

编写可扩展、可重用、高质量的JavaScript应用程序和库

JavaScript 面向对象编程指南

Object-Oriented JavaScript

[加] Stoyan Stefanov 著
凌杰 译

JavaScript 面向对象编程指南

如今，招聘Web开发者的职位要求中，具备JavaScript的知识已经是决定性的因素了。过去，我们只是偶尔在网页中简单地嵌入几行JavaScript代码，而现在已经拥有了各种程序库及扩展性应用构架，以用于各种“胖客户端”以及AJAX类型的Web应用程序。

本书着重于介绍JavaScript在面向对象方面的特性，以展示如何构建健壮的、可维护的、功能强大的应用程序及程序库。阅读完本书，读者将会在面试有关JavaScript的职位时游刃有余，甚至会凭借某些连面试官都不知道的知识给他们留下深刻的印象。

如果想让自己的JavaScript开发能力提升到一个新的水平，这本书将是不错的选择。

本书的目标读者

本书并不要求读者在JavaScript方面有任何基础，我们将会从零开始打下深厚的JavaScript语言功底。即便读者对JavaScript代码已经有了一定程度的了解，恐怕也会对该语言无所不能的魅力感到惊讶。

本书鼓励读者亲自动手编写代码，因为学习一种编程语言最好的办法就是编写代码。本书鼓励读者在Firebug控制台中输入代码，看看这些代码是如何工作的，应该如何调整和把玩它们。在每一章的末尾，都有一定量的练习题，以帮助读者检查自己学到的内容。

本书内容

- JavaScript作为一门浏览器语言的核心思想；
- 面向对象编程的基础知识及其在JavaScript中的运用；
- 数据类型、操作符以及流程控制语句；
- 函数、闭包、对象和原型等概念，以代码重用为目的的继承模式；
- BOM、DOM、浏览器事件、AJAX和JSON；
- 如何实现JavaScript中缺失的面向对象特性，如对象的私有成员与私有方法；
- 如何应用适当的编程模式，发挥JavaScript语言特有的优势；
- 如何应用设计模式解决常见问题。

作者简介

Stoyan Stefanov是雅虎公司的Web开发人员和Zend认证的工程师。他是雅虎性能开发工具“YSlow”项目的领导人，并参与了“PEAR”库等其他开源项目。他也是*JavaScript Patterns*一书的作者。



人民邮电出版社·信息技术分社
<http://weibo.com/ptpitbooks>

美术编辑：王建国

分类建议：计算机/程序设计/JavaScript
人民邮电出版社网址：www.ptpress.com.cn

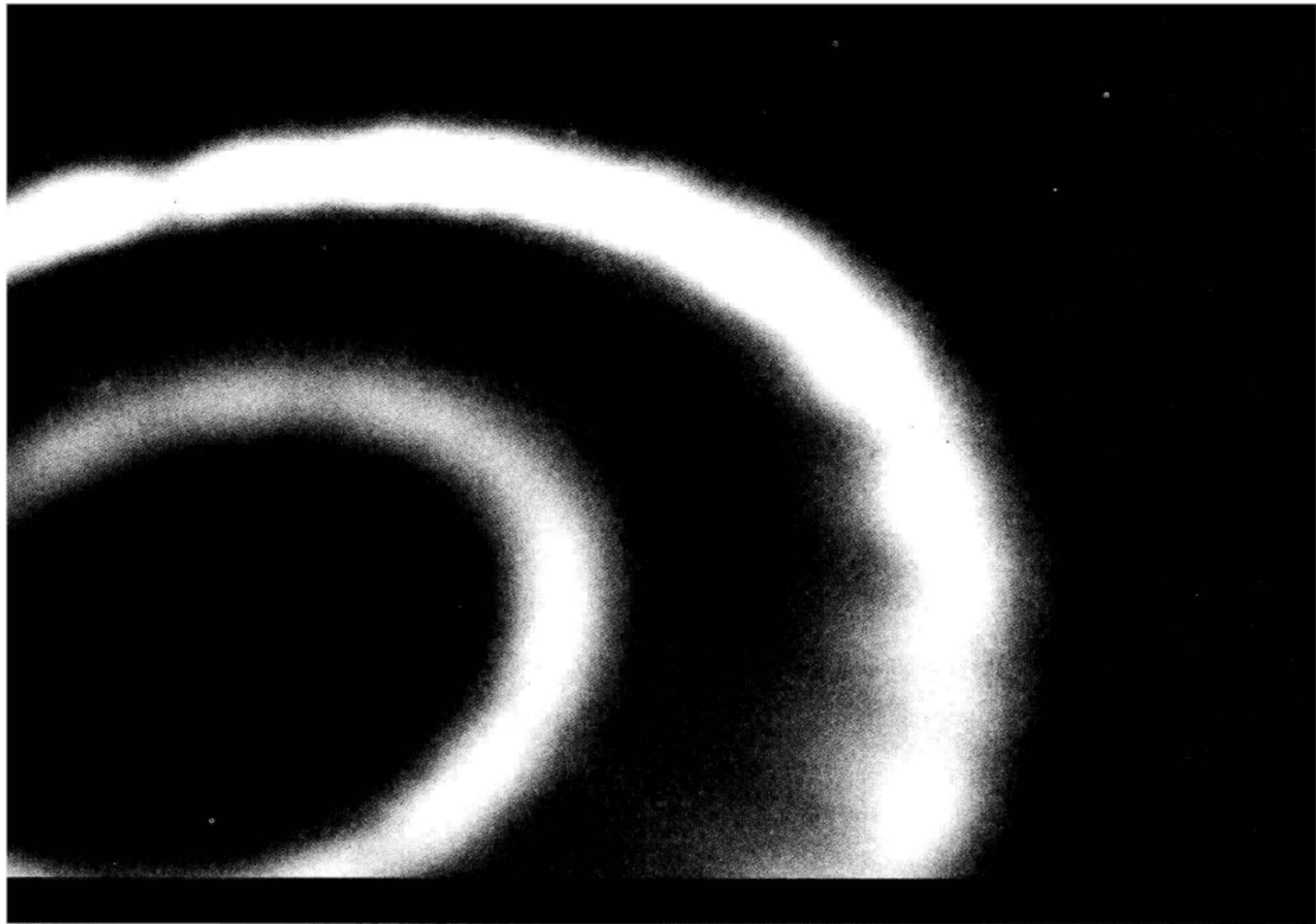
ISBN 978-7-115-30904-4



9 787115 309044 >

ISBN 978-7-115-30904-4

定价：59.00 元



JavaScript 面向对象编程指南

[加] Stoyan Stefanov 著
凌杰 译

人民邮电出版社
北京

图书在版编目 (C I P) 数据

JavaScript面向对象编程指南 / (加) 斯托扬
(Stefanov, S.) 著 ; 凌杰译. — 北京 : 人民邮电出版
社, 2013. 3
ISBN 978-7-115-30904-4

I. ①J… II. ①斯… ②凌… III. ①
JAVA语言—程序设计—指南 IV. ①TP312-62

中国版本图书馆CIP数据核字(2013)第016139号

版权声明

Copyright © 2008 Packt Publishing. First published in the English language under the title *Object-Oriented JavaScript*.

All rights reserved.

本书由美国 Packt Publishing 公司授权人民邮电出版社出版。未经出版者书面许可, 对本书的任何部分不得以任何方式或任何手段复制和传播。

版权所有, 侵权必究。

JavaScript 面向对象编程指南

- ◆ 著 [加]Stoyan Stefanov
- 译 凌 杰
- 责任编辑 陈冀康
- ◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街 14 号
邮编 100061 电子邮件 315@ptpress.com.cn
网址 <http://www.ptpress.com.cn>
三河市海波印务有限公司印刷
- ◆ 开本: 800×1000 1/16
印张: 20.75
字数: 404 千字 2013 年 3 月第 1 版
印数: 1-3 000 册 2013 年 3 月河北第 1 次印刷

著作权合同登记号 图字: 01-2012-4600 号

ISBN 978-7-115-30904-4

定价: 59.00 元

读者服务热线: (010)67132692 印装质量热线: (010)67129223
反盗版热线: (010)67171154

目录

第 1 章 引言	1
1.1 回顾历史	1
1.2 变革之风	3
1.3 分析现状	3
1.4 展望未来	4
1.5 面向对象程序设计	5
1.5.1 对象	5
1.5.2 类	6
1.5.3 封装	6
1.5.4 聚合	7
1.5.5 继承	7
1.5.6 多态	8
1.6 OOP 概述	8
1.7 训练环境设置	9
1.8 使用 Firebug 控制台	10
1.9 本章小结	11
第 2 章 基本数据类型、数组、循环及条件表达式	13
2.1 变量	13
2.2 操作符	15
2.3 基本数据类型	18
2.3.1 查看类型操作符——typeof	19
2.3.2 数字	19

2.3.3	字符串	23
2.3.4	布尔值	26
2.3.5	Undefined 与 null	32
2.4	基本数据类型综述	34
2.5	数组	35
2.5.1	增加、更新数组元素	36
2.5.2	删除元素	36
2.5.3	数组的数组	37
2.6	条件与循环	38
2.6.1	代码块	38
2.6.2	循环	44
2.7	注释	49
2.8	本章小结	49
2.9	练习题	50
第 3 章	函数	52
3.1	什么是函数	53
3.1.1	调用函数	53
3.1.2	参数	53
3.2	预定义函数	55
3.2.1	parseInt()	56
3.2.2	parseFloat()	57
3.2.3	isNaN()	58
3.2.4	isFinite()	58
3.2.5	URI 的编码与反编码	59
3.2.6	eval()	59
3.2.7	一点惊喜——alert()函数	60
3.3	变量的作用域	60
3.4	函数也是数据	62
3.4.1	匿名函数	63
3.4.2	回调函数	64
3.4.3	回调示例	65
3.4.4	自调函数	66
3.4.5	内部（私有）函数	67
3.4.6	返回函数的函数	68

3.4.7 能重写自己的函数	68
3.5 闭包	70
3.5.1 作用域链	70
3.5.2 词法作用域	71
3.5.3 利用闭包突破作用域链	72
3.5.4 Getter 与 Setter	78
3.5.5 迭代器	79
3.6 本章小结	80
3.7 练习题	80
第 4 章 对象	82
4.1 从数组到对象	82
4.1.1 元素、属性、方法	84
4.1.2 哈希表、关联型数组	85
4.1.3 访问对象的属性	85
4.1.4 调用对象的方法	86
4.1.5 修改属性与方法	87
4.1.6 使用 this 值	88
4.1.7 构造器函数	88
4.1.8 全局对象	90
4.1.9 构造器属性	91
4.1.10 instanceof 操作符	92
4.1.11 返回对象的函数	92
4.1.12 传递对象	93
4.1.13 对象比较	94
4.1.14 Firebug 控制台中的对象	95
4.2 内建对象	96
4.2.1 Object	97
4.2.2 Array	98
4.2.3 Function	102
4.2.4 Boolean	107
4.2.5 Number	109
4.2.6 String	110
4.2.7 Math	115
4.2.8 Date	117
4.2.9 RegExp	121

4.2.10 Error 对象	127
4.3 本章小结	131
4.4 练习题	132
第 5 章 原型	135
5.1 原型属性	135
5.1.1 利用原型添加方法与属性	136
5.1.2 使用原型的方法与属性	137
5.1.3 自身属性与原型属性	138
5.1.4 利用自身属性重写原型属性	139
5.1.5 isPrototypeOf()方法	143
5.1.6 神秘的_proto_链接	143
5.2 扩展内建对象	145
5.2.1 关于扩展内建对象的讨论	146
5.2.2 一些原型陷阱	147
5.3 本章小结	150
5.4 练习题	150
第 6 章 继承	152
6.1 原型链	152
6.1.1 原型链示例	153
6.1.2 将共享属性迁移到原型中去	156
6.2 只继承于原型	158
6.3 uber——子对象访问父对象的方式	161
6.4 将继承部分封装成函数	163
6.5 属性拷贝	163
6.6 小心处理引用拷贝	165
6.7 对象之间的继承	167
6.8 深拷贝	169
6.9 object()	171
6.10 原型继承与属性拷贝的混合应用	172
6.11 多重继承	173
6.12 寄生式继承	175
6.13 构造器借用	176

6.14	本章小结	179
6.15	案例学习：图形绘制	183
6.15.1	分析	183
6.15.2	实现	184
6.15.3	测试	188
6.16	练习题	189
第 7 章	浏览器环境	190
7.1	在 HTML 页面中引入 JavaScript 代码	190
7.2	概述：BOM 与 DOM	191
7.3	BOM	192
7.3.1	window 对象再探	192
7.3.2	window.navigator	193
7.3.3	Firebug 的备忘功能	193
7.3.4	window.location	194
7.3.5	window.history	195
7.3.6	window.frames	196
7.3.7	window.screen	197
7.3.8	window.open()/close()	198
7.3.9	window.moveTo()、window.resizeTo()	199
7.3.10	window.alert()、window.prompt()、window.confirm()	199
7.3.11	window.setTimeout()、window.setInterval()	201
7.3.12	window.document	202
7.4	DOM	202
7.4.1	Core DOM 与 HTML DOM	204
7.4.2	DOM 节点的访问	206
7.4.3	DOM 节点的修改	215
7.4.4	新建节点	218
7.4.5	移除节点	221
7.4.6	只适用于 HTML 的 DOM 对象	223
7.5	事件	227
7.5.1	内联 HTML 属性法	227
7.5.2	元素属性法	227
7.5.3	DOM 的事件监听器	228
7.5.4	捕捉法与冒泡法	229

7.5.5	阻断传播	231
7.5.6	防止默认行为	233
7.5.7	跨浏览器事件监听器	233
7.5.8	事件类型	235
7.6	XMLHttpRequest 对象	236
7.6.1	发送请求	236
7.6.2	处理响应	237
7.6.3	在早于 7 的 IE 版本中创建 XMLHttpRequest 对象	238
7.6.4	A 代表异步	239
7.6.5	X 代表 XML	240
7.6.6	实例示范	240
7.7	本章小结	242
7.8	练习题	244
第 8 章	编程模式与设计模式	247
8.1	编程模式	248
8.1.1	行为隔离	248
8.1.2	命名空间	250
8.1.3	初始化分支	253
8.1.4	延迟定义	254
8.1.5	配置对象	255
8.1.6	私有属性和方法	257
8.1.7	特权函数	258
8.1.8	私有函数的公有化	258
8.1.9	自执行函数	259
8.1.10	链式调用	260
8.1.11	JSON	261
8.2	设计模式	262
8.2.1	单件模式 1	263
8.2.2	单件模式 2	263
8.2.3	工厂模式	264
8.2.4	装饰器模式	266
8.2.5	观察者模式	269
8.3	本章小结	272

附录 A 保留字.....	273
附录 B 内建函数.....	276
附录 C 内建对象.....	279
附录 D 正则表达式.....	305

第 1 章

引言

众所周知，时下所流行的这些 Web 应用，例如 Yahoo! Maps、Google Maps、Yahoo! Mail、My Yahoo!、Gmail、Digg 以及 YouTube 等都是一些明显的共同特征，即：它们都是 Web2.0 时代的应用程序，都有非常丰富的人性化交互界面，而这往往意味着大量的 JavaScript 应用。事实上，JavaScript 最初也只不过是一种内嵌于 HTML 语句中的单行式脚本语言。但如今已经今非昔比了，对于它今天所拥有的面向对象特性来说，无论是在可重用性方面，还是在可扩展性方面都已经足以支持我们去实现任何网站项目中的行为逻辑了。毕竟，对于今天的标准来说，任何一个符合规范的 Web 页面都应该包含以下三个要素：内容（HTML）、外观（CSS）和行为（JavaScript）。

通常来说，JavaScript 程序的运行必须要依赖于某个宿主环境。其中最常见的当然就是我们的 Web 浏览器了，但请注意，浏览器并不是 JavaScript 代码唯一的宿主环境。事实上，我们可以利用 JavaScript 来创建各种类型的插件工具、应用扩展以及其他形式的组件。总之，学习 JavaScript 语言是一件一举多得的事情，我们可以通过学习这种语言，来编写各种不同的应用程序。

这本书将着重于介绍 JavaScript 语言本身，特别是其中的面向对象特性。我们会从零开始讲解这些内容，也就是说，读这本书无需具备任何的程序设计基础。另外，除了有一章内容是专门为 Web 浏览器环境而写的以外，本书其余部分介绍的都是 JavaScript 的一般特性，适用于任何支持该语言的执行环境。

现在，让我们进入第 1 章的学习吧。首先，我们需要先来了解一下 JavaScript 背后的发展历程，而后我们才能逐步引入面向对象编程方面的基本概念。

1.1 回顾历史

起初，Web 站点事实上只不过是一个静态的 HTML 文档集，这些文档之间仅依靠一些

简单的超链接 (hyperlinks) 绑定在一起。但很快, 随着 Web 业务的快速普及和增长, 网站管理者越来越希望自己所创建的 Web 页面能处理更多的事情。例如, 他们希望网站具有更丰富的用户交互能力, 以便能完成一些简单的任务 (比如验证表单之类), 加强与服务器端的信息交互。那时候, 他们有两种选择: Java applets (后来被证明失败了) 和 LiveScript。其中, LiveScript 就是 1995 年由 Netscape 公司开发的程序设计语言。Netscape 2.0 之后, 它正式被更名为 JavaScript。

不久, 这种对 Web 页面中静态元素进行扩展的方式就在业界大放异彩, 令其他的浏览器厂商也纷纷效仿, 推出了自己的类似产品。例如, Microsoft 公司随后就发布了支持 JScript 的 Internet Explorer (IE) 3.0。该语言在 JavaScript 的基础上引入了一些 IE 独有的特性。最终, 为了使语言的实现更趋向于标准化, 一个叫做 ECMAScript (欧洲计算机制造商协会) 的组织应运而生了。这才有了我们今天所看到的这份被叫做 ECMA-262 的标准文档。目前在业界广为流行的 JavaScript 也只是遵守该标准的一种实现而已。

无论结果是好是坏, JavaScript 在随后爆发的第一次浏览器大战 (大约是在 1996 年到 2001 年间) 中得到了迅速的普及。那时正值互联网发展的第一波繁荣期, 主要由 Netscape 和 Microsoft 这两大浏览器厂商在争夺市场份额。在此过程中, 他们不断地往各自的浏览器中添加新的特性和各种版本的 JavaScript 实现。但他们彼此之间又缺乏共同遵守的标准, 这给 JavaScript 的开发带来大量的负面影响, 也给开发人员带来巨大的痛苦。因为在这种情况下, 我们通常只能针对某一个具体的浏览器来编写脚本。如果我们把在这个浏览器上开发的脚本拿到其他浏览器上测试, 就会发现它们完全不能工作。与此同时, 由于浏览器厂商都在忙于继续增加新的浏览器特性, 以至于根本没能及时更新相应的工具, 这导致了开发工具的严重滞后。

尽管浏览器厂商引入的不兼容性使 Web 开发人员感到难以忍受, 但这还只是问题的一个方面。而另一方面的问题则出在开发人员自己身上, 他们在自己的 Web 页面中使用了太多的新特性, 总是迫不及待地想引入浏览器提供的每一项新功能, 以“加强”自己的页面。例如状态栏中的动画、闪烁的颜色、闪烁的文本、会摇晃的浏览器窗口、屏幕上的雪花效果、能跟踪对象的鼠标光标等, 而这往往都是以牺牲实用性为代价的。这种滥用现象极大地损坏了 JavaScript 在业界的名声, 以至于那些“真正的程序员” (这里特指那些具有更成熟的编程语言背景的开发人员, 例如 Java 或 C/C++ 程序员) 对 JavaScript 根本不屑一顾, 或者仅仅把它当做一种用于前端设计的玩具。

出于上述原因, JavaScript 语言在一些 Web 项目中遭到了强烈抵制。某些项目甚至完全拒绝在客户端上进行任何的程序设计, 转而只信任他们自己可以掌控的服务器端。确实, 在当时的情况下, 也没有什么理由值得我们花费双倍的时间来为这些不同的浏览器设计项

目，然后再花更多的时间去调试它们。

1.2 变革之风

这种情况一直持续到第一次浏览器大战结束。但在随后的几年中，Web 开发领域在一系列历史进程的推动下，终于发生了一些非常积极的变化。

- ◆ Microsoft 公司赢得了战争，但在之后的五年中（这或多或少算得上一个互联网时代了），他们停止了继续向 Internet Explorer 和 JScript 中添加新特性的动作，这给了其他浏览器充分的时间，使它们能够在功能上逐步完成对 IE 的追赶和超越。
- ◆ Web 标准在移动开发领域的重要性在开发人员和浏览器厂商那里得到一致的认可。这是很自然的，毕竟对于开发人员来说，谁也不想因为不同的浏览器而花费双倍（甚至更多）的开发时间，这促使各方都越来越倾向于遵守统一的开发标准。尽管目前，我们离建立一个完全统一的标准化环境还有很长的路要走，但目标已经很明确了，相信终会有实现的那一天的。
- ◆ 开发人员和技术本身也日趋成熟了，更多的人开始将注意力转移到东西本身的可用性上，并以此为基础，逐步加强在技术和功能方面的开发力度。

在这种健康环境的影响下，开发人员开始谋求一种更好的新型开发模式，以取代这些现有的开发方式。而随着类似 Gmail 和 Google Maps 这样的应用程序的相继出现，客户端的程序设计也开始逐渐变得丰富起来。显然，如今的 JavaScript 已经成为一种成熟的、在某些方面独一无二的、具有强大原型体系的面向对象语言。关于这点，最好的例子莫过于是对 XMLHttpRequest 对象的重新发现和推广，该对象起初不过是一个 IE-only 特性，但如今已经得到绝大多数浏览器的支持。通过 XMLHttpRequest 对象，JavaScript 可以用 HTTP 请求的形式从服务器上获得所需的新鲜内容，实现了页面的局部更新。这样一来，我们就不必每次都刷新整个页面。随着 XMLHttpRequest 对象的广泛应用，一种类桌面式的 Web 应用程序模式诞生了，我们称之为 AJAX 的应用程序。

1.3 分析现状

关于 JavaScript 语言，最有意思的是它必须要在一个宿主环境中运行。其中受欢迎的宿主环境当然就是浏览器了，但这并不是我们唯一的选择。JavaScript 完全可以运行在服务器端、桌面以及富媒体环境中。如今，我们可以使用 JavaScript 来实现以下功能：

- ◆ 创建具有强大而丰富的 Web 应用程序（这种应用程序往往运行于 Web 浏览器中，例如 Gmail）。
- ◆ 编写类似 ASP 这样的服务器端脚本，或者使用 Rhino（这是一种用 Java 实现的 JavaScript 引擎）这样的框架进行编程。
- ◆ 创建某些富媒体式的应用程序（如 Flash、Flex），这其中用到的 ActionScript 就是一种基于 ECMAScript 标准的脚本语言。
- ◆ 编写 Windows 桌面自动化管理脚本任务，我们可以使用 Windows 自带的脚本宿主环境。
- ◆ 为一些桌面应用程序编写扩展或插件，例如 Firefox、Dreamweaver、Fiddler。
- ◆ 创建一些桌面型 Web 应用程序，这些应用程序往往会使用离线型数据库来存储信息，例如 Google Gears。
- ◆ 创建 Yahoo! Widgets、Mac Dashboard 这样的小工具或某些桌面型 Adobe Air 应用程序。

当然，这里列出的也远远不是该语言应用的全部。JavaScript 应用的确发端于 Web 页面，但如今，几乎可以说它们已经无所不在了。

1.4 展望未来

对于未来的情况，我们在这里只能做一些猜测。但几乎可以肯定地说，JavaScript 语言必将还会有它的一席之地。毕竟，在过去相当长的一段时间里，JavaScript 在被严重低估、始终未得到充分利用（或者被错误地滥用了）的情况下，依然几乎每天都能有很多新的、有趣的 JavaScript 应用被开发出来。尽管它们都是一些简单的、内嵌于 HTML 标签中（例如 onclick 事件）的单行式代码。而如今的开发人员所面对的商业开发往往要复杂得多，这需要良好的设计和规划，以及合适的应用扩展和程序库。JavaScript 必将在其中得到真正的用武之地，开发人员无疑会更加重视它独有的面向对象特性，以获取越来越多的便利。

一旦 JavaScript 成为未来招聘中的“必需项”，您在这方面的知识储备就会成为能否成功应聘某些 Web 开发职位的决定性因素。例如，我们在面试时常会被问到这样的问题：“JavaScript 是一种面向对象语言吗？如果是，您在 JavaScript 中是如何实现继承的呢？”读了这本书之后，您就会对这样的面试有充分的准备，并有可能凭借一些连面试官自己都不知道的知识而打动他们。

1.5 面向对象的程序设计

在我们深入学习 JavaScript 之前，首先要了解一下“面向对象”的具体含义，以及这种程序设计风格的主要特征。下面我们列出了一系列在面向对象程序设计（OOP）中最常用到的概念：

- ◆ 对象、方法、属性
- ◆ 类
- ◆ 封装
- ◆ 聚合
- ◆ 重用与继承
- ◆ 多态

现在，让我们来进行逐一阐述。

1.5.1 对象

既然这种程序设计风格叫做面向对象，那么它的重点就在于对象。而所谓的对象，实质上是指“事物”（包括人和物）在程序设计语言中的表现形式。这里的“事物”可以是任何东西（如某个客观存在的对象，或者某些较为抽象的概念）。例如，对于猫这种常见对象来说，我们可以看到它们具有某些明确的特征（如颜色、名字、体型等），能执行某些动作（如喵喵叫、睡觉、躲起来、逃跑等）。在 OOP 语义中，这些对象特征就叫做属性，而那些动作就称之为方法。

此外，我们还有一个口语方面的类比^①：

- ◆ 对象往往是用名词来表示的（如 book、person）
- ◆ 方法一般都是些动词（如 read、run）
- ◆ 属性值则往往是一些形容词

我们可以来试一下。例如，在“The black cat sleeps on my head”这个句子中，“the cat”（名词）就是一个对象，“black”（形容词）则是一个颜色属性值，而“sleep”（动词）则代

^① 这里应该特指英文环境，在中文这种形而上的语言环境中，这种类比或许并不是太合适。——译者注

表一个动作，也就是 OOP 语义中的方法。甚至，为了进一步证明这种类比的合理性，我们也可以将句子中的“on my head”看做动作“sleep”的一个限定条件，因此，它也可以被当做传递给 sleep 方法的一个参数。

1.5.2 类

在现实生活中，相似的对象之间往往都有一些共同的组成特征。例如蜂鸟和老鹰都具有鸟类的特征，因此它们可以被统称为鸟类。在 OOP 中，类实际上就是对象的设计蓝图或者制作配方。“对象”这个词，我们有时候也叫做“实例”，所以我们可以说老鹰是鸟类的实例^①。我们能基于相同的类创建出许多不同的对象。因为类更多的是一种模板，而对象就是在这些模板的基础上被创建出来的。

但是我们要明白，JavaScript 与 C++或 Java 这种传统的面向对象语言不同，它实际上压根儿没有类。该语言的一切都是基于对象的，其所依靠的是一套原型系统（这里的原型（prototype）实际上也是一种对象，我们稍后再来详细讨论这个问题）。在传统的面向对象语言中，我们一般会这样描述自己的做法：“我基于 Person 类创建了一个叫做 Bob 的新对象。”，而在这种基于原型的面向对象语言中，我们则会这样描述：“我将现有的 Person 对象扩展成了一个叫做 Bob 的新对象。”

1.5.3 封装

封装则是另一个 OOP 相关的概念，它主要用于阐述对象中所包含（或封装）的内容，它们通常由两部分组成：

- ◆ 相关的数据（用于存储属性）。
- ◆ 基于这些数据所能做的事（所能调用的方法）。

但除此之外，封装这个术语中还包含了一层隐藏信息的概念，这完全是另一方面的问题。因此，我们在理解这个概念时，必须要留意它在具体的 OOP 语境中的含义。

以一个 MP3 播放器为例。如果假设这是一个对象，那么作为该对象的用户，我们无疑需要一些类似于像按钮、显示屏这样的工作接口。这些接口能帮助我们使用该对象（如播放歌曲之类）。至于它们内部是如何工作的，我们并不清楚，而且多数情况下也不会在乎这些。换句话说，这些接口的实现对我们来说是不可见的。同样的，在 OOP 中也是如此。当我们在代码中调用一个对象的方法时，无论该对象是来自我们自己的实现还是某个第三方

^① 至少在中文环境中，老鹰更像是鸟类的一个子类。希望读者在理解对象与类的关系时，不要过分依赖这种类比。——译者注

库，我们都不需要知道该方法是如何工作的。在编译型语言中，我们甚至都无法查看这些对象的工作代码。而由于 JavaScript 是一种解释型语言，源代码是可以查看的。但至少在这个概念上它们是一致的，即我们只需要知道所操作对象的接口，而不必去关心它的具体实现。

关于信息隐藏，还有另一方面内容，即方法与属性的可见性。在某些语言中，我们可以通过 `public`、`private`、`protected` 这些关键字来限定方法和属性的可见性。这种限定分类定义了对象用户所能访问的层次。例如，`private` 方法只有其所在对象内部的代码才有权访问，而 `public` 方法则是任何人都能访问的。在 JavaScript 中，尽管所有的方法和属性都是 `public` 的，但是我们将会看到，该语言还是提供了一些隐藏数据的方法，以保护程序的隐密性。

1.5.4 聚合

所谓聚合，有时候也叫做组合，实际上是指我们将几个现有对象合并成一个新对象的过程。总之，这个概念所强调的就是这种将多个对象合而为一的能力。通过聚合这种强有力的方法，我们可以将一个问题分解成多个更小的问题。这样一来，问题就会显得更易于管理（便于我们各个击破）。当一个问题域的复杂程度令我们难以接受时，我们就可以考虑将它分解成若干子问题区，并且必要的话，这些问题区还可以再继续分解成更小的分区。这样做有利于我们从几个不同的抽象层次来考虑这个问题。例如，个人电脑是一个非常复杂的对象，我们不可能知道它启动时所发生的全部事情。但如果我们将这个问题的抽象级别降低到一定的程度，只关注它几个组件对象的初始化工作，例如监视器对象、鼠标对象、键盘对象等，我们就很容易深入了解这些子对象情况，然后再将这些部分的结果合并起来，之前那个复杂问题就迎刃而解了。

我们还可以找到其他类似情况，例如 `Book` 是由一个或多个 `author` 对象、`publisher` 对象、若干 `chapter` 对象以及一组 `table` 对象等合并（聚合）而成的对象。

1.5.5 继承

通过继承这种方式，我们可以非常优雅地实现对现有代码的重用。例如，我们有一个叫做 `Person` 的一般性对象，其中包含一些姓名、出生日期之类的属性，以及一些功能性函数，如步行、谈话、睡觉、吃饭等。然后，当我们发现自己需要一个 `Programmer` 对象时，当然，这时候你可以再将 `Person` 对象中所有的方法与属性重新实现一遍，但除此之外还有一种更聪明的做法，即我们可以让 `Programmer` 继承自 `Person`，这样就省去了我们不少工作。因为 `Programmer` 对象只需要实现属于它自己的那部分特殊功能（例如“编写代码”），而其余部分只需重用 `Person` 的实现即可。

在传统的 OOP 环境中，继承通常指的是类与类之间的关系，但由于 JavaScript 中不存

在类，因此继承只能发生在对象之间。

当一个对象继承自另一个对象时，通常会往其中加入新的方法，以扩展被继承的老对象。我们通常将这一过程称之为“B 继承自 A”或者“B 扩展自 A”。另外对于新对象来说，它也可以根据自己的需要，从继承而来那组方法中选择几个来重新定义。这样做并不会改变对象的接口，因为方法的名字是相同的，只不过当我们调用新对象时，该方法的行为与之前不同了。我们将这种重定义继承方法的过程叫做覆写。

1.5.6 多态

在之前的例子中，我们的 Programmer 对象继承了上一级对象 Person 的所有方法。这意味着这两个对象都实现了“talk”等方法。现在，我们的代码中有一个叫做 Bob 的变量，即便是在我们不知道它是一个 Person 对象还是一个 Programmer 对象情况下，也依然可以直接调用该对象的“talk”方法，而不必担心这会影响代码的正常工作。类似这种不同对象通过相同的方法调用来实现各自行为的能力，我们就称之为多态。

1.6 OOP 概述

如果您在面向对象程序设计方面是一个新手，或者您不能确定自己是否真的理解了上面这些概念，请不必太担心。以后我们还会通过一些代码来为您具体分析它们。尽管这些概念说起来好像很复杂、很高级，但一旦进入真正的实践，事情往往就要简单得多。

话虽如此，但还是先让我们再来复习一下这些概念吧（见表 1-1）。

表 1-1

特征描述	相应概念
Bob 是一个男人（后者是一个对象）	对象
Bob 出生于 1980 年 6 月 1 日，男性，黑头发	属性
Bob 能吃饭、睡觉、吃饭、喝水、做梦，以及记录自己的年龄	方法
Bob 是 Programmer 类的一个实例	传统 OOP 中的类
Bob 是一个由 Programmer 对象扩展而来的新对象	基于原型 OOP 中的原型对象
Bob 对象中包含了数据（例如出生日期）和基于这些数据的方法（例如记录年龄）	封装

(续表)

特征描述	相应概念
我们并不需要知道其记录年龄的方法是如何实现的。对象通常都可以拥有一些私有数据，例如对闰年二月的天数，我们就不知道，而且也不会想知道	信息隐藏
Bob 只是是整个 Web 开发团队对象的一部分，此外还包含了一个 Designer 对象 Jill，已经一个 ProjectManager 对象 Jack	聚合、组合
Designer、ProjectManager、Programmer 都是分别扩展自 Person 对象的新对象	继承
我们可以随时调用 Bob、Jill 和 Jack 这三个对象各自的 talk 方法，它们都可以正常工作，尽管这些方法会产生不同的结果（如 Bob 可能谈的更多的是代码的性能，Jill 则更倾向于谈代码的优雅性，而 Jack 强调的是最后期限）。总之，每个对象都可以重新自定义它们的继承方法 talk	多态、方法覆写

1.7 训练环境设置

在这本书中，我们在代码编写方面更强调的是“自己动手”，因为在作者的理念中，学好一门编程语言最好的途径就是编写代码。因此，这里没有任何的代码可以供您下载，以便直接复制/粘贴。正好相反，您必须亲自输入代码，并观察它们是如何工作的，需要做哪些调整，然后周而复始地摆弄它们。因而，当您在尝试这些代码示例时，我们建议您使用 Firebug 控制台这样的工具。下面就让我们来看看这些工具是如何使用的。

获取所需的工具

作为开发人员来说，他们的机器上大多都应该已经安装了 Firefox，并将它作为日常浏览器来使用。如果您还没有安装该 Web 浏览器，那么现在可以安装一个了。它完全免费，适用于任何平台——Windows、Linux 或者 Mac 都可以，您可以到 <http://www.mozilla.com/firefox/> 中去下载。

Firefox 是一款扩展性很强的浏览器，如图 1-1 所示，它拥有大量可用的扩展插件（几乎全是用 JavaScript 实现的）。Firebug 就是其中比较受欢迎的一款扩展组件——事实上，该组件也是 Web 开发中不可或缺的工具，它提供了大量的、可用性很强的特性。您可以在 <http://www.getfirebug.com/> 中下载它，安装完成之后，我们可以在任何网页下按

F12（Windows 下）或者单击 Firefox 窗口右上角的小虫子按钮来启动 Firebug 界面。而这个界面就是我们接下来的重点——控制台。

1.8 使用 Firebug 控制台

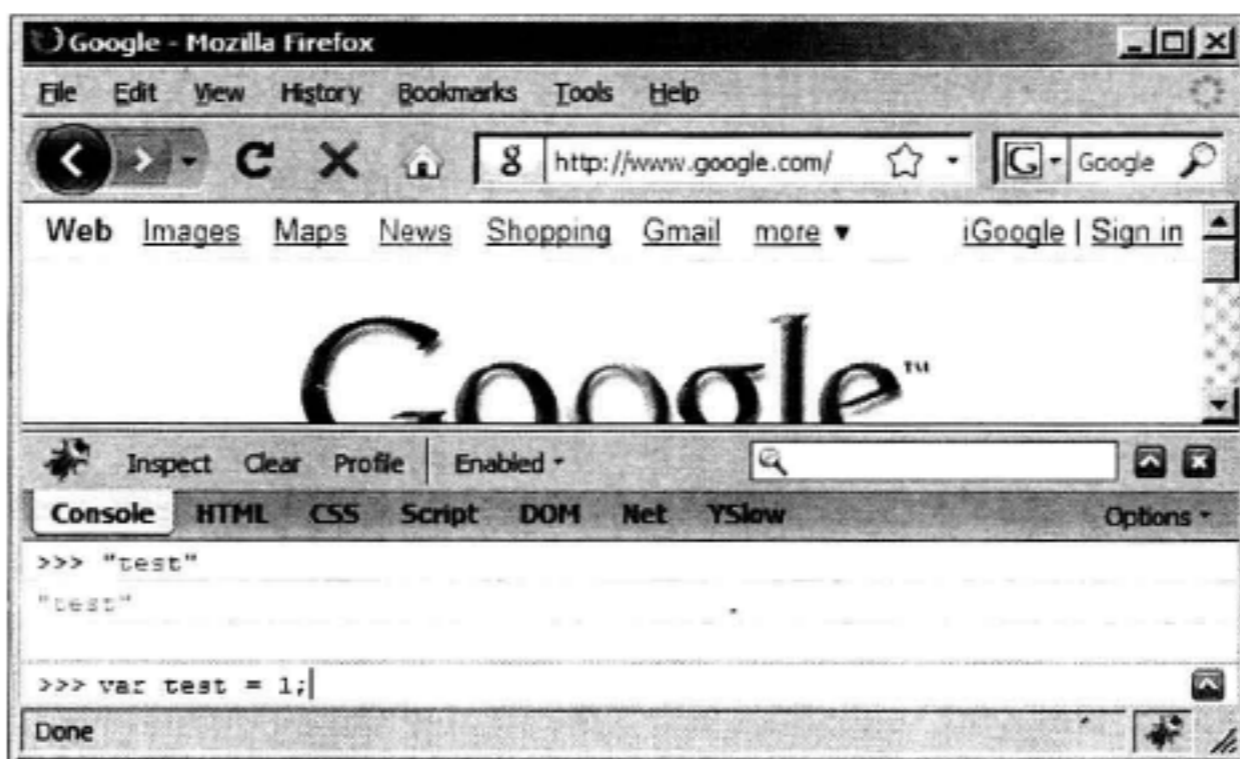


图 1-1

我们可以直接在 Firebug 控制台中输入代码，然后只要按 Enter 键，代码就会如期执行。而代码返回值就紧接着会在控制台中被打印出来。而且，这些代码会在当前所载入的页面环境中进行，例如，我们输入 `document.location.href` 就会得到当前页面的 URL。

此外，该控制台还具有一套自动完成功能，其工作方式与我们平时所用的操作系统命令行类似。举个例子，如果我们在其中输入 `docu`，然后按 Tab 键，`docu` 就会被自动补全为 `document`。这时候如果再输入一个“.”（点操作符），我们就可以通过重复按 Tab 键的方式来遍历 `document` 对象中所有可调用的方法和属性。

通过上下箭头键，我们可以随时从相关的列表找回已经执行过的命令，并在控制台重新执行它们。

通常情况下，控制台只能提供单行输入，但我们可以用分号分割的方式来执行多个 JavaScript 语句。如果还需更多的空格和语句行的话，我们也可以通过单击输入行上方状态栏右边的那个向上箭头键，来打开控制台的多命令行模式。如图 1-2 所示：

在图 1-2 中，我们通过一些代码将 Google 主页的 logo 换成了自己的图片。如您所见，我们可以在任何页面中测试我们的 JavaScript 代码。

此外，我们还应该设置一下 Firefox 中的配置选项，以便使控制台中的 JavaScript 警告等级更为严格。这将会有助于我们写出更高质量的代码，尽管警告并不等同于错误，但我们也应该尽量避免在编写代码时出现警告信息。例如，使用未声明变量并不是一个错误，但也不是一个好主意。因此 JavaScript 引擎应该会产生一个警告信息。如果我们将警告等级设置为“严格”，该信息就会在控制台中显示出来（见图 1-2）。下面，就让我们来设置一下：

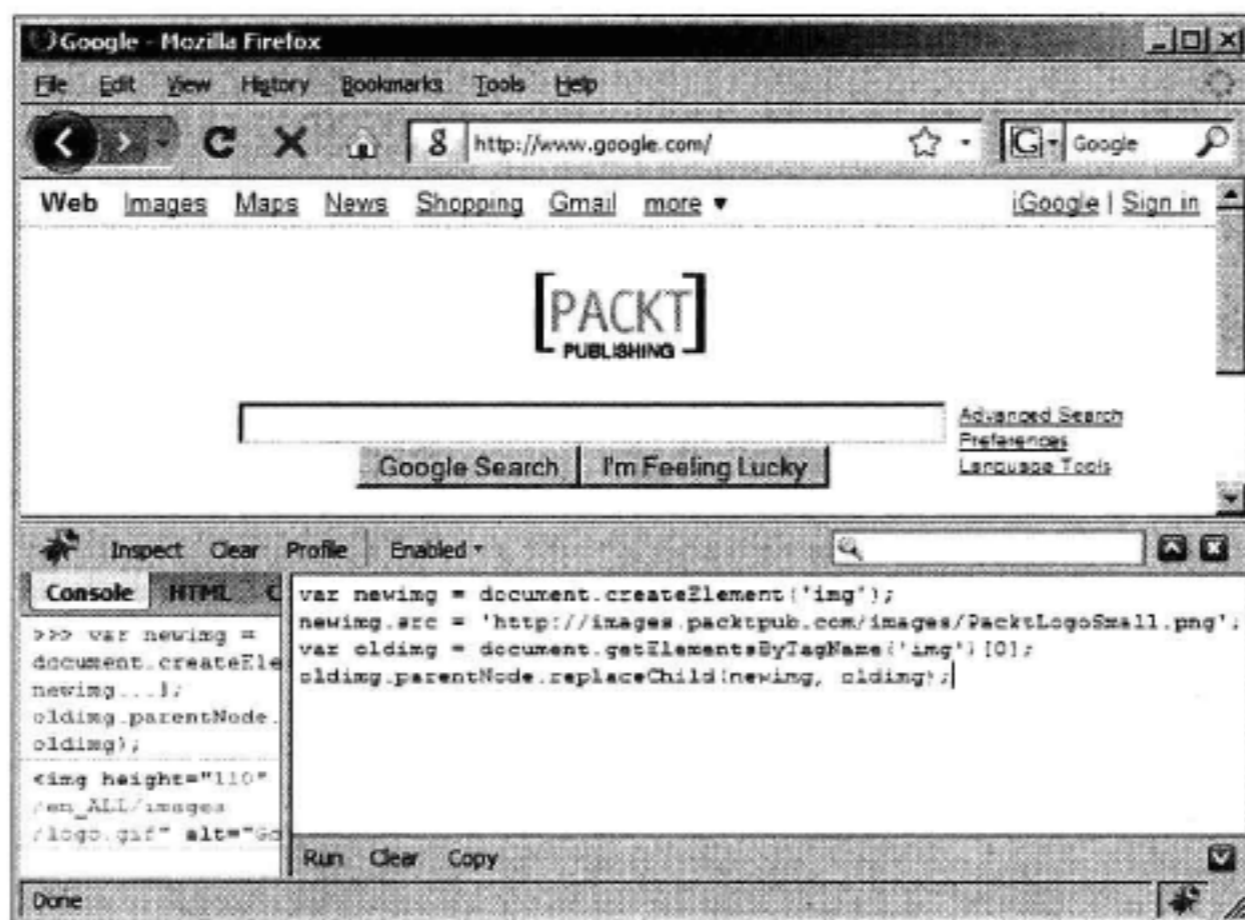


图 1-2

1. 在 Firefox 地址栏中输入 `about:config`。
2. 在过滤器的搜索栏中输入 `strict`，并按 Enter 键。
3. 双击 `javascript.options.strict` 选项所在的行，将其值改为 `true`。

1.9 本章小结

在这一章中，我们介绍了 JavaScript 这门语言的发展历程和现状。然后，我们还对面向对象程序设计的概念进行了一些基本论述。接着，我们向您详细阐述了为什么 JavaScript 不是传统的面向对象语言，而是一套独特的原型系统。最后，我们还介绍了本书的训练环境——Firebug 控制台的设置与运用。现在，您已经为下一步深入学习 JavaScript 语言，掌握其面向对象特性打下了一定的基础。本章末尾，我们列出了一些与本章内容相关的资料链接，以供您需要时参考。

- ◆ 在 YUI 视频网站 (<http://developer.yahoo.com/yui/theater/>) 上, 有 Douglas Crockford 先生^①的几个讲座是非常值得一看的。例如“Theory of the DOM”这一讲的第一部分就是关于浏览器历史的。而“The JavaScript Programming Language”这一讲的第一部分谈的就是 JavaScript 的发展史(当然也包括一些其他事情)。
- ◆ 关于 OOP 的概念, 我们可以参考 wikipedia 上的相关文章: http://en.wikipedia.org/wiki/Object-oriented_programming 以及 sun 公司所提供的 Java 文档(这里谈的是传统意义上的 OOP): <http://java.sun.com/docs/books/tutorial/java/concepts/index.html>。
- ◆ 关于 JavaScript 在今天的 application 情况, 我们或许可以从以下几个实例中获得一些启示: Yahoo! Widgets (<http://widgets.yahoo.com/>)、Google Maps (<http://maps.google.com>), 以及各个版本的 JavaScript 可视化语言处理器(<http://ejohn.org/blog/processingjs/>)。

^① Douglas Crockford 是 Web 开发领域最知名的技术权威之一, ECMA JavaScript2.0 标准化委员会委员。被 JavaScript 之父 Brendan Eich 称为 JavaScript 的大宗师 (Yoda)。曾任 Yahoo! 资深 JavaScript 架构师, 现任 PayPal 高级 JavaScript 架构师。——译者注

第 2 章

基本数据类型、数组、循环及 条件表达式

在深入学习 JavaScript 的面向对象特性之前，我们首先要了解一些基础性知识。在这一章中，我们将会从以下几个方面入手。

- ◆ JavaScript 中的基本数据类型，例如字符串和数字等。
- ◆ 数组。
- ◆ 常用操作符，例如+、-、delete、typeof 等。
- ◆ 控制流语句，例如循环和 if-else 条件表达式等。

2.1 变量

通常，变量都是用来存储数据的。当我们编写程序时，用变量来表示实际数据显然要方便些。尤其是当我们需要多次使用某个数字（例如 3.141592653589793）时，使用变量 pi 显然要比直接写数字值方便得多。而且，它们之所以能被称为“变量”，主要是它们所存储的数据在初始化之后仍然是可以改变的。正因为如此，我们在编写代码时往往会用变量来代表程序中的未知数据，例如某个计算的结果值。

变量的使用通常可分为两个步骤。

- ◆ 声明变量。
- ◆ 初始化变量，即给它一个初始值。

为了声明变量，我们需要使用 `var` 语句。像这样：

```
var a;  
var thisIsAVariable;  
var _and_this_too;  
var mix12three;
```

变量名可以由任何数字、字符及下划线组合而成。但要记住它不能以数字开头，像下面这样是不被允许的：

```
var 2three4five;
```

所谓的变量初始化，实际上指的是变量首次（或者最初）被赋值的时机。它可以有以下两种选择。

- ◆ 先声明变量，然后再初始化。
- ◆ 声明变量与初始化同步进行。

下面是后一种写法的例子：

```
var a = 1;
```

这样，我们就声明了一个名为 `a`、值为 `1` 的变量。

另外，我们也可以在单个 `var` 语句中同时声明（并初始化）多个变量，只要将它们分别用逗号分开即可，例如：

```
var v1, v2, v3 = 'hello', v4 = 4, v5;
```

区分大小写

在 JavaScript 语言中，变量名是区分大小写的。为了证明这一点，我们可以在 Firebug 控制台中测试下列语句（每输入一行按一次 Enter 键）：

```
var case_matters = 'lower';  
var CASE_MATTERS = 'upper';  
case_matters  
CASE_MATTERS
```

为了减少按键的次数，在输入第三行时，我们可以先键入 `ca` 然后按 Tab 键，控制台会自动将其补全为 `case_matters`。最后一行也是如此，我们只需先输入 `CA` 然后直接按 Tab 即可。输入完成之后，最终结果如图 2-1 所示：

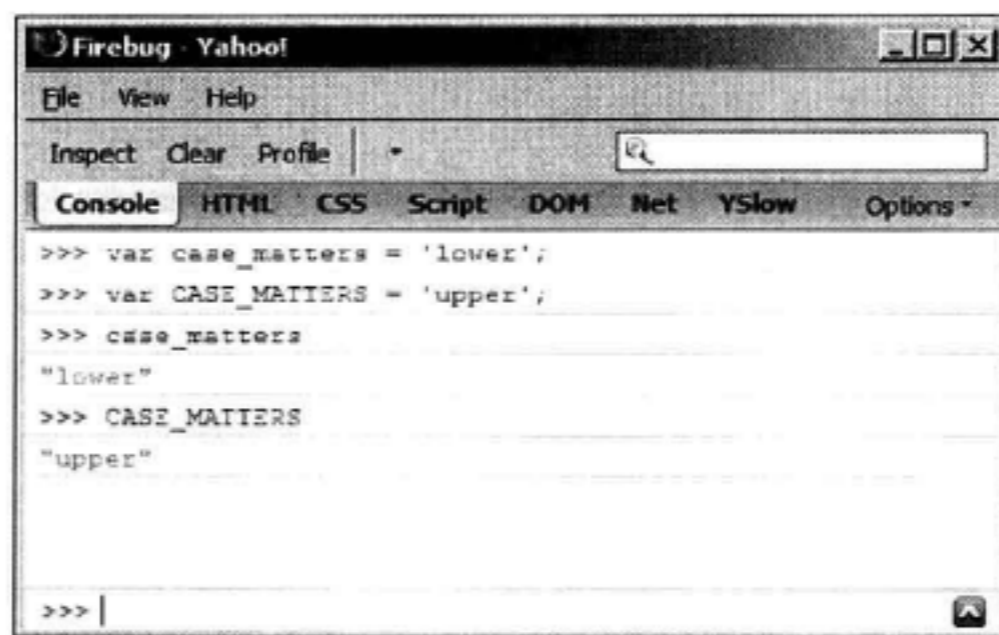


图 2-1

为方便起见，以后我们将用代码形式来代替截图。上面的例子可以表示如下：

```
>>> var case_matters = 'lower';
>>> var CASE_MATTERS = 'upper';
>>> case_matters
"lower"
>>> CASE_MATTERS
"upper"
```

如您所见，三个连续大于号（>>>）之后的内容是我们输入的代码，而其余部分则是控制台输出的结果。需要强调的是，当您测试类似的代码时，应该根据实验的实际情况来调整相关代码。这才能有助于您更好地理解语言的工作方式。

2.2 操作符

所谓操作符，通常指的是能对一两个输入执行某种操作，并返回结果的符号。为了更清晰地表达该术语的含义，我们先来看一个具体的示例：

```
>>> 1 + 2
3
```

这段代码中所包含的信息主要有以下几点。

- ◆ +是一个操作符。
- ◆ 该操作是一次加法运算。
- ◆ 输入值为 1 和 2（输入值也叫做操作数）。
- ◆ 结果值为 3。

这里的 1 和 2 都是直接参与加法运算的。现在，我们改用变量来表示它们，同时再另外声明一个变量来存储运算结果。具体如下：

```
>>> var a = 1;
>>> var b = 2;
>>> a + 1
2
>>> b + 2
4
>>> a + b
3
>>> var c = a + b;
>>> c
3
```

在表 2-1 中，我们列出了一些基本的算术运算符。

表 2-1

操作符	相关操作	代码示例
+	加法运算	>>> 1 + 2 3
-	减法运算	>>> 99.99 - 11 88.99
*	乘法运算	>>> 2 * 3 6
/	除法运算	>>> 6 / 4 1.5
%	取模运算，即求除法运算的余数	>>> 6 % 3 0 >>> 5 % 3 2 取模运算对于测试一个数的奇偶性很有用处，只需要让该数对 2 执行取模运算，返回 1 的便是奇数，返回 0 则都为偶数 >>> 4 % 2 0 >>> 5 % 2 1

(续表)

操作符	相关操作	代码示例
++	自增 1 运算	<p>当操作符后置时，操作会先返回该值，然后再递增 1</p> <pre>>>> var a = 123; var b = a++;</pre> <pre>>>> b</pre> <p>123</p> <pre>>>> a</pre> <p>124</p> <p>当操作符前置时，操作会先将值递增 1，然后再返回</p> <pre>>>> var a = 123; var b = ++a;</pre> <pre>>>> b</pre> <p>124</p> <pre>>>> a</pre> <p>124</p>
--	自减 1 运算	<p>操作符后置</p> <pre>>>> var a = 123; var b = a--;</pre> <pre>>>> b</pre> <p>123</p> <pre>>>> a</pre> <p>122</p> <p>操作符前置</p> <pre>>>> var a = 123; var b = --a;</pre> <pre>>>> b</pre> <p>122</p> <pre>>>> a</pre> <p>122</p>

事实上，当我们输入 `var a = 1;` 这样的语句时，所执行的也是一种操作。这种操作叫做纯赋值，因而 `=` 也被称为纯赋值操作符。

此外，JavaScript 中还有一组由算术运算和赋值操作组合而成的操作符。我们叫它复合

操作符。这些操作符能让我们的代码显得更为紧凑。下面来看几个示例：

```
>>> var a = 5;
>>> a += 3;
8
```

在该例中，`a += 3;`实际上就相当于 `a = a + 3;`的缩写形式。

```
>>> a -= 3;
5
```

同理，这里的 `a -= 3;` 等同于 `a = a - 3;`。

以此类推：

```
>>> a *= 2;
10
>>> a /= 5;
2
>>> a %= 2;
0
```

除了我们已经提到的算术运算与赋值操作以外，JavaScript 中还有其他各种类型的操作符。我们将会在后面的章节中陆续看到。

2.3 基本数据类型

我们在程序中所使用的任何值都是有类型的。在 JavaScript 中，主要包含以下几大基本数据类型。

1. 数字——包括浮点数与整数，例如 1、100、3.14。
2. 字符串——一序列由任意数量字符组成的序列，例如"a"、"one"、"one 2 three"。
3. 布尔值——true 或 false。
4. undefined——当我们试图访问一个不存在的变量时，就会得到一个特殊值：undefined。除此之外，使用一个未初始化的变量也会如此。因为 JavaScript 会自动将变量在初始化之前的值设定为 undefined。
5. null——这是另一种只包含一个值的特殊数据类型。所谓的 null 值，通常是指没

有值、空值，不代表任何东西。null 与 undefined 最大的不同在于，被赋予 null 的变量通常被认为是已经定义了的，只不过它不代表任何东西。关于这一点，我们稍后会通过一些具体的示例来解释。

任何不属于上述五种基本类型的值都会被认为是一个对象。甚至有时候我们也会将 null 视为对象，这会使人有些尴尬——这是一个不代表任何东西的对象（东西）。我们将在第 4 章中深入阐述对象的概念，现在我们只需要记住一点，JavaScript 中的数据类型主要分为以下两个部分。

- ◆ 基本类型（上面列出的五种类型）。
- ◆ 非基本类型（即对象）。

2.3.1 查看类型操作符——typeof

如果我们想知道某个变量或值的数据类型，可以调用一种叫做 typeof 的特殊操作符。该操作符会返回一个代表数据类型的字符串，它的值包括：“number”、“string”、“boolean”、“undefined”、“object”和“function”。在接下来的几节中，我们将会逐一展示对五种基本数据类型使用 typeof 操作符时的情况。

2.3.2 数字

最简单的数字类型当然就是整数了。如果我们将一个变量赋值为 1，并对其调用 typeof 操作符，控制台就会返回字符串“number”，请看下面的代码。此外要注意的是，当您第二次设置变量值时，就不需要再使用 var 语句了。

```
>>> var n = 1;
>>> typeof n;
"number"
>>> n = 1234;
>>> typeof n;
"number"
```

当然，这也同样适用于浮点数（即含小数部分的数字）：

```
>>> var n2 = 1.23;
>>> typeof n;①
"number"
```

^① 此处原文为 typeof n;，但根据上下文判断应属笔误，故更正为 typeof n2;。——译者注

除了对变量赋值以外，我们也可以直接对一个数值调用 `typeof`。例如：

```
>>> typeof 123;
"number"
```

2.3.2.1 八进制与十六进制

当一个数字以 0 开头时，就表示这是一个八进制数。例如，八进制数 0377 所代表的就是十进制数 255。

```
>>> var n3 = 0377;
>>> typeof n3;
"number"
>>> n3;
255
```

如您所见，例子中最后一行所输出的就是该八进制数的十进制表示形式。如果您对八进制数还不太熟悉，那么十六进制您一定不会感到陌生，毕竟，CSS 样式表中的颜色值使用的都是十六进制。

在 CSS 中，我们定义颜色的方式有以下两种。

- ◆ 使用十进制数分别指定 R（红）、G（绿）、B（蓝）的值^①，取值范围都为 0~255。例如 `rgb(0,0,0)` 代表黑色、`rgb(255,0,0)` 代表红色（红值达到最大值，而绿和蓝都为 0 值）。
- ◆ 使用十六进制数，两个数位代表一种色值，依次是 R、G、B。例如 `#000000` 代表黑色、`#ff0000` 代表红色，因为十六进制的 `ff` 就等于 255。

在 JavaScript 中，我们会用 `0x` 前缀来表示一个十六进制值（简称为 hex）。

```
>>> var n4 = 0x00;
>>> typeof n4;
"number"
>>> n4;
0
>>> var n5 = 0xff;
>>> typeof n5;
"number"
>>> n5;
255
```

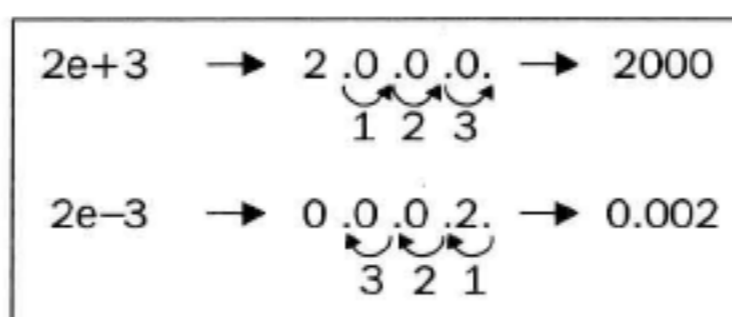
^① 三原色模式（RGB color model）是一种加色模型，是用三种原色——红色、绿色和蓝色的色光以不同的比例相加，以产生多种多样的色光。——译者注

2.3.2.2 指数表示法

一个数字可以表示成 $1e1$ （或者 $1e+1$ 、 $1E1$ 、 $1E+1$ ）这样的指数形式，意思是在数字 1 后面加 1 个 0，也就是 10。同理， $2e+3$ 的意思是在数字 2 后面加 3 个 0，也就是 2000。

```
>>> 1e1
10
>>> 1e+1
10
>>> 2e+3
2000
>>> typeof 2e+3;
"number"
```

此外，我们也可以将 $2e+3$ 理解为将数字 2 的小数点向右移三位。依照同理， $2e-3$ 也就能被理解为是将数字 2 的小数点左移三位。



```
>>> 2e-3
0.002
>>> 123.456E-3
0.123456
>>> typeof 2e-3
"number"
```

2.3.2.3 Infinity

在 JavaScript 中，还有一种叫做 Infinity 的特殊值。它所代表的是超出了 JavaScript 处理范围的数值。但 Infinity 依然是一个数字，我们可以在控制台使用 typeof 来测试 Infinity。当我们输入 $1e308$ 时，一切正常，但一旦将后面的 308 改成 309 就出界了。经实践证明，JavaScript 所能处理的最大值是 $1.7976931348623157e+308$ ，而最小值为 $5e-324$ 。

```
>>> Infinity
Infinity
>>> typeof Infinity
"number"
>>> 1e309
```



```
Infinity  
>>> 1e308  
1e+308
```

另外，任何数除 0 也为 `infinity`：

```
>>> var a = 6 / 0;  
>>> a  
Infinity
```

`Infinity` 表示的是最大数（或者比最大数还要大的数），那么最小数该如何表示呢？答案是在 `Infinity` 之前加一个负号：

```
>>> var i = -Infinity;  
>>> i  
-Infinity  
>>> typeof i  
"number"
```

但这是不是意味着我们可以得到双倍的 `Infinity` 呢？——毕竟我们可以从 0 加到 `Infinity`，也可以从 0 减到 `-Infinity`。好吧，这只是个玩笑。事实上这是不可能的，因为即便将正负 `Infinity` 相加，我们也不会得到 0，而是会得到一个叫做 `NaN`（Not A Number 的缩写，即不是数字）的东西。

```
>>> Infinity - Infinity  
NaN  
>>> -Infinity + Infinity  
NaN
```

而且，`Infinity` 与其他的任何操作数执行任何算术运算的结果，都是 `Infinity`。

```
>>> Infinity - 20  
Infinity  
>>> -Infinity * 3  
-Infinity  
>>> Infinity / 2  
Infinity  
>>> Infinity - 999999999999999999  
Infinity
```

2.3.2.4 NaN

还记得之前见过的那个 `NaN` 吗？尽管该值的名字叫做“不是数字”，但事实上它依然

属于数字，只不过是一种特殊的数字罢了。

```
>>> typeof NaN
"number"
>>> var a = NaN;
>>> a
NaN
```

如果我们在对一个假定的数字执行某个操作时失败了，就会得到一个 **NaN**。例如，当我们试图将 10 与字符 "f" 相乘时，其结果就会为 **NaN**，因为 "f" 显然是不支持乘法运算的。

```
>>> var a = 10 * "f";
>>> a
NaN
```

而且，**NaN** 是具有传染性的，只要我们的算术运算中存在一个 **NaN**，整个运算就会失败。

```
>>> 1 + 2 + NaN
NaN
```

2.3.3 字符串

字符串通常指的是一组用于表示文本的字符序列。在 JavaScript 中，一对双引号或单引号之间的任何值都会被视为一个字符串。也就是说，1 是一个数字的话，"1" 就是一个字符串了。在一个字符串上，typeof 操作符会返回 "string"。

```
>>> var s = "some characters";
>>> typeof s;
"string"
>>> var s = 'some characters and numbers 123 5.87';
>>> typeof s;
"string"
```

字符串中可以包含数字，例如：

```
>>> var s = '1';
>>> typeof s;
"string"
```

如果引号之间没有任何东西，它所表示的依然是一个字符串（即空字符串）：

```
>>> var s = ""; typeof s;
"string"
```

之前，当我们在两个数字之间使用加号时，所执行的是加法运算。但在字符串中，这是一个字符串拼接操作，它返回的是两个字符串拼接之后的结果。例如：

```
>>> var s1 = "one"; var s2 = "two"; var s = s1 + s2; s;
"onetwo"
>>> typeof s;
"string"
```

像+这样的双功能操作符可能会带来一些错误。因此，我们如果想执行拼接操作的话，最好确保其所有的操作数都是字符串。同样的，在执行数字相加时，我们也要确保其所有的操作数都是数字。至于如何做到这一点，我们将会在后继章节中详细讨论。

2.3.3.1 字符串转换

当我们将一个数字字符串用于算术运算中的操作数时，该字符串会在运算中被当做数字类型来使用。（由于加法操作符的歧义性，这条规则不适用于加法运算。）

```
>>> var s = '1'; s = 3 * s; typeof s;
"number"
>>> s
3
>>> var s = '1'; s++; typeof s;
"number"
>>> s
2
```

于是，将数字字符串转换为数字就有了一种偷懒的方法：只需将该字符串与 1 相乘即可。（当然，更好的选择是调用 `parseInt` 函数，关于这点，我们将会在下一章中介绍。）

```
>>> var s = "100"; typeof s;
"string"
>>> s = s * 1;
100
>>> typeof s;
"number"
```

如果转换操作失败了，我们就会得到一个 NaN 值。

```
>>> var d = '101 dalmatians';
>>> d * 1
NaN
```

此外，将其他类型转换为字符串也有一种偷懒的方法，只需要将其与空字符串连接即可：

```
>>> var n = 1;
>>> typeof n;
"number"
>>> n = "" + n;
"1"
>>> typeof n;
"string"
```

2.3.3.2 特殊字符串

在表 2-2 中，我们列出了一些具有特殊含义的字符串。

表 2-2

字符串	含义	示例
\ \\ \' \"	\是转义字符 当我们想要在字符串中使用引号时，就必须对它们进行转义，这样 JavaScript 才不会将其认作字符串的终止符 同理，当我们需要在字符串中使用反斜线本身时，也需要用另一个反斜线对其进行转义	<pre>>>> var s = 'I don't know'; 这样做是错误的，因为 JavaScript 会将“I don”视为字符串，而其余部分则将会被视为无效代码。正确做法如下： >>> var s = 'I don\'t know'; >>> var s = "I don\'t know"; >>> var s = "I don't know"; >>> var s = '"Hello", he said.'; >>> var s = "\"Hello\"", he said."; 转义转义字符本身： >>> var s = "1\\2"; s; "1\2"</pre>
\n	换行符	<pre>>>> var s = '\n1\n2\n3\n'; >>> s " 1 2 3 "</pre>

(续表)

字符串	含义	示例
<code>\r</code>	回车符	以下所有语句： <pre>>>> var s = '1\r2'; >>> var s = '1\n\r2'; >>> var s = '1\r\n2';</pre> 结果都为： <pre>>>> s "1 2"</pre>
<code>\t</code>	制表符	<pre>>>> var s = "1\t2" >>> s "1 2"</pre>
<code>\u</code>	<code>\u</code> 后面的字符将会被视为 Unicode 码	下面是作者的名字 Cyrillic 在保加利亚语中的写法： <pre>>>> "\u0421\u0442\u043E\u0442\u0438\u043D" "СТОЯН"</pre>

除此之外，还有一些很少被使用的特殊字符，例如：`\b`（退格符）、`\v`（纵向制表符）、`\f`（换页符）等。

2.3.4 布尔值

布尔类型中只有两种值：`true` 和 `false`。它们可用于引号以外的任何地方。

```
>>> var b = true; typeof b;
"boolean"
>>> var b = false; typeof b;
"boolean"
```

如果 `true` 或 `false` 在引号内，它就是一个字符串。

```
>>> var b = "true"; typeof b;
```

"string"

2.3.4.1 逻辑运算符

在 JavaScript 中，主要有三种逻辑运算符，它们都属于布尔运算。分别是：

- ◆ !——逻辑非（取反）。
- ◆ &&——逻辑与。
- ◆ ||——逻辑或。

在 JavaScript 中，如果我们想描述一些日常生活中非真即假的事物，就可以考虑使用逻辑非运算符：

```
>>> var b = !true;
>>> b;
false
```

如果在同一个值上执行两次逻辑非运算，其结果就等于原值^①：

```
>>> var b = !!true;
>>> b;
true
```

如果在一个非布尔值上执行逻辑运算，该值会在计算期间被转换为布尔值：

```
>>> var b = "one";
>>> !b;
false
```

如您所见，上例中的字符串"one"是先被转换为布尔值 true 然后再取反的，结果为 false。如果我们对它取反两次，结果就会为 true。例如：

```
>>> var b = "one";
>>> !!b;
true
```

使用双重取反操作可以很容易地将任何值转换为等效的布尔值。虽然这种方法很少被用到，但从另一个角度也说明了将其他类型的值转换为布尔值的重要性。而事实上，除了下面所列出特定值以外（它们将被转换为 false），其余大部分值在转换为布尔值时

^① 从上下文来看，此处应该特指布尔值。——译者注

都为 true。

- ◆ 空字符串 ""
- ◆ null
- ◆ undefined
- ◆ 数字 0
- ◆ 数字 NaN
- ◆ 布尔值 false

这 6 个值有时也会被我们统称为 falsy，而其他值则被称为 truthy（包括字符串 "0"、""、"false"）。

接下来，让我们来看看另外两个操作符——逻辑与和逻辑或的使用示例。当我们使用逻辑与操作符时，当且仅当该操作所有操作数为 true 时，它才为 true。而逻辑或操作则只需要至少一个操作数为 true 即可为 true。

```
>>> var b1 = true; var b2 = false;
>>> b1 || b2
true
>>> b1 && b2
false
```

在表 2-3 中，我们列出了所有可能的情况及其相应结果。

表 2-3

操作	结果
true && true	true
true && false	false
false && true	false
false && false	false
true true	true
true false	true
false true	true
false false	false

当然，我们也能连续执行若干个逻辑操作。例如：

```
>>> true && true && false && true
false
>>> false || true || false
true
```

我们还可以在同一个表达式中混合使用&&和||。不过在这种情况下，我们最好用括号来明确一下操作顺序。例如：

```
>>> false && false || true && true
true
>>> false && (false || true) && true
false
```

2.3.4.2 操作符优先级

您可能会想知道，为什么上例中的第一个表达式（false && false || true && true）结果为 true。答案在于操作符优先级。这看上去有点像数学，例如：

```
>>> 1 + 2 * 3
7
```

由于乘法运算的优先级高于加法，所以该表达式会先计算 2 * 3，这就相当于我们输入的表达式是：

```
>>> 1 + (2 * 3)
7
```

逻辑运算符也一样，!的优先级最高，因此在没有括号限定的情况下它将会被最先执行。然后，接下来的优先顺序是先&&后||。也就是说：

```
>>> false && false || true && true
true
```

与下面表达式等效：

```
>>> (false && false) || (true && true)
true
```



最佳方法：

尽量使用括号，而不是依靠操作符优先级来设定代码的执行顺序，这样我们的代码才能有更好的可读性。

2.3.4.3 惰性求值

如果在一个连续的逻辑操作中，操作结果在最后一个操作完成之前就已经明确了的话，那么该操作往往就不必再继续执行了，因为这已经不会对最终结果产生任何影响。例如，在下面这种情况中：

```
>>> true || false || true || false || true
true
```

在这里，所有的逻辑或运算符优先级都是相同的，只要其中任何一个操作数为 `true`，该表达式的结果就为 `true`。因而当第一个操作数被求值之后，无论后面的值是什么，结果都已经被确定了。于是我们可以允许 JavaScript 引擎偷个懒（好吧，这也是为了提高效率），在不影响最终结果的情况下省略一些不必要的求值操作。为此，我们可以在控制台中做个实验：

```
>>> var b = 5;
>>> true || (b = 6)
true
>>> b
5
>>> true && (b = 6)
6
>>> b
6
```

除此之外，上面的例子还向我们显示了另一个有趣的事情——如果 JavaScript 引擎在一个逻辑表达式中遇到一个非布尔类型的操作数，那么该操作数的值就会成为该表达式所返回的结果。例如：

```
>>> true || "something"
true
>>> true && "something"
"something"
```

通常情况下，这种行为是应该尽量避免的，因为它会使我们的代码变得难以理解。但在某些时候这样做也是有用的。例如，当我们不能确定某个变量是否已经被定义时，就可以像下面这样，即如果变量 `mynumber` 已经被定义了，就保留其原有值，否则就将它初始化为 10。

```
var mynumber = mynumber || 10;
```

这种做法简单而优雅，但是请注意，这也不是绝对安全的。如果这里的 `mynumber` 之前

被初始化为 0（或者是那 6 个 falsy 值中的任何一个），这段代码就不太可能如我们所愿了。

2.3.4.4 比较运算符

在 JavaScript 中，还有另外一组以布尔值为返回值类型的操作符，即比较操作符。下面让我们通过表 2-4 来了解一下它们以及相关的示例。

表 2-4

操作符	操作说明	代码示例
==	相等运算符 当两个操作数相等时返回 true。在该比较操作执行之前，两边的操作数会被自动转换为相同类型	<pre>>>> 1 == 1 true >>> 1 == 2 false >>> 1 == '1' true</pre>
===	等价运算符 当且仅当两个操作数的值和类型都相同时返回 true。这种比较往往更可靠，因为其幕后不存在任何形式的类型转换	<pre>>>> 1 === '1' false >>> 1 === 1 true</pre>
!=	不等运算符 当两个操作数不相等时返回 true（存在类型转换）	<pre>>>> 1 != 1 false >>> 1 != '1' false >>> 1 != '2' true</pre>
!==	不等价运算符 此操作内不允许类型转换。且当两个操作数的值或类型不相等时返回 true	<pre>>>> 1 !== 1 false >>> 1 !== '1' true</pre>
>	当且仅当左操作数大于右操作数时返回 true	<pre>>>> 1 > 1 false >>> 33 > 22 true</pre>

(续表)

操作符	操作说明	代码示例
>=	当且仅当左操作数大于或等于右操作数时返回 true	<pre>>>> 1 >= 1 true</pre>
<	当且仅当左操作数小于右操作数时返回 true	<pre>>>> 1 < 1 false >>> 1 < 2 true</pre>
<=	当且仅当左操作数小于或等于右操作数时返回 true	<pre>>>> 1 <= 1 true >>> 1 <= 2 true</pre>

还有一件有趣的事情要提醒读者注意：NaN 不等于任何东西，包括它自己。

```
>>> NaN == NaN
false
```

2.3.5 undefined 与 null

通常情况下，当我们试图访问某个不存在的或者未经赋值的变量时，就会得到一个 `undefined` 值。JavaScript 会自动将声明时没有进行初始化的变量设为 `undefined`。

当我们试图使用一个不存在的变量时，就会得到这样的错误信息：

```
>>> foo
foo is not defined
```

这时候，如果我们在该变量上调用 `typeof` 操作符，就会得到字符串 “`undefined`”：

```
>>> typeof foo
"undefined"
```

如果我们声明一个变量时没有对其进行赋值，调用该变量时并不会出错，但 `typeof` 操作符依然会返回 “`undefined`”。

```
>>> var somevar;
```

```
>>> somevar
>>> typeof somevar
"undefined"
```

而 `null` 值就完全是另一回事了。它不能通过 JavaScript 来自动赋值，只能通过我们的代码来完成。

```
>>> var somevar = null
null
>>> somevar
null
>>> typeof somevar
"object"
```

尽管 `undefined` 和 `null` 之间的差别微乎其微，但有时候也很重要。例如，当我们对其分别执行某种算术运算时，结果就会截然不同：

```
>>> var i = 1 + undefined; i;
NaN
>>> var i = 1 + null; i;
1
```

这是因为 `null` 和 `undefined` 在被转换为其他基本类型时，方法存在一定的区别，下面我们给出一些可能的转换类型。

转换成数字：

```
>>> 1*undefined
NaN
>>> 1*null
0
```

转换成布尔值：

```
>>> !!undefined
false
>>> !!null
false
```

转换成字符串：

```
>>> "" + null
```

```
"null"  
>>> "" + undefined  
"undefined"
```

2.4 基本数据类型综述

现在，让我们来快速总结一下目前为止所讨论过的内容。

- ◆ JavaScript 语言中有五大基本数据类型：
 - ◆ 数字
 - ◆ 字符串
 - ◆ 布尔值
 - ◆ undefined
 - ◆ null
- ◆ 任何不属于基本类型的东西都属于对象。
- ◆ 数字类型可以存储的数据包括：正负整数、浮点数、十六进制数与八进制数、指数以及特殊数值 NaN、Infinity、-Infinity。
- ◆ 字符串类型存储的是一对引号之间的所有字符。
- ◆ 布尔类型的值只有两个：true 和 false。
- ◆ null 类型的值只有一个：null。
- ◆ undefined 类型的值只有一个：undefined。
- ◆ 绝大部分值在转换为布尔类型时都为 true，但以下 6 种 falsy 值除外：
 - ◆ ""
 - ◆ null
 - ◆ undefined
 - ◆ 0
 - ◆ NaN
 - ◆ false

2.5 数组

现在，我们对 JavaScript 中的基本数据类型已经有了一定的了解，是时候将注意力转向更有趣的数据结构——数组了。

我们可以用一对不带任何内容的方括号来声明一个空数组变量，例如：

```
>>> var a = [];  
>>> typeof a;  
"object"
```

如您所见，`typeof` 在这里返回的是“**object**”。建议您先别管这个，待我们重点讨论对象时再回头来说明这个问题。

如果我们要定义一个带三个元素的数组，可以这样做：

```
>>> var a = [1,2,3];
```

只要在控制台中简单地输入数组名，就能打印出该数组中的所有内容：

```
>>> a  
[1, 2, 3]
```

那么，究竟什么是数组呢？简而言之，它就是一个用于存储数据的列表。与一次只能存储一个数据值的变量不同，我们可以用数组来存储任意数量的元素值。现在的问题是，我们应该如何访问数组中的各个数据值呢？

通常，元素在数组中的索引位置是从 0 开始编号的。也就是说，数组首元素的索引值（或者说位置值）应该是 0，第二个元素的索引值则是 1，以此类推。表 2-5 中所显示的就是之前那个三元素数组实例中的具体情况。

表 2-5

Index	Value
0	1
1	2
2	3

为了访问特定的数组元素，我们需要用一对方括号来指定元素的索引值。因此 `a[0]` 所

访问的就是数组 `a` 的首元素，而 `a[1]` 则代表第二个元素，以此类推。

```
>>> a[0]
1
>>> a[1]
2
```

2.5.1 增加、更新数组元素

我们可以通过索引来更新数组中的元素。例如在下面的代码中，我们更新了第三个元素（索引值为 2）的值，并将更新后的数组打印出来：

```
>>> a[2] = 'three';
"three"
>>> a
[1, 2, "three"]
```

另外，我们也可以通过索引一个之前不存在的位置，来为其添加更多的数组元素。

```
>>> a[3] = 'four';
"four"
>>> a
[1, 2, "three", "four"]
```

如果新元素被添加的位置与原数组末端之间存在一定的间隔，那么这之间的元素将会被自动设定为 `undefined` 值。例如：

```
>>> var a = [1,2,3];
>>> a[6] = 'new';
"new"
>>> a
[1, 2, 3, undefined, undefined, undefined, "new"]
```

2.5.2 删除元素

为了删除特定的元素，我们需要用到 `delete` 操作符。该操作符虽然不能真正移除一个元素，但它能将其设定为 `undefined`。元素被删除后，数组的长度并不会受到影响。

```
>>> var a = [1, 2, 3];
>>> delete a[1];
true
>>> a
```

```
[1, undefined, 3]
```

2.5.3 数组的数组

通常情况下，我们可以在数组中存放任何类型的值，包括另一个数组。

```
>>> var a = [1, "two", false, null, undefined];
>>> a
[1, "two", false, null, undefined]
>>> a[5] = [1,2,3]
[1, 2, 3]
>>> a
[1, "two", false, null, undefined, [1, 2, 3]]
```

让我们来看一个例子，在下面的代码中，我们定义了一个含有两个数组的数组：

```
>>> var a = [[1,2,3],[4,5,6]];
>>> a
[[1, 2, 3], [4, 5, 6]]
```

在该数组中，首元素 `a[0]` 本身也是一个数组。

```
>>> a[0]
[1, 2, 3]
```

如果想要访问内层数组中的特定元素，我们需要再加一组方括号。例如：

```
>>> a[0][0]
1
>>> a[1][2]
6
```

另外值得注意的是，我们通过这种访问数组方式来获取某个字符串中的特定字符。例如：

```
>>> var s = 'one';
>>> s[0]
"o"
>>> s[1]
"n"
>>> s[2]
"e"
```

除此之外，数组的使用方法还有很多（我们将会在第4章中详细介绍），现在先到此为

止，请记住以下内容。

- ◆ 数组是一种数据存储形式。
- ◆ 数组元素是可以被索引的。
- ◆ 数组中的元素索引是从 0 开始的，并且按照每个元素的位置依次递增。
- ◆ 我们是通过方括号中的索引值来访问数组元素的。
- ◆ 数组能存储任何类型的数据，包括另一个数组。

2.6 条件与循环

条件表达式是一种简单而强大的控制形式，它能够帮助我们控制一小段代码的执行走向。而循环则是一种可以让我们重复执行某段代码的操作。接下来，我们将会学习以下内容。

- ◆ if 条件表达式。
- ◆ switch 语句。
- ◆ while、do-while、for，以及 for-in 循环。

2.6.1 代码块

首先，我们需要先了解一下什么是代码块，这在条件表达式和循环体中随处可见。

所谓的代码块，通常指的是被包括在 0 对或多对大括号中的那一段代码。

```
{  
  var a = 1;  
  var b = 3;  
}
```

每个代码块中都可以内嵌另一个代码块，并且可以无限制地嵌套下去^①。

```
{  
  var a = 1;  
  var b = 3;  
  var c, d;  
}
```

^① 这里的“无限制”是纯语言意义上的，由于每一层嵌套都意味着内存中的一个独立栈，所以嵌套的深度必须考虑内存的实际使用情况。——译者注

```

{
  c = a + b;
  {
    d = a - b;
  }
}
}

```

最佳实践：

- ◆ 尽量使用分号来作为每一行的结束。尽管这在语法上是可选的，但对于开发来说是一个很好的习惯。为了让代码获得最佳的可读性，我们在代码块中的表达式最好是一行一个，并用分号彼此隔开。
- ◆ 尽量对代码块中的所有代码使用缩进格式。有些人会用 tab 来做缩进，而有些则会使用四个或两个空格。这都无关紧要，只要保持前后一致就行。在上面那个例子中，我们在最外层用了两个空格的缩进，在首层嵌套中用了四个空格，而第二层则是六个空格。
- ◆ 尽量使用大括号。当代码块中只有一个表达式时，大括号实际上是可选的。但为了增加代码的可读性和可维护性，我们最好还是养成加大括号的习惯，即使这不是必需的。



现在，准备好开始学习循环和条件语句了吗？另外要提醒的是，接下来我们需要将 Firebug 控制台切换到多行模式。

2.6.1.1 if 条件表达式

让我们先来看一个简单的 if 条件表达式：

```

var result = '';
if (a > 2) {
  result = 'a is greater than 2';
}

```

如您所见，该表达式通常主要由以下几个部分组成。

- ◆ if 语句。

- ◆ 括号中的条件部分——判断“a 是否大于 2”。
- ◆ 当 if 条件满足时所要执行的代码块。

其中，条件部分（即括号内的部分）通常由某些返回布尔值的操作组成，主要有以下几种形式。

- ◆ 逻辑类操作，包括!、&&、||等。
- ◆ 比较类操作，包括===、!==、>等。
- ◆ 一个可以转换为布尔类型的值或变量。
- ◆ 以上几种形式的组合。

除此之外，if 表达式中还有一个可选项，即如果条件部分的表达式返回 false 的话，我们也可以执行后面 else 子句中的代码块。例如：

```
if (a > 2) {
  result = 'a is greater than 2';
} else {
  result = 'a is NOT greater than 2';
}
```

而且，我们还可以在 if 和 else 之间插入无数个 else if 子句。例如：

```
if (a > 2 || a < -2) {
  result = 'a is not between -2 and 2';
} else if (a === 0 && b === 0) {
  result = 'both a and b are zeros';
} else if (a === b) {
  result = 'a and b are equal';
} else {
  result = 'I give up';
}
```

另外，我们也可以当前的 if 代码块中再内嵌一个新的条件语句。

```
if (a === 1) {
  if (b === 2) {
    result = 'a is 1 and b is 2';
  } else {
    result = 'a is 1 but b is not 2';
  }
} else {
```

```

    result = 'a is not 1, no idea about b';
}

```

2.6.1.2 检查变量是否存在

if 表达式在检查一个变量是否存在时往往非常有用。其中，最懒的方法就是其条件部分中直接使用变量，例如 `if(somevar){...}`。但这样做并不一定是最合适的。我们可以来测试一下。在下面这段代码中，我们将会检查程序中是否存在一个叫做 `somevar` 的变量，如果存在，就将变量 `result` 设置为 `yes`。

```

>>> var result = '';
>>> if (somevar){result = 'yes';}
somevar is not defined
>>> result;
""

```

这段代码显然是可以工作的，因为最终的结果肯定不会是“yes”。但首先，这段代码会产生一个警告信息：“**somevar is not defined**”，作为一个 JavaScript 高手，您肯定不希望自己的代码多此一举。其次，就算 `if(somevar)` 返回的是 `false`，也并不意味着 `somevar` 就一定没有定义，它也可以是任何一种被初始化为 falsy 值（如 `false` 或 `0`）的已声明变量。

所以在检查变量是否存在时，更好的选择是使用 `typeof`。

```

>>> if (typeof somevar !== "undefined"){result = 'yes';}
>>> result;
""

```

在这种情况下，`typeof` 返回的就是一个字符串，这样就可以与“undefined”进行直接比对。但需要注意的是，如果这里的 `somevar` 是一个已经声明但尚未赋值的变量，结果也是相同的。也就是说，我们实际上是在用 `typeof` 测试一个变量是否已经被初始化（或者说测试变量值是否为 `undefined`）。

```

>>> var somevar;
>>> if (typeof somevar !== "undefined"){result = 'yes';}
>>> result;
""
>>> somevar = undefined;
>>> if (typeof somevar !== "undefined"){result = 'yes';}
>>> result;
""

```

也就是说，只有当一个变量被定义并初始化为 `undefined` 以外的值时，`typeof` 所返回的类型才不会等于“`undefined`”。

```
>>> somevar = 123;
>>> if (typeof somevar !== "undefined"){result = 'yes';}
>>> result;
"yes"
```

2.6.1.3 替代 if 表达式

如果我们所面对的条件表达式非常简单，就要考虑用其他形式来替代 `if` 表达式。例如下面这段代码：

```
var a = 1;
var result = '';
if (a === 1) {
  result = "a is one";
} else {
  result = "a is not one";
}
```

我们完全可以将其简化为：

```
var result = (a === 1) ? "a is one" : "a is not one";
```

但需要提醒的是，这种语法通常只用于一些非常简单的条件逻辑，千万不要滥用。因为这样做很容易使我们的代码变得难以理解。

另外，这里的 `?` 操作符也叫做三无运算符。

2.6.1.4 switch 语句

当我们发现自己在 `if` 表达式中使用了太多的 `else if` 子句时，就应该要考虑用 `switch` 语句来替代 `if` 了。

```
var a = '1';
var result = '';
switch (a) {
  case 1:
    result = 'Number 1';
    break;
  case '1':
```

```
    result = 'String 1';
    break;
default:
    result = 'I don\'t know';
    break;
}
result;
```

显然，这段代码的执行结果为“**String 1**”。现在，让我们来看看 switch 表达式主要由哪几部分组成：

- ◆ switch 子句。
- ◆ 括号中的表达式，这里通常会是一个变量，但也可以是其他任何能返回某值的东西。
- ◆ 包含在大括号中的 case 序列块。
- ◆ 每个 case 语句后面有一个表达式，该表达式的结果将会与 switch 语句的表达式进行比对。如果比对的结果为 true，则 case 语句中冒号之后的代码将会被执行。
- ◆ break 语句是可选的，它实际上是 case 块的结束符，即当代码执行到 break 语句时，整个 switch 语句就执行完成了，否则就继续执行下一个 case 块，而这通常是应该避免的。
- ◆ default 语句也是可选的，其后的代码块只有在上面所有的 case 表达式都不为 true 时才会被执行。

换句话说，整个 switch 语句的执行应该可以分为以下几个步骤。

1. 对 switch 语句后面的括号部分进行求值，并记录结果。
2. 移动到第一个 case 块，将它的值与步骤 1 的结果进行比对。
3. 如果步骤 2 中的比对结果为 true，则执行该 case 块中的代码。
4. 在相关 case 块执行完成之后，如果遇到 break 语句就直接退出 switch。
5. 如果没有遇到 break 或步骤 2 中的比对结果为 false，就继续下一个 case 块，然后重复步骤 2 到 5 中的操作。
6. 如果依然还没有结束（也就是始终未能按照步骤 4 中的方式退出），就执行 default 语句后面的代码块。

最佳实践

◆ 对 case 进行缩进, 并对后面的代码部分进行再次缩进。不要忘了 break。

◆ 有时候, 我们会希望故意省略一些 break 语句, 当然, 这种叫做贯穿 (fall-through) 的做法在实际应用中并不常见, 因为它通常会被误认为是人为的遗漏。故而使用时往往需要在文档中加以说明。但从另一方面来说, 如果我们真的有意让两个相邻的 case 语句共享同一段代码的话, 这样做并没有什么不妥。只不过, 这不能改变相关的规则, 即如果执行代码是写在 case 语句之后的话, 它依然应该以 break 结尾。另外在缩进方面, break 是选择与 case 对齐还是与相关的代码块对齐, 完全取决于个人喜好, 只要保持风格的一致性即可。

尽量使用 default 语句。因为这可以使我们在 switch 找不到任何匹配的情况下, 也依然能做一些有意义的事情。



2.6.2 循环

通过 if-else 和 switch 语句, 我们可以在代码中采取不同的执行路径。当我们处于某种十字路口时, 就可以根据某个具体的条件来选择自己的走向。然而, 循环就完全是另一回事了, 我们可以利用它使代码在返回主路径之前先去执行某些重复操作。至于重复的次数, 则完全取决于我们设定在每次迭代之前 (或之后) 的条件值。

比如说, 我们的程序通常都是在 A 点到 B 点之间运行, 如果我们在这之间设置了一个条件 C, 而这个条件的值将会决定我们是否要进入循环 L。那么一旦进入了循环, 我们就必须在每次迭代完成之后对该条件进行重新求值, 以判断是否要执行下一次迭代。总之, 我们最终还是会回到通往 B 点的路径上来的。

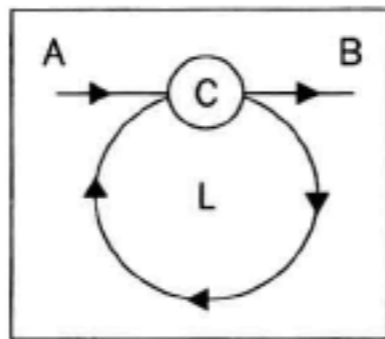
当某循环的条件永为 true 时, 它就成了一个无限循环。这意味着代码将会被“永远”困在循环中。这无疑是一个逻辑上的错误, 我们必须对此加以防范。

在 JavaScript 中, 循环主要有以下四种类型:

- ◆ while 循环
- ◆ do-while 循环

- ◆ for 循环
- ◆ for-in 循环

2.6.2.1 while 循环



while 循环是最为简单的一种循环，它们通常是这样的：

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

while 语句主要分为两个部分：小括号中的条件和大括号中的代码块。当且仅当条件值为 true 时，代码块才会被反复执行。

2.6.2.2 do-while 循环

do-while 循环实际上是 while 循环的一种轻微的变种。示例如下：

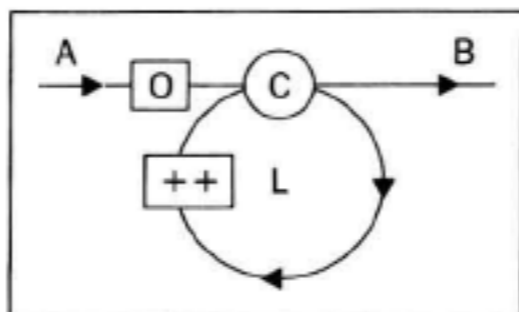
```
var i = 0;
do {
    i++;
} while (i < 10)
```

在这里，do 语句后面先出现的是代码块，然后才是条件。条件出现在代码块之后，这意味着代码块无论如何都会被执行一次，然后再去对条件部分进行求值。

如果我们将上面两个示例中的 i 初始化为 11 而不是 0 的话，第一个例子（while 循环）中，代码块将不会执行，i 最终的值仍然是 11，而第二个例子（do-while 循环）中的代码块将会被执行一次，而 i 的值也会变为 12。

2.6.2.3 for 循环

for 是使用得最为广泛的循环类型，也是我们最应该掌握的内容。实际上，这也只需要掌握一点点语法知识。



在条件 C 和代码块 L 的基础上，我们还需要增加以下两个部分的内容。

- ◆ 初始化部分——在进入循环之前所要执行的代码（即图中 O 所标志的内容）。
- ◆ 自增部分——每次迭代完成后所要执行的代码（即图中 ++ 所标志的内容）。

最常用的 for 循环模式主要包括以下内容。

- ◆ 在初始化部分中，我们会定义一个循环变量（通常命名为 i），例如 `var i = 0;`。
- ◆ 在条件部分中，我们会将 i 与循环边界值进行比对。例如 `i < 100`。
- ◆ 在自增部分中，我们会将循环变量 i 自增 1，如 `i++`。

下面来看一个具体示例：

```

var punishment = '';
for (var i = 0; i < 100; i++) {
    punishment += 'I will never do this again, ';
}
  
```

实际上，这三个部分（初始化、循环条件、自增操作）都可以写成用逗号分割的多重表达式。例如，我们可以重写一遍上面的例子，在其初始化部分中增加一个 punishment 变量的定义。

```

for (var i = 0, punishment = ''; i < 100; i++) {
    punishment += 'I will never do this again, ';
}
  
```

那么，我们能不能把循环体中的内容移到自增部分中去呢？当然可以，尤其当其中只有一行内容时。只不过这样的循环看上去有点令人尴尬，因为它没有循环体了。

```

for (var i = 0, punishment = '';
     i < 100;
     i++, punishment += 'I will never do this again, ')
{
    // nothing here
}
  
```

事实上，这三部分也都是可选的，上面的例子也完全可以写成下面这样：

```
var i = 0, punishment = '';
for (;;) {
    punishment += 'I will never do this again, ';
    if (++i == 100) {
        break;
    }
}
```

尽管代码重写之后的工作方式与原来相同，但它显得更长了，可读性也更差了。我们完全可以用 while 循环来取代它。但 for 循环可以使代码更紧凑、更严谨。它的三个部分（初始化、循环条件、自增操作）泾渭分明，语法也更为纯粹。这些都有利于我们理清程序的逻辑，从而避免类似于无限循环这样的麻烦。

另外，for 循环还可以彼此嵌套。下面，我们来看一个嵌套循环的具体示例。假设该例要打印一个 10 行 10 列的星号字符串，那么我们就可以用 i 来表示行数，j 则表示列数，以构成一个“图形”：

```
var res = '\n';
for(var i = 0; i < 10; i++) {
    for(var j = 0; j < 10; j++) {
        res += '* ';
    }
    res += '\n';
}
```

最终，该字符串输出如下：

```
"
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
"
```

另外，我们还可以用嵌套循环和取模运算画出一个雪花状的图形，代码如下：

```
var res = '\n', i, j;
```

```
for(i = 1; i <= 7; i++) {
  for(j = 1; j <= 15; j++) {
    res += (i * j) % 8 ? ' ' : '*';
  }
  res+= '\n';
}
"
```

```

      *
*   *   *
      *
* * * * * * *
      *
*   *   *
      *
"
```

2.6.2.4 for-in 循环

for-in 循环往往被用来遍历某个数组（或对象，这一点我们以后再讨论）中的元素。这似乎也是它唯一的用处，该循环不能用来替代 for 或 while 循环，执行某些一般性的重复操作。下面，我们来看一个 for-in 遍历数组元素的示例。当然，例子仅供参考。毕竟对于 for-in 循环来说，它最适用的场合依然是对象，以及用于常规 for 循环的数组。

在下面的示例中，我们将遍历数组中的所有元素，并打印出当前所在的索引位置（即键值）和元素值。

```
var a = [ 'a', 'b', 'c', 'x', 'y', 'z'];
var result = '\n';
for (var i in a) {
  result += 'index: ' + i + ', value: ' + a[i] + '\n';
}
```

结果如下：

```
"
index: 0, value: a
index: 1, value: b
index: 2, value: c
index: 3, value: x
```

```
index: 4, value: y
index: 5, value: z
"
```

2.7 注释

现在，我们来看本章最后一个内容：注释。通过注释这种形式，我们可以将自己的一些想法放在 JavaScript 代码中。由于注释中的内容会被 JavaScript 引擎自动忽略掉，因此它们不会对程序产生任何影响。而当您几个月后重新考虑这段代码，或将其转让给其他人维护时，这些注释就会显得非常重要。

注释的形式主要有以下两种。

- ◆ 单行注释——以//开头并直至该行结束。
- ◆ 多行注释——以/*开头，并以*/结尾，其中可以包括一行或多行内容。但要记住，注释首尾符之间的任何代码都将会被忽略。

具体示例如下：

```
// beginning of line
var a = 1; // anywhere on the line
/* multi-line comment on a single line */
/*
    comment
    that spans
    several lines
*/
```

甚至，有些实用工具（例如 JSDoc）可以从我们的代码中提取相关的注释，并据此生成有意义的项目文档。

2.8 本章小结

在这一章中，我们学习了编写一个 JavaScript 程序所需要的基本组件。现在，您应该已经掌握了以下几种基本数据类型。

- ◆ 数字
- ◆ 字符串

- ◆ 布尔值
- ◆ `undefined`
- ◆ `null`

你也已经了解了一些基本的操作符。

- ◆ 算术运算符: `+`、`-`、`*`、`/`、`%`。
- ◆ 自增(减)运算符: `++`、`--`。
- ◆ 赋值运算符: `=`、`+=`、`-=`、`*=`、`/=`、`%=`。
- ◆ 特殊操作符: `typeof`、`delete`。
- ◆ 逻辑运算符: `&&`、`||`、`!`。
- ◆ 比较运算符: `==`、`===`、`!=`、`!==`、`<`、`>`、`>=`、`<=`。

另外,我们还学习了如何使用数组来存储和访问数据。最后,我们还为您介绍了几种不同的控制程序流程的方法——条件(`if-else` 和 `switch` 语句)和循环(`while`、`do-while`、`for`、`for-in` 语句)。

本章的信息量确实不小,因此我们建议您通过下面的练习巩固一下。在继续深入下一章的学习之前,我们需要给自己一些鼓励。

2.9 练习题

1. 如果我们在控制台中执行下列语句,结果分别是什么?为什么?

- ◆ `var a; typeof a;`
- ◆ `var s = '1s'; s++;`
- ◆ `!!"false"`
- ◆ `!!undefined`
- ◆ `typeof -Infinity`
- ◆ `10 % "0"`
- ◆ `undefined == null`
- ◆ `false === ""`

- ◆ `typeof "2E+2"`
- ◆ `a = 3e+3; a++;`

2. 执行下面的语句后，`v` 的值会是什么？

```
>>> var v = v || 10;
```

如果将 `v` 分别设置为 100、0、`null`，或者卸载它（即 `delete v`），结果又将是什么？

3. 编写一个打印乘法口诀表的脚本程序。提示：使用嵌套循环来实现。

第 3 章

函数

掌握函数对于学习任何程序设计语言来说都是非常重要的。尤其对于 JavaScript 来说，更是如此，因为该语言中的很多功能、灵活性以及表达能力都来自函数。例如，绝大部分语言都有自己专门的面向对象语法，而 JavaScript 就是通过函数来实现的。在这一章中，我们首先要掌握如下内容：

- ◆ 如何定义和使用函数
- ◆ 如何向函数传递参数
- ◆ 了解我们可以“免费”调用哪些预定义函数
- ◆ 了解 JavaScript 中的变量作用域
- ◆ 理解“函数也是数据”的概念，并将其视为一种特殊的数据类型。

理解了上述内容之后，我们就可以继续深入本章的第二部分。在这一部分中，您将会看一些有趣的函数应用：

- ◆ 匿名函数的调用
- ◆ 回调函数
- ◆ 自调函数
- ◆ 内嵌函数（在函数内部定义的函数）
- ◆ 以函数为返回值的函数
- ◆ 能重定义自身的函数
- ◆ 闭包

3.1 什么是函数

所谓函数，本质上是一种代码的分组形式。我们可以通过这种形式赋予某组代码一个名字，便于日后重用。下面，我们来示范一下函数的声明：

```
function sum(a, b) {  
    var c = a + b;  
    return c;  
}
```

一般来说，函数声明通常由以下几部分组成：

- ◆ function 子句。
- ◆ 函数名称，即这里的 sum。
- ◆ 函数所需的参数，即这里的 a、b。一个函数通常都具有 0 个或多个参数。参数之间用逗号分割。
- ◆ 函数所要执行的代码块，我们称之为函数体。
- ◆ return 子句。函数通常都会有返回值，如果某个函数没有显式的返回值，我们就会默认它的返回值为 undefined。

需要注意的是，一个函数只能有一个返回值，如果我们需要同时返回多个值，可以考虑将其放进一个数组里，以数组元素的形式返回。

3.1.1 调用函数

如果我们使用一个函数，就必须要去调用它。调用的方式很简单，只需在函数名后面加一对用以传递参数的括号即可。另外，对于“调用 (to call)”这种操作，我们有时也可以将其称之为“请求 (to invoke)”某个函数。

现在，让我们来调用一下 sum() 函数，先将两个参数传递给该函数，然后再将函数的返回值赋值给变量 result。具体如下：

```
>>> var result = sum(1, 2);  
>>> result;  
3
```

3.1.2 参数

在定义一个函数的同时，我们往往会设置该函数所需的调用参数。当然，您也可以不

给它设定参数，但如果您设定了，而又在调用时忘了传递相关的参数值，JavaScript 引擎就会自动将其设定为 `undefined`。例如在下面这个调用中，函数返回的是 `NaN`，因为这里试图将 1 与 `undefined` 相加。

```
>>> sum(1)
NaN
```

而对于那些已经传递进来的参数，JavaScript 是来者不拒的。所以即便我们传递的参数过多，多余的那部分也只会默默地忽略掉。

```
>>> sum(1, 2, 3, 4, 5)
3
```

而且，我们实际上还可以创建一些在参数数量方面更为灵活的函数。这得益于每个函数内部都有一个内建的 `arguments` 数组，它能返回函数所接收的所有参数。例如：

```
>>> function args() { return arguments; }
>>> args();
[]
>>> args( 1, 2, 3, 4, true, 'ninja');
[1, 2, 3, 4, true, "ninja"]
```

通过 `arguments` 数组，我们可以进一步完善 `sum()` 函数的功能，使之能对任意数量的参数执行求和运算。

```
function sumOnSteroids() {
  var i, res = 0;
  var number_of_params = arguments.length;
  for (i = 0; i < number_of_params; i++) {
    res += arguments[i];
  }
  return res;
}
```

下面，我们用不同数量的参数（包括没有参数）来测试该函数，看看它是否能按照我们预计的方式工作：

```
>>> sumOnSteroids(1, 1, 1);
3
>>> sumOnSteroids(1, 2, 3, 4);
10
>>> sumOnSteroids(1, 2, 3, 4, 4, 3, 2, 1);
```

```
20
>>> sumOnSteroids(5);
5
>>> sumOnSteroids();
0
```


其中，表达式 `arguments.length` 返回的是函数被调用时所接收的参数数量。如果您对这段代码中的某些语法不太熟悉，也不必太担心，我们将会在第4章中详细讨论它们。到那时，您会发现 `arguments` 实际上不是一个数组，而是一个类似数组的对象。

3.2 预定义函数

JavaScript 引擎中有一组可供随时调用的内建函数。下面，让我们来了解一下这些函数。在这一过程中，我们会通过一系列具体的函数实践，来帮助您掌握这些函数的参数和返回值，以便最终实现熟练应用。这些内建函数包括：

- ◆ `parseInt()`
- ◆ `parseFloat()`
- ◆ `isNaN()`
- ◆ `isFinite()`
- ◆ `encodeURIComponent()`
- ◆ `decodeURIComponent()`
- ◆ `eval()`

黑盒函数



一般来说，当我们调用一个函数时，程序是不需要知道该函数内部的工作细节的。我们可以将其看做一个黑盒子，您只需要给它一些值（作为输入参数），就能获取它输出的返回结果。这种思维适用于任何函数——既包括 JavaScript 中的内建函数，也包括由任何个人或集体所创建的函数。

3.2.1 parseInt()

`parseInt()` 会试图将其收到的任何输入值（通常是字符串）转换成整数类型输出。如果转换失败就返回 `NaN`。

```
>>> parseInt('123')
123
>>> parseInt('abc123')
NaN
>>> parseInt('1abc23')
1
>>> parseInt('123abc')
123
```

除此之外，该函数还有个可选的第二参数：`radix`，它负责设定函数所期望的数字类型——十进制、十六进制、二进制等。在下面的例子中，如果试图以十进制输出字符串 `FF`，结果就会为 `NaN`。而改为十六进制，我们就会得到 `255`。

```
>>> parseInt('FF', 10)
NaN
>>> parseInt('FF', 16)
255
```

再来看一个将字符串转换为十进制和八进制的例子：

```
>>> parseInt('0377', 10)
377
>>> parseInt('0377', 8)
255
```

如果我们在调用 `parseInt()` 时没有指定第二参数，函数就会将其默认为十进制，但有两种情况例外：

- ◆ 如果首参数字符串是 `0x` 开头，第二参数就会被默认为十六进制（也就是默认其为十六进制数）。
- ◆ 如果首参数以 `0` 开头，第二参数就会被默认为八进制（也就是默认其为八进制数）。

```
>>> parseInt('377')
377
>>> parseInt('0377')
```

```
255
```

```
>>> parseInt('0x377')
```

```
887
```

当然，明确指定 `radix` 值总是最安全的。如果您省略了它，尽管 99% 的代码依然能够正常运作（毕竟最常用的还是十进制数），但我们偶尔还是会在调试时发现一些小问题。例如，当我们从日历中读取日期时，对于 **08** 这样的数据，如果不设定 `radix` 参数可能会导致意想不到的结果。

3.2.2 parseFloat()

`parseFloat()` 的功能与 `parseInt()` 基本相同，只不过它支持将输入值转换为十进制数。因此，该函数只有一个参数。

```
>>> parseFloat('123')
```

```
123
```

```
>>> parseFloat('1.23')
```

```
1.23
```

```
>>> parseFloat('1.23abc.00')
```

```
1.23
```

```
>>> parseFloat('a.bc1.23')
```

```
NaN
```

与 `parseInt` 相同，`parseFloat` 在遇到第一个异常字符时也会放弃，无论剩余的那部分字符串是否可用。

```
>>> parseFloat('a123.34')
```

```
NaN
```

```
>>> parseFloat('12a3.34')
```

```
12
```

此外，`parseFloat()` 还可以接受指数形式的数据（这点与 `parseInt()` 不同）。

```
>>> parseFloat('123e-2')
```

```
1.23
```

```
>>> parseFloat('123e2')
```

```
12300
```

```
>>> parseInt('1e10')
```

```
1
```

3.2.3 isNaN()

通过 `isNaN()`，我们可以确定某个输入值是否是一个可以参与算术运算的数字。因而，该函数也可以用来检测 `parseInt()` 和 `parseFloat()` 的调用成功与否。

```
>>> isNaN(NaN)
true
>>> isNaN(123)
false
>>> isNaN(1.23)
false
>>> isNaN(parseInt('abc123'))
true
```

该函数也会始终试图将其所接收的输入转换为数字，例如：

```
>>> isNaN('1.23')
false
>>> isNaN('a1.23')
true
```

`isNaN()` 函数是非常有用的，因为 `NaN` 自己不存在等值的概念，也就是说表达式 `NaN===NaN` 返回的是 **false**，这确实让人觉得有点匪夷所思^①。

3.2.4 isFinite()

`isFinite()` 可以用来检查输入是否是一个既非 `infinity` 也非 `NaN` 的数字。

```
>>> isFinite(Infinity)
false
>>> isFinite(-Infinity)
false
>>> isFinite(12)
true
>>> isFinite(1e308)
true
>>> isFinite(1e309)
false
```

^① 事实上，读者可以将 `NaN` 理解为一个集合，同属于一个集合的值自然未必是等值的。——译者注

关于后两个调用的结果，我们可以回忆第2章中的内容，即 JavaScript 中的最大数字为 $1.7976931348623157e+308$ 。

3.2.5 URI 的编码与反编码

在 URL (Uniform Resource Locator, 统一资源定位符) 或 URI (Uniform Resource Identifier, 统一资源标识符) 中，有一些字符是具有特殊含义的。如果我们想“转义”这些字符，就可以去调用函数 `encodeURIComponent()` 或 `encodeURI()`。前者会返回一个可用的 URL，而后者则会认为我们所传递的仅仅是 URL 的一部分。例如，对于下面这个查询字符串来说，这两个函数所返回的字符编码分别是：

```
>>> var url = 'http://www.packtpub.com/script.php?q=this and that';
>>> encodeURI(url);
"http://www.packtpub.com/script.php?q=this and that"
>>> encodeURIComponent(url);
"http%3A%2F%2Fwww.packtpub.com%2Fscript.php%3Fq%3Dthis%20and%20that"
```

`encodeURI()` 和 `encodeURIComponent()` 分别都有各自对应的反转函数：`decodeURI()` 和 `decodeURIComponent()`。另外，我们有时候还会在一些较旧的代码中看到相似的转义函数和反转义函数，但我们并不赞成使用这些函数来执行相关的操作。

3.2.6 eval()

`eval()` 会将其输入字符串当做 JavaScript 代码来执行。

```
>>> eval('var ii = 2;')
>>> ii
2
```

所以，这里的 `eval('var ii = 2;')` 与表达式 `var ii = 2` 的执行结果是相同的。

尽管，`eval()` 在某些情况下还是很有用的，但如果有选择的话，我们应该尽量避免使用它。毕竟在大多数情况下，我们都应该有更优雅的选择，这些选择通常也更易于编写和维护。对于许多经验丰富的 JavaScript 程序员来说，“Eval is evil” (Eval 是魔鬼) 是一句至理名言。

因为 `eval()` 是这样一种函数：

- ◆ 性能方面——它是一种由函数执行的“动态”代码，显然要比直接执行脚本慢得多。
- ◆ 安全性方面——JavaScript 拥有的功能很强大，但这也意味着很大的不确定性，如

果您对放在 `eval()` 函数中的代码没有太多把握，最好还是不要这样使用。

3.2.7 一点惊喜——`alert()`函数

接下来，让我们来看一个非常常见的函数——`alert()`。该函数不是 JavaScript 核心的一部分（即它没有包括在 ECMA 标准中），而是由宿主环境——浏览器所提供的，是一个用于显示文本的消息对话框。这对于某些调试很有帮助。当然，作为调试工具来说，显然 Firebug 更适合一些。

图 3-1 就是一个 `alert("hello")` 的执行效果图：

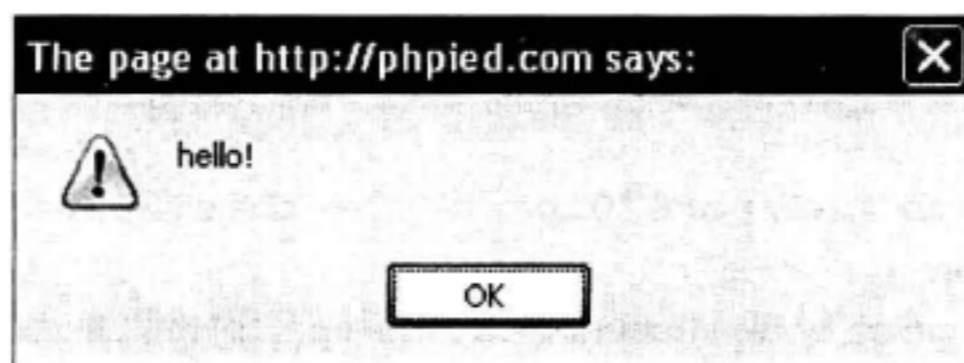


图 3-1

当然，在使用这个函数之前，我们必须明白这样做会阻塞当前的浏览器线程。也就是说，在 `alert()` 的执行窗口关闭之前，当前所有的代码都会暂停执行。因此，对于一个忙碌的 AJAX 应用程序来说，`alert()` 通常不是一个好的选择。

3.3 变量的作用域

这是一个至关重要的问题。特别是当我们从别的语言转向 JavaScript 时，必须要明白一点，即在 JavaScript 中，我们不能为变量定义特定的块作用域，但可以定义其所属的函数域。也就是说，如果变量是在某个函数中定义的，那么它在函数以外的地方是不可见的。而如果该变量是定义在 `if` 或者 `for` 这样的代码块中的，它在代码块之外是可见的。另外，在 JavaScript 中，术语“全局变量”指的是声明在所有函数之外的变量，而与之相对的是“局部变量”，所指的则是在某个函数中定义的变量。其中，函数内的代码可以像访问自己的局部变量那样访问全局变量，反之则不行。

下面来看一个具体示例，请注意两点：

- ◆ 函数 `f()` 可以访问变量 `global`。
- ◆ 在函数 `f()` 以外，变量 `local` 是不存在的。

```

var global = 1;
function f() {
  var local = 2;
  global++;
  return global;
}
>>> f();
2
>>> f();
3
>>> local
local is not defined

```

这里还有一点很重要，即如果我们声明一个变量时没有使用 `var` 语句，该变量就会被默认为全局变量。让我们来看一个具体示例，如图 3-2 所示。

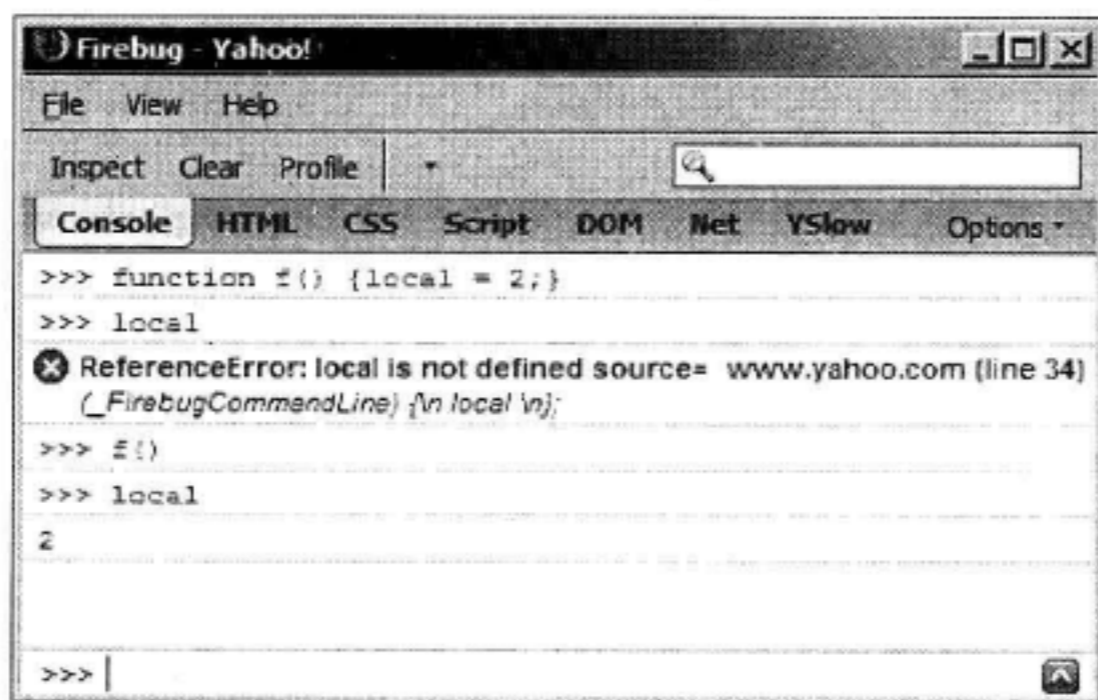


图 3-2

让我们来看看上面究竟发生了些什么。首先，我们在函数 `f()` 中定义了一个变量 `local`。在该函数被调用之前，这个变量是不存在的。该变量会在该函数首次被调用时被创建，并赋予全局作用域。这使得我们可以在该函数以外的地方访问它。

最佳实践



- ◆ 尽量将全局变量的数量降到最低。因为如果有两个人在同一段脚本的不同函数中使用了相同的全局变量，就很容易导致不可预测的结果和难以察觉的 bug。
- ◆ 总是使用 `var` 语句来声明变量。

下面，我们再来看一个有趣的示例，它显示了关于本地和全局作用域的另一个重要问题。


```
var a = 123;
function f() {
    alert(a);
    var a = 1;
    alert(a);
}
f();
```

您可能会想当然地认为 `alert()` 第一次显示的是 123（也就是全局变量 `a` 的值），而第二次显示的是 1（即局部变量 `a`）。但事实并非如此，第一个 `alert()` 实际上显示的是“**undefined**”，这是因为函数域始终优先于全局域，所以局部变量 `a` 会覆盖掉所有与它同名的全局变量，尽管在 `alert()` 第一次被调用时，`a` 还没有被正式定义（即该值为 **undefined**），但该变量本身已经存在于本地空间了。

3.4 函数也是数据

对于 JavaScript 来说，有一个概念对于我们日后的学习至关重要——即函数是一种数据类型。也就是说，下面两种函数定义在本质上是相同的。

```
function f(){return 1;}
var f = function(){return 1;}
```

其中，第二种定义方式通常被叫做函数标识记法（`function literal notation`）。

如果我们对函数变量调用 `typeof`，操作符返回的字符串将会是“`function`”。

```
>>> function f(){return 1;}
>>> typeof f
"function"
```

所以，JavaScript 中的函数是一种数据，只不过这种特殊的数据类型有两个重要的特性：

- ◆ 它们所包含的是代码。
- ◆ 它们是可执行的（或者说是可调用的）。

和我们之前看到的一样，要调用某个函数，只需要在它的名字后面加一对括号即可。我们再来看一个示例，下面这段代码工作与函数的定义方式无关，它演示的是如何像变量

那样使用函数——也就是说，我们可以将它拷贝给不同的变量，甚至删除。

```
>>> var sum = function(a, b) {return a + b;}
>>> var add = sum;
>>> delete sum
true
>>> typeof sum;
"undefined"
>>> typeof add;
"function"
>>> add(1, 2);
3
```

由于函数也是赋值给变量的一种数据，所以函数的命名规则与一般变量相同——即函数名不能以数字开头，可以由任意的字母、数字和下划线组合而成。

3.4.1 匿名函数

在 JavaScript 中，程序中总是会有一些随处都能看见的数据片段。例如我们的代码中可能有很多这样的片段：

```
>>> "test"; [1,2,3]; undefined; null; 1;
```

这段代码看上去有些奇怪，因为它实际上什么也做不了，但它是符合语法的，并不会引发任何错误。我们可以将这样的片段看做代码中的一段匿名数据——说它匿名是因为这段数据既没有赋值给某个变量，也没有被赋予任何名字。

而且我们已经知道，函数在本质上与其他变量并无区别，因此它也可以在没有名字的情况下被使用。例如：

```
>>> function(a){return a;}
```

现在，为了让散落在代码中的那些匿名数据片段不会真的一无是处，我们可以将它们设置成函数，这样它们就有了两种优雅的用法：

- ◆ 我们可以将匿名函数作为参数传递给其他函数，这样，接收方函数就能利用我们所传递的函数来完成某些事情。
- ◆ 我们可以定义某个匿名函数来执行某些一次性任务。

接下来，我们来看两个具体的应用示例，通过其中的细节来进一步了解匿名函数。

3.4.2 回调函数

既然函数可以像其他数据那样赋值给某个变量，可以被定义、删除、拷贝，那为什么就不能被当成参数传递给其他函数呢？

下面的示例中，我们定义了一个以两个函数为参数的函数。该函数会分别执行这两个参数函数，并返回它们的返回值之和。

```
function invoke_and_add(a, b) {
  return a() + b();
}
```

现在让我们来简单定义一下这两个参与加法运算的函数，它们只是单纯地返回一个硬编码值：

```
function one() {
  return 1;
}
function two() {
  return 2;
}
```

下面，我们只需将这两个函数传递给目标函数 `invoke_and_add()`，就可以得到执行结果了：

```
>>> invoke_and_add(one, two);
3
```

事实上，我们也可以直接用匿名函数来代替 `one()`和 `two()`，以作为目标函数的参数，例如：

```
invoke_and_add(function(){return 1;}, function(){return 2;})
```

当我们将函数 A 传递给函数 B，并由 B 来执行 A 时，A 就成了一个回调函数（callback functions）。如果这时 A 还是一个无名函数，我们就称它为匿名回调函数。

那么，应该什么时候使用回调函数呢？下面就让我们通过几个应用实例来示范一下回调函数的优势。

- ◆ 它可以让我们在不命名的情况下传递函数（这意味着可以节省全局变量）。

- ◆ 我们可以将一个函数调用操作委托给另一个函数（这意味着可以节省一些代码编写工作）。
- ◆ 它们也有助于提升性能。

3.4.3 回调示例

在编程过程中，我们通常需要将一个函数的返回值传递给另一个函数。在下面的例子中，我们定义了两个函数：第一个是 `multiplyByTwo()`，该函数会通过一个循环将其所接受的三个参数分别乘以 2，并以数组的形式返回结果；第二个函数 `addOne()` 只接受一个值，然后将它加 1 并返回即可。

```
function multiplyByTwo(a, b, c) {
  var i, ar = [];
  for(i = 0; i < 3; i++) {
    ar[i] = arguments[i] * 2;
  }
  return ar;
}
function addOne(a) {
  return a + 1;
}
```

现在，我们来测试一下这两个函数：

```
>>> multiplyByTwo(1, 2, 3);
[2, 4, 6]
>>> addOne(100)
101
```

接下来，我们要实现这三个元素在两个函数之间的传递，这需要定义一个用于存储元素的数组。我们先从 `multiplyByTwo()` 的调用开始：

```
>>> var myarr = [];
>>> myarr = multiplyByTwo(10, 20, 30);
[20, 40, 60]
```

然后，用循环遍历每个元素，并将它们分别传递给 `addOne()`。

```
>>> for (var i = 0; i < 3; i++) {myarr[i] = addOne(myarr[i]);}
>>> myarr
[21, 41, 61]
```

如您所见，这段代码可以工作，但是显然还有一定的改善空间。特别是这里使用了两个循环，如果数据量很大或循环操作很复杂的话，开销一定不小。因此，我们需要将它们合二为一。这就需要对 `multiplyByTwo()` 函数做一些改动，使其接受一个回调函数，并在每次迭代操作中调用它。具体如下：

```
function multiplyByTwo(a, b, c, callback) {
  var i, ar = [];
  for(i = 0; i < 3; i++) {
    ar[i] = callback(arguments[i] * 2);
  }
  return ar;
}
```

函数修改完成之后，之前的工作只需要一次函数调用就够了，我们只需像下面这样将初始值和回调函数传递给它即可：

```
>>> myarr = multiplyByTwo(1, 2, 3, addOne);
[3, 5, 7]
```

我们还可以用匿名函数来代替 `addOne()`，这样做可以节省一个额外的全局变量。

```
>>> myarr = multiplyByTwo(1, 2, 3, function(a){return a + 1});
[3, 5, 7]
```

而且，使用匿名函数也更易于随时根据需求调整代码。例如：

```
>>> myarr = multiplyByTwo(1, 2, 3, function(a){return a + 2});
[4, 6, 8]
```

3.4.4 自调函数

目前我们已经讨论了匿名函数在回调方面的应用。接下来，我们来看匿名函数的另一个应用示例——这种函数可以在定义后自行调用。比如：

```
(
  function() {
    alert('boo');
  }
)()
```

这种语法看上去有点吓人，但其实很简单——我们只需将匿名函数的定义放进一对括

号中，然后外面再紧跟一对括号即可。其中，第二对括号起到的是“立即调用”的作用，同时它也是我们向匿名函数传递参数的地方。

```
(
  function(name) {
    alert('Hello ' + name + '!');
  }
)('dude')
```

使用自调匿名函数的好处在于这样不会产生任何全局变量。当然，缺点是这样的函数是无法重复执行的（除非您将它放在某个循环或其他函数中）。这也使得匿名自调函数最适合于执行一些一次性的或初始化的任务。

3.4.5 内部（私有）函数

想必我们都记得，函数与其他类型的值本质上是一样的，因此，没有什么理由可以阻止我们在一个函数内部定义另一个函数。

```
function a(param) {
  function b(theinput) {
    return theinput * 2;
  };
  return 'The result is ' + b(param);
};
```

我们也可以改用函数标识记法来写这段代码：

```
var a = function(param) {
  var b = function(theinput) {
    return theinput * 2;
  };
  return 'The result is ' + b(param);
};
```

当我们调用全局函数 `a()` 时，本地函数 `b()` 也会在其内部被调用。由于 `b()` 是本地函数，它在 `a()` 以外的地方是不可见的，所以我们也能将它称之为私有函数。

```
>>> a(2);
"The result is 4"
>>> a(8);
"The result is 16"
```

```
>>> b(2);  
b is not defined
```

使用私有函数的好处主要有以下几点：

- ◆ 有助于我们确保全局名字空间的纯净性（这意味着命名冲突的机会很小）。
- ◆ 私有性——这使我们可以选择只将一些必要的函数暴露给“外部世界”，并保留属于自己的函数，使它们不为该应用程序的其他部分所用。

3.4.6 返回函数的函数

正如之前所提到的，函数始终都会有一个返回值，即便不是显式返回，它也会隐式返回一个 `undefined`。既然函数能返回一个唯一值，那么这个值就也有可能是另一个函数。例如：

```
function a() {  
    alert('A!');  
    return function(){  
        alert('B!');  
    };  
}
```

在这个例子中，函数 `a()` 会在执行它的工作（说“A!”）之后返回另一个函数 `b()`。而 `b()` 又会去执行另外一些事情（说“B!”）。我们只需将该返回值赋值给某个变量，然后就可以像使用一般函数那样调用它了。

```
>>> var newFunc = a();  
>>> newFunc();
```

如您所见，上面第一行执行的是 `alert("A!")`，第二行才是 `alert("B!")`。

如果您想让返回的函数立即执行，也可以不用将它赋值给变量，直接在该调用后面再加一对括号即可，效果是一样的。

```
>>> a()();
```

3.4.7 能重写自己的函数

由于一个函数可以返回另一个函数，因此我们可以用新的函数来覆盖旧的。例如在之

前的例子中，我们也可以通过 `a()` 的返回值来重写 `a()` 函数自己：

```
>>> a = a();
```

当前这句依然只会执行 `alert("A!")`，但如果我们再次调用 `a()`，它就会执行 `alert("B!")` 了。

这对于要执行某些一次性初始化工作的函数来说会非常有用。这样一来，该函数可以在第一次被调用后重写自己，从而避免了每次调用时重复一些不必要的操作。

在上面的例子中，我们是外面来重定义该函数的——即我们将函数返回值赋值给函数本身。但我们也可以让函数从内部重写自己。例如：

```
function a() {
  alert('A!');
  a = function(){
    alert('B!');
  };
}
```

这样一来，当我们第一次调用该函数时：

- ◆ `alert("A!")` 将会被执行（可以视之为一次性的准备操作）。
- ◆ 全局变量 `a` 将会被重定义，并被赋予新的函数。

而如果该函数再被调用的话，被执行的就将是 `alert("B!")` 了。

下面，我们来看一个组合型的应用示例，其中有些技术我们将会在本章最后几节中讨论。

```
var a = function() {
  function someSetup() {
    var setup = 'done';
  }
  function actualWork() {
    alert('Worky-worky');
  }
  someSetup();
  return actualWork;
}();
```

在这个例子中：

- ◆ 我们使用了私有函数——`someSetup()` 和 `actualWork()`。
- ◆ 我们也使用了自调函数——函数 `a()` 的定义后面有一对括号，因此它会执行自行调用。
- ◆ 当该函数第一次被调用时，它会调用 `someSetup()`，并返回函数变量 `actualWork` 的引用。请注意，返回值中是不带括号的，因此该结果仅仅是一个函数引用，并不会产生函数调用。
- ◆ 由于这里的执行语句是以 `var a = ...` 开头的，因而该自调函数所返回的值会重新赋值给 `a`。

如果我们想测试一下自己对上述内容的理解，可以尝试回答一下这个问题：上面的代码在以下情景中分别会 `alert()` 什么内容？

- ◆ 当它最初被载入。
- ◆ 之后再次调用 `a()` 时。

这项技术对于某些浏览器相关的操作会相当有用。因为在不同浏览器中，实现相同任务的方法可能是不同的，我们都知道浏览器的特性不可能因为函数调用而发生任何改变，因此，最好的选择就是让函数根据其当前所在的浏览器来重定义自己。这就是所谓的“浏览器特性探测”技术，关于这方面的应用示例，我们会在本书后面的章节中给予展示。

3.5 闭包

在本章剩下的部分中，我们来谈谈闭包（正好用来关闭本章^①）。最初接触闭包可能会有一些困难，如果您在首次阅读时没能“抓住”重点，大可不必感到灰心丧气。后面的章节中还有大量的实例可以供您理解它们，所以，如果您觉得现在没有完全理解，可以在以后涉及相关话题时再回过头来看看这部分内容。

在我们讨论闭包之前，需要先复习一下 JavaScript 中作用域的概念，并对它再进行一些扩展。

3.5.1 作用域链

如您所知，JavaScript 与很多程序设计语言不同，它不存在大括号级的作用域，但它有函数作用域，也就是说，在函数内定义的变量在函数外是不可见的。但如果该变量是在某

^① 这里作者玩了一把双关语，因为闭包（closures）这个词也可以理解为“关闭”。——译者注

个代码块中定义的（如在某个 `if` 或 `for` 语句中），它在代码块外是可见的。

```
>>> var a = 1; function f(){var b = 1; return a;}
>>> f();
1
>>> b
b is not defined
```

在这里，变量 `a` 是属于全局域的，而变量 `b` 的作用域就在函数 `f()` 内了。所以：

- ◆ 在 `f()` 内，`a` 和 `b` 都是可见的。
- ◆ 在 `f()` 外，`a` 是可见的，`b` 则不可见。

另外，如果我们在函数 `f()` 中定义了另一个函数 `n()`，那么，在 `n()` 中可以访问的变量既可以来自它自身的作用域，也可以来自其“父级”作用域。这就形成了一条作用域链（`scope chain`），该链的长度（或深度）则取决于我们的需要。

```
var a = 1;
function f(){
  var b = 1;
  function n() {
    var c = 3;
  }
}
```

3.5.2 词法作用域

在 JavaScript 中，每个函数都有一个自己的词法作用域。也就是说，每个函数在被定义时（而非执行时）都会创建一个属于自己的环境（即作用域）。让我们来看一个具体示例：

```
>>> function f1(){var a = 1; f2();}
>>> function f2(){return a;}
>>> f1();
a is not defined
```

在上面的代码中，我们在函数 `f1()` 中调用了函数 `f2()`。由于局部变量 `a` 也在 `f1()` 中，所以人们可能认为 `f2()` 是可以访问 `a` 的，但事实并非如此。因为当 `f2()` 被定义时（不是执行时），变量 `a` 是不可见的。和 `f1()` 一样，它那时候只能访问自身作用域和全局作用域中的内容。也就是说，这里的 `f1()`、`f2()` 之间不存在共享的词法作用域。

尽管，当函数被定义时，它会“记录”自身所在的环境和相关的作用域链。但这并不

意味着函数也会对其作用域中的每个变量进行记录，正好相反——我们可以在函数中对变量执行添加、移除和更新等操作，但函数只会看到该变量的最终状态。例如，如果我们在上面的例子中再增加一个全局变量 `a`，`f2()` 就可以访问它了。因为 `f2()` 知道访问全局环境的路径，它可以访问该环境中的所有东西。另外还需要提醒的是，即使 `f2()` 还没有被定义，我们也可以在 `f1()` 的定义中包含对 `f2()` 的调用。因为对于 `f1()` 而言，在其所知的作用域中的任何东西都是可用的。

```
>>> function f1(){var a = 1; return f2();}
>>> function f2(){return a;}
>>> f1();
a is not defined
>>> var a = 5;
>>> f1();
5
>>> a = 55;
>>> f1();
55
>>> delete a;
true
>>> f1();
a is not defined
```

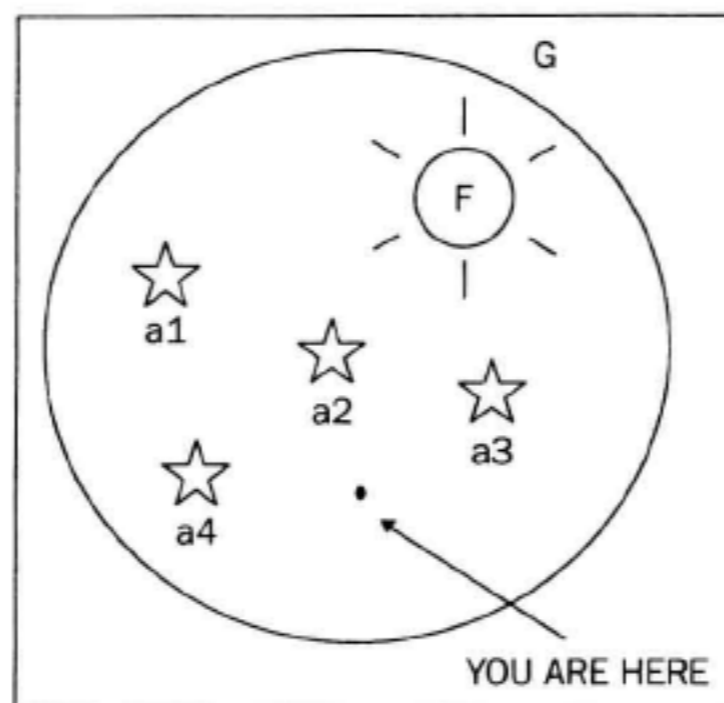
这就赋予了 JavaScript 极大的灵活性——我们可以通过添加/删除操作来实现变量的重复添加，并且完全不会影响程序的运行。这意味着我们可以在上述实验的过程中删除 `f2` 函数，并重新给它定义一个完全不同的执行体，而最终 `f1()` 依然能够正常工作。因为在这里，`f1()` 只需知道它应该如何访问自身作用域即可，不需要知道该作用域在什么时候发生了什么事。例如，我们可以继续上面的例子：

```
>>> delete f2;
true
>>> f1()
f2 is not defined
>>> var f2 = function(){return a * 2;}
>>> var a = 5;
5
>>> f1();
10
```

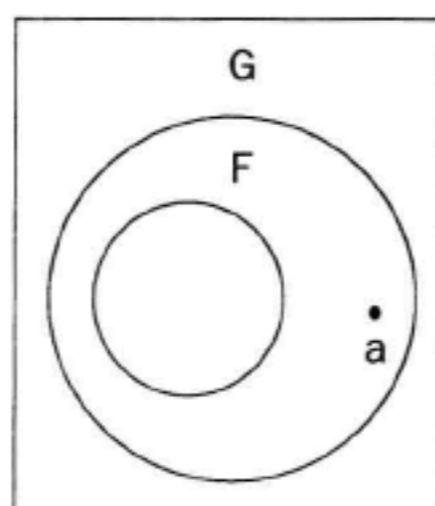
3.5.3 利用闭包突破作用域链

现在，让我们先通过图示的方式来介绍一下闭包的概念。

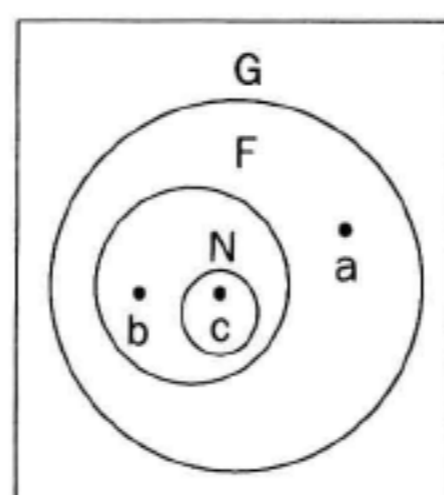
首先是全局作用域，我们可以将其看做包含一切的宇宙。



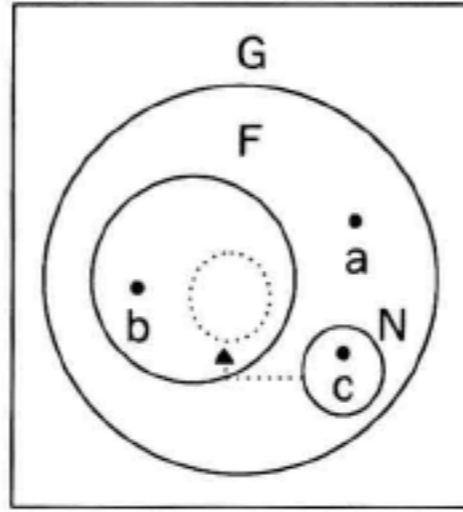
其中可以包含各种变量（如 a ）和函数（如 F ）。



每个函数也都会拥有一块属于自己的私用空间，用以存储一些别的变量（和函数）。所以，我们最终可以把示意图画成这样：



在上图中，如果我们在 a 点，那么就位于全局空间中。而如果是在 b 点，我们就在函数 F 的空间里，在这里我们既可以访问全局空间，也可以访问 F 空间。如果我们在 c 点，那就位于函数 N 中，这里我们可以访问的空间包括全局空间、 F 空间和 N 空间。其中， a 和 b 之间是不连通的，因为 b 在 F 以外是不可见的。但如果愿意的话，我们是可以将 c 点和 b 点连通起来的，或者说将 N 与 b 连通起来。当我们将 N 的空间扩展到 F 以外，并止步于全局空间以内时，就产生了一件有趣的东西——闭包。



知道接下来会发生什么吗？N 将会和 a 一样置身于全局空间。而且由于函数还记得它
在被定义时所设定的环境，因此它依然可以访问 F 空间并使用 b。这很有趣，因为现在 N
和 a 同处于一个空间，但 N 可以访问 b，而 a 不能。

那么，N 究竟是如何突破作用域链的呢？我们只需要将它们升级为全局变量（不使用
var 语句）或通过 F 传递（或返回）给全局空间即可。下面，我们来看看具体是怎么做的。

3.5.3.1 闭包#1

下面，我们先来看一个函数：

```
function f(){
  var b = "b";
  return function(){
    return b;
  }
}
```

这个函数含有一个局部变量 b，它在全局空间里是不可见的。

```
>>> b
b is not defined
```

接下来，我们来看一下 f() 的返回值。这是另一个函数，我们可以把它看做上图中的
函数 N。该函数有自己的私用空间，同时也可以访问 f() 的空间和全局空间，所以 b 对它
来说是可见的。因为 f() 是可以在全局空间中被调用的（它是一个全局函数），所以我们可以
将它的返回值赋值给另一个全局变量，从而生成一个可以访问 f() 私有空间的新全局函
数。

```
>>> var n = f();
>>> n();
"b"
```

3.5.3.2 闭包#2

下面这个例子的最终结果与之前相同，但在实现方法上存在着一些细微的不同。在这里 `f()` 不再返回函数了，而是直接在函数体内创建一个新的全局函数。

首先，我们需要声明一个全局函数的占位符。尽管这种占位符不是必须的，但最好还是声明一下，然后，我们就可以将函数 `f()` 定义如下：

```
var n;
function f(){
  var b = "b";
  n = function(){
    return b;
  }
}
```

现在，让我们来看看 `f()` 被调用时会发生什么：

```
>>> f();
```

我们在 `f()` 中定义了一个新的函数，并且没有在这里使用 `var` 语句，因此它应该是属于全局的。由于 `n()` 是在 `f()` 内部定义的，它可以访问 `f()` 的作用域，所以即使该函数后来升级成了全局函数，但它依然可以保留对 `f()` 作用域的访问权。

```
>>> n();
"b"
```

3.5.3.3 相关定义与闭包#3

根据目前的讨论，我们可以说，如果一个函数需要在其父级函数返回之后留住对父级作用域的链接的话^①，就必须为此建立一个闭包。

而由于函数通常都会将自身的参数视为局部变量。因此我们创建返回函数时，也可以令其返回父级函数的参数。例如：

```
function f(arg) {
  var n = function(){
    return arg;
  }
}
```

^① 如上例子所示，`f` 是 `n` 的父级函数，在 `f` 返回之后，`n` 依然可以访问 `f` 中的局部变量 `b`。——译者注

```
};  
arg++;  
return n;  
}
```

然后我们可以这样调用函数：

```
>>> var m = f(123);  
>>> m();  
124
```

请注意，当我们的返回函数被调用时^①，`arg++`已经执行过一次递增操作了。所以 `m()` 返回的是更新后的值。由此我们可以看出，函数所绑定的是作用域本身，而不是该作用域中的变量或变量当前所返回的值。

3.5.3.4 循环中的闭包

由这种闭包所导致的 bug 往往很难被发现，因为它们总是表面上看起来一切正常。

让我们来看一个三次性的循环操作，它在每次迭代中都会创建一个返回当前循环序号的新函数。该新函数会被添加到一个数组中，并最终返回。具体代码如下：

```
function f() {  
  var a = [];  
  var i;  
  for(i = 0; i < 3; i++) {  
    a[i] = function(){  
      return i;  
    }  
  }  
  return a;  
}
```

下面，我们来运行一下函数，并将结果赋值给数组 `a`。

```
>>> var a = f();
```

现在，我们拥有了一个包含三个函数的数组。您可以通过在每个数组元素后面加一对括号来调用它们。按通常的估计，它们应该会依照循环顺序分别输出 0、1 和 2，下面就让

^① `n` 被赋值时函数并没有被调用，调用是在 `n` 被求值，也就是执行 `return n;` 语句时被调用的。——译者注

我们来试试：

```
>>> a[0]()
3
>>> a[1]()
3
>>> a[2]()
3
```

事实上不是这样的。这是怎么回事？原来我们在这里创建三个闭包，它们都指向了一个共同的局部变量 `i`。但是，闭包并不会记录它们的值，它们所拥有的只是一个 `i` 的连接（即引用），因此只能返回 `i` 的当前值。由于循环结束时 `i` 的值为 **3**，所以这三个函数都指向了这一共同值。

（为什么结果是 3 不是 2 呢？这也是一个值得思考的问题，它能帮助您更好地理解 `for` 循环。）

那么，应该如何纠正这种行为呢？显然我们需要三个不同的变量。或许换一种闭包形式不失为一种优雅的解决方案：

```
function f() {
  var a = [];
  var i;
  for(i = 0; i < 3; i++) {
    a[i] = (function(x) {
      return function() {
        return x;
      }
    })(i);
  }
  return a;
}
```

这样就能获得我们预期的结果了：

```
>>> var a = f();
>>> a[0]();
0
>>> a[1]();
1
>>> a[2]();
2
```


在这里，我们不再直接创建一个返回 `i` 的函数了，而是将 `i` 传递给了一个自调函数。在该函数中，`i` 就被赋值给了局部变量 `x`，这样一来，每次迭代中的 `x` 就会拥有各自不同的值了。

或者，我们也可以定义一个“正常点”（不使用自调函数）的内部函数来实现相同的功能。关键是在每次迭代操作中，我们要在中间函数内将 `i` 的值“本地化”。

```
function f() {
  function makeClosure(x) {
    return function() {
      return x;
    }
  }
  var a = [];
  var i;
  for(i = 0; i < 3; i++) {
    a[i] = makeClosure(i);
  }
  return a;
}
```

3.5.4 Getter 与 Setter

接下来，让我们再来看两个关于闭包的应用示例。首先是创建 `getter` 和 `setter`。假设现在有一个属于特殊区间的变量，我们不想将它暴露给外部。因为这样一来，其他部分的代码就有修改它的可能，所以我们需要将它保护在相关函数的内部，然后再提供两个额外的函数——一个用于访问变量，另一个用于设置变量。并在设置函数中引入某些验证措施，以便在赋值之前给予该变量一定的保护（当然，为了能让下面的示例简洁一些，我们忽略掉了验证相关的部分）。

我们需要将 `getter` 和 `setter` 这两个函数放在一个共同的函数中，并在该函数中定义 `secret` 变量，这使得两个函数能够共享同一作用域。具体代码如下：

```
var getValue, setValue;
(function() {
  var secret = 0;
  getValue = function() {
    return secret;
  };
  setValue = function(v) {
```

```

        secret = v;
    };
})();

```

在这里，所有一切都是通过一个匿名自调函数来实现的，我们在其中定义了全局函数 `setValue()` 和 `getValue()`，并以此来确保局部变量 `secret` 的不可直接访问性。

```

>>> getValue()
0
>>> setValue(123)
>>> getValue()
123

```

3.5.5 迭代器

在最后一个关于闭包应用的示例（这也是本章的最后一个示例）中，我们将向您展示闭包在实现迭代器方面的功能。

通常情况下，我们都知道如何用循环来遍历一个简单的数组，但是有时候我们需要面对更为复杂的数据结构，它们通常会有着与数组截然不同的序列规则。这时候就需要将一些“谁是下一个”的复杂逻辑封装成易于使用的 `next()` 函数，然后，我们只需要简单地调用 `next()` 就能实现对于相关的遍历操作了。在下面这个例子中，我们将依然通过简单数组，而不是复杂的数据结构来说明问题。

下面是一个接受数组输入的初始化函数，我们在其中定义了一个私有指针，该指针会始终指向数组中的下一个元素。

```

function setup(x) {
    var i = 0;
    return function() {
        return x[i++];
    };
}

```

现在，我们只需用一组数据来调用一下 `setup()`，就会创建出我们所需要的 `next()` 函数，具体如下：

```

>>> var next = setup(['a', 'b', 'c']);

```

这是一种既简单又好玩的循环形式：我们只需重复调用一个函数，就可以不停地获取下一个元素。

```
>>> next();  
"a"  
>>> next();  
"b"  
>>> next();  
"c"
```

3.6 本章小结

现在，我们已经完成了对于 JavaScript 函数的基本概念介绍，为今后学习 JavaScript 的面向对象特性，以及相关的现代编程模式打下了一定的基础。在这之前，我们一直在刻意回避有关面向对象特性的内容，但往后，本书将带您深入这些更为有趣的内容。下面，让我们再来花一点时间回顾一下本章所讨论的内容：

- ◆ 如何定义和调用函数的基础知识
- ◆ 函数的参数及其灵活性
- ◆ 内置函数——包括 `parseInt()`、`parseFloat()`、`isNaN()`、`isFinite()`、`eval()` 以及对 URL 执行编码、反编码操作的四个相关函数。
- ◆ JavaScript 变量的作用域——它们没有大括号级作用域，只有函数级作用域。并且，每个函数都有自己的词法作用域和作用域链。
- ◆ 函数也是一种数据——即函数可以跟其他数据一样被赋值给一个变量，我们可以据此实现大量有趣的应用。例如
 - 私有函数和私有变量
 - 匿名函数
 - 回调函数
 - 自调函数
 - 能重写自身的函数
 - 闭包

3.7 练习题

1. 编写一个将十六进制值转换为颜色的函数，以蓝色为例，`#0000FF` 应被表示成

“`rgb(0,0,255)`”的形式。然后将函数命名为 `getRGB()`，并用以下代码进行测试。

```
>>> var a = getRGB("#00FF00");
>>> a;
"rgb(0, 255, 0)"
```

2. 如果在控制台中执行以下各行，分别会输出什么内容？

```
>>> parseInt(1e1)
>>> parseInt('1e1')
>>> parseFloat('1e1')
>>> isFinite(0/10)
>>> isFinite(20/0)
>>> isNaN(parseInt(NaN));
```

3. 下面代码中，`alert()`弹出的内容会是什么？

```
var a = 1;
function f() {
  var a = 2;
  function n() {
    alert(a);
  }
  n();
}
f();
```

4. 以下所有示例都会弹出“**Boo!**”警告框，您能解释其中原因吗？

```
4.1. var f = alert;
    eval('f("Boo!")');
4.2. var e;
    var f = alert;
    eval('e=f')('Boo!');
4.3. (
    function() {
      return alert;
    }
  )()('Boo!');
```

第 4 章

对象

到目前为止，我们已经了解了 JavaScript 中的基本数据类型、数组及函数，现在是时候学习本书最重要的一部分内容——对象了。在这一章中，我们将介绍以下内容：

- ◆ 如何创建并使用对象。
- ◆ 什么是构造器函数。
- ◆ JavaScript 中的内置对象及其运用。

4.1 从数组到对象

正如我们在第 2 章中所介绍的，数组实际上就是一组值的列表。该表中的每一个值都有自己的索引值（即数字键值），索引序列从 0 开始，依次递增。例如：

```
>>>> var myarr = ['red', 'blue', 'yellow', 'purple'];
>>> myarr;
["red", "blue", "yellow", "purple"]
>>> myarr[0]
"red"
>>> myarr[3]
"purple"
```

如果我们将索引键单独排成一列，再把对应的值排成另一列，就会列出这样一个键/值表，如表 4-1 所示。

表 4-1

索引键	对应值
0	Red

(续表)

索引键	对应值
1	Blue
2	Yellow
3	Purple

事实上，对象的情况跟数组很相似，唯一的不同是它的键值类型是自定义的。也就是说，我们的索引方式不再局限于数字了，而可以使用一些更为人性化的键值，比如 `first_name`、`age` 等。

下面，让我们通过一个简单的示例来看看对象是由哪几部分组成的：

```
var hero = {
  breed: 'Turtle',
  occupation: 'Ninja'
};
```

正如我们所见：

- ◆ 这里有一个用于表示该对象的变量名 `hero`。
- ◆ 与定义数组时所用的中括号 `[]` 不同，对象使用的是大括号 `{}`。
- ◆ 括号中用逗号分割着的是组成该对象的元素（通常被称之为属性）。
- ◆ 键/值对之间用冒号分割，例如，`key: value`。

有时候，我们还可以在键值（属性名）上面加一对引号，例如，下面三行代码所定义的内容是完全相同的：

```
var o = {prop: 1};
var o = {"prop": 1};
var o = {'prop': 1};
```

通常情况下，我们不建议您属性名上面加引号（这也能减少一些输入），但在以下这些情境中，引号是必须的：

- ◆ 如果属性名是 JavaScript 中的保留字之一的话（具体可参考附录 A）。
- ◆ 如果属性名中包含空格或其他特殊字符的话（包括任何除字母、数字及下划线以外的字符）。

- ◆ 如果属性名以数字开头的话。

总而言之，如果我们所选的属性名不符合 JavaScript 中的变量命名规则，就必须对其施加一对引号。

下面，让我们来看一个怪异的对象定义：

```
var o = {
  something: 1,
  'yes or no': 'yes',
  '!@#$$%^&*': true
};
```

但该对象是合法的，因为我们在它的第二和第三个属性名上加了引号，否则一定会出错。

在本章稍后的内容中，我们还会介绍除了[]和{}以外的定义数组和对象的方法。但我们首先要明白当前这种方法的术语名词：用[]定义数组的方法我们称之为数组文本标识法(array literal notation)，而同样的，用大括号{}定义对象的方法就叫做对象文本标识法(object literal notation)。

4.1.1 元素、属性、方法

说到数组的时候，我们常说其中包含的是元素。而当我们说对象时，就会说其中包含的是属性。实际上对于 JavaScript 来说，它们并没有本质的不同，只是在技术术语上的表达习惯有所不同罢了。这也是它区别于其他程序设计语言的地方。

另外，对象的属性也可以是一个函数，因为函数本身也是一种数据，在这种情况下，我们会称该属性为方法。例如：

```
var dog = {
  name: 'Benji',
  talk: function(){
    alert('Woof, woof!');
  }
};
```

当然，我们也可以像下面这样，在数组中存储一些函数元素并在需要时调用它们，但这在实践中并不多见。

```
>>> var a = [];
```

```
>>> a[0] = function(what){alert(what);};  
>>> a[0]('Boo!');
```

4.1.2 哈希表、关联型数组

在一些程序设计语言中，通常都会存在着两种不同的数组形式：

- ◆ 一般性数组，也叫做索引型数组或者枚举型数组（通常以数字为键值）。
- ◆ 关联型数组，也叫做哈希表（通常以字符串为键值）。

在 JavaScript 中，我们会用数组来表示索引型数组，而用对象来表示关联型数组。因此，如果我们想在 JavaScript 中使用哈希表，就必须要用到对象。

4.1.3 访问对象属性

通常情况下，我们可以通过以下两种方式来访问对象的属性：

- ◆ 一种是中括号表示法，例如 `hero['occupation']`。
- ◆ 另一种则是点号表示法，例如 `hero.occupation`。

尽管相对而言，点号表示法更易于读写，但也不是总能适用的。其中的规则与属性命名原则相同，即如果我们所访问的属性没有一个合法的名字，它就不能通过点号表示法来访问。

让我们来看一个具体的对象：

```
var hero = {  
  breed: 'Turtle',  
  occupation: 'Ninja'  
};
```

下面我们用点号表示法来访问属性：

```
>>> hero.breed;  
"Turtle"
```

再用中括号表示法来访问属性：

```
>>> hero['occupation'];  
"Ninja"
```

如果我们访问的属性不存在，代码就会返回 `undefined`。


```
>>> 'Hair color is ' + hero.hair_color;
"Hair color is undefined"
```

另外，由于对象中可以包含任何类型的数据，自然也包括其他对象：

```
var book = {
  name: 'Catch-22',
  published: 1961,
  author: {
    firstname: 'Joseph',
    lastname: 'Heller'
  }
};
```

在这里，如果我们想访问 book 对象的 author 属性对象的 firstname 属性，就需要这样：

```
>>> book.author.firstname
"Joseph"
```

当然，也可以连续使用中括号表示法，例如：

```
>>> book['author']['lastname']
"Heller"
```

甚至可以混合使用这两种表示法，例如：

```
>>> book.author['lastname']
"Heller"
>>> book['author'].lastname
"Heller"
```

另外还有一种情况，即如果我们要访问的属性名是不确定的，就必须使用中括号表示法了，它允许我们在运行时通过变量来实现相关属性的动态存取。

```
>>> var key = 'firstname';
>>> book.author[key];
"Joseph"
```

4.1.4 调用对象方法

由于对象方法实际上只是一个函数类型属性，因此它们的访问方式与属性完全相同，

即用点号表示法或中括号表示法均可。而其调用（请求）方式也与其他函数相同，在指定的方法名后加一对括号即可。例如下面的 say 方法：

```
var hero = {
  breed: 'Turtle',
  occupation: 'Ninja',
  say: function() {
    return 'I am ' + hero.occupation;
  }
}
>>> hero.say();
"I am Ninja"
```

如果调用方法时需要传递一些参数，做法也和一般函数一样。例如：

```
>>> hero.say('a', 'b', 'c');
```

另外，由于我们可以像访问数组一样用中括号来访问属性，因此这意味着我们同样可以用中括号来调用方法，尽管这种做法在实践中并不常见。

```
>>> hero['say']();
```



最佳做法提示：尽量别使用引号

1. 尽量使用点号表示法来访问对象的方法与属性。
2. 不要在对象中使用带引号的属性标识。

4.1.5 修改属性与方法

由于 JavaScript 是一种动态语言，所以它允许我们随时对现存对象的属性和方法进行修改。其中自然也包括添加与删除属性。因此，我们也可以先创建一个空对象，稍后再为它添加属性。下面，让我们来看看具体是怎么做的：

首先创建一个空对象：

```
>>> var hero = {};
```

这时候，如果我们访问一个不存在的属性，就会：

```
>>> typeof hero.breed
"undefined"
```

现在，我们来为该对象添加一些属性和方法：

```
>>> hero.breed = 'turtle';
>>> hero.name = 'Leonardo';
>>> hero.sayName = function() {return hero.name;};
```

然后调用该方法:

```
>>> hero.sayName();
"Leonardo"
```

接下来, 我们删除一个属性:

```
>>> delete hero.name;
true
```

然后再调用该方法, 它就不能正常工作了:

```
>>> hero.sayName();
reference to undefined property hero.name
```

4.1.6 使用 this 值

在之前的示例中, 方法 `sayName()` 是直接通过 `hero.name` 来访问 `hero` 对象的 `name` 属性的。而事实上, 当我们处于某个对象方法内部时, 还可以用另一种方法来访问同一对象的属性, 即该对象的特殊值 `this`。例如:

```
var hero = {
  name: 'Rafaelo',
  sayName: function() {
    return this.name;
  }
}
>>> hero.sayName();
"Rafaelo"
```

也就是说, 当我们引用 `this` 值时, 实际上所引用的就是“这个对象”或者“当前对象”。

4.1.7 构造器函数

另外, 我们还可以通过构造器函数 (constructor function) 的方式来创建对象。下面来

看一个例子：

```
function Hero() {
  this.occupation = 'Ninja';
}
```

为了能使用该函数来创建对象，我们需要使用 **new** 操作符，例如：

```
>>> var hero = new Hero();
>>> hero.occupation;
"Ninja"
```

使用构造器函数的好处在于，它可以在创建对象时接收一些参数。下面，我们就来修改一下上面的构造函数，使它可以通过接收参数的方式来设定 `name` 属性：

```
function Hero(name) {
  this.name = name;
  this.occupation = 'Ninja';
  this.whoAreYou = function() {
    return "I'm " + this.name + " and I'm a " + this.occupation;
  }
}
```

现在，我们就能利用同一个构造器来创建不同的对象了：

```
>>> var h1 = new Hero('Michelangelo');
>>> var h2 = new Hero('Donatello');
>>> h1.whoAreYou();
"I'm Michelangelo and I'm a Ninja"
>>> h2.whoAreYou();
"I'm Donatello and I'm a Ninja"
```

依照惯例，我们应该将构造器函数的首字母大写，以便显著地区别于其他的一般函数。另外，如果我们在调用一个构造器函数时忽略了 **new** 操作符，尽管代码不会出错，但它的行为可能会令人出乎预料，例如：

```
>>> var h = Hero('Leonardo');
>>> typeof h
"undefined"
```

能看出来上面发生了什么吗？由于这里没有使用 `new` 操作符，因此我们不是在创建一个新的对象。这个函数调用与其他函数并没有区别，这里的 `h` 值应该就是该函数的返回值。

而由于该函数不返回任何东西（它没有返回值），所以它实际上返回的是 **undefined** 值，并将该值赋值给了 `h`。

那么，在这种情况下 `this` 引用的是什么呢？答案是全局对象。

4.1.8 全局对象

之前，我们已经讨论过全局变量（以及应该如何避免使用它们）和 JavaScript 程序在宿主环境（例如浏览器）中的具体运行情况。现在，我们又学习了对象的相关知识，是时候了解一些真相了：事实上，程序所在的宿主环境一般都会为其提供一个全局对象，而所谓的全局变量其实都只不过是该对象的属性罢了。

例如当程序的宿主环境是 Web 浏览器时，它所提供的全局对象就是 `window` 了。

下面，我们来看一个具体示例。首先，我们在所有函数之外声明了一个全局变量，例如：

```
>>> var a = 1;
```

然后，我们就可以通过各种不同的方式来访问该全局变量了：

- ◆ 可以当做一个变量 `a` 来访问。
- ◆ 也可以当做全局对象的一个属性来访问，例如 `window['a']` 或者 `window.a`。

现在，让我们回过头去分析一下刚才那个不使用 `new` 操作符调用构造器函数的情况，那时候，`this` 值指向的是全局对象，并且所有的属性设置都是针对 `this` 所代表 `window` 对象的。

也就是说，当我们声明了一个构造函数，但又不通过 `new` 来调用它时，代码就会返回“**undefined**”。

```
>>> function Hero(name) {this.name = name;}
>>> var h = Hero('Leonardo');
>>> typeof h
"undefined"
>>> typeof h.name
h has no properties
```

由于我们在 `Hero` 中使用了 `this`，所以这里就会创建一个全局变量（同时也是全局对象的一个属性）。

```
>>> name
"Leonardo"
>>> window.name
"Leonardo"
```

而如果我们使用 `new` 来调用相同的构造函数，我们就会创建一个新对象，并且 `this` 也会自动指向该对象。

```
>>> var h2 = new Hero('Michelangelo');
>>> typeof h2
"object"
>>> h2.name
"Michelangelo"
```

除此之外，我们在第3章所见的那些函数也都可以当做 `window` 对象方法来调用，例如下面两个调用的效果完全相同。

```
>>> parseInt('101 dalmatians')
101
>>> window.parseInt('101 dalmatians')
101
```

4.1.9 构造器属性

当我们创建对象时，实际上同时也赋予了该对象一种特殊的属性——即构造器属性（`constructor property`）。该属性实际上是一个指向用于创建该对象的构造器函数的引用。

例如，我们继续之前的例子：

```
>>> h2.constructor
Hero (name)
```

当然，由于构造器属性所引用的是一个函数，因此我们也可以利用它来创建一个其他新对象。例如像下面这样，大意就是：“无论对象 `h2` 有没有被创建，我们都可以用它来创建另一个对象”。

```
>>> var h3 = new h2.constructor('Rafaello');
>>> h3.name;
"Rafaello"
```

另外，如果对象是通过对象文本标识法创建的，那么实际上它就是由内建构造器 `Object()`

函数所创建的（关于这一点，我们稍后还会再做详细介绍）。

```
>>> var o = {};  
>>> o.constructor;  
Object()  
>>> typeof o.constructor;  
"function"
```

4.1.10 instanceof 操作符

通过 `instanceof` 操作符，我们可以测试一个对象是不是由某个指定的构造器函数所创建的。例如：

```
>>> function Hero() {}  
>>> var h = new Hero();  
>>> var o = {};  
>>> h instanceof Hero;  
true  
>>> h instanceof Object;  
true  
>>> o instanceof Object;  
true
```

请注意，这里的函数名后面没有加括号（即不是 `h instanceof Hero()`），因为这里不是函数调用，所以我们只需要像使用其他变量一样，引用该函数的名字即可。

4.1.11 返回对象的函数

除了使用 `new` 操作符来调用构造器函数以外，我们也可以抛开 `new` 操作符，只用一般函数来创建对象。这就需要有一个能执行某些预备工作，并以对象为返回值的函数。

例如，下面就有一个用于产生对象的简单函数 `factory()`：

```
function factory(name) {  
  return {  
    name: name  
  };  
}
```

然后我们调用 `factory()`：

```
>>> var o = factory('one');
```

```
>>> o.name
"one"
>>> o.constructor
Object()
```

实际上，构造器函数也是可以返回对象的，只不过在 `this` 值的使用上会有所不同。这意味着我们需要修改构造器函数的默认行为。下面，我们来看看具体是怎么做的。

这是构造器的一般用法：

```
>>> function C() {this.a = 1;}
>>> var c = new C();
>>> c.a
1
```

但现在要考虑的是这种用法：

```
>>> function C2() {this.a = 1; return {b: 2};}
>>> var c2 = new C2();
>>> typeof c2.a
"undefined"
>>> c2.b
2
```

能看出来发生了什么吗？在这里，构造器返回的不再是包含属性 `a` 的 `this` 对象，而是另一个包含属性 `b` 的对象^①。但这也只有在函数的返回值是一个对象时才会发生，而当我们企图返回的是一个非对象类型时，该构造器将会照常返回 `this`。

4.1.12 传递对象

当我们拷贝某个对象或者将它传递给某个函数时，往往传递的都是该对象的引用。因此我们在引用上所做的任何改动，实际上都会影响它所引用的原对象。

在下面的示例中，我们将会看到对象是如何赋值给另一个变量的，并且，如果我们对该拷贝做一些改变操作的话，原对象也会跟着被改变：

```
>>> var original = {howmany: 1};
>>> var copy = original;
>>> copy.howmany
```

^① 注意，`return` 语句中使用的是大括号，也就是说 `{b:2}` 是一个独立的对象。——译者注


```
1
>>> copy.howmany = 100;
100
>>> original.howmany
100
```

同样的，将对象传递给函数的情况也大抵如此：

```
>>> var original = {howmany: 100};
>>> var nullify = function(o) {o.howmany = 0;}
>>> nullify(original);
>>> original.howmany
0
```

4.1.13 对象比较

当我们对对象进行比较操作时，当且仅当两个引用指向同一个对象时为 **true**。而如果是不同的对象，即使它们碰巧拥有相同的属性和方法，比较操作也会返回 **false**。

下面，我们来创建两个看上去完全相同的对象：

```
>>> var fido = {breed: 'dog'};
>>> var benji = {breed: 'dog'};
```

然后，我们对它们进行比较，操作将会返回 **false**：

```
>>> benji === fido
false
>>> benji == fido
false
```

我们可以新建一个变量 `mydog`，并将其中一个对象赋值给它。这样一来 `mydog` 实际上就指向了这个变量。

```
>>> var mydog = benji;
```

在这种情况下，`mydog` 与 `benji` 所指向的对象是相同的（也就是说，改变 `mydog` 的属性就等同于改变 `benji`），比较操作就会返回 **true**。

```
>>> mydog === benji
true
```

并且，由于 `fido` 是一个不同的对象，所以它不能与 `mydog` 进行比较。

```
>>> mydog === fido
false
```

4.1.14 Firebug 控制台中的对象

在进一步深入介绍 JavaScript 的内建对象之前，让我们先来了解一些对象在 Firebug 中的工作情况。

到目前为止，我们已经在本章测试了许多示例，应该已经注意到了对象在控制台中是如何显示的。如果我们创建并命名了一个对象，就会获得一个用于描述该对象所含属性的字符串（如果属性过多，就会只显示头几个），如图 4-1 所示。

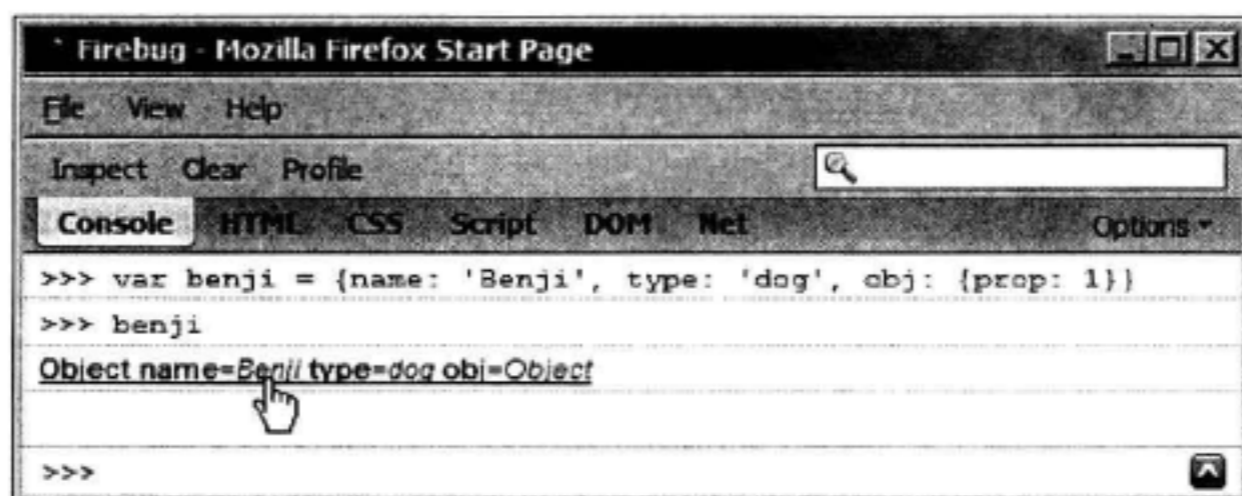


图 4-1

如果我们单击其中的某个对象，就会进入 Firebug 的 DOM 选项卡（见图 4-2），其中列出了该对象的所有属性。如果这之中的某个属性本身也是一个对象，那么它的前面就会呈现一个加号（+）以便我们继续展开它。总之，该工具可以方便我们轻松地了解目标对象中所含的内容。

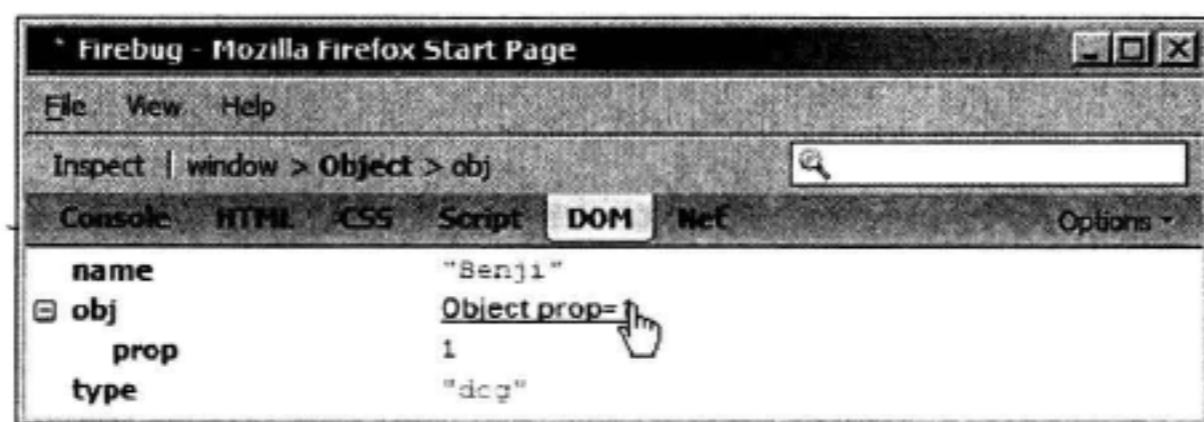


图 4-2

另外，控制台也为我们提供了一个叫做 `console` 的对象和一系列的方法，例如 `console.log()`、`console.error()` 和 `console.info()`。我们能通过这些函数所显示的值来了解我们想知道的控制台信息。

`console.log()` 既可以在我们想进行某种快速测试时提供一些便利，也可以在我们处理某些真实脚本时记录一些中间调试信息。例如在下面，我们示范了如何在循环中使用该函数（见图 4-3）：

```
>>> for(var i = 0; i < 5; i++) { console.log(i); }  
0  
1  
2  
3  
4
```

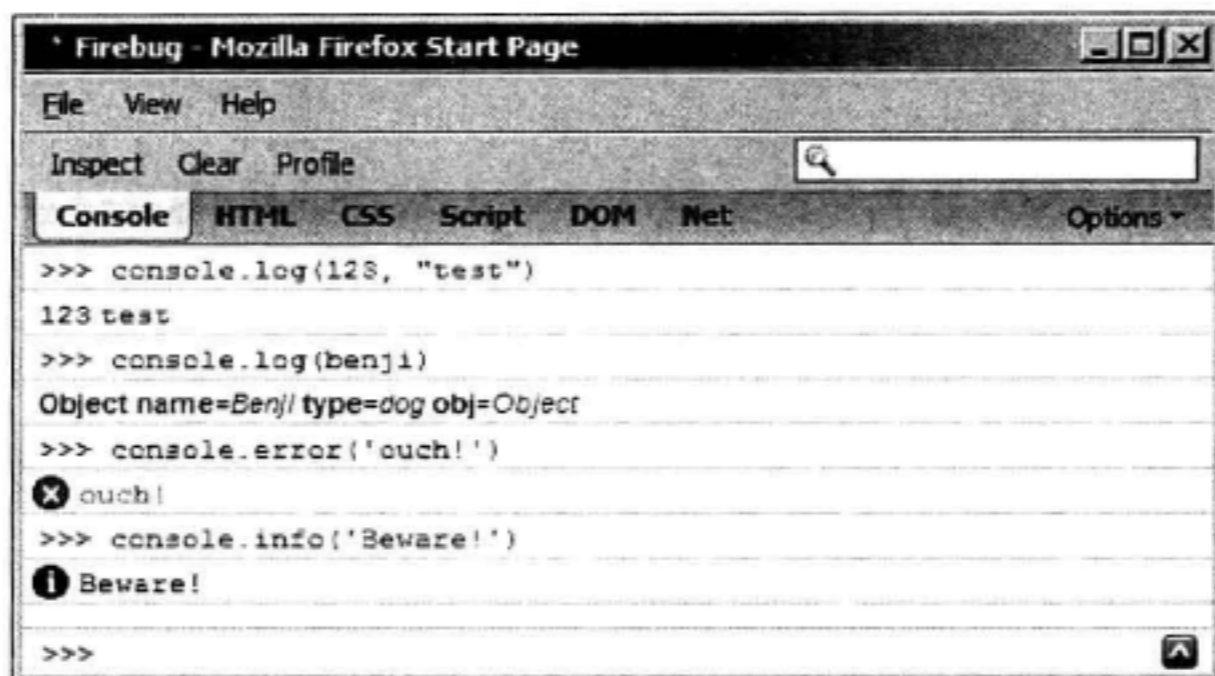


图 4-3

4.2 内建对象

到目前为止，本章所使用的实际上都是 `Object()` 构造器函数，它会在我们使用对象文本标识法，或访问相关构造器属性时返回新建的对象。而 `Object()` 也只是 JavaScript 中众多内建构造器之一，在本章接下来的内容中，我们将会为您介绍一一介绍其余的内建构造器。

内建对象大致上可以分为三个组：

- ◆ 数据封装类对象——包括 `Object`、`Array`、`Boolean`、`Number` 和 `String`。这些对象代表着 JavaScript 中不同的数据类型，并且都拥有各自不同的 `typeof` 返回值（这点我们在第 2 章中讨论过），以及 `undefined` 和 `null` 状态。
- ◆ 工具类对象——包括 `Math`、`Date`、`RegExp` 等用于提供遍历的对象。
- ◆ 错误类对象——包括一般性错误对象以及其他各种更特殊的错误类对象。它们可以在某些异常发生时帮助我们纠正程序工作状态。

在本章，我们只讨论这些内建对象的一小部分方法。如果想获得更完整的资料，读者可以参考附录 C 中的内容。

另外值得一提的是，不要去纠结什么是内建对象，什么是内建构造器，实际上它们是一回事。要不了多久您就会明白，无论是函数还是构造器函数，最后都是对象。

4.2.1 Object^①

Object 是 JavaScript 中所有对象的父级对象，这意味着我们创建的所有对象都继承于此。为了新建一个空对象，我们既可以用对象文本标识法也可以调用 `Object()` 构造器函数，即下面这两行代码的执行结果是等价的：

```
>>> var o = {};
>>> var o = new Object();
```

所谓的空对象，实际上并非是完全无用的，它还是包含了一些方法和属性的。下面，我们来看看其中的一小部分：

- ◆ 返回构造器函数的构造器属性。
- ◆ 返回对象描述字符串的 `toString()` 方法。
- ◆ 返回对象单值描述信息的 `valueOf()` 方法。一般情况下，返回的就是对象本身。

现在来看看这些方法的实际应用。首先，我们来创建一个对象：

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

然后调用 `toString()` 方法，返回该对象的描述字符串：

```
>>> o.toString()
"[object Object]"
```

`toString()` 方法会在某些需要用字符串来表示对象的时候被 JavaScript 内部调用。例如 `alert()` 的工作就需要用到这样的字符串。所以，如果我们将对象传递给了一个 `alert()` 函数，`toString()` 方法就会在后台被调用，也就是说，下面两行代码的执行结果是相同的：

```
>>> alert(o)
>>> alert(o.toString())
```

另一种会使用字符串描述文本的地方是字符串连接操作，如果我们将某个对象与字符串进行连接，那么该对象就先调用自身的 `toString()` 方法：

```
>>> "An object: " + o
```

^① 由于这本身就是一个 JavaScript 对象的名称，这里就不进行翻译处理了。——译者注

```
"An object: [object Object]"
```

`valueOf()` 方法也是一个所有的对象所共有的方法。对于简单的对象来说（即构造器是 `Object` 的对象），`valueOf()` 方法返回的就是对象自己。

```
>>> o.valueOf() === o
true
```

总而言之：

- ◆ 我们创建对象时既可以用 `var o = {}` 的形式（即执行对象文本标识法），也可以用 `var o = new Object()`。
- ◆ 无论是多复杂的对象，它都是继承自 `Object` 对象的，并且拥有其所有的方法（例如 `toString()`）和属性（例如构造器）。

4.2.2 Array

`Array()` 是一个用来构建数组的内建构造器函数，例如：

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

这与下面的数组文本标识法是等效的：

```
>>> var a = [];
```

无论数组是以什么方式创建的，我们都能照常往里添加元素：

```
>>> a[0] = 1; a[1] = 2; a;
[1, 2]
```

当我们使用 `Array()` 构造器创建新数组时，也可以通过传值的方式为其设定元素。

```
>>> var a = new Array(1, 2, 3, 'four');
>>> a;
[1, 2, 3, "four"]
```

但是如果我们传递给该构造器的是一个数字，就会出现一种异常情况，即该数值会被认为是数组的长度。

```
>>> var a2 = new Array(5);
>>> a2;
```

```
[undefined, undefined, undefined, undefined, undefined]
```

既然数组是由构造器来创建的，那么这是否意味着数组实际上是一个对象呢？的确如此，我们可以用 `typeof` 操作符来验证一下：

```
>>> typeof a;
"object"
```

由于数组也是对象，那么就说明它也继承了 `Object` 的所有方法和属性。

```
>>> a.toString();
"1,2,3,four"
>>> a.valueOf()
[1, 2, 3, "four"]
>>> a.constructor
Array()
```

尽管数组也是一种对象，但还是有一些特殊之处的，因为：

- ◆ 它们的属性名都是从 0 开始递增，并自动生成数值。
- ◆ 它们都拥有一个用于记录数组中元素数量的 `length` 属性。
- ◆ 它们都是在父级对象的基础上扩展额外的内建方法。

下面来实际验证一下对象与数组之间的区别，让我们从创建空对象 `o` 和空数组 `a` 开始：

```
>>> var a = [], o = {};
```

首先，定义数组对象时会自动生成一个 `length` 属性。而这在一般对象中是没有的：

```
>>> a.length
0
>>> typeof o.length
"undefined"
```

在为数组和对象添加数字和非数字属性方面，两者并没有什么区别：

```
>>> a[0] = 1; o[0] = 1;
>>> a.prop = 2; o.prop = 2;
```

`length` 属性通常会随着数字属性的数量而更新，而非数字属性则会被忽略。

```
>>> a.length
```

1

当然，我们也可以手动设置 `length` 属性。如果设置的值大于当前数组中元素数量，剩下的那部分会被自动创建（值为 `undefined`）空对象所填充。

```
>>> a.length = 5
5
>>> a
[1, undefined, undefined, undefined, undefined]
```

而如果我们设置的 `length` 值小于当前元素数，多出的那部分元素将会被移除：

```
>>> a.length = 2;
2
>>> a
[1, undefined]
```

值得关注的数组方法

除了从父级对象那里继承的方法以外，数组对象中还有一些更为有用的方法，例如 `sort()`、`join()` 和 `slice()` 等（完整的方法列表见附录 C）。

下面，我们将通过一个数组来试验一下这些方法：

```
>>> var a = [3, 5, 1, 7, 'test'];
```

`push()` 方法会在数组的末端添加一个新元素，而 `pop()` 方法则会移除最后一个元素，也就是说 `a.push("new")` 就相当于 `a[a.length] = "new"`，而 `a.pop()` 则与 `a.length--` 的结果相同。

另外，`push()` 返回的是改变后的数组长度，而 `pop` 所返回的则是被移除的元素。

```
>>> a.push('new')
6
>>> a
[3, 5, 1, 7, "test", "new"]
>>> a.pop()
"new"
>>> a
[3, 5, 1, 7, "test"]
```

而 `sort()` 方法则是用于给数组排序的，它会返回修改后的数组，在下面的示例中，

排序完成后，a 和 b 所指向的数组是相同的：

```
>>> var b = a.sort();
>>> b
[1, 3, 5, 7, "test"]
>>> a
[1, 3, 5, 7, "test"]
```

join() 方法会返回一个由目标数组中所有元素值连接而成的字符串，另外，我们还可以通过该方法的参数来设定这些元素之间的字符串。例如：

```
>>> a.join(' is not ');
"1 is not 3 is not 5 is not 7 is not test"
```

slice() 方法会在不修改目标数组的情况下返回其中的某个片段，该片段的首尾索引位置将由 slice() 的头两个参数来指定（都以 0 为基数）。

```
>>> b = a.slice(1, 3);
[3, 5]
>>> b = a.slice(0, 1);
[1]
>>> b = a.slice(0, 2);
[1, 3]
```

所有的截取完成之后，原数组的状态不变：

```
>>> a
[1, 3, 5, 7, "test"]
```

splice() 则是会修改目标数组的。它会移除并返回指定切片，并且在可选情况下，它还会用指定的新元素来填补被切除的空缺。该方法的头两个参数所指定的是要移除切片的首尾索引位置，其他参数则是用于填补的新元素值。

```
>>> b = a.splice(1, 2, 100, 101, 102);
[3, 5]
>>> a
[1, 100, 101, 102, 7, "test"]
```

当然，用于填补空缺的新元素是可选的，我们也可以直接跳过：

```
>>> a.splice(1, 3)
[100, 101, 102]
```



```
>>> a
[1, 7, "test"]
```

4.2.3 Function

之前，我们已经了解了函数是一种特殊的数据类型，但事实还远不止如此，它实际上是一种对象。函数对象的内建构造器是 `Function()`，我们可以将它作为创建函数的一种备选方式（但我们并不推荐这种方式）。

也就是说，下面三种定义函数的方式是等效的：

```
>>> function sum(a, b) {return a + b;};
>>> sum(1, 2)
3
>>> var sum = function(a, b) {return a + b;};
>>> sum(1, 2)
3
>>> var sum = new Function('a', 'b', 'return a + b;');
>>> sum(1, 2)
3
```

如果我们使用的是 `Function()` 构造器的话，就必须通过其参数来设定函数的参数名（通常是字符串）和函数体中的代码（也是字符串）。JavaScript 引擎自会对这些源代码进行解析^①，然后直接创建新函数，这样一来，就会带来与 `eval()` 相似的缺点。因此我们要尽量避免使用 `Function()` 构造器来定义函数。

如果我们需要用 `Function` 构造器来创建一个拥有许多参数的函数，需要注意一件事，即这些参数可以是一个由逗号分割而成的单列表，所以，下面例子中的这些函数定义是相同的：

```
>>> var first = new Function('a, b, c, d', 'return arguments;');
>>> first(1,2,3,4);
[1, 2, 3, 4]
>>> var second = new Function('a, b, c', 'd', 'return arguments;');
>>> second(1,2,3,4);
[1, 2, 3, 4]
>>> var third = new Function('a', 'b', 'c', 'd',
'return arguments;');
>>> third(1,2,3,4);
[1, 2, 3, 4]
```

^① 这是因为 JavaScript 引擎无法检查字符串（即您所传递的参数）中的内容。——译者注



最佳做法建议

请尽量避免使用 `Function()` 构造器。因为它与 `eval` 和 `setTimeout()`（关于该函数的讨论，我们稍后会看到）一样，始终会以字符串的形式通过 JavaScript 的代码检查。

4.2.3.1 Function 对象的属性

与其他对象不同的是，函数对象中含有一个构造器属性，其引用的就是 `Function()` 构造器函数。

```
>>> function myfunc(a){return a;}
>>> myfunc.constructor
Function()
```

另外，`Function` 对象中也有一个 `length` 属性，用于记录该函数所拥有的参数数量。

```
>>> function myfunc(a, b, c){return true;}
>>> myfunc.length
3
```

有趣的是，该对象中还有一个在 ECMA 标准之外的属性。该属性对于浏览器来说还是非常重要的，那就是所谓的 `caller` 属性。这个属性会返回一个调用该函数对象的外层函数引用。也就是说，如果我們是在函数 `B()` 中调用函数 `A()` 的，那么只要我们在 `A()` 中调用了 `A.caller`，结果就会返回函数 `B()`。

```
>>> function A(){return A.caller;}
>>> function B(){return A();}
>>> B()
B()
```

在我们希望自己的函数能根据其调用函数作出不同的反应时，该属性会显得非常有用。值得一提的是，如果我们是在全局域调用 `A()` 的（也就是说，它没有任何外层函数），`A.caller` 的值就会为 `null`。

```
>>> A()
null
```

当然，函数对象中最重要的还是 `prototype` 属性。我们将会在第 5 章中详细介绍该属性，现在我们只是简单做个说明：

- ◆ 每个函数的 `prototype` 属性中都包含了一个对象。
- ◆ 它只有在该函数是构造器时才会发挥作用。
- ◆ 该函数创建的所有对象都会持有一个该 `prototype` 属性的引用，并可以将其当做自身的属性来使用。

下面，我们来演示一下 `prototype` 属性的使用。我们从一个简单的对象开始，其中只有一个 `name` 属性和一个 `say()` 方法：

```
var some_obj = {
  name: 'Ninja',
  say: function(){
    return 'I am a ' + this.name;
  }
}
```

如果我们在这里创建一个空函数（`hollow function`）。我们可以看到该函数的 `prototype` 属性是一个空对象。

```
>>> function F(){}
>>> typeof F.prototype
"object"
```

现在，如果我们对该 `prototype` 属性进行修改，就会发现一些有趣的事情：当前默认的空对象被直接替换成了其他对象。下面我们将 `some_obj` 赋值给这个 `prototype`：

```
>>> F.prototype = some_obj;
```

现在，如果我们将 `F()` 当做一个构造器函数来创建对象，那么新对象 `obj` 就会拥有对 `F.prototype` 属性的访问权（如果它存在的话）。

```
>>> var obj = new F();
>>> obj.name
"Ninja"
>>> obj.say()
"I am a Ninja"
```

关于 `prototype` 属性的更多内容，我们会在第 5 章中做详细的介绍，

4.2.3.2 Function 对象的方法

所有的 `Function` 对象都是继承自父级对象 `Object` 的，因此它默认就拥有其父级对象

的所有方法。例如 `toString()`。当我们对一个函数调用 `toString()` 方法时，所得到的就是该函数的源代码。

```
>>> function myfunc(a, b, c) {return a + b + c;}
>>> myfunc.toString()
"function myfunc(a, b, c) {
  return a + b + c;
}"
```

但如果我们用这种方法来查看内建函数的源码的话，就只会得到一个毫无用处的字符串 “[native code]”。

```
>>> eval.toString()
"function eval() {
  [native code]
}"
```

另外，函数对象中还有两个非常有用的方法：`call()` 和 `apply()`。通过这两个方法，我们就能让对象去借用其他对象中的方法，为己所用。这也是一种重用代码的方式，非常简单且实用。

下面，我们来定义一个 `some_obj` 对象，其中包含了一个 `say()` 方法

```
var some_obj = {
  name: 'Ninja',
  say: function(who){
    return 'Haya ' + who + ', I am a ' + this.name;
  }
}
```

这样一来，我们就可以调用该对象的 `say()` 方法，并在其中使用 `this.name` 来访问对象的 `name` 属性。

```
>>> some_obj.say('Dude');
"Haya Dude, I am a Ninja"
```

现在，让我们再来创建一个 `my_obj` 对象，它只有一个 `name` 属性：

```
>>> my_obj = {name: 'Scripting guru'};
```

显然，`my_obj` 也适用于 `some_obj` 的 `say()` 方法，因此希望将它当做自身的方法来

调用。在这种情况下，我们就可以试试 `say()` 函数的对象方法 `call()`：

```
>>> some_obj.say.call(my_obj, 'Dude');
"Haya Dude, I am a Scripting guru"
```

成功了！但您明白这是怎么回事吗？由于我们在调用 `say()` 函数的对象方法 `call()` 时传递了两个参数：对象 `my_obj` 和字符串“**Dude**”。这样一来，当 `say()` 被调用时，其中的 `this` 就被自动设置成了 `my_obj` 对象的引用。因而我们看到，`this.name` 返回的不再是“**Ninja**”，而是“**Scripting guru**”了^①。

如果我们调用 `call` 方法时需要传递更多的参数，可以在后面依次加入它们：

```
some_obj.someMethod.call(my_obj, 'a', 'b', 'c');
```

如果我们没有将对象传递给 `call()` 的首参数，或者传递给它的是 `null`，它的调用对象将会被默认为全局对象^②。

`apply()` 的工作方式与 `call()` 基本相同，唯一的不同之处在于参数的传递形式，这里目标函数所需要的参数都是通过一个数组来传递的。所以，下面两行代码的作用是等效的：

```
some_obj.someMethod.apply(my_obj, ['a', 'b', 'c']);
some_obj.someMethod.call(my_obj, 'a', 'b', 'c');
```

因而，对于之前的示例，我们也可以这样写：

```
>>> some_obj.say.apply(my_obj, ['Dude']);
"Haya Dude, I am a Scripting guru"
```

4.2.3.3 重新认识 `arguments` 对象

在第 3 章中，我们已经掌握了如何在一个函数中通过 `arguments` 来访问传递给该函数的所有参数。例如：

```
>>> function f() {return arguments;}
>>> f(1,2,3)
[1, 2, 3]
```

尽管 `arguments` 看上去像是一个数组，但它实际上是一个类似数组的对象。它和数

^① 实际上就是通过 `call` 的首参数修改了对象函数的 `this` 值。——译者注

^② 即 `this` 指向的是全局对象。——译者注

组相似是因为其中也包含索引元素和 `length` 属性。但相似之处也就到此为止了，因为如果 `arguments` 是一个单纯的数组，它是不会提供一些像 `sort()`、`slice()` 这样的数组方法的。

另外，`arguments` 对象中还有一个值得关注的属性——`callee` 属性。该属性引用的是当前被调用的函数对象。也就是说，如果我们所建函数的返回值是 `arguments.callee`，那么该函数在被调用时就会返回自身的引用。

```
>>> function f(){return arguments.callee;}
>>> f()
f()
```

此外，我们还可以通过 `arguments.callee` 属性来实现匿名函数的递归调用，例如：

```
(
  function(count){
    if (count < 5) {
      alert(count);
      arguments.callee(++count);
    }
  }
)(1)
```

如您所见，我们在这里定义了一个函数。该函数会将其 `count` 参数的值显示在 `alert` 窗口中，并将参数递增后继续调用自身。而封装在其大括号内的函数会负责设定其参数，以决定函数于何时结束，在这里，我们传递给内部的值是 1，因此 `alert` 窗口会弹出四次，分别显示 1、2、3、4。

4.2.4 Boolean

继续我们的 JavaScript 内建对象之旅，接下来介绍的对象相对要容易得多，只不过是些基本数据类型的封装，它们包括 `Boolean`、`Number`、`String`。

在第 2 章中，我们已经学习了大量关于 `Boolean` 类型的应用。现在，我们要介绍的是与 `Boolean()` 构造器相关的内容。

```
>>> var b = new Boolean();
```

在这里最重要的一点是，我们必须明白这里所创建的 `b` 是一个新的对象，它不是一个基本的布尔值。如果想获得这样的基本类型值，我们可以调用该对象的 `valueOf()` 方法

(继承自 Object 对象)。

```
>>> var b = new Boolean();
>>> typeof b
"object"
>>> typeof b.valueOf()
"boolean"
>>> b.valueOf()
false
```

总体而言，用 `Boolean()` 构造器所创建的对象并没有多少实用性，因为它并没有提供来自父级对象以外的任何方法和属性。

相反，`Boolean()` 在被当做一般函数时，或许会比用于 `new` 操作符更实用一些，因为我们可以通过这种方式将一些非布尔值转换为布尔值（其效果相当于进行两次取反操作：`!!value`）。

```
>>> Boolean("test")
true
>>> Boolean("")
false
>>> Boolean({})
true
```

而且，在 JavaScript 中，除了那六种 falsy 值外，其他所有的都属于 truthy 值^①，其中也包括空对象。这就意味着所有由 `new Boolean()` 语句创建的布尔对象都等于 `true`，因为它们都是对象。

现在，我们来创建两个布尔对象，一个为 truthy，一个为 falsy：

```
>>> var b1 = new Boolean(true)
>>> b1.valueOf()
true
>>> var b2 = new Boolean(false)
>>> b2.valueOf()
false
```

然后将这两个对象转换为基本的布尔值的话，我们会看到它们的值都是 `true`。因为所有对象都属于 truthy 值。

^① 关于 falsy 和 truthy，作者在第 2 章中已经讨论过了。——译者注

```
>>> Boolean(b1)
true
>>> Boolean(b2)
true
```

4.2.5 Number

Number() 函数的用法与 Boolean() 基本相同，即：

- ◆ 在被当做一般函数时，它会试图将任何值转换为数字，这与 parseInt() 或 parseFloat() 起到的作用基本相同。

- ◆ 在被当做构造器函数时（即用于 new 操作符），它会创建一个对象。

```
>>> var n = Number('12.12');
>>> n
12.12
>>> typeof n
"number"
>>> var n = new Number('12.12');
>>> typeof n
"object"
```

由于函数本身也是对象，所以会拥有一些属性。在 Number() 函数中，有一些内置属性是值得我们注意的（它们是不可更改的）：

```
>>> Number.MAX_VALUE
1.7976931348623157e+308
>>> Number.MIN_VALUE
5e-324
>>> Number.POSITIVE_INFINITY
Infinity
>>> Number.NEGATIVE_INFINITY
-Infinity
>>> Number.NaN
NaN
```

此外，Number 对象中还提供了三个方法，它们分别是：toFixed()、toPrecision() 和 toExponential()（详细内容见附录 C）。

```
>>> var n = new Number(123.456)
>>> n.toFixed(1)
```



```
"123.5"
```

值得一提的是，我们在不显式创建 `Number` 对象的情况下也能调用这三个方法。例如，在下面这种情况下，`Number` 对象的创建（与销毁）将全部在后台进行：

```
>>> (12345).toExponential()
"1.2345e+4"
```

与所有的对象一样，`Number` 对象也提供了自己的 `toString()` 方法。但值得注意的是，该对象的 `toString()` 方法有一个可选的 `radix` 参数（它的默认值是 10）。

```
>>> var n = new Number(255);
>>> n.toString();
"255"
>>> n.toString(10);
"255"
>>> n.toString(16);
"ff"
>>> (3).toString(2);
"11"
>>> (3).toString(10);
"3"
```

4.2.6 String

我们可以通过 `String()` 构造器函数来新建 `String` 对象。该对象为我们提供了一系列用于文本操作的方法，但您最好还是使用基本的字符串类型。

下面，我们通过一个示例来看看 `String` 对象与基本的字符串类型之间有什么区别。

```
>>> var primitive = 'Hello';
>>> typeof primitive;
"string"
>>> var obj = new String('world');
>>> typeof obj;
"object"
```

`String` 对象实际上就像是一个字符数组，其中也包括用于每个字符的索引属性，以及整体的 `length` 属性。

```
>>> obj[0]
"w"
```

```
>>> obj[4]
"d"
>>> obj.length
5
```

如果我们想获得 `String` 对象的基本类型值，可以调用该对象的 `valueOf()` 或 `toString()` 方法（都继承自 `Object` 对象），如果我们必须在一个字符串环境中使用 `String` 对象，或许就始终需要用到 `toString()` 方法。

```
>>> obj.valueOf()
"world"
>>> obj.toString()
"world"
>>> obj + ""
"world"
```

而基本类型的字符串就不是对象了，因此它们不含有任何属性和方法。但 `JavaScript` 还是为我们提供了一些将基本字符串类型转换为 `String` 对象的语法。

例如在下面的示例中，当我们将一个基本字符串当做对象来使用时，后台就会执行相应的 `String` 对象创建（与销毁）操作。

```
>>> "potato".length
6
>>> "tomato"[0]
"t"
>>> "potato"["potato".length - 1]
"o"
```

最后我们再来看一个说明基本字符串与 `String` 对象之间区别的例子：当它们被转换成布尔值时，尽管空字符串属于 `falsy` 值，但所有的 `String` 对象都是 `truthy` 值。

```
>>> Boolean("")
false
>>> Boolean(new String(""))
true
```

与 `Number()` 和 `Boolean()` 类似，如果我们不通过 `new` 操作符来调用 `String()`，它就会试图将其参数转换为一个基本字符串。因此，如果其参数是一个对象的话，这就等于调用该对象的 `toString()` 方法。

```
>>> String(1)
```

```
"1"  
>>> String({p: 1})  
"[object Object]"  
>>> String([1,2,3])  
"1,2,3"
```

String 对象中值得关注的方法

下面，让我们来示范一下部分 `String` 对象方法的调用（如果想获得完整的方法列表，可以参考附录 C）。

首先从新建 `String` 对象开始：

```
>>> var s = new String("Couch potato");
```

接下来是用于字符串大小写转换的方法，`toUpperCase()` 与 `toLowerCase()`：

```
>>> s.toUpperCase()  
"COUCH POTATO"  
>>> s.toLowerCase()  
"couch potato"
```

`charAt()` 方法返回的是我们指定位置的字符，这与中括号的作用相当（字符串本身就是—个字符数组）。

```
>>> s.charAt(0);  
"C"  
>>> s[0]  
"C"
```

如果我们传递给 `charAt()` 方法的位置并不存在，它就会返回一个空字符串：

```
>>> s.charAt(101)  
""
```

`indexOf()` 方法可以帮助我们实现字符串内部搜索，该方法在遇到匹配字符时会返回第一次匹配位置的索引值。由于该索引值是从 0 开始计数的，所以字符串“Couch”中第二个字符“o”的位置索引值为 1。

```
>>> s.indexOf('o')  
1
```

另外，我们也可以通过可选参数指定搜索开始的位置（以索引值的形式）。例如下面所

找到的就是字符串中的第二个“o”，因为我们指定的搜索是从索引2处开始的。

```
>>> s.indexOf('o', 2)
7
```

如果我们想让搜索从字符串的末端开始，可以调用 `lastIndexOf()` 方法（但索引值的技术方法不变）。

```
>>> s.lastIndexOf('o')
11
```

当然，上述方法的搜索对象不仅仅局限于字符，也可以用于字符串搜索。并且搜索是区分大小写的。

```
>>> s.indexOf('Couch')
0
```

如果方法找不到匹配对象，返回的位置索引值就为-1：

```
>>> s.indexOf('couch')
-1
```

如果我们想进行一次大小写无关的搜索，可以将字符串进行大小写转换后再执行搜索：

```
>>> s.toLowerCase().indexOf('couch')
0
```

如果相关的搜索方法返回的索引值是 0，就说明字符串的匹配部分是从 0 处开始的。这有可能会给 `if` 语句的使用带来某些混淆因素，我们像下面这样使用 `if` 语句，就会将索引值 0 隐式地转换为布尔值 `false`，虽然这种写法没有什么语法错误，但在逻辑上却完全错了：

```
if (s.indexOf('Couch')) {...}
```

正确的做法是：当我们用 `if` 语句检测一个字符串中是否包含另一个字符串时，可以用数字-1 来做 `indexOf()` 结果的比较参照：

```
if (s.indexOf('Couch') !== -1) {...}
```

接下来，我们要介绍的是 `slice()` 和 `substring()`，这两个方法都可以用于返回目标字符串中指定的区间：

```
>>> s.slice(1, 5)
"ouch"
>>> s.substring(1, 5)
"ouch"
```

需要提醒的是，这两个方法的第二个参数所指定的都是区间的末端位置，而不是该区间的长度。不同之处在于对负值参数的处理方式，`substring()`方法会将其视为从0开始的计数形式，而`slice()`方法则会将其与字符串的长度相加。因此，如果我们传给它们的参数是(1, -1)的话，它们的情况分别是`substring(1, 0)`和`slice(1, s.length-1)`。

```
>>> s.slice(1, -1)
"ouch potat"
>>> s.substring(1, -1)
"C"
```

`split()`方法可以根据我们所传递的分割字符串，将目标字符串分割成一个数组。例如：

```
>>> s.split(" ")
["Couch", "potato"]
```

`split()`可以被视为`join()`的反操作，后者则会将一个数组合并成一个字符串。例如：

```
>>> s.split(' ').join(' ');
"Couch potato"
```

`concat()`方法通常用于追加字符串，它的功能与基本字符串类型的+操作符类似：

```
>>> s.concat("es")
"Couch potatoes"
```

需要注意的是，到目前为止，我们所讨论的方法返回的都是一个新的基本字符串，它们所做的任何修改都不会源字符串。我们所有的方法调用都不会影响原始字符串的值。

```
>>> s.valueOf()
"Couch potato"
```

通常情况下，我们会用`indexOf()`和`lastIndexOf()`方法进行字符串内搜索，但除此之外还有一些功能更为强大的方法（如`search()`、`match()`、`replace()`等），它们可以以正则表达式为参数来执行搜索任务。关于正则表达式，我们将会在讨论`RegExp()`

构造器函数时加以详细讨论。

现在，数据封装类对象已经全部介绍完了，接下来，我们要介绍一些工具类对象，它们分别是 `Math`、`Date` 和 `RegExp`。

4.2.7 Math

`Math` 与我们之前所见过的其他内建对象是有些区别的，既不能当做一般函数来调用，也不能用于 `new` 操作符来创建对象。实际上，`Math` 只是一个包含一系列方法和属性、用于数学计算的內建对象。

`Math` 的属性都是不可修改的，因此它们都以名字大写的方式来表示自己与一般属性变量的不同。下面让我们来看看这些属性：

数字 π ：

```
>>> Math.PI
3.141592653589793
```

2 的平方根：

```
>>> Math.SQRT2
1.4142135623730951
```

欧拉常数 e ：^①

```
>>> Math.E
2.718281828459045
```

2 的自然对数：

```
>>> Math.LN2
0.6931471805599453
```

10 的自然对数：

```
>>> Math.LN10
2.302585092994046
```

现在，您知道下次该如何忽悠朋友们了吧？（无论出于怎么样的尴尬理由）当他们开

^① 即自然对数的底数。——译者注

始使劲回想“e 的值是什么？我怎么忘记了”时，我们只需要轻松地在控制台输入 `Math.E`，就会立即得到答案。

接下来，我们再来看看 `Math` 对象所提供的一些方法（完整的方法列表请见附录 C）。

首先是生成随机数：

```
>>> Math.random()
0.3649461670235814
```

`random()` 所返回的是 0 到 1 之间的某个数，所以如果我们想要获得 0 到 100 之间的某个数的话，就可以这样：

```
>>> 100 * Math.random()
```

如果我们需要获取的是某个 `max` 和 `min` 之间的值，可以通过一个公式 $(\text{max} - \text{min}) * \text{Math.random()} + \text{min}$ 来获取，例如，我们想获取的是 2 到 10 之间的某个数，就可以这样：

```
>>> 8 * Math.random() + 2
9.175650496668485
```

如果这里需要的只是一个整数的话，我们既可以自己选择一种舍入方法来使用——`floor()` 用于舍弃，`ceil()` 用于取入，也可以直接调用 `round()` 方法来进行四舍五入。例如，下面的执行结果不是 0 就是 1：

```
>>> Math.round(Math.random())
```

如果我们想获得一个数字集合中的最大值或最小值，则可以调用 `max()` 和 `min()` 方法。所以，当我们对一个表单中所输入的月份进行验证时，可以用下面的方式来确保相关的数据能正常工作：

```
>>> Math.min(Math.max(1, input), 12)
```

除此之外，`Math` 对象还提供了一些用于执行数学计算的方法，这些计算是我们不需要去专门设计即可使用的。这意味着当我们想要执行指数运算时只需要调用 `pow()` 方法即可，而求平方根时只需要调用 `sqrt()`，另外还包括其他所有的三角函数计算——`sin()`、`cos()`、`atan()` 等等。例如，

求 2 的 8 次方：

```
>>> Math.pow(2, 8)
256
```

求9的平方根:

```
>>> Math.sqrt(9)
3
```

4.2.8 Date

`Date()` 是用于创建 `Date` 对象的构造器函数，我们在用它创建对象时可以传递以下几种参数：

- ◆ 无参数（默认为当天的日期）。
- ◆ 一个用于表现日期的字符串。
- ◆ 分别传递日、月、时间等值。
- ◆ 一个 `timestamp` 值。^①

下面是一个表示当天日期和时间的对象示例：

```
>>> new Date()
Tue Jan 08 2008 01:10:42 GMT-0800 (Pacific Standard Time)
```

（默认情况下，`Firebug` 会显示其控制台中所有对象的 `toString()` 结果，因此这里的长字符串“Tue Jan 08....”实际上就相当于我们调用该 `Date` 对象的 `toString()` 方法所获得的内容。）

接下来，我们看一些用字符串初始化 `Date` 对象的示例，注意它们各自不同的格式以及所指定的时间。

```
>>> new Date('2009 11 12')
Thu Nov 12 2009 00:00:00 GMT-0800 (Pacific Standard Time)
>>> new Date('1 1 2012')
Sun Jan 01 2012 00:00:00 GMT-0800 (Pacific Standard Time)
>>> new Date('1 mar 2012 5:30')
Thu Mar 01 2012 05:30:00 GMT-0800 (Pacific Standard Time)
```

^① UNIX 时间，或称 POSIX 时间是 UNIX 或类 UNIX 系统使用的时间表示方式：从协调世界时 1970 年 1 月 1 日 0 时 0 分 0 秒起至现在的总秒数，不包括闰秒。——译者注

在 JavaScript 中，通过不同的字符串来指定一个具体的日期是一种不错的做法，但这样做显然不够精确。更好的选择是向 `Date()` 构造器传递一些具体的数值，其中包括：

- ◆ 年份。
- ◆ 月份：从 0（1 月）到 11（12 月）。
- ◆ 日期：从 1 到 31。
- ◆ 时数：从 0 到 23。
- ◆ 分钟：从 0 到 59
- ◆ 秒钟：从 0 到 59。
- ◆ 毫秒数：从 0 到 999。

现在让我们来看一些具体示例。

如果我们传递所有参数：

```
>>> new Date(2008, 0, 1, 17, 05, 03, 120)
Tue Jan 01 2008 17:05:03 GMT-0800 (Pacific Standard Time)
```

如果只传递日期和时钟值：

```
>>> new Date(2008, 0, 1, 17)
Tue Jan 01 2008 17:00:00 GMT-0800 (Pacific Standard Time)
```

在这里，我们需要注意一件事，由于月份是从 0 开始的，所以这里的 1 指的是 2 月：

```
>>> new Date(2008, 1, 28)
Thu Feb 28 2008 00:00:00 GMT-0800 (Pacific Standard Time)
```

如果我们所传递的值越过了被允许的范围，`Date` 对象会自行启动“溢出式”前进处理。例如，由于 2008 年 2 月不存在 30 日这一天，所以它会自动解释为该年的 3 月 1 日（2008 年为闰年）。

```
>>> new Date(2008, 1, 29)
Fri Feb 29 2008 00:00:00 GMT-0800 (Pacific Standard Time)
>>> new Date(2008, 1, 30)
Sat Mar 01 2008 00:00:00 GMT-0800 (Pacific Standard Time)
```

类似地，如果我们传递的是 12 月 32 日，就会被自动解释为来年的 1 月 1 日：

```
>>> new Date(2008, 11, 31)
Wed Dec 31 2008 00:00:00 GMT-0800 (Pacific Standard Time)
>>> new Date(2008, 11, 32)
Thu Jan 01 2009 00:00:00 GMT-0800 (Pacific Standard Time)
```

最后，我们也可以通过 `timestamp` 的方式来初始化一个 `Date` 对象（这是一个以毫秒为单位的 UNIX 纪元方式，开始于 1970 年 1 月 1 日）。

```
>>> new Date(1199865795109)
Wed Jan 09 2008 00:03:15 GMT-0800 (Pacific Standard Time)
```

如果我们在调用 `Date()` 时没有使用 `new` 操作符，那么无论是否传递了参数，所得字符串的内容始终都将是当前的日期和时间（就像下面示例所运行的那样）：

```
>>> Date()
"Thu Jan 17 2008 23:11:32 GMT-0800 (Pacific Standard Time)"
>>> Date(1, 2, 3, "it doesn't matter");
"Thu Jan 17 2008 23:11:35 GMT-0800 (Pacific Standard Time)"
```

Date 对象方法的工作方式

一旦我们创建了 `Date` 对象，就可以调用该对象中的许多方法。其中使用最多的都是一些名为 `set*()` 或 `get*()` 的方法，例如 `getMonth()`、`setMonth()`、`getHours()`、`setHours()` 等等。下面我们来看一些具体的示例。

首先，新建一个 `Date` 对象：

```
>>> var d = new Date();
>>> d.toString();
"Wed Jan 09 2008 00:26:39 GMT-0800 (Pacific Standard Time)"
```

然后，将其月份设置成 3 月（记住，月份数是从 0 开始的）：

```
>>> d.setMonth(2);
1205051199562
>>> d.toString();
"Sun Mar 09 2008 00:26:39 GMT-0800 (Pacific Standard Time)"
```

接着，我们读取月份数：

```
>>> d.getMonth();
2
```

除了这些实例方法以外，Date() 函数或对象中还有另外一种属性。这些属性在不经实例化情况下使用，工作方式与 Math 的方法基本相同。在基于 class 概念的程序设计语言中，它们往往被称之为“静态”方法，因为它们的调用不需要依托对象示例。

例如，Date.parse() 方法会将其所接收的字符串转换成相应的 timestamp 格式，并返回：

```
>>> Date.parse('Jan 1, 2008')
1199174400000
```

而 Date.UTC() 方法则可以接受包括年份、月份、日期等在内的所有参数，并以此产生一个相应的、符合格林尼治时标准的 timestamp 值：

```
>>> Date.UTC(2008, 0, 1)
1199145600000
```

由于用 Date 创建对象时可以接受一个 timestamp 参数，因此我们也可以直接将 Date.UTC() 的结果传递给该构造器。在下面的示例中，我们演示了如何在新建 Date 对象的过程中，将 UTC() 返回的格林尼治时间转换为本地时间：

```
>>> new Date(Date.UTC(2008, 0, 1));
Mon Dec 31 2007 16:00:00 GMT-0800 (Pacific Standard Time)
>>> new Date(2008, 0, 1);
Tue Jan 01 2008 00:00:00 GMT-0800 (Pacific Standard Time)
```

下面，我们再来看最后一个关于 Date 对象的工作示例。假如，我很好奇自己 2012 年的生日是星期几，就可以这样：

```
>>> var d = new Date(2012, 5, 20);
>>> d.getDay();
3
```

由于星期数是从 0（星期日）开始计数的，因此，3 应该代表了星期三。所以：

```
>>> d.toString();
"Wed Jun 20 2012"
```

好吧，星期三是不错，但那显然不是一个搞派对的最佳日子。接下来我要弄一个循环，看看从 2012 年到 3012 年有多少个 6 月 20 日是星期五，并查看一下这些日子在一周当中的分布情况。（嗯，毕竟现在的医学很发达，相信大家到了 3012 年还是会精神抖擞的。）

首先，我们来初始化一个包含七个元素的数组，每个元素都分别对应着一周中的一天，以充当计数器。也就是说，在循环到 3012 年的过程中，我们将会根据执行情况递增相关的计数器：

```
var stats = [0,0,0,0,0,0,0];
```

接下来就是该循环的实现：

```
for (var i = 2012; i < 3012; i++) {  
    stats[new Date(i, 5, 20).getDay()]++;  
}
```

然后，我们来看看结果：

```
>>> stats;  
[139, 145, 139, 146, 143, 143, 145]
```

哇哦！有 143 个星期五和 145 个星期六，不错不错！

4.2.9 RegExp

正则表达式是一种强大的文本搜索和处理方式。如果您熟悉 SQL 的话，就可以将正则表达式看做一种类似于 SQL 的工具。我们可以用 SQL 来搜索并更新数据库中的数据，同样的，我们也能用正则表达式来查找并更新文本中的某个具体片段。

关于正则表达式的语法，不同的语言有着不同的实现（就像“方言”），JavaScript 所采用的是 Perl 5 的语法。

另外，为简便起见，人们经常会将“regular expression”缩写成“regex”或者“regexp”。

一个正则表达式通常由以下部分组成：

- ◆ 一个用于匹配的模式文本。
- ◆ 用 0 个或多个修饰符（也叫做标志）描述的匹配模式细节。

该匹配模式也可以是简单的全字符文本，但在这种情况下，我们多半会使用 `indexOf()` 这样的方法，很少会用到正则表达式。在大多数情况下，匹配模式往往都要更为复杂，也更难以理解。事实上，掌握正则表达式是一个很大的问题，我们也不打算在这里详细讨论它们。接下来，我们只会介绍它在 JavaScript 中的语法，以及可用于正则表达式的对象和方法。另外，我们还在附录 D 中提供了一份完整的匹配模式写法指南，以供读者参考。

在 JavaScript 中，我们通常会利用内建构造器 `RegExp()` 来创建正则表达式对象，例如：

```
>>> var re = new RegExp("j.*t");
```

另外，`RegExp` 对象还有一种更为简便的文本定义方式：

```
>>> var re = /j.*t/;
```

在上面的示例中，“`j.*t`”就是我们之前说的正则表达式模式。其具体含义是：“匹配任何以 `j` 开头、`t` 结尾的字符串，且这两个字符之间可以包含 1 个或多个字符。”。其中的 `*` 号的意思就是“0 个或多个单元”，而这里的点号 (`.`) 所表示的是“任意字符”。当然，当我们向 `RegExp` 构造器传递该模式时，还必须将它放在一对引号中。

4.2.9.1 RegExp 对象的属性

以下是一个正则表达式对象所拥有的属性：

- ◆ `global`：如果该属性值为 `false`（这也是默认值），相关搜索在找到第一个匹配位置时就会停止。如果需要找出所有的匹配位置，将其设置为 `true` 即可。
- ◆ `ignoreCase`：设置大小写相关性，默认为 `false`。
- ◆ `multiline`：设置是否跨行搜索的选项，默认为 `false`。
- ◆ `lastIndex`：搜索开始的索引位，默认值为 0。
- ◆ `source`：用于存储正则表达式匹配模式的属性。

另外，除了 `lastIndex` 外，上面所有属性在对象创建之后就都不能再被修改了。

而且，前三个属性是可以通过 `regex` 修饰符来表示的。当我们通过构造器来创建 `regex` 对象时，可以向构造器的第二参数传递下列字符中的任意组合：

- ◆ “`g`”代表 `global`。
- ◆ “`i`”代表 `ignoreCase`。
- ◆ “`m`”代表 `multiline`。

这些字符可以以任意顺序传递，只要它们被传递给了构造器，相应的修饰符就会被设置为 `true`。例如在下面的示例中，我们将所有的修饰符都设置成了 `true`：

```
>>> var re = new RegExp('j.*t', 'gmi');
```

现在来验证一下：

```
>>> re.global;
true
```

不过，这里的修饰符一旦被设置了就不能更改了：

```
>>> re.global = false;
>>> re.global
true
```

另外，我们也可以通通过文本方式来设置这种 regex 的修饰符，只需将它们加在斜线后面：

```
>>> var re = /j.*t/ig;
>>> re.global
true
```

4.2.9.2 RegExp 对象的方法

RegExp 对象中有两种可用于查找匹配内容的方法：`test()` 和 `exec()`。这两个函数的参数都是一个字符串，但 `test()` 方法返回的是一个布尔值（找到匹配内容时为 `true`，否则就为 `false`），而 `exec()` 返回的则是一个由匹配字符串组成的数组。显然，`exec()` 能做的工作更多，而 `test()` 只有在我们不需要匹配的具体内容时才会有用处。人们通常会用正则表达式来执行某些验证操作，在这种情况下往往使用 `test()` 就足够了。

例如，下面的表达式是不匹配的，因为目标是大写的 J：

```
>>> /j.*t/.test("Javascript")
false
```

如果将其改成大小写无关的，结果就正确了：

```
>>> /j.*t/i.test("Javascript")
true
```

同样的，我们也可以用测试一下 `exec()` 方法，并访问它所返回数组的首元素：

```
>>> /j.*t/i.exec("Javascript")[0]
"Javascript"
```

4.2.9.3 以正则表达式为参数的字符串方法

在本章前面，我们曾向您介绍过如何使用 `String` 对象的 `indexOf` 和 `lastIndexOf` 方法来搜索文本。但这些方法只能用于纯字符串式的搜索，如果想获得更强大的文本搜索能力就需要用到正则表达式了。`String` 对象也为我们提供了这种能力。

在 `String` 对象中，以正则表达式对象为参数的方法主要有以下这些：

- ◆ `match()` 方法：返回的是一个包含匹配内容的数组。
- ◆ `search()` 方法：返回的是第一个匹配内容所在的位置。
- ◆ `replace()` 方法：该方法能将匹配的文本替换成指定的字符串。
- ◆ `split()` 方法：能根据指定的正则表达式将目标字符串分割成若干个数组元素。

4.2.9.4 `search()`与 `match()`

下面来看一些 `search()` 与 `match()` 方法的用例。首先，我们来新建一个 `String` 对象：

```
>>> var s = new String('HelloJavaScriptWorld');
```

然后调用其 `match()` 方法，结果返回的数组中只有一个匹配对象：

```
>>> s.match(/a/);  
["a"]
```

接下来，我们对其施加 `g` 修饰符，进行 `global` 搜索，这样一来返回的数组中就有了两个结果：

```
>>> s.match(/a/g);  
["a", "a"]
```

下面进行大小写无关的匹配操作：

```
>>> s.match(/j.*a/i);  
["Java"]
```

而 `search()` 方法则会返回匹配字符串的索引位置：

```
>>> s.search(/j.*a/i);  
5
```

4.2.9.5 replace()

replace()方法可以将相关的匹配文本替换成某些其他字符串。在下面的示例中，我们移除了目标字符串中的所有大写字符（实际上是替换为空字符串）：

```
>>> s.replace(/[A-Z]/g, '');
"elloavacriptorld"
```

如果我们忽略了 g 修饰符，结果就只有首个匹配字符被替换掉：

```
>>> s.replace(/[A-Z]/, '');
"elloJavaScriptWorld"
```

当某个匹配对象被找到时，如果我们想让相关的替换字符串中包含匹配的文本，可以使用 \$& 修饰符。例如，下面我们在每一个匹配字符前面加了一个下划线：

```
>>> s.replace(/[A-Z]/g, "_$&");
"_Hello_Java_Script_World"
```

如果正则表达式中分了组（即带括号），那么可以用 \$1 来表示匹配分组中的第一组，而 \$2 则表示第二组，以此类推。

```
>>> s.replace(/([A-Z])/g, "_$1");
"_Hello_Java_Script_World"
```

假设我们的 web 页面上有一个注册表单，上面会要求用户输入 E-mail 地址，用户名和密码。当用户输入他们的 E-mail 地址时，我们可以利用 JavaScript 将 E-mail 的前半部分提炼出来，作为后面用户名字段的建议：

```
>>> var email = "stoyan@phpied.com";
>>> var username = email.replace(/(.*?)@.*/, "$1");
>>> username;
"stoyan"
```

4.2.9.6 回调式替换

当我们需要执行一些特定的替换操作时，也可以通过返回字符串的函数来完成。这样，我们就可以在执行替换操作之前实现一些必要的处理逻辑：

```
>>> function replaceCallback(match) {return "_" +
```



```
        match.toLowerCase();}
>>> s.replace(/[A-Z]/g, replaceCallback);
"_hello_java_script_world"
```

该回调函数可以接受一系列的参数（在上面的示例中，我们忽略了所有参数，但首参数是依然存在的）。

- ◆ 首参数是正则表达式所匹配的内容。
- ◆ 尾参数则是被搜索的字符串。
- ◆ 尾参数之前的参数表示的是匹配内容所在的位置。
- ◆ 剩下的参数可以是由 regex 模式所分组的所有匹配字符串组。

下面让我们来具体测试一下。首先，我们新建一个参数数组，以便以后传递给回调函数：

```
>>> var glob;
```

下一步是定义一个正则表达式，我们将 E-mail 地址分成三个匹配组，具体格式形如 something@something.something：

```
>>> var re = /(.*?)@(.*?)\.(.*?) /;
```

最后就是定义相应的回调函数了，它会接受 glob 数组中的参数，并返回相应的替换内容：

```
var callback = function(){
    glob = arguments;
    return arguments[1] + ' at ' + arguments[2] + ' dot ' +
        arguments[3];
}
```

然后我们就可以这样调用它们了：

```
>>> "stoyan@phpied.com".replace(re, callback);
"stoyan at phpied dot com"
```

下面是该回调函数返回的参数内容：

```
>>> glob
["stoyan@phpied.com", "stoyan", "phpied", "com", 0,
"stoyan@phpied.com"]
```

4.2.9.7 split()

我们之前已经了解 `split()` 方法，它能根据指定的分割字符串将我们的输入字符串分割成一个数组。下面就是我们用逗号将字符串分割的结果：

```
>>> var csv = 'one, two,three ,four';
>>> csv.split(',');
["one", " two", "three ", "four"]
```

由于上面的输入字符串中逗号前后的空格有些不一致的情况，这导致生成的数组也会出现多余的空格。如果我们使用正则表达式，就可以在这里用 “`\s*`” 修饰符来解决，意思就是“匹配 0 个或多个空格”：

```
>>> csv.split(/\s*,\s*/)
["one", "two", "three", "four"]
```

4.2.9.8 用字符串来代替过于简单的 `regexp` 对象

关于我们刚刚讨论的四个方法（`split()`、`match()`、`search()` 和 `replace()`），还有最后一件事不得不提，即这些方法可以接受的参数不仅仅是一些正则表达式，也包括字符串。它们会将接收到的字符串参数自动转换成 `regex` 对象，就像我们直接传递 `new RegExp()` 一样。

例如，下面是用字符串执行替换的：

```
>>> "test".replace('t', 'r')
"rest"
```

但它与下面的调用是等价的：

```
>>> "test".replace(new RegExp('t'), 'r')
"rest"
```

当然，在执行这种字符串传递时，我们就不能像平时使用构造器或者 `regex` 文本法那样设置表达式修饰符了。

4.2.10 Error 对象

当我们的代码中有错误发生时，一个好的处理机制可以帮助我们理解错误发生的条件，并能以一种较为优雅的方式来纠正错误。在 JavaScript 中，我们将会使用 `try`、`catch` 及

`finally` 语句组合来处理错误。当程序中出现错误时，就会抛出一个 `Error` 对象，该对象可能由以下几个内建构造器中的一个产生而成，它们包括 `EvalError`、`RangeError`、`ReferenceError`、`SyntaxError`、`TypeError` 和 `URIError` 等，所有这些构造器都继承自 `Error` 对象。

下面，我们来主动触发一个错误，看看会发生些什么。假设下面的示例中调用了一个并不存在的函数，例如我们在 `Firebug` 控制台中输入的是：

```
>>> iDontExist();
```

我们就会看到这样的内容（见图 4-4）：

并且，我们也会看到右下角的 `Firebug` 通用图标发生了变化（见图 4-5）：

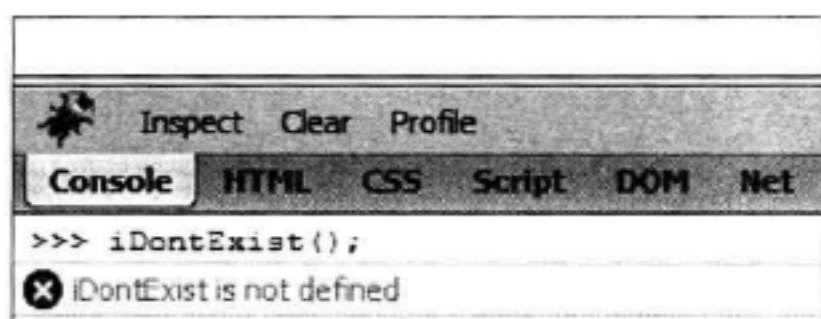


图 4-4

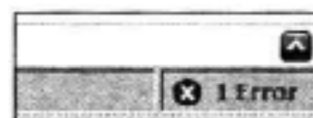


图 4-5

如果我们打开 `Firefox` 的错误控制台（单击工具（`Tools`）菜单，并选择错误控制台（`Error Console`）），就会看到以下内容（见图 4-6）：

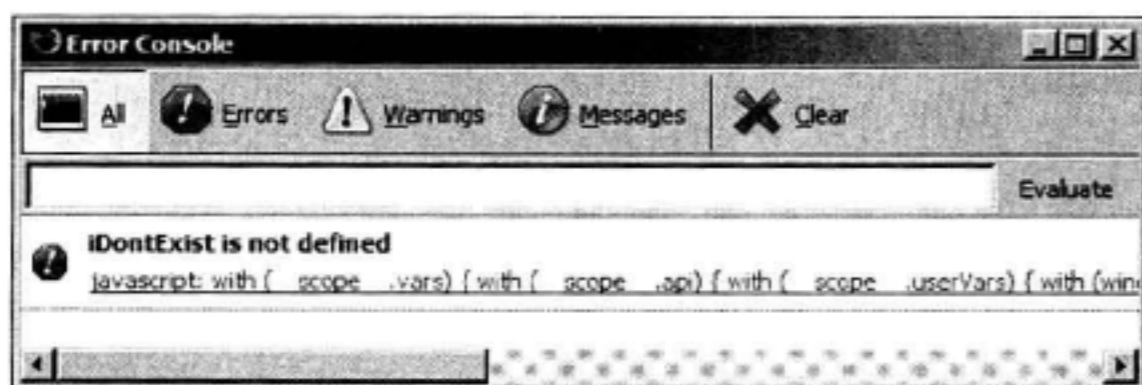


图 4-6

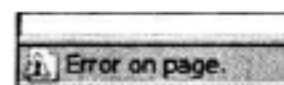


图 4-7

错误显示的方式在各浏览器和宿主环境中差别可能会很大。例如在 `Internet Explorer` 中，此类信息会出现在窗口的右下角（见图 4-7）。

如果双击这条消息，我们就会看到更详细的信息：

尽管一些浏览器的配置允许我们忽略掉某些错误，但不能因此就假设我们所有的用户都会屏蔽错误显示。由于他们没有这方面的经验，我们也有责任替他们处理掉这些错误（见图 4-8）。如果我们没有在代码中捕获（`catch`）这些错误，它们就会被直接上抛给用户，这会使错误显得不可预测并难以处理。幸运的是，错误捕获很容易，只需要我们使用 `try`

语句和 `catch` 语句即可。

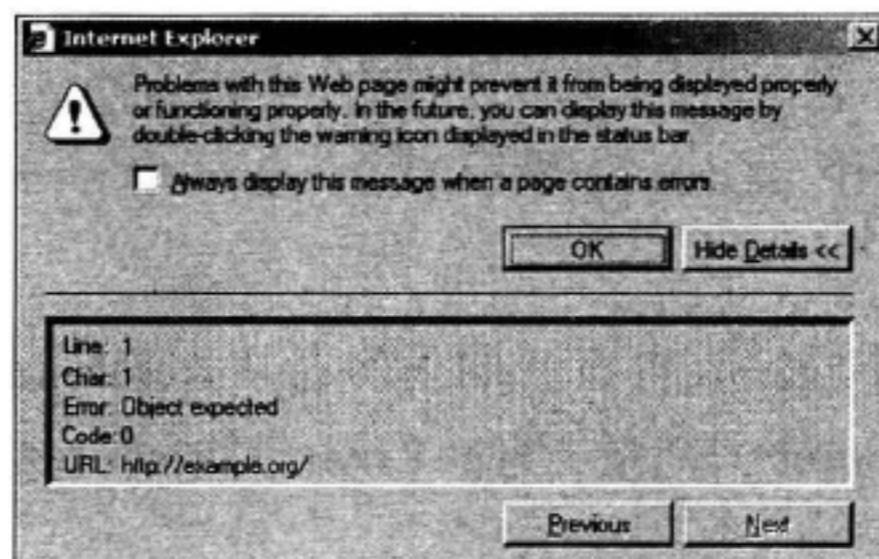


图 4-8

例如在下面代码中，我们就不会看到之前截图中的那些错误显示了：

```
try {
    iDontExist();
} catch (e) {
    // do nothing
}
```

如您所见，这里包含以下内容：

- ◆ `try` 语句及其代码块。
- ◆ `catch` 语句及其参数变量和代码块。

另外，`finally` 语句并没有在这个例子中出现，这是一个可选项，主要用于执行一些无论如何（无论有没有错误发生）都要执行的内容。

在上面的示例中，我们并没有在 `catch` 语句后面的代码块中写入任何内容，但实际上我们可以在这里加入一些用于修复错误的代码，或者至少可以将该应用程序错误的一些特定情况反馈给用户。

`catch` 语句的参数（括号中的）实际上是一个 `Error` 对象。跟其他对象一样，它也提供一系列有用的方法与属性。遗憾的是，不同的浏览器对于这些方法与属性都有着各自不同的实现，但其中有两个属性的实现还是基本相同的，那就是 `e.name` 和 `e.message`。

现在，让我们来看看这段代码：

```
try {
    iDontExist();
} catch (e) {
    alert(e.name + ': ' + e.message);
}
```

```
    } finally {  
        alert('Finally!');  
    }  
}
```

如您所见，这里的第一个 `alert()` 显示了 `e.name` 和 `e.message`，而后一个则显示了“**Finally!**”字样。

在 Firefox 中，第一个 `alert()` 将显示的内容是“**ReferenceError: iDontExist is not defined**”。而在 Internet Explorer 中则是“**TypeError: Object expected**”。总之，这里向我们传递了两个信息：

- ◆ `e.name` 所包含的是构造当前 Error 对象的构造器名称。
- ◆ 由于 Error 对象在各宿主环境（浏览器）中的表现并不一致，因此在这里我们需要使用一些技巧，以便我们的代码能处理各种类型的错误（即 `e.name` 的值）。

当然，我们也用 `new Error()` 或者其他 Error 对象构造器来自定义一个 Error 对象，然后告诉 JavaScript 引擎某个特定的条件，并使用 `throw` 语句来抛出该对象。

下面来看一个具体的示例，假设我们需要调用一个 `maybeExists()` 函数，该函数会执行某些求和运算。在这种情况下，如果我们想统一错误处理方式的话，那么无论 `maybeExists()` 函数是否存在，或者求和运算是否会出现问题，我们的代码都应该这样写：

```
try {  
    var total = maybeExists();  
    if (total === 0) {  
        throw new Error('Division by zero!');  
    } else {  
        alert(50 / total);  
    }  
} catch (e) {  
    alert(e.name + ': ' + e.message);  
} finally {  
    alert('Finally!');  
}
```

根据 `maybeExists()` 函数的存在与否及其返回值，这段代码会弹出几种不同的信息：

- ◆ 如果 `maybeExists()` 函数不存在，我们在 Firefox 中将会得到的信息为“**ReferenceError: maybeExists() is not defined**”，而在 IE 中则为“**TypeError: Object expected**”。
- ◆ 如果 `maybeExists()` 返回值为 0，我们将得到的信息是“**Error: Division by zero!**”。

- ◆ 如果 `maybeExists()` 的返回值为 2，我们将得到的 `alert` 信息是 25.0
- ◆ 在以上所有的情况下，程序都会弹出第二个 `alert` 窗口，内容为“**Finally!**”。

另外，这里抛出的是 `new Error('Division by zero!')` 而不是一般性的 `Error` 对象了，这样一来，我们可以根据自身的需要来做更多的设定。例如可以利用 `throw new RangeError('Division by zero!')` 语句来抛出，或者不用任何构造器，直接定义一个一般对象抛出：

```
throw {
  name: "MyError",
  message: "OMG! Something terrible has happened"
}
```

4.3 本章小结

在第 2 章中，我们学习了 JavaScript 中的五大基本数据类型（`number`、`string`、`boolean`、`null` 和 `undefined`），而且，我们也说过除这些基本类型以外的任何数据都属于对象。在本章，我们又了解了以下内容：

- ◆ 对象与数组很类似，但允许我们指定键值。
- ◆ 对象通常都会拥有若干个属性。
- ◆ 其中有些属性可以当做函数使用（函数本身也是数据，例如 `var f = function() {};`）。这些属性通常称为方法。
- ◆ 数组本身也可以看做拥有一系列数字属性，并外加一个 `length` 属性的对象。
- ◆ `Array` 对象中有着一系列非常有用的方法（例如 `sort()` 或 `slice()`）。
- ◆ 函数也是一种对象，它们本身也有属性（例如 `length` 和 `prototype`）和方法（例如 `call()` 和 `apply()`）。

对于五种基本数据类型，除了 `undefined`（这本来就不代表任何东西）和 `null`（这本身也是个对象）外，其他三个都有相应的构造器函数，分别是 `Number()`、`String()` 以及 `Boolean()`。通过这些我们可以创建出相应的对象。通过将 these 基本类型封装成对象，我们就可以在其中集成一些有用的工作方法。

`Number()`、`String()` 以及 `Boolean()` 的调用可分为两种形式：

- ◆ 使用 `new` 操作符调用——用于新建对象。

- ◆ 不使用 `new` 操作符调用——用于将任意值转换成基本数据类型。

此外，我们还学习了一系列内建构造器函数，其中包括 `Object()`、`Array()`、`Function()`、`Date()`、`RegExp()` 和 `Error()`，以及不属于构造器的 `Math` 对象。

现在，我们应该明白对象在 JavaScript 程序设计中的中心地位，几乎所有的东西都是对象或者可以封装成对象的。

最后，让我们再来熟悉一下对象的文本标识法（见表 4-2）。

表 4-2

名称	文本记法	构造器	相关示例
对象	{}	<code>new Object()</code>	{prop: 1}
数组	[]	<code>new Array()</code>	[1,2,3,'test']
正则表达式	/pattern/modifiers	<code>new RegExp('pattern', 'modifiers')</code>	/java.*/img

4.4 练习题

1. 请看下列代码：

```
function F() {
  function C() {
    return this;
  }
  return C();
}
var o = new F();
```

请问上面的 `this` 值指向的是全局对象还是对象 `o`？

2. 下面代码的执行结果会是什么？

```
function C(){
  this.a = 1;
  return false;
}
console.log(typeof new C());
```

3. 下面这段代码的执行结果又将是什么？

```
>>> c = [1, 2, [1, 2]];
```

```
>>> c.sort();
>>> c.join('--');
>>> console.log(c);
```

4. 在 `String()` 构造器不存在的情况下自定义一个 `MyString()` 的构造器函数。记住，由于 `String()` 不存在，因此您在写该构造器函数时不能使用任何属于内建 `String` 对象的方法和属性。并且要让您所创建的对象通过以下测试：

```
>>> var s = new MyString('hello');
>>> s.length;
5
>>> s[0];
"h"
>>> s.toString();
"hello"
>>> s.valueOf();
"hello"
>>> s.charAt(1);
"e"
>>> s.charAt('2');
"l"
>>> s.charAt('e');
"h"
>>> s.concat(' world!');
"hello world!"
>>> s.slice(1,3);
"el"
>>> s.slice(0,-1);
"hell"
>>> s.split('e');
["h", "llo"]
>>> s.split('l');
["he", "", "o"]
```



我们可以将输入字符串当做一个数组，用 `for-in` 循环来进行遍历。

5. 更新上面的 `MyString()` 构造器，为其添加一个 `reverse()` 方法。



可以参照数组中 `reverse()` 的方法来实现。

6. 在 `Array()` 构造器以及相关的数组文本标识法都不存在的情况下，自定义一个类似的 `MyArray()` 构造器，并令其通过以下测试：

```
>>> var a = new MyArray(1,2,3,"test");
>>> a.toString();
"1,2,3,test"
>>> a.length;
4
>>> a[a.length - 1]
"test"
>>> a.push('boo');
5
>>> a.toString();
"1,2,3,test,boo"
>>> a.pop();
[1, 2, 3, "test"]
>>> a.toString();
"1,2,3,test"
>>> a.join(',')
"1,2,3,test"
>>> a.join(' isn\'t ')
"1 isn't 2 isn't 3 isn't test"
```

如果您觉得这个练习很有趣，可以不用止步于 `join()` 方法，继续为其创建尽可能多的方法。

7. 在 `Math` 对象不存在的情况下，创建一个类似的 `MyMath` 对象，并为其添加以下方法：

- ◆ `MyMath.rand(min, max, inclusive)`——随机返回 `min` 到 `max` 区间中的一个数，并且在 `inclusive` 为 `true` 时为闭区间（这也是默认情况）。
- ◆ `MyMath.min(array)`——返回目标数组中的最小值。
- ◆ `MyMath.max(array)`——返回目标数组中的最大值。

第 5 章

原型

在本章中，我们将着重介绍函数对象中的原型（prototype）属性。对于 JavaScript 的学习来说，理解原型的工作原理是非常重要的的一环，毕竟，它的对象模型是完全基于原型的。当然，原型其实并不是很难理解，只不过由于这是一个全新的概念，我们接受起来需要一定的时间而已。事实上这些东西在 JavaScript 中几乎无处不在（闭包除外），只要我们“领悟”了其中的原理，一切都会显得格外的简单明了。而且在之后的内容中，本书还会围绕着原型展开大量的示例演示，以便我们巩固并加深对这一概念的熟悉程度。

本章将要讨论以下话题。

- ◆ 每个函数中都有一个 prototype 属性，该属性所存储的就是原型对象。
- ◆ 为原型对象添加属性。
- ◆ 使用原型对象中的新增属性。
- ◆ 区分对象自身属性与原型属性。
- ◆ `__proto__`，用于保存各对象原型的神秘链接。
- ◆ 介绍原型方法，例如 `isPrototypeOf()`、`hasOwnProperty()`、`propertyIsEnumerable()`。
- ◆ 如何（利用原型）强化数组或字符串这样的内建对象。

5.1 原型属性

在 JavaScript 中，函数本身也是一个包含了方法和属性的对象。经过之前的学习，相信我们对它的一些方法（如 `apply()` 和 `call()`）及属性（如 `length` 和 `constructor`）并不

陌生。接下来，我们要介绍的是函数对象的另一个属性——`prototype`。

众所周知，只要我们像下面这样简单地定义一个函数 `foo()`，就可以像访问其他对象一样访问该函数的属性：

```
>>> function foo(a, b){return a * b;}
>>> foo.length
2
>>> foo.constructor
Function()
```

其实这些（在函数定义时被创建的）属性中就包括了 `prototype` 属性，它的初始值是一个空对象。

```
>>> typeof foo.prototype
"object"
```

当然，我们也可以自己添加该属性，就像这样：

```
>>> foo.prototype = {}
```

我们也可以赋予这个空对象一些方法和属性，这并不会对 `foo` 函数本身造成什么影响，只是不能被用做构造器罢了。

5.1.1 利用原型添加方法与属性

在第 4 章中，我们已经学习了如何定义构造器函数，并用它来新建（构造）对象。这种做法的主要意图是通过 `new` 操作符来调用函数，以达到访问对象 `this` 值的目的，然后，构造器就可以将其所创建的对象返回给我们。这样，我们就有了一种赋予新建对象一定功能（即为其添加属性和方法）的方法。

下面，让我们通过一个具体的构造器函数 `Gadget()`，来看看它是如何在新建对象时添加属性与方法的。

```
function Gadget(name, color) {
  this.name = name;
  this.color = color;
  this.whatAreYou = function() {
    return 'I am a ' + this.color + ' ' + this.name;
  }
}
```

添加属性和方法还有另一种方式，即通过构造器函数的 `prototype` 属性来增加该构造器所能提供的功能。现在就让我们来为上面的构造器增加两个属性（即 `price` 和 `rating`）和一个方法（即 `getInfo()`）。由于 `prototype` 属性包含的是一个对象，所以我们可以这样：

```
Gadget.prototype.price = 100;
Gadget.prototype.rating = 3;
Gadget.prototype.getInfo = function() {
    return 'Rating: ' + this.rating + ', price: ' + this.price;
};
```

如果不想将它们逐一添加到原型对象中去，也可以另外定义一个对象，然后将其覆盖到之前的原型上：

```
Gadget.prototype = {
    price: 100,
    rating: 3,
    getInfo: function() {
        return 'Rating: ' + this.rating + ', price: ' + this.price;
    }
};
```

5.1.2 使用原型的方法与属性

在向原型中添加完所有的方法和属性后，就可以直接用该构造器来新建对象了。例如，我们用构造器 `Gadget()` 来新建一个 `newtoy` 对象，然后，就可以访问之前所定义的那些属性和方法了。

```
>>> var newtoy = new Gadget('webcam', 'black');
>>> newtoy.name;
"webcam"
>>> newtoy.color;
"black"
>>> newtoy.whatAreYou();
"I am a black webcam"
>>> newtoy.price;
100
>>> newtoy.rating;
3
>>> newtoy.getInfo();
"Rating: 3, price: 100"
```

对于原型来说，最重要的是我们要理解它的“驻留”（live）概念。由于在 JavaScript 中，对象都是通过传引用的方式来传递的，因此我们所创建的每个新对象实体中并没有一份属于自己原型副本。知道这意味着什么吗？这就是说，我们可以随时修改原型，并且与之相关的对象也都会继承这一改变（甚至可能会影响在修改之前就已经创建了的对象）。

下面继续之前的例子，让我们再向原型中添加一个新方法：

```
Gadget.prototype.get = function(what) {  
    return this[what];  
};
```

然后我们就会看到，即便 newtoy 对象在 get() 方法定义之前就已经被创建了，但我们依然可以在该对象中访问新增的方法：

```
>>> newtoy.get('price');  
100  
>>> newtoy.get('color');  
"black"
```

5.1.3 自身属性与原型属性

在之前关于 getInfo() 的那个示例中，我们是使用 this 指针来完成对象访问的，但其实直接引用 Gadget.prototype 也可以完成同样的操作：

```
Gadget.prototype.getInfo = function() {  
    return 'Rating: ' + Gadget.prototype.rating + ', price: ' + Gadget.  
prototype.price;  
};
```

这难道有什么不同吗？要回答这个问题，就需要更深入地理解原型的工作原理。

让我们再回到之前的那个 newtoy 对象：

```
>>> var newtoy = new Gadget('webcam', 'black');
```

当我们访问 newtoy 的某个属性，例如 newtoy.name 时，JavaScript 引擎就会遍历该对象的所有属性，并查找一个叫做 name 的属性。如果找到了就会立即返回其值。

```
>>> newtoy.name  
"webcam"
```

那么，如果我们访问 `rating` 属性又会发生什么呢？JavaScript 依然会查询 `newtoy` 对象的所有属性，但这一回它找不到一个叫 `rating` 的属性了。接下来，脚本引擎就会去查询用于创建当前对象的构造器函数的原型（等价于我们直接访问 `newtoy.constructor.prototype`）。如果在原型中找到了该属性，就立即使用该属性。

```
>>> newtoy.rating
3
```

这种方式与直接访问原型属性是一样的。每个对象都有属于自己的构造器属性，其所引用的就是用于创建该对象的那个函数，所以我们可以像下面这样：

```
>>> newtoy.constructor
Gadget(name, color)
>>> newtoy.constructor.prototype.rating
3
```

现在，我们再来仔细想想这一过程。每个对象都会有一个构造器，而原型本身也是一个对象，这意味着它也必须有一个构造器，而这个构造器又会有自己的原型。换句话说，我们可以这样做：

```
>>> newtoy.constructor.prototype.constructor
Gadget(name, color)
>>> newtoy.constructor.prototype.constructor.prototype
Object price=100 rating=3
```

这个结构可能一直会持续下去，并最终取决于原型链（`prototype chain`）的长度，但其最后一环肯定是 `Object` 内建对象，因为它是最高级的父级对象。实际上，如果我们调用的是 `newtoy.toString()`，那么在 `newtoy` 对象及其原型中都不会找到 `toString()` 方法。最后我们调用的只能是 `Object` 对象的 `toString()` 方法。

```
>>> newtoy.toString()
"[object Object]"
```

5.1.4 利用自身属性重写原型属性

通过上面的讨论，我们知道如果在一个对象自身属性中没有找到指定的属性，就可以去（如果存在的话）原型链中查找相关的属性。但是，如果遇上对象的自身属性与原型属性同名又该怎么办呢？答案是对象自身属性的优先级高于原型属性。

让我们来看一个具体的示例，即同一个属性名同时出现在对象的自身属性和原型

属性中：

```
function Gadget(name) {
  this.name = name;
}
Gadget.prototype.name = 'foo';
"foo"
```

然后我们新建一个对象，并访问该对象自身的 name 属性：

```
>>> var toy = new Gadget('camera');
>>> toy.name;
"camera"
```

这时候，如果我们删除这个属性，同名的原型属性就会“浮出水面”：

```
>>> delete toy.name;
true
>>> toy.name;
"foo"
```

当然，我们随时都可以重建这个对象的自身属性：

```
>>> toy.name = 'camera';
>>> toy.name;
"camera"
```

枚举属性

如果想获得某个对象所有属性的列表，我们可以使用 for-in 循环。在第 2 章中，我们已经知道了如何使用该循环来遍历数组中的所有元素：

```
var a = [1, 2, 3];
for (var i in a) {
  console.log(a[i]);
}
```

而数组本身就是对象，因此我们可以用同样的 for-in 循环来遍历对象：

```
var o = {p1: 1, p2: 2};
for (var i in o) {
  console.log(i + '=' + o[i]);
}
```

产生的结果如下：

```
p1=1  
p2=2
```

在这里，有些细节需要留意。

- ◆ 并不是所有的属性都会在 `for-in` 循环中显示。例如（数组的）`length` 属性和 `constructor` 属性就不会被显示。那些已经被显示的属性被称为是可枚举的，我们可以通过各个对象所提供的 `propertyIsEnumerable()` 方法来判断其中有哪些可枚举的属性。
- ◆ 原型链中的各个原型属性也会被显示出来，当然前提是它们是可枚举的。我们可以通过对象的 `hasOwnProperty()` 方法来判断一个属性是对象自身属性还是原型属性。
- ◆ 对于所有的原型属性，`propertyIsEnumerable()` 都会返回 `false`，包括那些在 `for-in` 循环中可枚举的属性。

下面来看看这些方法具体是如何使用的。首先，我们来定义一个简化版的 `Gadget()`：

```
function Gadget(name, color) {  
    this.name = name;  
    this.color = color;  
    this.someMethod = function(){return 1;}  
}  
Gadget.prototype.price = 100;  
Gadget.prototype.rating = 3;
```

然后新建一个对象：

```
var newtoy = new Gadget('webcam', 'black');
```

现在，如果对它执行 `for-in` 循环，就会列出该对象中的所有属性，包括原型中的属性：

```
for (var prop in newtoy) {  
    console.log(prop + ' = ' + newtoy[prop]);  
}
```

其结果甚至包括该对象的方法（因为方法本质上也可以被视为是函数类型的属性）：


```
name = webcam
color = black
someMethod = function () { return 1; }
price = 100
rating = 3
```

如果要对对象属性和原型属性做一个区分，就需要调用 `hasOwnProperty()` 方法，我们可以先来试一下：

```
>>> newtoy.hasOwnProperty('name')
true
>>> newtoy.hasOwnProperty('price')
false
```

下面我们再来循环一次，不过这次只显示对象的自身属性：

```
for (var prop in newtoy) {
  if (newtoy.hasOwnProperty(prop)) {
    console.log(prop + '=' + newtoy[prop]);
  }
}
```

结果为：

```
name=webcam
color=black
someMethod=function () { return 1; }
```

现在我们来试试 `propertyIsEnumerable()`，该方法会对所有的非内建对象属性返回 `true`：

```
>>> newtoy.propertyIsEnumerable('name')
true
```

而对于内建属性和方法来说，它们大部分都是不可枚举的：

```
>>> newtoy.propertyIsEnumerable('constructor')
false
```

另外，任何来自原型链中的属性也是不可枚举的：

```
>>> newtoy.propertyIsEnumerable('price')
false
```

但是需要注意的是，如果 `propertyIsEnumerable()` 的调用是来自原型链上的某个对象，那么该对象中的属性是可枚举的。

```
>>> newtoy.constructor.prototype.propertyIsEnumerable('price')
true
```

5.1.5 isPrototypeOf()方法

每个对象中都会有一个 `isPrototypeOf()` 方法，这个方法会告诉我们当前对象是否是另一个对象的原型。

让我们先来定义一个简单的对象 `monkey`：

```
var monkey = {
  hair: true,
  feeds: 'bananas',
  breathes: 'air'
};
```

然后，我们再创建一个叫做 `Human()` 的构造器函数，并将其原型属性设置为指向 `monkey`：

```
function Human(name) {
  this.name = name;
}
Human.prototype = monkey;
```

现在，如果我们新建一个叫做 `George` 的 `Human` 对象，并向它提问“`monkey` 是 `george` 的原型吗？”，答案是 `true`。

```
>>> var george = new Human('George');
>>> monkey.isPrototypeOf(george)
true
```

5.1.6 神秘的__proto__链接

现在，我们已经了解了当我们访问一个在当前对象中不存在的属性时，相关的原型属性就会被纳入查询范围。

下面让我们再回到那个用 `monkey` 对象做原型的 `Human()` 对象构造器。

```
var monkey = {
```

```
    feeds: 'bananas',  
    breathes: 'air'  
};  
function Human() {}  
Human.prototype = monkey;
```

这次我们来创建一个 `developer` 对象，并赋予它一些属性：

```
var developer = new Human();  
developer.feeds = 'pizza';  
developer.hacks = 'JavaScript';
```

接着，我们来访问一些属性，例如 `developer` 对象的 `hacks` 属性：

```
>>> developer.hacks  
"JavaScript"
```

当然，`feeds` 也一样可以在该对象中找到：

```
>>> developer.feeds  
"pizza"
```

但 `breathes` 在 `developer` 对象自身的属性中是不存在的，所以就得出原型中查询，就好像其中有一个神秘的链接指向了相关的原型对象。

```
>>> developer.breathes  
"air"
```

我们能自己从 `developer` 对象中获得相关的原型对象吗？好吧，我们确实可以做到，只要将构造器作为一个中转站，执行类似 `developer.constructor.prototype` 这样的调用，确实可以得到 `monkey` 对象。但问题在于这种做法非常不可靠，因为构造器中有很多复杂的信息，而且随时都有可能被重写，甚至将它重写成一个非对象属性也不会影响原型链本身的功能。

例如，我们将其 `constructor` 属性设置为一个字符串：

```
>>> developer.constructor = 'junk'  
"junk"
```

这看起来似乎将原型搞得一团糟：

```
>>> typeof developer.constructor.prototype  
"undefined"
```

但事实并非如此，因为我们依然能访问 `developer` 的 `breathes` 属性，并返回“**air**”：

```
>>> developer.breathes
"air"
```

这一切都显示了对象中确实存在一个指向相关原型的链接，这个神秘的链接在 Firefox 中被叫做 `__proto__` 属性（“`proto`”这个词的两边各有两条下划线）。

```
>>> developer.__proto__
Object feeds=bananas breathes=air
```

当然，出于学习的目的来调用这种神秘的属性是无可厚非的，但如果是在实际的脚本编写中，这并不是一个好主意。因为该属性在 Internet Explorer 这样的浏览器中是不存在的，因此脚本就不能实现跨平台了。下面我们来看一个具体的示例。假设我们创建了一系列以 `monkey` 对象为原型的对象，而现在我们希望对这些对象做一些共同的改变。在这种情况下，我们可以从改变 `monkey` 着手，并令其他对象实体来继承这一改变。

```
>>> monkey.test = 1
1
>>> developer.test
1
```

另外需要提示的是，`__proto__` 与 `prototype` 并不是等价的。`__proto__` 实际上是某个实体对象的属性，而 `prototype` 则是属于构造器函数的属性。

```
>>> typeof developer.__proto__
"object"
>>> typeof developer.prototype
"undefined"
```

千万要记住，`__proto__` 只能在学习或调试的环境下使用。

5.2 扩展内建对象

在 JavaScript 中，内建对象的构造器函数（例如 `Array`、`String`、`Object` 和 `Function`）都是可以通过其原型来进行扩展的。这意味着我们可以做一些事情，例如只要往数组原型中添加新的方法，就可以使其在所有的数组可用。下面，我们就来试试看。

PHP 中有一个叫做 `in_array()` 的函数，主要用于查询数组中是否存在某个特定的值。

JavaScript 中则没有一个叫做 `inArray()` 的方法, 因此, 下面我们通过 `Array.prototype` 来实现一个。

```
Array.prototype.inArray = function(needle) {
  for (var i = 0, len = this.length; i < len; i++) {
    if (this[i] === needle) {
      return true;
    }
  }
  return false;
}
```

现在, 所有的数据对象都拥有了一个新方法, 我们来测试一下:

```
>>> var a = ['red', 'green', 'blue'];
>>> a.inArray('red');
true
>>> a.inArray('yellow');
false
```

这很简单! 我们可以再做一次。假设我们的应用程序需要一个反转字符串的功能, 并且也觉得 `String` 对象应该有一个 `reverse()` 方法, 毕竟数组是有 `reverse()` 方法的。其实, 在 `String` 的原型中添加一个 `reverse()` 方法也很容易, 我们可以借助于 `Array.prototype.reverse()` 方法 (这与第 4 章中的某道练习题很相似)。

```
String.prototype.reverse = function() {
  return Array.prototype.reverse.apply(this.split('')).join('');
}
```

在这段代码中, 我们实际上是先利用 `split()` 方法将目标字符串转换成数组, 然后再调用该数组的 `reverse()` 方法产生一个反向数组。最后通过 `join()` 方法将结果数组转换为字符串。下面我们来测试一下这个新方法。

```
>>> "Stoyan".reverse();
"nayotS"
```

5.2.1 关于扩展内建对象的讨论

由于通过原型来扩展内建对象是一项非常强大的技术, 有了它, 我们几乎可以随心所欲地重塑 JavaScript 语言的能力。但也正是由于它有如此强大的威力, 我们在选择使用这

项能力时就必须慎之又慎。

例如，有一个非常流行的 JavaScript 库叫做 Prototype，该库的作者就很喜欢这种方法，以至于直接用它作为库的名字。通过这个库，我们可以像使用 Ruby 语言一样使用 JavaScript 的方法。

而另一个流行 JavaScript 库 YUI (Yahoo! User Interface) 库作者的观点则正好相反。他们不希望以任何形式修改语言的内建对象。原因在于一旦人们已经掌握了 JavaScript 这门语言，总是希望无论在什么库环境下，语言都能以相同的方式工作，如果随意修改其核心对象，就有可能使该库的用户因无法做出正确的预估而导致某些不必要的错误。

事实上，随着 JavaScript 自身的变化以及浏览器新版本的不断出现，其支持的功能会越来越多，没准我们今天所缺失的，想通过原型来扩展的功能，明天就会出现在内建方法中。在这种情况下，我们设计的方法就没有了长期的需求性。然而，我们已经编写了大量的方法代码，那么这些方法是否与那些新增的内建方法实现存在着一些细微的不同呢？

最后，我们可以在实现某个方法时先检测一下是否有现成的方法存在，例如在最近的那个示例中，代码应该这样写：

```
if (!String.prototype.reverse) {
  String.prototype.reverse = function() {
    return Array.prototype.reverse.apply(this.split('')).join('');
  }
}
```



最佳实践

如果您想要通过原型为某个对象添加一个新属性，务必先检查一下该属性是否已经存在。

5.2.2 一些原型陷阱

在处理原型问题时，我们需要特别注意以下两种行为。

- ◆ 当我们对原型对象执行完全替换时，可能会触发原型链中某种异常 (exception)。
- ◆ `prototype.constructor` 属性是不可靠的。

下面，我们来新建一个简单的构造器函数，并用它再创建两个对象：

```
>>> function Dog(){this.tail = true;}
>>> var benji = new Dog();
>>> var rusty = new Dog();
```

即便在对象创建之后，我们也依然能为原型添加属性，并且所有的相关对象都可以随时访问这些新属性。现在，让我们放一个 `say()` 方法进去：

```
>>> Dog.prototype.say = function(){return 'Woof!';}
```

这样，上面的两个对象都可以访问该新方法了：

```
>>> benji.say();  
"Woof!"  
>>> rusty.say();  
"Woof!"
```

如果我们总结一下这些对象的情况，就会发现到目前为止该构造器函数只是被用来新建对象，因此一切正常。

```
>>> benji.constructor;  
Dog()  
>>> rusty.constructor;  
Dog()
```

但值得注意的是，如果我们这里访问的是该原型对象的构造器，返回的也是 `Dog()`。这就不太对了，因为这时候它的原型对象应该是一个由 `Object()` 创建的一般对象，并不拥有 `Dog()` 所构造的对象所拥有的属性。

```
>>> benji.constructor.prototype.constructor  
Dog()  
>>> typeof benji.constructor.prototype.tail  
"undefined"
```

现在，我们用一个自定义的新对象完全覆盖掉原有的原型对象：

```
>>> Dog.prototype = {paws: 4, hair: true};
```

事实证明，这确实会使原有对象不能访问原型的新增属性，但它们依然能通过那个神秘的链接与原有的原型对象保持联系。

```
>>> typeof benji.paws  
"undefined"  
>>> benji.say()  
"Woof!"  
>>> typeof benji.__proto__.say
```

```
"function"
>>> typeof benji.__proto__.paws
"undefined"
```

而我们之后创建的所有对象使用的都是被更新后的 prototype 对象。

```
>>> var lucy = new Dog();
>>> lucy.say()
TypeError: lucy.say is not a function
>>> lucy.paws
4
```

并且，其秘密链接 __proto__ 也指向了新的 prototype 对象：

```
>>> typeof lucy.__proto__.say
"undefined"
>>> typeof lucy.__proto__.paws
"number"
```

但这时候，新对象的 constructor 属性就不能再保持正确了，原本应该是 Dog() 的引用却指向了 Object()。

```
>>> lucy.constructor
Object()
>>> benji.constructor
Dog()
```

而其中最令人困惑的部分则发生在我们查看其构造器的 prototype 属性时：

```
>>> typeof lucy.constructor.prototype.paws
"undefined"
>>> typeof benji.constructor.prototype.paws
"number"
```

当然，我们可以通过下面两行代码来解决上述所有的异常行为：

```
>>> Dog.prototype = {paws: 4, hair: true};
>>> Dog.prototype.constructor = Dog;
```



最佳实践

当我们重写某对象的 prototype 时，重置相应的 constructor 属性是一个好习惯。

5.3 本章小结

现在，让我们来总结一下本章所讨论的几个最重要的话题。

- ◆ 在 JavaScript 中，所有函数都会拥有一个叫做 `prototype` 的属性，默认初始值为空对象。
- ◆ 我们可以在相关的原型对象中添加新的方法和属性，甚至可以用自定义对象来完全替换掉原有的原型对象。
- ◆ 当我们通过某个构造器函数来新建对象时（使用 `new` 操作符），这些对象就会自动拥有一个指向各自 `prototype` 属性的神秘链接，并且可以通过它来访问相关原型对象的属性。
- ◆ 对象自身属性的优先级要高于其原型对象中的同名属性。
- ◆ 我们可以通过 `hasOwnProperty()` 方法来区分对象自身属性和原型属性。
- ◆ 原型链的存在：如果我们在一个对象 `foo` 中访问一个并不存在的属性 `bar`，即当我们访问 `foo.bar` 时，JavaScript 引擎就会搜索该对象的原型的 `bar` 属性。如果依然没有找到 `bar` 属性，则会继续搜索其原型的原型，以此类推，直到最高级的父级对象 `Object`。
- ◆ 我们可以对内建的构造器函数进行扩展，以便所有的对象都能引用我们添加的功能。如果将某个函数赋值给 `Array.prototype.flip`，所有的数组对象都能立即增添一个 `flip()` 方法，如 `[1,2,3].flip()`。另外，在添加相关的方法和属性之前，应该做一些对已有方法的检测工作，这将会大大增加脚本对于未来环境的适应能力。

5.4 练习题

1. 创建一个名为 `shape` 的对象，并为该对象设置一个 `type` 属性和一个 `getType()` 方法。
2. 定义一个原型为 `shape` 的 `Triangle()` 构造器函数，用 `Triangle()` 创建的对象应该具有三个对象属性——`a`、`b`、`c`，分别用于表示三角形的三条边。
3. 在对象原型中添加一个名为 `getPerimeter()` 的新方法。
4. 使用下面的代码来测试您之前的实现：

```
>>> var t = new Triangle(1, 2, 3);
>>> t.constructor
Triangle(a, b, c)
>>> shape.isPrototypeOf(t)
true
>>> t.getPerimeter()
6
>>> t.getType()
"triangle"
```

5. 用循环遍历对象 `t`，列出其所有的属性和方法（不包括原型部分的）。
6. 修改上面的实现，使其能在下面的代码中正常工作：

```
>>> [1,2,3,4,5,6,7,8,9].shuffle()
[2, 4, 1, 8, 9, 6, 5, 3, 7]
```

第 6 章

继承

如果回顾一下我们在第 1 章中所讨论的内容，就会发现，我们当时所列出的、有关 JavaScript 中面向对象程序设计的各项话题，现在几乎都已经涉及了。我们了解了对象、方法与属性。我们也知道了 JavaScript 中没有类的概念，但可以用构造器函数来替代。有封装吗？显然有，对象本身就包括数据以及与这些数据有关的行为（即方法）。有聚合吗？当然，一个对象中可以包含其他对象，事实上也一直如此，因为对象方法是靠函数来实现的，而函数本身就是对象。下面，就让我们把焦点转移到有关继承（inheritance）的部分吧。毕竟这也是个非常重要的特性，正因为有了它，我们才能实现代码的重用，做点偷懒的事，这不正是我们从事计算机程序设计的初衷吗？

JavaScript 是一种动态的程序设计语言，因而它对于同一个任务往往会同时存在几种不同的解决方案。在继承问题上也不例外。在本章中，我们将会从 ECMAScript 标准指定的模式开始，为您介绍一系列常见的继承模式。只有很好地理解这些模式，我们才能在具体的工程中选择正确的模式或模式组合。

另外在本章某些地方，我们还会提到 Douglas Crockford 的名字。事实上，在谈论有关 JavaScript 中的继承问题时，我们已经很难不提这位人物所做的工作。除了在第 1 章中所推荐的那些视频（<http://developer.yahoo.com/yui/theater/>）以外，我们也建议读者去阅读一下他个人网站中的文章（<http://crockford.com/javascript>）。

6.1 原型链

让我们先从默认的继承模式开始，即通过原型来实现继承关系链。

正如我们之前所了解的，JavaScript 中的每个函数中都有一个名为 `prototype` 的对象属

性。该函数被 `new` 操作符调用时会创建出一个对象，并且该对象中会有一个指向其原型对象的秘密链接。通过该秘密链接（在某些环境中，该链接名为 `__proto__`），我们就可以在新建的对象中调用相关原型对象的方法和属性。

而原型对象自身也具有对象固有的普遍特征，因此本身也包含了指向其原型的链接。由此就形成了一条链，我们称之为原型链。

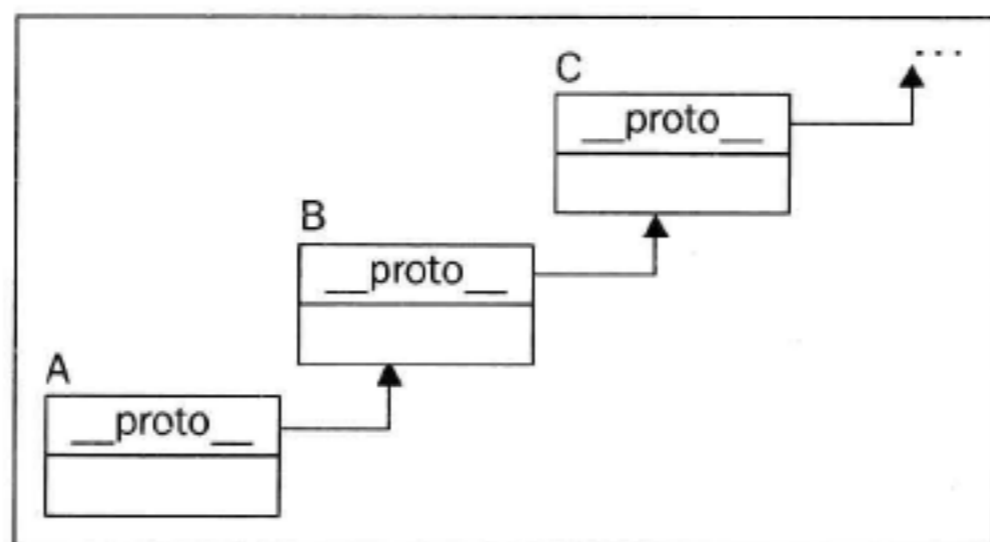


图 6-1

如图 6-1 所示，在对象 **A** 的一系列属性中，有一个叫做 `__proto__` 的隐藏属性，它指向了另一个对象 **B**。而 **B** 的 `__proto__` 属性又指向了对象 **C**，以此类推，直至链条末端的 `Object` 对象，该对象是 JavaScript 中的最高级父对象，语言中所有对象都必须继承自它。

这些都很好理解，但这有什么实际意义吗？显然有，正因为有了这些技术，我们才可以在某个属性不在对象 **A** 中而在对象 **B** 中时，依然将它当做 **A** 的属性来访问。同样的，如果对象 **B** 中也没有该属性，还可以继续到对象 **C** 中去寻找。这就是继承的作用，它能使每个对象都能访问其继承链上的任何一个属性。

在后面内容中，我们将会演示一系列不同的继承应用，这些示例将由一组层次分明的结构组成。具体地说，就是一组以通用性对象 `Shape` 为父对象二维图形对象序列（包括 `Triangle`、`Rectangle` 等）。

6.1.1 原型链示例

原型链是 ECMAScript 标准指定的默认继承方式。下面，我们就用这种方式来实现之前所描述的层次结构吧，首先我们来定义三个构造器函数：

```

function Shape() {
    this.name = 'shape';
    this.toString = function() {return this.name;};
}
function TwoDShape() {

```

```
    this.name = '2D shape';
  }
  function Triangle(side, height) {
    this.name = 'Triangle';
    this.side = side;
    this.height = height;
    this.getArea = function(){return this.side * this.height / 2;};
  }
```

接下来，就是我们施展继承魔法的代码了：

```
TwoDShape.prototype = new Shape();
Triangle.prototype = new TwoDShape();
```

明白上面发生了什么吗？在这里，我们将对象直接创建在 `TwoDShape` 对象的 `prototype` 属性中，并没有去扩展这些对象的原有原型。也就是说，我们用构造器 `Shape()`（通过 `new` 操作符）另建了一个新的实体，然后用它去覆盖该对象的原型。`Triangle` 对象也一样，它的 `prototype` 属性是由构造器 `TwoDShape()` 负责重建的（通过 `new` 操作符）。这样做的重点在于：由于 JavaScript 与我们所熟悉的其他程序设计语言（例如 C++、java 或 php）都不一样，它是一种完全依靠对象的语言，其中没有类（class）的概念。因此我们需要直接用 `new Shape()` 构造一个实体，然后才能通过该实体的属性完成相关的继承工作，而不是直接继承自 `Shape()` 构造器。另外这也确保了在继承实现之后，我们对 `Shape()` 所进行的任何修改、重写甚至删除，都不会对 `TwoDShape()` 产生影响，因为我们所继承的只是由该构造器所建的一个实体。

正如在第 5 章中所提到的，当我们对对象的 `prototype` 属性进行完全重写时（这不同于简单的功能扩展），有可能会对对象 `constructor` 属性产生一定的负面影响。所以，在我们完成相关的继承关系设定后，对这些对象的 `constructor` 属性进行相应的重置是一个非常好的习惯。

```
TwoDShape.prototype.constructor = TwoDShape;
Triangle.prototype.constructor = Triangle;
```

下面，我们来测试一下目前为止所实现的内容，先创建一个 `Triangle` 对象，然后调用它的 `getArea()` 方法：

```
>>> var my = new Triangle(5, 10);
>>> my.getArea();
```

25

尽管该对象中并没有属于自己的 `toString()` 方法，但我们依然可以调用它所继承的

方法。并且该方法 `toString()` 显然是与 `my` 对象紧密绑定在一起的。

```
>>> my.toString()
"Triangle"
```

接下来，我们关注一下 JavaScript 引擎在 `my.toString()` 被调用时究竟做了哪些事：

- ◆ 首先，它会遍历 `my` 对象中的所有属性，但没有找到一个叫做 `toString()` 的方法。
- ◆ 接着再去查看 `my.__proto__` 所指向的对象，该对象应该是在继承关系构建过程中由 `new TwoDShape()` 所创建的实体。
- ◆ 显然，JavaScript 引擎在遍历 `TwoDShape` 实体的过程中依然不会找到 `toString()` 方法，然后，它又会继续检查该实体的 `__proto__` 属性。这时候，该 `__proto__` 属性所指向的实体是由 `new Shape()` 所创建的。
- ◆ 终于，在 `new Shape()` 所创建的实体中找到了 `toString()` 方法。
- ◆ 最后，该方法就会在 `my` 对象中被调用，并且其 `this` 也指向了 `my`。

如果我们向 `my` 对象询问：“您的构造器函数是哪一个？”，它应该是能够给出正确答案的。因为我们在构建继承关系时已经对相关的 `constructor` 属性进行了重置。

```
>>> my.constructor
Triangle(side, height)
```

通过 `instanceof` 操作符，我们可以验证 `my` 对象同时是上述三个构造器的实例：

```
>>> my instanceof Shape
true
>>> my instanceof TwoDShape
true
>>> my instanceof Triangle
true
>>> my instanceof Array
false
```

同样的，当我们以 `my` 参数调用这些构造器原型的 `isPrototypeOf()` 方法时，结果也是如此：

```
>>> Shape.prototype.isPrototypeOf(my)
true
>>> TwoDShape.prototype.isPrototypeOf(my)
```

```
true
>>> Triangle.prototype.isPrototypeOf(my)
true
>>> String.prototype.isPrototypeOf(my)
false
```

我们也可以使用其他两个构造器来创建对象，用 `new TwoDShape()` 所创建的对象也可以获得继承自 `Shape()` 的 `toString()` 方法。

```
>>> var td = new TwoDShape();
>>> td.constructor
TwoDShape ()
>>> td.toString()
"2D shape"
>>> var s = new Shape();
>>> s.constructor
Shape ()
```

6.1.2 将共享属性迁移到原型中去

当我们用某一个构造器创建对象时，其属性就被添加到 `this` 中去。这会使某些不能通过实体改变的属性出现一些效率低下的情况。例如，在上面的示例中，`Shape()` 构造器是这样定义的：

```
function Shape() {
  this.name = 'shape';
}
```

这种实现意味着每当我们用 `new Shape()` 新建对象时，每个实体都会有一个全新的 `name` 属性，并在内存中拥有自己独立的存储空间。而事实上，我们也可以选择将 `name` 属性添加到所有实体所共享的原型对象中去：

```
function Shape() {}
Shape.prototype.name = 'shape';
```

这样一来，每当我们再用 `new Shape()` 新建对象时，新对象中就不再含有属于自己的 `name` 属性了，而是被添加进了该对象的原型中。虽然这样做通常会更有效率，但这也只是针对对象实体中的不可变属性而言的，另外，这种方式也同样适用于对象中的共享性方法。

现在，让我们来改善一下之前的示例，将其所有的方法和那些符合条件的属性添加到

原型对象中去，就 `Shape()` 和 `TwoDShape()` 而言，几乎所有东西都是可以共享的：

```
function Shape() {}
// augment prototype
Shape.prototype.name = 'shape';
Shape.prototype.toString = function() {return this.name;};
function TwoDShape() {}
// take care of inheritance
TwoDShape.prototype = new Shape();
TwoDShape.prototype.constructor = TwoDShape;
// augment prototype
TwoDShape.prototype.name = '2D shape';
```

如您所见，我们通常会在对原型对象进行扩展之前，先完成相关的继承关系构建，否则 `TwoDShape.prototype` 中的后续新内容有可能会抹掉我们所继承来的东西。

而 `Triangle` 构造器的情况稍许有些不同，因为由 `new Triangle()` 所创建的各个对象在尺寸上是各不相同的。因此，该对象的 `side` 和 `height` 这两个属性必须保持自身所有，而剩余的属性和 `getArea()` 方法，都可以设置共享。另外，需要再次强调一次，我们必须在扩展原型对象之前完成继承关系的构建。

```
function Triangle(side, height) {
  this.side = side;
  this.height = height;
}
// take care of inheritance
Triangle.prototype = new TwoDShape();
Triangle.prototype.constructor = Triangle;
// augment prototype
Triangle.prototype.name = 'Triangle';
Triangle.prototype.getArea = function(){return this.side * this.height / 2;};
```

修改完成之后，之前所有的测试代码都可以同样的方式应用于当前版本，例如：

```
>>> var my = new Triangle(5, 10);
>>> my.getArea()
25
>>> my.toString()
"Triangle"
```

如您所见，实际上调用 `my.toString()` 的区别仅仅存在于幕后的某些少量操作。主要区别也就是方法的查找操作将更多地发生在 `Shape.prototype` 中，而不再需要像前面

示例中那样，到由 `new Shape()` 所创建的实体对象中查找了。

另外，我们也可以通过 `hasOwnProperty()` 方法来明确对象自身属性与其原型链属性的区别。

```
>>> my.hasOwnProperty('side')
true
>>> my.hasOwnProperty('name')
false
```

而调用 `isPrototypeOf()` 方法和 `instanceof` 操作符的工作方式与之前并无区别，例如：

```
>>> TwoDShape.prototype.isPrototypeOf(my)
true
>>> my instanceof Shape
true
```

6.2 只继承于原型

正如上面所说，出于效率考虑，我们应该尽可能地将一些可重用的属性和方法添加到原型中去。如果形成了这样一个好习惯，我们仅仅依靠原型就能完成继承关系的构建了。由于原型中的所有代码都是可重用的，这意味着继承自 `Shape.prototype` 比继承自 `new Shape()` 所创建的实体要好得多。毕竟，`new Shape()` 方式会将 `Shape` 的属性设定为对象自身属性，这样的代码是不可重用的（因而要将其设置在原型中），但我们也采取以下方式对效率做一些改善：

- ◆ 不要单独为继承关系创建新对象。
- ◆ 尽量减少运行时方法搜索，例如 `toString()`。

下面就是更改后的代码，我们用高亮显示被修改的部分：

```
function Shape() {}
// augment prototype
Shape.prototype.name = 'shape';
Shape.prototype.toString = function() {return this.name;};

function TwoDShape() {}
// take care of inheritance
TwoDShape.prototype = Shape.prototype;
```

```
TwoDShape.prototype.constructor = TwoDShape;
// augment prototype
TwoDShape.prototype.name = '2D shape';

function Triangle(side, height) {
  this.side = side;
  this.height = height;
}
// take care of inheritance
Triangle.prototype = TwoDShape.prototype;
Triangle.prototype.constructor = Triangle;
// augment prototype
Triangle.prototype.name = 'Triangle';
Triangle.prototype.getArea = function(){return this.side * this.height
/ 2;}
```

测试结果依然相同：

```
>>> var my = new Triangle(5, 10);
>>> my.getArea()
25
>>> my.toString()
"Triangle"
```

但是，这样做会令 `my.toString()` 方法的查找有什么不同吗？首先，JavaScript 引擎同样会先查看 `my` 对象中有没有 `toString()` 方法。自然，它不会找到，于是就会转而去搜索该对象的原型属性。此时该原型已经指向了 `TwoDShape` 的原型，而后者指向的又是 `Shape.prototype`。更重要的是，由于这里所采用的都是引用传递而不是值传递，所以这里的方法查询步骤由（之前示例中的）四步或（本章首例中的）三步直接被精简成两步。

这样简单地拷贝原型从效率上来说固然会更好一些，但也有它的副作用。由于子对象与父对象指向的是同一个对象，所以一旦子对象对其原型进行了修改，父对象也会随即被改变，甚至所有的继承关系也都是如此。

例如下面这行代码：

```
Triangle.prototype.name = 'Triangle';
```

它对该对象的 `name` 属性进行了修改，于是 `Shape.prototype.name` 也随之被改变了。也就是说，当我们再用 `new Shape()` 新建对象时，新对象的 `name` 属性也会是 **“Triangle”**：

```
>>> var s = new Shape()
>>> s.name
"Triangle"
```

临时构造器——new F()

正如上面所说，如果所有属性都指向了一个相同的对象，父对象就会受到子对象属性的影响。要解决这个问题，就必须利用某种中介来打破这种连锁关系。我们可以用一个临时构造器函数来充当中介。即我们创建一个空函数 `f()`，并将其原型设置为父级构造器。然后，我们既可以用 `new F()` 来创建一些不包含父对象属性的对象，同时又可以从父对象 `prototype` 属性中继承一切了。

下面是修改之后的代码：

```
function Shape() {}
// augment prototype
Shape.prototype.name = 'shape';
Shape.prototype.toString = function() {return this.name;};

function TwoDShape() {}
// take care of inheritance
var F = function() {};
F.prototype = Shape.prototype;
TwoDShape.prototype = new F();
TwoDShape.prototype.constructor = TwoDShape;
// augment prototype
TwoDShape.prototype.name = '2D shape';

function Triangle(side, height) {
  this.side = side;
  this.height = height;
}
// take care of inheritance
var F = function() {};
F.prototype = TwoDShape.prototype;
Triangle.prototype = new F();
Triangle.prototype.constructor = Triangle;
// augment prototype
Triangle.prototype.name = 'Triangle';
Triangle.prototype.getArea = function() {return this.side * this.height / 2;};
```

下面，我们来创建一个 `triangle` 对象，并测试其方法：

```

>>> var my = new Triangle(5, 10);
>>> my.getArea()
25
>>> my.toString()
"Triangle"

```

通过这种方法，我们就可以在保持原型链的基础上使父对象的属性摆脱子对象的影响了：

```

>> my.__proto__.__proto__.__proto__.constructor
Shape ()
>>> var s = new Shape();
>>> s.name
"shape"

```

与此同时，该方法也对一种意见提供了支持，即尽量将要共享的属性与方法添加到原型中，然后只围绕原型构建继承关系。也就是说，这种主张不鼓励将对象的自身属性纳入继承关系，其背后的根源在于如果对象自身属性被设定得太过具体，会令其丧失可重用性。

6.3 uber——子对象访问父对象的方式

在传统的面向对象语言中，通常都会提供一种用于子类访问父类（有时也叫超类）的特殊语法，因为我们在实现子类方法往往需要其父类方法的额外辅助。在这种情况下，子类通常就要去调用父类中的同名方法，以便最终完成工作。

JavaScript 中虽然没有这种特殊语法，但是要实现类似的功能还是很容易的。接下来，让我们再对之前的示例做一些修改，在构建继承关系的过程中引入一个 `uber` 属性，并令其指向其父级原型对象：

```

function Shape() {}
// augment prototype
Shape.prototype.name = 'shape';
Shape.prototype.toString = function() {
  var result = [];
  if (this.constructor.uber) {
    result[result.length] = this.constructor.uber.toString();
  }
  result[result.length] = this.name;
  return result.join(', ');
};

```

```
function TwoDShape() {}
// take care of inheritance
var F = function() {};
F.prototype = Shape.prototype;
TwoDShape.prototype = new F();
TwoDShape.prototype.constructor = TwoDShape;
TwoDShape.uber = Shape.prototype;
// augment prototype
TwoDShape.prototype.name = '2D shape';

function Triangle(side, height) {
  this.side = side;
  this.height = height;
}

// take care of inheritance
var F = function() {};
F.prototype = TwoDShape.prototype;
Triangle.prototype = new F();
Triangle.prototype.constructor = Triangle;
Triangle.uber = TwoDShape.prototype;
// augment prototype
Triangle.prototype.name = 'Triangle';
Triangle.prototype.getArea = function(){return this.side * this.height
/ 2;}
```

在这里，我们主要新增了以下内容：

- ◆ 将 `uber` 属性设置成了指向其父级原型的引用。
- ◆ 对 `toString()` 方法进行了更新。

在此之前，`toString()` 所做的仅仅是返回 `this.name` 的内容而已。现在我们为它新增了一项额外任务，即检查对象中是否存在 `this.constructor.uber` 属性，如果存在，就先调用该属性的 `toString` 方法。在这里，`this.constructor` 本身是一个函数，而 `this.constructor.uber` 则是指向当前对象父级原型的引用。因而，当我们调用 `Triangle` 实体的 `toString()` 方法时，其原型链上所有的 `toString()` 都会被调用：

```
>>> var my = new Triangle(5, 10);
>>> my.toString()
```

```
"shape, 2D shape, Triangle"
```

另外，`uber` 属性的名字原本应该是“`superclass`”，但这样一来好像显得 JavaScript 中有

了类的概念，或许应该叫做“super”（就像 Java 那样），但 super 一词在 JavaScript 中属于保留字。因而，Douglass Crockford 建议采用德语中与“super”同义词“uber”，这个主意看起来不错，挺酷的。

6.4 将继承部分封装成函数

下面，我们要将这些实现继承关系的代码提炼出来，并迁入一个叫做 extend() 的可重用函数中：

```
function extend(Child, Parent) {  
    var F = function(){};  
    F.prototype = Parent.prototype;  
    Child.prototype = new F();  
    Child.prototype.constructor = Child;  
    Child.uber = Parent.prototype;  
}
```

通过应用上面的函数（读者也可以自行再定义一个），我们既可以使代码保持简洁。又能将其重用在构建继承关系的任务中。这种方式能让我们通过以下简单的调用来实现基础：

```
extend(TwoDShape, Shape);
```

以及：

```
extend(Triangle, TwoDShape);
```

而这也是 YUI (Yahoo! User Interface) 库在实现继承关系时所采用的方法，即通过它的 extend() 方法。例如，如果我们想在 YUI 中设定 Triangle 继承于 Shape 的关系，就可以这样：

```
YAHOO.lang.extend(Triangle, Shape)
```

6.5 属性拷贝

接下来，让我们尝试一个与之前略有不同的方法。在构建可重用的继承代码时，我们也可以简单地将父对象的属性拷贝给子对象，参照之前的 extend() 接口，我们可以创建

一个 `extend2()` 函数，该函数也接受两个构造器函数为参数，并将 `parent` 的原型属性全部拷贝给 `child` 原型，其中包括方法，因为方法本身也是一种函数类型的属性。

```
function extend2(Child, Parent) {
  var p = Parent.prototype;
  var c = Child.prototype;
  for (var i in p) {
    c[i] = p[i];
  }
  c.uber = p;
}
```

如您所见，我们通过一个简单的循环遍历了函数所接受的所有属性。在之前的示例中，如果子对象需要访问父对象的方法，我们可以通过设置 `uber` 属性来实现。而这里的情况与之前有所不同，由于我们已经完成对 `child` 的原型进行扩展，不需要再重置 `Child.prototype.constructor` 属性了，因为它不会再被完全覆盖了，因此在这里，`constructor` 属性所指向的值是正确的。

与之前的方法相比，这个方法在效率上显然略逊一筹。因为这里执行的是子对象原型的逐一拷贝，而非简单的原型链查询。所以我们必须要记住，这种方式仅适用于只包含基本数据类型的对象，所有的对象类型（包括函数与数组）都是不可复制的，因为它们只支持引用传递。

下面我们来看看具体的应用示例，以下有两个构造器函数 `Shape()` 和 `TwoDShape()`。其中，`Shape()` 的原型中包含了一个基本类型属性 `name`，和一个非基本类型属性——`toString()` 方法：

```
var Shape = function(){};
var TwoDShape = function(){};
Shape.prototype.name = 'shape';
Shape.prototype.toString = function(){return this.name;};
```

如果我们通过 `extend()` 方法来实现继承，那么 `name` 属性既不会是 `TwoDShape()` 实例的属性，也不会成为其原型对象的属性，但是子对象依然可以通过继承方式来访问该属性。

```
>>> extend(TwoDShape, Shape);
>>> var td = new TwoDShape();
>>> td.name
"shape"
>>> TwoDShape.prototype.name
"shape"
```

```

>>> td.__proto__.name
"shape"
>>> td.hasOwnProperty('name')
false
>>> td.__proto__.hasOwnProperty('name')
false

```

而如果继承是通过 `extend2()` 方法来实现的，`TwoDShape()` 的原型中就会拷贝有属于自己的 `name` 属性。同样的，其中也会拷贝有属于自己的 `toString` 方法，但这只是一个函数引用，函数本身并没有被再次创建。

```

>>> extend2(TwoDShape, Shape);
>>> var td = new TwoDShape();
>>> td.__proto__.hasOwnProperty('name')
true
>>> td.__proto__.hasOwnProperty('toString')
true
>>> td.__proto__.toString === Shape.prototype.toString
true

```

如您所见，上面两个 `toString()` 方法实际是同一个函数对象。之所以这样做，也是因为这样的方法重建其实是完全没有必要的。

显然，之所以说 `extend2()` 方法的效率要低于 `extend()` 方法，主要是前者对部分原型属性进行了重建。当然了，这对于只包含基本数据类型的对象来说，未必真的就如此糟糕。而且，这样做还能使属性查找操作更多地停留在对象本身，从而减少了原型链上的查找。

6.6 小心处理引用拷贝

事实上，对象类型（包括函数与数组）通常都是以引用形式来进行拷贝的，这有时会导致一些不可预测的结果。

下面，我们来创建两个构造器函数，并在第一个构造器的原型中添加一些属性：

```

>>> var A = function() {}, B = function() {};
>>> A.prototype.stuff = [1,2,3];
[1, 2, 3]
>>> A.prototype.name = 'a';
"a"

```

现在，我们让 `B` 继承 `A`（通过 `extend()` 或 `extend2()` 来实现）：


```
>>> extend2(B, A);
```

这里使用的是 `extend2()`，即 B 的原型继承了 A 的原型属性，并将其变成了自身属性。

```
>>> B.prototype.hasOwnProperty('name')
true
>>> B.prototype.hasOwnProperty('stuff')
true
```

其中，`name` 属于基本类型属性，创建的是一份全新的拷贝。而 `stuff` 属性是一个数组对象，它所执行的是引用拷贝：

```
>>> B.prototype.stuff
[1, 2, 3]
>>> B.prototype.stuff === A.prototype.stuff
true
```

因而，如果改变 B 中的 `name` 拷贝，不会对 A 产生影响：

```
>>> B.prototype.name += 'b'
"ab"
>>> A.prototype.name
"a"
```

但如果改变的是 B 的 `stuff` 属性，A 就会受到影响了，因为这两个属性引用的是同一个数组：

```
>>> B.prototype.stuff.push(4, 5, 6);
6
>>> A.prototype.stuff
[1, 2, 3, 4, 5, 6]
```

当然，如果我们用另一个对象对 B 的 `stuff` 属性进行完全重写（而不是修改现有属性），事情就完全不一样了。在这种情况下，A 的 `stuff` 属性会继续引用原有对象，而 B 的 `stuff` 属性则指向了新的对象。

```
>>> B.prototype.stuff = ['a', 'b', 'c'];
["a", "b", "c"]
>>> A.prototype.stuff
```

[1, 2, 3, 4, 5, 6]

这里的主要思想是，当某些东西被创建为一个对象时，它们就被存储在内存中的某个物理位置，相关的变量和属性就会指向这些位置（见图 6-2）。而当我们把一个新对象赋值给 `B.prototype.stuff` 时，就相当于告诉它：“喂，忘了那个旧对象吧，将指针转移到现在这个新对象上来”。

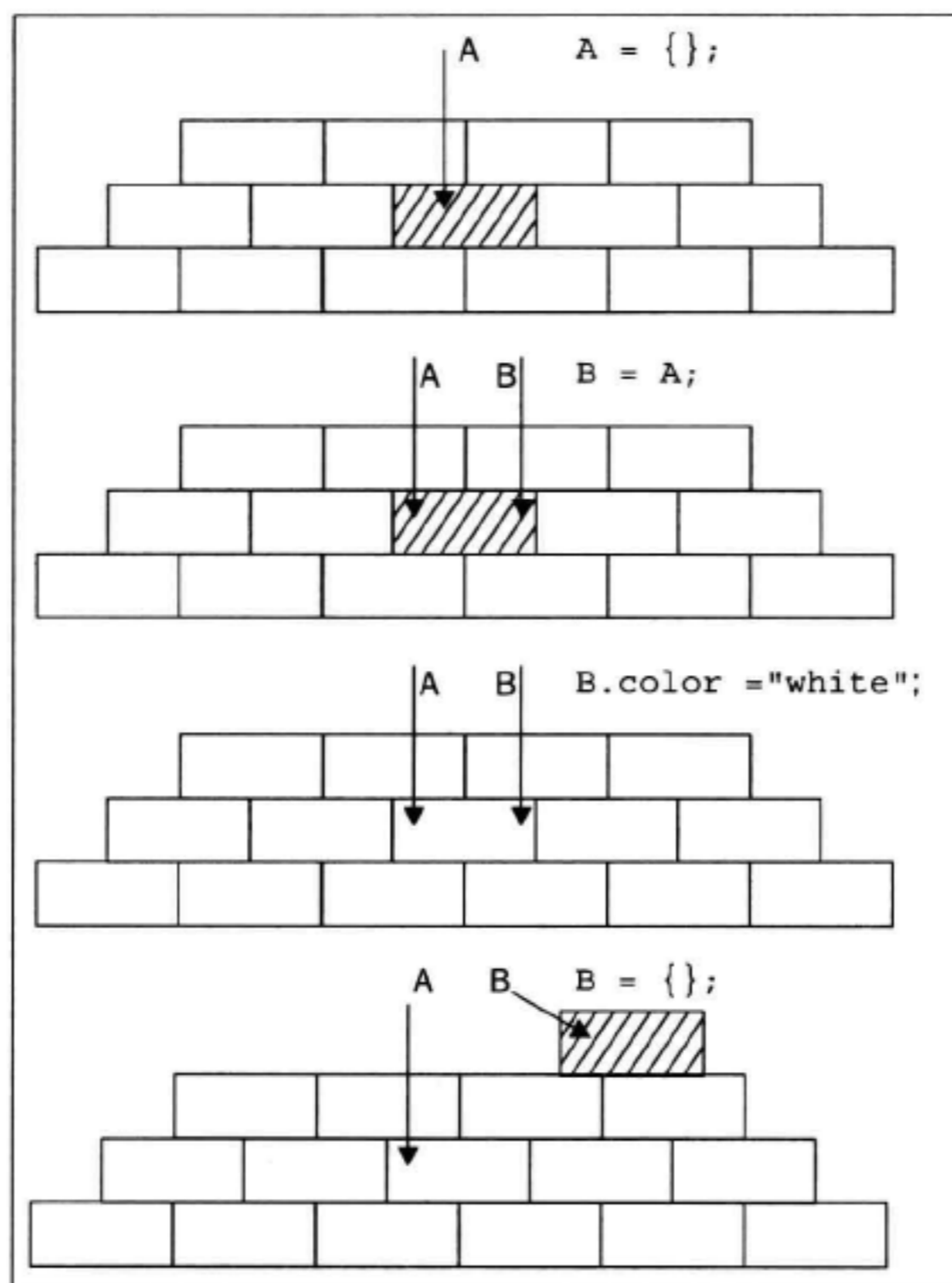


图 6-2

6.7 对象之间的继承

到目前为止，本章所有的示例都是以构造器创建对象为前提的，并且，我们在这些用于创建对象的构造器中引入了从其他构造器中继承而来的属性。但实际上，我们也可以丢开构造器，直接通过对象标识法来创建对象，并且这样做还能减少我们的实际输入。但是，它们是如何实现继承的呢？

在 Java 或 PHP 中，我们是通过类定义来构建不同类之间的继承关系的。由此可见，传统意义上的面向对象是依靠类来完成的。但 JavaScript 中没有类的概念，因此，那些具有传统编程背景的程序员自然而然地会将构造器函数当做类，因为两者在使用方式上是最为接近的。此外，JavaScript 中也提供了 `new` 操作符，这使得 JavaScript 与 Java 的相似程度更为接近。无论如何，所有的一切最终都要回到对象层面上来。例如在本章的第一个示例中，我们使用的语法是这样的：

```
Child.prototype = new Parent();
```

尽管这里的 `Child` 构造器（您也可以将其视为类）是从 `Parent` 继承而来的，但对象本身则是通过 `new Parent()` 调用来创建的。这就是为什么我们说这是一种仿传统的继承模式，它尽管很像传统继承，但终究不是（因为这里不存在任何类的调用）。

那么，我们为什么不能拿掉这个中间人（即构造器/类），直接在对象之间构建继承关系呢？在 `extend2()` 方法中，父原型对象的属性被逐一拷贝给了子原型对象，而这两个原型本质上也都是对象。接下来，让我们将原型和构造器忘了，尝试在对象之间进行直接属性拷贝吧。

首先，我们用 `var o = {}` 语句创建一个对象作为画板，然后再逐步为其添加属性。但这次我们不通过 `this` 来实现，而是直接将现有对象的属性全部拷贝过来。例如在下面的实现中，函数将接受一个对象并返回它的副本。

```
function extendCopy(p) {
    var c = {};
    for (var i in p) {
        c[i] = p[i];
    }
    c.uber = p;
    return c;
}
```

单纯的属性全拷贝是一种非常简单的模式，但适用范围很广。例如，Firebug 后台有一个 `extend()` 函数，其代码所用的就是这种方式。另外，在某些流行的 JavaScript 程序库，例如 jQuery 的早期版本中，原型采取的就是这种基本模式。

下面来看看这个函数的实际应用。首先，我们需要一个基本对象：

```
var shape = {
    name: 'shape',
```

```
    toString: function() {return this.name;}  
}
```

接着我们就可以根据这个旧对象来创建一个新的对象了，只需调用上面的 `extendCopy()` 函数，该函数会返回一个新对象。然后，我们可以继续对这个新对象进行扩展，添加额外的功能。

```
var twoDee = extendCopy(shape);  
twoDee.name = '2D shape';  
twoDee.toString = function(){return this.uber.toString() + ', ' +  
this.name;};
```

下面，我们让 `triangle` 对象继承一个 2D 图形对象：

```
var triangle = extendCopy(twoDee);  
triangle.name = 'Triangle';  
triangle.getArea = function(){return this.side * this.height / 2;}
```

使用该 `triangle`：

```
>>> triangle.side = 5; triangle.height = 10; triangle.getArea();  
25  
>>> triangle.toString();  
"shape, 2D shape, Triangle"
```

对于这种方法而言，可能的问题就在于初始化一个新 `triangle` 对象的过程过于繁琐。因为我们必须要对该对象的 `side` 和 `height` 值进行手动设置，这与之前直接将相关的值传递给构造器函数是不一样的。但这方面的问题只需要调用一个函数就能轻易解决，例如与构造器函数类似的 `init()` 方法（如果您使用 PHP5，也可以调用 `__construct()`），我们只需要在调用时将这两个值以参数形式传递给它即可。

6.8 深拷贝

在之前的讨论中，`extendCopy()` 函数所用的创建方式叫做浅拷贝（`shallow copy`）。与之相对的，当然就是所谓的深拷贝（`deep copy`）了。经过之前章节（即 6.6）的讨论，我们已经知道当对象类型的属性被拷贝时，实际上拷贝的只是该对象在内存中的位置指针。这一过程就是所谓的浅拷贝，在这种情况下，如果我们修改了拷贝对象，就等同于修改了原对象。而深拷贝则可以帮助我们避免这方面的问题。

深拷贝的实现方式与浅拷贝基本相同，也需要通过遍历对象的属性来进行拷贝操作。

只是在遇到一个对象引用性的属性时，我们需要再次对其调用深拷贝函数：

```
function deepCopy(p, c) {
  var c = c || {};
  for (var i in p) {
    if (typeof p[i] === 'object') {
      c[i] = (p[i].constructor === Array) ? [] : {};
      deepCopy(p[i], c[i]);
    } else {
      c[i] = p[i];
    }
  }
  return c;
}
```

现在我们来创建一个包含数组和子对象属性的对象：

```
var parent = {
  numbers: [1, 2, 3],
  letters: ['a', 'b', 'c'],
  obj: {
    prop: 1
  },
  bool: true
};
```

下面，我们分别用深拷贝和浅拷贝测试一下，就会发现深拷贝与浅拷贝不同。对它的 `numbers` 属性进行更新不会对原对象产生影响。

```
>>> var mydeep = deepCopy(parent);
>>> var myshallow = extendCopy(parent);
>>> mydeep.numbers.push(4, 5, 6);
6
>>> mydeep.numbers
[1, 2, 3, 4, 5, 6]
>>> parent.numbers
[1, 2, 3]
>>> myshallow.numbers.push(10)
4
>>> myshallow.numbers
[1, 2, 3, 10]
>>> parent.numbers[1, 2, 3, 10]
>>> mydeep.numbers
```

```
[1, 2, 3, 4, 5, 6]
```

在 jQuery 的较新版本中，继承关系的实现通常都会采用深拷贝的形式。

6.9 object()

基于这种在对象之间直接构建继承关系的理念，Douglas Crockford 为我们提出了一个建议，即可以用 `object()` 函数来接收父对象，并返回一个以该对象为原型的新对象。

```
function object(o) {  
  function F() {}  
  F.prototype = o;  
  return new F();  
}
```

如果我们需要访问 `uber` 属性，可以继续 `object()` 函数，具体如下：

```
function object(o) {  
  var n;  
  function F() {}  
  F.prototype = o;  
  n = new F();  
  n.uber = o;  
  return n;  
}
```

这个函数的使用与 `extendCopy()` 基本相同：我们只需要将某个对象（例如 `twoDee`）传递给它，并由此创建一个新对象。然后再对新对象进行后续的扩展处理。

```
var triangle = object(twoDee);  
triangle.name = 'Triangle';  
triangle.getArea = function(){return this.side * this.height / 2;};
```

新 `triangle` 对象的行为依然不变：

```
>>> triangle.toString()  
"shape, 2D shape, Triangle"
```

这种模式也被称为原型继承，因为在这里，我们将父对象设置成了子对象的原型。

6.10 原型继承与属性拷贝的混合应用

对于继承应用来说，主要目标就是将一些现有的功能归为己有。也就是说，我们在新建一个对象时，通常首先应该继承于现有对象，然后再为其添加额外的方法与属性。对此，我们可以通过一个函数调用来完成，并且在其中混合使用我们刚才所讨论的两种方式。

具体而言就是：

- ◆ 使用原型继承的方式克隆（clone）现存对象。
- ◆ 而对其他对象使用属性拷贝（copy）的方式。

```
function objectPlus(o, stuff) {
  var n;
  function F() {}
  F.prototype = o;
  n = new F();
  n.uber = o;

  for (var i in stuff) {
    n[i] = stuff[i];
  }
  return n;
}
```

这个函数接受两个参数，其中对象 `o` 用于继承，而另一个对象 `stuff` 则用于拷贝方法与属性。下面我们来看看实际应用。

首先，需要一个基本对象 `shape`：

```
var shape = {
  name: 'shape',
  toString: function() {return this.name;}
};
```

接着再创建一个继承于 `shape` 的 2D 对象，并为其添加更多的属性。这些额外的属性由一个用文本标识法所创建的匿名对象提供。

```
var twoDee = objectPlus(shape, {
  name: '2D shape',
  toString: function(){return this.uber.toString() + ', ' + this.name}
});
```

现在，我们来创建一个继承于 2D 对象的 `triangle` 对象，并为其添加一些额外的属性。

```
var triangle = objectPlus(twoDee, {
  name: 'Triangle',
  getArea: function(){return this.side * this.height / 2;},
  side: 0,
  height: 0
});
```

下面我们来测试一下：创建一个具体的 `triangle` 对象 `my`，并自定义其 `side` 和 `height` 属性。

```
>>> var my = objectPlus(triangle, {side: 4, height: 4});
>>> my.getArea()
8
>>> my.toString()
"shape, 2D shape, Triangle, Triangle"
```

这里的不同之处在于，当 `toString()` 函数被执行时，**Triangle** 的 `name` 属性会被重复两次。这是因为我们在具现化实例时是继承于 `triangle` 对象的，所以这里多了一层继承关系。我们也可以给该实例一个新的 `name` 属性。例如：

```
>>> var my = objectPlus(triangle, {side: 4, height: 4,
                                name: 'My 4x4'});
>>> my.toString()
"shape, 2D shape, Triangle, My 4x4"
```

6.11 多重继承

所谓的多重继承，通常指的是一个子对象中有不止一个父对象的继承模式。对于这种模式，有些面向对象程序语言支持，有些则不支持。我们可以对它们进行一些甄别，自行判断在复杂的应用程序设计中多重继承是否能带来方便，或者是否有这种必要使用它，以及它是否会比原型链的方式更好。但无论如何，对于 JavaScript 这样的动态语言来说，实现多重继承是很简单的，尽管语言本身没有为此提供特殊的语法单元。现在，让我们暂且先离开一下这个讨论多重继承利弊的漫漫长夜，去实现中感受一下多重继承的用法吧。

多重继承实现是极其简单的，我们只需要延续属性拷贝法的继承思路依次扩展对象，

就不会对其所继承的对象数量参数输入性限制。

下面，我们来创建一个 `multi()` 函数，它可以接受任意数量的输入性对象。然后，我们在其中实现了一个双重循环，内层循环用于拷贝属性，而外层循环则用于遍历函数参数中所传递进来的所有对象。

```
function multi() {
  var n = {}, stuff, j = 0, len = arguments.length;
  for (j = 0; j < len; j++) {
    stuff = arguments[j];
    for (var i in stuff) {
      n[i] = stuff[i];
    }
  }
  return n;
}
```

现在来测试一下：首先，我们需要创建 `shape`、`twoDee` 以及一个匿名对象。然后调用 `multi()` 函数时将这三个对象传递给它，该函数会返回新建的 `triangle` 对象。

```
var shape = {
  name: 'shape',
  toString: function() {return this.name;}
};

var twoDee = {
  name: '2D shape',
  dimensions: 2
};

var triangle = multi(shape, twoDee, {
  name: 'Triangle',
  getArea: function(){return this.side * this.height / 2;},
  side: 5,
  height: 10
});
```

然后，让我们来看看它是否可以工作：

```
>>> triangle.getArea()
```

```
>>> triangle.dimensions
2
>>> triangle.toString()
"Triangle"
```

要注意的是，`multi()` 中的循环是按照对象的输入顺序来进行遍历的。如果其中两个对象拥有相同的实现，通常以后一个对象为准。

混合插入

在这里，我们或许需要了解一种叫做混合插入（mixins）的技术。这项技术在一些语言（例如 Ruby）中非常受欢迎，我们可以将其看做一种为对象提供某些实用功能的技术，只不过，它并不是通过子对象的继承与扩展来完成的。我们之前所讨论的多重继承实际上正是基于这种技术理念来实现的。也就是说，每当我们新建一个对象时，可以选择将其他对象的内容混合到我们的新对象中去，只要将它们全部传递给 `multi()` 函数，我们就可以在不建立相关继承关系树的情况下获得这些对象的功能。

6.12 寄生式继承

如果我们想要在 JavaScript 中实现多种不同的继承模式，并且渴望多了解一些这方面的知识。这里可以再为您介绍一种叫做寄生式继承的模式。这是由 Douglas Crockford 所提出的技术，基本内容是，我们可以在创建对象的函数中直接吸收其他对象的功能，然后对其进行扩展并返回。“就好像所有的工作都是自己做的一样”。

下面，我们用对象标识法定义了一个普通对象，这时它还看不出有任何被寄生的可能性：

```
var twoD = {
  name: '2D shape',
  dimensions: 2
};
```

然后我们来编写用于创建 `triangle` 对象的函数：

- ◆ 将 `twoD` 对象克隆进一个叫做 `that` 的对象，这一步可以使用我们之前所讨论过的任何方法，例如使用 `object()` 函数或者执行全属性拷贝。
- ◆ 扩展 `that` 对象，添加更多的属性。

◆ 返回 that 对象。

```
function triangle(s, h) {
  var that = object(twoD);
  that.name = 'Triangle';
  that.getArea = function(){return this.side * this.height / 2;};
  that.side = s;
  that.height = h;
  return that;
}
```

由于 `triangle()` 只是个一般函数，不属于构造器，所以调用它通常是不需要 `new` 操作符的。但由于该函数返回的是一个对象，所以即便我们在函数调用时错误地使用了 `new` 操作符，它也会按照预定的方式工作。

```
>>> var t = triangle(5, 10);
>>> t.dimensions
2
>>> var t2 = new triangle(5,5);
>>> t2.getArea();
12.5
```

注意，这里的 `that` 只是一个名字，并不存在与 `this` 用法类似的特殊含义。

6.13 构造器借用

我们再来看一种继承实现（这是本章最后一个了，我保证）。这需要再次从构造器函数入手，这回不直接使用对象了。由于在这种继承模式中，子对象构造器可以通过 `call()` 或 `apply()` 方法来调用父对象的构造器，因而，它通常被称为构造器盗用法（*stealing a constructor*），或者构造器借用法（*borrowing a constructor*，如果您想更含蓄一点的话）。

尽管 `call()` 和 `apply()` 这两个方法在第 4 章中均已经讨论过，但这里我们要更进一步。正如您所知，这两个方法都允许我们将某个指定对象的 `this` 值与一个函数的调用绑定起来。这对于继承而言，就意味着子对象的构造器在调用父对象构造器时，也可以将子对象中新建的 `this` 对象与父对象的 `this` 值绑定起来。

下面，我们来构建一个父类构造器 `Shape()`：

```
function Shape(id) {
  this.id = id;
}
Shape.prototype.name = 'shape';
Shape.prototype.toString = function(){return this.name;};
```

现在我们来定义 `Triangle()` 构造器，在其中通过 `apply()` 方法来调用 `Shape()` 构造器，并将相关的 `this` 值（即 `new Triangle()` 所创建的示例）和其他一些参数传递该方法。

```
function Triangle() {
  Shape.apply(this, arguments);
}
Triangle.prototype.name = 'Triangle';
```

注意，这里无论是 `Triangle()` 还是 `Shape()` 都在其各自的原型中添加些额外的属性。

下面，我们来测试一下，先新建一个 `triangle` 对象：

```
>>> var t = new Triangle(101);
>>> t.name
"Triangle"
```

在这里，新的 `triangle` 对象继承了其父对象的 `id` 属性，但它并没有继承父对象原型中的其他任何东西：

```
>>> t.id
101
>>> t.toString();
"[object Object]"
```

之所以 `triangle` 对象中不包含 `Shape` 的原型属性，是因为我们从来没有调用 `new Shape()` 创建任何一个实例，自然其原型也从来没有被用到。这很容易做到，例如在本章最初的那个示例中，，我们可以对 `Triangle()` 构造器进行如下重定义：

```
function Triangle() {
  Shape.apply(this, arguments);
}
Triangle.prototype = new Shape();
Triangle.prototype.name = 'Triangle';
```

在这种继承模式中，父对象的属性是以子对象自身属性的身份来重建的（这与原型链模式中的子对象属性正好相反）。这也体现了构造器借用法的一大优势：当我们创建一个继承于数组或者其他对象类型的子对象时，将获得一个完完全全的新值（不是一个引用），对它做任何修改都不会影响其父对象。

但这种模式也是有缺点的，因为这种情况下父对象的构造器往往会被调用两次：一次发生在通过 `apply()` 方法继承其自身属性时，而另一次则发生在通过 `new` 操作符继承其原型时。这样一来，父对象的自身属性事实上被继承了两次。下面让我们来做一个简单的演示：

```
function Shape(id) {
  this.id = id;
}
function Triangle() {
  Shape.apply(this, arguments);
}
Triangle.prototype = new Shape(101);
```

然后我们新建一个实例：

```
>>> var t = new Triangle(202);
>>> t.id
202
```

如您所见，对象中有一个自身属性 `id`，但它并非来自原型链中，我们可以执行如下验证：

```
>>> t.__proto__.id
101
>>> delete t.id
true
>>> t.id
101
```

借用构造器与原型复制

对于这种由于构造器的双重调用而带来的重复执行问题，实际上是很容易更正的。我们可以在父对象构造器上调用 `apply()` 方法，以获得其全部的自身属性，然后再用一个简单的迭代器对其原型属性执行逐项拷贝（这也可以使用之前讨论的 `extend2()` 方法来完成）。例如：

```
function Shape(id) {
```

```

    this.id = id;
  }
  Shape.prototype.name = 'shape';
  Shape.prototype.toString = function(){return this.name;};

  function Triangle() {
    Shape.apply(this, arguments);
  }
  extend2(Triangle, Shape);
  Triangle.prototype.name = 'Triangle';

```

下面测试一下：

```

>>> var t = new Triangle(101);
>>> t.toString();

"Triangle"
>>> t.id
101

```

这样一来，双重继承就不见了：

```

>>> typeof t.__proto__.id
"undefined"

```

如果必要的话，`extend2()`还可以访问对象的 `uber` 属性：

```

>>> t.uber.name
"shape"

```

6.14 本章小结

在本章，我们学习了一系列用于实现继承的方法（模式）。它们大致上可以分为两类：

- ◆ 基于构造器工作的模式。
- ◆ 基于对象工作的模式。

此外，我们也可以基于以下条件对这些模式进行分类（见表 6-1）：

- ◆ 是否使用原型。
- ◆ 是否执行属性拷贝。
- ◆ 两者都有（即执行原型属性拷贝）。

表 6-1

方法编号	方法名称	代码示例	所属模式	技术注解
1	原型链法 (仿传统)	<pre>Child.prototype = new Parent();</pre>	基于构造器工作模式 使用原型链模式	ECMA 标准中的默认继承机制。 提示：我们可以将方法与属性集中可重用的部分迁移到原型链中，而将不可重用的那部分设置为对象的自身属性
2	仅从原型继承法	<pre>Child.prototype = Parent.prototype;</pre>	基于构造器工作模式 原型拷贝模式 (不存在原型，所有的对象共享一个原型对象)	由于该模式在构建继承关系时不需要新建对象实例，效率上会有较好的表现。 原型链上的查询也会比较快，因为这里根本不存在链，而缺点在于，对于对象的修改会影响其父对象
3	临时构造器法	<pre>function extend(Child, Parent) { var F = function(){}; F.prototype = Parent.prototype; Child.prototype = new F(); Child.prototype.constructor = Child; Child.uber = Parent.prototype; }</pre>	基于构造器工作模式 使用原型链模式	此模式不同于 1 号方法，它只继承父对象的原型属性，而对于其自身属性（也就是被构造器添加到 this 值中的属性）则不予继承。 该模式在 YUI 和 Extjs 等程序库中均有应用。 另外，该模式还为我们访问父对象提供了便利的方式（即通过 uber 属性）
4	原型属性拷贝法	<pre>function extend2(Child, Parent) { var p = Parent.prototype; var c = Child.prototype; for (var i in p) { c[i] = p[i]; } c.uber = p; }</pre>	基于构造器工作模式 拷贝属性模式 使用原型模式	将父对象原型中的内容全部转换成子对象原型属性。 无需为继承单独创建对象实例。 原型链本身也更短

(续表)

方法编号	方法名称	代码示例	所属模式	技术注解
5	全属性拷贝法 (即浅拷贝法)	<pre>function extendCopy(p) { var c = {}; for (var i in p) { c[i] = p[i]; } c.uber = p; return c; }</pre>	基于对象工作模式 属性拷贝模式	非常简单化。 在 Firebug、jQuery 以及 Prototype.js 的早期版本中都有应用。 引入了浅拷贝的概念。 不能作用于原型属性
6	深拷贝法	同上, 只需在遇到对象类型时重复调用上述函数即可	基于对象工作模式 属性拷贝模式	与方法 5 基本相同, 但所有对象执行的都是值传递。 在 jQuery 近期版本中被广泛应用
7	原型继承法	<pre>function object(o) { function F() {} F.prototype = o; return new F(); }</pre>	基于对象工作模式 使用原型链模式	丢开仿类机制, 直接在对象之间构建继承关系。 发挥原型固有优势
8	扩展与增强模式	<pre>function objectPlus(o, stuff) { var n; function F() {} F.prototype = o; n = new F(); n.uber = o; for (var i in stuff) { n[i] = stuff[i]; } return n; }</pre>	基于对象工作模式 使用原型链模式 属性拷贝模式	该方法实际上是原型继承法(方法 7)和属性拷贝法(方法 5)的混合应用。 它通过一个函数一次性完成对象的继承与扩展

(续表)

方法编号	方法名称	代码示例	所属模式	技术注解
9	多重继承法	<pre>function multi() { var n = {}, stuff, j = 0, len = arguments.length; for (j = 0; j < len; j++) { stuff = arguments[j]; for (var i in stuff) { n[i] = stuff[i]; } } return n; }</pre>	<p>基于对象工作模式</p> <p>属性拷贝模式</p>	<p>一种混合插入式 (mixin-style) 继承实现。</p> <p>它会按照父对象的出现顺序依次对它们执行属性全拷贝</p>
10	寄生继承法	<pre>function parasite(victim) { var that = object(victim); that.more = 1; return that; }</pre>	<p>基于对象工作模式</p> <p>使用原型链模式</p>	<p>该方法通过一个类似构造器的函数来创建对象。</p> <p>该函数会执行相应的对象拷贝，并对其进行扩展，然后返回该拷贝</p>
11	构造器借用法	<pre>function Child() { Parent.apply(this, arguments); }</pre>	<p>基于构造器工作模式</p>	<p>该方法可以只继承父对象的自身属性，也可以与方法 1 结合使用，以便从原型中继承相关内容。</p> <p>它便于我们的子对象继承某个对象的具体属性（并且还有可能是引用类属性）时，选择最简单的处理方式</p>
12	构造器借用与属性拷贝法	<pre>function Child() { Parent.apply(this, arguments); } extend2(Child, Parent);</pre>	<p>使用构造器工作模式</p> <p>使用原型链模式</p> <p>属性拷贝模式</p>	<p>该方法是方法 11 与方法 4 的结合体，</p> <p>它允许我们在不重复调用父对象构造器的情况下同时继承其自身属性和原型属性</p>

面对这么多选择,我们应该如何做出正确的选择呢?事实上这取决于我们的设计风格、性能需求、具体项目任务及团队。例如,您是否更习惯于从类的角度来解决问题?那么基于构造器工作模式更适合您。或者您可能只关心该“类”的某些具体实例,那么可能使用基于对象的模式更合适。

那么,继承实现是否只有这些呢?当然不是,我们可以从上面的表中选择任何一种模式,也可以混合使用它们,甚至我们也可以写出我们自己的方法。重点在于必须理解并属性这些对象、原型以及构造器的工作方式,剩下的就简单了。

6.15 案例学习: 图形绘制

下面,让我们用一个更为具体的继承应用示例来作为本章的结尾吧。示例的任务是计算各种不同图形的面积和边界,然后将它们绘制出来。并且,要求在这过程中尽可能地实现代码重用。

6.15.1 分析

首先,我们要将所有对象的公共部分定义成一个构造器,即 Shape。然后我们基于这个构造器分别构建我们的 Triangle、Rectangle 和 Square 构造器,它们将全部继承于 Shape。其中, Square 实际上可以被当做一个长宽度相等 Rectangle,因此当我们构建 Square 时可以直接重用 Rectangle。

下面,我们来定义 Shape 对象,首先,我们要定义一个带 x 、 y 坐标的 point 对象。因为图形一般都是由若干个 point 组成的。例如,定义一个 Triangle 对象需要三个 point 对象,而定义一个 Rectangle 对象(即使是尽可能简单)也至少需要定义一个 point 对象和其长宽度。而图形的周长则是其长宽度的综合,而一个图形的面积则应该有这些图形自己来实现。

这样一来, Shape 体系中的共有功能主要包括:

- ◆ 一个能根据给定的 point 绘制出图形 draw() 方法。
- ◆ 一个 getParameter() 方法。
- ◆ 一个用存储 point 对象的数组属性。
- ◆ 其他必须的属性与方法。

关于绘制部分,我们还将用到<canvas>标签。尽管 IE 并不支持这一特性,但管它呢,这不过是个实验。

当然，还有两个辅助构造器不能不提——即 Point 和 Line。其中，Point 用于定义图形，而 Line 则用于计算给定两个点之间的距离。

当然，读者也可以到 <http://www.phpied.com/files/canvas/> 中去运行该工作示例，只需打开 Firebug 控制台，然后按部就班新建图形即可。

6.15.2 实现

首先，我们要在空白的 HTML 页面中添加一个 canvas 标签：

```
<canvas height="600" width="800" id="canvas" />
```

然后再插入 <script> 标签，我们的 JavaScript 代码就要放在这里：

```
<script type="text/javascript">
  // ... code goes here
</script>
```

下面，我们来实现 JavaScript 部分的工作。

首先是辅助构造器 Point，大体实现如下：

```
function Point(x, y) {
  this.x = x;
  this.y = y;
}
```

要注意的是，该画布（即 canvas）的坐标系是从 $x=0$ 、 $y=0$ 这点开始的，即图 6-3 中的左上角，而右下角的坐标则是 $x=800$ 、 $y=600$ 。

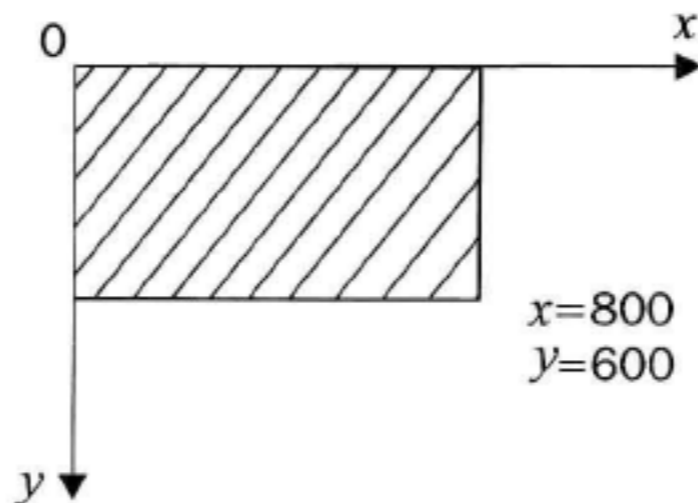


图 6-3

接下来，轮到构造器 Line 了，它将会根据勾股定理 $a^2 + b^2 = c^2$ 这样的公式计算出给定两点之间的直线距离（假设这两点位于一个直角三角形的斜边两端）。

```
function Line(p1, p2) {
```

```
    this.p1 = p1;
    this.p2 = p2;
    this.length = Math.sqrt(Math.pow(p1.x - p2.x, 2) + Math.pow(p1.y -
p2.y, 2));
}
```

下一步,我们就可以进入 Shape 构造器的定义了。该构造器需要有一个自己的 points 属性 (以及链接这些 point 的 Lines 属性)。另外我们还需要一个初始化方法 init(), 用于定义其原型。

```
function Shape() {
    this.points = [];
    this.lines = [];
    this.init();
}
```

接下来的一大部分内容就是定义 Shape.Prototype 的方法了。下面我们用对象标识法来定义这些方法。其中,我们对每个方法做了相关的注释说明。

```
Shape.prototype = {
    // reset pointer to constructor
    constructor: Shape,
    // initialization, sets this.context to point
    // to the context of the canvas object
    init: function() {
        if (typeof this.context === 'undefined') {
            var canvas = document.getElementById('canvas');
            Shape.prototype.context = canvas.getContext('2d');
        }
    },
    // method that draws a shape by looping through this.points
    draw: function() {
        var ctx = this.context;
        ctx.strokeStyle = this.getColor();
        ctx.beginPath();
        ctx.moveTo(this.points[0].x, this.points[0].y);
        for(var i = 1; i < this.points.length; i++) {
            ctx.lineTo(this.points[i].x, this.points[i].y);
        }
        ctx.closePath();
        ctx.stroke();
    },
}
```

```
// method that generates a random color
getColor: function() {
    var rgb = [];
    for (var i = 0; i < 3; i++) {
        rgb[i] = Math.round(255 * Math.random());
    }
    return 'rgb(' + rgb.join(',') + ')';
},
// method that loops through the points array,
// creates Line instances and adds them to this.lines
getLines: function() {
    if (this.lines.length > 0) {
        return this.lines;
    }
    var lines = [];
    for(var i = 0; i < this.points.length; i++) {
        lines[i] = new Line(this.points[i], (this.points[i+1]) ?
            this.points[i+1] : this.points[0]);
    }
    this.lines = lines;
    return lines;
},
// shell method, to be implemented by children
getArea: function() {},
// sums the lengths of all lines
getPerimeter: function(){
    var lines = this.getLines();
    var perim = 0;
    for (var i = 0; i < lines.length; i++) {
        perim += lines[i].length;
    }
    return perim;
}
}
```

接着是子对象构造器，先从 **Triangle** 开始：

```
function Triangle(a, b, c){
    this.points = [a, b, c];
    this.getArea = function(){
        var p = this.getPerimeter();
        var s = p / 2;
        return Math.sqrt(
```

```

        s
        * (s - this.lines[0].length)
        * (s - this.lines[1].length)
        * (s - this.lines[2].length)
    );
};
}

```

在 Triangle 构造器中，我们会将其接收到的三个 point 对象赋值给 this.points（此为对象自身属性）。然后再利用海伦公式（Heron's formula）^①实现其 getArea() 方法，公式如下：

$$\text{Area} = s(s-a)(s-b)(s-c)$$

其中，s 为半周长（即周长除以 2）。

接下来轮到 Rectangle 构造器了，该对象所接收的参数是一个 point 对象（即左上角位置）和两边的长度。然后再以该 point 起点，自行填充其 points 数组。

```

function Rectangle(p, side_a, side_b){
    this.points = [
        p,
        new Point(p.x + side_a, p.y),           // top right
        new Point(p.x + side_a, p.y + side_b), // bottom right
        new Point(p.x, p.y + side_b)           // bottom left
    ];
    this.getArea = function() {return side_a * side_b;};
}

```

最后一个子对象构造器是 Square。由于 Square 是 Rectangle 的一种特例，所以对于它的实现，我们可以重用 Rectangle，而其中最简单的莫过于构造器借用法了。

```

function Square(p, side){
    Rectangle.call(this, p, side, side);
}

```

到目前为止，所有构造器的实现都已经完成。让我们好好掌握这种继承关系吧，几乎所有的仿传统模式（即工作方式是基于构造器而非对象的模式）都是这样做的。下面，让

^①译者注：海伦公式（Heron's formula 或 Hero's formula），又译希罗公式、希伦公式、海龙公式，此公式能利用三角形的三条边长来求取三角形面积。最早出自《Metrica》一书，后者是一部古代数学知识的结集，相传由数学家希罗在公元 60 年前后提出。

我们来试着将其修改为原型链模式，并提供一个简化版本（第一种方法本章之前已经讨论过了）。在该模式中，我们需要新建一个父对象实体，然后直接将其设置为子对象的原型。这样一来，我们就没有必要为每个子对象都创建新的实体了——因为它们可以通过原型实现完全共享。

```
(function () {  
    var s = new Shape();  
    Triangle.prototype = s;  
    Rectangle.prototype = s;  
    Square.prototype = s;  
})();
```

6.15.3 测试

下面我们来绘制一些图形，测试一下代码。首先来定义 `Triangle` 对象的三个 `point`:

```
>>> var p1 = new Point(100, 100);  
>>> var p2 = new Point(300, 100);  
>>> var p3 = new Point(200, 0);
```

然后将这三个 `point` 传递给 `Triangle` 构造器，以创建一个 `Triangle` 实例:

```
>>> var t = new Triangle(p1, p2, p3);
```

接着，我们就可以调用相关的方法在画布上绘制出三角形，并计算出它的面积与周长:

```
>>> t.draw();  
>>> t.getPerimeter()  
482.842712474619  
>>> t.getArea()  
10000.000000000002
```

接下来是 `Rectangle` 的实例化:

```
>>> var r = new Rectangle(new Point(200, 200), 50, 100);  
>>> r.draw();  
>>> r.getArea()  
5000  
>>> r.getPerimeter()  
300
```

最后是 Square:

```
>>> var s = new Square(new Point(130, 130), 50);
>>> s.draw();
>>> s.getArea()
2500
>>> s.getPerimeter()
200
```

如果想给这些图形绘制增加一些乐趣，我们也可以像下面这样，在绘制 Square 时偷个懒，重用 triangle 的 point。

```
>>> new Square(p1, 200).draw()
```

最终测试结果如图 6-4 所示:



图 6-4

6.16 练习题

利用上面的画布示例展开实践，尝试各种不同的东西，例如：

1. 绘制出一些 Triangle、Square、Rectangle 图形。
2. 添加更多的图形构造器，例如 Trapezoid、Rhombus、Kite、Diamond 以及 Pentagon 等。如果您还想对 canvas 标签有更多的了解，也可以创建一个 Circle 构造器，该构造器需要您重写父对象的 draw() 方法。
3. 考虑一下，是否还有其他方式可以实现并使用这些类型继承关系？
4. 请选择一个子对象能通过 uber 属性访问的方法，并为其添加新的功能，使得父对象可以追踪到该方法所属的子对象。例如，或许我们可以在父对象中建立一个用于存储其所有子对象的数组属性。

第 7 章

浏览器环境

众所周知，JavaScript 程序是不能脱离宿主环境而独立运行的。到目前为止，本书所讨论的大部分内容都是围绕着 ECMAScript/JavaScript 核心标准，以及多种不同的宿主环境来展开的。下面，就让我们将焦点转移到浏览器这个当下最流行、也是最常见的 JavaScript 宿主环境上来吧。在这一章中，我们将学习以下内容：

- ◆ BOM (Browser Object Model, 即浏览器对象模型)
- ◆ DOM (Document Object Model, 即文档对象模型)
- ◆ 浏览器事件监听
- ◆ XMLHttpRequest 对象

7.1 在 HTML 页面中引入 JavaScript 代码

要想在 HTML 页面中引入 JavaScript 代码，我们需要用到 `<script>` 标签：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html>
  <head>
    <title>JS test</title>
    <script type="text/javascript" src="somefile.js"></script>
  </head>
  <body>
    <script type="text/javascript">
      var a = 1;
      a++;
    </script>
  </body>
</html>
```

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

在上面的示例中，第一个<script>标志引入的是一个外部文件 `somefile.js`，其中包含了相关的 JavaScript 代码。而第二个<script>标签则是直接在 HTML 页面中引入了 JavaScript 代码。这两个标签都设置有 `type` 属性。该属性是 XHTML1.0 标准所规定的内容，尽管没有它代码一样可以工作。但在浏览器执行 JavaScript 代码的过程中，如果我们在页面中设置了这个属性，可以保证代码的执行顺序。也就是说，这可以使我们在 `somefile.js` 中所定义的变量，在第二个<script>区块中依然可用。

7.2 概述：BOM 与 DOM

通常情况下，页面中的 JavaScript 代码都有一系列可以访问的对象，它们可以分成两组：

- ◆ 当前载入页面所拥有的对象（页面有时也可以叫做文档）。
- ◆ 页面以外事物所拥有的对象（即浏览器窗口和桌面屏幕）。

对于第一个对象集合，我们称之为文档对象模型（即 DOM），而第二个则叫做浏览器对象模型（即 BOM）。

其中，DOM 是一个标准化组件，由世界万维网联合会（W3C）负责制定，并拥有多个不同的版本。这些版本我们称之为 level，例如 DOM level 1、DOM level 2，以及迄今为止的最后一个版本——DOM level 3。尽管，现代浏览器对这些标准级别的实现程度各不相同，但一般来说，它们基本上都完全实现了 DOM level 1。DOM 实际上是标准化之后才出现的产物，在这之前，各浏览器都有各自访问文档的实现。其中，有相当一部分是旧时代遗留下来的产品（即 W3C 标准产生之前所实现的部分），我们将其统称为 DOM 0。尽管，实际上并没有一个叫做 DOM level 0 的标准存在，但其中相当的一部分已经成了事实上的标准，因为几乎所有的主流浏览器对此提供了全面的支持，也正因为如此，它们中的一些内容也被写入 DOM level 1 标准。至于其他在 DOM level 1 中找不到的 DOM 0 的内容，都属于特定浏览器的特性，这里就不必讨论了。

而 BOM 则不属于标准化组件。与 DOM 0 相似，它的一部分对象集得到了所有主流浏览器的支持，而另一部分则属于浏览器特性。

本章将只讨论 BOM 和 DOM level 1 中跨浏览器的那部分子集。但即便是这些“安全”

的子集也是一个很大的话题，也不是本书所能完全覆盖的，您可以参考以下资源：

- ◆ 由 Mozilla 公司所提供的 Firefox DOM 参考资料：http://developer.mozilla.org/en/docs/Gecko_DOM_Reference
- ◆ Microsoft 在线文档：[http://msdn2.microsoft.com/en-us/library/ms533050\(vs.85\).aspx](http://msdn2.microsoft.com/en-us/library/ms533050(vs.85).aspx)
- ◆ W3C 中的 DOM 技术参考：<http://www.w3.org/DOM/DOMTR>

7.3 BOM

BOM（即浏览器对象模型）是一个用于访问浏览器和计算机屏幕的对象集合。我们可以通过全局对象 `window` 和 `window.screen` 来访问这些对象。

7.3.1 window 对象再探

正如您所知，在 JavaScript 中，每个宿主环境都有一个全局对象。具体到浏览器环境中，这就是 `window` 对象了。环境中所有的全局变量实际上都是该对象的属性。

```
>>> window.somevar = 1;
1
>>> somevar
1
```

同样的，所有的 JavaScript 核心函数（即我们在第 2 章中所讨论的）也都只是 `window` 对象的方法。

```
>>> parseInt('123a456')
123
>>> window.parseInt('123a456')
123
```

除了作为全局对象以外，`window` 对象还有另一个作用，就是提供各自关于浏览器环境的私有数据。例如 `window` 对象中的各种 `frame`、`iframe`、弹出窗以及浏览器标签页等对象。

下面，我们来看一些 `window` 对象中与浏览器有关的属性。当然，这些属性在各个浏览器的表现中可能各不相同，所以我们将尽量局限于那些为现代主流浏览器所共同实现的、最为可靠的属性。

7.3.2 window.navigator

`navigator` 是一个用于反映浏览器本身及其功能信息的对象。例如其中的 `navigator.userAgent` 是一个用于浏览器识别的长字符串。也就是说在 Firefox 中，我们将得到如下信息：

```
>>> window.navigator.userAgent
"Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.8.1.12) Gecko/20080201
Firefox/2.0.0.12"
```

而在 Microsoft 的 Internet Explorer 中，`userAgent` 返回的字符串则是：

```
Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; .NET CLR 1.1.4322; .NET
CLR 2.0.50727; .NET CLR 3.0.04506.30)
```

由于各种浏览器的功能是各不相同的，开发人员需要根据 `userAgent` 字符串来识别不同的浏览器，并提供不同版本的代码。例如在下面的代码中，我们就是通过搜索“MSIE”子串来识别 Internet Explorer：

```
if (navigator.userAgent.indexOf('MSIE') !== -1) {
    // this is IE
} else {
    // not IE
}
```

当然了，最好还是不要过份依赖于这种用户代理字符串，特性监听法（也叫做功能检测法）无疑是更好的选择。因为这种字符串很难追踪到所有的浏览器以及其各种版本。所以，直接检查我们使用的功能在用户浏览器中是否存在要简单得多，例如：

```
if (typeof window.addEventListener === 'function') {
    // feature is supported, let's use it
} else {
    // hmm, this feature is not supported, will have to
    // think of another way
}
```

另外，还有一个原因也促使我们避免使用这种用户代理检测法，即在某些浏览器中，用户是可以对该字符串进行修改，并将其伪装成其他浏览器的。

7.3.3 Firebug 的备忘功能

Firebug 控制台提供了一种便利的对象检索功能，其功能涵盖了 BOM 和 DOM 中所有

的对象。因此通常情况下，我们只要在控制台中输入：

```
>>> navigator
```

然后单击其结果，或者我们也可以输入：

```
>>> console.dir(navigator)
```

都会得到一份完整的属性列表，以及这些属性的当前值（见图 7-1）。

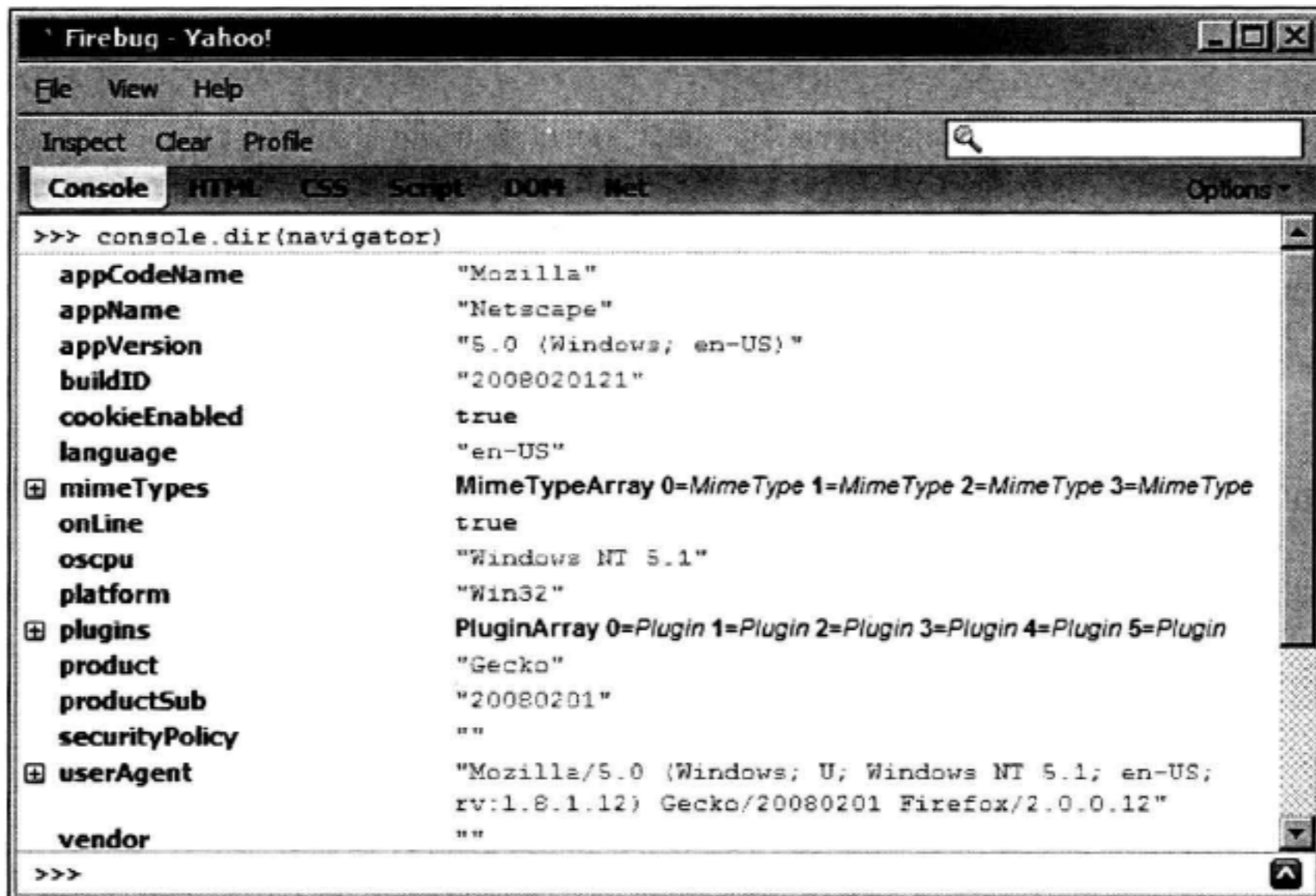


图 7-1

7.3.4 window.location

location 属性是一个用于存储当前载入页面 URL 信息的对象。例如其中的 location.href 显示的是完整的 URL，而 location.hostname 则只显示相关的域名信息。下面，我们通过一个简单的循环列出 location 对象的完整属性列表。

假设我们的页面 URL 如下：

```
http://search.phpied.com:8080/search?p=javascript#results
```

```
>>> for(var i in location) {console.log(i + ' = "' + location[i] +
    '"') }
```

```
href = "http://search.phpied.com:8080/search?p=javascript#results"
```

```
hash = "#results"
```

```
host = "search.phpied.com:8080"
```

```
hostname = "search.phpied.com"  
pathname = "/search"  
port = "8080"  
protocol = "http:"  
search = "?p=javascript"
```

另外，location 对象还提供了三个方法，分别是 reload()、assign() 和 replace()。

需要注意的是，页面导航存在着许多种不同的方式，下面列出的只是其中一小部分：

```
>>> window.location.href = 'http://www.packtpub.com'  
>>> location.href = 'http://www.packtpub.com'  
>>> location = 'http://www.packtpub.com'  
>>> location.assign('http://www.packtpub.com')
```

replace() 方法的作用与 assign 基本相同，只不过它不会在浏览器的历史记录表中留下记录：

```
>>> location.replace('http://www.yahoo.com')
```

另外，如果我们想重新载入某个页面，可以调用：

```
>>> location.reload()
```

或者，也可以让 location.href 属性再次指向自己，比如：

```
>>> window.location.href = window.location.href
```

还可以再简化一下：

```
>>> location = location
```

7.3.5 window.history

window.history 属性允许我们在同一个浏览器会话 (session) 中存储有限数量的访问记录。例如，我们可以通过以下方式来查看用户在这之前访问了多少页面：

```
>>> window.history.length  
5
```

当然，这种访问是受限制的，例如像下面这样是不被允许的：

```
>>> window.history[0]
```

但是我们可以在当前用户会话中对各页面进行来回切换，就像您在浏览器中单击后退/前进按钮一样：

```
>>> history.forward()
>>> history.back()
```

另外，我们也可以使用 `history.go()` 来实现页面跳转，例如，下面的调用效果和 `history.back()` 相同：

```
>>> history.go(-1);
```

接下来是后退两页的情况：

```
>>> history.go(-2);
```

如果想重载当前页，可以这样：

```
>>> history.go(0);
```

7.3.6 window.frames

`window.frames` 属性是当前页面中所有 `frame` 元素的集合。要注意的是，这里并没有对 `frame` 和 `iframe` 做出区分。而且，无论当前页面中是否存在 `frame` 元素，`window.frames` 属性总是存在的，并指向了 `window` 对象本身。

```
>>> window.frames === window
true
```

我们可以通过检查其 `length` 属性来了解当前页面中是否存在 `frame` 元素：

```
>>> frames.length
1
```

另外，每个 `frame` 元素都包含一个别的页面，这些页面也都拥有属于自己的全局 `window` 对象。例如，我们的页面中有一个 `iframe` 元素：

```
<iframe name="myframe" src="about:blank" />
```

在这种情况下，如果想访问 `iframe` 元素的 `window` 对象，可以选择下面方式中的任何一种：

```
>>> window.frames[0]
>>> window.frames[0].window
>>> frames[0].window
>>> frames[0]
```

通过父级页面，我们可以到访问子 `frame` 元素的属性。例如，您可以用以下方式来实现 `frame` 元素的重载：

```
>>> frames[0].window.location.reload()
```

同样的，我们也可以通过子元素来访问父级页面：

```
>>> frames[0].parent === window
true
```

另外，通过一个叫做 `top` 的属性，我们可以访问到当前最顶层页面（即包含所有其他 `frame` 元素的页面）中的任何 `frame` 元素：

```
>>> window.frames[0].window.top === window
true
>>> window.frames[0].window.top === window.top
true
>>> window.frames[0].window.top === top
true
```

除此之外还有一个 `self` 属性，它的作用与 `window` 基本相同。

```
>>> self === window
true
>>> frames[0].self == frames[0].window
true
```

如果 `frame` 元素拥有名字属性，我们就可以丢开索引，而通过名字来访问该 `frame`。

```
>>> window.frames['myframe'] === window.frames[0]
true
```

7.3.7 window.screen

`screen` 属性所提供的是浏览器以外的桌面信息。例如，`screen.colorDepth` 属性所包含的是当前显示器的色位（表示的是颜色质量）。这对于某些统计化操作来说，会非常有用。

```
>>> window.screen.colorDepth
```


另外，我们还可以查看当前屏幕的实际状态（如分辨率）：

```
>>> screen.width
1440
>>> screen.availWidth
1440
>>> screen.height
900
>>> screen.availHeight
847
```

其中，`height` 和 `availHeight` 之间的不同之处在于，`height` 指的是总分辨率，而 `availHeight` 指的是除去操作系统菜单（例如 `windows` 的任务栏）以外的子区域。同样的，`availWidth` 的情况也是如此。

7.3.8 `window.open()/close()`

在上面我们探索了一些 `windows` 对象中最常见的跨浏览器属性。接下来，我们再看一些方法。其中，`open()` 是一个可以让我们打开新浏览器窗口的方法（即弹出窗）。如今，多数浏览器的策略及其用户设置都会阻止浏览器的弹出窗（以防止这种技术的商业化滥用），但在一般情况下，如果该操作是由用户发起的话，我们就应该允许新窗口弹出。否则，没有用户的允许，我们任何打开新窗口的举动都有可能被阻止。

`window.open()` 方法主要接受以下参数：

- ◆ 要载入新窗口的 URL。
- ◆ 新窗口的名字，用于新窗体 `form` 标签的属性值。
- ◆ 还有一个以逗号分割的功能性列表，例如：
 - 尺寸的可调整性——即是否允许用户调整新窗口大小。
 - 弹出窗的长与宽。
 - 状态栏——用于设置状态栏的可见性。
 - 其他。

而 `window.open()` 方法会返回一个新建浏览器实例的 `window` 对象引用，例如：

```
var win = window.open('http://www.packtpub.com', 'packt',
                      'width=300, height=300, resizable=yes');
```

如您所见，`win` 指向的就是该弹出窗的 `window` 对象。我们可以通过检查 `win` 是否为 `false` 值来判断弹出窗是否被屏蔽了。

`win.close()` 方法则是用来关闭新窗口的。

总而言之，在设置关于打开窗口这方面功能的访问性和可用性时，您最好要有充足的理由。如果我们自己都不想被网站中弹出的窗口骚扰的话，为什么还要将其强加给用户呢？尽管这种做法有它合理的地方，例如填表是为用户提供帮助信息等，但我们完全可以通过在页面中插入浮动的 `<div>` 标签方法来解决这一问题。

7.3.9 `window.moveTo()`、`window.resizeTo()`

继续刚才所谈的“伎俩”，实际上，我们还有许多方法可以让用户开放他们的浏览器设置，允许我们：

- ◆ 调用 `window.moveTo(100, 100)` 将当前浏览器窗口移动到屏幕坐标 $x = 100$, $y = 100$ 的位置（指的是窗口左上角的坐标）。
- ◆ 调用 `window.moveBy(10, -10)` 将窗口的当前位置右移 10 个单位，并同时上移 10 个单位。
- ◆ 调用与前面 `move` 类方法相似的 `window.resizeTo(x, y)` 和 `window.resizeBy(x, y)`，只不过这里做的不是移动位置，而是调整窗口的大小。

但必须再次强调一遍，我们并不建议读者使用这些方法来解决实际问题。

7.3.10 `window.alert()`、`window.prompt()`、`window.confirm()`

在第 2 章中，我们已经接触了 `alert()` 函数。现在我们又知道了该函数只是全局对象的一个方法。也就是说，`alert('Watch out!')` 和 `window.alert('Watch out!')` 这两个函数的执行结果是完全相同的。

`alert()` 并不属于 ECMAScript，但它是一个 BOM 方法。除此之外，BOM 中还有两个方法可以让我们以系统消息的形式与用户进行交互，它们分别是：

- ◆ `confirm()` 方法，它为用户提供了两个选项——**OK** 与 **Cancel**。
- ◆ `prompt()` 方法，它为用户提供了一定的文本输入功能。

下面来看看它们是如何工作的：

```
>>> var answer = confirm('Are you cool?');  
      console.log(answer);
```

如您所见，这段代码会弹出类似这样的窗口（具体的外观还要取决于浏览器和操作系统），如图 7-2 所示。



图 7-2

在这里，我们将会注意到两点：

- ◆ 在我们关闭该窗口之前，Firebug 控制台将会停止接受任何输入，这意味着 JavaScript 代码在此处会暂停执行，以等待用户的回复。
- ◆ 如果单击的是“OK”，方法将会返回 **true**，而如果单击的是“Cancel”或者按 X 图标（也可以按 ESC 键）关闭该窗口则会返回 **false**。

这样一来，我们就可以根据用户的回答来设定了，例如：

```
if (confirm('Are you sure you want to delete this item?')) {  
    // delete  
} else {  
    // abort  
}
```

当然，我们还必须确保在 JavaScript（或者 spiders 搜索引擎）被禁用时能提供些备用方案。

`window.prompt()` 方法呈现给用户的是一个用于输入文本的对话框，例如：

```
>>> var answer = prompt('And your name was?'); console.log(answer);
```

其对话框如图 7-3 所示（在 Windows/Firefox 环境中）：

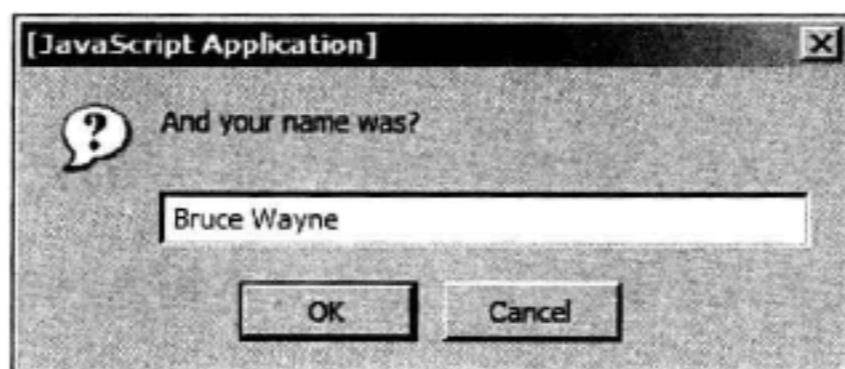


图 7-3

其回复值可能会出现以下情况：

- ◆ 如果我们直接按 **Cancel** 或者 **X** 图标以及 **ESC** 键退出，对话框将会返回 `null`。
- ◆ 如果我们没有输入任何东西就直接单击 **OK** 或回车，对话框将会返回 ""（即空字符串）。
- ◆ 如果我们输入了一些内容之后单击 **OK**（或回车）。对话框就会返回相应的文本字符串。

另外，该函数还可以接受第二个字符串参数，主要用做输入框中的默认值。

7.3.11 `window.setTimeout()`、`window.setInterval()`

`setTimeout()`、`setInterval()` 这两个方法主要被用于某些代码片段的执行调度，其中 `setTimeout()` 用于在指定的毫秒数后执行某段既定代码，而 `setInterval()` 则用于每隔一段毫秒数重新执行这段代码。

下面来看一个在 2 秒（即 2000 毫秒）之后弹出 `alert` 窗口示例：

```
>>> function boo() {alert('Boo!');}
>>> setTimeout(boo, 2000);
4
```

如您所见，该函数返回了一个整数 **4**，该整数是该函数的超时 ID。我们可以用这个 ID 调用 `clearTimeout()` 方法来取消当前的超时。在下面的示例中，如果我们的动作够快，在 2 秒之前取消了超时，`alert` 窗口就永远不会都出现了。

```
>>> var id = setTimeout(boo, 2000);
>>> clearTimeout(id);
```

现在，让我们对 `boo()` 做些改动，换成一种不那么骚扰的方式：

```
>>> function boo() {console.log('boo')};
```

接着，我们在 `setInterval()` 中调用 `boo()`，每 2 秒执行一次，直到我们调用 `clearInterval()` 函数取消相关的执行调度为止。

```
>>> var id = setInterval(boo, 2000);
boo
boo
boo
```

```
boo  
boo  
boo  
>>> clearInterval(id)
```

要注意的是，上面两个函数的首参数都是一个指向回调函数的指针。该参数是一个可以被 `eval()` 函数执行的字符串。但 `eval()` 的危险之处是众所周知的，因此它是应该尽量被避免的。那么，我们怎么传递参数给该函数呢？在这种情况下，最好还是将相关的函数调用封装成另一个函数。

例如，下面代码在语法上是正确的，但做法并不值得推荐：

```
var id = setInterval("alert('boo, boo')", 2000);
```

显然我们还有更合适的选择：

```
var id = setInterval(  
    function() {  
        alert('boo, boo')  
    }, 2000  
);
```

7.3.12 window.document

`window.document` 是一个 BOM 对象，表示的是当前所载入的文档（即页面）。但它的方法和属性同时也属于 DOM 对象所涵盖的范围。现在，让我们深吸一口气（也许在本章末尾的练习题中，我们还会再次涉及 BOM 的），深入 DOM 领域中去吧！

7.4 DOM

简而言之，DOM（即文档对象模型）是一种将 XML 或 HTML 文档解析成树形节点的方法。通过 DOM 的方法与属性，我们就可以访问到页面中的任何元素，并进行元素的修改、删除以及添加等操作。同时，DOM 也是一套语言独立的 API（Application Programming Interface，即应用程序接口）体系，它不仅在 JavaScript 中有相关的实现，在其他语言中也有实现。例如，我们可以在服务器端用 PHP 的 DOM 实现（<http://php.net/dom>）来产生相关的页面。

下面我们来看一个具体的 HTML 页面：

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html>
  <head>
    <title>My page</title>
  </head>
  <body>
    <p class="opener">first paragraph</p>
    <p><em>second</em> paragraph</p>
    <p id="closer">final</p>
    <!-- and that's about it -->
  </body>
</html>

```

我们来看页面中的第二段（<p>second paragraph</p>），首先看到的是 p 标签，它包含在 body 标签中。因此，我们可以说 body 是 p 的父节点，而 p 是一个子节点。同理，页面中的第一段和第三段也都是 body 的子节点，同时是第二段的兄弟节点。而 em 标签又是第二个 p 标签的子节点，也就是说 p 是它的父节点。如果我们将这些父子关系图形化，就会看到一个树状族谱（见图 7-4），我们将其称之为 DOM 树。

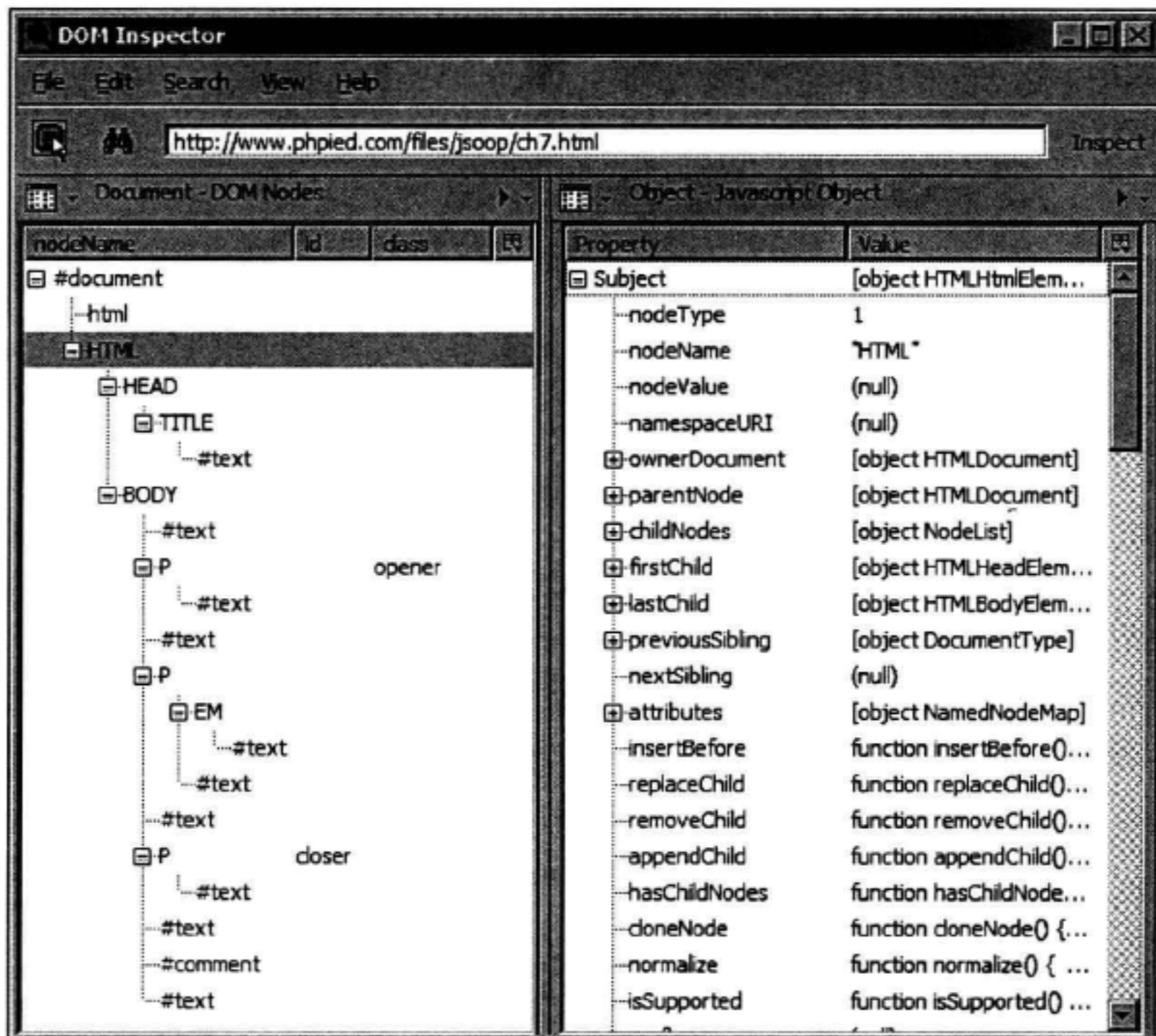


图 7-4

从图 7-4 中可以看到，页面中所有的标签都可以以树节点的形式显示出来。即便是分散在各处的单词 `#text` 实际上也是一个节点，只不过类型有所不同——它属于文本节点。例如 `EM` 标签中的 `#text` 的内容就是单词“second”。另外，空白符也会被认为是文本节点，这就是为什么 `body` 标签与首个 `p` 标签之间会有一个 `#text` 节点，它实际上并不包含任何文本，只有空格符而已。最后，HTML 中的注释也会被认为是一个树上节点，也就是说，包括在 `<!--` 和 `-->` 标签之间所有的 HTML 注释，都会被认为是一个 `#comment` 节点。

另外，图 7-4 所显示的是 Firefox 的 **DOM Inspector** 扩展，该扩展在默认情况下是不安装的。因此，如果我们运行的是 Firefox 2，可能就需要重新安装一下现有的 Firefox（这并不会造成我们的偏好设置和插件丢失），并在安装过程中选择 **Custom**（自定义安装）而不是 **Standard**（标准安装），然后选中 **DOM Inspector** 选项。而如果我们运行的是 Firefox 3，就可以以插件的形式来获得 **DOM Inspector**，该插件的地址是 <https://addons.mozilla.org/en-us/firefox/addon/6622>。

一旦安装完成，我们就可以打开 **Tools** 菜单，并单击 **DOM Inspector** 来打开该功能。

在 **DOM Inspector** 中，左侧所显示的就是 DOM 树，而右侧显示的则是被选中节点的具体信息。之前截图中所显示的是 **JavaScript Object** 面板中的 HTML 节点。该面板并不是 **DOM Inspector** 的默认视图，我们可以通过单击窗口上面的“Views”图标来打开它，如图 7-5 所示。

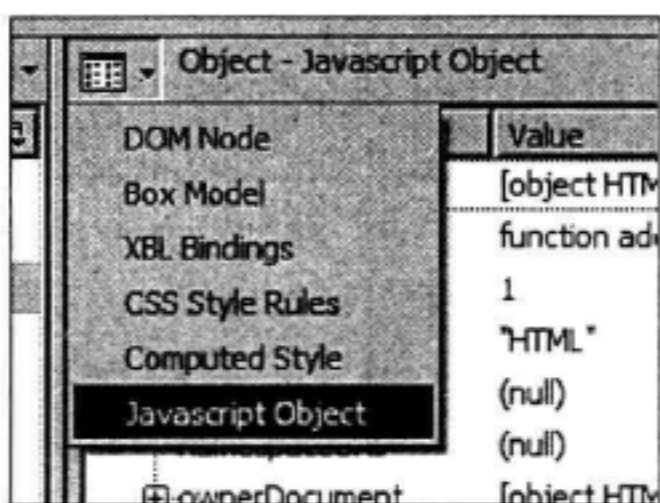


图 7-5

DOM 树中的每一个节点都是一个对象，并且 **DOM Inspector** 的 **JavaScript Object** 视图中列出这些对象所有的方法与属性。另外，我们还可以查看这些对象各自都是由哪些构造器函数来创建的。尽管这种功能在日常工作中并不是很有用，但了解这些还是很有意思的。例如，`window.document` 是由 `HTMLDocument()` 构造器创建的，用来代表 `head` 标签的对象则是由 `HTMLHeadElement()` 负责创建的，等等。但是，这些构造器并不能被用来创建对象。

7.4.1 Core DOM 与 HTML DOM

在接触更有实际意义的示例之前，我们还需要最后做一次概念性的梳理。众所周知，

DOM 既能解析 XML 文档，也能解析 HTML 文档。实际上，HTML 文档本身也可以被当做一种特殊的 XML 文档。因此，我们可以将 DOM Level 1 中用于解析所有 XML 文档的那部分称之为 Core DOM。而将在 Core DOM 基础上进行扩展的那部分称之为 HTML DOM。当然，HTML DOM 并不适用于所有的 XML 文档，它只适用于 HTML 文档。下面，就让我们来看一些属于 Core DOM 和 HTML DOM 构造器示例，如表 7-1 所示。

表 7-1

构造器	父级构造器	所属组件	注释说明
Node		Core DOM	树上所有的节点都属于 Node
Document	Node	Core DOM	Document 对象，主要用于表示 XML 文档项目
HTMLDocument	Document	HTML DOM	即 window.document 或其简写 document 所指向的对象。是前一对象的 HTML 定制版，应用十分广泛
Element	Node	Core DOM	在源文档中，每一个标签都是一个元素，所以， <code><p></p></code> 标签也叫做“p 元素”
HTMLElement	Element	HTML DOM	这是一个通用性构造器，所有与 HTML 元素有关的构造器都继承于该对象
HTMLBodyElement	HTMLElement	HTML DOM	用于表示 <code><body></code> 的标签的元素
HTMLLinkElement	HTMLElement	HTML DOM	代表一个 A 元素（即 <code></code> 标签）
.....	HTMLElement	HTML DOM	剩下所有的 HTML 页面元素
CharacterData	Node	Core DOM	文本处理类的通用性构造器
Text	CharacterData	Core DOM	即插入在标签中的文本节点。例如在 <code>second</code> 这句代码中，就包含了 EM 元素节点和值为“second”的文本节点
Comment	CharacterData	Core DOM	即 <code><!--any comment--></code>

(续表)

构造器	父级构造器	所属组件	注释说明
Attr	Node	Core DOM	用于代表各标签中的属性，例如在代码<p id="closer">中，属性 id 也是一个 DOM 对象，由 Attr() 负责创建
NodeList		Core DOM	即节点列表，是一个用于存储对象，拥有自身 length 属性的数组类对象
NamedNodeMap		Core DOM	其功能与上一个对象相同。不同之处在于，该对象中的元素是通过对象名而不是数字索引来访问的
NamedNodeMap		HTML DOM	其功能也与前两个对象类似，但它是为 HTML 特性量身定制的

当然，这里并没有列出所有的 Core DOM 和 HTML DOM 对象，如果读者想获得完整列表，可以参考链接 <http://www.w3.org/TR/DOM-Level-1/> 中的内容。

现在，我们已经对 DOM 理论背后的实用性有了更深入的理解。在接下来的章节中，我们将继续学习：

- ◆ 访问 DOM 节点
- ◆ 修改 DOM 节点
- ◆ 创建新的 DOM 节点
- ◆ 移除 DOM 节点

7.4.2 DOM 节点的访问

在我们进行表单验证或图片替换这样的操作之前，首先需要访问到这些要检查或修改的元素。幸运的是，访问这些元素的方法有很多，我们既可以使用 DOM 树的方式进行遍历，也可以使用快捷方式进行导航。

当然了，我们最好还是亲自将这些新对象与方法都体验一遍。因此接下来，我们的示例将始终围绕 DOM 一节开头所展示的那个简单文档来展开。需要的话，读者也可以直接通过访问 <http://www.phpied.com/files/jsoop/ch7.html> 来获取该页面。现在，

让我们打开 Firebug 控制台，开始吧。

7.4.2.1 文档节点

`document` 对象给定的就是我们当前所访问的文档。为了对该对象进行进一步探索，我们需要再次用到 Firebug 的备忘功能。下面，在控制台中输入 `document`，然后单击其返回结果（见图 7-6）。

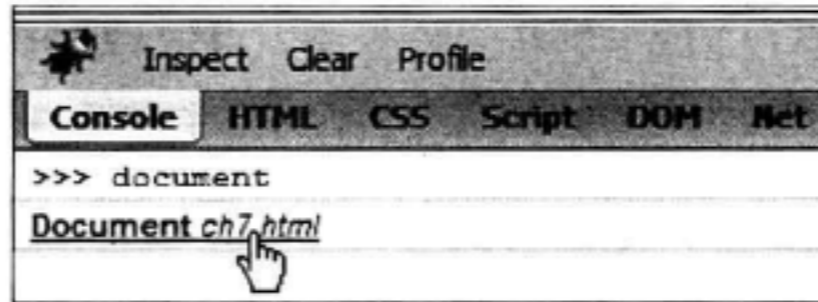


图 7-6

然后，通过浏览器窗口中的 **DOM** 标签，我们就可以浏览到 `document` 对象中所有的属性与方法了（见图 7-7）。

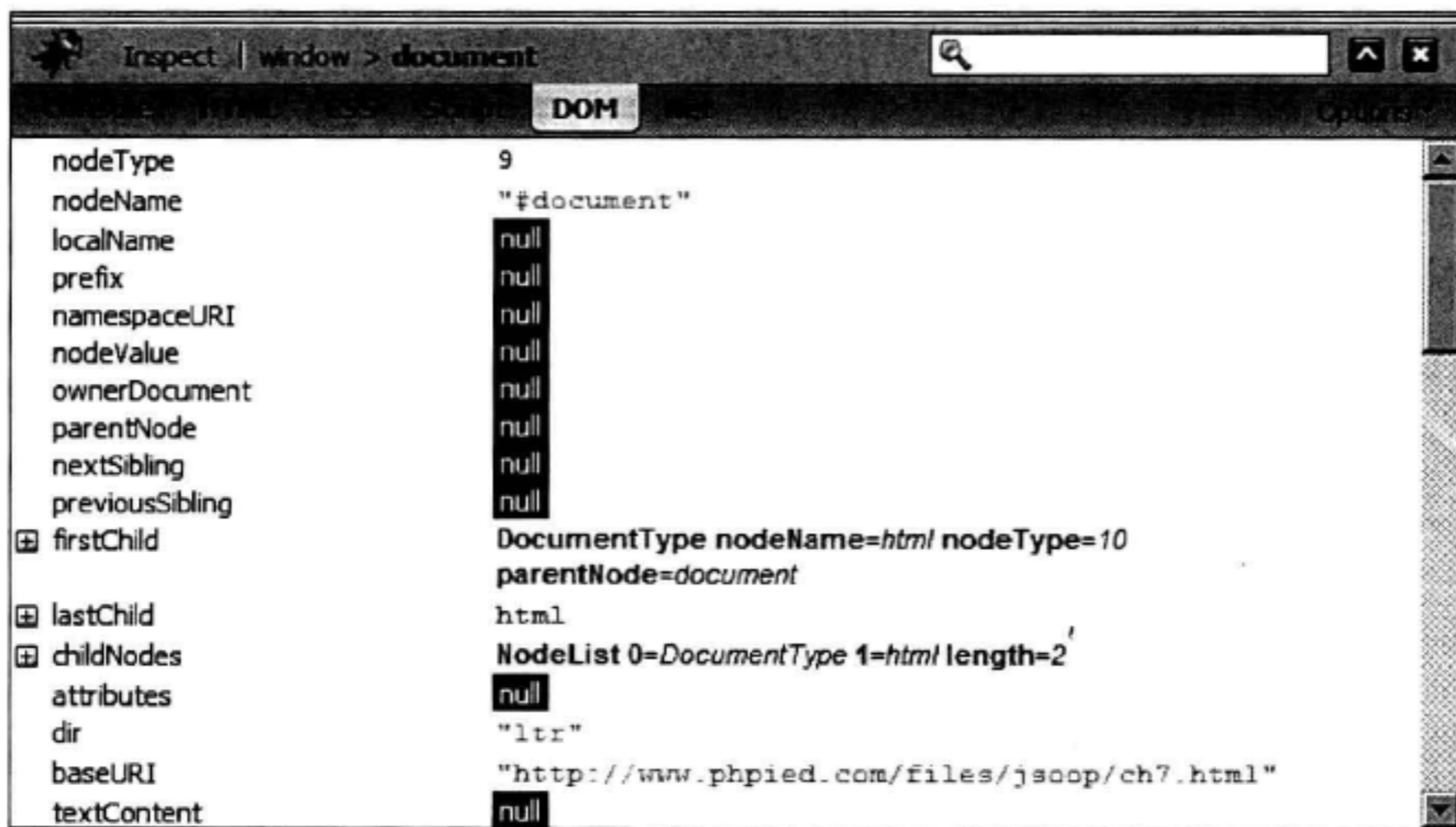


图 7-7

如您所见，图中所有的节点（包括文档类节点、文本类节点、元素类节点以及属性类节点）都拥有属于自己的 `nodeType`、`nodeName` 和 `nodeValue` 等属性。例如：

```
>>> document.nodeType
9
```

在 DOM 中，节点类型有 12 种，每种类型分别用一个整数来表示。正如您所见，`document` 节点的类型是 9，其他最常用的节点类型还有 1（元素）、2（属性）、3（文本）。

另外，这些节点也都有各自的名字。对于 HTML 标签来说，名字一般就是具体标签的名字（即 `tagName` 属性）。而对于文本节点来说，其名字就是 `#text`。那么，`document` 节点呢？我们可以来看一下：

```
>>> document.nodeName
"#document"
```

同时节点也都有各自的节点值，例如，文本节点的值就是它的实际文本。但 `document` 节点中却不包含任何值：

```
>>> document.nodeValue
null
```

7.4.2.2 documentElement

现在，让我们将注意力转移到树结构上来。通常来说，每个 XML 文档都会有一个用于封装文档中其他内容的根节点。具体到 HTML 文档上，这个根节点就是 `<html>` 标签，我们可以通过 `document` 对象的 `documentElement` 属性来访问它。

```
>>> document.documentElement
<html>
```

该属性的 `nodeType` 值为 1（即这是一个元素类节点）：

```
>>> document.documentElement.nodeType
1
```

对于元素类节点来说，其 `nodeName` 和 `tagName` 属性就等于该标签本身的名字：

```
>>> document.documentElement.nodeName
"HTML"
>>> document.documentElement.tagName
"HTML"
```

7.4.2.3 子节点

如果要检查一个节点是否存在子节点，我们可以调用该节点的 `hasChildNodes()` 方法：

```
>>> document.documentElement.hasChildNodes()
true
```

HTML 元素有两个子节点——即 head 元素和 body 元素。我们可以通过该元素中的 childNodes 数组类集合来访问它们。

```
>>> document.documentElement.childNodes.length
2
>>> document.documentElement.childNodes[0]
<head>
>>> document.documentElement.childNodes[1]
<body>
```

任何子节点都可以通过其自身的 parentNode 来访问它的父节点：

```
>>> document.documentElement.childNodes[1].parentNode
<html>
```

下面，我们将 body 元素的引用赋值给一个变量：

```
>>> var bd = document.documentElement.childNodes[1];
```

现在来看看该元素中有几个子节点：

```
>>> bd.childNodes.length
9
```

作为复习，我们再来看看文档的 body 部分：

```
<body>
  <p class="opener">first paragraph</p>
  <p><em>second</em> paragraph</p>
  <p id="closer">final</p>
  <!-- and that's about it -->
</body>
```

那么，body 中的子节点是 9 个吗？让我们来看看，3 个段落加 1 个注释是 4 个节点。然后，这 4 个节点之间的空白处至少有 3 个文本类节点。这样一来，目前为止就有 7 个了。另外，body 与首个 p 标签之间有一个空白处，那是第 8 个，而 comment 元素与 </body> 标签之间也有一个空白处，那又是一个文本类节点。一共是 9 个子节点。

7.4.2.4 属性

由于 body 的第一个子节点是个空白，因此，第二个子节点（索引为 1）是实际上的第

一个段落：

```
>>> bd.childNodes[1]
<p class="opener">
```

我们可以通过元素的 `hasAttributes()` 方法来检查该元素中是否存在属性：

```
>>> bd.childNodes[1].hasAttributes()
true
```

那么，该元素中有几个属性呢？当前示例中只有一个，即 `class` 属性。

```
>>> bd.childNodes[1].attributes.length
1
```

我们可以通过索引值，或属性名来访问一个属性。除此之外，我们也可以调用 `getAttribute()` 方法来获取相关的属性值。

```
>>> bd.childNodes[1].attributes[0].nodeName
"class"
>>> bd.childNodes[1].attributes[0].nodeValue
"opener"
>>> bd.childNodes[1].attributes['class'].nodeValue
"opener"
>>> bd.childNodes[1].getAttribute('class')
"opener"
```

7.4.2.5 访问标签中的内容

下面，我们以第一段为例：

```
>>> bd.childNodes[1].nodeName
"P"
```

我们可以通过该元素的 `textContent` 属性来获取段落中的文本内容。如果我们使用的是不支持 `textContent` 属性的 IE 浏览器，则通过另一个叫 `innerText` 的属性来返回相同的值。

```
>>> bd.childNodes[1].textContent
"first paragraph "
```

另外，我们也可以通过 `innerHTML` 属性来解决上述问题。尽管该属性不属于标准 DOM

组件，但几乎所有的主流浏览器对它提供了支持。该属性会返回指定节点中所有的 HTML 代码。因此，我们也会看到该属性与 `document` 对象之间的不同之处，后者返回的是一个可追踪 DOM 节点树，而前者返回的只是标签字符串而已。但由于其使用极其方便，以至于我们随时都可以使用。

```
>>> bd.childNodes[1].innerHTML
"first paragraph "
```

由于第一段落中只有文本，所以它的 `innerHTML` 值和 `textContent` (IE 中的 `innerText`) 完全相同。但到了第二段落中，由于其中还包含了 `em` 代码，两者的不同就会显现出来：

```
>>> bd.childNodes[3].innerHTML
"<em>second</em> paragraph"
>>> bd.childNodes[3].textContent
"second paragraph"
```

除此之外，获得第一段落的文本内容还有一种方式，即访问 `p` 节点内的文本节点，读取它的 `nodeValue` 属性：

```
>>> bd.childNodes[1].childNodes.length
1
>>> bd.childNodes[1].childNodes[0].nodeName
"#text"
>>> bd.childNodes[1].childNodes[0].nodeValue
"first paragraph "
```

7.4.2.6 DOM 访问的快捷方法

通过 `childNodes`、`parentNode`、`nodeName`、`nodeValue` 以及 `attributes` 这些集合，我们可以在树结构的上下层之间实现自由导航，并处理相关的文档操作。但别忘了，空白处也会成为一个文本类节点，这件事会给这种 DOM 工作方式带来一些不稳定性^①。因为在这种情况下，只要页面发生一些细微变化，我们的脚本或许就不能正常工作了。另外，如果我们访问的树节点深度更深一些，我们或许就要为此写更多的代码。这就是为什么我们需要一些更便捷的方法来解决这个问题。这些方法分别是 `getElementsByTagName()`、`getElementsByName()` 和 `getElementById()`。

^① 译者注：由于代码自动排版等因素，空白处的位置、数量总是不确定的，这会给文本节点的数量带来不确定性和不稳定性。

`getElementsByTagName()` 以标签名(即元素节点的名字)为参数, 返回当前 HTML 页面中所有匹配该标签名的节点集合(一个数组类对象):

```
>>> document.getElementsByTagName('p').length
3
```

列表中的各项可以用中括号法或 `item()` 方法来进行索引(从 0 开始)访问。但我们并不推荐 `item()` 方法, 与之相比, 中括号法显然更具有有一致性, 输入也更为简短:

```
>>> document.getElementsByTagName('p')[0]
<p class="opener">
>>> document.getElementsByTagName('p').item(0)
<p class="opener">
```

下面我们来获取第一个 `p` 元素中的内容:

```
>>> document.getElementsByTagName('p')[0].innerHTML
"first paragraph "
```

Accessing the last p:

```
>>> document.getElementsByTagName('p')[2]
<p id="closer">
```

对于这些元素的属性, 我们可以通过 `attributes` 集合, 或者上面所提到的 `getAttribute()` 方法来进行访问。但我们还可以使用一种更为简便的方法, 即在运行时直接将属性名当做元素对象的属性来访问。例如, 如果想获取其 `id` 属性的值, 我们就可以直接将 `id` 当做一个属性。

```
>>> document.getElementsByTagName('p')[2].id
"closer"
```

当然, 这种方法对于第一段落中的 `class` 属性不起作用。这种异常情况的原因在于“`class`”这个词在 ECMAScript 中被设置成了保留字。对此, 我们只需要改用 `className` 即可:

```
>>> document.getElementsByTagName('p')[0].className
"opener"
```

另外, 我们也可以直接调用 `getElementsByTagName()` 方法来获取页面中的所有元素:

```
>>> document.getElementsByTagName('*').length
9
```

由于在 IE 早期版本中，“*”是一个非法的标签名，所以在这里我们可以改用 IE 所支持的集合 `document.all` 来返回页面中的所有元素，尽管我们事实上很少会用到其中每一个元素。但无论如何，从 IE 7 开始，`document.getElementsByTagName('*')` 总算得到了支持，但它返回的是页面中的所有节点，而不仅仅是元素类节点。

在上面介绍的快捷方式中，还有一个 `getElementById()` 方法。这可能是最常用的元素访问方法了。只要我们为元素们设定好各自的 ID，然后就能轻松地访问这些元素：

```
>>> document.getElementById('closer')
<p id="closer">
```

7.4.2.7 兄弟节点、body 元素及首尾子节点

关于 DOM 树的导航操作，`nextSibling` 与 `previousSibling` 这两个属性也提供了一些便利。例如，如果我们获得了某个元素的引用：

```
>>> var para = document.getElementById('closer')
>>> para.nextSibling
"\n "
>>> para.previousSibling
"\n "
>>> para.previousSibling.previousSibling
<p>
>>> para.previousSibling.previousSibling.previousSibling
"\n "
>>> para.previousSibling.previousSibling.nextSibling.nextSibling
<p id="closer">
```

对于 body 元素来说，以下是一些常用的快捷方式：

```
>>> document.body
<body>
>>> document.body.nextSibling
null
>>> document.body.previousSibling
<head>
```

另外，`firstChild/lastChild` 这两个属性也是非常有用的。其中，`firstChild` 等价

于 `childNodes[0]`，而 `lastChild` 则等价于 `childNodes[childNodes.length - 1]`。

```
>>> document.body.firstChild
"\n "
>>> document.body.lastChild
"\n "
>>> document.body.lastChild.previousSibling
Comment length=21 nodeName=#comment
>>> document.body.lastChild.previousSibling.nodeValue
" and that's about it "
```

下面，我们用一张图来详细解析一下 `body` 与这三个段落之间的族谱关系。当然，为了简单起见。我们在图 7-8 中省略了所有因空白处而形成的文本类节点。

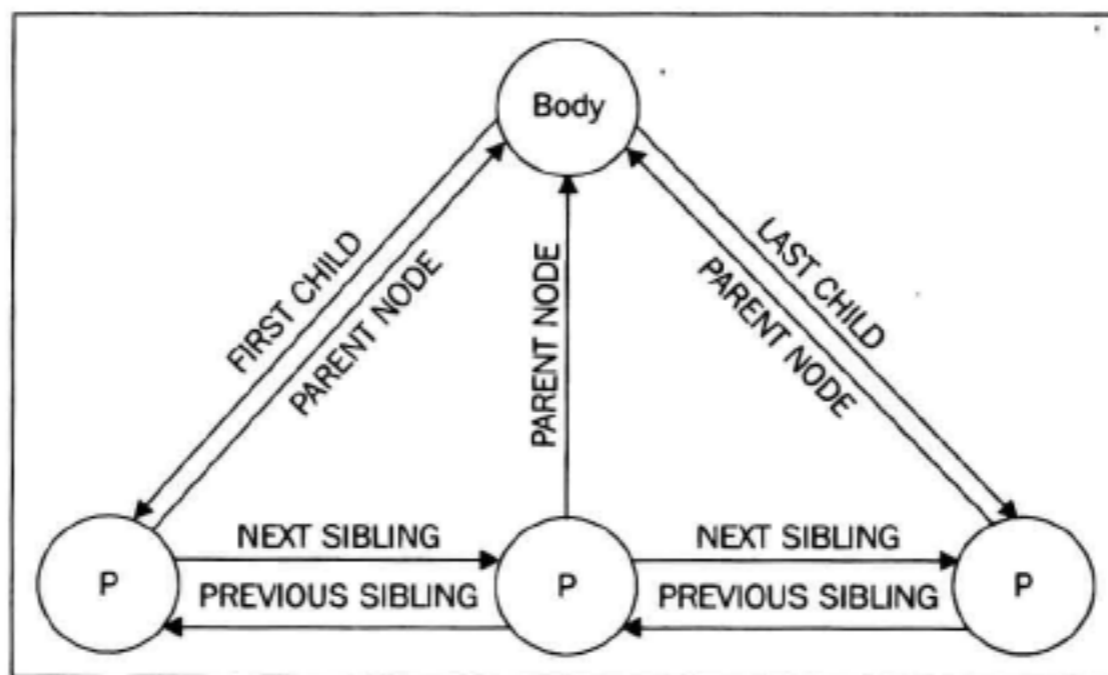


图 7-8

7.4.2.8 遍历 DOM

通过对以上操作的封装，我们可以得到一个用来从既定节点开始，遍历 DOM 树上所有节点的函数：

```
function walkDOM(n) {
  do {
    console.log(n);
    if (n.hasChildNodes()) {
      walkDOM(n.firstChild)
    }
  } while (n = n.nextSibling)
}
```

下面，我们可以来测试一下：

```
>>> walkDOM(document.documentElement)
>>> walkDOM(document.body)
```

7.4.3 DOM 节点的修改

现在，我们已经掌握了许多访问 DOM 树节点及其属性的方法。下面我们来看看如何对这些节点进行修改。

首先，我们将指向最后段落的指针赋值给变量 my：

```
>>> var my = document.getElementById('closer');
```

接下来，我们就能轻松地通过修改对象的 innerHTML 值来修改段落中的文本：

```
>>> my.innerHTML = 'final!!!';  
"final!!!"
```

由于 innerHTML 可以接受任何 HTML 代码串，所以我们可以用它在当前的 DOM 树中再新建一个 em 节点：

```
>>> my.innerHTML = '<em>my</em> final';  
"<em>my</em> final"
```

这样一来，新的 em 节点就成为该树结构的一部分：

```
>>> my.firstChild  
<em>  
>>> my.firstChild.firstChild  
"my"
```

除此之外，我们还可以通过修改既定文本类节点的 nodeValue 属性来实现相关的文本修改：

```
>>> my.firstChild.firstChild.nodeValue = 'your';  
"your"
```

7.4.3.1 修改样式

通常情况下，节点中的内容是不太需要修改的，但这是指演示内容。元素对象中还有一个 style 属性，这是一个用来反映当前 CSS 样式的属性。例如，通过修改某段落的 style 属性，就可以给它加上一个红色的边框：

```
>>> my.style.border = "1px solid red";  
"1px solid red"
```

另外，CSS 属性中的短线（即“-”）在 JavaScript 中是不可用的。对于这种情况，我们只需要直接跳过并将下一个单词的首字母大写即可。例如，padding-top 可以写成 paddingTop、margin-left 可以写成 marginLeft 等，以此类推。

```
>>> my.style.fontWeight = 'bold';  
"bold"
```

即便是那些在初始化时没有被设置的属性，我们也是可以修改的：

```
>>> my.align = "right";  
"right"  
>>> my.name  
>>> my.name = 'myname';  
"myname"  
>>> my.id  
"closer"  
>>> my.id = 'further'  
"further"
```

现在，我们来看看标签在上面所有修改完成之后的样子：

```
>>> my  
<p id="further" align="right" style="border: 1px solid red; font-weight:  
bold;">
```

7.4.3.2 玩转表单

正如之前所述，JavaScript 是一种很好的客户端输入验证方式，它能暂时替我们保存一些将要返回给服务器端的信息。下面，让我们以当下最流行的页面——Google.com 的表单为例，来实际操练一下表单操作（见图 7-9）。

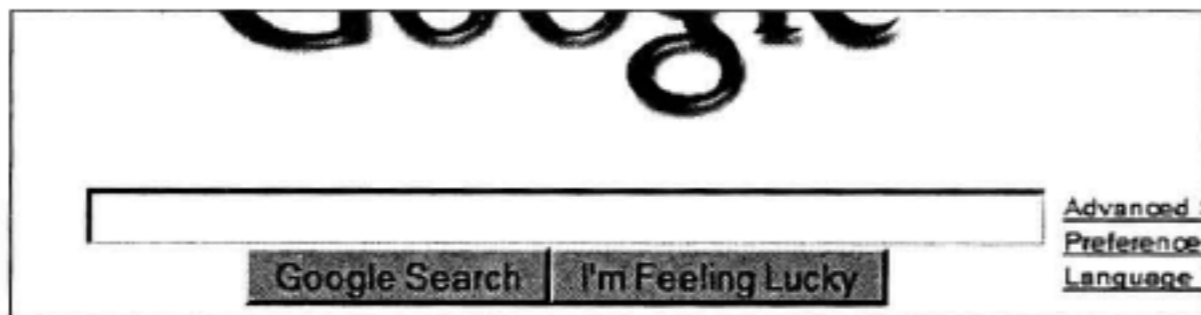


图 7-9

首先，我们要选取所有的 input 字段：

```
>>> var inputs = document.getElementsByTagName('input');  
>>> inputs.length;
```

然后打印出 `inputs[0]`、`inputs[1]` 等元素，这时候，我们会依次看到四个 `input` 对象：首先是一个隐藏字段，接着是一个用于查询的搜索框，最后是一个“**Google Search**”按钮和一个“**I'm Feeling Lucky**”按钮（见图 7-10）。

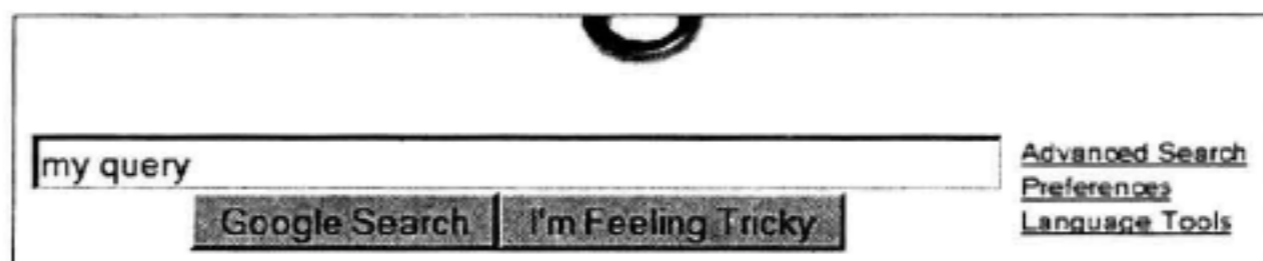


图 7-10

下面是搜索框的访问：

```
>>> inputs[1].name;
"q"
```

另外，我们还可以通过该对象的 `value` 属性来修改搜索内容：

```
>>> inputs[1].value = 'my query';
"my query"
```

现在，让我们来找些乐子吧。先将按钮中的单词“**Lucky**”替换成“**Tricky**”：

```
>>> inputs[3].value = inputs[3].value.replace(/Lu/, 'Tri');
"I'm Feeling Tricky"
```

下面，我们来实现这个“tricky”功能，即令按钮一秒钟显示或隐藏一次。我们可以通过一个叫做 `toggle()` 的简单函数来实现。当该函数每次被调用时，它会自动检查该按钮的 CSS 属性 `visibility` 值，如果为“`hidden`”，则将其设置为“`visible`”。反之亦然。

```
function toggle(){
  var st = document.getElementsByTagName('input')[3].style;
  st.visibility = (st.visibility === 'hidden') ? 'visible': 'hidden';
}
```

当然了，该函数不是靠手动调用的，我们还得设置一个计时器，令其每秒钟被调用一次：

```
>>> var myint = setInterval(toggle, 1000);
```

知道会有什么效果吗？按钮会不停地闪烁（这给单击带来了一定的难度）。当然，如果您玩厌了，只要取消计时器即可。

```
>>> clearInterval(myint)
```

7.4.4 新建节点

通常情况下，我们可以用 `createElement()` 和 `createTextNode()` 这两个方法来创建新节点。而 `appendChild()` 方法则可以用来将新节点添加到 DOM 树结构中。

下面，我们创建一个新的 `p` 元素，并对它的 `innerHTML` 属性进行设置：

```
>>> var myp = document.createElement('p');
>>> myp.innerHTML = 'yet another';
"yet another"
```

一般来说，被新建的元素会自动获得所有的默认属性，例如 `style`，我们可以对它进行修改：

```
>>> myp.style
CSSStyleDeclaration length=0
>>> myp.style.border = '2px dotted blue'
"2px dotted blue"
```

通过 `appendChild()` 方法，我们可以将新节点添加到 DOM 树结构中去。并且，该方法应该是在 `document.body` 上被调用的，这指定了新节点应该被创建在该对象最后一个子节点的后面。

```
>>> document.body.appendChild(myp)
<p style="border: 2px dotted blue;">
```

下面是一张新节点载入页面之后的效果图（见图 7-11）：

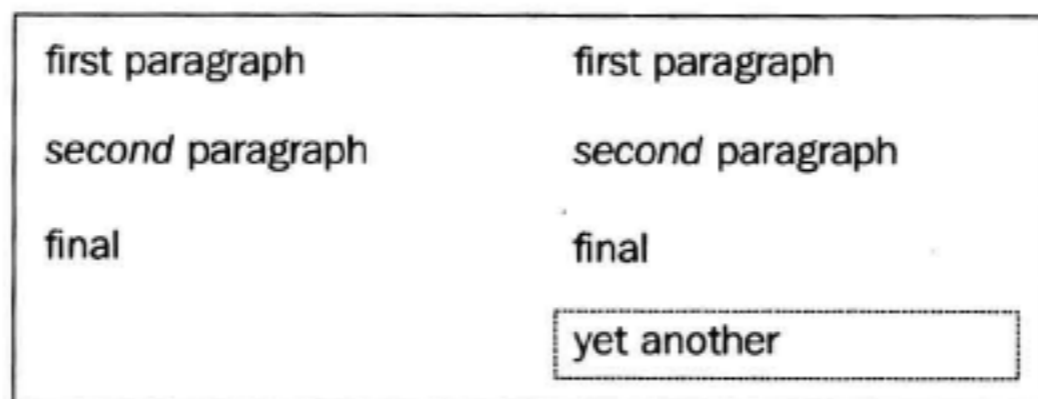


图 7-11

7.4.4.1 纯 DOM 方法

通常情况下，使用 `innerHTML` 来设置内容会更便捷一些，如果要使用纯 DOM 方法，我们就必须：

1. 新建一个内容为“yet another”的文本节点。

2. 再新建一个段落节点。
3. 将文本节点添加为段落节点的子节点。
4. 将段落节点添加为 `body` 的子节点。

通过这种方式，我们可以创建任意数量的文本节点和元素，并随心所欲地安排它们之间的嵌套关系。让我们再来看一个例子，如果您想将下面的 HTML 代码加入 `body` 元素的后端：

```
<p>one more paragraph<strong>bold</strong></p>
```

也就是说，我们所要提交的东西有如下结构：

```
P element
  text node with value "one more paragraph"
  STRONG element
    text node with value "bold"
```

让我们来看看完成这个代码应该怎么写：

```
// create P
var myp = document.createElement('p');
// create text node and append to P
var myt = document.createTextNode('one more paragraph')
myp.appendChild(myt);
// create STRONG and append another text node to it
var str = document.createElement('strong');
str.appendChild(document.createTextNode('bold'));
// append STRONG to P
myp.appendChild(str);
// append P to BODY
document.body.appendChild(myp);
```

7.4.4.2 cloneNode()

另外，拷贝（或克隆）现有节点也是一种创建节点的方法。这需要用到 `cloneNode()` 方法，该方法有一个布尔类型的参数（`true` = 深拷贝，包括所有子节点；`false` = 浅拷贝，只针对当前节点）。下面，让我们来测试一下该方法。

首先，我们获取需要克隆元素的引用：

```
>>> var el = document.getElementsByTagName('p')[1];
```

现在，`e1` 指向了页面中的第二个段落，内容如下：

```
<p><em>second</em> paragraph</p>
```

然后，我们来建立一份 `e1` 的浅拷贝，并将其添加到 `body` 元素的末端：

```
>>> document.body.appendChild(e1.cloneNode(false))
```

这时候，我们在页面上不会看出有什么变化，因为浅拷贝只复制了 `p` 节点，并没有包含它的任何子节点。这意味着该段落中文本（即其中的文本类节点）并没有复制过来。也就是说，这行代码的作用就相对于：

```
>>> document.body.appendChild(document.createElement('p'));
```

但如果我们现在创建的是一份深拷贝，那么以 `P` 元素为首的整个 `DOM` 子树都将会被拷贝过来，其中包含了文本节点和 `EM` 元素。

```
>>> document.body.appendChild(e1.cloneNode(true))
```

如果您愿意的话，也可以只拷贝其中的 `EM` 元素：

```
>>> document.body.appendChild(e1.firstChild.cloneNode(true))
```

```
<em>
```

或者只拷贝内容为“`second`”的文本节点：

```
>>> document.body.appendChild(e1.firstChild.  
                                firstChild.cloneNode(false))
```

```
"second"
```

7.4.4.3 insertBefore()

通过 `appendChild()` 方法，我们只能将新节点添加到指定节点的末端。如果想更精确地控制插入节点的位置，我们还可以使用 `insertBefore()` 方法。该方法与 `appendChild()` 基本相同，只不过它多了一个额外参数，该参数可以用于指定将新节点插入哪一个元素的前面。例如在下面的代码中，文本节点被插入 `body` 元素的末端：

```
>>> document.body.appendChild(document.createTextNode('boo!'));
```

但我们也可以将同样的文本节点添加为 `body` 元素的第一个子节点：

```
document.body.insertBefore(
  document.createTextNode('boo!'),
  document.body.firstChild
);
```

7.4.5 移除节点

要想从 DOM 树中移除一个节点，我们可以调用 `removeChild()`。下面，让我们再次以 `body` 元素为例：

```
<body>
  <p class="opener">first paragraph</p>
  <p><em>second</em> paragraph</p>
  <p id="closer">final</p>
  <!-- and that's about it -->
</body>
```

下面，我们移除第二段落：

```
>>> var myp = document.getElementsByTagName('p')[1];
>>> var removed = document.body.removeChild(myp);
```

如果我们稍后还需要用到被移除的节点的话，可以保存该方法的返回值。尽管该节点已经不在 DOM 树结构中，但我们依然可对其调用所有的 DOM 方法：

```
>>> removed
<p>
>>> removed.firstChild
<em>
```

除此之外，还有一个 `replaceChild()` 方法，该方法可以在移除一个节点的同时将另一个节点放在该位置。下面，我们来看看之前移除节点之后的情况，现在的树结构应该是这样：

```
<body>
  <p class="opener">first paragraph</p>
  <p id="closer">final</p>
  <!-- and that's about it -->
</body>
```

现在，第二段已经变成了 ID 为“closer”的元素：


```
>>> var p = document.getElementsByTagName('p')[1];
>>> p
<p id="closer">
```

下面，我们用 `removed` 变量中的段落替换掉当前段落：

```
>>> var replaced = document.body.replaceChild(removed, p);
```

与 `removeChild()` 相似，`replaceChild()` 方法也会返回被移除节点的引用：

```
>>> replaced
<p id="closer">
```

现在，`body` 元素中的内容如下：

```
<body>
  <p class="opener">first paragraph</p>
  <p><em>second</em> paragraph</p>
  <!-- and that's about it -->
</body>
```

如果我们想将某个子树中的内容一并抹去的话，最便捷的方式是就将它的 `innerHTML` 设置为空字符串。下面我们移除 `body` 中的所有子节点：

```
>>> document.body.innerHTML = '';
""
```

我们来测试一下：

```
>>> document.body.firstChild
null
```

使用 `innerHTML` 来移除确实很容易，但如果我们只使用纯 DOM 方法的话，就必须对其所有的子节点进行遍历并逐个删除它们。下面，我们给出了一个用于删除某个指定节点所有子节点的函数：

```
function removeAll(n) {
  while (n.firstChild) {
    n.removeChild(n.firstChild);
  }
}
```

如果我们想删除 `body` 中的所有子节点，将页面变成一个空 `<body></body>` 的话，可以：

```
>>> removeAll(document.body);
```

7.4.6 只适用于 HTML 的 DOM 对象

正如我们所知，文档对象模型同时适用于 XML 和 HTML 文档。前面，我们已经学习了如何对树结构进行遍历，并添加、删除、修改任何 XML 文档树中的节点。但是，还有一些对象和属性是只适用于 HTML 的。

例如，`document.body` 就是一个纯 HTML 对象。但它的应用是如此的常见，只要 HTML 文档中包含了 `<body>` 标签就可以访问，其功能等价于 `document.getElementsByTagName('body')[0]`，但调用方式则要友好得多。

`document.body` 是一个典型的、根据史前标准 DOM Level 0 和 HTML 特性扩展而来的 DOM 对象。像 `document.body` 这样的对象还有不少，在这些对象中，有些在 Core DOM 组件中是找不到等价物的，而有些则是可以找到的。但普遍都在 DOM 0 标准的基础上做了一定的简化。下面，让我们来了解一下这些对象。

7.4.6.1 访问文档的基本方法

与如今 DOM 组件可以访问页面中的任何元素（甚至包括注释和空白处）不同的是，JavaScript 所能访问的内容只局限于一些 HTML 文档中的元素。其主要有以下一系列集合对象组成：

- ◆ `document.images`——当前页面中所有图片的集合，等价于 Core DOM 组件中的 `document.getElementsByTagName('img')` 调用。
- ◆ `document.applets`——等价于 `document.getElementsByTagName('applets')`。
- ◆ `document.links`
- ◆ `document.anchors`
- ◆ `document.forms`

其中，`document.links` 是一个列表，其中包含了页面中所有的 `` 标签，也就是页面中所有含有 `href` 属性的 A 标签。而 `document.anchors` 中包含的则是所有带 `name` 属性的链接（即 ``）。

而其中使用最广泛的还是要数 `document.forms` 集合了, 这是一个 `<form>` 标签的列表。也就是说, 我们可以这样访问页面中的第一个 `form` 元素:

```
>>> document.forms[0]
```

这就相当于我们调用:

```
>>> document.getElementsByTagName('forms')[0]
```

`forms` 集合中包含一系列的 `input` 字段和按钮, 我们可以通过该对象的 `elements` 属性来访问它们。下面我们访问页面中第一个 `form` 元素中的第一个 `input` 字段:

```
>>> document.forms[0].elements[0]
```

一旦我们获得了某个元素的访问权, 就可以访问该标签对象的 `attributes` 属性。现在假设第一个 `form` 元素的首字段如下:

```
<input name="search" id="search" type="text" size="50"
      maxlength="255" value="Enter email..." />
```

那么, 我们就可以通过某种方法改变该字段中的文本 (即其 `value` 属性的值), 例如:

```
>>> document.forms[0].elements[0].value = 'me@example.org'
"me@example.org"
```

如果想动态设置该字段的显示属性的话, 我们也可以:

```
>>> document.forms[0].elements[0].disabled = true;
```

另外, 如果 `form` 本身或者 `form` 中的元素拥有 `name` 属性的话, 我们也可以通过名字来访问:

```
>>> document.forms[0].elements['search']; // array notation
>>> document.forms[0].elements.search;    // object property
```

7.4.6.2 document.write()

通过 `document.write()` 方法, 我们可以在当前页面载入时插入一些 HTML 元素, 例如, 我们可以:

```
<p>It is now <script>document.write("<em>" + new Date() + "</em>");</script></p>
```

其效果与我们直接在 HTML 文档中插入相关日期相同:

```
<p>It is now <em>Sat Feb 23 2008 17:48:04 GMT-0800  
(Pacific Standard Time)</em></p>
```

另外,我们还有一个 `document.writeln()` 方法,其功能与 `document.write()` 基本相同,只不过该方法会在末尾自动加入换行符“`\n`”,也就是说,下面两行代码是等效的:

```
>>> document.write('boo!\n');  
>>> document.writeln('boo!');
```

需要注意的是,我们只能在页面被载入时调用 `document.write()` 方法,如果我们也试图在页面载入之后调用该方法,整个页面的内容都会被替换掉。

事实上,我们很少需要用到 `document.write()` 方法,如果您觉得需要的话,那就应该先尝试一下其他方法。毕竟就修改页面内容而言,DOM Level 1 所提供的方法要简单灵活得多。

7.4.6.3 Cookies、Title、Referrer、Domain

在这一节中,我们还将为您介绍另外四个属性,这些属性都属于从 DOM Level 0 移植到 DOM Level 1 的 HTML 扩展。并且,这些属性与之前所介绍的属性不一样,它们在 Core DOM 中并没有等价物。

`document.cookie` 属性实际上是一个字符串,其中存储了用于往返服务器端与客户端之间的 `cookie` 信息。每当服务器向浏览器发送页面时,HTTP 头信息中往往都包含了一些用于设置 `cookie` 的内容。以便在客户端再向服务器发送请求时返回带有该头信息的 `cookie`。通过 `document.cookie` 属性,我们可以对浏览器发送服务器的 `cookie` 信息进行某些操作。下面我们来看一个示例,先访问 `cnn.com` 网站,然后在控制台中键入 `document.cookie`:

```
>>> document.cookie  
"CNNid=Ga50a0c6f-14404-1198821758-6; SelectedEdition=www; s_sess=  
%20s_dslv%...
```

`document.title` 属性则是被用来修改页面在浏览器窗口中所显示的标题的。下面依然以 `cnn.com` 网站为例,我们可以这样做:

```
>>> document.title = 'My title'  
"My title"
```

然后，就会看到如下效果（见图 7-12）：

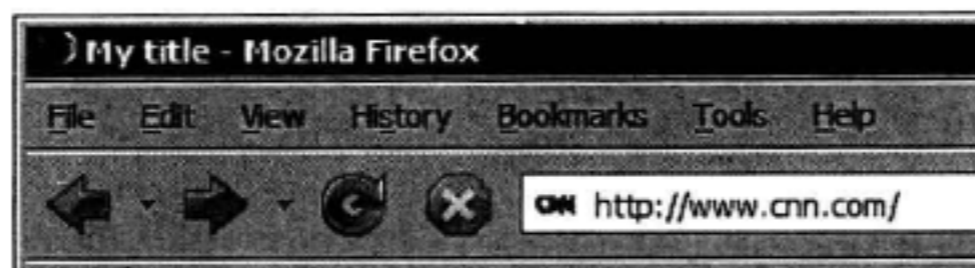


图 7-12

但要注意的是，这里并没有改变<title>标签本身的值，只是改变了其在浏览器窗口中的显示内容。所以，该集合并不等价于 `document.getElementsByTagName('title')[0]`。

`document.referrer` 中记录的是我们之前所访问过的页面 URL。这与浏览器在请求页面时所发送的 HTTP 头信息中的 Referer 值是相同的（要注意的是，HTTP 头信息中的 Referer 在拼写上是错误的，而 `document.referrer` 则是正确的）。例如，如果您是通过 Yahoo!搜索来访问 CNN 主页的，我们就会看到如下信息：

```
>>> document.referrer
"http://search.yahoo.com/search?p=cnn&ei=UTF-8&fr=moz2"
```

通过 `document.domain`，我们可以对当前所载入页面的域名进行访问。这对于某些跨域调用来说是非常有用的。可以想象一下，如果 `yahoo.com` 的主页中有一个 `frame` 或 `iframe` 标签，其中所载入的内容却是来自 `music.yahoo.com` 主机上的。根据一般浏览器的安全规则，通常情况下是不允许该页面与其 `iframe` 中的内容进行交互的。这时候，如果我们想实现这两个页面之间“交谈”，就需要用 `document.domain` 将相关的域全都设置为 `yahoo.com`。

需要注意的是，域的设置只能朝着更非具体化的方向进行。例如，`www.yahoo.com` 的域可以被改为 `yahoo.com`，但 `yahoo.com` 的域就不能再被改为 `www.yahoo.com` 或其他非 `yahoo` 域名了。

```
>>> document.domain
"www.yahoo.com"
>>> document.domain = 'yahoo.com'
"yahoo.com"
>>> document.domain = 'www.yahoo.com'
Illegal document.domain value" code: "1009
>>> document.domain = 'www.example.org'
Illegal document.domain value" code: "1009
```

在本章之前的内容中，我们曾经为您介绍过 `window.location` 对象。那么，实际上

我们也可以使用 `document.location` 来实现相同的功能：

```
>>> window.location === document.location
true
```

7.5 事件

想象一下，如果您突然在收音机里听到有人宣布：“大事件！重大事件！外星人登陆地球了！”或许您的反应是“耶，我无所谓！”，但有些听众可能会觉得“这下世界和平了！”，而另一些则可能会觉得“这下所有人都要死了！”。同样的，浏览器中所发生的事件也能以广播“收听”和监听的形式传递给相关的代码。这些事件包括：

- ◆ 用户单击某一按钮。
- ◆ 用户在某一表单域中输入字符。
- ◆ 某页面载入完成。

我们可以为这些事件指定相应的 JavaScript 函数（它们通常被称为事件监听器或事件处理器），这样一来，浏览器就会在相关事件发生时执行既定的函数。下面，我们来看看具体是如何实现的。

7.5.1 内联 HTML 属性法

最简便的方式就是通过标签的特定属性来添加事件，例如：

```
<div onclick="alert('Ouch!')">click</div>
```

在这种情况下，只要该 `<div>` 所在的区域被用户单击了，就会触发该标签的单击事件。与此同时，其 `onclick` 属性中的字符串就会被当做 JavaScript 代码来执行。尽管，这里并没有显式指定监听单击事件的函数，但相关环境在幕后已经为此创建了一个函数，函数的代码就等于我们为 `onclick` 属性设定的值。

7.5.2 元素属性法

关于单击事件函数，我们还有另一种编写方式，那就是将其设置为 DOM 元素节点的属性。例如：

```
<div id="my-div">click</div>
<script type="text/javascript">
```

```
var myelement = document.getElementById('my-div');
myelement.onclick = function() {
    alert('Ouch!');
    alert('And double ouch!');
}
</script>
```

事实上这也是一种更好的选择。因为这种方式可以帮助我们理清<div>与相关 JavaScript 代码之间的关系。一般情况下。我们总是希望页面中的内容归 HTML、行为归 JavaScript、格式归 CSS，并且三者之间应该能够彼此独立，互不干扰。

但这个方法也是有缺点的，因为这种做法只允许我们指定一个事件函数，这就好像我们的收音机只能有一个听众一样。当然，我们可以对多个事件使用同一个处理函数，但这样做始终不太方便，就好像我们每次都让所有的收音机听众都集中在一个房间里一样。

7.5.3 DOM 的事件监听器

对于浏览器来说，最佳的事件处理方式当然莫过于出自 DOM Level 2 的事件监听器了。通过这种方式，我们可以为一个事件指定多个监听器函数。当事件被触发时，所有的监听器函数都会被执行。而且，这些监听器之间不需要知道彼此的存在，它们的工作是彼此独立的。任何一个函数的加入或退出都不会影响其他监听器的工作。

现在，让我们回到上一节中的那个简单标志页（您也可以直接访问 <http://www.phpied.com/files/jsoop/ch7.html>），我们所拥有的标志是：

```
<p id="closer">final</p>
```

下面我们通过 `addEventListener()` 方法给单击事件赋予相关的监听器，代码如下：

```
>>> var mypara = document.getElementById('closer');
>>> mypara.addEventListener('click', function()
    {alert('Boo!')}, false);
>>> mypara.addEventListener('click', console.log, false);
```

如您所见，`addEventListener()` 方法是基于某一节点对象来调用的。它的首参数是一个事件类型的参数，第二个参数是一个函数指针，它可以是 `function(){alert('Boo!')}` 这样的匿名函数，也可以是 `console.log()` 这样的现存函数。该监听器函数会在相关事件发生时被调用，调用时会接收到一个事件对象参数。如果我们运行上面的代码，并单击页面中的最后一段，就会在 Firebug 控制台中看到该事件对象的日志信息（见图 7-13）：

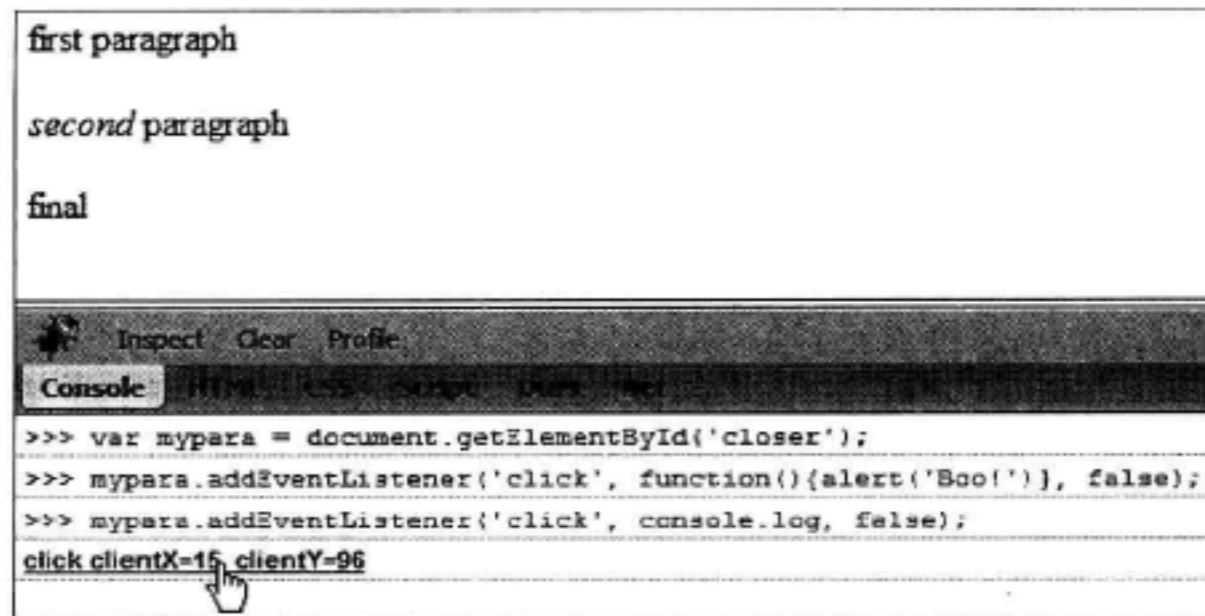


图 7-13

单击其中的某个事件对象，我们可以查看其对象属性（见图 7-14）：

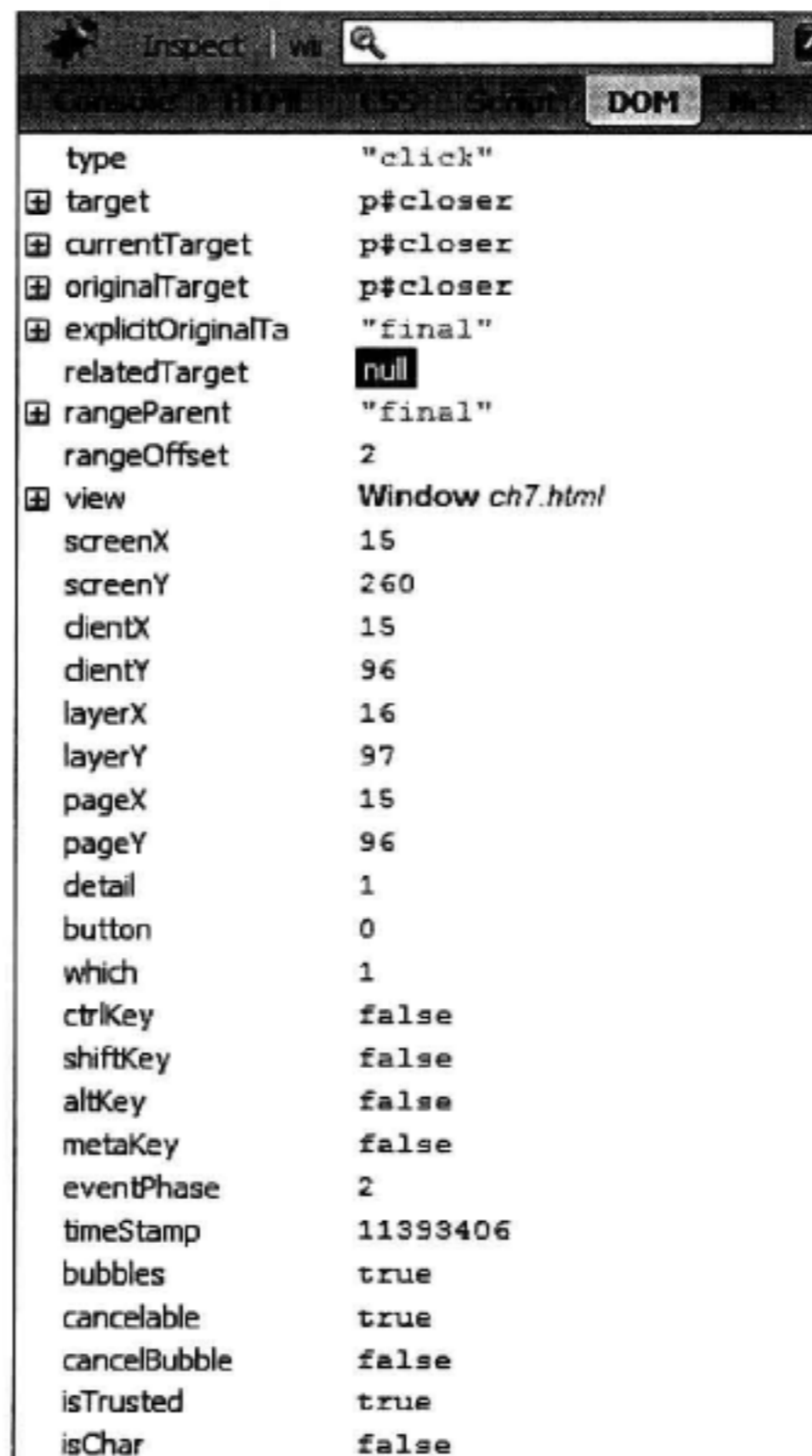


图 7-14

7.5.4 捕捉法与冒泡法

在之前调用 `addEventListener()` 方法的过程中，还存在一个第三参数 `false`。下

面我们来看看这个参数是什么。

假设我们有一个无序的链接列表，例如：

```
<body>
  <ul>
    <li><a href="http://phpied.com">my blog</a></li>
  </ul>
</body>
```

当我们单击该链接时，实际上我们也单击了列表项、列表、<body>乃至整个 document 对象。总而言之，对该链接的单击也可以看做对 document 对象的单击，这就形成了一个事件传播链。这种事件传播过程的实现通常有两种方式：

- ◆ 捕捉法——单击首先发生在 document 上，然后依次传递给 body、列表、列表项，并最终到达该链接。
- ◆ 冒泡法——单击首先发生在链接上，然后逐层向上冒泡，直至 document 对象。

按照 DOM Level 2 的建议，事件传播应该分成三个阶段：先在标签上使用捕捉法，而后使用冒泡法。也就是说，事件传播的路径应该是先从 document 到相关链接（或标签），然后回到 document。如果想要了解某一事件当前所处的阶段，我们可以去访问事件对象的 eventPhase 属性（见图 7-15）。

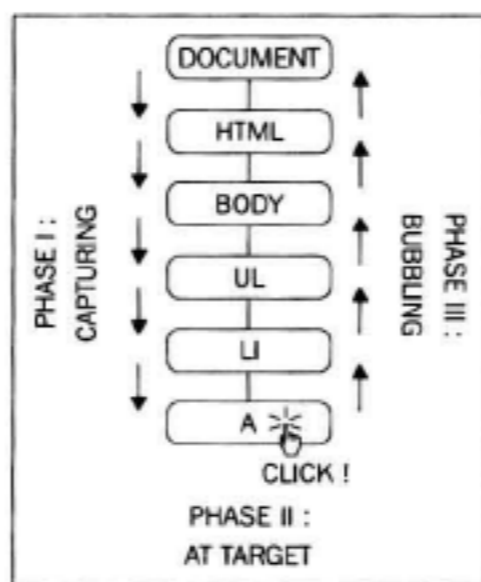


图 7-15

从历史上来说，IE 和 Netscape（当时业界并没有一个统一的标准可以遵循）的相关实现是高度不统一的。IE 使用冒泡法，而 Netscape 则只使用捕捉法。而在 DOM 标准建立之后，Firefox、opera 以及 Safari 都基本上完整地实现了上述的三个阶段，只有 IE 还依然坚持只使用冒泡法。

那么，有哪些因素会在实际上影响到事件的传播呢？

- ◆ 通过 `addEventListener()` 的第三个参数，我们可以决定代码是否采用捕捉法

来处理事件。然而，为了让我们的代码适用于更多的浏览器，最好还是始终将其设置为 `false`，即只使用冒泡法来处理事件。

- ◆ 我们也可以在监听器函数中阻断事件的传播，令其停止向上冒泡，这样一来，事件就不会再到达 `document` 对象那里了。为了做到这一点，我们就必须去调用相关事件对象的 `stopPropagation()` 方法（相关示例我们将会在下节中看到）。
- ◆ 另外，我们还可以采用事件委托的方式来解决这一问题。例如，如果某个 `<div>` 中有 10 个按钮，那么，通常每个按钮都需要一个事件监听器，这样一来，我们就要设置 10 个监听器函数。而更聪明的做法是，我们只为整个 `<div>` 设置一个监听器，当事件发生时，让它自己去判断被单击的是哪一个按钮。

为公平起见，我们还是要介绍一下在 IE 中使用事件捕捉的方式，即使用 `setCapture()` 和 `releaseCapture()` 方法，但是这种方式只适用于处理鼠标类事件，对于其他类型的事件（例如键盘类事件）则不起作用。

7.5.5 阻断传播

下面，我们来演示一下如何让事件停止它的冒泡式传播。首先，我们回到之前的测试文档，现有的标签是：

```
<p id="closer">final</p>
```

然后，我们来定义一个用于处理该段落单击事件的函数：

```
>>> function paraHandler(){alert('clicked paragraph');}
```

现在，我们将该函数设置为单击事件的监听器：

```
>>> var para = document.getElementById('closer');
>>> para.addEventListener('click', paraHandler, false);
```

同时，我们还可以将该单击监听器设置给 `body`、`document`，乃至整个浏览器的 `window` 对象：

```
>>> document.body.addEventListener('click', function(){alert
    ('clicked body')}, false);
>>> document.addEventListener('click', function(){alert
    ('clicked doc')}, false);
>>> window.addEventListener('click', function(){alert
    ('clicked window')}, false);
```

需要注意的是，按照 DOM 标准来说，`window` 事件是不存在的。这就是为什么 DOM 指的是文档而不是浏览器。在 IE 中，单击事件通常是不会被传播到窗口这一级的，但 Firefox 可以。

现在，如果我们单击一下该段落，就会看到四个警告窗，它们分别是：

- ◆ **clicked paragraph**
- ◆ **clicked body**
- ◆ **clicked doc**
- ◆ **clicked window**

这诠释了同一单击事件从具体标签向整个窗口传播的全过程（也就是向上冒泡的全过程）。

`addEventListener()` 方法的对立面就是 `removeEventListener()`，该方法的参数与前者相同。下面，我们移除该段落上的监听器。

```
>>> para.removeEventListener('click', paraHandler, false);
```

现在如果再次单击段落，就只会弹出 `body`、`document` 对象及 `window` 对象的单击事件窗，不再有针对该段落的弹出窗了。

下面，我们来阻断事件的传播。首先要定义一个以事件对象为参数的函数，并在函数内对该对象调用 `stopPropagation()` 方法：

```
function paraHandler(e) {  
    alert('clicked paragraph');  
    e.stopPropagation();  
}
```

然后我们添加修改后的监听器：

```
>>> para.addEventListener('click', paraHandler, false);
```

现在如果我们再单击段落，就会看到弹出窗只有一个了，因为该事件不会再被上传给 `body`、`document` 和 `window` 了。

要提醒的是，由匿名函数所定义的监听器是不能被移除的。因为如果我们要移除某个监听器，就必须获得之前那一个指定为监听器函数的指针。但任何两个匿名函数在内存中都是彼此独立的，即便它们的函数体完全相同也无济于事。

```
document.body.removeEventListener('click',
    function(){
        alert('clicked body')
    },
    false); // does NOT remove the handler
```

7.5.6 防止默认行为

在浏览器模型中，有些事件自身就存在一些预定义行为。例如，单击链接会载入另一个页面。对此，我们可以为该链接设置监听器，并禁用其默认行为。要做到这一点，我们只需要在相关事件对象上调用 `preventDefault()` 方法。

下面，我们来麻烦一下我们的访客，让他们在每次单击链接之后，回答一个问题：“**Are you sure you want to follow this link?**”。每当他们单击的是 Cancel（即 `confirm()` 返回 `false`）时，`preventDefault()` 方法就会被调用：

```
// all links
var all_links = document.getElementsByTagName('a');
for (var i = 0; i < all_links.length; i++) { // loop all links
    all_links[i].addEventListener(
        'click', // event type
        function(e) { // handler
            if (!confirm('Are you sure you want to follow this link?')) {
                e.preventDefault();
            }
        },
        false); // don't use capturing
}
```

需要提醒的是，并不是所有事件的默认行为都是可禁止的。尽管大部分事件是可以的，但如果真的有必要确定一下，我们可以去检测事件对象的 `cancellable` 属性。

7.5.7 跨浏览器事件监听器

正如我们所说过，现在绝大部分的浏览器都已经完全实现了 DOM Level 1 标准。然而，事件方面的标准化是到 DOM Level 2 才完成的。这就导致了一个结果，IE 与 Firefox、Opera 及 Safari 等其他浏览器在这方面的实现有着不少的差异。

让我们再引入一个事件示例，该示例将会在控制台中返回被单击元素（即目标元素）的 `nodeName` 属性值：

```
document.addEventListener('click', function(e) {
  console.log(e.target.nodeName);
}, false);
```

接下来，我们仔细看看 IE 的实现究竟有哪些不同之处：

- ◆ IE 中没有 `addEventListener()` 方法，但它们从 IE5 开始就提供了一个叫做 `attachEvent()` 的等效方法。对于更早期的版本，我们就只能通过属性方法（例如 `onclick` 属性）来解决问题了。
- ◆ 对于单击事件来说，使用 `attachEvent()` 就等同于使用 `onclick` 属性。
- ◆ 如果我们使用老式手法来进行事件监听（例如，通过将某个函数赋值给 `onclick` 属性），那么当该回调函数被调用时，它不会获得相关的事件参数。但只要我们设置了事件监听器，IE 中总会有一个全局对象 `window.evnet` 会指向该事件。
- ◆ 在 IE 的事件对象中，没有用于反映触发事件元素的 `target` 属性，但我们可以使用它的等效属性 `srcElement`。
- ◆ 正如之前所提到的，IE 不支持事件捕捉法，而只使用冒泡法来运作。
- ◆ IE 中没有 `stopPropagation()` 方法，我们可以通过将 `only-IE` 属性 `cancelBubble` 设置为 `true` 来完成相同的操作。
- ◆ IE 中没有 `preventDefault()` 方法，我们可以通过将 `only-IE` 属性 `returnValue` 设置为 `false` 来完成相同的操作。
- ◆ 对于事件的取消监听操作，IE 中使用的不是 `removeEventListener()` 方法，我们要调用的是 `detachEvent()` 方法。

这样一来，我们就将原型的代码修改成跨浏览器版本了：

```
function callback(evt) {
  // prep work
  evt = evt || window.event;
  var target = (typeof evt.target !== 'undefined') ? evt.target :
  evt.srcElement;
  // actual callback work
  console.log(target.nodeName);
}
// start listening for click events
if (document.addEventListener) { // FF
  document.addEventListener('click', callback, false);
} else if (document.attachEvent) { // IE
```

```
    document.attachEvent('onclick', callback);  
  } else {  
    document.onclick = callback;  
  }  
}
```

7.5.8 事件类型

现在，我们已经了解了如何处理跨浏览器事件，但至今为止所有的示例都是关于单击事件的。那么，除此之外还有哪些事件呢？您可能已经猜到了，不同的浏览器支持的事件也是不同的。其中有一部分事件是跨浏览器的，而另一部分则是这些浏览器独有的。关于完整的事件列表，您可能需要去查看相关浏览器的文档。在这里，我们只讨论跨浏览器事件的话题：

- ◆ 鼠标类事件。
- ◆ 鼠标键的松开、按下、单击（按下并松开一次算单击一次）、双击。
- ◆ 鼠标的悬停（指鼠标悬停某元素上方）、移出（指鼠标从某元素上方离开）、拖动。
- ◆ 键盘类事件。
- ◆ 键盘键的按下、输入、松开（这三个事件是按顺序排列的）。
- ◆ 载入/窗口类事件。
- ◆ 载入（图片、页面或其他组件完成载入操作）、卸载（指用户离开当前页面）、卸载之前（由脚本提供的、允许用户终止卸载的选项）。
- ◆ 中止（指用户在 Firefox 中停止载入页面，或者在 IE 中止图片载入）、错误（指在 IE 或 Firefox 中发生了 JavaScript 错误，或 IE 中图片载入失败）。
- ◆ 大小重置（指浏览器窗口大小被重置）、滚动条（指页面进行了滚动操作）、上下文菜单（即右键菜单应用）。
- ◆ 表单类事件。
- ◆ 获得焦点（指某字段获得输入）、失去焦点（指离开该字段）。
- ◆ 改变（指改变某字段的值后离开）、选中（指某文本字段中的文本被选中）。
- ◆ 重置、提交。

到这里，有关事件的内容算讨论完了。请读者参考本章最后的练习题，选一些富有挑战性的题目来实际体验一下这些跨浏览器事件的处理操作。

7.6 XMLHttpRequest 对象

`XMLHttpRequest()` 是一个用构建 HTTP 请求的 JavaScript 对象（即构造器）。从历史上来说，`XMLHttpRequest`（简称 XHR）最初在 IE 浏览器中是以 ActiveX 对象的形式被引入的。但正式实现该对象则是始于 IE7，那时候也只是该浏览器中的一个本地对象，后来逐渐被 Firefox、Safari、Opera 等其他浏览器所接受，并形成了一种通用的跨浏览器实现，这就是所谓的 AJAX 应用。这种应用模式可以使我们无须每次都通过刷新整个页面来获取新内容。我们可以利用 JavaScript 将相关的 HTTP 请求发送给服务器端，然后根据服务器端的响应来局部更新页面。总而言之，通过这种方式构建出来的页面在许多响应方式上会更类似于桌面应用。

实际上，AJAX 就是在 JavaScript 和 XML 之间所建立的一种异步联系。

- ◆ 之所以是异步，是因为我们的代码在发送 HTTP 请求之后，不需要特地停下来等待服务器响应，可以继续执行其他任务，待相关信息到达时自然会收到通知（通常以事件的形式出现）。
- ◆ JavaScript——它的作用很明显，XHR 对象就是用 JavaScript 来创建的。
- ◆ 至于用 XML，则是因为开发者最初设计这种 HTTP 请求就是用来获取 XML 文档，并用其中的数据来更新页面的。尽管如今这种做法已经不太常见了，但是这种方式也可以用来获取纯文本格式的数据，这能使得一些 JSON 格式的数据，或简单的 HTML 元素更容易被插入相关的页面中。

关于 `XMLHttpRequest` 对象的用法，主要可以分为两个有效步骤：

- ◆ 发送请求——在这一步骤中，我们需要完成 `XMLHttpRequest` 对象的构建，并为其设置事件监听器。
- ◆ 处理响应——在这一步骤中，事件监听器会在服务器的响应信息到达时发出通知，然后代码就会忙于从中提取有用的信息。

7.6.1 发送请求

首先，我们来简单地创建一个对象（对于不同的浏览器，可能在细节上会略有些不同）：

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

接下来要做的就是为该对象设置一个能触发 `readystatechange` 事件的事件监听器：

```
xhr.onreadystatechange = myCallback;
```

然后，我们需要去调用其 `open()` 方法，具体如下：

```
xhr.open('GET', 'somefile.txt', true);
```

如您所见，第一个参数指定是 HTTP 请求的类型（包括 GET、POST、HEAD 等）。GET 和 POST 是最常见的类型，当需要发送的数据不是很多时，我们一般会用 GET 类型，否则就会使用 POST 类型。而第二个参数则是我们所请求目标的 URL。在这个示例中，我们所请求的是一个与当前页面处于同一目录的文本文件 `somefile.txt`。最后一个参数是一个布尔类型的值，它决定了请求是否按照异步的方式进行（是就为 `true`，否则就为 `false`）。

当然了，最后是发送请求。

```
xhr.send('');
```

另外只要我们愿意，可以用 `send()` 方法在发送请求时附带任何数据。对于 GET 类请求来说，这里所发送的是一个空字符串。因为该数据将被包含在 URL 中。而对于 POST 请求来说，这就成了被包含在表单数据 `key=value&key2=value2` 中的一个查询字符串。

这样一来，请求被发送出去之后，我们的代码（以及用户）就可以将注意力转向其他任务。待它收到服务器端响应时，会自动启动回调函数 `myCallback`。

7.6.2 处理响应

我们已经为 `readystatechange` 事件设置了监听器，那么这个事件究竟是怎么回事呢？

原来，每个 XHR 对象中都有一个叫做 `readyState` 的属性。一旦我们改变了该属性的值，就会触发 `readystatechange` 事件。该属性可能的状态值如下：

- ◆ 0——未初始化状态
- ◆ 1——载入请求状态
- ◆ 2——载入完成状态
- ◆ 3——请求交互状态
- ◆ 4——请求完成状态

当 `readyState` 的值为 **4** 时，就意味着服务器端的响应信息已经返回，可以开始处理了。在 `myCallback` 函数中，除了确定 `readyState` 的值是 **4** 之外，我们还必须检查一下 HTTP 请求的状态码。因为如果目标 URL 实际上并不存在，我们会收到一个值为 **404** 的状态码（表示未找到文件），正常情况下该值应该为 **200**。因此，`myCallback` 有必要对该值进行检查，该状态码可以通过 XHR 对象的 `status` 属性来获得。

一旦确定了 `xhr.readyState` 的值为 **4** 并且 `xhr.status` 的值为 **200**，我们就可以通过 `xhr.responseText` 来访问目标 URL 中的内容了。下面，我们看看如何在 `myCallback` 中实现用简单的 `alert()` 方法来显示目标 URL 中的内容：

```
function myCallback() {
    if (xhr.readyState < 4) {
        return; // not ready yet
    }
    if (xhr.status !== 200) {
        alert('Error!'); // the HTTP status code is not OK
        return;
    }
    // all is fine, do the work
    alert(xhr.responseText);
}
```

一旦我们获得了所请求的东西，就可以将其添加到页面中，或者用于某些计算以及其他我们所能想到的地方。

总而言之，这两个处理步骤（发送请求、处理响应）是整个 XHR/AJAX 编程方式的核心部分。现在我们已经基本掌握了，可以去构建下一个 Gmail 或 yahoo! Map 了。哦，对了，我们还得介绍一个与众不同的主流浏览器。

7.6.3 在早于 7 的 IE 版本中创建 XMLHttpRequest 对象

在早于版本 7 的 Internet Explorer 浏览器中，XMLHttpRequest 对象是以 ActiveX 对象的形式存在的，因此创建 XHR 实例的方式会有些小小的不同，具体如下：

```
var xhr = new ActiveXObject('MSXML2.XMLHTTP.3.0');
```

其中，`MSXML2.XMLHTTP.3.0` 是我们所要创建对象的标识符。因为实际上，XMLHttpRequest 对象有几个不同的版本，如果访问我们网页的客户没有安装最新的版本，在放弃他们之前，或许您应该试试前两个版本。

对于一个完整的跨浏览器解决方案而言，我们应该首先对用户浏览器所支持的 XMLHttpRequest 对象进行检查，如果该浏览器中没有这个对象，我们就得使用 IE 方案。因此，整个创建 XHR 实例的过程应该像这样：

```
var ids = ['MSXML2.XMLHTTP.3.0',
          'MSXML2.XMLHTTP',
          'Microsoft.XMLHTTP'];
var xhr;
if (typeof window.XMLHttpRequest === 'function') {
    xhr = new XMLHttpRequest();
} else {
    for (var i = 0; i < ids.length; i++) {
        try {
            xhr = new ActiveXObject(ids[i]);
            break;
        } catch (e) {}
    }
}
```

下面来看看这段代码究竟做了哪些事。首先，数组 `ids` 是一个包含了所有可能的 ActiveX 对象的 ID 列表。变量 `xhr` 则是一个将要指向新建 XHR 对象的指针。然后，我们的代码会先测试一下 `windows.XMLHttpRequest` 对象，看看这是否是一个函数。如果是，就意味着当前浏览器支持 `XMLHttpRequest()` 构造器的（也就是说，该浏览器是 Firefox、Safari、Opera、IE7（及其后续版本）中的一个）；如果不是，那么代码就得通过遍历 `ids` 中的可能项来尝试着创建对象。`catch(e)` 则可以捕获其中创建失败的项目并使循环继续。如此，只要有一个 XHR 对象被成功创建，我们就可以提前退出循环。

正如您所知，这段代码的用处会非常多，所以最好还是将其抽象成一个函数。实际上，在本章后面的练习题中就有一题要求我们创建属于我们自己的 AJAX 工具集。

7.6.4 A 代表异步

现在，我们已经了解了如何创建一个 XHR 对象，只需给它一个既定的 URL，然后处理相关的请求响应即可。但如果我们异步发送了两个请求会发生什么呢？或者说，如果第二个请求的响应先于第一个请求返回会发生什么？

在前面的例子中，XHR 对象都是属于全局域的，`myCallback` 要根据这个全局对象的存在状态来访问它的 `readyState` 和 `responseText` 属性。除此之外还有一种方法，可以让我们摆脱对全局对象的依赖，那就是将我们的回调函数封装到一个闭包中去。下面我

们来看看具体如何做：

```
var xhr = new XMLHttpRequest();
xhr.onreadystatechange = (function(myxhr){
    return function(){myCallback(myxhr);}
})(xhr);
xhr.open('GET', 'somefile.txt', true);
xhr.send('');
```

在这种情况下，myCallback 将会以参数的形式接收相关的 XHR 对象，这就避免使用全局空间的问题。同时，这也意味着当该请求再次获得响应信息时，原来的 xhr 变量就可以被第二次请求重置甚至销毁了。因为我们在闭包内保留了该对象的原有信息。

7.6.5 X 代表 XML

尽管作为数据传输格式来说，最近 JSON（我们会在第 8 章中介绍）在风头上已经盖过了 XML，但 XML 仍然是我们的一个选择。除了 responseText 属性外，XHR 对象还有另一个名为 responseXML 的属性。如果我们向一个 XML 文档发送一个 HTTP 请求，该属性就会指向该 XML 的 DOM document 对象。因此，对于该文档的操作，我们可以对它调用之前所讨论的方法，例如 getElementByTagName()、getElementById() 等。

7.6.6 实例示范

下面，让我们通过一个具体的实例来总结一下关于 XHR 对象的各种话题。您也可以在 <http://www.phpied.com/files/jsoop/xhr.html> 中找到相关页面，并测试该示例中的操作。

该主页 xhr.html 是一个非常简单的静态页面，其中只含有三个 <div> 元素：

```
<div id="text">Text will be here</div>
<div id="html">HTML will be here</div>
<div id="xml">XML will be here</div>
```

然后，我们在 Firebug 控制台中输入相关代码，向三个文件发送请求，并将它们各自的内容载入相关的 <div> 中。

这三个文件所载入的分别是：

- ◆ content.txt——一段简单的文本，内容为 “I am a text file”。

◆ content.html——一段 HTML 代码，具体如下：

```
"I am <strong>formatted</strong> <em>HTML</em>"
```

◆ content.xml——一个 XML 文档，内容如下：

```
<?xml version="1.0" ?>
<root>
  I'm XML data.
</root>
```

要注意的是，上面所提到的所有文件都应该与 xhr.html 存在同一个目录中。因为出于安全因素，我们只能对相同域中的文件使用 XMLHttpRequest 请求。

我们先来完成请求/响应部分的抽象，创建函数如下：

```
function request(url, callback) {
  var xhr = new XMLHttpRequest();
  xhr.onreadystatechange = (function(myxhr){
    return function(){
      callback(myxhr);
    }
  })(xhr);
  xhr.open('GET', url, true);
  xhr.send('');
}
```

该函数接受两个参数，一个是我们所请求的 URL，另一个则是响应返回后所要调用的回调函数。接下来，我们要调用三次该函数，每个文件一次，具体如下：

```
request(
  'http://www.phpied.com/files/jsoop/content.txt',
  function(o) {
    document.getElementById('text').innerHTML = o.responseText;
  }
);
request(
  'http://www.phpied.com/files/jsoop/content.html',
  function(o) {
    document.getElementById('html').innerHTML = o.responseText;
  }
);
request(
  'http://www.phpied.com/files/jsoop/content.xml',
  function(o) {
```

```

document.getElementById('xml').innerHTML =
    o.responseXML.getElementsByTagName('root')[0]
        .firstChild.nodeValue;
}
);

```

在这里，回调函数都是以内联的方式来定义的。前两个函数的实现很类似，它们都只需要用其所请求文件中的内容替换掉相关<div>中的 HTML 文本即可。第三个函数则略有不同，因为它涉及一个 XML 文档。首先，我们需要通过 `o.responseXML` 调用来访问该 XML 文档的 DOM 对象。然后再调用 `getElementsByTagName()` 获取页面中所有 <root> 标签的列表（实际上只有一项），<root> 标签的 `firstChild` 是一个文本节点，所以我们用其 `nodeValue` 属性来获取这段文本（即“I’m XML data”），并用它替换掉 `id="xml">` 中的 HTML 内容。整体效果如图 7-16 所示：

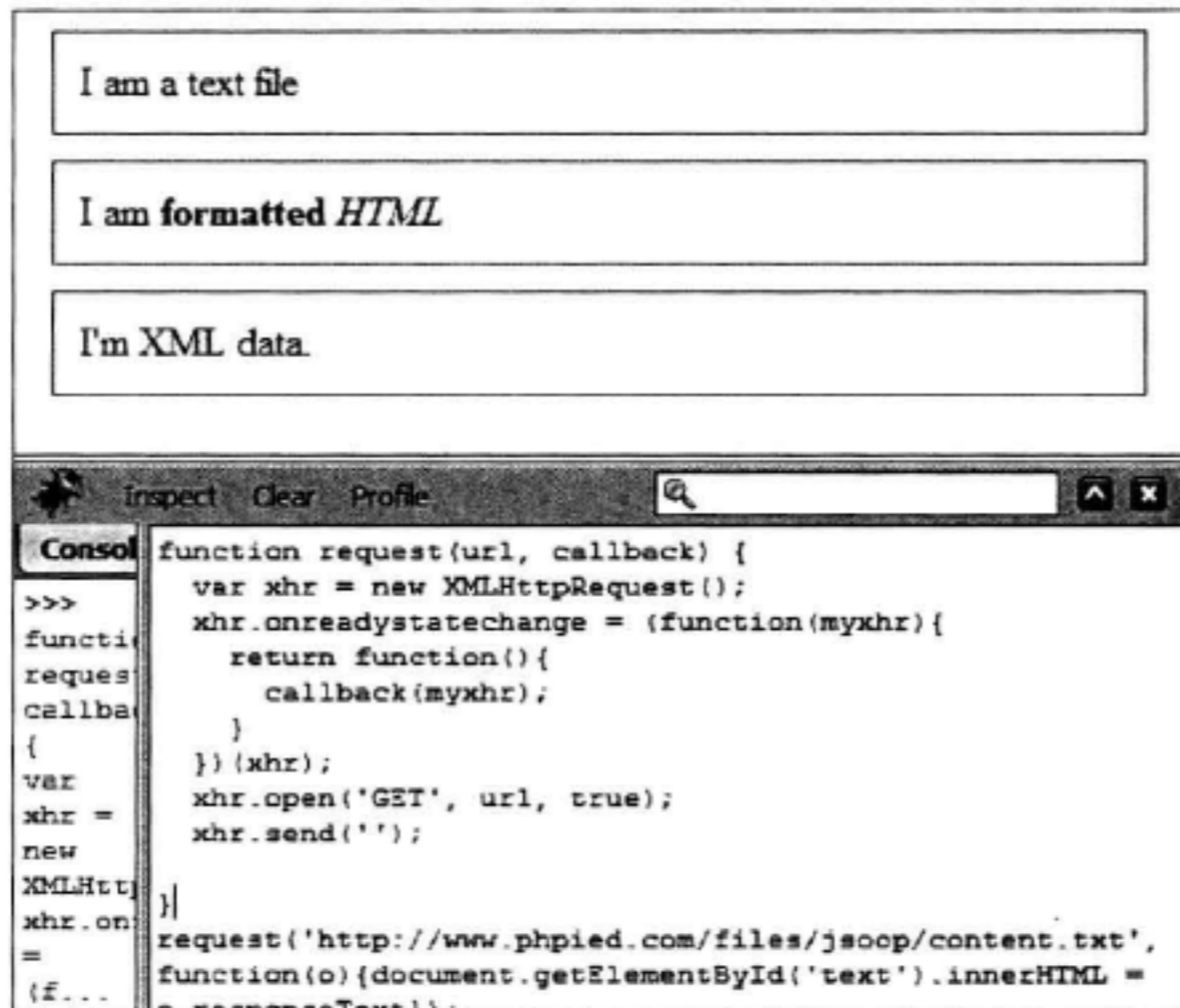


图 7-16

对于 XML 文档上的操作，我们也可以通过调用 `o.responseXML.documentElement` 来获取 <root> 元素，以取代 `o.responseXML.getElementsByTagName('root')[0]`。记住，`documentElement` 所指向的就是一个 XML 文档的根节点。特别对于 HTML 文档来说，它的根节点始终都是 <html> 标签。

7.7 本章小结

本章所涉及的内容相当多。首先，我们介绍了一系列跨浏览器的 BOM（浏览器对象模

型) 对象, 其中主要包括:

- ◆ 全局对象 `window` 的系列属性, 例如 `navigator`、`location`、`history`、`frames`、`screen` 等。
- ◆ 及其方法, 例如 `setInterval()` 和 `setTimeout()`; `alert()`、`confirm()` 和 `prompt()`; `moveTo/By()` 和 `resizeTo/By()`。

然后, 我们介绍了有关 DOM (文件对象模型) 的内容, 这是一个以树型结构来表示 HTML (或 XML) 文档的方法, 其中的每一个标签或文本都是该树结构上的节点。我们详细介绍了:

- ◆ 节点访问:
 - ◆ 通过 `parentNode`、`childNodes`、`firstChild`、`lastChild`、`nextSibling`、`previousSibling` 带有父/子关联性的属性来访问。
 - ◆ 通过 `getElementById()`、`getElementsByTagName()`、`getElementsByName()` 等方法来访问。
- ◆ 节点修改:
 - ◆ 通过 `innerHTML` 或 `innerText/textContent` 属性来进行。
 - ◆ 通过 `nodeValue` 或 `setAttribute()` 以及对象属性中的相关属性来进行。
 - ◆ 通过 `removeChild()` 或 `replaceChild()` 来移除节点。
 - ◆ 以及通过 `appendChild()`、`cloneNode()`、`insertBefore()` 等方法来添加新节点。

另外, 我们还介绍了一些从 DOM 0 (这是正式标准化之前的产物) 中移植到 DOM Level 1 中的属性, 其中包括:

- ◆ 一系列集合对象。例如 `document` 对象的 `forms`、`images`、`links`、`anchors` 以及 `applets`。但相对来说, DOM 1 中的 `getElementsByTagName()` 显然更为灵活实用。
- ◆ `document.body`, 这是一种能方便访问 `<body>` 元素的特定属性。
- ◆ 另外, 我们还介绍了 `document` 中的 `title`、`cookie`、`referrer`、`domain` 四大特殊属性。

接着, 我们为您介绍介绍了浏览器事件的传播方式。尽管它们要实现跨浏览器模式并不容

易，但也是完全有可能的。由于事件是以冒泡形式传播的，因此，我们可以将监听任务设置得更全局化。另外，我们还介绍了如何阻断事件的传播路径，以及如何改变其默认行为。

最后，我们还学习了有关 XMLHttpRequest 对象的知识，该对象也允许我们构建一个具有即时相应能力的 Web 页面，主要分为两个步骤：

- ◆ 首先，向服务器发送 HTTP 请求，以获得相关数据。
- ◆ 然后，处理服务器的响应信息，并更新页面中的相关部分。

7.8 练习题

在本章之前，我们的练习题都是可以在各自章节的正文中找到解决方案的。但这一次，您会发现有些练习题需要我们对本书以外的内容有更多的了解（或实践经验）。

1. BOM

作为 BOM 的练习来说，我们可以试着写出许多错误的、富有骚扰性的、对用户非常不友好的代码，以及所有非常 Web 1.0 的东西。例如晃动的浏览器窗口，就是将窗口上下左右不停地移动，以形成一种类似地震的感觉。这需要通过某种 `move*()` 函数来完成，其中需要多次调用 `setInterval()`，最后还需要通过 `setTimeout()` 方法来令其停止操作；或者我们也可以令浏览器弹出一个 200×200 的窗口，然后将其大小渐变成 400×400 ；或者我们可以更简单一些，将日期/时间实时显示在浏览器的标题栏中（通过 `window.status` 属性），并像钟表一样每秒钟更新一次。然而，需要提醒您的是，上面这些操作在默认情况下通常是被禁止的，因此需要更改一下浏览器的默认选项，毕竟人们已经受够了这些令人心烦的“效果”（在 Firefox 中，我们可以在菜单 **Tools | Options | Content | Enable JavaScript | Advanced** 中找到这些选项）。

2. DOM

2.1 换一种不同的方式来实现 `walkDOM()` 方法，以回调函数参数的形式来代替用 `console.log()` 硬编码的方式。

2.2 使用 `innerHTML` 来移除相关内容确实很方便（即 `document.body.innerHTML = ""`），但未必总是最好的选择。如果在其中有元素被设置了事件监听器，那么当该元素被移除时，IE 并不会解除该元素与监听器之间的关联。这就有可能会造成浏览器中内存泄漏，因为它们所引用的内容已经不存在了。因此，我们在实现一个通用的移除 DOM 节点函数时，首先要移除相关的事件监听器。我们可以遍历目标节点的属性，检查这些属性值是否

属于函数类型，如果是（例如最常见的 `onclick` 属性），我们就必须在该元素节点被删除之前将该属性设置为 `null`。

2.3 创建一个叫做 `include()` 的函数，该函数可以将外部脚本引入当前页面，这样一来，我们就等于创建了一个可以动态设置 `src` 属性的 `<script>` 标签。该函数应通过如下测试：

```
>>> include('somescript.js');
```

2.4 下面我们将用在 2.3 题中实现的函数来调用来自 Yahoo! 的 JavaScript 搜索服务。该服务文档位于 `http://developer.yahoo.com/search/web/V1/webSearch.html`。当我们构建相关 URL 请求时，需要做一些设置：`output=json`、`callback=console.log`。这样一来，该服务器调用的返回结果（实质上是一个 JavaScript 对象）就会在控制台中被打印出来。当然了，我们也可以选择实现一个更有用的函数来代替 `console.log`。

3. 事件

3.1 创建一个叫做 `myevent` 的跨浏览器事件工具集（也叫对象集），其中应该包含以下方法：

- ◆ `addListener(element, event_name, callback)` —— 其中的 `element` 参数也可以是一个元素数组。
- ◆ `removeListener(element, event_name, callback)`。
- ◆ `getEvent(event)` —— 对于 IE 的早期版本，我们可以通过检查 `window.event` 属性来实现。
- ◆ `getTarget(event)`。
- ◆ `stopPropagation(event)`。
- ◆ `preventDefault(event)`。

其用例如下：

```
function myCallback(e) {  
    e = myevent.getEvent(e);  
    alert(myevent.getTarget(e).href);  
    myevent.stopPropagation(e);  
    myevent.preventDefault(e);  
}  
myevent.addListener(document.links, 'click', myCallback);
```


执行这段示例代码应该会使该文档中所有的链接失效，只不过，它们在被单击时会弹出一个 `alert()` 窗口，以显示其 `href` 属性。

3.2 创建一个以像素定位的 `<div>` 元素，坐标为 `x=100px`，`y=100px`。然后编写代码使 `<div>` 元素能按照以下按键 J（左）、K（右）、M（下）、I（上）的操作方式在页面中移动。并且，在编写过程中可以重用您在 3.1 题中实现的事件工具集。

4. XMLHttpRequest 对象

4.1 创建一个叫做 `ajax` 的 XHR 工具集（也叫对象集），其用法如下：

```
function myCallback(xhr) {
    alert(xhr.responseText);
}
ajax.request('somefile.txt', 'get', myCallback);
ajax.request('script.php', 'post', myCallback,
    'first=John&last=Smith');
```

4.2 用 Firebug 构建一个 AJAX 化的 Google 搜索组件。我们可以通过 JavaScript 将相关组件“插”到当前页面的某一部分中。以便我们可以通过该页面向 `google.com` 发送 XHR 请求，这样一来，我们就可以在不载入任何第二页面的情况下进行搜索或访问 `google.com`，并且能在不刷新当前页面的情况下载入其搜索结果。我们鼓励您在实现过程中重用您自己的事件工具集（来自 3.1 题）和 AJAX 工具集（来自 4.1 题）。其具体步骤应该如下：

- ◆ 为相关表单的 `submit` 事件设置事件监听器，并屏蔽该事件的默认行为，使该表单不能正常提交。
- ◆ 新建一个 XHR 对象，然后请求以下 URL: `http://www.google.com/search?q = myquery` 中的页面。其中 `myquery` 所代表的是我们在搜索字段中输入的内容。
- ◆ 在 XHR 对象的回调函数中，我们在 `<body>` 后面添加了一个 `id="content"` 的 `<div>` 元素，并将其 `innerHTML` 设置为该 XHR 对象的 `responseText` 属性。

在这种方法中，我们只需要在搜索字段中输入要搜索的内容并按回车，就可以在不载入其他页面的情况下获得搜索结果。而且，这些代码应该是可重用的，如果您想要分多次显示不同的内容，只需要更新我们所创建的 `<div>` 容器中的 HTML 即可。

第 8 章

编程模式与设计模式

到目前为止，我们已经掌握了 JavaScript 的面向对象特性，如原型和继承，并且接触了一些使用浏览器对象的实例。接下来，我们将介绍一些 JavaScript 中的常用模式及其使用方法。首先，我们要了解什么是模式，简单地说，模式就是专门为某些常见问题开发的、优秀的解决方案。

通常，当我们面对一个新的编程问题时，往往会立刻发现眼前的这个问题与我们之前解决过的某个问题有很多相似之处。这时候，您或许就可以考虑将这些问题抽象归类，以寻求一个通用性的解决方案。而所谓模式，实际上就是一系列经过实践证明的、针对某类问题的、具有可重用性的解决方案（或者是寻求解决方案的方法）。有时候，模式仅仅是一个用于帮助我们思考的想法或名字。例如，当您与团队中其他开发人员讨论某类问题或方案时，模式可以被当做一个术语来使用，以使交流变得更容易一些。

而有时候我们所面对的问题可能要更特殊一些，以至于可能根本找不到任何适用的模式。这时候，切忌盲目使用模式，生搬硬套在任何时候都不是一个好主意。因为在这种情况下，往往不使用模式要比为了套用某个现有模式而去强行改变问题本身要好得多。

在本章，我们将讨论的模式主要分为两大类：

- ◆ 编程模式——一些专门为 JavaScript 语言开发出的最佳实践方案。
- ◆ 设计模式——这些模式与具体语言无关，它们主要来自“GoF”的《设计模式》^①一书。

^①《设计模式：可复用面向对象软件的基础》（*Design Patterns: Elements of Reusable Object-Oriented Software*）是软件工程领域有关软件设计的一本书，提出和总结了对于一些常见软件设计问题的标准解决方案，称为软件设计模式。该书作者为：Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides，后以“四人帮”（Gang of Four, GoF）著称。——译者注

8.1 编程模式

在本章的第一部分中，我们首先要讨论一些与 JavaScript 语言特性密切相关的模式。其中有些模式主要用来组织代码（如命名空间模式），有些则与性能改善有关（如延迟定义和初始化时分支），还有些会涉及一些 JavaScript 语言缺失的特性（比如私有属性）。总而言之，本节将讨论以下几种模式：

- ◆ 行为隔离
- ◆ 命名空间
- ◆ 初始化分支
- ◆ 延迟定义
- ◆ 配置对象
- ◆ 私有变量和方法
- ◆ 特权方法
- ◆ 私有函数的共有化
- ◆ 自执行的方法
- ◆ 链式调用
- ◆ JSON

8.1.1 行为隔离

正如我们所知，一个网页通常有三个要素：

- ◆ 内容（HTML）
- ◆ 外观（CSS）
- ◆ 行为（JavaScript）

8.1.1.1 内容

HTML 所代表的是网页的内容，也就是文字。它们由一小部分 HTML 标签来负责标记。这些标签描述了与内容相关的语义，例如和标签可用于导航菜单，因为后者只

是一组链接而已。

通常情况下，内容（HTML）中是不应该包含格式化元素的。可视化格式之类的元素应该属于外观层的东西，通常交由 CSS 来实现，这意味着我们应该：

- ◆ 尽量避免在 HTML 标签中使用样式类的属性。
- ◆ 不要使用与外观有关的 HTML 标签，例如。
- ◆ 尽量根据语义需要来选择标签，而不是去考虑浏览器会如何绘制它们。例如，开发人员有时候对<div>标签的使用实际上不如<p>标签来得更合适。同理，我们应该更多地使用和而不是和<i>，因为后者更强调的是外观而不是语义。

8.1.1.2 外观

要将外观与内容分开，有一种好方法就是对浏览器默认的绘制行为进行重置，例如 YUI 库中的 reset.css。这样一来，浏览器默认的绘制方式就不会影响我们对语义标签的选择了。

8.1.1.3 行为

网页中的第三要素是行为。行为也应该做到与内容及外观分离。行为通常是由 JavaScript 负责定义的，即由<script>标签来标记。这些脚本代码通常会被存放在单独的文件中。这意味着我们使用的不是类似于 onclick, onmouseover 这样的内嵌属性，而是利用前几章中曾经介绍过的 addEventListener/attachEvent 方法来进行事件定义。

关于行为与内容的隔离，我们通常有以下几条原则性策略：

- ◆ 尽可能少用<script>标签。
- ◆ 尽量不要使用内嵌事件的处理方法。
- ◆ 尽量不要使用 CSS 表达式。
- ◆ 当 JavaScript 被用户禁用时，我们要动态地添加一些表示无目标的替换标记。
- ◆ 在内容末尾、<body>标签之前，插入一个 external.js 文件。

8.1.1.4 行为隔离实例

下面，假设我们有一个搜索表单，该表单中的内容需要通过 JavaScript 来验证。在这里，我们没有在 form 标签内使用任何 JavaScript 代码，而只是在<body>标签结束之前插入一个<script>标签，并令其指向一个外部脚本文件。

```
<body>
  <form id="myform" method="post" action="server.php">
    <fieldset>
      <legend>Search</legend>
      <input
        name="search"
        id="search"
        type="text"
      />
      <input type="submit" />
    </fieldset>
  </form>
  <script type="text/javascript" src="behaviors.js"></script>
</body>
```

而在 behaviors.js 文件中，我们为 submit 事件设定了一个处理方法，用以检查输入文本框是否为空。若为空，则不提交表格。以下是 behaviors.js 的完整实现，在其中，我们用到了第 7 章练习题中所实现的 myevent 工具：

```
// init
myevent.addListener('myform', 'submit', function(e){
  // no need to propagate further
  e = myevent.getEvent(e);
  myevent.stopPropagation(e);
  // validate
  var el = document.getElementById('search');
  if (!el.value) { // too bad, field is empty
    myevent.preventDefault(e); // prevent the form submission
    alert('Please enter a search string');
  }
});
```

8.1.2 命名空间

为了减少命名冲突，我们通常都会尽量减少使用全局变量的机会。但这并不能根本解决问题，更好的办法是将变量和方法定义在不同的命名空间中。这种方法的实质就是只定义一个全局变量，并将其他变量和方法定义为该变量的属性。

8.1.2.1 将对象用做命名空间

首先，我们来新建一个全局变量 MYAPP：

```
// global namespace
var MYAPP = MYAPP || {};
```

然后，我们将上面用到的 `myevent` 全局对象设定为 `MYAPP` 的属性：

```
// sub-object
MYAPP.event = {};
```

接着再为其添加方法：

```
// object together with the method declarations
MYAPP.event = {
  addListener: function(el, type, fn) {
    // .. do the thing
  },
  removeListener: function(el, type, fn) {
    // ...
  },
  getEvent: function(e) {
    // ...
  }
  // ... other methods or properties
};
```

8.1.2.2 命名空间中的构造器应用

我们也可以在命名空间中使用构造器函数。在本例中，`DOM` 工具本身就定义了一个 `Element` 构造器，通过它我们可以很方便地创建 `DOM` 元素。

```
MYAPP.dom = {};
MYAPP.dom.Element = function(type, prop) {
  var tmp = document.createElement(type);
  for (var i in prop) {
    tmp.setAttribute(i, prop[i]);
  }
  return tmp;
}
```

同样的，您也可以使用 `Text` 构造器来创建文本类节点：

```
MYAPP.dom.Text = function(txt) {
  return document.createTextNode(txt);
}
```

然后使用该构造器在网页底部创建一个链接：

```
var e11 = new MYAPP.dom.Element(
    'a',
    {href:'http://phpied.com'}
);
var e12 = new MYAPP.dom.Text('click me');
e11.appendChild(e12);
document.body.appendChild(e11);
```

8.1.2.3 namespace()方法

如今，很多库中（比如 YUI）都实现了一个叫做 namespace 的工具方法，其调用方法如下：

```
MYAPP.namespace('dom.style');
```

这等价于：

```
MYAPP.dom = {};
MYAPP.dom.style = {};
```

下面，我们来看看这个 namespace() 方法是如何实现的。首先，我们创建一个数组，用于存放由“.”分割的输入字符串；然后将该数组中的每个元素都添加为全局对象的属性。

```
var MYAPP = {};
MYAPP.namespace = function(name) {
    var parts = name.split('.');
    var current = MYAPP;
    for (var i in parts) {
        if (!current[parts[i]]) {
            current[parts[i]] = {};
        }
        current = current[parts[i]];
    }
}
```

测试新的方法：

```
MYAPP.namespace('event');
MYAPP.namespace('dom.style');
```

上述代码等价于以下调用：

```
var MYAPP = {
  event: {},
  dom: {
    style: {}
  }
}
```

8.1.3 初始化分支

我们在第 7 章曾经提到过，不同的浏览器对于相同或相似的方法可能有不同的实现。这时，您需要依据当前的浏览器版本来选择对应的执行分支。这类分支可能有很多，因而可能会减缓脚本执行速度。

但非要等到运行时才能分支吗？我们完全可以在加载脚本时，在模块初始化的过程中就将部分代码进行分支处理。这显然更有利于提高效率。利用 JavaScript 代码可以动态定义的特性，我们可以为不同的浏览器定制不同的实现方法。下面我们来看一个具体示例。

首先，我们定义了一个命名空间并声明了一些事件处理方法。

```
var MYAPP = {};
MYAPP.event = {
  addListener: null,
  removeListener: null
};
```

注意，此时无论是添加还是删除事件监听的方法都还没有被定义，它们将根据具体的浏览器特性探测的结果，被赋予不同的实现。

```
if (typeof window.addEventListener === 'function') {
  MYAPP.event.addListener = function(el, type, fn) {
    el.addEventListener(type, fn, false);
  };
  MYAPP.event.removeListener = function(el, type, fn) {
    el.removeEventListener(type, fn, false);
  };
} else if (typeof document.attachEvent === 'function'){ // IE
  MYAPP.event.addListener = function(el, type, fn) {
    el.attachEvent('on' + type, fn);
  };
  MYAPP.event.removeListener = function(el, type, fn) {
```



```
        el.detachEvent('on' + type, fn);
    };
} else { // older browsers
    MYAPP.event.addListener = function(el, type, fn) {
        el['on' + type] = fn;
    };
    MYAPP.event.removeListener = function(el, type, fn) {
        el['on' + type] = null;
    };
};
```

一旦上述脚本被执行，我们就定义了与浏览器特性相关的 `addListener()` 和 `removeListener()` 方法。如此，当它们再次被调用时，就不需要再探测浏览器特性了，脚本会执行得更快。

要提醒的是，在检查浏览器特性时，请尽量不要对一个特性做过多的假设。在上例中，我们就没有遵从这一原则。因为我们只检查了浏览器对 `add*` 方法的支持，然后就直接定义了相应的 `add*` 和 `remove*` 方法。如果下一版本的 IE 同时实现了 `addEventListener()` 和 `removeEventListener()` 方法，这么做是正确的。但请想象一下，如果 IE 只实现了 `stopPropagation()` 却没有实现 `preventDefault()` 方法，而我们又没有对它们分别检测，会导致什么后果？显然，我们会想当然地认为 `preventDefault()` 会被实现，而我们很可能又会想当然地认为 `addEventListener()` 方法没有被定义，结果又导致我们必须为 IE 浏览器编写专用的处理函数。请记住，这些代码可能会在目前 IE 中正常工作，但不等于今后的版本也一样。为了避免自定义函数在新版本浏览器可能会增加的功能，我们应该单独检测每个可能会用到的浏览器特性，千万不要只做一些泛泛的假设。

8.1.4 延迟定义

延迟定义模式与之前的初始化分支模式很相似。不同之处在于，该模式下的分支只有在相关函数第一次被调用时才会发生——即只有函数被调用时，它才会以最佳实现改写自己。在初始化分支模式中，模块初始化时 `if` 分支必然会发生，而在延迟定义模式中，这可能根本就不会发生——因为某些函数可能永远不会被调用。同时，延迟定义模式也会使得初始化过程更为轻量，因为不需要再做分支检测。

接下来，我们通过一个 `addListener()` 方法定义来演示一下这个模式。该方法将以泛型的方式来实现——即在它第一次被调用时，首先会检查浏览器支持的功能，然后为自己选择最合适的实现，最后调用自身以完成真正的事件添加。当下一次再调用该方法时，

就会直接调用它选择的新方法而不再需要做分支判断。

```
var MYAPP = {};
MYAPP.myevent = {
  addListener: function(el, type, fn){
    if (typeof el.addEventListener === 'function') {
      MYAPP.myevent.addListener = function(el, type, fn) {
        el.addEventListener(type, fn, false);
      };
    } else if (typeof el.attachEvent === 'function'){
      MYAPP.myevent.addListener = function(el, type, fn) {
        el.attachEvent('on' + type, fn);
      };
    } else {
      MYAPP.myevent.addListener = function(el, type, fn) {
        el['on' + type] = fn;
      };
    }
    MYAPP.myevent.addListener(el, type, fn);
  }
};
```

8.1.5 配置对象

该模式往往适用于有很多个参数的函数或方法。但关于“很多”的理解，每个人可能都不一样，但一般来说，当一个函数的参数多于三个时，使用起来就多少会有些不太方便，因为我们不太容易记住这些参数的顺序，尤其是当其中还有默认参数存在的时候。

但我们可以用对象来代替多个参数。也就是说，让这些参数都成为某一个对象的属性。这在面对一些配置型参数时会显得尤为适合，因为它们中往往存在多个缺省参数。简而言之，用单个对象来替代多个参数有以下几点优势：

- ◆ 不用考虑参数的顺序。
- ◆ 可以跳过某些参数的设置。
- ◆ 函数的扩展性更强，可以适应将来的扩展需要。
- ◆ 代码的可读性更好，因为在代码中我们看到的是配置对象的属性名称。

下面，假设我们要设计一个 `Button` 构造器，将其用于创建输入性按钮。它的参数包括一段文本，即按钮的显示内容（`<input>` 标签的 `value` 属性），和一个可缺省的 `type` 参数。

```
// a constructor that creates buttons
var MYAPP = {};
MYAPP.dom = {};
MYAPP.dom.Button = function(text, type) {
    var b = document.createElement('input');
    b.type = type || 'submit';
    b.value = text;
    return b;
}
```

该构造器很简单，只需要传递给它一个字符串，然后就可以把新创建的按钮加入文档：

```
document.body.appendChild(new MYAPP.dom.Button('puuush'));
```

到目前位置一切看起来都很好，但接下来，我们需要为按钮设置更多属性，例如颜色和字体。结果，这个构造器的定义最终就可能会变成这样：

```
MYAPP.dom.Button = function(text, type, color, border, font) {
    // ....
}
```

这显然就不太方便了，比如当我们可能只想设置第三个和第五个参数，而跳过第二个和第四个时，就必须这样：

```
new MYAPP.dom.Button('puuush', null, 'white', null,
    'Arial, Verdana, sans-serif');
```

这时候，更好的选择就是用一个可配置对象参数来代替所有的参数配置，这样一来，函数定义看起来就可能是这样：

```
MYAPP.dom.Button = function(text, conf) {
    var type = conf.type || 'submit';
    var font = conf.font || 'Verdana';
    // ...
}
```

其使用方法如下：

```
var config = {
    font: 'Arial, Verdana, sans-serif',
    color: 'white'
};
new MYAPP.dom.Button('puuush', config);
```

另一个例子：

```
document.body.appendChild(  
    new MYAPP.dom.Button('dude', {color: 'red'})  
);
```

如您所见，我们可以方便地设置部分参数，并可以随意改变参数设置的顺序。同时，由于我们是通过名字来设置参数的，因此代码也显得更易读，更友好。

8.1.6 私有属性和方法

在 JavaScript 中，我们没有可以用于设置对象属性访问权限的修饰符。但一般语言通常有以下访问修饰符：

- ◆ **Public**——对象的属性(或方法)可以被所有人访问。
- ◆ **Private**——只有对象自己可以访问这些属性。
- ◆ **Protected**——仅该对象或其继承者才能访问这些属性。

尽管 JavaScript 中没有特殊的语法来标记私有属性，但是根据第 3 章内容，我们可以在构造器中通过使用局部变量和函数的方式来实现类似的权限控制。

下面继续以 **Button** 构造器为例，我们为它定义了一个 **style** 局部变量，用于表示所有的默认样式参数，并且还定义了一个 **setStyle()** 的局部方法。该变量和方法对于构造器之外的代码来说是不可见的。下面，我们将为您演示 **Button** 对象是如何使用这些局部的私有属性的：

```
var MYAPP = {};  
MYAPP.dom = {};  
MYAPP.dom.Button = function(text, conf) {  
    var styles = {  
        font: 'Verdana',  
        border: '1px solid black',  
        color: 'black',  
        background: 'grey'  
    };  
    function setStyles() {  
        for (var i in styles) {  
            b.style[i] = conf[i] || styles[i];  
        }  
    }  
}
```

```
    conf = conf || {};  
    var b = document.createElement('input');  
    b.type = conf['type'] || 'submit';  
    b.value = text;  
    setStyles();  
    return b;  
};
```

在这段代码中，`styles` 是一个私有属性，而 `setStyle()` 则是一个私有方法。构造器可以在内部调用它们（它们也可以访问构造器中的任何对象），但它们却不能被外部代码所调用。

8.1.7 特权函数

特权函数（这个概念是由 Douglas Crockford 提出的）实际上只是一些普通的公共函数，但它们却可以访问对象的私有方法或属性。其作用就像一座桥梁，将私有特性以一种可控的方式暴露给外部使用者。

例如，我们可以为上例添加一个 `getDefaults()` 方法，返回 `styles` 对象。如此一来，`Button` 构造器之外的代码就能访问到这些样式，但不能修改它们。这时 `getDefaults()` 就是一个特权函数。

8.1.8 私有函数的公有化

假设我们定义了一个函数，但并不想让外界修改它，于是将其设为私有。但有时候我们又希望让某些外部代码能访问到它，这该如何实现呢？解决方案是将这个私有函数赋值给一个公有属性。

下面，我们将 `_setStyle()` 和 `_getStyle()` 定义为私有方法，但同时又将它们分别赋值给公有方法 `setStyle()` 和 `getStyle()`：

```
var MYAPP = {};  
MYAPP.dom = (function() {  
    var _setStyle = function(el, prop, value) {  
        console.log('setStyle');  
    };  
    var _getStyle = function(el, prop) {  
        console.log('getStyle');  
    };  
    return {
```

```

    setStyle: _setStyle,
    getStyle: _getStyle,
    yetAnother: _setStyle
  };
})();

```

在这种情况下，当 `MYAPP.dom.setStyle()` 被调用时，`_setStyle()` 也会被调用。也可以在外面改写 `setStyle()` 方法，

```
MYAPP.dom.setStyle = function(){alert('b')};
```

也就是说：

- ◆ `MYAPP.dom.setStyle` 指向的是新的方法。
- ◆ `MYAPP.dom.yetAnother` 仍然指向 `_setStyle()`。
- ◆ `_setStyle` 方法随时可以被内部的代码调用。

8.1.9 自执行函数

另一个保证全局命名空间不被污染的模式是，把代码封装在一个匿名函数中并立刻自行调用。如此一来，该函数中的所有变量都是局部的（假设我们使用了 `var` 关键字），并在函数返回时被销毁（前提是它们不属于闭包）。本书第3章已经详细讨论过该模式。

```

(function(){
  // code goes here...
})();

```

该模式特别适用于某些脚本加载时所执行的一次性初始化任务。

自执行方法也可用于创建和返回对象。如果我们创建对象的过程很复杂，并且需要做一些初始化工作，那么我们就可以把第一部分相关的初始化工作设置为一个自执行函数，然后通过它来返回一个对象——它可以访问初始化部分定义的任何私有属性。

```

var MYAPP = {};
MYAPP.dom = function(){
  // initialization code...
  function _private(){
    // ... body
  }
  return {

```

```
    getStyle: function(el, prop) {
        console.log('getStyle');
        _private();
    },
    setStyle: function(el, prop, value) {
        console.log('setStyle');
    }
};
}();
```

8.1.10 链式调用

通过链式调用模式，我们可以在单行代码中一次性调用多个方法，就好像它们被链接在了一起。当我们需要连续调用若干个彼此相关的方法时，会带来很大的方便。实际上，我们就是通过前一个方法的结果（即返回对象）来调用下一个方法的，因此不需要中间变量。

通常情况下，任何一个新建的构造器都能立即作用到某个 DOM 元素上去，例如在接下来的代码中，我们用构造器新建了一个 `` 元素，然后将其添加到 `<body>` 元素中：

```
var obj = new MYAPP.dom.Element('span');
obj.setText('hello');
obj.setStyle('color', 'red');
obj.setStyle('font', 'Verdana');
document.body.appendChild(obj);
```

众所周知，构造器返回的是新建对象的 `this` 指针。同样的，我们也可以让 `setText()` 和 `setStyle()` 方法返回 `this`，这样，我们就可以直接用这些方法所返回的实例来调用其他方法，这就是所谓的链式调用：

```
var obj = new MYAPP.dom.Element('span');
obj.setText('hello')
    .setStyle('color', 'red')
    .setStyle('font', 'Verdana');
document.body.appendChild(obj);
```

实际上，我们甚至不需要定义 `obj` 变量，如果在新对象被添加到 DOM 树之后就不再需要访问它的话。那么我们可以这样写：

```
document.body.appendChild(
    new MYAPP.dom.Element('span')
        .setText('hello')
```

```

        .setStyle('color', 'red')
        .setStyle('font', 'Verdana')
    );

```

jQuery 中使用了大量的链式调用模式——这也可能是该库颇为流行的一个原因。

8.1.11 JSON

在编程模式这一节末尾，我们来简单介绍一下 JSON。从本质上说，JSON 本身不能算编程模式，但可以说，JSON 的使用确实是一种很有用的模式。

JSON 本身实际上是一种轻量级的数据交换格式。当使用 XMLHttpRequest() 接收服务器端的数据时，通常使用的就是 JSON 而不是 XML。JSON 是 JavaScript Object Notation 的缩写，它除了使用极为方便之外，没有什么特别的。JSON 格式由对象和数组标记的数据构成。下面是 JSON 字符串的一个例子，来自服务器响应的 XHR 请求：

```

{
  'name': 'Stoyan',
  'family': 'Stefanov',
  'books': ['phpBB2', 'phpBB UG', 'PEAR']
}

```

与其相对应的 XML 应该如下：

```

<?xml version="1.1" encoding="iso-8859-1"?>
<response>
  <name>Stoyan</name>
  <family>Stefanov</family>
  <books>
    <book>phpBB2</book>
    <book>phpBB UG</book>
    <book>PEAR</book>
  </books>
</response>

```

首先，我们可以看到 JSON 是多么轻量，它只用了很少的字节来表示数据。但使用 JSON 的最大好处是，JavaScript 可以很容易地处理它。假设我们发送了一个 XHR 请求并得到了一个 JSON 字符串，它保存在 XHR 的 responseText 属性中，然后，我们调用 eval() 将该字符串转换为 JavaScript 对象：

```

var obj = eval( '(' + xhr.responseText + ')' );

```


接着，我们就可以通过 `obj` 的属性来访问这些数据了：

```
alert(obj.name); // Stoyan
alert(obj.books[2]); // PEAR
```

由于 `eval()` 有安全隐患问题，所以最好使用一些 JavaScript 库(<http://json.org>)来处理 JSON 数据，这样做也很方便：

```
var obj = JSON.parse(xhr.responseText);
```

正因为 JSON 简洁的特点，它很快成为一种流行的、与语言无关的数据交换格式。我们可以很容易地在服务器端使用喜欢的语言创建 JSON 对象，例如，可以用 PHP 提供 `json_encode()` 方法将数组序列化为 JSON 字符串，再用 `json_decode()` 方法还原数组。

8.2 设计模式

在本章第二部分中，我们将为您介绍如何使用 JavaScript 来演绎《设计模式：可复用面向对象软件的基础》一书中介绍的部分设计模式。该书很有影响力，通常也被称为 **Book of Four** 或者 **Gang of Four** 或者 **GoF**（代表该书的四位作者）。这本书中所涉及的模式大致上可以分为三组：

- ◆ 创建型模式：涉及对象的创建与初始化。
- ◆ 结构型模式：描述了如何组合对象以提供新的功能。
- ◆ 行为型模式：描述了对象之间如何通信。

GoF 一共介绍了 23 个模式，自此书发行以来，人们又发现了更多的模式。在这本书中，我们只介绍其中的四个，并通过一些具体 JavaScript 的实例加以说明。请记住，提到模式，我们更关注的是它们接口之间的关系，而不是内部的实现细节。一旦您掌握了一种设计模式，实现起来很容易，尤其对于 JavaScript 这样的动态语言来说。

下面是我们将要介绍的模式：

- ◆ 单件模式
- ◆ 工厂模式
- ◆ 装饰器模式
- ◆ 观察者模式

8.2.1 单件模式 1

单件是一个创建型的设计模式，它主要考虑的是创建对象的方式。当我们需要创建一种类型或一个类的唯一对象时，就可以使用该模式。在一般的语言中，这意味着这一个类只能被创建一个实例对象，如果之后再尝试创建该对象的话，代码就只会返回原来的实例。

但由于 JavaScript 本身没有类的概念，所以单件成为了默认行为，也是最自然的模式。每个对象都是一个单件。如果我们不对它进行拷贝，也不把它作为另一个对象的原型，那么它的类型将不会改变。

JavaScript 中最基本的单件模式实现是使用对象文本标识法：

```
var single = {};
```

8.2.2 单件模式 2

但当我们想用类似于类的语法来实现单件模式时，事情就会变得更有趣一些。例如，假设我们有一个叫做 `Logger()` 的构造器，而我们想这么使用它：

```
var my_log = new Logger();
my_log.log('some event');
// ... 1000 lines of code later ...
var other_log = new Logger();
other_log.log('some new event');
alert(other_log === my_log); // true
```

这段代码所要表达的意思是，尽管这里多次使用了 `new`，但实际上所创建的对象实例却始终只有一个，后续的构造器调用过程中所返回的始终是同一个对象。这是怎么做到的呢？

8.2.2.1 全局变量

解决方案之一是用全局变量来保存这个唯一的实例。在这种情况下，我们的构造器看起来像这样：

```
function Logger() {
  if (typeof global_log === "undefined") {
    global_log = this;
  }
  return global_log;
}
```

使用这个构造器将达到预期的结果：

```
var a = new Logger();
var b = new Logger();
alert(a === b); // true
```

这样做的缺陷是使用了全局变量，它在任何时候都有可能被覆盖掉，从而导致这个实例被丢失。反之亦然，全局变量也随时有可能覆盖别的对象。

8.2.2.2 构造器属性

正如我们所知道的，函数也是一种对象，本身也有属性。因此，我们也可以将这个唯一实例设置为构造器本身的属性。

```
function Logger() {
  if (typeof Logger.single_instance === "undefined") {
    Logger.single_instance = this;
  }
  return Logger.single_instance;
}
```

在这种情况下，当我们调用 `var a = new Logger()` 语句时，`a` 就会指向一个新建的 `Logger.single_instance` 属性。但接下来如果我们再调用 `var b = new Logger()` 语句得到的 `b` 将会指向同一个 `Logger.single_instance` 属性。这正是我们想要的结果。

上述方法很显然解决了全局变量所带来的问题，因为没有全局变量被创建。它的唯一缺陷是 `Logger` 构造器的属性是公有的，因此它随时有可能会被覆盖，如此一来，这个唯一实例可能会被丢失或被修改。

8.2.2.3 使用私有属性

上述问题的解决方案是使用私有属性。我们已经知道如何使用闭包来保护一个变量，作为一个练习，请用此方法实现单件。

8.2.3 工厂模式

工厂模式也属于来创建对象的创建型模式。当我们有多个相似的对象而又不知道应该先使用哪种时，就可以考虑使用工厂模式。在该模式下，代码将会根据具体的输入或其他既定规则，自行决定创建哪种类型的对象。

下面，假设我们三个不同的构造器，它们所实现的功能是相似的。它们所创建的对象都将接受一个 URL 类型的参数，但处理细节稍有不同。例如，它们分别创建的是一个文本 DOM 节点、一个链接以及一个图像。

```
var MYAPP = {};  
MYAPP.dom = {};  
MYAPP.dom.Text = function() {  
  this.insert = function(when) {  
    var txt = document.createTextNode(this.url);  
    when.appendChild(txt);  
  };  
};  
MYAPP.dom.Link = function() {  
  this.insert = function(when) {  
    var link = document.createElement('a');  
    link.href = this.url;  
    link.appendChild(document.createTextNode(this.url));  
    when.appendChild(link);  
  };  
};  
MYAPP.dom.Image = function() {  
  this.insert = function(when) {  
    var im = document.createElement('img');  
    im.src = this.url;  
    when.appendChild(im);  
  };  
};
```

使用三个构造器的方法都一样：设置 url 属性并调用 insert() 方法。

```
var o = new MYAPP.dom.Image();  
o.url = 'http://images.packtpub.com/images/PacktLogoSmall.png';  
o.insert(document.body);  
var o = new MYAPP.dom.Text();  
o.url = 'http://images.packtpub.com/images/PacktLogoSmall.png';  
o.insert(document.body);  
var o = new MYAPP.dom.Link();  
o.url = 'http://images.packtpub.com/images/PacktLogoSmall.png';  
o.insert(document.body);
```

但我们并不知道应该先创建哪一种对象，例如，程序需要根据用户所按的按钮来决定对象的创建。假设 type 中包含了被创建对象的类型，我们可以用 if 或者 switch 语句

写出如下代码：

```
var o;
if (type === 'Image') {
  o = new MYAPP.dom.Image();
}
if (type === 'Link') {
  o = new MYAPP.dom.Link();
}
if (type === 'Text') {
  o = new MYAPP.dom.Text();
}
o.url = 'http://...';
o.insert();
```

这段代码可以工作，但如果构造器很多，代码就会很长。尤其当我们在写一个库或框架时，就有可能根本不知道构造器函数的名字。这时候，就应该考虑将这种动态创建对象的操作委托给一个工厂函数。

下面，让我们来给 MYAPP.dom 工具添加一个工厂方法：

```
MYAPP.dom.factory = function(type) {
  return new MYAPP.dom[type];
}
```

然后我们就可以把上面的三个 if 替换掉了：

```
var o = MYAPP.dom.factory(type);
o.url = 'http://...';
o.insert();
```

在这个例子中，factory() 方法是很简单的，但在实际使用中，我们可能需要对该函数的 type 参数值进行相关的验证，并且对所有的对象做一些相同的设置工作。

8.2.4 装饰器模式

装饰器模式是一种结构型模式，它与对象的创建无关，主要考虑的是如何拓展对象的功能。也就是说，除了使用线性式（父—子—孙）继承方式之外，我们也可以为一个基础对象创建若干个装饰对象以拓展其功能。然后，由我们的程序自行选择不同的装饰器，并按不同的顺序使用它们。在不同的程序中我们可能会面临不同的需求，并从同样的装饰器

集合中选择不同的子集。在下面的代码中，我们为您演示了装饰器模式的一种使用方法：

```
var obj = {
  function: doSomething() {
    console.log('sure, asap');
  },
  // ...
};
obj = obj.getDecorator('deco1');
obj = obj.getDecorator('deco13');
obj = obj.getDecorator('deco5');
obj.doSomething();
```

这个例子的开头使用了一个拥有 `doSomething()` 方法的简单对象，接着，我们通过名字来选择不同的装饰器。这里的每一个装饰器都有一个 `doSomething()` 方法——它会先调用前一个装饰器的 `doSomething()` 方法，然后再执行自己的特有代码。每次添加一个装饰器时，我们都会覆盖基础 `obj`。最后，选择完所有装饰器后，调用 `doSomething()` 方法时，它会顺序调用每个装饰器的 `doSomething()` 方法。下面，我们再来看一个具体的实例。

8.2.4.1 装饰一棵圣诞树

下面来看一个装饰器模式的实例：装饰一棵圣诞树。首先我们来实现 `decorate()` 方法。

```
var tree = {};
tree.decorate = function() {
  alert('Make sure the tree won\'t fall');
};
```

接着，再定义 `getDecorator()` 方法，该方法用于添加额外的装饰器。装饰器被实现为构造器函数，都继承自 `tree` 对象。

```
tree.getDecorator = function(deco) {
  tree[deco].prototype = this;
  return new tree[deco];
};
```

下面来创建第一个装饰器 `RedBalls()`，我们将它设为 `tree` 的一个属性（以保持全局命名空间的纯净）。`RedBall` 对象也提供了 `decorate()` 方法，注意它先调用了父类的 `decorate()` 方法。

```
tree.RedBalls = function() {
  this.decorate = function() {
    this.RedBalls.prototype.decorate();
    alert('Put on some red balls');
  }
};
```

然后，我们用同样的方法来分别添加 `BlueBalls()` 和 `Angel()` 装饰器：

```
tree.BlueBalls = function() {
  this.decorate = function() {
    this.BlueBalls.prototype.decorate();
    alert('Add blue balls');
  }
};
tree.Angel = function() {
  this.decorate = function() {
    this.Angel.prototype.decorate();
    alert('An angel on the top');
  }
};
```

再把所有的装饰器都添加到基础对象中：

```
tree = tree.getDecorator('BlueBalls');
tree = tree.getDecorator('Angel');
tree = tree.getDecorator('RedBalls');
```

最后，运行 `decorate()` 方法：

```
tree.decorate();
```

最终，当我们执行 `decorate()` 方法时，将依次得到如下警告信息：

- ◆ 确保树不会倒。
- ◆ 添加蓝色的球。
- ◆ 在树顶放一个小天使。
- ◆ 再放一些红色的球。

由此可见，我们可以创建很多装饰器，然后按照需求选择和组合它们。

8.2.5 观察者模式

观察者模式（有时也称为订阅者/发行商模式）是一种行为型模式，主要用于处理不同对象之间的交互通信问题。观察者模式中通常会包含两类对象：

- ◆ 一个或多个发行商对象：当有重要的事情发生时，会通知订阅者。
- ◆ 一个或多个订阅者对象：它们追随一个或多个发行商，监听它们的通知，并作出相应的反应。

看上去，观察者模式似乎与第7章中所讨论浏览器事件很相似。确实如此，浏览器事件正是该模式的一个典型应用。浏览器是发行商：当一个事件（如 `onclick`）发生时，它会发出通知。事件订阅者会监听这类事件，并在事件发生时被通知。浏览器（发行商）为每个订阅者发送一个事件对象，但在我们自己的实现中，大可不必使用事件对象，反之，可以使用任何合适的数据类型。

通常来说，观察者模式可分为两类：推送和拉动。推送模式中是由发行商负责将消息通知给各个订阅者。而拉动模式则要求订阅者主动跟踪发行商的状态变化。

下面，我们来看一个推送模式的实例。我们把与观察者相关的代码放到一个单独的对象中，然后以该对象为一个混合类，将它的功能加到发行商对象中。如此一来，任何一个对象都可以成为发行商，而任何一个功能型对象都可以成为订阅者。观察者对象中应该有如下属性和方法：

- ◆ 由回调函数构成的订阅者数组。
- ◆ 用于添加和删除订阅者的 `addSubscriber()` 和 `removeSubscriber()` 方法。
- ◆ `publish()` 方法，授受并传递数据给订阅者。
- ◆ `make()` 方法，将任意对象转变为一个发行商并为其添加上述方法。

以下是一个观察者对象的实现代码，其中包含了订阅相关的方法，并可以将任意对象转变为发行商。

```
var observer = {
  addSubscriber: function(callback) {
    this.subscribers[this.subscribers.length] = callback;
  },
  removeSubscriber: function(callback) {
    for (var i = 0; i < this.subscribers.length; i++) {
```



```
        if (this.subscribers[i] === callback) {
            delete(this.subscribers[i]);
        }
    },
    publish: function(what) {
        for (var i = 0; i < this.subscribers.length; i++) {
            if (typeof this.subscribers[i] === 'function') {
                this.subscribers[i](what);
            }
        }
    },
    make: function(o) { // turns an object into a publisher
        for (var i in this) {
            o[i] = this[i];
            o.subscribers = [];
        }
    }
};
```

接下来，我们来创建一些订阅者。订阅者可以是任意对象，它们的唯一职责是在某些重要事件发生时调用 `publish()` 方法。下面是一个 `blogger` 对象，每当新博客准备好时，就会调用 `publish()` 方法。

```
var blogger = {
    writeBlogPost: function() {
        var content = 'Today is ' + new Date();
        this.publish(content);
    }
};
```

另有一个 `LA Times` 对象，每当新一期的报刊出来时，就会调用 `publish()` 方法。

```
var la_times = {
    newIssue: function() {
        var paper = 'Martians have landed on Earth!';
        this.publish(paper);
    }
};
```

它们都很容易转变为发行商：

```
observer.make(blogger);
observer.make(la_times);
```

与此同时，准备两个简单对象 jack 和 jill：

```
var jack = {
  read: function(what) {
    console.log('I just read that ' + what)
  }
};
var jill = {
  gossip: function(what) {
    console.log('You didn\'t hear it from me, but ' + what)
  }
};
```

他们可以订阅 blogger 对象，只需要提供事件发生时的回调函数。

```
blogger.addSubscriber(jack.read);
blogger.addSubscriber(jill.gossip);
```

当 blogger 写了新的博客时会发生什么事呢？结果就是 jack 和 jill 会收到通知：

```
>>> blogger.writeBlogPost();
I just read that Today is Sun Apr 06 2008 00:43:54 GMT-0700
(Pacific Daylight Time)
You didn't hear it from me, but Today is Sun Apr 06 2008 00:43:54 GMT-0700
(Pacific Daylight Time)
```

任何时候 jill 都可以取消订阅。于是当博主写了另一篇博客时，jill 就不会再收到通知消息。

```
>>> blogger.removeSubscriber(jill.gossip);
>>> blogger.writeBlogPost();
I just read that Today is Sun Apr 06 2008 00:44:37 GMT-0700
(Pacific Daylight Time)
```

jill 也可以订阅 LA Times，因为一个订阅者可以对应多个发行商。

```
>>> la_times.addSubscriber(jill.gossip);
```

如此，当 LA Times 发行新的期刊后，jill 就会收到通知并执行 jill.gossip() 方法。

```
>>> la_times.newIssue();
```

要是再有人不明白，大概是火星人真要登陆地球了吧！

8.3 本章小结

在最后一章中，我们学习了一些 JavaScript 语言中通用的编程模式，了解了如何使程序简洁、干净，运行得更快，以便能更好地与其他程序或库工作。然后我们讨论了如何实现 Book of Four 一书中介绍的部分设计模式。这些内容充分证明了，JavaScript 作为一门功能全面的语言，可以很容易地实现这些经典模式。设计模式是一个很大的主题，读者可以通过 JSPatterns.com 网站与本书著者进一步讨论 JavaScript 中的模式。

现在，您已经拥有了足够丰富的知识，可以运用面向对象的编程概念创建可扩展、可重用、高质量的 JavaScript 程序与库了。祝您好运！

附录 A

保留字

在这篇附录中，我们列出了两个保留字列表。第一个是当前所用的保留字列表，第二个则是为将来预备的保留字列表。

总而言之，这些单词是不能被用做变量名的。

```
var break = 1; // syntax error
```

如果我们需要在对象属性中使用这些词，就必须将其用引号括起来。

```
var o = {break: 1}; // OK in Firefox, error in IE
var o = {'break': 1}; // OK
alert(o.break); // error in IE
alert(o['break']); // OK
```

现役保留字

- ◆ break
- ◆ case
- ◆ catch
- ◆ continue
- ◆ default
- ◆ delete

- ◆ do
- ◆ else
- ◆ finally
- ◆ for
- ◆ function
- ◆ if
- ◆ in
- ◆ instanceof
- ◆ new
- ◆ return
- ◆ switch
- ◆ this
- ◆ throw
- ◆ try
- ◆ typeof
- ◆ var
- ◆ void
- ◆ while
- ◆ with

预备保留字

- ◆ abstract
- ◆ boolean
- ◆ byte
- ◆ char
- ◆ class

-
- ◆ const
 - ◆ debugger
 - ◆ double
 - ◆ enum
 - ◆ export
 - ◆ extends
 - ◆ final
 - ◆ float
 - ◆ goto
 - ◆ implements
 - ◆ import
 - ◆ int
 - ◆ interface
 - ◆ long
 - ◆ native
 - ◆ package
 - ◆ private
 - ◆ protected
 - ◆ public
 - ◆ short
 - ◆ static
 - ◆ super
 - ◆ synchronized
 - ◆ throws
 - ◆ transient
 - ◆ volatile

附录 B

内建函数

在这篇附录中，我们列出了在第 3 章中所讨论过的所有内建函数（即全局对象的方法列表如表 B-1 所示）。

表 B-1

函数名	相关说明
<code>parseInt()</code>	<p>该函数有两个参数：一个输入对象及一个进制基数 <code>radix</code>。它主要用于将输入转换成整数值并返回，如果转换失败就返回 <code>NaN</code>。另外，函数会忽略输入中所包含的指数信息。<code>radix</code> 的默认值为 10（即十进制），但由于忽略该参数可能会导致一些不可预测的结果（例如当您输入 08 这样的数值时），所以最好还是始终明确指定它的值。</p> <pre>>>> parseInt('10e+3') 10 >>> parseInt('FF') NaN >>> parseInt('FF', 16) 255</pre>
<code>parseFloat()</code>	<p>该函数会试图将其接受的参数转换成浮点数值并返回。它也不能处理输入中的指数信息。</p> <pre>>>> parseFloat('10e+3')</pre>

(续表)

函数名	相关说明
parseFloat()	<p>10000</p> <pre>>>> parseFloat('123.456test')</pre> <p>123.456</p>
isNaN()	<p>该函数名是“Is Not a Number”的缩写，它主要用于判断其参数是否是一个有效数字，如果是就返回 true，否则就返回 false。另外，该函数总是会先尝试将输入值转换成数字。</p> <pre>>>> isNaN(NaN)</pre> <p>true</p> <pre>>>> isNaN(123)</pre> <p>false</p> <pre>>>> isNaN(parseInt('FF'))</pre> <p>true</p> <pre>>>> isNaN(parseInt('FF', 16))</pre> <p>False</p>
isFinite()	<p>在该函数中，如果我们的输入是一个数字（或者可以转换为数字），但又不属于 Infinity 或 -Infinity，就返回 true，否则就返回 false。</p> <pre>>>> isFinite(1e+1000)</pre> <p>false</p> <pre>>>> isFinite(-Infinity)</pre> <p>false</p> <pre>>>> isFinite("123")</pre> <p>True</p>
encodeURIComponent()	<p>该函数会将输入转换为符合 URL 编码的字符串，关于这种 URL 编码的详细信息，读者可以参考 wikipedia 中的相关文章： http:// en.wikipedia.org/wiki/Url_encode</p> <pre>>>> encodeURIComponent ('http://phpied.com/') "http%3A%2F%2Fphpied.com%2F"</pre> <pre>>>> encodeURIComponent ('some script?key=v@lue')</pre> <p>"some%20script%3Fkey%3Dv%40lue"</p>

(续表)

函数名	相关说明
decodeURIComponent ()	<p>该函数主要用于解码其所接受的 URL 编码字符串。</p> <pre>>>> decodeURIComponent ('%20%40')</pre> <p>" @"</p>
encodeURIComponent ()	<p>该函数主要用于将输入转换为 URL 编码，但它始终假定其所接受的是一个完整的 URL，因此它所编码的部分不包括目标 URL 的协议（例如 http://）和域名（例如 www.phpied.com）</p>
encodeURIComponent ()	<pre>>>> encodeURIComponent ('http://phpied.com/') "http://phpied.com/" >>> encodeURIComponent ('some script?key=v@lue') "some%20script?key=v@lue"</pre>
decodeURI ()	<p>该函数主要用于执行 encodeURIComponent () 的反操作。</p> <pre>>>> decodeURI ("some%20script?key=v@lue") "some script?key=v@lue"</pre>
eval ()	<p>该函数会执行其接收到的 JavaScript 代码串，并返回代码串中最后一个表达式的执行结果。</p> <p>可能的话，请尽量避免使用该函数。</p> <pre>>>> eval ('1+2') 3 >>> eval ('parseInt ("123")') 123 >>> eval ('new Array (1,2,3)') [1, 2, 3] >>> eval ('new Array (1,2,3); 1+1;') 2</pre>

附录 C

内建对象

在这篇附录中,我们列出了 ECMAScript 标准中所描述的所有内建构造函数,以及用这些构造函数所创建对象的方法与属性。

Object

`Object()` 是用于创建 `Object` 对象的构造器,例如:

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

当然,我们也可以使用对象标识法来实现同样的效果:

```
>>> var o = {}; // recommended
```

该构造器可以接受任何类型的参数,并且它会自动识别参数的类型,并选择更合适的构造器来完成相关操作。例如,如果我们传递给 `new Object()` 构造器的是一个字符串,那么就相当于调用了 `new String()` 构造器。尽管这种做法不值得推荐,但仍然是可用的。

```
>>> var o = new Object('something');
```

```
>>> o.constructor
```

```
String()
```

```
>>> var o = new Object(123);
```

```
>>> o.constructor
```

```
Number()
```

语言中其他所有的对象,无论是内建的还是自定义的,都继承于 Object 对象(见表 C-1、表 C-2)。因此几乎所有的类型都可以调用该对象的方法与属性。

Object 构造器的成员

表 C-1

属性/方法	相关说明
Object.prototype	<p>该属性是所有对象的原型(包括 Object 对象本身),语言中的其他对象正是通过在该属性上添加东西来实现它们之间的继承关系的。</p> <pre>>>> var s = new String('noodles'); >>> Object.prototype.custom = 1; 1 >>> s.custom 1</pre>

用 Object()构造器所建对象的成员

表 C-2

属性/方法	相关说明
Constructor	<p>该属性指向 Object() 本身。</p> <pre>>>> Object.prototype. constructor === Object true >>> var o = new Object(); >>> o.constructor === Object True</pre>
toString(radix)	<p>该方法返回的是一个用于描述目标对象的字符串。特别地,当目标是一个 Number 对象时,我们还可以传递一个用于进制数的参数 radix,该参数的默认值为 10。</p> <pre>>>> var o = {prop: 1}; >>> o.toString() "[object Object]" >>> var n = new Number(255); >>> n.toString() "255" >>> n.toString(16) "ff"</pre>

(续表)

属性/方法	相关说明
toLocaleString()	<p>该方法与 toString() 基本相同, 只不过它会根据目标当前所在的区域做一些匹配操作, 以应对类似 Date() 这类对象的实现需求, 提供一些符合特定区域的值, 例如各种不同的日期格式</p>
valueOf()	<p>该方法返回的是 this 值, 即对象本身。但在其他类型的对象中, 它有可能会返回一个不同的值, 例如 Number 对象返回的是它的基本数值, 而 Date 对象返回的是一个时间戳 (timestamp)。</p> <pre data-bbox="913 1108 1563 1968"> >>> var o = {}; >>> typeof o.valueOf() "object" >>> var n = new Number(101); >>> typeof n.valueOf() "number" >>> var d = new Date(); >>> typeof d.valueOf() "number" >>> d.valueOf() 1208158875493 </pre>
hasOwnProperty(prop)	<p>该方法仅在目标属性为对象自身属性时返回 true, 而当该属性是从原型链中继承而来或根本就不存在时返回 false。</p> <pre data-bbox="913 2186 1653 2542"> >>> var o = {prop: 1}; >>> o.hasOwnProperty('prop') true >>> o.hasOwnProperty('toString') false </pre>

(续表)

属性/方法	相关说明
isPrototypeOf(obj)	<p>如果目标对象是当前对象的原型，该方法就会返回 true，而且，当前对象所在原型链上的所有对象都能通过该测试，并不局限于它的直系关系。</p> <pre>>>> var s = new String(''); >>> Object.prototype.isPrototypeOf(s) true >>> String.prototype.isPrototypeOf(s) true >>> Array.prototype.isPrototypeOf(s) false</pre>
propertyIsEnumerable(prop)	<p>如果目标属性能在 for-in 循环中被显示出来，该方法就返回 true。</p> <pre>>>> var a = [1,2,3]; >>> a.propertyIsEnumerable('length') false >>> a.propertyIsEnumerable(0) true</pre>

Array

Array() 是一个用来创建数组对象的构造器（见表 C-3）：

```
>>> var a = new Array(1,2,3);
```

当然，我们同样也能使用数组标识法：

```
>>> var a = [1,2,3]; //recommended
```

需要注意的是，如果我们传递给 Array() 构造器的是一个数字，该数字就会被设定为数组的长度。构造器将会根据该长度来创建数组，并将每个元素位置以 undefined 值填充。

```
>>> var a = new Array(3);
>>> a.length
3
>>> a
[undefined, undefined, undefined]
```

这种情况有时候会导致一些意想不到的情况。例如，下面是一个用数组标识法创建的数组：

```
>>> var a = [3.14]
>>> a
[3.14]
```

然而，如果我们将该浮点数传递给 `Array()` 构造器的话，就会出错了：

```
>>> var a = new Array(3.14)
invalid array length
```

Array 对象的成员

表 C-3

属性/方法	相关说明
length	该属性返回的是数组中元素的个数。 <pre>>>> [1,2,3,4].length 4</pre>
concat(i1, i2, i3,...)	该方法主要用于合并数组。 <pre>>>> [1,2].concat([10,20], [300,400]) [1, 2, 10, 20, 300, 400]</pre>
join(separator)	该方法用于将数组中的元素连成一个字符串。我们可以通过参数来指定元素之间的分割字符串，其默认值是逗号。 <pre>>>> [1,2,3].join() "1,2,3" >>> [1,2,3].join(' ') "1 2 3" >>> [1,2,3].join(' is less than ') "1 is less than 2 is less than 3"</pre>

(续表)

属性/方法	相关说明
pop()	<p>该方法用于移除数组中的最后一个元素，并将其返回。</p> <pre>>>> var a = ['un', 'deux', 'trois']; >>> a.pop() "trois" >>> a ["une", "deux"]</pre>
push(i1, i2, i3,...)	<p>该方法用于新元素添加到数组的末尾，并返回修改后的数组长度。</p> <pre>>>> var a = []; >>> a.push('zig', 'zag', 'zebra', 'zoo'); 4</pre>
reverse()	<p>该方法用于反转数组中的元素顺序，并返回修改后的数组。</p> <pre>>>> var a = [1,2,3]; >>> a.reverse() [3, 2, 1] >>> a [3, 2, 1]</pre>
shift()	<p>该方法与 pop() 基本相同，只不过这里移除的是首元素，而不是最后一个元素。</p> <pre>>>> var a = [1,2,3]; >>> a.shift(); 1 >>> a [2, 3]</pre>
slice(start_index, end_index)	<p>该方法用于截取数组的某一部分，但不会对原数组进行任何修改。</p> <pre>>>> var a = ['apple', 'banana', 'js', 'css', 'orange']; >>> a.slice(2,4) ["js", "css"] >>> a ["apple", "banana", "js", "css", "orange"]</pre>

(续表)

属性/方法	相关说明
<p>sort(callback)</p>	<p>该方法主要用于数组的排序，它有一个可选参数，是一个回调函数，我们可以用它来自定义排序规则。该函数以两个数组元素为参数，两个参数相等时返回 0，第一个参数大时返回 1，第二个参数大时返回-1。下面我们来演示一个按数字顺序排序的自定义函数（默认是按照字符顺序的）：</p> <pre>function customSort(a, b){ if (a > b) return 1; if (a < b) return -1; return 0; }</pre> <p>然后，我们将其应用于 sort() 方法：</p> <pre>>>> var a = [101, 99, 1, 5]; >>> a.sort(); [1, 101, 5, 99] >>> a.sort(customSort); [1, 5, 99, 101] >>> [7, 6, 5, 9].sort(customSort); [5, 6, 7, 9]</pre>
<p>splice(start, delete_count, i1, i2, i3,...)</p>	<p>该方法可能是数组函数当中最强大的一个了，它可在删除元素的同时添加新的元素。第一个参数所表示的是说要删除元素的始起位置，第二个参数代表的是要删除元素的个数，其余参数则都是一些将要插入在此处的新元素。</p> <pre>>>> var a = ['apple', 'banana', 'js', 'css', 'orange']; >>> a.splice(2, 2, 'pear', 'pineapple'); ["js", "css"] >>> a ["apple", "banana", "pear", "pineapple", "orange"]</pre>

(续表)

属性/方法	相关说明
<code>unshift(i1, i2, i3, ...)</code>	<p>该方法的功能与 <code>push()</code> 方法类似，只不过元素将会被添加到数组的开始处，而不是末尾处。另外和 <code>shift()</code> 方法一样，它也会在添加完元素后返回修改后的数组长度。</p> <pre>>>> var a = [1,2,3]; >>> a.unshift('one', 'two'); 5 >>> a ["one", "two", 1, 2, 3]</pre>

Function

在 JavaScript 中，函数也是一种对象，可以通过 `Function()` 构造器来定义，例如：

```
>>> var sum = new Function('a', 'b', 'return a + b;');
```

这与下面的函数标识法是完全等效的：

```
>>> var sum = function(a, b){return a + b;};
```

当然，我们还有更常见的做法：

```
>>> function sum(a, b){return a + b;}
```

但我们更鼓励您使用函数标识法，而不是 `Function()` 构造器。

Function 对象的成员

表 C-4

属性/方法	相关说明
<code>apply(this_obj, params_array)</code>	<p>该方法主要用于在当前对象的 <code>this</code> 值上调用其他函数。<code>apply()</code> 的第一个参数所引用的是将要绑定到 <code>this</code> 值上的函数对象。而第二个参数是一个数组，用于存储调用该函数对象时所需的参数。</p> <pre>function whatIsIt(){</pre>

(续表)

属性/方法	相关说明
<code>apply(this_obj, params_array)</code>	<pre>return this.toString(); } >>> var myObj = {}; >>> whatIsIt.apply(myObj); "[object Object]" >>> whatIsIt.apply(window); "[object Window]"</pre>
<code>call(this_obj, p1, p2, p3, ...)</code>	该方法与 <code>apply()</code> 基本相同，只不过其调用函数所需的参数是一个一个传递的，而不再是一个数组。
<code>length</code>	<p>该属性返回的是函数所拥有的参数个数。</p> <pre>>>> alert.length 1 >>> parseInt.length 2</pre>

Boolean

`Boolean()` 构造器所创建的是一个布尔类型的对象（这并不等同于基本布尔类型）。由于这种布尔对象的实际作用很有限，因此这里将它列出来，完全只是出于知识的完整性考虑。

```
>>> var b = new Boolean();
>>> b.valueOf()
false
>>> b.toString()
"false"
```

需要注意的是，布尔对象与基本布尔值并不相同。正如我们所了解的，所有对象本质上都属于 `truthy` 值。

```
>>> b === false
false
```

```
>>> typeof b
"object"
```

另外，除了从 Object 中继承来的内容外，Boolean 对象中并没有任何其他属性。

Number

下面我们来创建一个数字对象：

```
>>> var n = new Number(101);
>>> typeof n
"object"
>>> n.valueOf();
101
```

需要注意的是，Number 对象并不等同于基本数字类型，但如果我们在某个基本数字类型值上调用了 Number 类方法，那么该基本数字类型就会被自动转换成 Number 对象，例如：

```
>>> var n = 123;
>>> typeof n;
"number"
>>> n.toString()
"123"
```

Number()构造器的成员

表 C-5

属性/方法	相关说明
Number.MAX_VALUE	<p>该属性返回是一个常量（不可变的），表示该对象所能取的最大值。</p> <pre>>>> Number.MAX_VALUE 1.7976931348623157e+308</pre> <pre>>>> Number.MAX_VALUE = 101; Number.MAX_VALUE is read-only</pre>
Number.MIN_VALUE	<p>该属性返回的是 JavaScript 中的最小值。</p> <pre>>>> Number.MIN_VALUE 5e-324</pre>

(续表)

属性/方法	相关说明
Number.NaN	<p>该属性返回的是一个表示“Not A Number”的值。</p> <pre>>>> Number.NaN</pre> <p>NaN</p> <p>NaN 与任何值都不相等，包括它自己。</p> <pre>>>> Number.NaN === NaN</pre> <p>false</p> <p>另外，使用 Number.NaN 要比直接写 NaN 可靠一些，因为后者有时候会导致一些错误的覆盖性操作。例如：</p> <pre>>>> NaN = 1; // don't do this!</pre> <p>1</p> <pre>>>> NaN</pre> <p>1</p> <pre>>>> Number.NaN</pre> <p>NaN</p>
Number.POSITIVE_INFINITY	<p>该属性表示的是数字中的无穷大。同理，这种形式也比全局的 infinity 常量值（实际上就是全局对象的属性）可靠，因为它是只读的</p>
Number.NEGATIVE_INFINITY	<p>该属性与上面相同，但返回的是负无穷</p>

Number 对象的成员

表 C-6

属性/方法	相关说明
toFixed(fractionDigits)	<p>该方法将返回一个字符串，以定点小数的形式来表示某一数字，并进行四舍五入。</p> <pre>>>> var n = new Number(Math.PI);</pre> <pre>>>> n.valueOf();</pre> <p>3.141592653589793</p> <pre>>>> n.toFixed(3)</pre> <p>"3.142"</p>

(续表)

属性/方法	相关说明
<code>toExponential(fractionDigits)</code>	<p>该方法将返回一个字符串，以指数形式来表示某一数字，并进行四舍五入。</p> <pre>>>> var n = new Number(56789); >>> n.toExponential(2) "5.68e+4"</pre>
<code>toPrecision(precision)</code>	<p>该方法将返回一个字符串，其表示的数字形式既可以是指数型的，也可以是定点小数型的。</p> <pre>>>> var n = new Number(56789); >>> n.toPrecision(2) "5.7e+4" >>> n.toPrecision(5) "56789" >>> n.toPrecision(4) "5.679e+4" >>> var n = new Number(Math.PI); >>> n.toPrecision(4) "3.142"</pre>

String

`String()` 是一个用创建字符串对象的构造器，如果我们在一个基本字符串值上调用属于该对象的方法，那么该字符串就会被自动转换为 `String` 对象。

下面，我们来创建一个字符串对象和一个基本字符串：

```
>>> var s_obj = new String('something');
>>> var s_prim = 'something';
>>> typeof s_obj
"object"
>>> typeof s_prim
"string"
```

当我们使用`===`比较该对象和基本类型时，它们是不相等的。

```
>>> s_obj === s_prim
false
>>> s_obj == s_prim
true
```

`length` is a property of string objects:

```
>>> s_obj.length
9
```

如果我们在一个不属于对象的基本字符串上访问 `length` 属性，该字符串就会自动被转换成相应的对象，并成功返回字符串长度，比如：

```
>>> "something".length
9
```

String()构造器的成员

表 C-7

属性/方法	相关说明
<code>String.fromCharCode (code1, code2, code3, ...)</code>	该方法会根据用户输入的字符编码来创建字符串，并将其返回。 <pre>>>> String.fromCharCode (115, 99, 114, 105, 112, 116);</pre> "script"

String 对象的成员

表 C-8

属性/方法	相关说明
<code>length</code>	该属性返回字符串中的字符数量。 <pre>>>> new String('four').length</pre> 4
<code>charAt (pos)</code>	该方法返回指定位置处的字符，位置从 0 开始计数。 <pre>>>> "script".charAt (0);</pre> "s"

(续表)

属性/方法	相关说明
charCodeAt (pos)	<p>该方法返回指定位置处字符的编码。</p> <pre>>>> "script".charCodeAt(0);</pre> <p>115</p>
concat(str1, str2,)	<p>该方法利用当前字符串将输入中的各个子串连接成一个新的字符串。</p> <pre>>>> "".concat('zig', '-', 'zag');</pre> <p>"zig-zag"</p>
indexOf (needle, start)	<p>该方法用于返回匹配串的起始位置。第二个参数是可选的，用于指定搜索的开始位置。如果方法没有找到匹配串，就会返回-1。</p> <pre>>>> "javascript".indexOf('scr')</pre> <p>4 <pre>>>> "javascript".indexOf('scr', 5)</pre> <p>-1</p> </p>
lastIndexOf (needle, start)	<p>该方法与 indexOf() 的功能基本相同，只不过它的搜索是从后面开始的，例如我们要搜索字符串中的最后一个“a”：</p> <pre>>>> "javascript".lastIndexOf('a')</pre> <p>3</p>
localeCompare (needle)	<p>该方法会将两个字符串放在当前区域内进行比对，如果两个字符串完全相同就返回 0，如果 needle 中的字符序列靠前就返回 1，否则就返回-1。</p> <pre>>>> "script".localeCompare('crypt')</pre> <p>1 <pre>>>> "script".localeCompare('sscript')</pre> <p>-1 <pre>>>> "script".localeCompare('script')</pre> <p>0</p> </p></p>

(续表)

属性/方法	相关说明
<code>match(regex)</code>	<p>该方法会根据其接受的正则表达式对象返回一个容纳所有匹配串的数组。</p> <pre>>>> "R2-D2 and C-3PO".match(/[0-9]/g) ["2", "2", "3"]</pre>
<code>replace(needle, replacement)</code>	<p>该方法用于替换字符串对象中所有匹配相关正则表达式的内容。另外，我们还可以通过 <code>replacement</code> 参数设置一系列不同的匹配模式，例如 <code>\$1</code>、<code>\$2</code>、…… <code>\$9</code>。</p> <pre>>>> "R2-D2".replace(/2/g, '-two') "R-two-D-two" >>> "R2-D2".replace(/(2)/g, '\$1\$1') "R22-D22"</pre>
<code>search(regex)</code>	<p>该方法会返回第一个匹配正则表达式的子串的位置。</p> <pre>>>> "C-3PO".search(/[0-9]/) 2</pre>
<code>slice(start, end)</code>	<p>该方法会返回字符串中 <code>start</code> 到 <code>end</code> 之间的部分。如果 <code>start</code> 的值是个负数，那么开始位置实际上等于 <code>length+start</code>，同样的，如果 <code>end</code> 的值是负数，其结束位置等于 <code>length+end</code></p> <pre>>>> "R2-D2 and C-3PO".slice(4,13) "2 and C-3" >>> "R2-D2 and C-3PO".slice(4,-1) "2 and C-3P"</pre>
<code>split(separator, limit)</code>	<p>该方法可以将字符串转换成一个数组。它的第二个参数 <code>limit</code> 是可选的，而 <code>separator</code> 参数也可以是一个正则表达式。</p> <pre>>>> "1,2,3,4".split(',') ["1", "2", "3", "4"] >>> "1,2,3,4".split(',', 2) ["1", "2"]</pre>

(续表)

属性/方法	相关说明
substring(start, end)	<p>该方法的功能与 slice() 基本相同。如果 start 或 end 为负值或无效值时，它们会被视为 0。如果它们的值大于 length，则都会被视为 length。如果 end 大于 start，则它们会自动交换彼此的值。</p> <pre>>>> "R2-D2 and C-3PO".substring(4, 13) "2 and C-3" >>> "R2-D2 and C-3PO".substring(13, 4) "2 and C-3"</pre>
toLowerCase() toLocaleLowerCase()	<p>这两种方法都可将字符串内容转换为小写。</p> <pre>>>> "JAVA".toLowerCase() "java"</pre>
toUpperCase() toLocaleUpperCase()	<p>这两种方法都可将字符串内容转换为大写。</p> <pre>>>> "script".toUpperCase() "SCRIPT"</pre>

Date

Date()构造器可以有以下几种不同的输入类型：

- ◆ 我们可以分别将年、月、日、小时、分钟以及毫秒的值传递给构造器，例如：

```
>>> new Date(2011, 0, 1, 13, 30, 35, 500)
Sat Jan 01 2011 13:30:35 GMT-0800 (Pacific Standard Time)
```

- ◆ 上面所列出的这些参数，都是可以跳过的，在这种情况下它们会被默认为 0。要注意的是，月份的值是从 0（一月）到 11（十二月）的，小时的值是从 0 到 23 的，分钟和秒数的值都是 0 到 59，毫秒数则是从 0 到 999。

- ◆ 我们可以传递给构造器一个时间戳：

```
>>> new Date(1293917435500)
Sat Jan 01 2011 13:30:35 GMT-0800 (Pacific Standard Time)
```

- ◆ 如果我们没有传递给构造器任何参数，它就会返回当前日期/时间：

```
>>> new Date()
Fri Apr 18 2008 01:13:00 GMT-0700 (Pacific Daylight Time)
```

- ◆ 如果我们传递的是一个字符串，那么它会自动分析并提取该字符串中的有效日期信息：

```
>>> new Date('May 4, 2008')
Sun May 04 2008 00:00:00 GMT-0700 (Pacific Daylight Time)
```

Date()构造器的成员

表 C-9

属性/方法	相关说明
Date.parse(string)	该方法的作用与直接将字符串传递给 new Date() 类似，它会分析参数字符串中的日期信息并返回相应的时间戳，如果失败则返回 NaN。 >>> Date.parse('May 4, 2008') 1209884400000 >>> Date.parse('4th') NaN
Date.UTC(year, month, date, hours, minutes, seconds, ms)	该方法返回的也是一个时间戳，只不过这回是 UTC 时间（即 Coordinated Universal Time），不是本地时间。 >>> Date.UTC (2011, 0, 1, 13, 30, 35, 500) 1293888635500

Date 对象的成员

表 C-10

属性/方法	相关说明及示例
toUTCString()	该方法与 toString() 基本相同，但返回的是 UTC 时间，下面我们来看看太平洋时间（PST）、本地时间与 UTC 之间究竟有哪些不同： >>> var d = new Date(2010, 0, 1); >>> d.toString() "Fri Jan 01 2010 00:00:00 GMT-0800 (Pacific Standard Time)" >>> d.toUTCString() "Fri, 01 Jan 2010 08:00:00 GMT"

(续表)

属性/方法	相关说明及示例
toDateString()	<p>该方法只返回 toString() 中的日期部分:</p> <pre>>>> new Date(2010, 0, 1).toDateString();</pre> <p>"Fri Jan 01 2010"</p>
toTimeString()	<p>该方法只返回 toString() 中的时钟部分:</p> <pre>>>> new Date(2010, 0, 1).toTimeString();</pre> <p>"00:00:00 GMT-0800 (Pacific Standard Time)"</p>
toLocaleString() toLocaleDateString() toLocaleTimeString()	<p>这三个方法基本上分别与 toString()、toDateString() 以及 toTimeString() 等效。但格式更为友好, 能使用当前用户所设置的方式来显示信息。</p> <pre>>>> new Date(2010, 0, 1).toString();</pre> <p>"Fri Jan 01 2010 00:00:00 GMT-0800 (Pacific Standard Time)"</p> <pre>>>> new Date(2010, 0, 1).toLocaleString();</pre> <p>"Friday, January 01, 2010 12:00:00 AM"</p>
getTime() setTime(time)	<p>这组方法用于获取或设置某一 Date 对象中的时间 (以时间戳的形式)。在下面的示例中, 我们演示了如何创建一个 Date 对象, 并将它的日期前移一天:</p> <pre>>>> var d = new Date(2010, 0, 1);</pre> <pre>>>> d.getTime();</pre> <p>1262332800000 <pre>>>> d.setTime(d.getTime() + 1000 * 60 * 60 * 24);</pre> <p>1262419200000</p> <pre>>>> d.toLocaleString()</pre> <p>"Saturday, January 02, 2010 12:00:00 AM"</p> </p>
getFullYear() getUTCFullYear() setFullYear(year, month, date) setUTCFullYear(year, month, date)	<p>这组方法用于获取或设置 Date 对象中的全年份信息 (包括本地时间和 UTC 时间)。在这里不能用 getYear(), 因为它并不适用于公元两千年之后的年时, 所以还是使用 getFullYear() 方法为好</p>

(续表)

属性/方法	相关说明及示例
<pre>getFullYear() getUTCFullYear() setFullYear(year, month, date) setUTCFullYear(year, month, date)</pre>	<pre>>>> var d = new Date(2010, 0, 1); >>> d.getFullYear() 110 >>> d.getFullYear() 2010 >>> d.setFullYear(2011) 1293868800000 >>> d Sat Jan 01 2011 00:00:00 GMT-0800 (Pacific Standard Time)</pre>
<pre>getMonth() getUTCMonth() setMonth(month, date) setUTCMonth(month, date)</pre>	<p>这组方法用于获取或设置 Date 对象中的月份信息，它是从 0（一月）开始计数的：</p> <pre>>>> var d = new Date(2010, 0, 1); >>> d.getMonth() 0 >>> d.setMonth(11) 1291190400000 >>> d.toLocaleDateString() "Wednesday, December 01, 2010"</pre>
<pre>getDate() getUTCDate() setDate(date) setUTCDate(date)</pre>	<p>这组方法用于获取或设置 Date 对象中的日期信息。</p> <pre>>>> var d = new Date(2010, 0, 1); >>> d.toLocaleDateString() "Friday, January 01, 2010"</pre>

(续表)

属性/方法	相关说明及示例
<pre>getDate() getUTCDate() setDate(date) setUTCDate(date)</pre>	<pre>>>> d.getDate(); 1 >>> d.setDate(31); 1264924800000 >>> d.toLocaleDateString() "Sunday, January 31, 2010"</pre>
<pre>getHours() getUTCHours() setHours(hour, min, sec, ms) setUTCHours(hour, min, sec, ms) getMinutes() getUTCMinutes() setMinutes(min, sec, ms) setUTCMinutes(min, sec, ms) getSeconds() getUTCSeconds() setSeconds(sec, ms) setUTCSeconds(sec, ms) getMilliseconds() getUTCMilliseconds() setMilliseconds(ms) setUTCMilliseconds(ms)</pre>	<p>这组方法分别用于获取或设置 Date 对象中的小时、分钟、秒数及毫秒数信息。它们都是从 0 开始计数的。</p> <pre>>>> var d = new Date(2010, 0, 1); >>> d.getHours() + ':' + d.getMinutes() "0:0" >>> d.setMinutes(59) 1262336399000 >>> d.getHours() + ':' + d.getMinutes() "0:59"</pre>
<pre>getTimezoneOffset()</pre>	<p>该方法用于返回本地时间与 UTC 时间之间的差距，以分钟为单位。例如下面实现的是 PST（即 Pacific Standard Time）与 UTC 之间的差距：</p>

(续表)

属性/方法	相关说明及示例
getTimezoneOffset()	<pre>>>> new Date().getTimezoneOffset() 420 >>> 420/60 7</pre>
getDay() getUTCDay()	<p>这组方法返回的是当前时间的星期数，从 0（星期日）开始：</p> <pre>>>> var d = new Date(2010, 0, 1); >>> d.toLocaleDateString() "Friday, January 01, 2010" >>> d.getDay() 5 >>> var d = new Date(2010, 0, 3); >>> d.toLocaleDateString() "Sunday, January 03, 2010" >>> d.getDay() 0</pre>

Math

Math 对象的情况与其他内建对象稍许有些不同，因为它不能被用做构造器来创建对象。实际上，它只不过是一组相关函数和常量的集合而已。下面我们通过一些具体的实例来看看究竟有哪些不同：

```
>>> typeof String.prototype
"object"
>>> typeof Date.prototype
"object"
>>> typeof Math.prototype
"undefined"
>>> typeof Math
"object"
>>> typeof String
"function"
```

Math 对象的成员

表 C-11

属性/方法	相关说明
Math.E Math.LN10 Math.LN2 Math.LOG2E Math.LOG10E Math.PI Math.SQRT1_2 Math.SQRT2	<p>这里列出的都是一些常用的数学常量，都是只读的。下面是它们各自的值：</p> <pre>>>> Math.E 2.718281828459045 >>> Math.LN10 2.302585092994046 >>> Math.LN2 0.6931471805599453 >>> Math.LOG2E 1.4426950408889634 >>> Math.LOG10E 0.4342944819032518 >>> Math.PI 3.141592653589793 >>> Math.SQRT1_2 0.7071067811865476 >>> Math.SQRT2 1.4142135623730951</pre>
Math.acos(x) Math.asin(x) Math.atan(x) Math.atan2(y, x) Math.cos(x) Math.sin(x) Math.tan(x)	<p>这是对象中的三角函数集合</p>
Math.round(x) Math.floor(x) Math.ceil(x)	<p>round() 方法用于返回最接近本值的整数，而 ceil() 用于向上取整，floor() 则用于向下取整。</p> <pre>>>> Math.round(5.5)</pre>

(续表)

属性/方法	相关说明
Math.round(x)	6
Math.floor(x)	>>> Math.floor(5.5)
Math.ceil(x)	5 >>> Math.ceil(5.1)
	6
Math.max(num1, num2, num3, ...)	max() 和 min() 这两个方法分别用于返回其参数中的最大值和最小值。但如果参数列表中有一个值为 NaN, 那么两个方法都返回 NaN。 >>> Math.max(2, 101, 4.5)
Math.min(num1, num2, num3, ...)	101 >>> Math.min(2, 101, 4.5)
	2
Math.abs(x)	该方法用于返回参数的绝对值。 >>> Math.abs(-101) 101 >>> Math.abs(101) 101
Math.exp(x)	函数: Math.E 的 x 次方
Math.log(x)	取 x 的自然对数
Math.sqrt(x)	取 x 的平方根。 >>> Math.sqrt(9) 3 >>> Math.sqrt(2) === Math.SQRT2 True
Math.pow(x, y)	取 x 的 y 次方。 >>> Math.pow(3, 2) 9
Math.random()	该方法用于返回 0 到 1 之间的随机数 (包括 0)。 >>> Math.random() 0.8279076443185321

RegExp

`RegExp()` 是一个用于创建正则表达式对象的构造器，其第一个参数是正则表达式的匹配模式，第二个参数则是该匹配模式的修饰符。

```
>>> var re = new RegExp('[dn]o+dle', 'gmi');
```

该对象的模式可以匹配“noodle”、“doodle”、“dooodle”等。当然，我们也可以用正则表达式标识法来创建同样的对象：

```
>>> var re = (/[dn]o+dle/gmi); // recommended
```

关于正则表达式的更详细信息，读者可以参考第 4 章和附录 D 中的相关内容。

RegExp 对象的成员

表 C-12

属性/方法	相关说明
<code>global</code>	只读属性，当且仅当 <code>regexp</code> 对象被设置了 <code>g</code> 修饰符时为 <code>true</code>
<code>ignoreCase</code>	只读属性，当且仅当 <code>regexp</code> 对象被设置了 <code>i</code> 修饰符时为 <code>true</code>
<code>multiline</code>	只读属性，当且仅当 <code>regexp</code> 对象被设置了 <code>m</code> 修饰符时为 <code>true</code>
<code>lastIndex</code>	<p>该方法用于返回字符串中下一个匹配串的位置。当然，该方法也只有在 <code>test()</code> 和 <code>exec()</code> 成功匹配之后，且当 <code>g</code> (<code>global</code>) 修饰符被设定时才有效。</p> <pre>>>> var re = /[dn]o+dle/g; >>> re.lastIndex 0 >>> re.exec("noodle doodle"); ["noodle"] >>> re.lastIndex 6 >>> re.exec("noodle doodle"); ["doodle"] >>> re.lastIndex</pre>

(续表)

属性/方法	相关说明
lastIndex	<pre> 13 >>> re.exec("noodle doodle"); null >>> re.lastIndex 0 </pre>
source	<p>只读属性，返回的是该正则表达式的模式（不包含修饰符）。</p> <pre> >>> var re = /[nd]o+dle/gmi; >>> re.source "[nd]o+dle" </pre>
exec(string)	<p>该方法会对其输入字符串进行正则匹配，一旦匹配成功，就以数组的形式返回所有的匹配串或匹配分组。并且，当对象被设置有 g 修饰符时，该方法会自动确定第一个匹配串，并对 lastIndex 属性进行相关的设置。但如果匹配不成功，该方法就返回 null。</p> <pre> >>> var re = /([dn])(o+)dle/g; >>> re.exec("noodle doodle"); ["noodle", "n", "oo"] >>> re.exec("noodle doodle"); ["doodle", "d", "oo"] </pre>
test(string)	<p>该方法的功能与 exec() 相同，只不过它只返回 true 或 false。</p> <pre> >>> /noo/.test('Noodle') false >>> /noo/i.test('Noodle') true </pre>

Error 对象

通常情况下，Error 对象是由程序的运行环境（如浏览器）或其代码本身来负责创建的。

```
>>> var e = new Error('jaavcsritp is _not_ how you spell it');
>>> typeof e
"object"
```

除了 `Error()` 构造器本身，`Error` 对象还要另外留出派生对象，它们分别是：

- ◆ `EvalError`
- ◆ `RangeError`
- ◆ `ReferenceError`
- ◆ `SyntaxError`
- ◆ `TypeError`
- ◆ `URIError`

Error 对象的成员

表 C-13

属性名	相关说明
Name	<p>该属性返回的是创建当前错误对象的构造器的名称。</p> <pre>>>> var e = new EvalError('Oops'); >>> e.name "EvalError"</pre>
message	<p>该属性返回的是当前错误对象中的具体信息：</p> <pre>>>> var e = new Error('Oops... again'); >>> e.message "Oops... again"</pre>

附录 D

正则表达式

当我们使用（第 4 章中所讨论的）正则表达式时，可以对文本字符串进行如下匹配：

```
>>> "some text".match(/me/)
["me"]
```

但问题真正的关键是正则表达式的匹配模式，而不是这些字符串文本。在表 D-1 中，我们详细列出了各种不同模式的语法，并提供了相关的示例，以供读者参考。

表 D-1

匹配模式	相关说明
[abc]	这里匹配的是字符类信息。 <pre>>>> "some text".match(/[otx]/g) ["o", "t", "x", "t"]</pre>
[a-z]	这里匹配的是某一区间内的字符类信息。例如，[a-d]就相当于[abcd]，[a-z]就表示我们要匹配的是所有的小写字母，而[a-zA-Z0-9_]则表示匹配所有字母、数字及下划线。 <pre>>>> "Some Text".match(/[a-z]/g) ["o", "m", "e", "e", "x", "t"] >>> "Some Text".match(/[a-zA-Z]/g) ["S", "o", "m", "e", "T", "e", "x", "t"]</pre>

(续表)

匹配模式	相关说明
[^abc]	<p>这里匹配的是所有不属于表达式限定范围内的字符。</p> <pre>>>> "Some Text".match(/[^a-z]/g) ["S", " ", "T"]</pre>
a b	<p>这里匹配的是 a 或者 b。中间那个竖杠是“或者”的意思，该符号可以在同一表达式中多次使用。</p> <pre>>>> "Some Text".match(/t T/g); ["T", "t"] >>> "Some Text".match(/t T Some/g); ["Some", "T", "t"]</pre>
a(=?b)	<p>这里匹配的是所有 a 后面跟着 b 的信息。</p> <pre>>>> "Some Text".match(/Some(=?Tex)/g); null >>> "Some Text".match(/Some(=? Tex)/g); ["Some"]</pre>
a(?!b)	<p>这里匹配的是所有 a 后面不跟着 b 的信息。</p> <pre>>>> "Some Text".match(/Some(?! Tex)/g); null >>> "Some Text".match(/Some(?!Tex)/g); ["Some"]</pre>
\	<p>反斜杠主要用于帮助我们匹配一些模式文本中的特殊字符。</p> <pre>>>> "R2-D2".match(/[2-3]/g) ["2", "2"] >>> "R2-D2".match(/[2\-3]/g) ["2", "-", "2"]</pre>
\n \r \f \t \v	换行符 回车符 换页符 横向制表符 纵向制表符

(续表)

匹配模式	相关说明
\s	<p>这里匹配的是空白符，包括上面五个转义字符。</p> <pre>>>> "R2\n D2".match(/\s/g) ["\n", " "]</pre>
\S	<p>这里正好与上面相反，匹配的是除空白符以外的所有内容，就相当于<code>[^\s]</code></p> <pre>>>> "R2\n D2".match(/\S/g) ["R", "2", "D", "2"]</pre>
\w	<p>这里匹配的是所有的字母、数字或下划线，就相当于<code>[A-Za-z0-9_]</code>。</p> <pre>>>> "Some text!".match(/\w/g) ["S", "o", "m", "e", "t", "e", "x", "t"]</pre>
\W	<p>这里匹配的正好与<code>\w</code>相反。</p> <pre>>>> "Some text!".match(/\W/g) [" ", "!"]</pre>
\d	<p>这里匹配的是所有的数字类信息，就相当于<code>[0-9]</code>。</p> <pre>>>> "R2-D2 and C-3PO".match(/\d/g) ["2", "2", "3"]</pre>
\D	<p>这里正好与<code>\d</code>相反，匹配的是非数字类信息，就相当于<code>[^0-9]</code>或<code>[^\d]</code>。</p> <pre>>>> "R2-D2 and C-3PO".match(/\D/g) ["R", "-", "D", " ", "a", "n", "d", " ", "C", "-", "P", "O"]</pre>
\b	<p>这里匹配的是一个单词的边界，例如空格或标点符号。</p> <p>下面匹配的是后面跟着 2 的 R 或 D:</p> <pre>>>> "R2D2 and C-3PO".match(/[RD]2/g) ["R2", "D2"]</pre> <p>如果在上面的模式中加入该匹配符，匹配的就只有单词末尾的那一个了:</p> <pre>>>> "R2D2 and C-3PO".match(/[RD]2\b/g) ["D2"]</pre> <p>同样的，如果我们在其中输入一个破折号，也可以被当做一个单词的末尾。</p> <pre>>>> "R2-D2 and C-3PO".match(/[RD]2\b/g) ["R2", "D2"]</pre>

(续表)

匹配模式	相关说明
\B	<p>这里的匹配操作与\b正好相反。</p> <pre>>>> "R2-D2 and C-3PO".match(/[RD]2\b/g) null >>> "R2D2 and C-3PO".match(/[RD]2\B/g) ["R2"]</pre>
[\b]	<p>这里匹配的是退格键符 (backspace)</p>
\0	<p>这里匹配的是 null 值</p>
\u0000	<p>这里匹配的是一个 Unicode 字符，并且是以一个四位的十六进制数来表示的。</p> <pre>>>> "СТОЯН".match(/\u0441\u0442\u043E/) ["сто"]</pre>
\x00	<p>这里匹配的是一个字符，该字符的编码是以一个两位的十六进制数来表示的。</p> <pre>>>> "dude".match(/\x64/g) ["d", "d"]</pre>
^	<p>这里匹配的是字符串开头部分。另外，如果我们对该模式设置了 m 修饰符 (多行)，那么它匹配的是每一行的开头。</p> <pre>>>> "regular\nregular\nexpression".match(/r/g); ["r", "r", "r", "r", "r"] >>> "regular\nregular\nexpression".match(/^r/g); ["r"] >>> "regular\nregular\nexpression".match(/^r/mg); ["r", "r"]</pre>
\$	<p>这里匹配的是输入消息的末尾部分。另外，如果我们对该模式设置了多行修饰符，那么它匹配的是每一行的末尾。</p> <pre>>>> "regular\nregular\nexpression".match(/r\$/g); null >>> "regular\nregular\nexpression".match(/r\$/mg); ["r", "r"]</pre>

(续表)

匹配模式	相关说明
	<p>这里匹配的是除了换行符或换行符以外的任何字符。</p> <pre>>>> "regular".match(/r./g); ["re"] >>> "regular".match(/r.../g); ["regu"]</pre>
*	<p>这里匹配的是模式中间出现 0 次或多次的内容。例如，<code>/.*/</code> 可以匹配任何内容（包括空串）。</p> <pre>>>> "".match(/.*/) [""] >>> "anything".match(/.*/) ["anything"] >>> "anything".match(/n.*h/) ["nyth"]</pre>
?	<p>这里匹配的是模式中间出现 0 次或 1 次的内容。</p> <pre>>>> "anything".match(/ny?/g) ["ny", "n"]</pre>
+	<p>这里匹配的是模式中间出现至少 1 次（或多次）的内容。</p> <pre>>>> "anything".match(/ny+/g) ["ny"] >>> "R2-D2 and C-3PO".match(/[a-z]/gi) ["R", "D", "a", "n", "d", "C", "P", "O"] >>> "R2-D2 and C-3PO".match(/[a-z]+/gi) ["R", "D", "and", "C", "PO"]</pre>
{n}	<p>这里匹配的是模式中间出现过 n 次的内容。</p> <pre>>>> "regular expression".match(/s/g) ["s", "s"] >>> "regular expression".match(/s{2}/g) ["ss"] >>> "regular expression".match(/\b\w{3}/g) ["reg", "exp"]</pre>

(续表)

匹配模式	相关说明
{min,max}	<p>这里匹配的是在模式中出现次数在 max 到 min 之间的信息。如果我们省略了 max, 就意味着没有最多次数, 只有最少次数。但 min 是不能省略的。</p> <p>例如, 如果我们在输入 “doodle” 这个词时输入了 10 个 “o” :</p> <pre>>>> "dooooooooooodle".match(/o/g) ["o", "o", "o", "o", "o", "o", "o", "o", "o", "o"] >>> "dooooooooooodle".match(/o{2}/g) ["oo", "oo", "oo", "oo", "oo"] >>>"dooooooooooodle".match(/o{2,}/g) ["oooooooo"] >>>"dooooooooooodle".match(/o{2,6}/g) ["oooooo", "oooo"]</pre>
(pattern)	<p>当某个匹配模式被放在括号内时, 就表明匹配该模式的匹配串是可替换的, 因此它也被称为捕获模式。</p> <p>这些被捕获的匹配串可以分别用 \$1、\$2…\$9 等参数来表示。</p> <p>例如, 我们可以将匹配串中所有的 “r” 都重复一次:</p> <pre>>>> "regular expression".replace(/(r)/g, '\$1\$1') "rregularr expression"</pre> <p>或我们将所有匹配 “re” 的内容都替换成 “er” :</p> <pre>>>> "regular expression".replace (/ (r) (e) /g, '\$2\$1') "ergular experssion"</pre>
(?:pattern)	<p>这不是捕获模式, 也就是说这里不能用 \$1、\$2 等参数来记录匹配串。</p> <p>例如在下面的示例中, 当我们对 “re” 进行匹配时, \$1 记住的不是 “r”, 而是第二个模式所匹配的结果:</p> <pre>>>> "regular expression".replace (/(?:r) (e) /g, '\$1\$1') "eegular expession"</pre>

有时候, 模式中的某些特殊字符所代表的意义往往不止一种, 例如 ^、?、\b 等, 因此在我们使用时有必要对此稍加留意。