二进制表示，首先必须得到 18 的二进制表示，如下所示：

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

接下来，计算二进制反码，如下所示：

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

最后，在二进制反码上加 1，如下所示：

```
1111 1111 1111 1111 1111 1111 1110 1101
                                      1
---------------------------------------
1111 1111 1111 1111 1111 1111 1110 1110
```

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

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

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

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

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

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

第 2 章 ECMAScript 基础

2.9 运算符（2）

2. **位运算** NOT

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

(1) 把运算数转换成 32 位数字；

(2) 把二进制形式转换成它的二进制反码；

(3) 把二进制反码转换成浮点数。

例如：

```
var iNum1 = 25;       //25 is equal to 00000000000000000000000000011001
var iNum2 = ~iNum1;    //convert to 11111111111111111111111111100110
alert(iNum2);         //outputs "-26"
```

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

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

3. **位运算** AND

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

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

例如，要对数字 25 和 3 进行 AND 运算，代码如下所示：

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

25 和 3 进行 AND 运算的结果是 1。为什么？分析如下：

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

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

4. **位运算** OR

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

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

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

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

25 和 3 进行 OR 运算的结果是 27：

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

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

5. **位运算** XOR

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

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

对 25 和 3 进行 XOR 运算，代码如下：

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

25 和 3 进行 XOR 运算的结果是 26：

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

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

6. 左移运算

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

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

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

注意，左移操作保留数字的符号位。例如，如果把–2 左移 5 位，得到的是–64，而不是 64。“符号仍然存储在第 32 位中吗？”是的，不过这在 ECMAScript 后台进行，开发者不能直接访问第 32 个数位。即使输出二进制字符串形式的负数，显示的也是负号形式（例如，–2 将显示为–10，而不是 11111111111111111111111111111110）。

图 2-7

7. 有符号右移运算

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

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

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

图 2-7

8.

无符号右移运算

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

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

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

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

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

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

```
alert(iUnsignedNeg64.toString(2));
```

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

2.9.3 Boolean 运算符

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

1. 逻辑 NOT

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

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

通常，该运算符用于控制循环（后面有相关的讨论）：

```
var bFound = false;
var i = 0;

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

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

判断 ECMAScript 变量的 Boolean 值时，也可以使用逻辑 NOT 运算符。这样做需要在一行代码中使用两个逻辑 NOT 运算符。无论运算数是什么类型的，第一个 NOT 运算符返回 Boolean 值。第二个 NOT 将对该 Boolean 值求负，从而给出变量真正的 Boolean 值。

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

运行这个例子，生成的输出如下所示：

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

2. 逻辑 AND 运算符

在 ECMAScript 中，逻辑 AND 运算符用双和号（&&）表示：

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

下面的真值表描述了逻辑 AND 运算符的行为：

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

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

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

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

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

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

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

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

3. 逻辑 OR 运算符

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

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

下面的真值表描述了逻辑 OR 运算符的行为：

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

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

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

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

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

与前面的例子相同，变量 bUnknown 是未定义的。不过，由于变量 bTrue 的值为 true，bUnknown 不会被计算，因此输出的是"true"。如果把 bTrue 的值改为 false，将发生错误：

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

第 2 章 ECMAScript 基础

2.9 运算符（3）

2.9.4 乘性运算符

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

1. 乘法运算符

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

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

不过，在处理特殊值时，ECMAScript 中的乘法还有一些特殊行为：

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

- Infinity 乘以 Infinity，结果为 Infinity。

2. 除法运算符

除法运算符由斜线（/）表示，用第二个运算数除第一个运算数：

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

与乘法运算符相似，对于特殊的值，除法运算符也有特殊行为：

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

3. 取模运算符

取模（余数）运算符由百分号（%）表示，使用方式如下：

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

与其他乘性运算符相似，对于特殊的值，取模运算符也有特殊行为：

- 如果运算数都是数字，执行常规的算术除法运算，返回除法运算得到的余数。
- 如果被除数是 Infinity，或者除数是 0，结果为 NaN。

- Infinity 被 Infinity 除，结果为 NaN。
- 如果除数是无穷大的数，结果为被除数。
- 如果被除数为 0，结果为 0。

2.9.5 加性运算符

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

1. 加法运算符

加法运算符（+）的用法如你所料：

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

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

- 某个运算数是 NaN，结果为 NaN。
- Infinity 加 Infinity，结果为 Infinity。
- -Infinity 加-Infinity，结果为-Infinity。
- Infinity 加-Infinity，结果为 NaN。
- +0 加+0，结果为+0。
- –0 加+0，结果为+0。
- –0 加–0，结果为–0。

不过，如果某个运算数是字符串，那么采用下列规则：

- 如果两个运算数都是字符串，把第二个字符串连接到第一个字符串上。

❑ 如果只有一个运算数是字符串，把另一个运算数转换成字符串，结果是两个字符串连接成的字符串。

例如：

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

这段代码说明了加法运算符的两种模式之间的差别。正常情况下，5+5 等于 10（原始数值），如上面代码段中的前两行代码所示。不过，如果把一个运算数改为字符串"5"，那么结果将变为"55"（原始的字符串值），因为另一个运算数也会被转换成字符串。

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

2. **减法运算符**

减法运算符(-)是另一个十分常用的运算符：

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

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

❑ 如果两个运算数都是数字，将执行算术减法，返回结果。

❑ 某个运算数是 NaN，结果为 NaN。

❑ Infinity 减 Infinity，结果为 NaN。

❑ -Infinity 减-Infinity，结果为 NaN。

❑ Infinity 减-Infinity，结果为 Infinity。

- -Infinity 减 Infinity，结果为-Infinity。
- +0 减+0，结果为+0。
- –0 减–0，结果为–0。
- –0 减–0，结果为+0。
- 某个运算数不是数字，结果为 NaN。

2.9.6 关系运算符

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

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

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

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

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

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

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

另一种棘手的状况发生在比较两个字符串形式的数字时，例如：

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

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

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

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

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

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

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

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

第 2 章 ECMAScript 基础

2.9 运算符（4）

2.9.7 等性运算符

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

1. 等号和非等号

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

执行类型转换的基本规则如下：

- 如果一个运算数是 Boolean 值，在检查相等性之前，把它转换成数字值。false 转换成 0，true 转换成 1。

- 如果一个运算数是字符串，另一个是数字，在检查相等性之前，要尝试把字符串转换成数字。

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

- 如果一个运算数是对象，另一个是数字，在检查相等性之前，要尝试把对象转换成数字。

在进行比较时，该运算符还遵守下列规则：

- 值 null 和 undefined 相等。

- 在检查相等性时，不能把 null 和 undefined 转换成其他值。

- 如果某个运算数是 NaN，等号将返回 false，非等号将返回 true。重要提示：即使两个运算数都是 NaN，等号仍然返回 false，因为根据规则，NaN 不等于 NaN。

- 如果两个运算数都是对象，那么比较的是它们的引用值。如果两个运算数指向同一个对象，那么等号返回 true，否则两个运算数不等。

下表列出了一些特殊情况及它们的结果：

表 达 式	值
null = = undefined	true
"NaN" = = NaN	false
5 = = NaN	false
NaN = = NaN	false
NaN != NaN	true
false = =0	true
true = =1	true
true = = 2	false
undefined = = 0	false
null = = 0	false
"5" = = 5	true

2. 全等号和非全等号

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

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

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

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

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

这里，第一个警告使用非等号，把字符串"55"转换成数字 55，使得它与第二个运算数 55 相等。因此，计算结果为 false，因为两个运算数被看作是相等的。第二

个警告使用的是非全等号。该运算是在问：“sNum 与 iNum 不同吗？”这个问题的答案是：“是的（true）”，因为 sNum 是字符串，而 iNum 是数字，它们当然不同。

2.9.8 条件运算符

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

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

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

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

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

2.9.9 赋值运算符

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

```
var iNum = 10;
```

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

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

可以用一个复合赋值运算符改写第二行代码：

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

每种主要的算术运算以及其他几个运算都有复合赋值运算符：

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

2.9.10 逗号运算符

用逗号运算符可以在一条语句中执行多个运算。例如：

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

逗号运算符最常用于变量声明中。

第 2 章 ECMAScript 基础

2.10 语句

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

2.10.1 if 语句

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

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

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

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

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

还可以串联使用多个 if 语句，如下所示：

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

例如：

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

}
```

2.10.2 迭代语句

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

1. do-while 语句

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

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

例如：

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

2. while 语句

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

```
while(expression) statement
```

例如：

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

3. for 语句

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

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

例如：

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

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

4. for-in 语句

for-in 语句是严格的迭代语句，用于枚举对象的属性。语法如下：

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

例如：

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

这里，for-in 语句用于显示 BOM window 对象的所有属性。前面讨论过的方法 property- IsEnumerable()是 ECMAScript 中专门用于说明属性是否可以用 for-in 语句访问的方法。

2.10.3 有标签的语句

可以用下列语法给语句加标签，以便以后调用：

```
label: statement
```

例如：

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

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

2.10.4 break 语句和 continue 语句

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

```
var iNum = 0;

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

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

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

```
var iNum = 0;

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

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

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

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

```
var iNum = 0;

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

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

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

```
var iNum = 0;

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

        }
        iNum++;
    }
}

alert(iNum);    //outputs "95"
```

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

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

2.10.5 with 语句

with 语句用于设置代码在特定对象中的作用域。它的语法如下：

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

例如：

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

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

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

2.10.6 switch 语句

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

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

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

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

switch 语句主要是为避免让开发者编写下面这种代码：

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

等价的 switch 语句如下：

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

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

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

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

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

第 2 章 ECMAScript 基础

2.11 函数

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

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

例如：

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

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

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

这段代码生成的警告窗口如图 2-8 所示。

图 2-8

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

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

下面的代码把 sum 函数返回的值赋予一个变量：

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

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

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

一个函数中可以有多个 return 语句，如下所示：

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

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

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

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

    alert(sMessage);
}
```

这段代码中，如果消息不等于字符串"bye"，就永远都不会显示警告消息。

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

2.11.1 无重载

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

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

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

doAdd(10);
```

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

2.11.2 arguments 对象

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

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

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

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

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

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

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

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

用 arguments 对象判断传递给函数的参数个数，即可模拟函数重载：

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

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

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

2.11.3 Function 类

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

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

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

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

还可以如下定义它：

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

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

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

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

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

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

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

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

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

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

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

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

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

图 2-9

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

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

因为函数是引用类型，所以它们也有属性和方法。ECMAScript 定义的属性 length 声明了函数期望的参数个数。例如：

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

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

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

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

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

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

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

这段代码输出了 doAdd()函数的文本（如图 2-9 所示）。

还有两个 Function 类的方法与对象的讨论相关，下一章将讨论它们。

2.11.4 闭包

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

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

function sayHelloWorld() {
    alert(sMessage);
}

sayHelloWorld();
```

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

在一个函数中定义另一个函数会使闭包变得更复杂，如下所示：

```
var iBaseNum = 10;

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

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

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

第 2 章 ECMAScript 基础

2.12 小结

这一章介绍了 ECMAScript 的基础：

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

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

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

第 3 章 对象基础

3.1 面向对象术语

ECMAScript 对象是 JavaScript 比较特殊的特性之一。第 2 章介绍过一切都是对象（包括函

数）的概念。本章的重点是如何操作及使用这些对象，以及如何创建自己的对象，以根据需要增加专用的功能。

3.1 面向对象术语

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

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

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

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

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

3.1.1 面向对象语言的要求

一种面向对象语言需要向开发者提供四种基本能力：

(1) 封装——把相关的信息（无论数据或方法）存储在对象中的能力。

(2) 聚集——把一个对象存储在另一个对象内的能力。

(3) 继承——由另一个类（或多个类）得来类的属性和方法的能力。

(4) 多态——编写能以多种方法运行的函数或方法的能力。

ECMAScript 支持这些要求，因此可被看作面向对象的。

3.1.2 对象的构成

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

第 3 章 对象基础

3.2 对象应用

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

3.2.1 声明和实例化

对象是用关键字 new 后跟要实例化的类的名字创建的：

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

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

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

3.2.2 对象引用

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

3.2.3 对象废除

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

把对象的所有引用都设置为 null，可以强制性的废除对象。例如：

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

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

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

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

3.2.4 早绑定和晚绑定

所谓绑定（binding），即把对象的接口与对象实例结合在一起的方法。

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

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

第 3 章 对象基础

3.3 对象的类型：本地对象(1)

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

3.3.1 本地对象

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

```
Object          Function        Array           String
Boolean         Number          Date            RegExp
Error           EvalError       RangeError      ReferenceError
SyntaxError     TypeError       URIError
```

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

1. Array 类

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

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

如果预先知道数组中项的个数，可以用参数传递数组的大小：

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

使用这两个方法，一点要使用括号，与它们在 Java 中的用法相似：

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

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

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

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

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

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

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

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

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

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

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

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

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

还可以用字面量表示定义 Array 对象，即使用方括号（[和]），用逗号分隔值。例如，可以用下面的形式重写前面的例子：

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

注意，在这个例子中，未明确使用 Array 类。方括号暗示把其中的值存放在 Array 对象中。用这种方式声明的数组与用传统方式声明的数组相同。

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

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

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

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

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

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

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

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

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

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

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

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

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

的 Array 对象（包括原始数组中的项和新的项）。例如：

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

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

第 3 章 对象基础

3.3 对象的类型：本地对象(2)

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

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

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

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

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

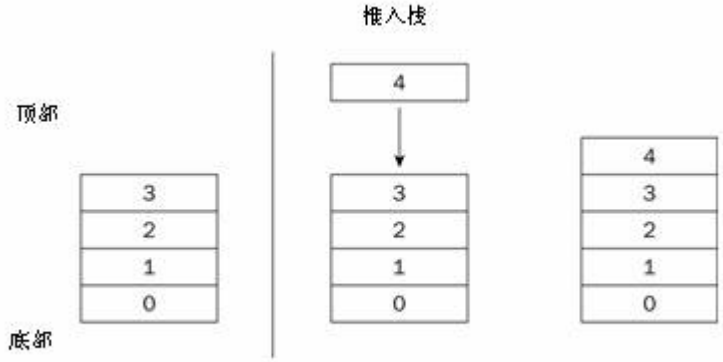

图 3-1

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

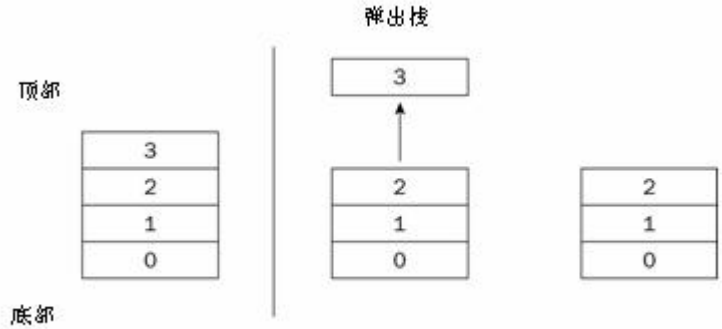

图 3-2

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

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

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

有项都被加入。然后，调用 pop()方法，它只返回最后一项"yellow"，这个项被存入变量 vItem。然后数组只剩下两个字符串，即"red"和"green"。

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

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

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

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

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

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

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

结果

Put

开始

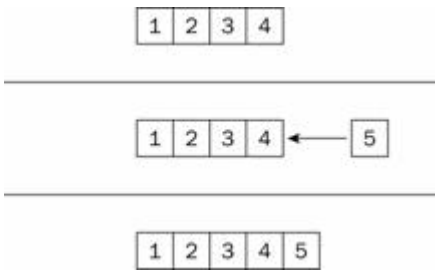

图 3-3

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

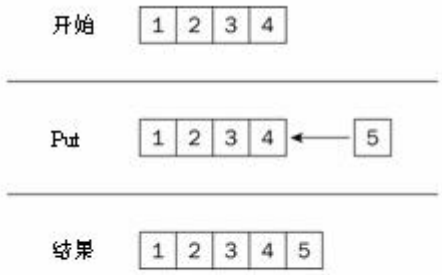

图 3-3

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

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

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

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

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

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

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

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

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

在对数字 3、32、2 和 5 进行排序时，方法 sort()把数组项的顺序调整为 2、3、32 和 5。如前所述，出现这种情况的是因为，数字是被转换成字符串，然后再按照字符代码进行比较的。这个问题可以克服。我将在第 12 章中对其进一步讨论。

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

- 删除——只需要声明两个参数，就可以从数组中删除任意多个项，这两个参数是要删除的第一个项的位置和要删除的项的个数。例如 arr.splice(0,2)将删除数组 arr 中的前两项。

- 替换而不删除——声明三个参数就可以把数据项插入指定的位置，这三个参数是起始位置、0（要删除的数组项的个数）和要插入的项。此外，还可以用第四个、第五个或更多个参数指定其他要删除的项。例如，arr.splice(2,0. "red", "green")将在位置 2 处插入"red"和"green"。

- 替换并删除——声明三个参数就可以把数据项插入指定的位置，这三个参数是起始位置、要删除的数组项的个数以及要插入的项。此外，还可以指定要插入的更多的项。要插入的项的个数不必等于删除的项的个数。例如，arr.splice(2,1, "red","green")将删除数组 arr 中位置 2 处的项，然后在位置 2 处插入"red"和"green"。

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

2. Date 类

ECMAScript 中的 Date 类基于 Java 中的 java.util.Date 类的早期版本。与 Java 一样，ECMAScript 把日期存储为距离 UTC 时间 1970 年 1 月 1 日凌晨 12 点的毫秒数。UTC 是 Universal Time Code，即通用时间代码（也叫做 Greenwich Mean Time，格林尼治标准时间）的缩写，是所有时区的基准标准时间。以毫秒数存储时间可以确保 Java 和 ECMAScript 免受恐怖的“千年虫”问题的侵害，该问题在

20 世纪 90 年代后期影响了许多较老的大型计算机。计算机可以精确地表示出 1970 年 1 月 1 日之前及之后 285 616 年的日期,这意味着除非你可以活过 200000 年，否则日期存储不成问题。

用下面代码可以创建新的 Date 对象：

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

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

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

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

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

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

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

如果传递给 parse()方法的字符串不能转换成日期，该函数返回 NaN。

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

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

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

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

创建日期的第二种方法是直接声明 UTC()方法接受的参数：

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

声明参数的顺序相同，（除了年和月外）它们不必都出现。

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

还有其他几个用于创建特定日期的字符串表示的方法：

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

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

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

还可用 getTimezoneOffset()方法判断时区使用的是否是夏令时。实现这一点需要创建任意年份的 1 月 1 日的日期，然后创建该年份的 7 月 1 日的日期，比较时区偏移量。如果分钟数不等，说明该时区使用的是夏令时，如果相等，则该时区使用的不是夏令时。

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

Date 类其余的方法（列在下表中）均用于设置或获取日期值的某部分。

方　法	说　明

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

（续）

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

第 3 章　对象基础

3.3　对象的类型：内置对象

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

1. Global 对象

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

```
var pointer = Global;
```

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

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

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

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

这段代码输出两个值：

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

可以看到，除空格外，第一个 URI 无任何改变，空格被替换为%20。第二个 URI 中的所有非字母数字字符都被替换成它们对应的编码，基本上使这个 URI 变得无用。这就是 encodeURI()可以处理完整 URI，而 encodeURIComponent()只能处理附加在已有 URI 末尾的字符串的原因。

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

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

这段代码输出两个值：

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

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

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

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

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

这行代码的功能等价于下面的代码：

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

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

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

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

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

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

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

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

属　性	说　明
undefined	Undefined 类型的字面量
NaN	非数的专用数值
Infinity	无穷大值的专用数值
Object	Object 的构造函数
Array	Array 的构造函数
Function	Function 的构造函数

Boolean	Boolean 的构造函数
String	String 的构造函数
Number	Number 的构造函数
Date	Date 的构造函数
RegExp	RegExp 的构造函数
Error	Error 的构造函数
EvalError	EvalError 的构造函数
RangeError	RangeError 的构造函数
ReferenceError	ReferenceError 的构造函数
SyntaxError	SyntaxError 的构造函数
TypeError	TypeError 的构造函数
URIError	URIError 的构造函数

2. Math 对象

Math 对象是在高中数学课就学过的内置对象。它知道解决最复杂的数学问题的所有公式，如果给它要处理的数字，即能计算出结果。

Math 对象有几个属性，主要是数学界的专用值。下表类出了这些属性：

属　性	说　明
E	值 e，自然对数的底
LN10	10 的自然对数
LN2	2 的自然对数
LOG2E	以 2 为底 E 的对数
LOG10E	以 10 为底 E 的对数
PI	值 π
SQRT1_2	1/2 的平方根
SQRT2	2 的平方根

虽然这些值的意义与用法不在本书讨论范围内，但如果清楚它们是什么，在需要时，即可使用它们。

Math 对象还包括许多专门用于执行简单的及复杂的数学计算的方法。

方法 min()和 max()用于判断一组数中的最大值和最小值。这两个方法都可接受任意多个参数：

```
var iMax = Math.max(3, 54, 32, 16);
alert(iMax);    //outputs "54"
```

```
var iMin = Math.min(3, 54, 32, 16);
alert(iMin);    //outputs "3"
```

对于数字 3、54、32 和 16，max()返回 54，min()返回 3。用这些方法，可免去用循环或 if 语句来判断一组数中的最大值。

另一个方法 abs()返回数字的绝对值。绝对值是负数的正值版本（正数的绝对值就是它自身）。

```
var iNegOne = Math.abs(-1);
alert(iNegOne);    //outputs "1"
var iPosOne = Math.abs(1);
alert(iPosOne);    //outputs "1"
```

这个例子中，abs(-1)返回 1，abs(1)也返回 1。

下一组方法用于把小数舍入成整数。处理舍入操作的方法有三个，即 ceil()、floor()和 round()，它们的处理方法不同：

- 方法 ceil()表示向上舍入函数，总是把数字向上舍入到最接近的值。
- 方法 floor()表示向下舍入函数，总是把数字向下舍入到最接近的值。
- 方法 round()表示标准的舍入函数，如果数字与下一个整数的差不超过 0.5，则向上舍入，否则向下舍入。这是在初中学过的舍入规则。

为说明每种方法的处理方式，考虑使用值 25.5：

```
alert(Math.ceil(25.5));     //outputs "26"
alert(Math.round(25.5));    //outputs "26"
alert(Math.floor(25.5));    //outputs "25"
```

对于 ceil()和 round()，传递 25.5，返回的是 26，而 floor()返回的是 25。注意不要交替使用这些方法，因为最后可能得到与预期不符的结果。

另一组方法与指数的用法有关。这些方法包括 exp()，用于把 Math.E 升到指定的幂；log()用于返回特定数字的自然对数；pow()用于把指定的数字升到指定的幂；sqrt()用于返回指定数字的平方根。

方法 exp()和 log()本质上功能相反，exp()把 Math.E 升到特定的幂，log()则判断 Math.E 的多少次指数才等于指定的值。例如：

```
var iNum = Math.log(Math.exp(10));
alert(iNum);
```

这里，首先用 exp()把 Math.E 升到 10 次幂，然后 log()返回 10，即等于数字 iNum 必需的指数。很多人都对此感到迷茫。全世界的高中生和数学系的大学生都被此类问题难倒过。如果你对自然对数一无所知，那么有可能永远都不需要为它编写代码。

方法 pow()用于把数字升到指定的幂，如把 2 升到 10 次幂（在数学中表示为 2^{10}）：

```
var iNum = Math.pow(2, 10);
```

pow()的第一个参数是基数，此例子中是 2。第二个参数是要升到的幂，此例子中是 10。

不建议把 Math.E 作为 pow()方法的基数。最好使用 exp()对 Math.E 进行升幂运算，因为它是专用运算，计算出的值更精确。

这组方法中的最后一个方法是 sqrt()，用于返回指定数字的平方根。它只有一个参数，即要求平方根的数字。要求 4 的平方根，只需要用一行代码：

```
var iNum = Math.sqrt(4);
alert(iNum);    //outputs "2"
```

当然，4 的平方根是 2，就是这行代码的输出。

你也许会问“为什么平方根必须利用指数”？实际上，数字平方根就是它的 1/2 次幂。例如，$2^{1/2}$就是 2 的平方根。

Math 对象还有一整套三角函数方法。下表列出了这些方法：

方　法	说　明
acos(x)	返回 x 的反余弦值
asin(x)	返回 x 的反正弦值

atan(x)	返回 x 的反正切值
atan2(y,x)	返回 y/x 的反余弦值
cos(x)	返回 x 的余弦值
sin(x)	返回 x 的正弦值
tan(x)	返回 x 的正切值

即使这些方法是 ECMA-262 定义的，结果也是由实现决定的，因为每个值的计算方法都有很多，从而使得不同的实现生成的结果的精度也不同。

Math 对象的最后一个方法是 random()，该方法返回一个 0 到 1 之间的随机数，不包括 0 和 1。这是在主页上显示随机引述或新闻的站点常用的工具。可用下面的形式调用 random()方法，在某个范围内选择随机数：

```
number = Math.floor(Math.random() * total_number_of_choices + first_possible_value)
```

这里使用方法 floor()，因为 random()返回的都是小数值，也就是说，用它乘以一个数，然后再加上一个数，得到的仍然是小数值。通常你想选择一个随机整数值。因此，必须使用 floor()方法。如果想选择一个 1 到 10 之间的数，代码如下：

```
var iNum = Math.floor(Math.random() * 10 + 1);
```

可能出现的值有 10 个（1 到 10），这些值中的第一个是 1。如果想选择 2 到 10 之间的值，代码如下：

```
var iNum = Math.floor(Math.random() * 9 + 2);
```

从 2 到 10，只有 9 个数字，所以选项总数为 9，其中第一个值是 2。许多时候，使用计算选项总数的函数和第一个可用的值更容易些：

```
function selectFrom(iFirstValue, iLastValue) {
    var iChoices = iLastValue - iFirstValue + 1;
    return Math.floor(Math.random() * iChoices + iFirstValue);
}

//select from between 2 and 10
var iNum = selectFrom(2, 10);
```

使用函数，可很容易地选择 Array 中的随机项：

```
var aColors = ["red", "green", "blue", "yellow", "black", "purple", "brown"];
var sColor = aColors[selectFrom(0, aColors.length-1)];
```

这里，selectFrom()函数的第二个参数是数组的长度减 1，即数组中最后一个元素的位置。

第 3 章 对象基础

3.3 对象的类型：宿主对象

所有非本地对象都是宿主对象（host object），即由 ECMAScript 实现的宿主环境提供的对象。所有 BOM 和 DOM 对象都是宿主对象，本书将在后面的章节讨论它们。

第 3 章 对象基础

3.4 作用域

任何程序设计语言的程序员都懂得作用域的概念，即某些变量的适用范围。

3.4.1 公用、受保护和私有作用域

在传统的面向对象程序设计中，主要关注于公用和私有作用域。公用作用域中的对象属性可以从对象外部访问，即开发者创建对象的实例后，就可使用它的公用属性。而私有作用域中的属性只能在对象内部访问，即对于外部世界来说，这些属性并不存在。这也意味着如果类定义了私有属性和方法，则它的子类也不能访问这些属性和方法。

最近，另一种作用域流行起来，即受保护作用域。虽然在不同语言中，受保护作用域的应用的规则不同，但一般说来，它都用于定义私有的属性和方法，只是这些属性和方法还能被其子类访问。

对 ECMAScript 讨论这些作用域几乎毫无意义，因为 ECMAScript 中只存在一种作用域——公用作用域。ECMAScript 中的所有对象的所有属性和方法都是公用的。因此，定义自己的类和对象时，必须格外小心。记住，所有属性和方法默认都是公用的。

许多开发者都在网上提出了有效的属性作用域模式，解决了 ECMAScript 的这种问题。由于缺少私有作用域，开发者们制定了一个规约，说明哪些属性和方法应该被看作私有的。这种规约规定在属性名前后加下划线。例如：

```
obj.__color__ = "red";
```

这段代码中，属性 color 是私有的。记住，这些下划线并不改变这些属性是公用属性的事实，它只是告诉其他开发者，应该把该属性看作私有的。

有些开发者还喜欢用单下划线说明私有成员，例如 obj._color。

3.4.2 静态作用域并非静态的

静态作用域定义的属性和方法任何时候都能从同一个位置访问。在 Java 中，类可具有静态属性和方法，无需实例化该类的对象，即可访问这些属性和方法，例如 java.net.URLEncoder 类，它的函数 encode()即是静态方法。

严格说来，ECMAScript 并没有静态作用域。不过，它可以给构造函数提供属性和方法。还记得吗，构造函数只是函数。函数是对象，对象可以有属性和方法。例如：

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

sayHi.alternate = function() {
    alert("hola");
};

sayHi();              //outputs "hi"
sayHi.alternate();    //outputs "hola"
```

这里，方法 alternate()实际上是函数 sayHi 的方法。可以像调用常规函数一样调用 sayHi()输出"hi"，也可以调用 sayHi.alternate()输出"hola"。即使如此，alternate()也是 sayHi()公用作用域中的方法，而不是静态方法。

3.4.3 关键字 this

在 ECMAScript 中，要掌握的最重要的概念之一是关键字 this 的用法，它用在对象的方法中。关键字 this 总是指向调用该方法的对象，例如：

```
var oCar = new Object;
oCar.color = "red";
oCar.showColor = function () {
    alert(this.color);    //outputs "red"
};
```

这里，关键字 this 用在对象的 showColor()方法中。在此环境中，this 等于 car，下面的代码与上面代码的功能相同：

```
var oCar = new Object;
oCar.color = "red";
oCar.showColor = function () {
    alert(oCar.color);    //outputs "red"
};
```

那么为什么使用 this 呢？因为在实例化对象时，总是不能确定开发者会使用什么样的变量名。使用 this，即可在任意多个地方重用同一个函数。考虑下面的例子：

```
function showColor() {
    alert(this.color);
}

var oCar1 = new Object;
oCar1.color = "red";
oCar1.showColor = showColor;

var oCar2 = new Object;
oCar2.color = "blue";
oCar2.showColor = showColor;

oCar1.showColor();    //outputs "red"
oCar2.showColor();    //outputs "blue"
```

在这段代码中，首先用 this 定义函数 showColor()，然后创建两个对象（oCar1 和 oCar2），一个对象的 color 属性被设置为"red"，另一个对象的 color 属性被设置为"blue"。两个对象都被赋予了属性 showColor，指向原始的 showColor()函数（注意这里不存在命名问题，因为一个是全局函数，而另一个是对象的属性）。调用每个对象的 showColor()方法，oCar1 的输出"red"，而 oCar2 的输出"blue"。这是因为调用 oCar1.showColor()时，函数中的 this 关键字等于 oCar1，调用 oCar2.showColor()时，函数中的 this 关键字等于 oCar2。

注意，引用对象的属性时，必须使用 this 关键字。例如，如果采用下面的代码，showColor()方法不能运行：

```
function showColor() {
    alert(color);
}
```

如果不用对象或 this 关键字引用变量，ECMAScript 就会把它看作局部变量或全局变量。然后该函数将查找名为 color 的局部或全局变量，但是不会找到的。结果如何？该函数将在警告中显示"null"。

第 3 章 对象基础

3.5 定义类或对象(1)

使用预定义对象的能力只是面向对象语言的能力的一部分。真正的强大之处在于能够创建自己专用的类和对象。与 ECMAScript 中的许多特性一样，可以用各种方法实现这一点。

3.5.1 工厂方式

由于对象的属性可在对象创建后动态定义，所以许多开发者都在初次引入 JavaScript 时编写类似下面的代码：

```
var oCar = new Object;
oCar.color = "red";
oCar.doors = 4;
oCar.mpg = 23;
oCar.showColor = function () {
    alert(this.color);
};
```

在这段代码中，创建对象 car。然后给它设置几个属性：它的颜色是红色，有四个门，每加仑油 23 英里。最后一个属性实际上是指向函数的指针，意味着该属性是个方法。执行这段代码后，就可以使用对象 car。问题是可能需要创建多个 car 实例。

要解决此问题，开发者创造了能创建并返回特定类型的对象的工厂函数（factory function）。例如，函数 createCar()可用于封装前面列出的创建 car 对象的操作：

```
function createCar() {
    var oTempCar = new Object;
    oTempCar.color = "red";
    oTempCar.doors = 4;
    oTempCar.mpg = 23;
    oTempCar.showColor = function () {
        alert(this.color)
    };

    return oTempCar;
}

var oCar1 = createCar();
var oCar2 = createCar();
```

这里，前面的所有代码都包含在 createCar()函数中，此外还有一行额外的代码，返回 car 对象作为（oTempCar）函数值。调用此函数时，将创建新对象，并赋予它所有必要的属性，复制出一个前面说明的 car 对象。使用该方法，可以容易地创建 car 对象的两个版本（oCar1 和 oCar2），它们的属性完全一样。当然，还可以修改 createCar()函数，给它传递各个属性的默认值，而不是赋予属性默认值：

```
function createCar(sColor, iDoors, iMpg) {
    var oTempCar = new Object;
    oTempCar.color = sColor;
    oTempCar.doors = iDoors;
    oTempCar.mpg = iMpg;
    oTempCar.showColor = function () {
        alert(this.color)
    };

    return oTempCar;
}

var oCar1 = createCar("red", 4, 23);
var oCar2 = createCar("blue", 3, 25);
oCar1.showColor();    //outputs "red"
oCar2.showColor();    //outputs "blue"
```

给 createCar()函数加上参数，即可为要创建的 car 对象的 color、doors 和 mpg 属性赋值。这使两个对象具有相同的属性，却有不同的属性值。

虽然 ECMAScript 越来越正式化，创建对象的方法却被置之不理，且其规范化至今还遭人反对。一部分是语义上的原因（它看起来不像使用带有构造函数的 new 运算符那么正规），一部分是功能上的原因。功能问题在于用这种方式必须创建对象的方法。前面的例子中，每次调用函数 createCar()，都要创建新函数 showColor()，意味着每个对象都有自己的 showColor()版本，事实上，每个对象都共享了同一个函数。

有些开发者在工厂函数外定义对象的方法，然后通过属性指向该方法，从而避开这个问题：

```
function showColor() {
    alert(this.color);
}

function createCar(sColor, iDoors, iMpg) {
    var oTempCar = new Object;
    oTempCar.color = sColor;
    oTempCar.doors = iDoors;
    oTempCar.mpg = iMpg;
    oTempCar.showColor = showColor;
    return oTempCar;
}

var oCar1 = createCar("red", 4, 23);
var oCar2 = createCar("blue", 3, 25);
oCar1.showColor();    //outputs "red"
oCar2.showColor();    //outputs "blue"
```

在这段重写的代码中，在函数 createCar()前定义了函数 showColor()。在 createCar()内部，赋予对象一个指向已经存在的 showColor()函数的指针。从功能上讲，这样解决了重复创建函数对象的问题，但该函数看起来不像对象的方法。

所有这些问题引发了开发者定义的构造函数的出现。

3.5.2 构造函数方式

创建构造函数就像定义工厂函数一样容易。第一步选择类名，即构造函数的名字。根据惯例，这个名字的首字母大写，以使它与首字母通常是小写的变量名区分开。除了这点不同，构造函数看起来很像工厂函数。考虑下面的例子：

```
function Car(sColor, iDoors, iMpg) {
    this.color = sColor;
    this.doors = iDoors;
    this.mpg = iMpg;
    this.showColor = function () {
        alert(this.color)
    };
}

var oCar1 = new Car("red", 4, 23);
var oCar2 = new Car("blue", 3, 25);
```

你可能已经注意到第一个差别了，在构造函数内部无创建对象，而是使用 this 关键字。使用 new 运算符调用构造函数时，在执行第一行代码前先创建一个对象，只有用 this 才能访问该对象。然后可以直接赋予 this 属性，默认情况下是构造函数的返回值（不必明确使用 return 运算符）。

现在，用 new 运算符和类名 car 创建对象，就更像创建 ECMAScript 中一般对象了。你也许会问，这种方式在管理函数方面是否存在与前一种方式相同的问题呢？是的。

就像工厂函数，构造函数会重复生成函数，为每个对象都创建独立的函数版本。不过，与工厂函数相似，也可以用外部函数重写构造函数，同样的，语义上无任何意义。这就是原型方式的优势所在。

第 3 章 对象基础

3.5 定义类或对象(2)

3.5.3 原型方式

该方式利用了对象的 prototype 属性，可把它看成创建新对象所依赖的原型。这里，用空构造函数来设置类名。然后所有的属性和方法都被直接赋予 prototype 属性。重新前面的例子，代码如下所示：

```
function Car() {
}

Car.prototype.color = "red";
Car.prototype.doors = 4;
Car.prototype.mpg = 23;
Car.prototype.showColor = function () {
    alert(this.color);
};

var oCar1 = new Car();
var oCar2 = new Car();
```

在这段代码中，首先定义构造函数（Car），其中无任何代码。接下来的几行代码，通过给 Car 的 prototype 属性添加属性定义 Car 对象的属性。调用 new Car()时，原型的所有属性都被立即赋予要创建的对象，意味着所有 Car 实例存放的都是指向 showColor()函数的指针。从语义上讲，所有属性看起来都属于一个对象，因此解决了前面两种方式的两个问题。此外，使用该方法，还能用 instanceof 运算符检查给定变量指向的对象的类型。因此，下面的代码将输出 true：

```
alert(oCar1 instanceof Car);   //outputs "true"
```

看起来是个非常好的解决方案。遗憾的是，并非尽如人意。

首先，这个构造函数没有参数。使用原型方式时，不能通过给构造函数传递参数初始化属性的值，因为 car1 和 car2 的 color 属性都等于"red"，doors 属性都等于 4，mpg 属性都等于 23。这意味必须在对象创建后才能改变属性的默认值，这点很令人讨厌，但还不至于是世界末日。真正的问题出现在属性指向的是对象，而不是函数时。函数共享不会造成任何问题，但对象却很少被多个实例共享的。考虑下面的例子：

```
function Car() {
}
```

```
Car.prototype.color = "red";
Car.prototype.doors = 4;
Car.prototype.mpg = 23;
Car.prototype.drivers = new Array("Mike", "Sue");
Car.prototype.showColor = function () {
    alert(this.color);
};

var oCar1 = new Car();
var oCar2 = new Car();

oCar1.drivers.push("Matt");

alert(oCar1.drivers);    //outputs "Mike,Sue,Matt"
alert(oCar2.drivers);    //outputs "Mike,Sue,Matt"
```

这里，属性 drivers 是指向 Array 对象的指针，该数组中包含两个名字"Mike"和"Sue"。由于 drivers 是引用值，Car 的两个实例都指向同一个数组。这意味着给 car1.drivers 添加值"Matt"，在 car2.drivers 中也能看到。输出这两个指针中的任何一个，结果都是显示字符串"Mike,Sue,Matt"。

由于创建对象时有这么多问题，你一定会想，是否有种合理的创建对象的方法呢？答案是联合使用构造函数和原型方式。

3.5.4 混合的构造函数/原型方式

联合使用构造函数和原型方式，就可像用其他程序设计语言一样创建对象。这种概念非常简单，即用构造函数定义对象的所有非函数属性，用原型方式定义对象的函数属性（方法）。结果所有函数都只创建一次，而每个对象都具有自己的对象属性实例。再重写前面的例子，代码如下：

```
function Car(sColor, iDoors, iMpg) {
    this.color = sColor;
    this.doors = iDoors;
    this.mpg = iMpg;
    this.drivers = new Array("Mike", "Sue");
}

Car.prototype.showColor = function () {
    alert(this.color);
};

var oCar1 = new Car("red", 4, 23);
var oCar2 = new Car("blue", 3, 25);

oCar1.drivers.push("Matt");

alert(oCar1.drivers);    //outputs "Mike,Sue,Matt"
alert(oCar2.drivers);    //outputs "Mike,Sue"
```

现在就更像创建一般对象了。所有的非函数属性都在构造函数中创建，意味着又可用构造函数的参数赋予属性默认值了。因为只创建 showColor()函数的一个实例，所以没有内存浪费。此外，给 oCar1 的 drivers 数组添加"Matt"值，不会影响 oCar2 的数组，所以输出这些数组的值时，oCar1.drivers 显示的是"Mike,Sue,Matt"，而 oCar2.drivers 显示的是"Mike,Sue"。由于使用了原型方式，所以仍然能利用 instanceof 运算符判断对象的类型。

这种方式是 ECMAScript 主要采用的方式，它具有其他方式的特性，却没有它们的副作用。不过，有些开发者仍觉得这种方法不够完美。

3.5.5 动态原型方法

对于习惯使用其他语言的开发者来说，使用混合的构造函数/原型方式感觉不那么和谐。毕竟，定义类时，大多数面向对象语言都对属性和方法进行了视觉上的封装。考虑下面的 Java 类：

```
class Car {
    public String color = "red";
    public int doors = 4;
    public int mpg = 23;

    public Car(String color, int doors, int mpg) {
        this.color = color;
        this.doors = doors;
        this.mpg = mpg;
    }

    public void showColor() {
        System.out.println(color);
    }
}
```

Java 很好的打包了 Car 类的所有属性和方法，因此看见这段代码就知道它要实现什么功能，它定义了一个对象的信息。批评混合的构造函数/原型方式的人认为，在构造函数内存找属性，在其外部找方法的做法不合逻辑。因此，他们设计了动态原型方法，以提供更友好的编码风格。

动态原型方法的基本想法与混合的构造函数/原型方式相同，即在构造函数内定义非函数属性，而函数属性则利用原型属性定义。唯一的区别是赋予对象方法的位置。下面是用动态原型方法重写的 Car 类：

```
function Car(sColor, iDoors, iMpg) {
    this.color = sColor;
    this.doors = iDoors;
    this.mpg = iMpg;
    this.drivers = new Array("Mike", "Sue");

    if (typeof Car._initialized == "undefined") {

        Car.prototype.showColor = function () {
            alert(this.color);
        };

        Car._initialized = true;
    }
}
```

直到检查 typeof Car._initialized 是否等于"undefined"之前，这个构造函数都未发生变化。这行代码是动态原型方法中最重要的部分。如果这个值未定义，构造函数将用原型方式继续定义对象的方法，然后把 Car._initialized 设置为 true。

如果这个值定义了（它的值为 true 时，typeof 的值为 Boolean），那么就不再创建该方法。简而言之，该方法使用标志（_initialized）来判断是否已给原型赋予了任何方法。该方法只创建并赋值一次，为取悦传统的 OOP 开发者，这段代码看起来更像其他语言中的类定义了。

3.5.6 混合工厂方式

这种方式通常是在不能应用前一种方式时的变通方法。它的目的是创建假构造函数，只返回另一种对象的新实例。这段代码看来与工厂函数非常相似：

```
function Car() {
    var oTempCar = new Object;
    oTempCar.color = "red";
    oTempCar.doors = 4;
    oTempCar.mpg = 23;
    oTempCar.showColor = function () {
        alert(this.color)
    };

    return oTempCar;
}
```

与经典方式不同，这种方式使用 new 运算符，使它看起来像真正的构造函数：

```
var car = new Car();
```

由于在 Car()构造函数内部调用了 new 运算符，所以将忽略第二个 new 运算符（位于构造函数之外）。在构造函数内部创建的对象被传递回变量 var。

这种方式在对象方法的内部管理方面与经典方式有着相同的问题。强烈建议：除非万不得已（请参阅第 15 章），还是避免使用这种方式。

3.5.7 采用哪种方式

如前所述，目前使用最广泛的是混合的构造函数/原型方式。此外，动态原型方法也很流行，在功能上与构造函数/原型方式等价。可以采用这两种方式中的任何一种。不过不要单独使用经典的构造函数或原型方式，因为这样会给代码引入问题。

3.5.8 实例

对象令人感兴趣的一点是用它们解决问题的方式。ECMAScript 中最常见的一个问题是字符串连接的性能。与其他语言类似，ECMAScript 的字符串是不可变的，即它们的值不能改变。考虑下面的代码：

```
var str = "hello ";
str += "world";
```

实际上，这段代码在幕后执行的步骤如下：

(1) 创建存储"hello"的字符串。

(2) 创建存储"world"的字符串。

(3) 创建存储连接结果的字符串。

(4) 把 str 的当前内容复制到结果中。

(5) 把"world"复制到结果中。

(6) 更新 str，使它指向结果。

每次完成字符串连接都会执行步骤 2 到 6，使得这种操作非常消耗资源。如果重复这一过程几百次，甚至几千次，就会造成性能问题。解决方法是用 Array 对象存储字符串，然后用 join()方法（参数是空字符串）创建最后的字符串。想像用下面的代码代替前面的代码：

```
var arr = new Array;
arr[0] = "hello ";
arr[1] = "world";
var str = arr.join("");
```

这样，无论在数组中引入多少字符串都不成问题，因为只在调用 join()方法时才会发生连接操作。此时，执行的步骤如下：

(1) 创建存储结果的字符串。

(2) 把每个字符串复制到结果中的合适位置。

虽然这种解决方法很好，但还有更好的方法。问题是这段代码不能确切反映出它的意图。要使它更容易理解，可以用 StringBuffer 类打包该功能：

```
function StringBuffer() {
    this.__strings__ = new Array;
}

StringBuffer.prototype.append = function (str) {
    this.__strings__.push(str);
};

StringBuffer.prototype.toString = function () {
    return this.__strings__.join("");
};
```

第一点要注意的是，这段代码是 strings 的属性，本意是私有属性。它只有两个方法，即 append()和 toString()方法。append()方法有一个参数，它把该参数附加到字符串数组中，toString()方法调用数组的 join()方法，返回真正连接成的字符串。要用 StringBuffer 对象连接一组字符串，可以用下面的代码：

```
var buffer = new StringBuffer();
buffer.append("hello ");
buffer.append("world");
var result = buffer.toString();
```

可用下面代码测试 StringBuffer 对象和传统的字符串连接方法的性能：

```
var d1 = new Date();
var str = "";
for (var i=0; i < 10000; i++) {
    str += "text";
}
var d2 = new Date();

document.write("Concatenation with plus: " + (d2.getTime() - d1.getTime()) + "
milliseconds");

var oBuffer = new StringBuffer();
d1 = new Date();
for (var i=0; i < 10000; i++) {
    oBuffer.append("text");
}
var sResult = buffer.toString();
d2 = new Date();

document.write("<br />Concatenation with StringBuffer: " + (d2.getTime() -
d1.getTime()) + " milliseconds");
```

这段代码对字符串连接进行两个测试，第一个使用加号，第二个使用 StringBuffer 类。每个操作都连接 10000 个字符串。日期值 d1 和 d2 用于判断完成操作需要的时间。记住，创建新 Date 对象时，如果没有参数，赋予对象的是当前的日期与时间。要计算连接操作历经多少时间，把日期的毫秒表示（getTime()方法的返回值）相减即可。这是衡量 JavaScript 性能的常用方法。该测试的结果应该说明使用 StringBuffer 类比使用加号节省了 100%~200%的时间。

第 3 章 对象基础

3.6 修改对象

创建对象只是使用 ECMAScript 的乐趣的一部分。你喜欢修改已有对象的行为吗？这在 ECMAScript 中是完全可能的，所以可为 String、Array、Number 或其他任意一种对象设计出你想要的任何方法，因为有无限的可能性。

还记得本章前面的小节中介绍的 prototype 属性吗？你已经知道，每个构造函数都有个 prototype 属性，可用于定义方法。你还不知道的是，在 ECMAScript 中，每个本地对象也有个用法完全相同的 prototype 属性。

3.6.1 创建新方法

可以用 prototype 属性为任何已有的类定义新方法，就像处理自己的类一样。例如，还记得 Number 类的 toString()方法吗，如果给它传递 16，它将输出十六进制的字符串。难道用 toHexString()方法处理这个操作不是更好吗？创建它很简单：

```
Number.prototype.toHexString = function () {
    return this.toString(16);
};
```

在此环境中，关键字 this 指向 Number 的实例，因此可完全访问 Number 的所有方法。有了这段代码，可实现下面操作：

```
var iNum = 15;
alert(iNum.toHexString());   //outputs "F"
```

由于数字 15 等于十六进制中的 F，因此警告将显示"F"。还记得将数组用作队列的讨论吗？唯一漏掉的是命名正确的方法。可以给 Array 类添加两个方法 enqueue()和 dequeue()，只让它们反复调用已有的 push()和 shift()方法即可：

```
Array.prototype.enqueue = function(vItem) {
    this.push(vItem);
};

Array.prototype.dequeue = function() {
    return this.shift();
};
```

当然，还可添加与已有方法无关的方法。例如，假设要判断某个项在数组中的位置，没有本地方法可以做这种事情。则可以轻松地创建下面的方法：

```
Array.prototype.indexOf = function (vItem) {

    for (var i=0; i < this.length; i++) {
        if (vItem == this[i]) {
            return i;
        }


    }
}

return -1;
}
```

该方法 indexOf()与 String 类的同名方法保持一致，在数组中检索每个项，直到发现与传进来的项相等的项为止。如果找到相等的项，则返回该项的位置，否则，返回-1。使用这种定义，可以采用下面的代码：

```
var aColors = new Array("red", "green", "yellow");
alert(aColors.indexOf("green"));    //outputs "1"
```

最后，如果想给 ECMAScript 中的每个本地对象添加新方法，必须在 Object 对象的 prototype 属性上定义它。如上一章所述，所有本地对象都继承了 Object 对象，所以对 Object 对象做任何改变，都会反应在所有本地对象中。例如，如果想添加一个用警告输出对象的当前值的方法，可以采用下面的代码：

```
Object.prototype.showValue = function () {
    alert(this.valueOf());
};

var str = "hello";
var iNum = 25;
str.showValue();    //outputs "hello"
iNum.showValue();    //outputs "25"
```

这里，String 和 Number 对象都从 Object 对象继承了 showValue()方法，分别在它们的对象上调用该方法，将显示"hello"和"25"。

3.6.2 重定义已有方法

就像能给已有的类定义新方法一样，也可重定义已有的方法。如前一章所述，函数名只是指向函数的指针，因此可以轻易地使它指向其他函数。如果修改了本地方法，如 toString()，会出现什么情况呢？

```
Function.prototype.toString = function () {
    return "Function code hidden";
};
```

前面的代码完全合法，运行结果完全符合预期：

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

alert(sayHi.toString());    //outputs "Function code hidden"
```

也许你还记得，第 2 章中介绍过 Function 的 toString()方法通常输出的是函数的源代码。覆盖该方法，可以返回另一个字符串（在这个例子中，返回"Function

code hidden"）。不过，toString()指向的原始函数怎样了呢？它将被无用存储单元回收程序回收，因为它被完全废弃了。没能够恢复原始函数的办法，所以在覆盖原始方法前，存储它的指针比较安全，以便以后的使用。你甚至可能在某种情况下在新方法中调用原始方法：

```
Function.prototype.originalToString = Function.prototype.toString;

Function.prototype.toString = function () {
    if (this.originalToString().length > 100) {
        return "Function too long to display.";
    } else {
        return this.originalToString();
    }
};
```

在这段代码中，第一行代码把对当前 toString()方法的引用保存在属性 originalTo- String 中。然后用定制的方法覆盖了 toString()方法。新方法将检查该函数源代码的长度是否大于 100。如果是，就返回错误消息，说明该函数代码太长，否则调用 originalToString()方法，返回函数的源代码。

3.6.3 极晚绑定

从技术上来说，根本不存在极晚绑定。本书采用该术语描述 ECMAScript 中的一种现象，即能够在对象实例化后再定义它的方法。例如：

```
var o = new Object;

Object.prototype.sayHi = function () {
    alert("hi");
};

o.sayHi();
```

在大多数程序设计语言中，必须在实例化对象之前定义对象的方法。这里，方法 sayHi()是在创建 Object 类的一个实例后才添加进来的。在传统语言中不仅没听说过这种操作，也没听说过该方法还会自动赋予 Object 的实例并能立即使用（接下来的一行）。

不建议使用极晚绑定方法，因为很难对其跟踪和记录。不过，还是应该了解这种可能。

第 3 章 对象基础

3.7 小结

ECMAScript 语言为 JavaScript 实现提供了完整的面向对象语言能力。在这一章中，学到了 ECMA-262 中定义的三种不同类型的对象：本地对象、内置对象和宿主对象。

探讨了 Array 对象和 Date 对象，学习了它们的方法、属性及各种古怪之处。此外还学习了两个内置对象，即 Global 和 Math 对象，并了解了 Global 对象与其他对象的区别。

这一章还介绍了完全定义自己的对象的能力，其中探讨了几种实现该能力的方法，讨论了它们的优点及缺点。

最后，学习了如何修改已有对象，加入新的方法或覆盖已有的方法。

下一章将结束对 JavaScript Core——ECMAScript 的介绍，讨论继承性的问题。

第 4 章 继承

4.1 继承机制实例

说明继承机制最简单的方法是，利用一个经典的例子——几何形状。实际上，几何形状只有两种，即椭圆形（是圆形的）和多边形（具有一定数量的边）。圆是椭圆形的一种，它只有一个焦点。三角形、矩形和五边形都是多边形的一种，具有不同数量的边。正方形是矩形的一种，所有的边等长。这就构成了一种完美的继承关系。

在这个例子中，形状（Shape）是椭圆形（Ellipse）和多边形（Polygon）的基类（base class）（所有类都由它继承而来）。椭圆具有一个属性 *foci*，说明椭圆具有的焦点的个数。圆形（Circle）继承了椭圆形，因此圆形是椭圆形的子类(subclass)，椭圆形是圆形的超类(superclass)。同样的，三角形(Triangle)、

矩形（Rectangle）和五边形（Pentagon）都是多边形的子类，多边形是它们的超类。最后，正方形（Square）继承了矩形。

最好用图来解释这种继承关系，这是 UML（统一建模语言）的用武之地。UML 的主要用途之一是，可视化地表示像继承这样的复杂对象关系。图 4-1 是解释 Shape 和它的子类之间关系的 UML 图示。

在 UML 中，每个方框表示一个类，由类名说明。Triangle、Rectangle 和 Pentagon 顶部的线段汇集在一起，指向 Shape，说明这些类都由 Shape 继承而来。同样的，从 Square 指向 Rectangle 的箭头说明了它们之间的继承关系。

如果有兴趣学习 UML，可以参考三位 UML 创始人所著的《UML 用户指南（第二版）》。

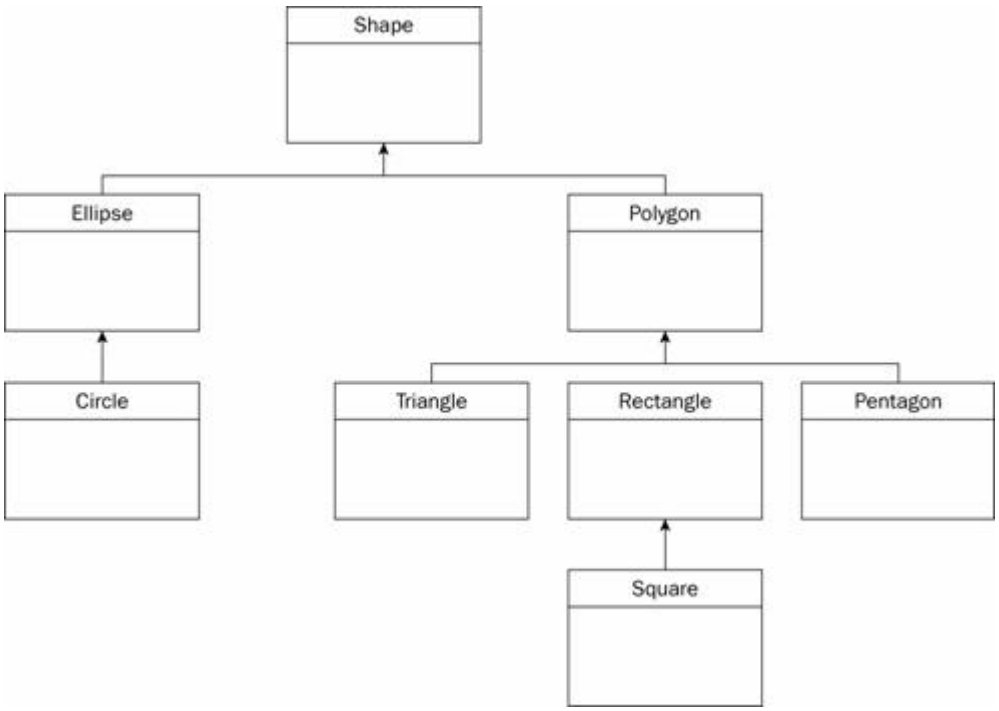

图　4-1

第 4 章　继承

.2 继承机制的实现(1)

要用 ECMAScript 实现继承机制，首先从基类入手。所有开发者定义的类都可作为基类。出于安全原因，本地类和宿主类不能作为基类，这样可以防止公用访问编译过的浏览器级的代码，因为这些代码可以被用于恶意攻击。

选定基类后，就可以创建它的子类了。是否使用基类完全由你决定。有时，你可能想创建一个不能直接使用的基类，它只是用于给子类提供通用的函数。在这种情况下，基类被看作抽象类。

尽管 ECMAScript 并没有像其他语言那样严格地定义抽象类，但有时它的确会创建一些不允许使用的类。通常，我们称这种类为抽象类。

创建的子类将继承超类的所有属性和方法，包括构造函数及方法的实现。记住，所有属性和方法都是公用的，因此子类可直接访问这些方法。子类还可添加超类中没有的新属性和方法，也可以覆盖超类中的属性和方法。

4.2.1 继承的方式

和其他功能一样，ECMAScript 中实现继承的方式不止一种。这是因为 JavaScript 中的继承机制并不是明确规定的，而是通过模仿实现的。这意味着所有的继承细节并非由解释程序处理。作为开发者，你有权决定最适用的继承方式。

1. 对象冒充

构想原始的 ECMAScript 时，根本没打算设计对象冒充（object masquerading）。它是在开发者开始理解函数的工作方式，尤其是如何在函数环境中使用 this 关键字后才发展出来的。

其原理如下：构造函数使用 this 关键字给所有属性和方法赋值（即采用类声明的构造函数方式）。因为构造函数只是一个函数，所以可使 ClassA 的构造函数成为 ClassB 的方法，然后调用它。ClassB 就会收到 ClassA 的构造函数中定义的属性和方法。例如，用下面的方式定义 ClassA 和 ClassB：

```
function ClassA(sColor) {
    this.color = sColor;
    this.sayColor = function () {
        alert(this.color);
    };
}

function ClassB(sColor) {
}
```

还记得吗？关键字 this 引用的是构造函数当前创建的对象。不过在这个方法中，this 指向的是所属的对象。这个原理是把 ClassA 作为常规函数来建立继承机制，而不是作为构造函数。如下使用构造函数 ClassB 可以实现继承机制：

```
function ClassB(sColor) {
    this.newMethod = ClassA;
    this.newMethod(sColor);
    delete this.newMethod;
}
```

在这段代码中，为 ClassA 赋予了方法 newMethod（记住，函数名只是指向它的指针）。然后调用该方法，传递给它的是 ClassB 构造函数的参数 sColor。最后一行代码删除了对 ClassA 的引用，这样以后就不能再调用它。

所有的新属性和新方法都必须在删除了新方法的代码行后定义。否则，可能会覆盖超类的相关属性和方法：

```
    function ClassB(sColor, sName) {
        this.newMethod = ClassA;
        this.newMethod(sColor);
        delete this.newMethod;

        this.name = sName;
        this.sayName = function () {
            alert(this.name);
        };
    }
```

为证明前面的代码有效，可以运行下面的例子：

```
var objA = new ClassA("red");
var objB = new ClassB("blue", "Nicholas");
objA.sayColor();    //outputs "red"
objB.sayColor();    //outputs "blue"
objB.sayName();     //outputs "Nicholas"
```

有趣的是，对象冒充可以支持多重继承。也就是说，一个类可以继承多个超类。用 UML 表示的多继承机制如图 4-2 所示。

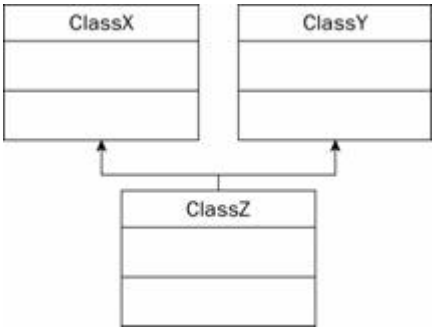

图 4-2

例如，如果存在两个类 ClassX 和 ClassY，ClassZ 想继承这两个类，可以使用下面的代码：

```
        function ClassZ() {
            this.newMethod = ClassX;
            this.newMethod();
            delete this.newMethod;

            this.newMethod = ClassY;
    this.newMethod();
    delete this.newMethod;
}
```

这里存在一个弊端，如果 ClassX 和 ClassY 具有同名的属性或方法，ClassY 具有高优先级，因为继承的是最后的类。除这点小问题之外，用对象冒充实现多继承机制轻而易举。

由于这种继承方法的流行，ECMAScript 的第三版为 Function 对象加入了两个新方法，即 call()和 apply()。

2. call()方法

call()方法是与经典的对象冒充方法最相似的方法。它的第一个参数用作 this 的对象。其他参数都直接传递给函数自身。例如：

```
function sayColor(sPrefix, sSuffix) {
    alert(sPrefix + this.color + sSuffix);
};

var obj = new Object();
obj.color = "red";

//outputs "The color is red, a very nice color indeed. "
sayColor.call(obj, "The color is ", ", a very nice color indeed. ");
```

在这个例子中，函数 sayColor()在对象外定义，即使它不属于任何对象，也可以引用关键字 this。对象 obj 的 color 属性等于"red"。调用 call()方法时，第一个参数是 obj，说明应该赋予 sayColor()函数中的 this 关键字值是 obj。第二个和第三个参数是字符串。它们与 sayColor()函数中的参数 prefix 和 suffix 匹配，最后生成的消息"The color is red, a very nice color indeed"将被显示出来。

要与继承机制的对象冒充方法一起使用该方法，只需将前三行的赋值、调用和删除代码替换即可：

```
function ClassB(sColor, sName) {
    //this.newMethod = ClassA;
    //this.newMethod(sColor);
    //delete this.newMethod;
    ClassA.call(this, sColor);

    this.name = sName;
    this.sayName = function () {
        alert(this.name);
    };
}
```

这里，想让 ClassA 中的关键字 this 等于新创建的 ClassB 对象，因此 this 是第一个参数。第二个参数 sColor 对两个类来说都是唯一的参数。

3. apply()方法

apply()方法有两个参数，用作 this 的对象和要传递给函数的参数的数组。例如：

```
function sayColor(sPrefix, sSuffix) {
    alert(sPrefix + this.color + sSuffix);
};

var obj = new Object();
obj.color = "red";

//outputs "The color is red, a very nice color indeed. "
sayColor.apply(obj, new Array("The color is ", ", a very nice color indeed."));
```

这个例子与前面的例子相同，只是现在调用的是 apply()方法。调用 apply()方法时，第一个参数仍是 obj，说明应该赋予 sayColor()中的 this 关键字值是 obj。第二个参数是由两个字符串构成的数组，与 sayColor()的参数 prefix 和 suffix 匹配。生成的消息仍是"The color is red, a very nice color indeed"，将被显示出来。

该方法也用于替换前三行的赋值、调用和删除新方法的代码：

```
function ClassB(sColor, sName) {
    //this.newMethod = ClassA;
    //this.newMethod(sColor);
    //delete this.newMethod;

    ClassA.apply(this, new Array(sColor));

    this.name = sName;
    this.sayName = function () {
        alert(this.name);
    };
}
```

同样的，第一个参数仍是 this。第二个参数是只有一个值 color 的数组。可以把 ClassB 的整个 arguments 对象作为第二个参数传递给 apply()方法：

```
function ClassB(sColor, sName) {
    //this.newMethod = ClassA;
    //this.newMethod(sColor);
    //delete this.newMethod;
    ClassA.apply(this, arguments);

    this.name = sName;
    this.sayName = function () {
        alert(this.name);
    };
}
```

当然，只有超类中的参数顺序与子类中的参数顺序完全一致时才可以传递参数对象。如果不是，就必须创建一个单独的数组，按照正确的顺序放置参数。此外，还可使用 call()方法。

4. 原型链

继承这种形式在 ECMAScript 中原本是用于原型链的。上一章介绍了定义类的原型方式。原型链扩展了这种方式，以一种有趣的方式实现继承机制。

在上一章中学过，prototype 对象是个模板，要实例化的对象都以这个模板为基础。总而言之，prototype 对象的任何属性和方法都被传递给那个类的所有实例。原型链利用这种功能来实现继承机制。

如果用原型方式重定义前面例子中的类，它们将变为下列形式：

```
function ClassA() {
}

ClassA.prototype.color = "red";
ClassA.prototype.sayColor = function () {
    alert(this.color);
};

function ClassB() {
}

ClassB.prototype = new ClassA();
```

原型链的神奇之处在于突出显示的代码行。这里，把 ClassB 的 prototype 属性设置成 ClassA 的实例。这很有意义，因为想要 ClassA 的所有属性和方法，但又不想逐个将它们赋予 ClassB 的 prototype 属性。还有比把 ClassA 的实例赋予 prototype 属性更好的方法吗？

注意，调用 ClassA 的构造函数时，没有给它传递参数。这在原型链中是标准做法。要确保构造函数没有任何参数。

与对象冒充相似，子类的所有属性和方法都必须出现在 prototype 属性被赋值后，因为在它之前赋值的所有方法都会被删除。为什么？因为 prototype 属性被替换成了新对象，添加了新方法的原始对象将被销毁。所以，为 ClassB 类添加 name 属性和 sayName()方法的代码如下：

```
function ClassB() {
}

ClassB.prototype = new ClassA();

ClassB.prototype.name = "";
ClassB.prototype.sayName = function () {
    alert(this.name);
};
```

可通过运行下面的例子测试这段代码：

```
var objA = new ClassA();
var objB = new ClassB();
objA.color = "red";
objB.color = "blue";
objB.name = "Nicholas";
objA.sayColor();    //outputs "red"
objB.sayColor();    //outputs "blue"
objB.sayName();     //outputs "Nicholas"
```

此外，在原型链中，instanceof 运算符的运行方式也很独特。对 ClassB 的所有实例，instanceof 为 ClassA 和 ClassB 都返回 true。例如：

```
var objB = new ClassB();
alert(objB instanceof ClassA);    //outputs "true";
alert(objB instanceof ClassB);    //outputs "true"
```

在 ECMAScript 的弱类型世界中，这是极其有用的工具，不过使用对象冒充时不能使用它。

原型链的弊端是不支持多重继承。记住，原型链会用另一类型的对象重写类的 prototype 属性。

5. 混合方式

这种继承方式使用构造函数定义类，并未使用任何原型。对象冒充的主要问题是必须使用构造函数方式（如上一章中学到的），这不是最好的选择。不过如果使用原型链，就无法使用带参构造函数了。开发者该如何选择呢？答案很简单，两者都用。

在前一章，你学过创建类的最好方式是用构造函数方式定义属性，用原型方式定义方法。这种方式同样适用于继承机制，用对象冒充继承构造函数的属性，用原型链继承 prototype 对象的方法。用这两种方式重写前面的例子，代码如下：

```
function ClassA(sColor) {
    this.color = sColor;
}

ClassA.prototype.sayColor = function () {
    alert(this.color);
};

function ClassB(sColor, sName) {

    ClassA.call(this, sColor);
    this.name = sName;
}

ClassB.prototype = new ClassA();

ClassB.prototype.sayName = function () {
    alert(this.name);
};
```

在此例子中，继承机制由两行突出显示的代码实现。在第一行突出显示的代码中，在 ClassB 构造函数中，用对象冒充继承 ClassA 类的 sColor 属性。在第二行突出显示的代码中，用原型链继承 ClassA 类的方法。由于这种混合方式使用了原型链，所以 instanceof 运算符仍能正确运行。

下面的例子测试了这段代码：

```
var objA = new ClassA("red");
var objB = new ClassB("blue", "Nicholas");
objA.sayColor();     //outputs "red"
objB.sayColor();     //outputs "blue"
objB.sayName();      //outputs "Nicholas"
```

第 4 章 继承

4.2 继承机制的实现(2):实例

在真正的 Web 站点和应用程序中，几乎不可能创建名为 ClassA 和 ClassB 的类，更可能的是创建表示特定事物（如形状）的类。考虑本章开头所述的形状的例子，Polygon、Triangle 和 Rectangle 类就构成了一组很好的探讨数据。

1. 创建基类

首先考虑 Polygon 类。哪些属性和方法是必需的？首先，一定要知道多边形的边数，所以应该加入整数属性 sides。还有什么是多边形必需的？也许你想知道多边形的面积，那么加入计算面积的方法 getArea()。图 4-3 展示了该类的 UML 表示。

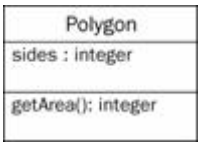

图 4-3

在 UML 中，属性由属性名和类型表示，位于紧接类名之下的单元中。方法位于属性之下，说明方法名和返回值的类型。

在 ECMAScript 中，可以如下编写类：

```
function Polygon(iSides) {
    this.sides  = iSides;
}

Polygon.prototype.getArea = function () {
    return 0;
};
```

注意，Polygon 类不够详细精确，还不能使用，方法 getArea()返回 0，因为它只是一个占位符，以便子类覆盖。

2. 创建子类

现在考虑创建 Triangle 类。三角形具有三条边，因此这个类必须覆盖 Polygon 类的 sides 属性，把它设置为 3。还要覆盖 getArea()方法，使用三角形的面积公式，即 1/2×底×高。但如何得到底和高的值呢？需要专门输入这两个值，所以必须创建 base 属性和 height 属性。Triangle 类的 UML 表示如图 4-4 所示。

该图只展示了 Triangle 类的新属性及覆盖过的方法。如果 Triangle 类没有覆盖 getArea()方法，图中将不会列出它。它将被看作从 Polygon 类保留下来的方法。完整的 UML 图还展示了 Polygon 和 Triangle 类之间的关系（图 4-5），使它显得更清楚。

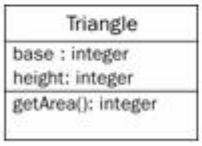

图 4-4

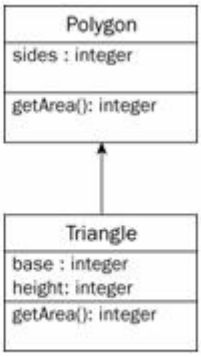

图 4-5

在 UML 中，决不会重复显示继承的属性和方法，除非该方法被覆盖（或被重载，这在 ECMAScript 中是不可能的）。

Triangle 类的代码如下：

```
function Triangle(iBase, iHeight) {
    Polygon.call(this, 3);
    this.base = iBase;
    this.height = iHeight;
}

Triangle.prototype = new Polygon();
Triangle.prototype.getArea = function () {
    return 0.5 * this.base * this.height;
};
```

注意，虽然 Polygon 的构造函数只接受一个参数 sides，Triangle 类的构造函数却接受两个参数，即 base 和 height。这因为三角形的边数是已知的，且不想让开发者改变它。因此，使用对象冒充时，3 作为对象的边数被传给 Polygon 的构造函数。然后，把 base 和 height 的值赋予适当的属性。

在用原型链继承方法后，Triangle 将覆盖 getArea()方法，提供为三角形面积定制的计算。

最后一个类是 Rectangle，它也继承 Polygon。矩形有四条边，面积是用长度×宽度计算的，长度和宽度即成为该类必需的属性。在前面的 UML 图中，要把 Rectangle 类填充在 Triangle 类的旁边，因为它们的超类都是 Polygon（如图 4-6 所示）。

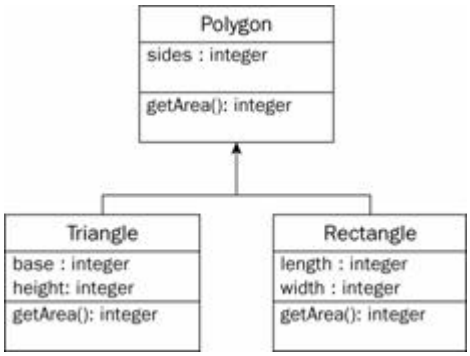

图 4-6

Rectangle 的 ECMAScript 代码如下：

```
function Rectangle(iLength, iWidth) {
    Polygon.call(this, 4);
    this.length = iLength;
    this.width = iWidth;

}

Rectangle.prototype = new Polygon();
Rectangle.prototype.getArea = function () {
    return this.length * this.width;
};
```

注意，Rectangle 构造函数不把 sides 作为参数，同样的，常量 4 被直接传给 Polygon 构造函数。与 Triangle 相似，Rectangle 引入了两个新的作为构造函数的参数的属性，然后覆盖 getArea()方法。

3. 测试代码

可以运行下面代码来测试为该示例创建的代码：

```
var triangle = new Triangle(12, 4);
var rectangle = new Rectangle(22, 10);

alert(triangle.sides);        //outputs "3"
alert(triangle.getArea());    //outputs "24"

alert(rectangle.sides);       //outputs "4"
alert(rectangle.getArea());   //outputs "220"
```

这段代码创建一个三角形，底为 12，高为 4，还创建一个矩形，长为 22，宽为 10。然后输出每种形状的边数及面积，证明 sides 属性的赋值正确，getArea()方法返回正确的值。三角形的面积应为 24，矩形的面积应该是 220。

4. 采用动态原型方法如何？

前面的例子用对象定义的混合构造函数/原型方式展示继承机制，那么可以使用动态原型来实现继承机制吗？不能。

继承机制不能采用动态化的原因是，prototype 对象的独特本性。看下面代码（这段代码不正确，却值得研究）：

```
function Polygon(iSides) {
    this.sides  = iSides;

    if (typeof Polygon._initialized == "undefined") {

        Polygon.prototype.getArea = function () {
            return 0;
        };

        Polygon._initialized = true;
    }
}

function Triangle(iBase, iHeight) {
```

```
    Polygon.call(this, 3);
    this.base = iBase;
    this.height = iHeight;

    if (typeof Triangle._initialized == "undefined") {

        Triangle.prototype = new Polygon();
        Triangle.prototype.getArea = function () {
            return 0.5 * this.base * this.height;
        };

        Triangle._initialized = true;
    }
}
```

上面的代码展示了用动态原型定义的 Polygon 和 Triangle 类。错误在于突出显示的设置 Triangle.prototype 属性的代码。从逻辑上讲，这个位置是正确的，但从功能上讲，却是无效的。从技术上说来，在代码运行前，对象已被实例化，并与原始的 prototype 对象联系在一起了。虽然用极晚绑定可使对原型对象的修改正确地反映出来，但替换 prototype 对象却不会对该对象产生任何影响。只有未来的对象实例才会反映出这种改变，这就使第一个实例变得不正确。

要正确使用动态原型实现继承机制，必须在构造函数外赋予新的 prototype 对象，如下所示：

```
function Triangle(iBase, iHeight) {
    Polygon.call(this, 3);
    this.base = iBase;
    this.height = iHeight;

    if (typeof Triangle._initialized == "undefined") {

        Triangle.prototype.getArea = function () {
            return 0.5 * this.base * this.height;
        };

        Triangle._initialized = true;
    }
}

Triangle.prototype = new Polygon();
```

这段代码有效，因为是在任何对象实例化前给 prototype 对象赋值的。遗憾的是，这意味着不能把这段代码完整的封装在构造函数中了，而这正是动态原型的主旨。

第 4 章 继承

4.3 其他继承方式（1）：zInherit

由于 ECMAScript 继承机制的限制（如缺少专有作用域，不能简单地访问超类的方法），世界各地的开发者都在为创建实现继承机制的其他方式而不懈地努力着。这一节讲解释可以代替标准 ECMAScript 继承机制的继承方式。

4.3.1 zInherit

原型链实际上是把对象的所有方法复制给类的 prototype 对象。还有其他方法可以实现它吗？有，利用 zInherit 库（可以从 http://www.nczonline.net/downloads 处下载），不必使用原型链，也可实现方法继承。这个小库支持所有的现代浏览器（Mozilla、IE、Opera、Safari）及一些旧的浏览器（Netscape 4.*x*、IE、Mac）。

要使用 zInherit 库，必须用<script/>标签加入 zinherit.js。第 5 章将详细讨论引入外部的 JavaScript 文件。

zInherit 库给 Object 类添加了两个方法，inheritFrom()和 instanceOf()。如你所料，inheritFrom()方法负担重任，负责复制指定类的所有方法。下面一行代码用原型链使 ClassB 继承 ClassA 的方法：

```
ClassB.prototype = new ClassA();
```

可用下面代码替换上面的代码：

```
ClassB.prototype.inheritFrom(ClassA);
```

inheritFrom()方法接受一个参数，即要复制的方法所属的类。注意，与原型链相对的是，这种方式并未真正创建要继承的类的实例，这样更安全，开发者也无需担心构造函数的参数。

为确保正确地实现继承，必须在原型赋值之处调用 inheritFrom()方法。

instanceOf()方法是 instanceof 运算符的替代品。因为这种方式根本不使用原型链，所以这行代码无效：

```
ClassB instanceof ClassA
```

instanceOf()方法弥补了这项损失，与 inheritFrom()一起使用，可以跟踪所有的超类：

```
ClassB.instanceOf(ClassA);
```

1. 再探多边形

整个多边形的例子都可用 zInherit 库重写，只需要替换两行（突出显示）代码：

```
function Polygon(iSides) {
    this.sides  = iSides;
}

Polygon.prototype.getArea = function () {
    return 0;
};

function Triangle(iBase, iHeight) {
    Polygon.call(this, 3);
    this.base = iBase;
    this.height = iHeight;
}

Triangle.prototype.inheritFrom(Polygon);

Triangle.prototype.getArea = function () {
    return 0.5 * this.base * this.height;
};

function Rectangle(iLength, iWidth) {
    Polygon.call(this, 4);
    this.length = iLength;
    this.width = iWidth;
}

Rectangle.prototype.inheritFrom(Polygon);

Rectangle.prototype.getArea = function () {
    return this.length * this.width;
};
```

可以用与前面相同的例子测试这段代码，只需添加两行代码，测试 instanceOf()方法的输出即可：

```
var triangle = new Triangle(12, 4);
var rectangle = new Rectangle(22, 10);

alert(triangle.sides);
alert(triangle.getArea());

alert(rectangle.sides);
alert(rectangle.getArea());

alert(triangle.instanceOf(Triangle));    //outputs "true"
alert(triangle.instanceOf(Polygon));     //outputs "true"

alert(rectangle.instanceOf(Rectangle));  //outputs "true"
alert(rectangle.instanceOf(Polygon));    //outputs "true"
```

最后四行代码用于测试 instanceOf()方法，都应该返回 true。

2. 动态原型支持

如前所述，原型链不能真正满足动态原型主旨，即把类的所有代码放置在它的构造函数中。zInherit 库修正了这个问题，它允许在构造函数内部调用 inheritFrom()方法。

让我们再看看前面的多边形动态原型的例子，现在加入 zInherit 库：

```
function Polygon(iSides) {
    this.sides  = iSides;

    if (typeof Polygon._initialized == "undefined") {

        Polygon.prototype.getArea = function () {
            return 0;
        };

        Polygon._initialized = true;
    }
}

function Triangle(iBase, iHeight) {
    Polygon.call(this, 3);
    this.base = iBase;
    this.height = iHeight;

    if (typeof Triangle._initialized == "undefined") {

        Triangle.prototype.inheritFrom(Polygon);
        Triangle.prototype.getArea = function () {
            return 0.5 * this.base * this.height;
        };

        Triangle._initialized = true;
    }
}
```

```
function Rectangle(iLength, iWidth) {
    Polygon.call(this, 4);
    this.length = iLength;
    this.width = iWidth;

    if (typeof Rectangle._initialized == "undefined") {

        Rectangle.prototype.inheritFrom(Polygon);
        Rectangle.prototype.getArea = function () {
            return this.length * this.width;
        };

        Rectangle._initialized = true;
    }
}
```

前面的代码中突出显示的两行代码实现了 Triangle 类和 Rectangle 类对 Polygon 类的继承。这种方法成功的原因是，使用 inheritFrom()方法时，未重写 prototype 对象，只是为其加入方法而已。使用这种方法，即可避开原型链的限制，实现动态原型本意。

3. 多重继承支持

zInherit 库最有用的特性之一是支持多重继承，原型链不支持这种能力。同样，使这种支持成为可能的关键是 inheritFrom()方法不替换 prototype 对象。

要继承属性和方法，inheritFrom()方法必须与对象冒充一起使用。考虑下面的例子：

```
function ClassX() {
    this.messageX = "This is the X message. ";

    if (typeof ClassX._initialized == "undefined") {

        ClassX.prototype.sayMessageX = function () {
            alert(this.messageX);
        };

        ClassX._initialized = true;
    }
}

function ClassY() {
    this.messageY = "This is the Y message. ";

    if (typeof ClassY._initialized == "undefined") {

        ClassY.prototype.sayMessageY = function () {
            alert(this.messageY);
        };

        ClassY._initialized = true;
    }
}
```

ClassX 和 ClassY 类都是小类，都只有一个属性及一个方法。假设现有 ClassZ 类，需要继承这两个类。可以如下定义该类：

```
    function ClassZ() {
        ClassX.apply(this);
        ClassY.apply(this);
        this.messageZ = "This is the Z message. ";

        if (typeof ClassZ._initialized == "undefined") {

            ClassZ.prototype.inheritFrom(ClassX);
            ClassZ.prototype.inheritFrom(ClassY);

        ClassZ.prototype.sayMessageZ = function () {
            alert(this.messageZ);
        };

        ClassZ._initialized = true;
    }
}
```

注意继承属性的两行代码（使用 apply()方法）和继承方法的两行代码（使用 inheritFrom()方法）。如前所述，发生继承的顺序非常重要。通常按照继承属性的顺序继承方法比较好（意味着如果先继承 ClassX 的属性，然后继承 ClassY 的属性，那么也应该按照这种顺序继承它们的方法）。

下面的代码是测试多重继承的示例：

```
var objZ = new ClassZ();
objZ.sayMessageX();     //outputs "This is X message. "
objZ.sayMessageY();     //outputs "This is Y message."
objZ.sayMessageZ();     //outputs "This is Z message."
```

前面的代码调用了三个方法：

(1) sayMessageX()方法是从 ClassX 继承而来，它访问 messageX 属性，该属性也是从 ClassX 继承而来。

(2) sayMessageY()方法是从 ClassY 继承而来，它访问 messageY 属性，该属性也是从 ClassY 继承而来。

(3) sayMessageZ()方法是在 ClassZ 中定义的，它访问 messageZ 属性，该属性也是在 ClassZ 在定义的。

这三种方法应该输出各属性对应的消息，说明多重继承成功。

第 4 章 继承

4.3 其他继承方式（1）：xbObjects

4.3.2 xbObjects

Netscape 的 DevEdge 站点（http://devedge.netscape.com）有许多对 Web 开发者有用的信息和脚本工具。工具之一是 xbObjects（可以从 http://archive.bclary.com/xbProjects- docs/xbObject/处下载），由 Netscape 公司的 Bob Clary 于 2001 年 Netscape 6（Mozilla 0.6）发布时编写而成。它支持从那时起的所有 Mozilla 版本及其他现代浏览器（IE、Opera 和 Safari）。

1. 目的

xbObjects 的目的是为 JavaScript 提供更强的面向对象范型，不止支持继承，还支持方法的重载和调用超类方法的能力。要实现这一点，xbObjects 需执行许多步。

第一步，必须注册类，此时，需定义它是由哪个类继承而来。用下面的调用可以实现这一点：

```
_classes.registerClass("Subclass_Name", "Superclass_Name");
```

这里，子类和超类名都以字符串形式传进来，而不是指向它们的构造函数的指针。这个调用必须放在指定子类的构造函数前。

如果新的类未继承任何类，调用 registerClass()时也可以只用第一个参数。

第二步，在构造函数内调用 defineClass()方法，传给它类名及被 Clary 称为原型函数（prototype function）的指针，该函数用于初始化对象的所有属性和方法（之后会介绍更多），例如：

```
_classes.registerClass("ClassA");

function ClassA(color) {
    _classes.defineClass("ClassA", prototypeFunction);

    function prototypeFunction() {
        //...
    }
}
```

可以看到，原型函数（prototypeFunction()）位于构造函数内部。它的主要用途是在适当的时候把所有方法赋予该类（在这一点上与动态原型相似）。

下一步（迄今为止是第三步）是为该类创建 init()方法。该方法负责设置该类的所有属性，它必须接受与构造函数相同的参数。作为一种规约，init()方法总是在 defineClass()方法后调用。例如：

```
_classes.registerClass("ClassA");

function ClassA(sColor) {
    _classes.defineClass("ClassA", prototypeFunction);

    this.init(sColor);

    function prototypeFunction() {

        ClassA.prototype.init = function (sColor) {
            this.parentMethod("init");
            this.color = sColor;
        };

    }
}
```

你可能已注意到 init()方法中调用的 parentMethod()方法。xbObjects 以这种方式允许类调用它的超类的方法。parentMethod()方法接受任意多个参数，但第一个参数总是要调用的父类方法的名字（该参数必须是字符串，而不是函数指针），所有其他参数都被传给父类的方法。

在这个例子中，首先调用 init()方法，这是 xbObjects 运行所必需的。即使 ClassA 未注册超类，xbObejcts 都会为它创建一个所有类的默认超类，即超类方法 init()所属的类。

第四步也是最后一步，在原型函数内添加其他类的方法：

```
_classes.registerClass("ClassA");

function ClassA(sColor) {
    _classes.defineClass("ClassA", prototypeFunction);

    this.init(sColor);

    function prototypeFunction() {

        ClassA.prototype.init = function (sColor) {
            this.parentMethod("init");
            this.color = sColor;
        };

        ClassA.prototype.sayColor = function () {
            alert(this.color);
        };

    }
}
```

然后，即可以以常规方式创建 ClassA 的实例：

```
var objA = new ClassA("red");
objA.sayColor();    //outputs "red"
```

2. 重载多边形

此时，你一定想知道是否可以用 xbObjects 重写多边形的例子，下面就是重写后的代码。

首先，重写 Polygon 类，非常简单：

```
        _classes.registerClass("Polygon");

        function Polygon(sides) {

            _classes.defineClass("Polygon", prototypeFunction);

            this.init(sides);

            function prototypeFunction() {

                Polygon.prototype.init = function(iSides) {
                    this.parentMethod("init");
                    this.sides = iSides;
                };

            Polygon.prototype.getArea = function () {
                return 0;
            };


    }
}
```

接下来，重写 Triangle 类，是这个例子中第一个真正用到继承的类：

```
    _classes.registerClass("Triangle", "Polygon");

    function Triangle(iBase, iHeight) {

        _classes.defineClass("Triangle", prototypeFunction);

        this.init(iBase,iHeight);

        function prototypeFunction() {
            Triangle.prototype.init = function(iBase, iHeight) {
                this.parentMethod("init", 3);
                this.base = iBase;
                this.height = iHeight;
            };

            Triangle.prototype.getArea = function () {
                return 0.5 * this.base * this.height;
            };
        }

    }
```

注意在构造函数之前调用 registerClass()，并建立继承关系。此外，init()方法的第一行调用超类的 init()方法，参数是 3，把 sides 属性设置成 3。余下的就是为 base 和 height 属性赋值。

Rectangle 类与 Triangle 类类似：

```
    _classes.registerClass("Rectangle", "Polygon");

    function Rectangle(iLength, iWidth) {

        _classes.defineClass("Rectangle", prototypeFunction);

        this.init(iLength, iWidth);

        function prototypeFunction() {
            Rectangle.prototype.init = function(iLength, iWidth) {
                this.parentMethod("init", 4);
                this.length = iLength;
                this.width = iWidth;
            }

        Rectangle.prototype.getArea = function () {
            return this.length * this.width;
        };

    }
}
```

该类与 Triangle 类之间的主要区别（除 registerClass()和 defineClass()类的调用外）是调用超类方法 init()的参数是 4。然后，添加 length 属性和 width 属性，覆盖 getArea()方法。

第 4 章 继承

4.4 小结

本章介绍了 ECMAScript（从而也是 JavaScript）中用对象冒充和原型链实现的继承概念。学会结合使用这些方式才是建立类之间的继承机制的最好方式。

最后，还介绍了其他两种建立继承机制的方式，即 zInherit 和 xbObjects。这些 JavaScript 库都可以从因特网上免费得到，它们为对象继承引入了新的不同的能力。

对 JavaScript 的核心 ECMAScript 的讨论到此为止。接下来的章节将在这个基础上介绍更多该语言与 Web 相关的内容。

第 5 章 浏览器中的 JavaScript

5.1 HTML 中的 JavaScript

在前面几章中，学习了 JavaScript 的核心 ECMAScript 以及该语言工作方式的基础知识。从本章开始，重点将转移到如何在 Web 浏览器中使用 JavaScript。

自 Netscape Navigator 2.0 初次引入 JavaScript 以来，Web 浏览器已有了长足的发展。今天的浏览器不再只能处理传统的 HTML 文件，它们能处理各种格式的文件。具有讽刺意味的是，这些文件中的大多数都采用 JavaScript 作为动态改变客户端内容的方式。这一章探讨如何把 JavaScript 嵌入 HTML 及其他语言，并介绍了 BOM（浏览器对象模型）的一些基本概念。

5.1 HTML 中的 JavaScript

当然，第一个利用嵌入式JavaScript的语言还是HTML，因此首先讨论的自然是如何在HTML中使用JavaScript。HTML中嵌入JavaScript是从引入用于JavaScript的标签和为HTML的一些通用部分增加了新特性开始的。

5.1.1 <script/>标签

HTML 页面中包含 JavaScript 使用<script/>标签实现的。该标签通常放置在页面的<head/>标签中，最初定义的<script/>标签具有一个或两个特性，language 特性声明要使用的脚本语言，src 特性是可选的，声明要加入页面的外部 JavaScript 文件。language 特性一般被设置为 JavaScript，不过也可用它声明 JavaScript 的确切版本，如 JavaScript 1.3（如果省略 language 特性，浏览器默认使用最新的 JavaScript 版本）。

尽管<script/>最初是为 JavaScript 设计的，但可以用于声明任意多种不同的客户端脚本语言，language 特性用于声明使用的代码的类型。例如，可把 language 特性设置为 VBScript，使用 IE 的 VBScript 语言（只适用于 Windows）。

如果未声明 src 特性，在<script/>中即可以任意形式编写 JavaScript 代码。如声明了 src 特性，那么<script/>中的代码可能就是无效的（由浏览器决定）。例如：

```
<html>
    <head>
        <title>Title of Page</title>
        <script language="JavaScript">
            var i = 0;
        </script>
        <script language="JavaScript" src="../scripts/external.js"></script>
    </head>
    <body>
        <!-- body goes here -->
    </body>
</html>
```

这个例子中既有内嵌的 JavaScript 代码，又有对外部 JavaScript 文件的链接。使用 src 特性，即可像引用图像和样式表一样引用 JavaScript 文件。

虽然大多数浏览器并未要求，但根据规约，外部 JavaScript 文件的扩展名应为.js（这样可以使用 JSP、PHP 或其他服务器端的脚本语言动态生成 JavaScript 代码）。

5.1.2 外部文件格式

外部 JavaScript 语言的格式非常简单。事实上，它们只包含 JavaScript 代码的纯文本文件。在外部文件中不需要<script/>标签，引用文件的<script/>标签出现在 HTML 页中。这使得外部 JavaScript 文件看起来很像其他程序设计语言的源代码文件。

例如，考虑下面的内嵌代码：

```
<html>
    <head>
        <title>Title of Page</title>
        <script language="JavaScript">
            function sayHi() {
                alert("Hi");
            }
        </script>
    </head>
    <body>
        <!-- body goes here -->
    </body>
</html>
```

要把函数 sayHi()放在外部文件 external.js 中，需要复制函数文本自身（如图 5-1 所示）。

external.js

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

图 5-1

然后可更新 HTML 代码，加入这个外部文件：

```
<html>
    <head>
        <title>Title of Page</title>
        <script language="JavaScript" src="external.js"></script>
    </head>
    <body>
        <!-- body goes here -->
    </body>
</html>
```

对于 JavaScript 源文件中可加入哪些代码并无规定，这意味着可以给 JavaScript 文件加入任意多个类定义、函数，等等。

5.1.3 内嵌代码和外部文件

何时应该采用内嵌代码，何时采用外部文件呢？虽然关于这一点并无确定而且简洁的规则可循，不过一般认为，大量的 JavaScript 代码不应内嵌在 HTML 文件中，原因如下：

- 安全性——只要查看页面的源代码，任何人都可确切地知道其中的代码做了什么。如果怀有恶意的开发者查看了源代码，就可能发现安全漏洞，危及整个站点或应用程序的安全。此外，在外部文件中还可加入版权和其他知识产权通告，而不打断页面流。

- 代码维护——如果 JavaScript 代码散布于多个页面，那么代码维护将变成一场恶梦。把所有 JavaScript 文件放在一个目录中要容易得多，这样在发生 JavaScript 错误时，就不会对放置代码的位置有任何疑问。

❑ 缓存——浏览器会根据特定的设置缓存所有外部链接的 JavaScript 文件，这意味着如果两个页面使用同一个文件，只需要下载该文件一次。这将加快下载速度。把同一段代码放在多个页面中，不止浪费，还增加了页面大小，从而增加下载时间。

5.1.4 标签放置

一般说来，所有代码和函数的定义都在 HTML 页的<head/>标签中，这样在显示页面主体后，代码就被完全装载进浏览器，可供使用了。唯一该出现在<body/>标签中的是调用前面定义的函数的代码。

<script/>放在<body/>内时，只要脚本所属的那部分页面被载入浏览器，脚本就会被执行。这样在载入整个页面之前，也可执行 JavaScript 代码。例如：

```
<html>
    <head>
        <title>Title of Page</title>
        <script language="JavaScript">
            function sayHi() {
                alert("Hi");

            }
        </script>
    </head>
    <body>
        <script language="JavaScript">
            sayHi();
        </script>
        <p>This is the first text the user will see.</p>
    </body>
</html>
```

在这段代码中，方法 sayHi()在页面显示所有文本前调用，这意味着警告消息将在文本"This is the first text the user will see."显示前弹出。建议不采用这种在页面的<body/>标签内调用 JavaScript 函数的方法，应该尽量避免它。相反的，建议在页面主体中只使用事件处理函数（event handler），例如：

```
<html>
    <head>
        <title>Title of Page</title>
        <script language="JavaScript">
            function sayHi() {
                alert("Hi");
            }
        </script>
    </head>
    <body>

        <input type="button" value="Call Function" onclick="sayHi()" />
    </body>
</html>
```

这里，使用<input/>标签创建一个按钮，点击它时调用 sayHi()方法。onclick 特性声明一个事件处理函数，即响应特定事件的代码。第 9 章将详细讨论事件和事件处理函数。

注意，开始载入页面时，JavaScript 就开始运行了，因此有可能调用尚未存在的函数。在前面的例子中，把原来的<script/>标签放在函数调用后就会引发错误：

```
<html>
    <head>
        <title>Title of Page</title>
    </head>
    <body>
        <script language="JavaScript">
            sayHi();
        </script>
        <p>This is the first text the user will see.</p>
        <script language="JavaScript">
            function sayHi() {
                alert("Hi");
            }
        </script>
    </body>
</html>
```

这个例子将引发错误，因为在定义 sayHi()之前就调用了它。由于 JavaScript 是从上到下载入的，所以在遇到第二个<script/>标签前，函数 sayHi()还不存在。注意这种问题，此外，如前所述，使用事件和事件处理函数调用 JavaScript 函数。

5.1.5 隐藏还是不隐藏

初次引入 JavaScript 时，只有一种浏览器支持它，因此大家开始关心，不支持 JavaScript 的浏览器如何处理<script/>标签及其中包含的代码。最后，设计了一种用于对旧的浏览器隐藏 JavaScript 代码（这是一个短语，在当今因特网上的许多 Web 站点的源代码中都能找到它）的格式。下面的代码在内嵌代码周围加入 HTML 注释，这样其他浏览器就不会在屏幕上显示这段代码。

```
<script language="JavaScript"><!-- hide from older browsers
    function sayHi() {
        alert("Hi");
    }
//-->
</script>
```

第一行紧接起始标签<script>开始一条 HTML 注释。这样做是有效的，因为浏览器仍然

把该行余下的部分看成 HTML 的一部分，JavaScript 代码从下一行开始。接下来的是常规的函数定义。第 2 行到最后一行是最有趣的部分，因为它以单行 JavaScript 注释标记（两个前斜线）开始，后面是 HTML 注释的结尾标记（-->）。这一行仍被看作 JavaScript 代码，所以单行注释标记是避免语法错所必需的。不过，旧的浏览器只承认 HMLT 注释的结束标记，因此，将忽略所有 JavaScript 代码。但是，支持 JavaScript 的浏览器只忽略该行，继续执行</script>标签。

尽管这种隐藏代码的方法在 Web 早期非常流行，今天却不再是必需的。目前，大多数 Web 浏览器都支持 JavaScript，而不支持 JavaScript 的浏览器通常足够聪明，自己就能够忽略 JavaScript 代码。

5.1.6 <noscript/>标签

不支持 JavaScript 的浏览器另外令人关注的是如何提供替代的内容。隐藏代码只是解决方法的一部分，开发者还需要一种方法，声明在 JavaScript 不能用时应该显示的内容。解决方法是采用<noscript/>标签，它可包含任何 HTML 代码（除<script/>）。支持或启用 JavaScript 的浏览器会忽略这些 HTML 代码，不支持或者禁用 JavaScript 的浏览器则显示<noscript/>的内容。例如：

```
<html>
    <head>
        <title>Title of Page</title>
        <script language="JavaScript">
            function sayHi() {
                alert("Hi");
            }
        </script>

    </head>
    <body>
        <script language="JavaScript">
             sayHi();
        </script>
        <noscript>
            <p>Your browser doesn't support JavaScript. If it did support
JavaScript, you would see this message: Hi!</p>
        </noscript>
        <p>This is the first text the user will see if JavaScript is enabled. If
JavaScript is disabled this is the second text the user will see.</p>
    </body>
</html>
```

在这个例子中，<noscript/>标签中有一条消息，告诉用户浏览器不支持 JavaScript。第 8 章解释<noscript/>的实际用法。

5.1.7 XHTML 中的改变

近来，随着 XHTML（可扩展 HTML）标准的出现，<script/>标签也经历了一些改变。该标签不再用 language 特性，而用 type 特性声明内嵌代码或要加入的外部文件的 mime 类型，JavaScript 的 mime 类型是"text/javascript"。例如：

```
<html>
    <head>
        <title>Title of Page</title>
        <script type="text/javascript">
            var i = 0;
        </script>
        <script type="text/javascript" src="../scripts/external.js"></script>
    </head>
    <body>
        <!-- body goes here -->
    </body>
</html>
```

即使许多浏览器不完全支持 XHTML，但大多数开发者现在都用 type 特性，而不用 language 特性，以提供更好的 XHTML 支持。省略 language 特性不会带来任何问题，因为如前所述，所有浏览器都默认<script/>的该属性值为 JavaScript。

XHTML 的第二个改变是使用 CDATA 段。XML 中的 CDATA 段用于声明不应被解析为标签的文本（XHTML 也是如此），这样就可以使用特殊字符，如小于（<）、大于（>）、和号（&）和双引号（"），而不必使用它们的字符实体。考虑下面的代码：

```
<script type="text/javascript">
    function compare(a, b) {
        if (a < b) {
            alert("A is less than B");
        } else if (a > b) {
            alert("A is greater than B");
        } else {

            alert("A is equal to B");
        }
    }
</script>
```

这个函数相当简单，它比较数字 a 和 b，然后显示消息说明它们的关系。但是，在 XHTML

中，这段代码是无效的，因为它使用了三个特殊符号，即小于、大于和双引号。要修正这个问题，必须分别用这三个字符的 XML 实体<、>和"替换它们：

```
<script type="text/javascript">
    function compare(a, b) {
        if (a &lt; b) {
            alert(&quot;A is less than B&quot;);
        } else if (a &gt; b) {
            alert(&quot;A is greater than B&quot;);
        } else {
            alert(&quot;A is equal to B&quot;);
        }
    }
</script>
```

这段代码存在两个问题。首先，开发者不习惯用 XML 实体编写代码。这使代码很难读懂。其次，在 JavaScript 中，这种代码实际上将视为有语法错，因为解释程序不知道 XML 实体的意思。用 CDATA 段即可以以常规形式（即易读的语法）编写 JavaScript 代码。正式加入 CDATA 段的方法如下：

```
<script type="text/javascript"><![CDATA[
    function compare(a, b) {
        if (a < b) {
            alert("A is less than B");
        } else if (a > b) {
            alert("A is greater than B");
        } else {
            alert("A is equal to B");
        }
    }
]]></script>
```

虽然这是正式方式，但还要记住，大多数浏览器都不完全支持 XHTML，这就带来主要问题，即这在 JavaScript 中是个语法错误，因为大多数浏览器还不认识 CDATA 段。

当前使用的解决方案模仿了“对旧浏览器隐藏”代码的方法。使用单行的 JavaScript 注释，可在不影响代码语法的情况下嵌入 CDATA 段：

```
    <script type="text/javascript">
    //<![CDATA[
        function compare(a, b) {
            if (a < b) {
                alert("A is less than B");
            } else if (a > b) {
                alert("A is greater than B");
            } else {
                alert("A is equal to B");
```

```
        }
    }
//]]>
</script>
```

现在，这段代码在不支持 XHTML 的浏览器中也可运行。

与 type 特性一样，随着开发者为浏览器中的 XHTML 准备更好的支持，CDATA 的这种用法也越来越流行。但是，为避免 CDATA 的问题，最好还是用外部文件引入 JavaScript 代码。

第 5 章 浏览器中的 JavaScript

5.2 SVG 中的 JavaScript

SVG 是一种崭露头角的基于 XML 的语言，用于在 Web 上绘制矢量图形。矢量图形不同于光栅图形（位图），它们定义的是三角形、线段及它们之间的关系，而不只是定义图像的每个像素的颜色。这样生成的图像无论大小，看起来都是相同的。随着这种语言的日益流行，矢量图形程序（如 Adobe Illustrator）已经开始加入 SVG 导出功能。

尽管目前没有浏览器内置支持 SVG（虽然 Mozilla 2.0 将支持它），但许多公司（包括著名的 Adobe 和 Corel）都在编写 SVG 插件，以便使大多数浏览器都能显示 SVG 图形。

5.2.1 SVG 基础

介绍 SVG 这种语言不在本书的讨论范围内。不过，对该语言稍有理解有助于 JavaScript 的讨论。

下面是一个简单的 SVG 示例：

```
<?xml version="1.0"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.0//EN"
"http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd">
<svg xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http://www.w3.org/1999/xlink"
width="100%" height="100%">
    <desc>
        An image of a square and a circle.
    </desc>
    <defs>
        <rect id="rect1" width="200" height="200" fill="red" x="10" y="10"
stroke="black"/>
        <circle id="circle1" r="100" fill="white" stroke="black" cx="200"
cy="200"/>
    </defs>
    <g>
        <use xlink:href="#rect1" />
        <use xlink:href="#circle1" />
    </g>
</svg>
```

这个例子将在正方形的右下角画一个圆（如图 5-2 所示）。

注意，SVG 文件的开头是 XML 序言（prolog）<?xml version="1.0"?>，声明该语言是基于 XML 的。接下来的 SVG DTD 是可有可无的，不过通常都会被加入。

最外层的标签是<svg/>，把该文件定义为 SVG 图像。特性 width 和 height 可被设置为任何值，包括百分数和像素，不过这里为简单起见，把它们设置为 100%。注意声明了两个 XML 命名空间，一个用于 SVG，一个用于 XLink。XLink 定义 href 这样的链接行为，将来的 XHTML 版本很可能支持它。目前，SVG 带动了对 XLink 的基本支持。

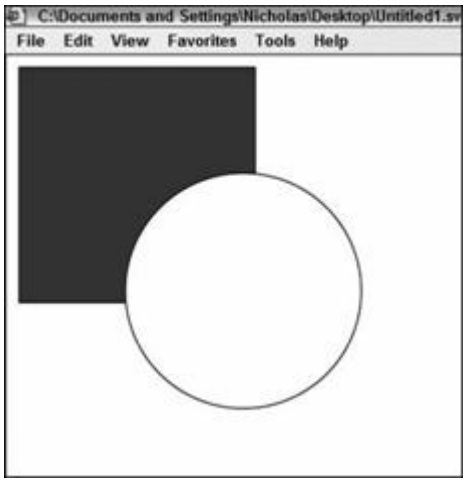

图 5-2

接下来的标签是<desc/>，包含图像的说明。可把<desc/>标签看作 HTML 中的<title/>标签，它说明了图像中包含的内容，而不是如何在页面上显示图像。紧接着是<defs/>标签，定义图像中要使用的资源及形状。在这个例子中，定义了一个矩形和一个圆。除非专门用在真正的图像中，否则这些形状不会被显示。

<defs/>之后是<g/>标签，即 group 的缩写。标签<g/>很特殊，因为它是最外层的，从而封装了该可视图像。在最外层的<g/>标签内可多次使用<g/>标签（就像 HTML 中的 ），以构成形状组。

在这个例子中，有两个<use/>标签指向<defs/>段中的形状。<use/>标签的 xlink:href 特性指向形状的 ID（前面有英镑符号#），从而把这个形状带到该可视图像中。<defs/>中定义的形状可用多个<use/>标签在图像中多次使用。SVG 的这种能力使它成为基于 XML 的语言中代码重用的典范。

当然，SVG 最令人兴奋的一点是对 JavaScript 的优秀支持，用 JavaScript 可以对 SVG 图像的各个部分进行操作。

5.2.2 SVG 中的<script/>标签

SVG 采用的<script/>标签与将 JavaScript 加入页面的<script/>标签相似。但这个<script/>标签不同于 HTML 中的<script/>标签：

- **type 特性是必需的**。type 特性可被设置为 text/javascript 或 text/ecmascript，不过从技术上讲，前者才是正确的。
- **language 特性是不合法的**。加入这个特性，SVG 代码就会无效。
- **内嵌代码必须使用 CDATA 段**。由于 SVG 是真正基于 XML 的语言，所以它能正确地支持 CDATA 段，因此内嵌代码使用特殊的 XML 字符时，必须使用 CDATA 段。
- **使用xlink:href代替src**。在SVG中，<script/>标签中没有src特性，而是使用xlink:href特性说明要引用的外部文件。

例如：

```
<?xml version="1.0"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.0//EN"
"http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd">
<svg xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http://www.w3.org/1999/xlink"
width="100%" height="100%">
    <desc>
        An image of a square and a circle.
    </desc>
    <script type="text/javascript"><![CDATA[
        var i = 0;
    ]]></script>
    <script type="text/javascript" xlink:href="../scripts/external.js"></script>
    <defs>
        <rect id="rect1" width="200" height="200" fill="red" x="10" y="10"
stroke="black"/>
        <circle id="circle1" r="100" fill="white" stroke="black" cx="200"
cy="200"/>
    </defs>
    <g>
        <use xlink:href="#rect1" />
        <use xlink:href="#circle1" />
    </g>
</svg>
```

这段代码中，SVG 有两个正确的<script/>标签。第一个包含内嵌代码，被 CDATA 段包围着，因此使用特殊字符不会产生任何问题。第二个使用 xlink:href 特性引用外部文件。

5.2.3 SVG 中的标签放置

由于 SVG 中不存在<head/>区，所以<script/>标签几乎可以放在任何位置。但是，通常它们放在以下地方：

- 紧接在<desc/>标签后；
- 在<defs/>标签中；
- 恰好位于最外层的<g/>标签前。

<script/>标签不能放在形状内部，如<recct/>或<circle/>，也不能放在滤光器、梯度或其他定义外观的标签内。

第 5 章 浏览器中的 JavaScript

5.3 BOM(1)

讨论浏览器中的 JavaScript，不能不讨论 BOM（浏览器对象模型），它提供了独立于内容而与浏览器窗口进行交互的对象。

BOM 由一系列相关的对象构成。图 5-3 展示了基本的 BOM 体系结构。

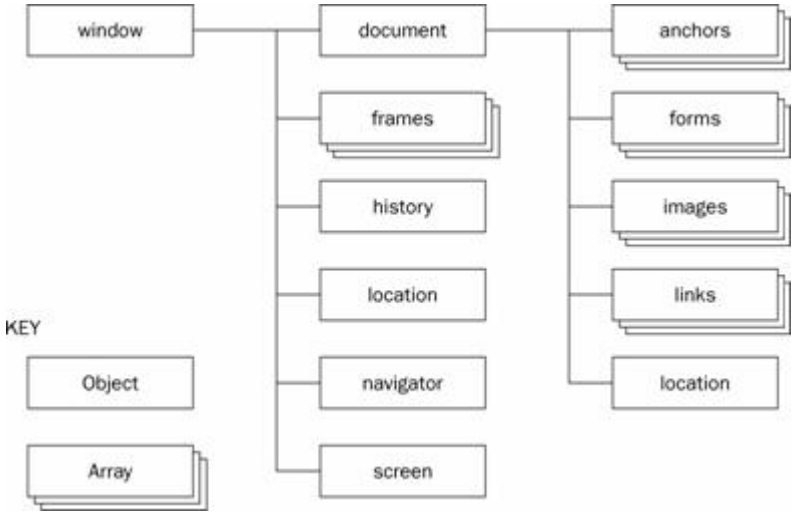

图　5-3

可以看到，window 对象是整个 BOM 的核心，所有对象和集合都以某种方式回接到 window 对象。我从这个对象着手讨论 BOM。

5.3.1　window 对象

window 对象表示整个浏览器窗口，但不必表示其中包含的内容。此外，window 还可用于移动或调整它表示的浏览器的大小，或者对它产生其他影响。

如果页面使用框架集合，每个框架都由它自己的 window 对象表示，存放在 frames 集合中。在 frames 集合中，可用数字（由 0 开始，从左到右，逐行的）或名字对框架进行索引。考虑下面的例子：

```
<html>
    <head>
        <title>Frameset Example</title>
</head>
<frameset rows="100,*">
    <frame src="frame.htm" name="topFrame" />
    <frameset cols="50%,50%">
        <frame src="anotherframe.htm" name="leftFrame" />
        <frame src="yetanotherframe.htm" name="rightFrame" />
    </frameset>
```

```
    </frameset>
</html>
```

这段代码创建一个框架集，其中包括一个顶层框架和两个底层框架。这里，可以用 window. frames[0]或 window.frames[“topFrame”]引用框架，也可以用 top 对象代替 window 对象引用这些框架（例如 top.frames[0]）。

top 对象指向的都是最顶层的（最外层的）框架，即浏览器窗口自身。这可以确保指向正确的框架。此后，如果在框架内编写代码，其中引用的 window 对象就只是指向该框架的指针。

由于 window 对象是整个 BOM 的中心，所以它享有一种特权，即不需要明确引用它。在引用函数、对象或集合时，解释程序都会查看 window 对象，所以 window.frames[0]可以只写作 frame[0]。要理解前面例子中引用框架的各种方法，请参考图 5-4。

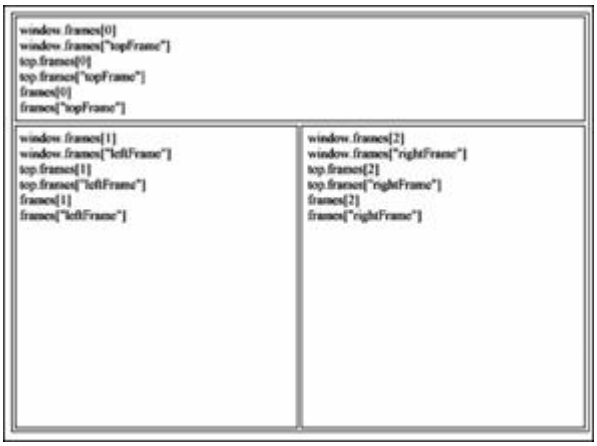

图 5-4

也可以直接用框架的名字访问它，如 window.leftFrame。不过 frames 集合更常用，因为它能确切地表示出代码的意图。

window 另一个实例是 parent。parent 对象与装载文件的框架集一起使用，要装载的文件也是框架集。假设名为 frameset1.htm 的文件包含下面的代码：

```
<html>
    <head>
        <title>Frameset Example</title>
    </head>
    <frameset rows="100,* ">
        <frame src="frame.htm" name="topFrame" />
        <frameset cols="50%,50%">
            <frame src="anotherframe.htm" name="leftFrame" />
            <frame src="anotherframeset.htm" name="rightFrame" />
        </frameset>
    </frameset>
</html>
```

如果是名为 anotherframeset.htm 的文件包含这段代码呢？

```
<html>
    <head>
        <title>Frameset Example</title>
    </head>
    <frameset cols="100,* ">
        <frame src="red.htm" name="redFrame" />
        <frame src="blue.htm" name="blueFrame" />
    </frameset>
</html>
```

第一个文件 frameset1.htm 被载入浏览器时，它将把 anotherframeset.htm 载入到 rightFrame。如果代码写在 redFrame（或 blueFrame）中，parent 对象就指向 frameset1.htm 中的 rightFrame。但如果代码写在 topFrame 中，parent 对象就指向 top 对象，因为浏览器窗口自身被看作所有顶层框架的父框架。

图 5-5 访问 window 对象的 name 属性，它存储的是框架的名字（不过一定是空白或 top），从而证明了这一点。

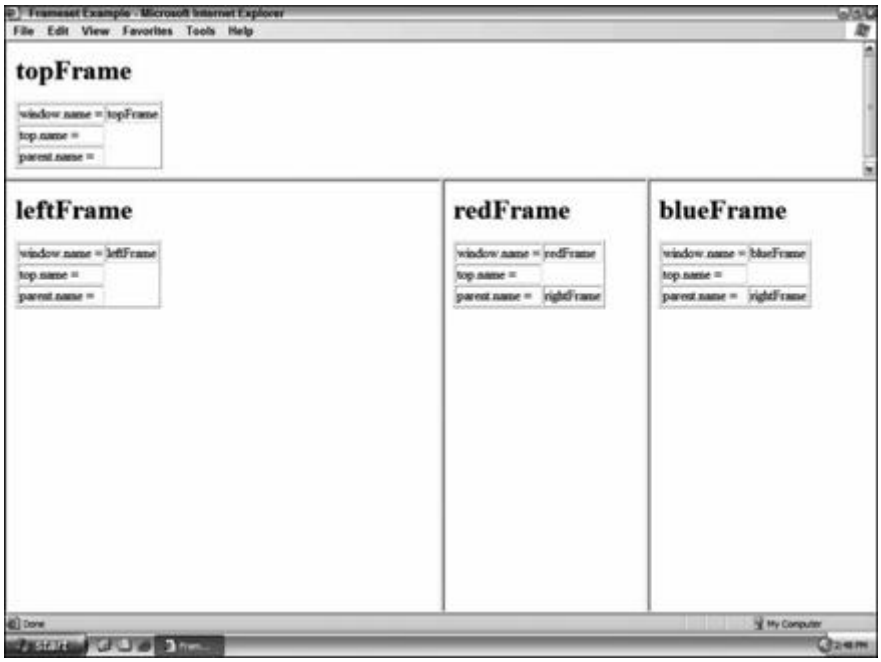

图 5-5

一个更加全局化的窗口指针是 self，它总是等于 window（是的，有点多余，加入它是因为它比 parent 更合适。它澄清了正在使用的不是框架的父框架，而是它自身）。

如果页面上没有框架，window 和 self 就等于 top，frames 集合的长度为 0。

还可以连锁引用 window，例如 parent.parent.frames["topFrame"]，不过通常不赞成使用这种形式，因为框架结构的任何改变都会造成代码错误。

1. 窗口操作

如前所述，window 对象对操作浏览器窗口（和框架）非常有用。这意味着，开发者可以移动或调整浏览器窗口的大小。可用四种方法实现这些操作：

- moveBy(dx,dy)——把浏览器窗口相对当前位置水平移动 dx 个像素，垂直移动 dy 个像素。dx 值为负数，向左移动窗口，dy 值为负数，向上移动窗口。

- moveTo(x,y)——移动浏览器窗口，使它的左上角位于用户屏幕的(x,y)处。可以使用负数，不过这样会把部分窗口移出屏幕的可视区域。

- resizeBy(dw,dh)——相对于浏览器窗口的当前大小，把它口的宽度调整 dw 个像素，高度调整 dy 个像素。dw 为负数，把缩小窗口的宽度，dy 为负数，缩小窗口的高度。

- resizeTo(w,h)——把窗口的宽度调整为 w，高度调整为 h。不能使用负数。

例如：

```
//move the window right by 10 pixels and down by 20 pixels
window.moveBy(10, 20);

//resize the window to have a width of 150 and a height of 300
window.resizeTo(150, 300);

//resize the window to be 150 pixels wider, but leave the height alone
window.resizeBy(150, 0);

//move back to the upper-left corner of the screen (0,0)
window.moveTo(0, 0);
```

假设既调整了窗口大小，又调整了它的位置，却没有记录这些改变。现在需要知道该窗口在屏幕上的位置以及它的尺寸。由于缺乏相应的标准，就产生了问题。

- IE 提供了 window.screenLeft 和 window.screenTop 对象来判断窗口的位置，但未提供任何判断窗口大小的方法。用 document.body.offsetWidth 和 document.body. offsetHeight 属性可以获取视口的大小（显示 HTML 页的区域），但它们不是标准属性。

- Mozilla提供window.screenX和window.screenY属性判断窗口的位置。它还提供了window.innerWidth和window.innerHeight属性来判断视口的大小，window.outerWidth和window.outerHeight属性判断浏览器窗口自身的大小。

- Opera 和 Safari 提供与 Mozilla 相同的工具。

所以，问题变成了解用户使用的浏览器。

尽管移动浏览器窗口和调整它的大小是种很酷的操作，但应该尽量少用它们。移动浏览器窗口和调整它的大小会对用户产生影响，因此专业的 Web 站点和 Web 应用程序都避免使用它们。

第 5 章 浏览器中的 JavaScript

5.3 BOM(2)

2. 导航和打开新窗口

用 JavaScript 可以导航到指定的 URL，并用 window.open()方法打开新窗口。该方法接受四个参数，即要载入新窗口的页面的 URL、新窗口的名字（为目

标所用）、特性字符串和说明是否用新载入的页面替换当前载入的页面的Boolean值。一般只用前三个参数，因为最后一个参数只有在调用window.open()方法却不打开新窗口时才有效。

如果用已有框架的名字作为window.open()方法的第二个参数调用它，那么URL所指的页面就会被载入该框架。例如，要把页面载入名为“topFrame”的框架，可以使用下面的代码：

```
window.open("http://www.wrox.com/", "topFrame");
```

这行代码的行为就像用户点击一个链接，该链接的href为http://www.wrox.com/，target为"topFrame"。专用的框架名_self、_parent、_top和_blank也是有效的。

如果声明的框架名无效，window.open()将打开新窗口，该窗口的特性由第三个参数（特性字符串）决定。如果省略第三个参数，将打开新的浏览器窗口，就像点击了target被设置为_blank的链接。这意味着新浏览器窗口的设置与默认浏览器窗口的设置（工具栏、地址栏和状态栏都是可见的）完全一样。

如果使用第三个参数，该方法就假设应该打开新窗口。特性字符串是用逗号分隔的设置列表，它定义新创建的窗口的某些方面。下表显示了各种设置：

设置	值	说　明
left	Number	说明新创建的窗口的左坐标。不能为负数*
top	Number	说明新创建的窗口的上坐标。不能为负数*
height	Number	设置新创建的窗口的高度。该数字不能小于100*
width	Number	设置新创建的窗口的宽度。该数字不能小于100*
resizable	yes,no	判断新窗口是否能通过拖动边线调整大小。默认值是no
scrollable	yes,no	判断新窗口的视口容不下要显示的内容时是否允许滚动。默认值是no
toolbar	yes,no	判断新窗口是否显示工具栏。默认值是no
status	yes,no	判断新窗口是否显示状态栏。默认值是no
location	yes,no	判断新窗口是否显示（Web）地址栏。默认值是no

* 将在第19章详细讨论这些浏览器的安全性特征。

如前所述，特性字符串是用逗号分隔的，因此在逗号或等号前后不能有空格。例如，下面的字符串是无效的：

```
window.open("http://www.wrox.com/", "wroxwindow",
            "height=150, width= 300, top=10, left= 10, resizable =yes");
```

由于逗号后和几个等号前后的空格，所以该字符串无效。删除空格，它就能正常运行：

```
window.open("http://www.wrox.com/", "wroxwindow",
            "height=150,width=300,top=10,left=10,resizable=yes");
```

window.open()方法将返回 window 对象作为它的函数值，该 window 对象就是新创建的窗口（如果给定的名字是已有的框架名，则为框架）。用这个对象，可以操作新创建的窗口：

```
var oNewWin = window.open("http://www.wrox.com/", "wroxwindow",
                          "height=150,width=300,top=10,left=10,resizable=yes");

oNewWin.moveTo(100, 100);
oNewWin.resizeTo(200, 200);
```

还可以使用该对象调用 close()方法关闭新创建的窗口：

```
oNewWin.close();
```

如果新创建的窗口中有代码，还可以用下面的代码关闭其自身：

```
window.close();
```

这段代码只对新创建的窗口有效。如果在主浏览器窗口中调用 window.close()方法，将得到一条消息：提示该脚本试图关闭窗口，询问是否真的要关闭该窗口。通用规则是，脚本可以关闭它们打开的任何窗口，但不能关闭其他窗口。

新窗口还有对打开它的窗口的引用，存放在 opener 属性中。只在新窗口的最高层 window 对象才有 opener 属性，这样用 top.opener 访问它会更安全。例如：

```
var oNewWin = window.open("http://www.wrox.com/", "wroxwindow",
                          "height=150,width=300,top=10,left=10,resizable=yes");

alert(oNewWin.opener == window);   //outputs "true"
```

在这个例子中，打开一个新窗口，然后测试它的 opener 属性是否等于 window 对象，从而证明 opener 属性确实指向 window 对象（该警告显示"true"）。

某些情况下，打开新窗口对用户有帮助，但一般说来，最好尽量少弹出窗口。许多企业都突然开始引入 Web 站点上的弹出式广告，大多数用户对此都觉得很讨厌。于是，许多用户都安装了弹出式窗口的拦截程序，除非用户允许打开某些弹出式窗口，否则它将拦截所有弹出式窗口。记住，弹出式窗口拦截程序并不知道合法弹出式窗口与广告之间的区别，因此最好在弹出窗口时警告用户。

3. 系统对话框

除弹出新的浏览器窗口，还可使用其他方法向用户弹出信息，即利用window对象的alert()、confirm()和prompt()方法。

你已对调用 alert()方法的语法了如指掌，因为迄今为止许多示例中都使用了该方法。它只接受一个参数，即要显示给用户的文本。调用 alert()方法后，浏览器将创建一个具有 OK 按钮的系统消息框，显示指定的文本。例如，下面的代码将显示图 5-6 所示的消息框：

```
alert("Hello world! ");
```

通常在提示用户注意某些不能控制的东西（如错误）时，使用警告对话框。通常用户在表单中输入无效数据时，显示警告对话框。

图 5-6

第二种类型的对话框通过调用 confirm()方法显示的。确认对话框看起来与警告对话框相似，因为都是向用户显示信息。这两种对话框的主要区别是确认对话框中除 OK 按钮外还有 Cancel 按钮，这样允许用户说明是否执行指定的动作。例如，下面的代码显示的确认对话框如图 5-7 所示：

```
confirm("Are you sure? ");
```

图 5-7

为判断用户点击的是OK按钮还是Cancel按钮，confirm()方法返回一个Boolean值，如果点击的是OK按钮，返回true，点击的是Cancel按钮，返回false。确认对话框的典型用法如下：

```
if (confirm("Are you sure? ")) {
    alert("I'm so glad you're sure! ");
} else {
     alert("I'm sorry to hear you're not sure. ");
}
```

在这个例子中，第一行代码向用户显示确认对话框，这个 confirm()方法是 if 语句的条件。如果用户点击 OK 按钮，显示的警告消息是"I’m so glad you’re sure!"，如果用户点击的是 Cancel 按钮，则显示的警告消息是"I’m sorry to hear you’re not sure!"。通常在用户尝试删除某些东西（例如删除他或她信箱中的电子邮件）时显示这种类型的提示。

最后的对话框通过调用 prompt()方法显示，如你所料，该对话框提示用户输入某些信息。除 OK 按钮和 Cancel 按钮外，该对话框还有文本框，要求用户在此输入某些数据。prompt()方法接受两个参数，即要显示给用户的文本和文本框中的默认文本（如果不需要，可以是空串）。下面的代码将显示图 5-8 所示的对话框：

```
prompt("What's your name? ", "Michael");
```

图 5-8

如果点击 OK 按钮，将文本框中的值作为函数值返回。如果点击 Cancel 按钮，返回 null。通常以下面的方式使用 prompt()方法：

```
var sResult = prompt("What is your name? ", "");
if (sResult != null) {
 alert("Welcome, " + sResult);
}
```

关于这三种对话框，还有最后几点要说。首先，所有对话框窗口都是系统窗口，意味着不同的操作系统（有时是不同的浏览器）显示的窗口可能不同。这也意味着你对以什么字体、颜色等外观显示窗口没有任何控制权。

其次，这些对话框是模式化的，意思是如果用户未点击 OK 按钮或 Cancel 按钮关闭该对话框，就不能在浏览器窗口中做任何操作。这是控制用户行为，以确保安全交付重要信息的通用方法。

4. **状态栏**

状态栏是底部边界内的区域，用于向用户显示信息（如图 5-9 所示）。

状态栏

图　5-9

一般说来，状态栏告诉了用户何时在载入页面，何时完成载入页面。但可以用 window 对象的两个属性设置它的值，即 status 和 defaultStatus 属性。如你所料，

status 可以使状态栏的文本暂时改变，而 defaultStatus 则可在用户离开当前页面前一直改变该文本。例如，在第一次载入页面时，可使用默认的状态栏消息：

```
window.defaultStatus = "You are surfing www.wrox.com. ";
```

你也许还想在用户把鼠标移到某个链接上时显示该链接的信息：

```
<a href="books.htm" onmouseover="window.status='Information on Wrox books.'
">Books</a>
```

在使用 JavaScript URL 时这点非常有用，因为默认情况下，当鼠标移到链接上时，浏览器默认在状态栏中显示 href 特性的值。设置 window.status 属性，可向用户隐瞒链接实现的细节：

```
<a href="javascript:goSomewhere(1,2,3,4)" onmouseover="window.status='Information
on Wrox books.' ">Books</a>
```

注意不要过度使用状态栏，以使它分散用户的注意力。例如，许多站点仍使用滚动消息代码在状态栏中滚动文本。这种技巧不仅没用，且很讨厌，非常的不专业，给那些完全可以不用它的 Web 站点或 Web 应用程序添加了业余的感觉。由于本书介绍的是专业 JavaScript，所以不介绍滚动文本代码。但如果你对此感兴趣，可以参考 http://javascript.internet.com/ scrolls/，在此可以找到大量此类脚本。

5. 时间间隔和暂停

Java 开发者熟悉对象的 wait()方法，可使程序暂停，在继续执行下一行代码前，等待指定的时间量。这种功能非常有用，遗憾地是，JavaScript 未提供相应的支持。但这种功能并非完全不能实现，有几种方法可以采用。

JavaScript 支持暂停和时间间隔，这可有效地告诉浏览器应该何时执行某行代码。所谓暂停，是在指定的毫秒数后执行指定的代码。时间间隔是反复执行指定的代码，每次执行之间等待指定的毫秒数。

可以用 window 对象的 setTimeout()方法设置暂停。该方法接受两个参数，要执行的代码和在执行它之前要等待的毫秒数（1/1000 秒）。第一个参数可以是代

码串（与 eval()函数的参数相同），也可是函数指针。例如，下面的代码都在 1 秒钟后显示一条警告：

```
setTimeout("alert('Hello world!') ", 1000);
setTimeout(function() { alert("Hello world!"); }, 1000);
```

当然，还可以引用以前定义的函数：

```
function sayHelloWorld() {
    alert("Hello world!");
}

setTimout(sayHelloWorld, 1000);
```

调用 setTimeout()时，它创建一个数字暂停 ID，与操作系统中的进程 ID 相似。暂停 ID 本质上是要延迟的进程的 ID，在调用 setTimeout()后，就不应该再执行它的代码。要取消还未执行的暂停，可调用 clearTimeout()方法，并将暂停 ID 传递给它：

```
var iTimeoutId = setTimeout("alert('Hello world!')", 1000);

//nevermind
clearTimeout(iTimeoutId);
```

你也许会问：“为什么要定义暂停，又在执行它之前将其取消呢?”请考虑现在大多数应用程序中可见的工具提示。当把鼠标移动到一个按钮上时，停留一会，等待出现黄色的文本框，提示该按钮的功能。如果只是短暂地把鼠标移到该按钮上，然后很快将其移到另一个按钮上，那么第一个按钮的工具提示就不会显示。这就是要在执行暂停代码前取消它的原因。因为你在执行代码前只想等待指定的时间量。如果用户的操作产生了不同的结果，则需要取消该暂停。

时间间隔与暂停的运行方式相似，只是它无限次地每隔指定的时间段就重复一次指定的代码。可调用 setInterval()方法设置时间间隔，它的参数与 setTimeout()相同，是要执行的代码和每次执行之间等待的毫秒数。例如：

```
setInterval("alert('Hello world!') ", 1000);
setInterval(function() { alert("Hello world!"); }, 1000);

function sayHelloWorld() {
    alert("Hello world!");
}

setInterval(sayHelloWorld, 1000);
```

此外，与 setTimeout()类似，setInterval()方法也创建时间间隔 ID，以标识要执行的代码。ClearInterval()方法可用这个 ID 阻止再次执行该代码。显然，这一点在使用时间间隔时更重要，因为如果不取消时间间隔，就会一直执行它，直到页面被卸载为止。下面是时间间隔用法的一个常见示例：

```
var iNum = 0;
var iMax = 100;
var iIntervalId = null;

function incNum() {
    iNum++;

    if (iNum == iMax) {
        clearInterval(iIntervalId);
    }
}

iIntervalId = setInterval(incNum, 500);
```

在这段代码中，每隔 500 毫秒，就对数字 iNum 进行一次增量运算，直到它达到最大值（iMax），此时该时间间隔将被清除。也可用暂停实现该操作，这样即不必跟踪时间间隔的 ID，代码如下：

```
var iNum = 0;
var iMax = 100;

function incNum() {
    iNum++;

    if (iNum != iMax) {

            setTimeout(incNum, 500);
        }
    }

    setTimeout(incNum, 500);
```

这段代码使用链接暂停，即 setTimeout()执行的代码也调用了 setTimeout()。如果在执行过增量运算后，iNum 不等于 iMax，就调用 setTimeout()方法。不必跟踪暂停 ID，也不必清除它，因为代码执行后，将销毁暂停 ID。

那么应该使用哪种方法呢？这由使用情况决定。在执行一组指定的代码前等待一段时间，则使用暂停。如果要反复执行某些代码，就使用时间间隔。

第 5 章 浏览器中的 JavaScript

5.3 BOM(3)

6. 历史

可以访问浏览器窗口的历史。所谓历史，是用户访问过的站点的列表。出于安全原因，所有导航只能通过历史完成，不能得到浏览器历史中包含的页面的URL。

不必通过时间机器实现历史导航，只需使用window对象的history属性及它的相关方法即可。

go()方法只有一个参数，即前进或后退的页面数。如果是负数，就在浏览器历史中后退。如果是正数，就前进（这种差别就像 Back 和 Forward 按钮之间的差别）。

因此，后退一页，可用下面的代码：

```
window.history.go(-1);
```

当然，window 对象的引用不是必需的，也可使用下面的代码：

```
history.go(-1);
```

通常用该方法创建网页中嵌入的 Back 按钮，例如：

```
<a href="javascript:history.go(-1)">Back to the previous page</a>
```

要前进一页，只需要使用正数 1：

```
history.go(1);
```

另外，用 back()和 forward()方法可以实现同样的操作：

```
//go back one
history.back();

//go forward one
history.forward();
```

这些代码更有意义一些，因为它们精确地反应出浏览器的 Back 和 Forward 按钮的行为。

虽然不能使用浏览器历史中的 URL，但可以用 length 属性查看历史中的页面数：

```
alert("There are currently " + history.length + " pages in history.");
```

如果想前进或后退多个页面，想知道是否可以这样做，那么上面的代码就非常有用。

5.3.2 document 对象

document 对象实际上是 window 对象的属性，如你所知，window 对象的任何属性和方法都可直接访问，所以下面这行代码将返回"true"：

```
alert(window.document == document);
```

这个对象的独特之处是它是唯一一个既属于 BOM 又属于 DOM（下一章将讨论 DOM 中的 document 对象）的对象。从 BOM 角度看，document 对象由一系列集合构成，这些集合可以访问文档的各个部分，并提供页面自身的信息。再有，由于 BOM 没有可以指导实现的标准，所以每个浏览器实现的 document 对象都稍有不同，这一节的重点是最常用的功能。

下表列出了 BOM 的 document 对象的一些通用属性：

属　性	说　明
alinkColor	激活的链接的颜色，如<body alink="color">定义的*
bgColor	页面的背景颜色，如<body bgcolor="color">定义的*
fgColor	页面的文本颜色，如<body text="color">定义的*
lastModified	最后修改页面的日期，是字符串

linkColor	链接的颜色，如<body link="color">定义的*
referrer	浏览器历史中后退一个位置的 URL
title	<title/>标签中显示的文本
URL	当前载入的页面的 URL
vlinkColor	访问过的链接的颜色，如<body vlink="color">定义的*

* 反对使用这些属性，因为它们引用了<body/>标签中的旧 HTML 特性。应该用样式表脚本代替它们。

lastModified 属性获取的是最后一次修改页面的日期的字符串表示，用作旁注，除非你想在主页上显示最后的修改日期（用服务器端的技术也可实现）。同样，referrer 属性用处也不大，除非你想跟踪用户是从哪里链接过来的（也许可以查看该用户是通过 Google 或是其他搜索引擎访问你的站点的）。同样也可以用服务器端的技术实现它。

title 属性是可读写的，所以可随时改变页面的标题，无论 HTML 页面的内容是什么。当站点使用了框架，且只有一个框架改变，其他框架保持不变时，该属性非常有用。可以用该属性改变框架的标题，从而反映出载入了新页面：

```
top.document.title = "New page title";
```

URL 属性也是可读可写的，所以可用它获取当前页面的 URL，或者把它设置为新的 URL，把窗口导航到新页面。例如：

```
document.URL = "http://www.wrox.com/";
```

如前所述，document 对象也有许多集合，提供对载入的页面的各个部分的访问。下表列出了这些集合：

集　合	说　明
anchors	页面中所有锚的集合（由表示）
applets	页面中所有 applet 的集合
embeds	页面中所有嵌入式对象的集合（由<embed/>标签表示）
forms	页面中所有表单的集合
images	页面中所有图像的集合
links	页面中所有链接的集合（由<a href="somewhere.htm"><a/>表示）

与 window.frame 集合相似，可用数字或名字引用 document 对象的每个集合，也就是说可用 document.images[0]或 document.images["image_name"]访问图像。考虑下面的 HTML 页面：

```
<html>
    <head>
        <title>Document Example</title>
    </head>
    <body>
        <p>Welcome to my <a href="home.htm">home</a> away from home.</p>
        <img src="home.jpg" align="right" name="imgHome" />
        <form method="post" action="submit.cgi" name="frmSubscribe">
            <input type="text" name="txtEmail" />
            <input type="submit" value="Subscribe" />
        </form>
    </body>
</html>
```

访问该文档各个部分的方法如下：

- 用 document.links[0]访问链接；
- 用 document.images[0]或 document.images["imgHome"]访问图像；
- 用 document.forms[0]或 document.forms["frmSubscribe"]访问表单。

此外，链接和图像等的所有特性都变成了该对象的属性。例如，document.images[0].src 是获取第一个图像的 src 特性的代码。

最后，BOM 的 document 对象还有几个方法。最常用的方法之一是 write()或它的兄弟方法 writeln()。这两个方法都接受一个参数，即要写入文档的字符串。如你所料，它们之间唯一的区别是 writeln()方法将在字符串末尾加一个换行符（\n）。

这两个方法都会把字符串的内容插入到调用它们的位置。这样浏览器就会像处理页面中的常规 HTML 代码一样处理这个文档字符串。考虑下面的页面：

```
<html>
    <head>
        <title>Document Write Example</title>
    </head>
    <body>
        <h1><script type="text/javascript">document.write("this is a
test")</script></h1>
     </body>
</html>
```

该页面在浏览器中看来与下面的页面一样：

```
<html>
    <head>
        <title>Document Write Example</title>
    </head>
    <body>
        <h1>this is a test</h1>
     </body>
</html>
```

可以使用这种功能动态地引入外部 JavaScript 文件。例如：

```
<html>
    <head>
        <title>Document Example</title>
        <script type="text/javascript">
            document.write("<script type=\"text/javascript\" src=\"external.js\">"
                           + "</scr" + "ipt>");
        </script>
    </head>
    <body>

    </body>
</html>
```

这段代码在页面上写了一个<script/>标签，将使浏览器像常规一样载入外部 JavaScript 文件。注意字符串"</script>"被分成两部分（"</src"和"ipt>"）。这是必要的，因为每当浏览器遇到</script>，它都假定其中的代码块是完整的（即使它出现在 JavaScript 字符串中）。假设前面的例子未把"</script>"分成两部分：

```
<html>
    <head>
        <title>Document Example</title>
        <script type="text/javascript">
            document.write("<script type=\"text/javascript\" src=\"external.js\">"
                           + "</script>");  //this will cause a problem
        </script>
    </head>
    <body>

    </body>
</html>
```

浏览器显示如下网页：

```
<html>
    <head>
        <title>Document Example</title>
        <script type="text/javascript">
            document.write("<script type=\"text/javascript\" src=\"external.js\">"
        </script>
        </script>
    </head>

     <body>

     </body>
 </html>
```

可以看到，忘记把字符串"</script>"分成两部分引起了严重的混乱。首先，在<script/>标签内有个语法错，因为 document.write()的调用漏掉了闭括号。其次，有两个</script>标签。这就是在使用 document.write()方法把<script/>标签写入页面时一定要把"</script>"字符串分开的原因。

记住，要插入内容属性，必须在完全载入页面前调用 write()和 writeln()方法。如果任何一个方法是在页面载入后调用的，它将抹去页面的内容，显示指定的内容。

第 5 章 浏览器中的 JavaScript

5.3 BOM(4)

与 write()和 writeln()方法密切相关的是 open()和 close()方法。open()方法用于打开已经载入的文档以便进行编写，close()方法用于关闭 open()方法打开的文档，本质上是告诉它显示写入其中的所有内容。通常把这些方法结合在一起，用于向框架或新打开的窗口中写入内容，如下所示：

```
var oNewWin = window.open("about:blank", "newwindow",
                          "height=150,width=300,top=10,left=10,resizable=yes");

oNewWin.document.open();
oNewWin.document.write("<html><head><title>New Window</title></head>");
oNewWin.document.write("<body>This is a new window!</body></html>");
oNewWin.document.close();
```

这个例子打开空白页（使用本地的"about:blank"URL），然后写入新页面。要正确实现这一操作，在调用 write()前，先调用 open()方法。写完后，调用 close()方法完成显示。当你想显示无需返回服务器的页面时，这种方法非常有用。

5.3.3 location 对象

BOM 中最有用的对象之一是 location 对象，它是 window 对象和 document 对象的属性（对此没有什么标准，导致了一些混乱）。location 对象表示载入窗口的 URL，此外，它还可以解析 URL：

- hash——如果 URL 包含#，该方法将返回该符号之后的内容（例如，http://www.somewhere.com/index#selection1 的 hash 等于"#selection1"）。
- host——服务器的名字（如 www.wrox.com）。
- hostname——通常等于 host，有时会省略前面的 www。
- href——当前载入的页面的完整 URL。
- pathname——URL 中主机名后的部分。例如，http://www.somewhere.com/pictures/index.htm 的 pathname 是"/pictures/index.htm"。
- port——URL 中声明的请求的端口。默认情况下，大多数 URL 没有端口信息，所以该属性通常是空白的。像 http://www.somewhere.com:8080/index.htm 这样的 URL 的 port 属性等于 8080。
- protocol——URL中使用的协议，即双斜杠(//)之前的部分。例如，http://www.wrox.com 中的protocol属性等于http:， ftp://www.wrox.com的protocol属性等于ftp:。
- search——执行 GET 请求的 URL 中的问号（?）后的部分，又称为查询字符串。例如，http://www.somewhere.com/search.htm?term=javascript 中的 search 属性等于 ?term=javascript

location.href 是最常用的属性，用于获取或设置窗口的 URL（在这一点上，它类似于 document.URL 属性）。改变该属性的值，就可导航到新页面：

```
location.href = "http://www.wrox.com/";
```

采用这种方式导航，新地址将被加到浏览器的历史栈中，放在前一个页面后，意味着 Back 按钮会导航到调用了该属性的页面。

assign()方法实现的是同样的操作：

```
location.assign("http://www.wrox.com");
```

这两种方法都可以采用，不过大多数开发者选用 location.href 属性，因为它更精确地表达了代码的意图。

如果不想让包含脚本的页面能从浏览器历史中访问，可使用 replace()方法。该方法所作的操作与 assign()方法一样，但它多了一步操作，即从浏览器历史中删除包含脚本的页面，这样就不能通过浏览器的 Back 和 Forward 按钮访问它了。你可以自己试试看：

```
<html>
    <head>
        <title>You won't be able to get back here</title>
    </head>
    <body>
        <p>Enjoy this page for a second, because you won't be coming back here.</p>
        <script type="text/javascript">
            setTimeout(function () {
                location.replace("http://www.wrox.com/");
            }, 1000);
        </script>
    </body>
</html>
```

把这个页面载入浏览器中，等它导航到新页面后，再点击 Back 按钮。

location 对象还有个 reload()方法，可重新载入当前页面。reload()方法有两种模式，即从浏览器缓存中重载，或从服务器端重载。究竟采用哪种模式由该方法的参数值决定，如果是 false，则从缓存中载入，如果是 true，则从服务器端载入（如果省略参数，默认值为 false）。

因此，要从服务器端重载当前页面，可以使用下面的代码：

```
location.reload(true);
```

要从缓存重载当前页面，可以采用下面两行代码中的任意一行：

```
location.reload(false);
location.reload();
```

reload()方法调用后的代码可能被执行，也可能不被执行，这由网络延迟和系统资源等因素决定。因此，最好把 reload()调用放在最后一行。

location 对象的最后一个方法是 toString()，它仅返回 location.href 的值。因此，下面两行代码是等价的：

```
alert(location);
alert(location.href);
```

本节采用示例介绍了 location 对象。记住，location 对象是 window 对象和 document 对象的属性，所以 window.location 和 document.location 互相等价，可以交换使用。

5.3.4 navigator 对象

navigator 对象是最早实现的 BOM 对象之一，Netscape Navigator 2.0 和 IE 3.0 引入了它。它包含大量有关 Web 浏览器的信息。它也是 window 对象的属性，可以用 window.navigator 引用它，也可以用 navigator 引用。

虽然微软公司最初把 Netscape 的浏览器称为 navigator，但 navigator 对象成了一种事实标准，用于提供 Web 浏览器的信息。（微软除 navigator 外，还有自己的对象 clientInfomation，但它们两个提供的数据完全相同。）

同样，缺乏标准阻碍了 navigator 对象的发展，因为各个浏览器决定支持该对象的属性和方法。下表列出了最常用的属性和方法以及最常用的四种浏览器（IE、Mozilla、Opera 和 Safari）中哪个支持它们。

属性/方法	说　明	IE	Moz	Op	Saf
appCodeName	浏览器代码名的字符串表示（如"Mozilla"）	×	×	×	×
appName	官方浏览器名的字符串表示	×	×	×	×
appMinorVersion	额外版本信息的字符串表示	×	—	—	—
appVersion	浏览器版本信息的字符串表示	×	×	×	×
browserLanguage*	浏览器或操作系统的语言的字符串表示	×	—	×	—
cookieEnabled	说明是否启用了 cookie 的 Boolean 值	×	×	×	—
cpuClass	CPU 类别的字符串表示（"x86"、"68K"、"Alpha"、"PPC"或"other"）	×	—	—	—
javaEnabled()	说明是否启用了 Java 的 Boolean 值	×	×	×	×
language	浏览器语言的字符串表示	—	×	×	×

（续）

属性/方法	说　明	IE	Moz	Op	Saf
mimeTypes	注册到浏览器的 mime 类型的数组	—	×	×	×
onLine	说明浏览器是否连接到因特网上的 Boolean 值	×	—	—	—
oscpu	操作系统或 CPU 的字符串表示	—	×	—	—
platform	运行浏览器的计算机平台的字符串表示	×	×	×	×
plugins	安装在浏览器中的插件的数组	×	×	×	×
preference()	用于设置浏览器首选项的函数	—	×	×	—
product	产品名的字符串表示（如"Gecko"）	—	×	—	×
productSub	有关产品的额外信息的字符串表示（如 Gecko 版本）	—	×	—	×
opsProfile		—	—	—	—
securityPolicy		—	×	—	—
systemLanguage*	操作系统语言的字符串表示	×	—	—	—
taintEnabled()	说明是否启用了数据感染的 Boolean 值	×	×	×	×
userAgent	用户代理头字符串的字符串表示	×	×	×	×
userLanuage*	操作系统语言的字符串表示	×	—	—	—
userProfile	允许访问浏览器用户档案的对象	×	—	—	—
vendor	品牌浏览器名的字符串表示（如"Netscape6"或"Netscape"）	—	×	—	×
vendorSub	品牌浏览器的额外信息的字符串表示（如 Netscape 的版本）	—	×	—	×

* 大多数情况下，browserLanguage、systemLanguage 和 userLanguage 相同。

在判断浏览器页面采用的是哪种浏览器方面时，navigator 对象非常有用。在因特网上可迅速检索到许多检测浏览器的方法，它们都大量地利用了 navigator 对象。第 9 章将介绍如何用 navigator 对象检测浏览器及操作系统。

5.3.5 screen 对象

虽然出于安全原因，有关用户系统的大多数信息都被隐藏了，但还可以用 screen 对象获取某些关于用户屏幕的信息（不出所料，它也是 window 对象的属性）。

screen 对象通常包含下列属性（不过，许多浏览器都加入了自己的属性）：

- availHeight——窗口可以使用的屏幕的高度（以像素计），其中包括操作系统元素（如 Windows 工具栏）需要的空间。

- availWidth——窗口可以使用的屏幕的宽度（以像素计）。
- colorDepth——用户表示颜色的位数。大多数系统采用 32 位。
- height——屏幕的高度，以像素计。
- width——屏幕的宽度，以像素计。

确定新窗口的大小时，availHeight 和 availWidth 属性非常有用。例如，可以使用下面的代码填充用户的屏幕：

```
window.moveTo(0, 0);
window.resizeTo(screen.availWidth, screen.availHeight);
```

此外，这些数据与站点的流量工具一起使用，可以判断用户的图形接受能力。

第 5 章 浏览器中的 JavaScript

5.4 小结

这一章介绍了 Web 浏览器中的 JavaScript。它涵盖了把 JavaScript 加入 HTML 和 SVG 页的方法，并解释了两者之间的差别。此外，还讨论了 XHTML 对将 JavaScript 加入 HTML 页的影响以及如何为此做好准备。

在本章后面的小节中，学到了 BOM 及它提供的各种对象。了解了 window 对象是 JavaScript 世界的核心，其他所有 BOM 对象都不过是 window 对象的属性。

这一章解释了如何操作浏览器窗口和框架，用 JavaScript 移动它们，调整它们的大小。用 location 对象，可以访问和改变窗口的地址。用 history 对象，可以在用户访问过的页面中前进或后退。

最后，学到了如何用 navigator 对象和 screen 对象获取用户浏览器和屏幕的信息。

第 6 章 DOM 基础

6.1 什么是 DOM?

自从第一次使用 HTML 将因特网上相关的文档连接起来后，DOM 也许是 Web 上最伟大的创新了。DOM 给予开发者空前的对 HTML 的访问能力，并使开发者能将 HTML 作为 XML 文档来处理和查看。DOM 代表着由微软公司和 Netscape 公司所引领的动态 HTML 向真正跨平台的、语言独立的解决方案的重要演变。

6.1 什么是 DOM?

在开始详细介绍什么是 DOM 之前，你首先要了解是什么促使了它的诞生。尽管 DOM 很大程度上受到浏览器中动态 HTML 发展的影响，但 W3C 还是将它最先应用于 XML。

第 6 章 DOM 基础

6.1.1 XML 简介

6.1.1 XML 简介

XML（可扩展标记语言）是从称为 SGML（标准通用标记语言）的更加古老的语言派生出来的。SGML 的主要目的是定义使用标签来表示数据的标记语言的语法。

标签由包围在一个小于号（<）和一个大于号（>）之间的文本组成，例如<tag>。起始标签（start tag）表示一个特定区域的开始，例如<start>；结束标签（end tag）定义了一个区域的结束，除了在小于号之后紧跟着一个斜线（/）外，和起始标签基本一样，例如</end>。SGML 还定义了标签的特性（attribute），它们是定义在小于号和大于号之间的值，例如<img src="picture.jpg">中的 src 特性。如果你觉得它看起来很熟悉的话，应该知道，基于 SGML 的语言的最著名实现就是原始的 HTML。

SGML 常用来定义针对 HTML 的文档类型定义（DTD），同时它也常用于编写 XML 的 DTD。SGML 的问题就在于，它允许出现一些奇怪的语法，这让创建 HTML 的解析器成为一个大难题：

- 某些起始标签不允许出现结束标签，例如 HTML 中<img>标签。包含了结束标签就会出现错误。

- 某些起始标签可以选择性出现结束标签或者隐含了结束标签，例如 HTML 中<p>标签，当出现另一个<p>标签或者某些其他标签时，便假设在这之前有一个结束标签。

- 某些起始标签要求必须出现结束标签，例如 HTML 中<script>标签。

- 标签可以以任何顺序嵌套。即使结束标签不按照起始标签的逆序出现也是允许的，例如，This is a sample string是正确的。

- 某些特性要求必须包含值，例如<img src="picture.jpg">中的 src 特性。

- 某些特性不要求一定有值，例如<td nowrap>中的 nowrap 特性。

- 定义特性的两边有没有加上双引号都是可以的，所以<img src="picture.jpg">和<img src=picture.jpg>都是允许的。

这些问题使建立一个 SGML 语言的解析器变成了一项艰巨的任务。判断何时应用以上规则的困难导致了 SGML 语言的定义一直停滞不前。以这些问题作为出发点，XML 逐渐步入我们的视野。

XML 去掉了之前令许多开发人员头疼的 SGML 的随意语法。在 XML 中，采用了如下的语法：

- 任何的起始标签都必须有一个结束标签。

- 可以采用另一种简化语法，可以在一个标签中同时表示起始和结束标签。这种语法是在大于符号之前紧跟一个斜线（/），例如<tag />。XML 解析器会将其翻译成<tag></tag>。

- 标签必须按合适的顺序进行嵌套，所以结束标签必须按镜像顺序匹配起始标签，例如this is a sample string。这好比是将起始和结束标签看作是数学中的左右括号：在没有关闭所有的内部括号之前，是不能关闭外面的括号的。

- 所有的特性都必须有值。

- 所有的特性都必须在值的周围加上双引号。

这些规则使得开发一个 XML 解析器要简便得多，而且也除去了解析 SGML 中花在判断何时何地应用那些奇怪语法规则上的工作。仅仅在 XML 出现后的前六年就衍生出多种不同的语言，包括 MathML、SVG、RDF、RSS、SOAP、XSLT、XSL-FO，而同时也将 HTML 改进为 XHTML。

如果需要关于 SGML 和 XML 具体技术上的对比，请查看 W3C 的注解，位于：

http://www.w3. org/TR/NOTE-sgml-xml.html

如今，XML 已经是世界上发展最快的技术之一。它的主要目的是使用文本以结构化的方式来表示数据。在某些方面，XML 文件也类似于数据库，提供数据的结构化视图。这里是一个 XML 文件的例子：

```
<?xml version="1.0"?>
<books>
    <!-- begin the list of books -->
    <book isbn="0764543555">
        <title>Professional JavaScript for Web Developers</title>
        <author>Nicholas C. Zakas</author>
        <desc><![CDATA[
Professional JavaScript for Web Developers brings you up to speed on the latest
innovations in the world of JavaScript. This book provides you with the details of
JavaScript implementations in Web browsers and introduces the new capabilities
relating to recently-developed technologies such as XML and Web Services.
        ]]></desc>
    </book>
```

141

```
    <?page render multiple authors ?>
    <book isbn="0764570773">
        <title>Beginning XML, 3rd Edition</title>
        <author>David Hunter</author>
        <author>Andrew Watt</author>
        <author>Jeff Rafter</author>
        <author>Jon Duckett</author>
        <author>Danny Ayers</author>
        <author>Nicholas Chase</author>
        <author>Joe Fawcett</author>
        <author>Tom Gaven</author>
        <author>Bill Patterson</author>
        <desc><![CDATA[
Beginning XML, 3rd Edition, like the first two editions, begins with a broad
overview of the technology and then focuses on specific facets of the various
specifications for the reader. This book teaches you all you need to know about XML:
what it is, how it works, what technologies surround it, and how it can best be used
in a variety of situations, from simple data transfer to using XML in your Web
pages. It builds on the strengths of the first and second editions, and provides new
material to reflect the changes in the XML landscape - notably RSS and SVG.
        ]]></desc>
    </book>
    <book isbn="0764543555">
        <title>Professional XML Development with Apache Tools</title>
        <author>Theodore W. Leung</author>
        <desc><![CDATA[
If you're a Java programmer working with XML, you probably already use some of the
tools developed by the Apache Software Foundation. This book is a code-intensive
guide to the Apache XML tools that are most relevant for Java developers, including
Xerces, Xalan, FOP, Cocoon, Axis, and Xindice.
        ]]></desc>
    </book>********
</books>
```

每个 XML 文档都由 XML 序言开始，在前面的代码中的第一行便是 XML 序言，<?xml version="1.0"?>。这一行代码会告诉解析器和浏览器，这个文件应该按照前面讨论过的 XML 规则进行解析。第二行代码，<books>，则是文档元素（document element），它是文件中最外面的标签（我们认为元素（element）是起始标签和结束标签之间的内容）。所有其他的标签必须包含在这个标签之内来组成一个有效的 XML 文件。XML 文件的第二行并不一定要包含文档元素；如果有注释或者其他内容，文档元素可以迟些出现。

范例文件中的第三行代码是注释，你会发现它与 HTML 中使用的注释风格是一样的。这是 XML 从 SGML 中继承的语法元素之一。

页面再往下的一些地方，可以发现<desc>标签里有一些特殊的语法。<![CDATA[ ]]>代码用于表示无需进行解析的文本，允许诸如大于号和小于号之类的特殊字符包含在文本中，而无需担心破坏 XML 的语法。文本必须出现在

<![CDATA[和]]>之间才能合适地避免被解析。这样的文本称为 Character Data Section，简称 CData Section。

下面的一行就是在第二本书的定义之前的：

```
<?page render multiple authors ?>
```

虽然它看上去很像 XML 序言，但实际上是一种称为处理指令（processing instruction）的不同类型的语法。处理指令（以下简称 PI）的目的是为了给处理页面的程序（例如 XML 解析器）提供额外的信息。PI 通常情况下是没有固定格式的，唯一的要求是紧随第一个问号必须至少有一个字母。在此之后，PI 可以包含除了小于号和大于号之外的任何字符串序列。

最常见的 PI 是用来指定 XML 文件的样式表：

```
<?xml-stylesheet type="text/css" href="MyStyles.css" ?>
```

这个 PI 一般会直接放在 XML 序言之后，通常由 Web 浏览器使用，来将 XML 数据以特殊的样式显示出来。

如果你对 XML 感兴趣，想学习更多关于它及其应用的内容，请参见人民邮电出版社即将出版的《XML 与 DOM 基础教程》。

第 6 章 DOM 基础

6.1.2 针对 XML 的 API

将 XML 定义为一种语言之后，就出现了使用常见的编程语言（如 Java）来同时表现和处理 XML 代码的需求。

首先出现的是 Java 上的 SAX（Simple API for XML）项目。SAX 提供了一个基于事件的 XML 解析的 API。从其本质上来说，SAX 解析器从文件的开头出发，从前向后解析，每当遇到起始标签或者结束标签、特性、文本或者其他的 XML 语法时，就会触发一个事件。然后，当事件发生时，具体要怎么做就由开发人员决定。

因为 SAX 解析器仅仅按照文本的方式来解析它们，所以 SAX 更轻量、更快速。而它们的主要缺点是在解析中无法停止、后退或者不从文件开始，直接访问 XML 结构中的指定部分。

DOM 是针对 XML 的基于树的 API。它关注的不仅仅是解析 XML 代码，而是使用一系列互相关联的对象来表示这些代码，而这些对象可以被修改且无需重新解析代码就能直接访问它们。

使用 DOM，只需解析代码一次来创建一个树的模型；某些时候会使用 SAX 解析器来完成它。在这个初始解析过程之后，XML 已经完全通过 DOM 模型来表现出来，同时也不再需要原始的代码。尽管 DOM 比 SAX 慢很多，而且，因为创建了相当多的对象而需要更多的开销，但由于它使用上的简便，因而成为 Web 浏览器和 JavaScript 最喜欢的方法。

注意 DOM 是语言无关的 API，这意味着它的实现并不与 Java、JavaScript 或者其他语言绑定。然而，鉴于本书的目的，我将大部分的注意力放在 JavaScript 的实现上。

第 6 章 DOM 基础

6.1.3 节点的层次

那么基于树的 API 到底指什么呢？当谈论 DOM 树（也称之为文档）的时候，实际上谈论的是节点（node）的层次。DOM 定义了 Node 的接口以及许多种节点类型来表示 XML 节点的多个方面：

- Document——最顶层的节点，所有的其他节点都是附属于它的。
- DocumentType——DTD 引用（使用<!DOCTYPE >语法）的对象表现形式，例如<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">。它不能包含子节点。
- DocumentFragment——可以像 Document 一样来保存其他节点。

❑ Element——表示起始标签和结束标签之间的内容，例如<tag></tag>或者<tag/>。这是唯一可以同时包含特性和子节点的节点类型。

❑ Attr——代表一对特性名和特性值。这个节点类型不能包含子节点。

❑ Text——代表 XML 文档中的在起始标签和结束标签之间，或者 CData Section 内包含的普通文本。这个节点类型不能包含子节点。

❑ CDataSection——<![CDATA[]]>的对象表现形式。这个节点类型仅能包含文本节点 Text 作为子节点。

❑ Entity——表示在 DTD 中的一个实体定义，例如<!ENTITY foo "foo">。这个节点类型不能包含子节点。

❑ EntityReference——代表一个实体引用，例如"。这个节点类型不能包含子节点。

❑ ProcessingInstruction——代表一个 PI。这个节点类型不能包含子节点。

❑ Comment——代表 XML 注释。这个节点类型不能包含子节点。

❑  Notation——代表在 DTD 中定义的记号。这个很少用到，所以在本书中不会讨论。

一个文档是由任意数量的节点的层次组成。考虑下面的 XML 文档：

```
<?xml version="1.0"?>
<employees>
    <!-- only employee -->
    <employee>
        <name>Michael Smith</name>
        <position>Software Engineer</position>
        <comments><![CDATA[
          His birthday is on 8/14/68.
        ]]></comments>
    </employee>""
</employees>
```

这段代码可以用一个 DOM 文档。

在图 6-1 中，每个矩形代表在 DOM 文档树中的一个节点，粗体文本表示节点的类型，非粗体的文本代表该节点的内容。

注释和<employee/>节点都被认为是<employees/>的子节点，因为它们在这棵树中直接在<employees/>节点的下面。同样的，我们也认为<employees/>是注释和<employee/>节点的父节点。

类似的，<name/>、<position/>以及<comments/>都被认为是<employee/>的子节点，同时，因为它们在 DOM 树中处于同一层上，有着相同的父节点，所以认为它们是兄弟（sibiling）关系。

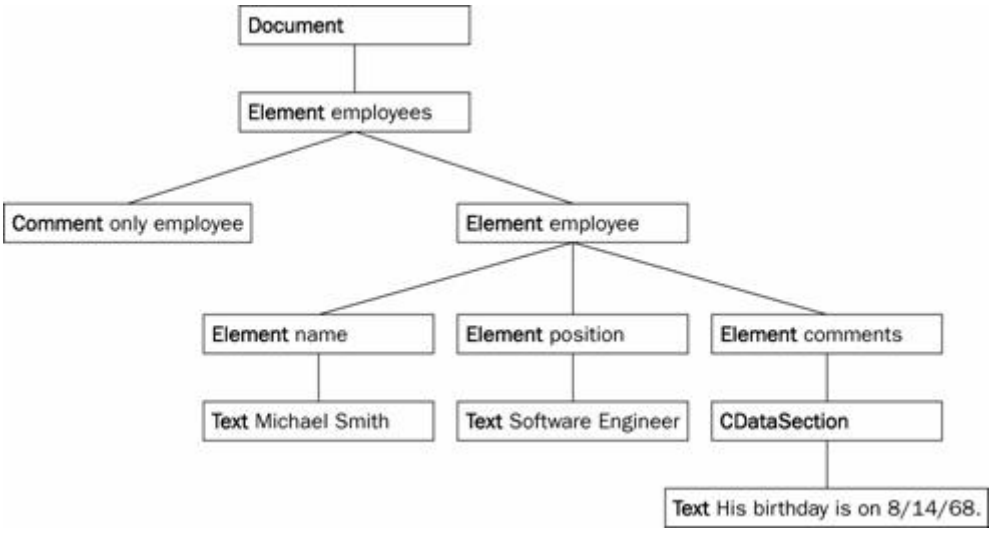

图 6-1

我们还认为<employees/>节点是这一节中所有节点的祖先，其中包括它自己的子节点（注释和<employee/>）以及子节点的子节点（<name/>、<position/>，等等，直到文本节点"His birthday is on 8/14/68"）。并认为文档节点是文档中所有节点的祖先。

Node 接口定义了对应不同节点类型的 12 个常量（它们会在即将讨论的 nodeType 特性中使用到）：

- Node.ELEMENT_NODE (1)

- Node.ATTRIBUTE_NODE (2)
- Node.TEXT_NODE (3)
- Node.CDATA_SECTION_NODE (4)
- Node.ENTITY_REFERENCE_NODE (5)
- Node.ENTITY_NODE (6)
- Node.PROCESSING_INSTRUCTION_NODE (7)
- Node.COMMENT_NODE (8)
- Node.DOCUMENT_NODE (9)
- Node.DOCUMENT_TYPE_NODE (10)
- Node.DOCUMENT_FRAGMENT_NODE (11)
- Node.NOTATION_NODE (12)

Node 接口也定义了一些所有节点类型都包含的特性和方法。我们在下面的表格中列出了这些特性和方法：

特性/方法	类型/返回类型	说　明
nodeName	String	节点的名字；根据节点的类型而定义
nodeValue	String	节点的值；根据节点的类型而定义
nodeType	Number	节点的类型常量值之一
ownerDocument	Document	指向这个节点所属的文档
firstChild	Node	指向在 childNodes 列表中的第一个节点
lastChild	Node	指向在 childNodes 列表中的最后一个节点
childNodes	NodeList	所有子节点的列表
previousSibling	Node	指向前一个兄弟节点；如果这个节点就是第一个兄弟节点，那么该值为 null
nextSibling	Node	指向后一个兄弟节点；如果这个节点就是最后一个兄弟节点，那么该值为 null

hasChildNodes()	Boolean	当 childNodes 包含一个或多个节点时，返回真
attributes	NamedNodeMap	包含了代表一个元素的特性的 Attr 对象；仅用于 Element 节点
appendChild(node)	Node	将 *node* 添加到 childNodes 的末尾
removeChild(node)	Node	从 childNodes 中删除 node
replaceChild (newnode, oldnode)	Node	将 childNodes 中的 oldnode 替换成 newnode
insertBefore (newnode, refnode)	Node	在 childNodes 中的 refnode 之前插入 newnode

除节点外，DOM 还定义了一些助手对象，它们可以和节点一起使用，但不是 DOM 文档必有的部分。

- NodeList——节点数组，按照数值进行索引；用来表示一个元素的子节点。

- NamedNodeMap——同时用数值和名字进行索引的节点表；用于表示元素特性。

这些助手对象为处理 DOM 文档提供附加的访问和遍历方法。具体用法将在后面讨论。

第 6 章 DOM 基础

6.1.4 特定语言的 DOM

任何基于 XML 的语言，如 XHTML 和 SVG，因为它们从技术上来说还是 XML，仍然可以利用刚刚介绍的核心 DOM。然而，很多语言会继续定义它们自己的 DOM 来扩展 XML 核心以提供语言的特色功能。

开发 XML DOM 的同时，W3C 还一起开发了一种特别针对 XHTML（以及 HTML）的 DOM。这个 DOM 定义了一个 HTMLDocument 和一个 HTMLElement 作为这种实现的基础。每个 HTML 元素通过它自己的 HTMLElement 类型来表示，例如 HTMLDivElement 代表了 ，但有少数元素除外，它们只包含 HTMLElement 提供的属性和方法。在本书后面的部分中，会不断介绍不同的 HTML DOM 的特点以及核心 XML DOM 的特点。

普通 HTML 并不是合法的 XML。然而，大部分当前的 Web 浏览器都很宽容，依然可以将一个 HTML 文档解析为合适的 DOM 文档（即使没有 XML 序言）。但是，在编写 Web 页面的时候最好使用 XHTML 代码以消除那些坏的编码习惯。

W3C也发布了一些特定语言的DOM，包括SVG（http://www.w3.org/TR/SVG），SMIL Animation（http://www.w3.org/TR/smil-animation）和MathML（http://www.w3.org/TR/MathML2）。

第 6 章 DOM 基础

6.2 对 DOM 的支持

正如前面所说的，并不是所有的浏览器对 DOM 的支持都一样。一般来说，Mozilla 对 DOM 标准支持最好，支持几乎所有的 DOM Level 2，以及部分 DOM Level 3。在 Mozilla 之后，Opera 和 Safari 也在完全支持上做了突出工作，极大地缩小了和标准之间的差距，支持几乎所有的 DOM Level 1 和大部分 DOM Level 2。这个领域中落在最后的是 IE，它对 DOM Level 1 的实现都还不完整，尚有很多方面有待完善。

第 6 章 DOM 基础

6.3 使用 DOM

document 对象是 BOM 的一部分，同时也是 HTML DOM 的 HTMLDocument 对象的一种表现形式，反过来说，它也是 XML DOM Document 对象。JavaScript 中的大部分处理 DOM 的过程都利用 document 对象，所以我们从这里开始讨论还是比较合理的。

第 6 章 DOM 基础

6.3.1 访问相关的节点

在下面的几节中考虑下面的 HTML 页面：

```
<html>
    <head>
        <title>DOM Example</title>
    </head>
    <body>
        <p>Hello World!</p>
        <p>Isn't this exciting?</p>
        <p>You're learning to use the DOM!</p>
    </body>
</html>
```

要访问<html/>元素（你应该明白这是该文件的 document 元素），你可以使用 document 的 documentElement 特性：

```
var oHtml = document.documentElement;
```

由于IE 5.5 中的DOM实现的错误，document.documentElement会返回<body/>元素。IE 6.0 已经修复了这个错误。

现在变量 oHtml 包含一个表示<html/>的 HTMLElement 对象。如果你想取得<head/>和<body/>元素，下面的可以实现：

```
var oHead = oHtml.firstChild;
var oBody = oHtml.lastChild;
```

也可以使用 childNodes 特性来完成同样的工作。只需把它当成普通的 JavaScript Array，使用方括号标记：

```
var oHead = oHtml.childNodes[0];
var oBody = oHtml.childNodes[1];
```

你还可以通过使用 childNodes.length 特性来获取子节点的数量：

```
alert(oHtml.childNodes.length);    //outputs "2"
```

注意方括号标记其实是 NodeList 在 JavaScript 中的简便实现。实际上正式的从 childNodes 列表中获取子节点的方法是使用 item()方法：

```
var oHead = oHtml.childNodes.item(0);
var oBody = oHtml.childNodes.item(1);
```

HTML DOM 页定义了 document.body 作为指向<body/>元素的指针：

```
var oBody = document.body;
```

有了 oHtml、oHead 和 oBody 这三个变量，就可以先尝试确定它们之间的关系：

```
alert(oHead.parentNode == oHtml);         //outputs "true"
alert(oBody.parentNode == oHtml);         //outputs "true"
alert(oBody.previousSibling == oHead);    //outputs "true"
alert(oHead.nextSibling == oBody);        //outputs "true"
alert(oHead.ownerDocument == document);   //outputs "true"
```

这一小段代码测试并验证了 oBody 和 oHead 的 parentNode 特性都是指向 oHtml 变量，同时使用 previousSibling 和 nextSibling 特性来建立它们之间的关系。最后一行确认了 oHead 的 ownerDocument 特性事实上是指向该文档。

不同浏览器在判断何为 Text 节点上存在一些差异。某些浏览器，如 Mozilla，认为元素之间的空白都是 Text 节点；而另一些浏览器，如 IE，会全部忽略这些空白。因为使用 Mozilla 方式很难确定哪些空白是 Text 节点，本书将采用 IE 的方式。

第 6 章 DOM 基础

6.3.2 检测节点类型

我们可以通过使用 nodeType 特性检验节点类型：

```
alert(document.nodeType);     //outputs "9"
alert(document.documentElement.nodeType);    //outputs "1"
```

这个例子中，document.nodeType 返回 9，等于 Node.DOCUMENT_NODE；同时 document.documentElement.nodeType 返回 1，等于 Node.ELEMENT_NODE。

也可以用 Node 常量来匹配这些值：

```
alert(document.nodeType == Node.DOCUMENT_NODE);     //outputs "true"
alert(document.documentElement.nodeType == Node.ELEMENT_NODE);    //outputs "true"
```

这段代码可以在 Mozilla 1.0+、Opera 7.0+和 Safari 1.0+上正常运行。不幸的是，IE 不支持这些常量，所以这些代码在 IE 上会产生错误。所幸，可以通过定义匹配节点类型的常量来纠正这种情况，正如下面这样：

```
if (typeof Node == "undefined") {
    var Node = {
        ELEMENT_NODE: 1,
        ATTRIBUTE_NODE: 2,
        TEXT_NODE: 3,
        CDATA_SECTION_NODE: 4,
        ENTITY_REFERENCE_NODE: 5,
        ENTITY_NODE: 6,
        PROCESSING_INSTRUCTION_NODE: 7,
        COMMENT_NODE: 8,
        DOCUMENT_NODE: 9,
        DOCUMENT_TYPE_NODE: 10,
        DOCUMENT_FRAGMENT_NODE: 11,
        NOTATION_NODE: 12
    }
}
```

当然，另外一种选择是直接使用数值（不过这容易让人混淆，因为很多人无法记住这些节点类型的值）。

第 6 章 DOM 基础

6.3.3 处理特性

正如前面提到的，即便 Node 接口已具有 attributes 方法，且已被所有类型的节点继承，然而，只有 Element 节点才能有特性。Element 节点的 attributes 属性其实是 NamedNodeMap，它提供一些用于访问和处理其内容的方法：

- getNamedItem(*name*)——返回 nodeName 属性值等于 *name* 的节点；
- removeNamedItem(*name*)——删除 nodeName 属性值等于 *name* 的节点；
- setNamedItem(*node*)——将 *node* 添加到列表中，按其 nodeName 属性进行索引；
-  item(*pos*)——像 NodeList 一样，返回在位置 *pos* 的节点；

请记住这些方法都是返回一个 Attr 节点，而非特性值。

NamedNodeMap 对象也有一个 length 属性来指示它所包含的节点的数量。

当 NamedNodeMap 用于表示特性时，其中每个节点都是 Attr 节点，它的 nodeName 属性被设置为特性名称，而 nodeValue 属性被设置为特性的值。例如，假设有这样一个元素：

```
<p style="color: red" id="p1">Hello world!</p>
```

同时，假设变量 oP 包含指向这个元素的一个引用。于是可以这样访问 id 特性的值：

```
var sId = oP.attributes.getNamedItem("id").nodeValue;
```

当然，还可以用数值方式访问 id 特性，但这样稍微有些不直观：

```
var sId = oP.attributes.item(1).nodeValue;
```

还可以通过给 nodeValue 属性赋新值来改变 id 特性：

```
oP.attributes.getNamedItem("id").nodeValue = "newId";
```

Attr 节点也有一个完全等同于（同时也完全同步于）nodeValue 属性的 value 属性，并且有 name 属性和 nodeName 属性保持同步。我们可以随意使用这些属性来修改或变更特性。

因为这个方法有些累赘，DOM 又定义了三个元素方法来帮助访问特性：

- getAttribute(*name*)——等于 attributes.getNamedItem(*name*).value；
- setAttribute(*name*, *newvalue*)——等于 attribute.getNamedItem(*name*).value = *newvalue*；
- removeAttribute(*name*)——等于 attributes.removeNamedItem(*name*)。

这些方法相当有用，可以直接处理特性值，完全地隐藏 Attr 节点。所以，要获取前面用的<p/>的 id 特性，只需这样做：

```
var sId = oP.getAttribute("id");
```

同时要更改 ID，可以这样做

```
oP.setAttribute("id", "newId");
```

正如你所看到的，这些方法要比使用 NamedNodeMap 的方法简洁得多。

第 6 章 DOM 基础

6.3.4 访问指定节点

现在已经知道如何访问父节点和子节点，但是如果想访问文档中位置很深的某个节点（或者一组节点），要怎么做呢？当然，你不想逐个检查子节点直到遇到要访问的那个节点。为在这种情况下助你一臂之力，DOM 提供一些方法来方便地访问指定的节点。

1. **getElementsByTagName()**

核心（XML）DOM 定义了 getElementsByTagName()方法，用来返回一个包含所有的 tagName(标签名)特性等于某个指定值的元素的 NodeList。在 Element 对象中，tagName 特性总是等于小于号之后紧随的名称——例如，<img />的 tagName 是"img"。下一行代码返回文档中所有<img />元素的列表：

```
var oImgs = document.getElementsByTagName("img");
```

在把所有图形都存于 oImgs 后，只需使用方括号标记或者 item()方法（getElementsByTag- Name()返回一个和 childNodes 一样的 NodeList），就可以像访问子节点那样逐个访问这些节点了：

```
alert(oImgs[0].tagName);    //outputs "IMG"
```

这行代码输出第一个图像的 tagName（标签名），输出的是"IMG"。由于某些原因，大部分浏览器按照大写来记录标签名，即使 XHTML 约定指出标签名应当全部小写：

但是假如你只想获取在某个页面第一个段落的所有图像呢？可以通过对第一个段落元素调用 getElementsByTagName()来完成，像这样：

```
var oPs = document.getElementsByTagname("p");
var oImgsInP = oPs[0].getElementsByTagName("img");
```

可以使用一个星号的方法来获取 document 中的所有元素：

```
var oAllElements = document.getElementsByTagName("*");
```

这行代码可以返回 document 中包含的所有元素而不管它们的标签名。

当参数是一个星号的时候，IE 6.0 并不返回所有的元素。必须使用 document.all 来替代它。

2. **getElementsByName()**

HTML DOM定义了getElementsByName()，它用来获取所有name特性等于指定值的元素的。考虑下面的HTML：

```
<html>
    <head>
        <title>DOM Example</title>
    </head>
    <body>
        <form method="post" action="dosomething.cgi">
            <fieldset>
                <legend>What color do you like?</legend>
                <input type="radio" name="radColor" value="red" /> Red<br />
                <input type="radio" name="radColor" value="green" /> Green<br />
                <input type="radio" name="radColor" value="blue" /> Blue<br />
            </fieldset>
            <input type="submit" value="Submit" />
        </form>
    </body>
</html>
```

这个页面会询问用户喜欢哪种颜色。所有单选按钮都用同样的名称（name特性），因为只要这个字段返回一个值（即选定的选项的value特性）即可。若要获得所有单选按钮元素的引用，可以使用下面的代码：

```
var oRadios = document.getElementsByName("radColor");
```

然后，就可以像处理其他元素那样处理这些单选按钮了：

```
alert(oRadios[0].getAttribute("value"));  //outputs "red"
```

IE 6.0和Opera 7.5在这个方法的使用上还存在一些错误。首先，它们还会返回id等于给定名称的元素。第二，它们仅仅检查<input/>和<img/>元素。

3. getElementById()

这是HTML DOM定义的第二种方法，它将返回id特性等于指定值的元素。在HTML中，id特性是唯一的——这意味着没有两个元素可以共享同一个id。毫无疑问这是从文档树中获取单个指定元素最快的方法。

假设有下列HTML页面：

```
<html>
    <head>
        <title>DOM Example</title>
    </head>
    <body>
        <p>Hello World!</p>
        <div id="div1">This is my first layer</div>
    </body>
</html>
```

要访问 ID 为"div1"的 元素，可以使用 getElementsByTagName()：

```
var oDivs = document.getElementsByTagName("div");
var oDiv1 = null;
for (var i=0; i < oDivs.length; i++){
    if (oDivs[i].getAttribute("id") == "div1") {
        oDiv1 = oDivs[i];
        break;
    }
}
```

或者，可以使用 getElementById()：

```
var oDiv1 = document.getElementById("div1");
```

可以看到，这种获取指定元素的引用的方法效率更高。

如果给定的 ID 匹配某个元素的 name 特性，IE 6.0 还会返回这个元素。这是一个 bug，也是你必须非常小心的一个问题。

第 6 章 DOM 基础

6.3.5 创建和操作节点

迄今为止，已经学过了如何访问文档中的不同节点，不过这仅仅是使用 DOM 所能实现的功能中的很小一部分。还能添加、删除、替换（或者其他操作）DOM 文档中的节点。正是这些功能使得 DOM 具有真正意义上的动态性。

1. 创建新节点

DOM Document（文档）中有一些方法用于创建不同类型的节点，即便在所有的浏览器中的浏览器 document 对象并不需要全部支持所有的方法。下面的表格列出了包含在 DOM Level 1 中的方法，并列出不同的浏览器是否支持项。

方　法	描　述	IE	MOZ	OP	SAF
createAttribute	用给定名称 name 创建特性节点	×	×	×	—

(name)					
createCDATASection (text)	用包含文本text的文本子节点创建一个CDATA Section	—	×	—	—
createComment(text)	创建包含文本 text 的注释节点	×	×	×	×

（续）

方　法	描　述	IE	MOZ	OP	SAF
createDocument Fragment()	创建文档碎片节点	×	×	×	×
createElement (tagname)	创建标签名为 tagname 的元素	×	×	×	×
createEntity Reference(name)	创建给定名称的实体引用节点	—	×	—	—
createProcessing Instruction(target, data)	创建包含给定 target 和 data 的 PI 节点	—	×	—	—
createTextNode(text)	创建包含文本 text 的文本节点	×	×	×	×

注：IE = Windows 的 IE 6；MOZ = 任意平台的 Mozilla 1.5；OP=任意平台的 Opera 7.5；SAF=MacOS 的 Safari 1.2

最常用到的几个方法是：createDocumentFragment()、createElement()和createTextNode()；其他的一些方法要么就是没什么用（createComment()），要么就是浏览器的支持不够，目前还不太能用。

2. createElement()、createTextNode()、appendChild()

假设有如下 HTML 页面：

```
<html>
    <head>
        <title>createElement() Example</title>
    </head>
    <body>

    </body>
</html>
```

现在想使用 DOM 来添加下列代码到上面这个页面中：

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

这里可以使用 createElement()和 createTextNode()来达到目的。下面是实现步骤：

首先，创建<p/>元素：

```
var oP = document.createElement("p");
```

第二，创建文本节点：

```
var oText = document.createTextNode("Hello World!");
```

下一步，把文本节点加入到元素中。可以用在本章前面简要提到的 appendChild()方法来完成这个任务。每种节点类型都有 appendChild()方法，它的用途是将给定的节点添加到某个节点的 childNodes 列表的尾部。在这个例子中，应将文本节点追加到<p/>元素中：

```
oP.appendChild(oText);
```

不过还没完成全部操作。已经创建了一个<p/>元素和一个文本节点，并且将它们关联在一起了，但这个元素在文档中仍然没有一席之地。要实际可见，必须将这个元素附加到 document.body 元素或者其中任意子节点上。然后，可以再次使用 appendChild()方法：

```
document.body.appendChild(oP);
```

要把这些代码放到可运行范例中，只要创建一个包含每一步的函数，并且使用 onload 事件句柄在页面载入后调用这个函数（关于事件将在第 9 章详细介绍）：

```
<html>
    <head>
        <title>createElement() Example</title>
        <script type="text/javascript">
            function createMessage() {
                var oP = document.createElement("p");
                var oText = document.createTextNode("Hello World! ");
                oP.appendChild(oText);
                document.body.appendChild(oP);
            }
        </script>
    </head>
    <body onload="createMessage()">

    </body>
</html>
```

当运行这段代码时，"Hello World!"的消息会自动显示，就好像它本来就是这个 HTML 文档的一部分。

在这里，我必须谨慎地告诉你所有的 DOM 操作必须在页面完全载入之后才能进行。当页面正在载入时，要向 DOM 插入相关代码是不可能的，因为在页面完全下载到客户端机器之前，是无法完全构建 DOM 树的。因为这个原因，必须使用 onload 事件句柄来执行所有的代码。

3. removeChild()、replaceChild()和 insertBefore()

自然的，可以添加一个节点，当然也可以删除一个节点，这就是 removeChild()所要做的事。这个方法接受一个参数，要删除的节点，然后将这个节点作为函数的返回值返回。所以，例如，如果有个已经包含"Hello World!"消息的页面，要把这个消息删除，可以使用类似下面方法：

```
<html>
    <head>
        <title>removeChild() Example</title>
        <script type="text/javascript">
            function removeMessage() {
                var oP = document.body.getElementsByTagName("p")[0];
                document.body.removeChild(oP);
            }
        </script>
    </head>
    <body onload="removeMessage()">
        <p>Hello World!</p>

    </body>
</html>
```

这个页面载后，显示空白页面，因为在你能瞥到消息之前，它已经被删除了。尽管它能运行，最好还是使用节点的 parentNode 特性来确保每次你都能访问到它真正的父节点：

```
<html>
    <head>
        <title>removeChild() Example</title>
        <script type="text/javascript">
            function removeMessage() {
                var oP = document.body.getElementsByTagName("p")[0];
                oP.parentNode.removeChild(oP);
            }
        </script>
    </head>
    <body onload="replaceMessage()">
        <p>Hello World!</p>
    </body>
</html>
```

但假如想将这个消息替换成新的内容，要怎么做呢？如果是这种情况，则可以使用 replaceChild()方法。

replaceChild()方法有两个参数：被添加的节点和被替换的节点。这样，可以创建一个包含新消息的元素，并用它替换原来包含"Hello World!"消息的<p/>元素。

```
<html>
    <head>
        <title>replaceChild() Example</title>
        <script type="text/javascript">
            function replaceMessage() {
                var oNewP = document.createElement("p");
                var oText = document.createTextNode("Hello Universe! ");
                oNewP.appendChild(oText);
                var oOldP = document.body.getElementsByTagName("p")[0];
                oOldP.parentNode.replaceChild(oNewP, oOldP);
            }
        </script>
    </head>
    <body onload="replaceMessage()">
        <p>Hello World!</p>
    </body>
</html>
```

这个范例页面将消息"Hello World!"替换成"Hello Universe!"注意这段代码仍然使用 parentNode 特性来确保使用了正确父节点来进行操作。

当然，可能想让两个消息同时出现。如果想让新消息出现在老消息之后，只要使用 appendChild()方法：

```
<html>
    <head>
        <title>appendChild() Example</title>
```

```
        <script type="text/javascript">
            function appendMessage() {
                var oNewP = document.createElement("p");
                var oText = document.createTextNode("Hello Universe! ");
                oNewP.appendChild(oText);
                document.body.appendChild(oNewP);
            }
        </script>
    </head>
    <body onload="appendMessage()">
        <p>Hello World!</p>
    </body>
</html>
```

然而，如果想让新消息出现在旧消息之前，就使用 insertBefore()方法。这个方法接受两个参数：要添加的节点和插在哪个节点之前。在这个例子中，第二个参数是包含"Hello World!"的<p/>元素：

```
<html>
    <head>
        <title>insertBefore() Example</title>
        <script type="text/javascript">
            function insertMessage() {
                var oNewP = document.createElement("p");
                var oText = document.createTextNode("Hello Universe! ");
                oNewP.appendChild(oText);
                var oOldP = document.getElementsByTagName("p")[0];
                document.body.insertBefore(oNewP, oOldP);
            }
        </script>
    </head>
    <body onload="insertMessage()">
        <p>Hello World!</p>
    </body>
</html>
```

4. createDocumentFragment()

 一旦把节点添加到 document.body（或者它的后代节点）中，页面就会更新并反映出这个变化。对于少量的更新，这是很好的，就像在前面的例子中那样。然而，当要向 document 添加大量数据时，如果逐个添加这些变动，这个过程有可能会十分缓慢。为解决这个问题，可以创建一个文档碎片，把所有的新节点附加其上，然后把文档碎片的内容一次性添加到 document 中。

假设你想创建十个新段落。若使用前面学到的方法，可能会写出这种代码：

```
var arrText = ["first", "second", "third", "fourth", "fifth", "sixth", "seventh",
"eighth", "ninth", "tenth"];

for (var i=0; i < arrText.length; i++) {
    var oP = document.createElement("p");
    var oText = document.createTextNode(arrText[i]);
    oP.appendChild(oText);
    document.body.appendChild(oP);}
```

这段代码运行良好，但问题是它调用了十次 document.body.appendChild()，每次都要产生一次页面刷新。这时，文档碎片就十分有用：

```
var arrText = ["first", "second", "third", "fourth", "fifth", "sixth", "seventh",
"eighth", "ninth", "tenth"];

var oFragment = document.createDocumentFragment();

for (var i=0; i < arrText.length; i++) {
    var oP = document.createElement("p");
    var oText = document.createTextNode(arrText[i]);
    oP.appendChild(oText);
    oFragment.appendChild(oP);
}

document.body.appendChild(oFragment);
```

在这段代码中，每个新的<p/>元素都被添加到文档碎片中。然后，这个碎片被作为参数传递给 appendChild()。这里对 appendChild()的调用实际上并不是把文档碎片节点本身追加到<body/>元素中；而是仅仅追加碎片中的子节点。然后，可以看到很明显的性能提升：调用 document.body.appendChild()一次来替代十次，这意味着只需要进行一次屏幕刷新。

第 6 章 DOM 基础

6.4 HTML DOM 特征功能

核心 DOM 的特性和方法是通用的，是为了在各种情况下操作所有 XML 文档而设计的。HTML DOM 的特性和方法是在专门针对 HTML 的同时也让一些 DOM 操作更加简便。这包括将特性作为属性进行访问的能力，以及特定于元素的属性和方法，这些扩展可以完成一些常见的任务，例如搭建表格，更加简便快速。

第 6 章 DOM 基础

6.4.1 让特性像属性一样

大部分情况下，HTML DOM元素中包 含的所有特性都是可作为属性。例如，假设有如下图像元素：

如果要使用核心的DOM来获取和设置src和border特性，那么要用getAttribute()和setAttribute()方法：

```
alert(oImg.getAttribute("src"));
alert(oImg.getAttribute("border"));
oImg.setAttribute("src", "mypicture2.jpg");
   oImg.setAttribute("border", "1");
```

然而，使用HTML DOM，可以使用同样名称的属性来获取和设置这些值：

```
alert(oImg.src);
alert(oImg.border);
oImg.src = "mypicture2.jpg";
oImg.border = "1";
```

唯一的特性名和属性名不一样的特例是class特性，它是用来指定应用于某个元素的一个CSS类，例如：

```
<div class="header"></div>
```

因为class在ECMAScript中是一个保留字，在JavaScript中，它不能被作为变量名、属性名或者函数名。于是，相应的属性名就变成className：

```
alert(oDiv.className);
oDiv.className = "footer";
```

通过使用属性来访问特性的方式来替代getAttribute()和setAttribute()并无实质性的益处，除了可能缩减代码的长度以及令代码变得易读。

IE在**setAttribute()**上有个很大的问题：当你使用它时，变更并不会总是正确地反应出来。如果你打算支持IE，最好尽可能使用属性。

第6章 DOM基础

6.4.2 table方法

假设想使用DOM来创建如下的HTML表格：

```
<table border="1" width="100%">    <tbody>
        <tr>
            <td>Cell 1,1</td>
            <td>Cell 2,1</td>
        </tr>
        <tr>
            <td>Cell 1,2</td>
            <td>Cell 2,2</td>
        </tr>
    </tbody>
</table>
```

如果想通过核心 DOM 方法来完成这个任务，你的代码可能会像这样：

```
//create the table
var oTable = document.createElement("table");
oTable.setAttribute("border", "1");
oTable.setAttribute("width", "100%");

//create the tbody
var oTBody = document.createElement("tbody");
oTable.appendChild(oTBody);

//create the first row
var oTR1 = document.createElement("tr");
oTBody.appendChild(oTR1);
var oTD11 = document.createElement("td");
oTD11.appendChild(document.createTextNode("Cell 1,1"));
oTR1.appendChild(oTD11);
var oTD21 = document.createElement("td");

oTD21.appendChild(document.createTextNode("Cell 2,1"));
oTR1.appendChild(oTD21);

//create the second row
var oTR2 = document.createElement("tr");
oTBody.appendChild(oTR2);
var oTD12 = document.createElement("td");
oTD12.appendChild(document.createTextNode("Cell 1,2"));
oTR2.appendChild(oTD12);
var oTD22 = document.createElement("td");
oTD22.appendChild(document.createTextNode("Cell 2,2"));
oTR2.appendChild(oTD22);
//add the table to the document body
document.body.appendChild(oTable);
```

这段代码十分的冗长而且有些难于理解。为了协助建立表格，HTML DOM 给<table/>、<tbody/>和<tr/>等元素添加了一些特性和方法。

给<table/>元素添加了以下内容：

- caption——指向<caption/>元素（如果存在）；
- tBodies——<tbody/>元素的集合；
- tFoot——指向<tfoot/>元素（如果存在）；

- tHead——指向<thead/>元素（如果存在）；
- rows——表格中所有行的集合；
- createTHead()——创建<thead/>元素并将其放入表格；
-  createTFoot()——创建<tfoot/>元素并将其放入表格；
- createCaption()——创建<caption/>元素并将其放入表格；
- deleteTHead()——删除<thead/>元素；
- deleteTFoot()——删除<tfoot/>元素；
- deleteCaption()——删除<caption />元素；
- deleteRow(*position*)——删除指定位置上的行；
- insertRow(*position*)——在 rows 集合中的指定位置上插入一个新行。

<tbody/>元素添加了以下内容：

- rows——<tbody/>中所有行的集合；
- deleteRow(*position*)——删除指定位置上的行；
- insertRow(*position*)——在 rows 集合中的指定位置上插入一个新行。

<tr/>元素中添加了以下内容：

- cells——<tr/>元素中所有的单元格的集合；
- deleteCell(*position*)——删除给定位置上的单元格；
- insertCell(*position*)——在 cells 集合的给定位置上插入一个新的单元格。

上面的一切意味着什么？简单地说，它意味着如果使用这些简便的属性和方法就可以大大降低创建表格的复杂度：

```
//create the table
var oTable = document.createElement("table");
oTable.setAttribute("border", "1");
oTable.setAttribute("width", "100%");

//create the tbody
var oTBody = document.createElement("tbody");
oTable.appendChild(oTBody);
//create the first row
oTBody.insertRow(0);
oTBody.rows[0].insertCell(0);
oTBody.rows[0].cells[0].appendChild(document.createTextNode("Cell 1,1"));
oTBody.rows[0].insertCell(1);
oTBody.rows[0].cells[1].appendChild(document.createTextNode("Cell 2,1"));

//create the second row
oTBody.insertRow(1);
oTBody.rows[1].insertCell(0);
oTBody.rows[1].cells[0].appendChild(document.createTextNode("Cell 1,2"));
oTBody.rows[1].insertCell(1);
oTBody.rows[1].cells[1].appendChild(document.createTextNode("Cell 2,2"));
//add the table to the document body
document.body.appendChild(oTable);
```

在这段代码中，创建<table/>和<tbody/>元素的方式没有改变。在这一段中改变的是如何创建两个表格行，用到了 HTML DOM Table（表格）的属性和方法。要创建第一行，对<tbody/>元素调用 insertRow()方法，并传给它一个参数 0，表示新增的一行应该放在什么位置上。然后，可以通过 oTBody.rows[0]来引用新增的这一行，因为这一行已经被自动创建并添加到<tbody/>元素中的第 0 个位置。

以类似的方式可以创建单元格——对<tr/>元素调用 insertCell()并传入要创建单元格的位置。可以通过 oTBody.rows[0].cells[0]来引用新创建的这个单元格，因为单元格已经被创建并插入到这一行的第 0 个位置。

虽然从技术角度上来说，两种代码都是正确的，但是使用这些特性和方法来创建表格使得代码变得更加有逻辑且更加易读。

第 6 章 DOM 基础

6.5 遍历 DOM

到目前为止，我们讨论的功能都仅仅是 DOM Level 1 的部分。本节将介绍一些 DOM Level 2 的功能，尤其是和遍历 DOM 文档相关的 DOM Level 2 遍历（traversal）和范围（range）规范中的对象。这些功能只有在 Mozilla 和 Konqueror/Safari 中才有。

第 6 章 DOM 基础

6.5.1 NodeIterator

第一个有关的对象是 NodeIterator，用它可以对 DOM 树进行深度优先的搜索，如果要查找页面中某个特定类型的信息（或者元素），这是相当有用的。要理解 NodeIterator 到底做了什么，考虑下面的 HTML 页面：

```
<html>
    <head>
        <title>Example</title>
    </head>
    <body>
        <p>Hello <b>World!</b></p>
    </body>
</html>
```

这个页面可以转换成由图 6-2 表示的 DOM 树。

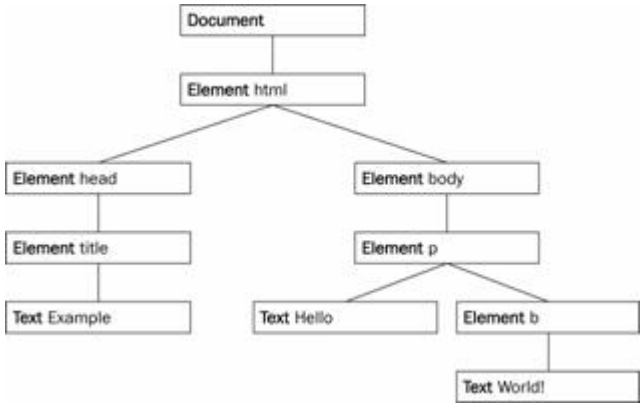

图 6-2

当使用 NodeIterator 时，可以从 document 元素（<html/>）开始，按照一种系统的路径——也就是大家熟知的深度优先搜索——遍历整个 DOM 树。在这种搜索方式中，遍历从父节点开始，到子节点，再到子节点的子节点，如此继续，尽可能往深入，直到不能再往下走为止。然后，遍历过程向上回退一层，并进入下一个子节点。例如，在前面展示的 DOM 树中，遍历过程在回退到<body/>前，先访问了<html/>，然后<head/>，然后<title/>，然后是文本节点"example"。图 6-3 显示了这个遍历过程的完整路径。

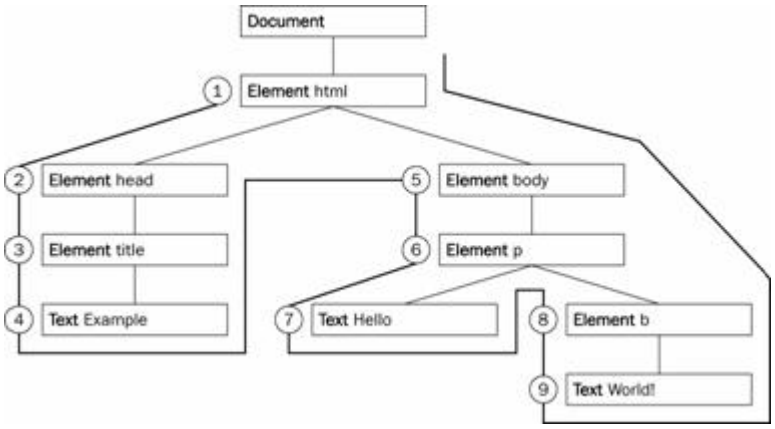

图 6-3

 最佳的思考深度优先搜索的方法就是从第一个节点左边的第一个节点起沿着树的最外沿画线。只要这条线从左侧经过某个节点，那么这个节点就是搜索中下一个出现的节点（图 6-3 中的粗线代表了这条线）。

要创建 NodeIterator 对象，请使用 document 对象的 createNodeIterator()方法。这个方法接受四个参数：

(1) root——从树中开始搜索的那个节点。

(2) whatToShow——一个数值代码，代表哪些节点需要访问。

(3) filter——NodeFilter 对象，用来决定需要忽略哪些节点。

(4) entityReferenceExpansion——布尔值，表示是否需要扩展实体引用。

通过应用下列一个或多个常量，whatToShow 参数可以决定哪些节点可以访问：

- NodeFilter.SHOW_ALL——显示所有的节点类型；
- NodeFilter.SHOW_ELEMENT——显示元素节点；
- NodeFilter.SHOW_ATTRIBUTE——显示特性节点；
- NodeFilter.SHOW_TEXT——显示文本节点；
- NodeFilter.SHOW_CDATA_SECTION——显示 CData section 节点；
- NodeFilter.SHOW_ENTITY_REFERENCE——显示实体引用节点；
- NodeFilter.SHOW_ENTITY——显示实体节点；
- NodeFilter.SHOW_PROCESSING_INSTRUCTION——显示 PI 节点；
- NodeFilter.SHOW_COMMENT——显示注释节点；

- NodeFilter.SHOW_DOCUMENT——显示文档节点；
- NodeFilter.SHOW_DOCUMENT_TYPE——显示文档类型节点；
- NodeFilter.SHOW_DOCUMENT_FRAGMENT——显示文档碎片节点；
- NodeFilter.SHOW_NOTATION——显示记号节点。

可以通过使用二进制或操作符来组合多个值：

```
var iWhatToShow = NodeFilter.SHOW_ELEMENT | NodeFilter.SHOW_TEXT;
```

createNodeIterator()的 filter（过滤器）参数可以指定一个自定义的 NodeFilter 对象，但是如果不想使用它的话，也可以留空（null）。

要创建最简单的访问所有节点类型的 NodeIterator 对象，可以使用下面的代码：

```
var iterator = document.createNodeIterator(document, NodeFilter.SHOW_ALL, null,
false);
```

要在搜索过程中前进或者后退，可以使用 nextNode()和 previousNode()方法：

```
    var node1 = iterator.nextNode();
    var node2 = iterator.nextNode();

var node3 = iterator.previousNode();
alert(node1 == node3); ///outputs "true"
```

例如，假设想列出某个区域内指定 中包含的所有元素。下列代码可以完成这个任务：

```
<html>
    <head>
        <title>NodeIterator Example</title>
        <script type="text/javascript">

           var iterator = null;

           function makeList() {
               var oDiv = document.getElementById("div1");
               iterator = document.createNodeIterator(oDiv,
NodeFilter.SHOW_ELEMENT, null, false);

               var oOutput = document.getElementById("text1");
               var oNode = iterator.nextNode();
               while (oNode) {
                   oOutput.value += oNode.tagName + "\n";
                   oNode = iterator.nextNode();
               }

           }

        </script>
    </head>
    <body>
        <div id="div1">
            <p>Hello <b>World!</b></p>
            <ul>
                <li>List item 1</li>
                <li>List item 2</li>
                <li>List item 3</li>
            </ul>
        </div>
        <textarea rows="10" cols="40" id="text1"></textarea><br />
        <input type="button" value="Make List" onclick="makeList()" />    </body>
</html>
```

当点击了按钮后，将使用包含在 div1 中的元素的标签名来填充<textarea/>：

P

B

UL

LI

LI

LI

183 但假设不想在结果中包含<p/>元素。这就不能仅使用 whatToShow 参数来完成。这种情况下，你需要自定义一个 NodeFilter 对象。

NodeFilter对象只有一个方法：acceptNode()。如果应该访问给定的节点，那么该方法返回NodeFilter.FILTER_ACCEPT；如果不应该访问给定节点，则返回NodeFilter.FILTER_REJECT。然而，不能使用NodeFilter类来创建这个对象，因为这个类

是一个抽象类。在Java或一些其他的语言中，必须重新定义一个NodeFilter的子类，不过，因为这是在JavaScript中，就不必这么做了。

现在，只要创建任意一个有acceptNode()方法的对象，就可以将它传给createNodeIter- ator()方法，如下所示：

```
var oFilter = new Object;
oFilter.acceptNode = function (oNode) {
    //filter logic goes here
};
```

若要禁止<p/>元素节点，只需检查 tagName 属性，如果它等于"P"，就返回NodeFilter. FILTER_REJECT：

```
var oFilter = new Object;
oFilter.acceptNode = function (oNode) {
    return (oNode.tagName == "P") ? NodeFilter.FILTER_REJECT :
NodeFilter.FILTER_ACCEPT;
};
```

如果将这段代码包含在前面的例子中，代码如下：

```
<html>
    <head>
        <title>NodeIterator Example</title>
        <script type="text/javascript">

            var iterator = null;

            function makeList() {
                var oDiv = document.getElementById("div1");
                var oFilter = new Object;
                oFilter.acceptNode = function (oNode) {
                    return (oNode.tagName == "P") ?
                            NodeFilter.FILTER_REJECT : NodeFilter.FILTER_ACCEPT;
                };

                iterator = document.createNodeIterator(oDiv,
NodeFilter.SHOW_ELEMENT, oFilter, false);

                var oOutput = document.getElementById("text1");
                var oNode = iterator.nextNode();
                while (oNode) {
                    oOutput.value += oNode.tagName + "\n";
                    oNode = iterator.nextNode();
                }

            }

        </script>
    </head>
    <body>
        <div id="div1">
            <p>Hello <b>World!</b></p>
            <ul>
                <li>List item 1</li>
                <li>List item 2</li>
                <li>List item 3</li>


            </ul>
        </div>
        <textarea rows="10" cols="40" id="text1"></textarea><br />
        <input type="button" value="Make List" onclick="makeList()" />    </body>
</html>
```

当这次再点击按钮时，<textarea/>中就会出现下列内容：

UL

LI

LI

LI

注意"P"和"B"都没有出现在列表中。这是因为排除<p/>元素后，不仅从迭代搜索中去掉了它，也去掉了它的所有后代节点。因为<b/>是<p/>的一个子节点，所以它也被跳过。

NodeIterator 对象展示了一种有序的自顶向下遍历整个 DOM 树（或者仅仅其中一部分）的方式。然而可能想遍历到树的特定区域时，再看看某个节点的兄弟节点或者子节点。如果是这种情况，可以使用 TreeWalker。

第 6 章 DOM 基础

6.5.2 TreeWalker

TreeWalker 有点像 NodeIterator 的大哥：它有 NodeIterator 所有的功能（nextNode()和 previousNode()），并且添加了一些遍历方法：

- parentNode()——进入当前节点的父节点；
- firstChild()——进入当前节点的第一个子节点；
- lastChild()——进入当前节点的最后一个节点；
- nextSibling()——进入当前节点的下一个兄弟节点；
- previousSibling()——进入当前节点的前一个兄弟节点。

要开始使用 TreeWalker，其实完全可以像使用 NodeIterator 那样，只要把 createNode- Iterator()的调用改为调用 createTreeWalker()，这个函数接受同样的参数：

```
<html>
    <head>
        <title>TreeWalker Example</title>
        <script type="text/javascript">

            var walker = null;

            function makeList() {
                var oDiv = document.getElementById("div1");
                var oFilter = new Object;
                oFilter.acceptNode = function (oNode) {
                    return (oNode.tagName == "P") ?
                            NodeFilter.FILTER_REJECT : NodeFilter.FILTER_ACCEPT;
                };

                walker = document.createTreeWalker(oDiv, NodeFilter.SHOW_ELEMENT,
oFilter, false);
```

```
                var oOutput = document.getElementById("text1");
                var oNode = walker.nextNode();
                while (oNode) {
                    oOutput.value += oNode.tagName + "\n";
                    oNode = walker.nextNode();
                }

            }

        </script>
    </head>
    <body>
        <div id="div1">
            <p>Hello <b>World!</b></p>
            <ul>
                <li>List item 1</li>
                <li>List item 2</li>
                <li>List item 3</li>
            </ul>
        </div>
        <textarea rows="10" cols="40" id="text1"></textarea><br />
        <input type="button" value="Make List" onclick="makeList()" />    </body>
</html>
```

自然，这样一个简单的例子不能展示 TreeWalker 的真正能力。当不想直接翻遍整个 DOM 树时，TreeWalker 十分有用。例如，假设只想访问前面所显示的 HTML 页面中的
元素。可以写一个只接受标签名为"LI"的元素的过滤器，而这里，可以使用 TreeWalker 来进行更有目的性的遍历：

```
<html>
    <head>
        <title>TreeWalker Example</title>
        <script type="text/javascript">

            var walker = null;

            function makeList() {
                var oDiv = document.getElementById("div1");

                walker = document.createTreeWalker(oDiv, NodeFilter.SHOW_ELEMENT,
```

```
null, false);

                var oOutput = document.getElementById("text1");
                walker.firstChild();   //go to <p>
                walker.nextSibling();  //go to <ul>
                var oNode = walker.firstChild();  //go to first <li>
                while (oNode) {
                    oOutput.value += oNode.tagName + "\n";
                    oNode = walker.nextSibling();
                }

            }

        </script>

    </head>
    <body>
        <div id="div1">
            <p>Hello <b>World!</b></p>
            <ul>
                <li>List item 1</li>
                <li>List item 2</li>
                <li>List item 3</li>
            </ul>
        </div>
        <textarea rows="10" cols="40" id="text1"></textarea><br />
        <input type="button" value="Make List" onclick="makeList()" />

        </body>
</html>
```

在这个例子中，创建了 TreeWalker 并立刻调用了 firstChild()方法，将 walker 指到<p/>元素(因为<p/>是 div1 的第一个子节点)。在接下来的一行里调用 nextSibling()后，walker 指到了<p/>的下一个兄弟节点
上。然后，调用 firstChild()返回到
下的第一个
元素。接下来，在循环内部，通过使用 nextSibling()方法对剩下的
元素进行迭代。

点击按钮后，应该会看到如下输出：

LI

LI

LI

最后要说的是，如果对将要遍历的 DOM 树的结构有所了解的话，TreeWalker 更加有用；而同时，如果并不知道这个结构的话，使用 NodeIterator 更加实际有效。

第 6 章 DOM 基础

6.6 测试与 DOM 标准的一致性

现在你肯定可以说出 DOM 的许多方面。正因如此，你需要一种方法来确定给定的 DOM 实现到底支持 DOM 的哪些部分。有趣的是，这个对象就叫做 implementation。

implementation 对象是 DOM 文档的一个特性，因此，也是浏览器 document 对象的一部分。implementation 唯一的方法是 hasFeature()，它接受两个参数：要检查的特征和特征的版本。例如，如果想检查对 XML DOM Level 1 的支持，可以这样调用：

```
var bXmlLevel1 = document.implementation.hasFeature("XML", "1.0");
```

下面的表格列出了所有的 DOM 特征以及相应需要检查的版本：

特　征	支持的版本	描　述
Core	1.0, 2.0, 3.0	基本的 DOM，给予了用层次树来表示文档的能力
XML	1.0，2.0，3.0	核心的 XML 扩展，增加了对 CDATA Section、处理指令和实体的支持
HTML	1.0，2.0	XML 的 HTML 扩展，增加了对 HTML 特定元素和实体的支持
Views	2.0	基于特定样式完成对文档的格式化
StyleSheets	2.0	为文档关联样式表
CSS	2.0	支持级联样式表 1（CSS Level 1）

（续）

特　征	支持的版本	描　述
CSS2	2.0	支持级联样式表 2（CSS Level 2）
Events	2.0	通用 DOM 事件
UIEvents	2.0	用户界面事件
MouseEvents	2.0	由鼠标引起的事件（点击、鼠标经过，等等）
MutationEvents	2.0	当 DOM 树发生改变时引发的事件
HTMLEvents	2.0	HTML 4.01 的事件

Range	2.0	操作 DOM 树中某个特定范围的对象和方法
Traversal	2.0	遍历 DOM 树的方法
LS	3.0	在文件和 DOM 树之间同步地载入和存储
LS-Async	3.0	在文件和 DOM 树之间异步地载入和存储
Validation	3.0	用于修改 DOM 树之后仍然保持其有效性的方法

尽管这个相当方便，但是，使用 implementation.hasFeature()有其明显的缺陷——决定 DOM 实现是否对 DOM 标准的不同的部分相一致的，正是去进行实现的人（或公司）。要让这个方法对于任何值都返回 true，那是很简单的，但这并不一定表示这个 DOM 实现真的和所有的标准都一致了。目前为止，最精确的浏览器要数 Mozilla，但它多少也有一些并不完全和 DOM 标准一致的地方，这个方法却返回为 true。

第 6 章 DOM 基础

6.7 DOM Level 3

2004 年 4 月，DOM Level 3 作为 W3C 的一个推荐标准被提出。目前为止，还没有哪个浏览器已经完全实现该标准，只有 Mozilla 已实现了一部分。还不知道 Web 浏览器要用多久才能把剩下的 DOM 特性加上去，因为 IE 近四年都没有重大的更新了（这里的意思是它对 DOM 的支持程度没有什么变化）。Mozilla 许诺将尽可能与标准保持兼容，它仍然继续在对 DOM 支持方面遥遥领先。然而，Opera 重写了它的核心浏览器组件来更好地支持 DOM 标准，并有一种和最新科技齐头并进的势头。甚至 Apple 的 Safari 浏览器（基于 Konqueror 的）也在不断跟进，有计划地尽可能多地实现 DOM 功能。

第 6 章 DOM 基础

6.8 小结

本章介绍了文档对象模型（DOM）的基本接口。了解到 DOM 如何将基于 XML 的文档组织成由任意节点组成的层次树。还学习了在文档中出现的不同的节点类型以及如何处理、添加、删除 DOM 树中的节点。

另外，这一章还讲述了针对 HTML DOM 的特征功能，如将一些特性转为对象属性和一些针对表格的方法，这些功能让编写 HTML 与传统的 DOM 方式相比更加简单。


最后，还学习了 DOM 遍历规范中的 NodeIterator 和 TreeWalker 对象，通过它们按照合理的方式遍历 DOM 树。

唯一没有在本章中介绍的DOM的主要部分是事件和事件处理，这些将在第 9 章中详细介绍。